Programování v UNIXu

(NSWI015)

verze: 1. ledna 2017

SISAL MFF UK, Malostranské nám. 25, 118 00 Praha 1

- Toto jsou oficiální materiály pro výuku předmětu *Programování v UNIXu* (kód předmětu NSWI015) na Matematicko-fyzikální fakultě Univerzity Karlovy.
- Tyto materiály jsou uveřejněny pod licencí Creative Commons BY-NC-SA 3.0. Soubory k ukázkovým příkladům uvedené výše jsou Public Domain, pokud v nich není uvedeno jinak.
- Tento předmět je 2/1, cvičení bude jednou za dva týdny.
- Všechny informace, které budete potřebovat, a materiály k přednášce jsou na http://mff.devnull.cz/, včetně aktuální verze těchto poznámkových slajdů (http://mff.devnull.cz/pvu/slides/). **Dříve, než začnete materiály používat, přesvědčte se, že není novější verze!**
- Ukázkové příklady jsou na http://mff.devnull.cz/pvu/src/, pokud se budu v poznámkách odkazovat na konkrétní příklad, link zde uvedený slouží jako kořenový adresář.
- Je potřeba se zapsat na cvičení v ISu.
- Zápočet je za zápočtový program.
- Zkouška má písemnou a ústní část.
- Zkoušet se bude to, co bude odpředneseno (kromě témat na vyplnění případně zbylého času). Většina informací je ve slajdech, ale řada důležitých podrobností může chybět.

- Předpoklady:
  - Uživatelská znalost UNIXu, programování v shellu na úrovni přednášky „Úvod do UNIXu" (NSWI095)
  - Znalost jazyka C
  - Znalost základních pojmů teorie operačních systémů
- Tento text je průběžně doplňován, ChangeLog začíná na straně 242. Upozornění na chyby prosím posílejte na kontakty, které najdete na výše uvedené stránce.

Obsah

- úvod, vývoj UNIXu a C, programátorské nástroje
- základní pojmy a konvence UNIXu a jeho API
- přístupová práva, periferní zařízení, systém souborů
- manipulace s procesy, spouštění programů
- signály
- synchronizace a komunikace procesů
- síťová komunikace
- vlákna, synchronizace vláken
- ??? - bude definováno později, podle toho kolik zbyde času

- Budeme se zabývat hlavně principy UNIXu a programováním pro UNIX pouze v jazyce C.
- **Přednáška je převážně o systémových voláních, tj. rozhraním mezi uživatelským prostorem a jádrem**.
- Při popisu API se budeme držet normy *Single UNIX Specification, version 3* (SUSv3). Systémy, které tuto specifikaci podporují, mohou používat označení UNIX 03. V současné době (10/2011) jsou to poslední verze systémů Solaris, AIX, HP-UX a Mac OS X na různých architekturách (http://www.opengroup.org/openbrand/register/xy.htm).
- Pro konkrétní příklady budu používat většinou systémy FreeBSD a Solaris.

Obsah

- **úvod, vývoj UNIXu a C, programátorské nástroje**
- základní pojmy a konvence UNIXu a jeho API
- přístupová práva, periferní zařízení, systém souborů
- manipulace s procesy, spouštění programů
- signály
- synchronizace a komunikace procesů
- síťová komunikace
- vlákna, synchronizace vláken
- ??? - bude definováno později, podle toho kolik zbyde času

Literatura v češtině

1. Skočovský, L.: **Principy a problémy operačního systému UNIX**. Science, 1993
2. Skočovský, Luděk: **UNIX, POSIX, Plan9**. L. Skočovský, Brno, 1998
3. Jelen, Milan: **UNIX V - programování v systému**. Grada, Praha 1993
4. **Linux - Dokumentační projekt**. Computer Press, 1998; http://www.cpress.cz/knihy/linux
5. Herout, Pavel: **Učebnice jazyka C**. 2 díly. Kopp, České Budějovice, 2004 (4., respektive 2. přepracované vydání)

Ohledně unixu doporučuji spíše literaturu v anglickém jazyce.

1. Všestranný úvod do UNIXu, ale dost stručná; Skočovský je autorem více českých knih o unixu, ale dnes jsou již více nebo méně zastaralé.
2. Pokročilejší pohled, ale předpokládá předběžné znalosti, místy těžko stravitelná.
3. Programování v C pro UNIX System V, práce se soubory a s procesy, System V IPC, nepopisuje např. vlákna a sítě.
4. O Linuxu bylo samozřejmě v češtině vydáno mnoho dalších knih.
5. Vynikající knihy o jazyce C.

Literatura - design a principy systému

1. Uresh Vahalia: **UNIX Internals: The New Frontiers**. Prentice Hall; 1st edition, 1995
2. Bach, Maurice J.: **The Design of the UNIX Operating System**. Prentice Hall, 1986
3. McKusick, M. K., Neville-Neil, G. V.: **The Design and Implementation of the FreeBSD Operating System**. Addison-Wesley, 2004
4. McDougall, R.; Mauro, J.: **Solaris Internals**. Prentice Hall; 2nd edition, 2006.
5. **Linux Documentation Project**. http://tldp.org/

Tyto knihy se zabývají stavbou unixu, použitými algoritmy, strukturami apod., nejsou to kniho o programování pod tímto systémem.

1. Skvělá kniha, zabývá se obecnými myšlenkami UNIXu a porovnává systémy SVR4.2, 4.4BSD, Solarix 2.x a Mach. 12/2005 mělo vyjít druhé, doplněné vydání. Po několika letech čekání přestalo druhé vydání figurovat v listingu na amazon.com, a tedy se budeme muset zřejmě smířit s tím, že nevyjde.
2. Klasická kniha o UNIXu, popis struktury a funkcí jádra UNIX System V Rel. 2, částečně i 3; přestože je to kniha z dnešního pohledu již zastaralá, lze ji pořád jednoznačně doporučit, protože to je jedna z nejlepších knih, co byla kdy o UNIXu napsána. V roce 1993 vyšel český překlad, **Principy operačního systému UNIX**, SAS.

3. Popis struktury a funkcí jádra FreeBSD 5.2; tato kniha navazuje na klasickou knihu **The Design and Implementation of the 4.4 BSD Operating System** od stejného autora (resp. jeden ze čtyř, uvedený jako první).

4. Nejlepší kniha o operačním systému Solaris. Obsahuje podrobné informace o tom, jak tento systém funguje včetně nejnovějších věcí z verze 10 jako jsou zóny, Crypto Framework, DTrace, Least Privilege model a další.

5. Domovská strana Linux dokumentačního projektu.

Literatura - programování

1. Stevens, W. R., Rago, S. A.: **Advanced Programming in UNIX(r) Environment**. Addison-Wesley, 2nd edition, 2005.
2. Rochkind, M. J.: **Advanced UNIX Programming**, Addison-Wesley; 2nd edition, 2004
3. Stevens, W. R., Fenner B., Rudoff, A. M.: **UNIX Network Programming, Vol. 1 – The Sockets Networking API**. Prentice Hall, 3rd edition, 2004
4. Butenhof, D. R.: **Programming with POSIX Threads**, Addison-Wesley; 1st edition, 1997
5. unixové specifikace, viz http://www.unix.org
6. manuálové stránky (zejm. sekce 2, 3)

1. Pravděpodobně není lepší knihy o programování pod unixem (neobsahuje síťové programování, to je v knize 3).

2. Aktualizované vydání další z klasických knih o programování pod unixem. Obsahuje i síťové programování a ač samozřejmě není tak podrobná jako spojení knih 1 a 3, může to být někdy naopak výhodou. Tuto knihu jednoznačně doporučuji, pokud chcete něco kupovat a chcete mít pro začátek jen jednu knihu. Pokud bych ji měl charakterizovat jednou větou, bylo by to, že autor vidí a dokáže ukázat souvislosti, což je velmi vzácná vlastnost.

3. Klasická kniha o síťovém programování, jedna z nejlepších k tomuto tématu; existuje i druhý díl **UNIX Network Programming, Volume 2: Interprocess Communications**, která se zabývá komunikací mezi procesy (roury, POSIX IPC, System V IPC, synchronizace vláken, RPC).

4. Velmi dobrá a podrobná kniha o programování s vlákny.

5. Domovská stránka posledních specifikací rozhraní UNIXu.

6. Podrobný popis jednotlivých funkcí (v Linuxu běžně ne zcela dostačující; manuálové stránky v tomto systému jsou často horší kvality než u systémů ostatních).

7. Kniha, která se nevešla na slajd a která překračuje obsah této přednášky: Gallmeister, B. R.: **POSIX.4 Programmers Guide: Programming for the Real World**, O'Reilly; 1st edition, 1995. Výborná kniha s krásnou obálkou o real-time rozšířeních POSIXu. Viz také strany 155 a 159.

... a spousta dalších knih, online dokumentací a internetových zdrojů, poslední dobou vychází poměrně hodně knih o Linuxu, zaměřených na používání i programování.

... jděte na http://www.amazon.com/ a zadejte klíčové slovo "unix". Pokud byste z Amazonu něco kupovali, dejte pozor na to, že mnoho knih má aktualizovaná vydání i po několika málo letech, někdy i levnější než ta původní, která jsou však stále na skladu a v on-line nabídce; tak ať zbytečně nekoupíte starší vydání než to aktuální. Navíc se vyplatí zkontrolovat i u příslušného vydavatelství, že není v brzké době naplánováno vydání nové – tato informace někdy na Amazonu je, někdy ne.

... na Amazonu se může vyplatit nakoupit knihy z druhé ruky, protože jsou často výrazně levnější než knihy nové. Problém je, že většinou není možné je poslat přímo do ČR, ale musí vám je někdo přivézt.

Literatura - historie UNIXu

- Peter Salus: **A Quarter Century of UNIX**, Addison-Wesley; 1st edition (1994)
- Libes D., Ressler, S.: **Life With Unix: A Guide for Everyone**, Prentice Hall (1989)
- **Open Sources: Voices from the Open Source Revolution**, kapitola **Twenty Years of Berkeley Unix From AT&T-Owned to Freely Redistributable**; O'Reilly (1999); on-line na webu: http://oreilly.com/openbook/opensources/book/index.html

... mnoho materiálů na webu; často však obsahující ne zcela přesné informace

- Kapitola o BSD Unixu z *Open Sources* napsaná Marshallem Kirk McKusickem je opravdu výborná.

(Pre)historie UNIXu

- 1925 – **Bell Telephone Laboratories** – výzkum v komunikacích (např. 1947: transistor) v rámci AT&T
- 1965 – BTL s General Electric a MIT vývoj OS **Multics** (MULTIplexed Information and Computing System)
- 1969 – Bell Labs opouští projekt, **Ken Thompson** píše assembler, základní OS a systém souborů pro PDP-7
- 1970 – Multi-cs ⇒ Uni-cs ⇒ Uni-x
- 1971 – UNIX V1, a portován na PDP-11
- prosinec 1971 – první edice *UNIX Programmer's Manual*

- AT&T = American Telephone and Telegraph Company
- Multics byl systém, který významně ovlivnil další vývoj operačních systémů. Obsahoval mnoho v té době inovativních myšlenek, z nichž ale ne všechny byly přijímány kladně. Významně ovlivnil právě UNIX, který mnoho myšlenek převzal a jeho nedostatky se naopak snažil napravit. Hlavní rozdíl byl asi ten, že UNIX byl navržen jako mnohem jednodušší systém, než Multics.
- po odchodu BTL z projektu Multics prodala GE svoji počítačovou divizi firmě Honeywell včetně projektu Multics, který se pak pod její patronací dále aktivně vyvíjel (virtuální paměť, multiprocesory, ...), až do roku 1985. Poslední instalace Multics-u fungovala na kanadském Ministerstvu obrany (Canadian Department of National Defence) a systém byl například ještě aktivně používán pro vojenské operace během války v Perském zálivu. Definitivní shutdown byl proveden 31. října 2000. Více informací je možné nalézt na http://www.multicians.org.
- před počátkem práce na vývojovém prostředí pro PDP-7 napsal Thompson program *Space Travel*, který byl vyvinut na jiném prostředí (Honeywell 635) a na pásce přenesen na PDP-7.
- celkem bylo 10 edicí tohoto manuálu, korespondující deseti verzím UNIXu vzniklých v BTL.
- UNIX V1 neměl volání `pipe` !!!

- manuál pro verzi 1: http://man.cat-v.org/unix-1st/ Stojí za to nahlédnout, jak jeho struktura ovlivnila vzhled dnešních manuálových stránek.
- **za povšimnutí stojí, že UNIX je zhruba o 10 let starší než DOS**
- systém Multics měl 9 hlavních cílů, jak popsáno v článku *Introduction and Overview of the Multics System* z roku 1965. Za nejzajímavější cíl bych považoval požadavek na nepřerušovaný běh systému.
- Multics byl napsaný v jazyce PL/I (Programming Language #1), tedy dříve než byl UNIX přepsaný do C !
- Multicsu byl v roce 1980 udělen jako prvnímu systému level B2. Po několik let to byl jediný systém s tímto bezpečnostím levelem.
- GE byla založena v roce 1892 sloučením dvou společností, z nichž jedna byla Edison General Electric Company založená roku 1879 Thomasem Alvou Edisonem (vynálezce žárovky, filmové kamery, ...); v současné době její dceřinné společnosti pokrývají mnoho oblastí, včetně dodávky jednoho ze dvou typů motorů pro Airbus 380 nebo bankovnictví.
- PDP = Programmed Data Processor. První typ, *PDP-1*, se prodávala za $120.000 v době, kdy se jiné počítače prodávaly za ceny přes milión. To byla také strategie fy DEC - pojem *computer* tehdy znamenal drahou věc, potřebující sál a tým lidí, který se o to všechno bude starat. Proto DEC své mašiny nenazýval počítači, ale pravě slovem *PDPs*.
- PDP-11 je legendární mašina od firmy DEC, postupně vznikaly verze PDP-1 az PDP-16, kromě PDP-2, PDP-13. Existují PDP-11 systémy, které ještě dnes běží, a také firmy, které pro ně vyrábějí náhradní díly.

Historie UNIXu, pokračování

- únor 1973 – UNIX V3 obsahoval *cc* překladač (jazyk C byl vytvořen **Dennisem Ritchiem** pro potřeby UNIXu)
- říjen 1973 – UNIX byl představen veřejnosti článkem *The UNIX Timesharing System* na konferenci ACM
- listopad 1973 – **UNIX V4 přepsán do jazyka C**
- 1975 – UNIX V6 byl první verzí UNIXu běžně k dostání mimo BTL
- 1979 – UNIX V7, pro mnohé "the last true UNIX", obsahoval *uucp*, Bourne shell; velikost kernelu byla pouze 40KB !!!
- 1979 – UNIX V7 portován na 32-bitový VAX-11
- 1980 – Microsoft příchází s XENIXem, který je založený na UNIXu V7

- ACM = Association for Computing Machinery, založena 1947. UNIX představili Ken Thompson a Dennis Ritchie.
- **akt přepsání UNIXu do jazyka C byl možná nejvýznamnějším momentem v historii tohoto systému** ⇒ UNIX mohl být mnohem jednodušeji portován na jiné architektury
- na verzi 6 je založena legendární kniha *A commentary on the Unix Operating System*, jejíž autorem je John Lions.
- Microsoft neprodával XENIX přímo, ale licencoval ho OEM výrobcům (Original Equipment Manufacturer) jako byl Intel, SCO a jiní. Jiné firmy pak XENIX dále portovaly na 286 (Intel) a 386 (SCO, 1987). Na webu je možné najít zajímavé informace popisující tuto dobu a tehdy kladný vztah Microsoftu k UNIXu.
- UNIX V7 měl cca 188 tisíc řádek zdrojového kódu v cca 1100 souborech (zjištěno pomocí `find`, `wc` a `awk` přes soubory se jménem `*.[cshy]`).
- pokud vás více zajímá historie unixu, podívejte se na Wikipedii na heslo "unix" a skrz odkazy máte na dlouho co číst.
- V roce 1973 byl UNIX víceuživatelský systém (konta s hesly) s podporou pro multiprocessing s ochranou procesů a stránkováním. Měl signály, roury, hierarchický systém souborů s mount pointy, souborová práva (User/group/other r/w/x), hard linky, zařízení přístupná jako soubory. Kernel byl napsán v jazyku C a v paměti zabíral jeho obraz 26 Kilobytů. Pro práci se soubory sloužily systémová volání `open()`, `close()`, `read()`, `write()`, `seek()`,

`pipe()`. K manipulaci s procesy volání `fork()`, `exec()`, `wait()`, `exit()`. Celkem bylo 48 syscallů, z nich existuje 35 do dnešní doby.

Divergence UNIXu

- pol. 70. let – uvolňování UNIXu na univerzity: především **University of California v Berkeley**
- 1979 – z UNIX/32V (zmíněný port na VAX) poskytnutého do Berkeley se vyvíjí **BSD Unix (Berkeley Software Distribution)** verze 3.0; poslední verze 4.4 v roce 1993
- 1982 **AT&T**, vlastník BTL, může vstoupit na trh počítačů (zakázáno od roku 1956) a přichází s verzí *System III* (1982) až *V.4* (1988) – tzv. *SVR4*
- vznikají UNIX International, OSF (Open Software Foundation), X/OPEN, ...
- 1991 – Linus Torvalds zahájil vývoj OS Linux, verze jádra 1.0 byla dokončena v r. 1994

- UNIX je univerzální operační systém fungující na široké škále počítačů od embedded a handheld systémů (Linux), přes osobní počítače až po velké servery a superpočítače.
- UNIX V3 = *UNIX verze 3*, UNIX V.4 = *system 5 release 4* atd., tj. UNIX V3 != SVR3.
- UNIX System III tedy není UNIX V3; v této době (pozdní 70. léta) bylo v BTL několik skupin, které přispívaly do vývoje UNIXu. Vx verze byly vyvíjeny v rámci *Computer Research Group*, další skupiny byly *Unix System Group* (USG), *Programmer's WorkBench* (PWB). Další větví UNIXu byl Columbus UNIX též v rámci BT. Na těchto různých verzích je právě založena verze System III. Zájemce o více informací odkazuji na web.
- UNIX se rozštěpil na dvě hlavní větve: AT&T a BSD, jednotliví výrobci přicházeli s vlastními modifikacemi. **Jednotlivé klony od sebe navzájem přebíraly vlastnosti.**
- System V R4 měl cca 1.5 milionu řádek zdrojového kódu v cca 5700 souborech (zjištěno pomocí `find`, `wc` a `awk` přes soubory se jménem `*.[cshy]`).
- Univerzita v Berkeley získala jako jedna z prvních licenci UNIXu v roce 1974. Během několika let studenti (jedním z nich byl Bill Joy, pozdější zakladatel firmy Sun Microsystems a autor C-shellu) vytvořili SW balík *Berkeley Software Distribution* (BSD) a prodávali ho v roce 1978 za $50. Tyto počáteční

verze BSD obsahovaly pouze SW a utility (první verze: Pascal překladač, editor *ex*), ne systém ani žádné jeho změny. To přišlo až s verzí 3BSD. verze 4BSD vzniká roku 1980 již jako projekt financovaný agenturou DARPA a vedený Billem Joyem. Trpí problémy spojenými s nedostatečným výkonem a vzniká tak vyladěný systém 4.1BSD dodávaný od roku 1981.

- 4.1BSD mělo být původně 5BSD, ale poté, co AT&T vzneslo námitky, že by si zákazníci mohli plést 5BSD se systémem System V, přešlo BSD na číslování 4.xBSD. Běžnou věcí bylo, že než psát vlastní kód, vývojáři z Berkeley se raději nejdříve podívali kolem, co je již hotové. Tak BSD například převzalo virtuální paměť z Machu a nebo NFS-kompatibilní kód vyvinutý na jedné kanadské univerzitě.

- výrobci hardware dodávali varianty UNIXu pro své počítače a komercializace tak ještě zhoršila situaci co týče diverzifikace totoho systému

- v 80-tých letech se proto začaly objevovat snahy o standardizaci. Standard říká, jak má vypadat systém navenek (pro uživatele, programátora a správce), nezabývá se implementací. Cílem je přenositelnost aplikací i uživatelů. Všechny systémy totiž z dálky vypadaly jako UNIX, ale při bližším prozkoumání se lišily v mnoha důležitých vlastnostech. System V a BSD se např. lišily v použitém filesystému, síťové architektuře i v architektuře virtuální paměti.

- když v roce 1987 firmy AT&T a Sun (jehož tehdejší SunOS byl založený na BSD) spojily svoje úsilí na vyvinutí jednoho systému, který by obsahoval to nejlepší z obou větví, kromě nadšených reakcí to vzbudilo i strach u mnoha dalších výrobců unixových systémů, kteří se báli, že by to pro obě firmy znamenalo obrovskou komerční výhodu. Vzniká proto Open Software Foundation (nezaměňovat za FSF), a zakládajícími členy byly mimo jiné firmy Hewlett-Packard, IBM a Digital. Z toho vzešlý systém OSF/1 ale nebyl příliš úspěšný, a dodával ho pouze Digital, který ho přejmenoval na Digital UNIX. Zajímavostí je, že systém je postavený na mikrojádru Mach. Po akvizici Digitalu Compaqem byl systém přejmenován na Tru64 a s tímto jménem je dále podporován firmou Hewlett-Packard, která se v roce 2002 s Compaqem spojila. Mezitím firmy AT&T a Sun kontrovaly založením UNIX International. Toto období přelomu 80-tých a 90-tých let se nazývá **Unix Wars** – boj o to, co bude "standardním unixem".

- OSF a UI se staly velkými rivaly, ale velmi rychle se střetly s nečekaným protivníkem - s firmou Microsoft.

- (1992) 386BSD založené na *Networking Release 2*; Bill Jolitz vytvořil 6 chybějících souborů a dal tak dohromady funkční BSD systém pro i386. Tento systém se stal základem systémů *NetBSD* a *FreeBSD* (a dalších, z těchto dvou systémů vycházejících).

- (1995) 4.4BSD-Lite Release 2, po které následuje rozpuštění CSRG, která skoro 20 let pilotovala vývoj BSD větve. Více již zmíněná kapitola o BSD Unixu.

Současné UNIXy

Hlavní komerční unixové systémy:

- Sun Microsystems: **SunOS** (není již dále vyvíjen), **Solaris**
- Apple: **Mac OS X**
- SGI: **IRIX**
- IBM: **AIX**
- HP: **HP-UX**, **Tru64 UNIX** (Compaq)
- SCO: **SCO Unix**
- Novell: **UNIXware**

Open source:

- **FreeBSD**, **NetBSD**, **OpenBSD**
- **Linux** distribuce

- Striktně technicky vzato se jako UNIX může označovat pouze systém, který prošel certifikací Single Unix Specification. Z výše uvedeného seznamu by to byly 4 operační systémy, které byly registrovány jako UNIX 03 na různých architekturách (http://www.opengroup.org/openbrand/register/). Ostatní systémy, které nebyly certifikovány, se označují jako Unix-like, ačkoliv v mnoha případech splňují většinů požadavků standardu. Běžně se nicméně používá označení Unix pro obě dvě skupiny.
- když jsem cca v roce 1998 projel nmapem všechny DNS root servery, abych zjistil na kterých systémech běží, bylo 80% z nich na SunOS/Solaris. IRIX je zase systém, který po mnoho let ovládal televizní/filmový průmysl (např. Pixar studio kde na IRIXu vznikly filmy jako *A Bug's Life*, *Toy Story* a další). A na AIX například běžel *Deep Blue*, paralelní superpočítač, který v roce 1997 porazil v šesti fascinujících zápasech 3.5 ku 2.5 úřadujícího velmistra šachu Garriho Kasparova. Jinými slovy – každý systém má svoje úspěchy.
- jak již bylo zmíněno, Tru64 UNIX vychází z OSF/1 firmy DEC. Ta předtím dodávala Ultrix, založený na BSD unixu.
- OpenSolaris byl projekt vzniklý v červnu 2005 a byl založen na podmnožině zdrojových textů vývojové verze Solarisu (kernel, knihovny, příkazy). Distribuce vzešlé z komunity jsou například LiveCD *BeleniX*, *SchilliX* a *Nexenta*. Přestože projekt OpenSolaris již dále nepokračuje, existuje open source fork *Illumos* a na něm založené distribuce *SmartOS* a *OpenIndiana*.
- pozor na to, že Linux je pouze jádro, ne systém, na rozdíl třeba od FreeBSD. Illumos je něco mezi tím, jádro + drivery, základní příkazy a knihovny,

a v binární podobě to má něco přes 100MB. V obou případech je tedy správné používat spojení "Linux (Illumos) distribuce", když se bavíme o celém systému.

- každá komerční varianta vycházela z jednoho ze dvou hlavních systémů – UNIX V nebo BSD, a přidávala si své vlastnosti. Díky mnoha verzím UNIXu tak vzniká velký počet různých standardů (strana 14). Nakonec se většina výrobců shodla na několika základních.

- graf znázorňující historii unixových systémů a závislosti mezi nimi na 19-ti A4 listech ve formátu PS/PDF je k nalezení na http://www.levenez.com/unix/

Standardy UNIXu

- **SVID** (System V Interface Definition)
  - „fialová kniha", kterou AT&T vydala poprvé v roce 1985
  - dnes ve verzi SVID3 (odpovídá SVR4)
- **POSIX** (Portable Operating System based on UNIX)
  - série standardů organizace IEEE značená P1003.xx, postupně je přejímá vrcholový nadnárodní orgán ISO
- **XPG** (X/Open Portability Guide)
  - doporučení konsorcia X/Open, které bylo založeno v r. 1984 předními výrobci platforem typu UNIX
- **Single UNIX Specification**
  - standard organizace The Open Group, vzniklé v roce 1996 sloučením X/Open a OSF
  - dnes Version 3 (**SUSv3**), předchozí Version 2 (**SUSv2**)
  - splnění je nutnou podmínkou pro užití obchodního názvu UNIX

- základní informace je, že oblast standardů týkající se unixových systémů je věc značně složitá a na první pohled velmi nepřehledná.
- AT&T dovolila výrobcům nazývat svoji komerční UNIX variantu "System V" pouze pokud splňovala podmínky standardu SVID. AT&T také publikovala *System V Verification Suite* (SVVS), které ověřilo, zda daný systém odpovídá standardu.
- POSIX (Portable Operating System Interface) je standardizační snaha organizace IEEE (Institute of Electrical and Electronics Engineers).
- SUSv3 je společný standard The Open Group, IEEE (Std. 1003.1, 2003 Edition) a ISO (ISO/IEC 9945-2003).
- Pro certifikaci operačního systému na Single Unix Specification je nutné aby systém (na dané architektuře, např. 64-bit x86) prošel sadou testů. Výsledky testů jsou pak vyhodnoceny. Testy samotné jsou sdruženy do tzv. *test suites*, což jsou sady automatických testů, které projdou systém a zjistí jestli splňuje rozhraní dané normou. Pro SUSv3 je takových test suites cca 10.
- Rozhraní specifikované normou POSIX.1-2008 se dělí na 4 základní skupiny: XSH (System Interfaces), XCU (Shell and Utilities), XBD (Base definitions). Z nich je co do počtu rozhraní nejobsáhlejší XSH, která popisuje více než 1000 rozhraní.
- Skupiny rozhraní POSIXu spolu se skupinou Xcurses, která je součástí Single Unix Specification (ale nikoliv součást POSIX báze v normě IEEE Std 1003.1-2001) zahrnují celkem 1742 rozhraní, které tvoří Single Unix Specification

(2003). Tabulky rozhraní SUS je možné získat zde: http://www.unix.org/version3/inttables.pdf

- komerční UNIXy většinou sledují Single UNIX Specification, splnění této normy je právě podmínkou pro užití názvu UNIX (značka UNIX 98 odpovídá SUSv2, značka UNIX 03 odpovídá SUSv3). Je postavena na bázi POSIXu. My se budeme držet SUSv3. Popis datových struktur a algoritmů jádra v tomto materiálu bude většinou vycházet ze System V Rel. 4.
- na Solarisu je obsáhlá manuálová stránka `standards(5)`, kde můžete na jednom místě nalézt mnoho informací týkající se standardů. Jednotlivé příkazy splňující danou normu jsou navíc umístěny do vyhrazených adresářů. Např. program `tr` je v adresářích `/usr/xpg4/bin/` a `/usr/xpg6/bin/`, v každém je verze příkazu splňující danou normu. Na přepínače a chování dané normou se pak lze spolehnout např. při psaní shellových skriptů.
- opět na Solarisu, podívejte se na hlavičkový soubor `/usr/include/sys/feature_tests.h`

POSIX

- tvrzení "tento systém je POSIX kompatibilní" nedává žádnou konkrétní informaci
  - asi podporuje POSIX1990 a možná i něco dalšího (co?)
- dotyčný buď neví co je POSIX nebo si myslí, že to nevíte vy
- jediná rozumná reakce je otázka "jaký POSIX?"
- POSIX je **rodina standardů**
- prvním dokumentem je *IEEE Std POSIX1003.1-1988*, později po vzniku dalších rozšíření neformálně odkazovaný jako POSIX.1
- poslední verze POSIX.1 je *IEEE Std 1003.1, 2004 Edition*
  - obsahuje v sobě již i to, co dříve definoval POSIX.2 (Shell and Utilities) a různá, dříve samostatná rozšíření

- prvním dokumentem je *IEEE Std POSIX1003.1-1988*, dříve označovaný prostě jako POSIX, pak odkazovaný jako *POSIX.1*, protože POSIXem se nyní míní sada vzájemně souvisejících standardů. POSIX.1 v té době obsahoval programovací API, tj. práce s procesy, signály, soubory, časovači atd. S malými změnami byl převzat organizací ISO (*ISO 9945-1:1990*), a je označovaný i jako POSIX1990. IEEE označení je *IEEE Std POSIX1003.1-1990*. Tento standard byl sám o sobě velký úspěch, ale stále ještě nespojoval tábory System V a BSD, protože v sobě například nezahrnoval BSD sockety nebo

IPC (semafory, zprávy, sdílená paměť) ze System V. Součástí standardu je i "POSIX conformance test suite (PCTS)", který je volně k dispozici.

- označení POSIX vymyslel Richard Stallman, tedy člověk, který v roce 1983 založil GNU projekt.
- důležitá rožšíření k IEEE Std 1003.1-1990 (jsou součástí IEEE Std 1003.1, 2004 Edition):
  - *IEEE Std 1003.1b-1993 Realtime Extension*, neformálně známý jako POSIX.4, protože to bylo jeho původní označení před přečíslováním; já budu toto rozšíření někdy také nazývat POSIX.4. Většina tohoto rozšíření je nepovinná, takže tvrzení "systém podporuje POSIX.1b" má ještě horší vypovídací hodnotu než "systém je POSIX kompatibilní", a to prakticky nulovou. Jediná povinná část POSIX.4 je malé doplnění k signálům oproti POSIX1990. Je proto nutné vždy uvést, co z POSIX.4 je implementováno – např. sdílená paměť, semafory, real-time signály, zamykání paměti, asynchronní I/O, časovače atd.
  - *IEEE Std 1003.1c-1995 Threads*, viz strana 213.
  - *IEEE Std 1003.1d-1999 Additional Realtime Extensions*
  - *IEEE Std 1003.1j-2000 Advanced Realtime Extensions*, viz strana 233.
  - ...
- standardy POSIX je možné nalézt na http://www.open-std.org/. HTML verze je volně k prohlížení, za PDF verzi se platí.

Jazyk C

- téměř celý UNIX je napsaný v C, pouze nejnižší strojově závislá část v assembleru ⇒ poměrně snadná přenositelnost
- navrhl Dennis Ritchie z Bell Laboratories v roce 1972.
- následník jazyka B od Kena Thomsona z Bell Laboratories.
- vytvořen jako prostředek pro přenos OS UNIX na jiné počítače – silná vazba na UNIX.
- varianty jazyka:
  - původní K&R C
  - standard ANSI/ISO C
- **úspěch jazyka C daleko přesáhl úspěch samotného UNIXu**

- CPL ⇒ BCPL ⇒ B (Thompson, interpret) ⇒ C
- K&R C – jazyk C tak, jak je popsaný v klasické knize Brian W. Kernighan, Dennis M. Ritchie: The C Programming Language (Prentice-Hall, 1978).
- v roce 1983 ANSI (American National Standards Institute) zformoval výbor pro vytvoření C standardu. Po dlouhém a pracném procesu byl v roce 1989 standard konečně hotov, a je známý nejčastěji jako "ANSI C", případně jako C89 (například překladač pro tuto normu se v Sun Studiu jmenuje `c89`, jelikož i to samotné jméno programu musí být podle normy). Druhé vydání K&R knihy (1988) je již upravené právě pro nadcházející ANSI C. V roce 1990 bylo ANSI C adoptováno organizací ISO jako ISO/IEC 9899:1990; tento C standard tak může být někdy označován i jako C90.
- se C standardem se pak nějakou dobu nehýbalo, až na konci 90-tých let prošel další revizí v rámci ISO a vzniká ISO 9899:1999, častěji označovaný jako C99. V roce 2000 pak naopak tento standard převzal ANSI.
- rozdíly mezi C89 a C99 jsou mimo jiné zahrnutí inline funkcí, definicí proměnných například i do `for` konstrukce, jednořádkových komentářů pomocí `//`, nových funkcí jako `snprintf` apod.
- specifikaci C standardu a mnoho dalších otevřených standardů je možné nalézt na http://www.open-std.org/.

Formáty dat

- pořadí bajtů – závisí na architektuře počítače
  - big endian: 0x11223344 =

	11	22	33	44
`addr +`	0	1	2	3

  - little endian: 0x11223344 =

	44	33	22	11
`addr +`	0	1	2	3

- řádky textových souborů končí v UNIXu znakem **LF** (nikoliv `CRLF`). Volání `putc('\n')` tedy píše pouze jeden znak.
- big endian – SPARC, MIPS, síťové pořadí bajtů
- little endian – Intel

- velký pozor na výstupy programů typu `hexdump`, které defaultně vypisují soubor v 16-ti bitových číslech, což svádí vidět soubor jinak, než jak je opravdu zapsaný na disku; viz příklad (i386, FreeBSD). První číslo v souboru je znak 'i', který ale reprezentuje nižších 8 bitů 16-ti bitového čísla, takže pokud vypíšeme první 2 bajty jako short integer, musí být reprezentace znaku 'i' (tj. číslo 69) až za 6a. Obdobně pro 'kl'.

```
$ echo -n ijkl > test
$ hexdump test
0000000 6a69 6c6b
0000004
```

  je samozřejmě možné použít jiný formát výstupu:

```
$ hexdump -C test
00000000 69 6a 6b 6c
00000004
```

- UNIX norma příkaz `hexdump` nemá, ale definuje `od` (octal dump), takže zde je jeho `hexdump`u ekvivalentní formát výpisu na SPARCu (Solaris); všimněte si změny oproti výpisu pod FreeBSD!

```
$ od -tx2 test
0000000 696a 6b6c
0000004
```

Deklarace a definice funkce

- K&R
  - deklarace

    ```
    návratový_typ indentifikátor();
    ```

  - definice

    ```
    návratový_typ indentifikátor(par [,par...])
    typ par;...
    { /* tělo funkce */ }
    ```

- ANSI
  - deklarace

    ```
    návratový_typ indentifikátor(typ par [,typ par...]);
    ```

  - definice

    ```
    návratový_typ indentifikátor(typ par [,typ par...])
    { /* tělo funkce */ }
    ```

- používejte pouze novější (ANSI) typ deklarací a vždy deklarujte prototypy funkcí, tj. inkludujte hlavičkové soubory. Výjimkou může asi jen to, pokud budete pracovat s kódem, který byl napsaný podle K&R.
- různými zápisy deklarací se dostáváme rovnou i k různým stylům psaní zdrojových textů. Některé systémy to příliš neřeší (Linux), jiné systémy mají velmi striktní pravidla pro psaní zdrojových textů (např. Solaris, viz on-line *C Style and Coding Standards for SunOS*: http://mff.devnull.cz/pvu/common/cstyle.ms.pdf). Snad každý UNIXový systém má program indent(1), který vám pomocí přepínačů přeformátuje jakýkoli C zdrojový text do požadovaného výstupu.

C style

- věc zdánlivě podřadná, přitom extrémně důležitá – úprava zdrojových kódů programu
- mnoho způsobů jak ano:

```
int
main(void)
{
        char c;
        int i = 0;

        printf("%d\n", i);
        return (0);
}
```

- u C stylu je nejdůležitější to, aby byl konzistentní. Pokud skupina programátorů pracuje na jednom projektu, není zas až tak důležité, na jakém stylu se dohodnou (pokud je alespoň trochu rozumný), ale aby se dohodli. Jednotný a dobře zvolený styl šetří čas a brání zbytečným chybám.

C style (cont.)

- mnoho způsobů jak **NE** (tzv. assembler styl):

```
int main(void) {
int i = 0; char c;
printf("%d\n", i);
return (0);
}
```

- nebo (schizofrenní styl):

```
int main(void) {
        int i = 0; char c;
        if (1)
        printf("%d\n", i);i=2;
return (0);
}
```

- pamatujte na to, že dobrý styl zdrojových kódů je i vizitkou programátora. Když se v rámci přijímacích pohovorů odevzdávají i ukázkové kódy, tak hlavní důvod překvapivě není ten, aby se zjistilo, že vám daný program funguje. Úprava samotného zdrojového textu je jedním z kriterií, protože to přeneseně může svědčit i o dalších skutečnostech – někdo např. bude odhadovat, že pokud píšete nečistý a neupravený kód, tak jste možná ještě nepracovali na něčem opravdu složitém či něčem zahrnujícím spolupráci s více programátory, protože v tom případě je rozumně čistý kód jednou z podmínek úspěchu a jednou ze zkušeností, které z takové spolupráce vycházejí. Toto je samozřejmě zčásti subjektivní názor, ale čistým kódem nic nezkazíte, minimálně nekazíte oči svým cvičícím.

- informace o C stylu používaném pro zdrojové kódy Solarisu, včetně skriptu, který vám zdrojáky zkontroluje a upozorní na případné nedostatky, je možné nalézt na http://mff.devnull.cz/pvu/common/cstyle.html. Pokud budete dodržovat tento styl při psaní zkouškových příkladů, ušetříte mi tím práci při vyhodnocování. Obecně je dobré, abyste si zkusili, co je to psát kód podle nějakého konkrétního stylu, a ten pro daný „projekt" dodržet.

- bohužel ani výše uvedený skript není všemocný. Je založený na kontrole pomocí regulárních výrazů, takže třeba ten assembler styl ze slajdu projde bez chyby, protože bez skutečné analýzy kódu není možné (?) kontrolovat správné odsazování. Vše ostatní je na něm, dle `cstyle` skriptu, korektní.

Utility

cc, **c99***, **gcc**†	překladač C
CC, **g++**†	překladač C++
ld	spojovací program (linker)
ldd	pro zjistění závislostí dynamického objektu
cxref*	křížové odkazy ve zdrojových textech v C
sccs*, **rcs**,**cvs**	správa verzí zdrojového kódu
make*	řízení překladu podle závislostí
ar*	správa knihoven objektových modulů
dbx, **gdb**†	debuggery
prof, **gprof**†	profilery

* SUSv3 † GNU

SUSv3

- standardní příkaz volání kompilátoru a linkeru C je `c99` (podle ISO normy pro C z roku 1999)
- `cb` (C program beautifier) není
- pro správu verzí je `sccs`
- debuggery a profilery nejsou

Konvence pro jména souborů

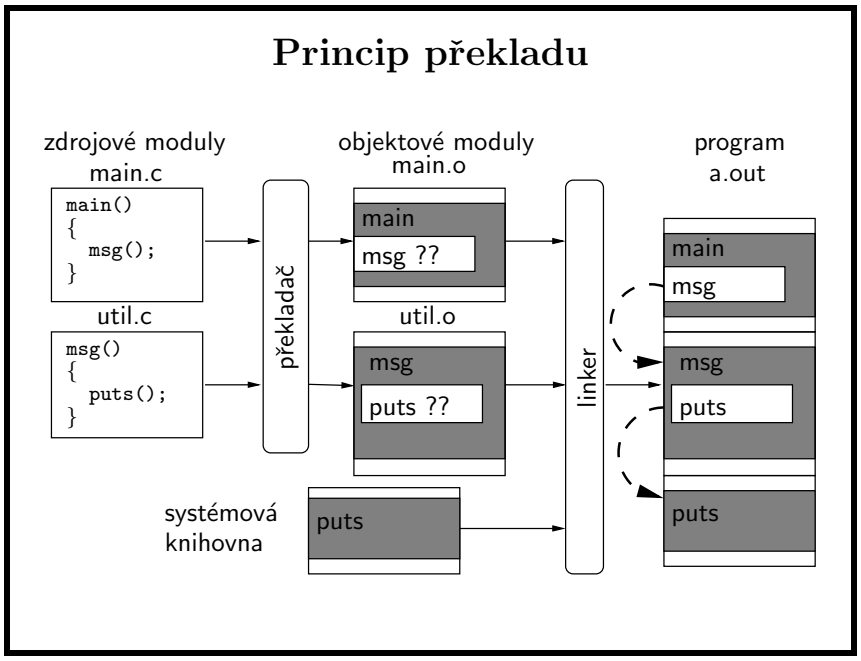
`*.c`	jména zdrojových souborů programů v C
`*.cc`	jména zdrojových souborů programů v C++
`*.h`	jména hlavičkových souborů (headerů)
`*.o`	přeložené moduly (object files)
`a.out`	jméno spustitelného souboru (výsledek úspěšné kompilace)

`/usr/include`	kořen stromu systémových headerů
`/usr/lib/lib*.a`	statické knihovny objektových modulů
`/usr/lib/lib*.so`	umístění dynamických sdílených knihoven objektových modulů

statické knihovny – při linkování se kód funkcí použitých z knihovny stane součástí výsledného spustitelného programu. Dnes se už moc nepoužívá.

sdílené knihovny – program obsahuje pouze odkaz na knihovnu, při spuštění programu se potřebné knihovny načtou do paměti ze souborů `*.so` a přilinkují.

- dnes se většinou používají sdílené knihovny, protože nezabírají tolik diskového prostoru (knihovna je na disku jednou, není součástí každého spustitelného souboru) a snadněji se upgradují (stačí instalovat novou verzi knihovny, není třeba přelinkovat programy). Poslední verze Solarisu už například vůbec neobsahuje `libc.a`, díky čemuž již programátor nemůže vytvořit statickou binárku, aniž by měl dostatečné znalosti systému.
- někdy se bez statických knihoven neobejdeme. V některých situacích není možné použít knihovny dynamické, spustitelné soubory jsou takzvané *standalone binaries* a použití naleznou například při bootování operačního systému.



- u složitějších programů bývá zvykem rozdělit zdrojový text programu do několika modulů, které obsahují příbuzné funkce a tyto moduly se pak mohou překládat zvlášť (dokonce každý modul může být v jiném jazyce a překládán jiným překladačem). Výhodou je jednak urychlení překladu (překládají se vždy jen moduly změněné od posledního překladu) a jednak flexibilita (některé moduly se mohou používat v různých programech). Pro řízení překladu se obvykle používá utilita `make`.

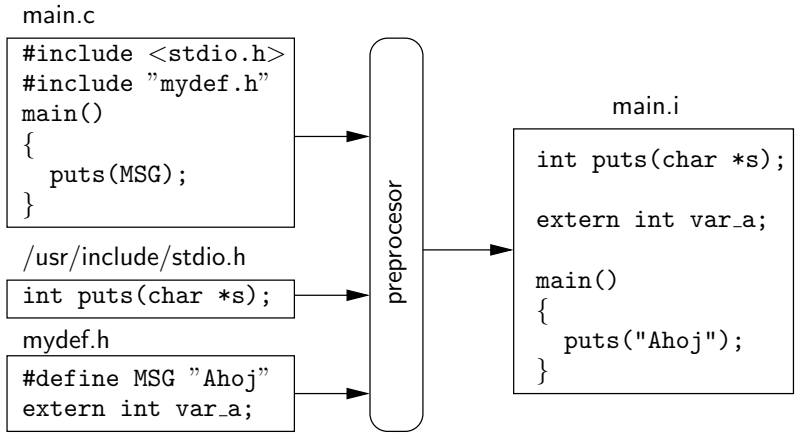
- *překladač* jednotlivé zdrojové moduly přeloží do tvaru tzv. *objektových modulů*, jež obsahují kód programu (včetně volání lokálních funkcí), ale namísto volání externích funkcí obsahují jen tabulku jejich jmen.

- po fázi překladu nastupuje *spojovací program* (též *linker* editor nebo *loader*), který zkompletuje výsledný program včetně vyřešení externích odkazů mezi moduly a systémovými knihovnami resp. mezi moduly navzájem.

- použité statické knihovny jsou zkopírovány do spustitelného souboru. Na sdílené knihovny jsou ve spustitelném souboru pouze odkazy a linkuje je runtime linker při každém spuštění programu. Více viz dynamický linker na straně 35.

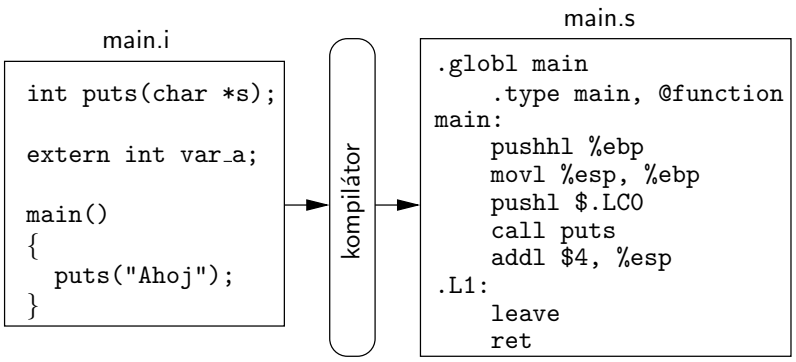
- pomocí parametrů linkeru lze určit, zda se budou používat statické nebo dynamické knihovny. Zdrojový kód je v obou případech stejný. Existuje i mechanismus (`dlopen`, `dlsym`...), pomocí kterého se za běhu programu vybere sdílená knihovna a dají se volat její funkce. Tímto způsobem můžete také zjistit, zda v systému je přítomna příslušná funkcionalita a pokud ne, zachovat se podle toho. Více na straně 144.

Překlad jednoho modulu (preprocesor)



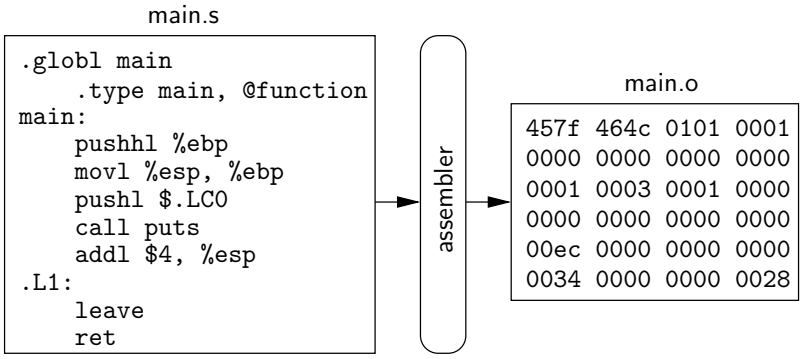
- preprocesor provádí expanzi maker, čtení vložených (include) souborů a vynechává komentáře.
- výstup preprocesoru lze získat pomocí `cc -E` případně přímo zavoláním `cpp`, nemusí to být ale vždy totéž protože některé překladače mají preprocesor integrován v sobě. Preprocesor můžete samozřejmě používat i pro jiné projekty, které s překladem zdrojových souborů v jazyce C nemusí mít vůbec nic společného.
- použití preprocesoru se může velmi hodit v situaci, kdy potřebujete zasáhnout do cizího kódu, plného podmínečných vkládání různých hlavičkových souborů a různých definic závislých na daných podmínkách. Při hledání původce chyby vám právě může hodně pomoci samostatného zpracování vstupního souboru pomocí preprocesuru, kde problém již většinou rozpoznáte snadno.
- `cpp` (nebo `cc -E` vám dokáže na standardní chybový výstup zobrazit i celý strom vkládaných souborů, což opět při podmínečných překladech může být velmi užitečná věc. Stačí pro to použít volbu `-H` a přesměrovat výstup do `/dev/null` čímž dostanete pouze hierarchii vkládaných hlavičkových souborů.

Překlad jednoho modulu (kompilátor)



- obrázek výše je příklad výstupu pro i386 platformu (32-bit, AT&T syntax).
- překlad z C do assembleru
- výstup této fáze překladu lze získat pomocí `cc -S`.

Překlad jednoho modulu (assembler)



- opět příklad výstupu pro i386 platformu (32-bit).
- překlad z assembleru do strojového kódu
- objektový modul je výsledkem příkazu `cc -c`.

Kompilátor

- volání:
  `cc [options] soubor ...`
- nejdůležitější přepínače:

`-o soubor`	jméno výsledného souboru
`-c`	pouze překlad (nelinkovat)
`-E`	pouze preprocesor (nepřekládat)
`-l`	slinkuj s příslušnou knihovnou
`-Ljméno`	přidej adresář pro hledání knihoven z `-l`
`-Olevel`	nastavení úrovně optimalizace
`-g`	překlad s ladicími informacemi
`-Djméno`	definuj makro pro preprocesor
`-Iadresář`	umístění `#include` souborů

- `-l`/`-L` jsou přepínače linker editoru, tj. kompilátor příslušné informace předá, ale jsou používány tak často, že jsou vloženy i do tohoto slajdu.
- kompilátor a linker mají mnoho dalších přepínačů ovlivňujících generovaný kód, vypisování varovných hlášení nebo variantu jazyka (K&R/ANSI). Je třeba nastudovat dokumentaci konkrétního produktu.

Předdefinovaná makra

`__FILE__`, `__LINE__`, `__DATE__`, `__TIME__`, `__cplusplus`, apod.	jsou standardní makra kompilátoru C/C++
`unix`	vždy definováno v Unixu
`mips`, `i386`, `sparc`	hardwarová architektura
`linux`, `sgi`, `sun`, `bsd`	klon operačního systému
`_POSIX_SOURCE`, `_XOPEN_SOURCE`	překlad podle příslušné normy

pro překlad podle určité normy by před prvním `#include` měl být řádek s definicí následujícího makra. Pak načtěte `unistd.h`.

UNIX 98	`#define _XOPEN_SOURCE 500`
SUSv3	`#define _XOPEN_SOURCE 600`
POSIX1990	`#define _POSIX_SOURCE`

- funguje to tak, že pomocí konkrétních maker definujete co chcete (např. `_POSIX_SOURCE`) a podle nastavení jiných maker (např. `_POSIX_VERSION`) pak zjistíte, co jste dostali. Musíte ale vždy po nastavení maker nainkludovat `unistd.h` a použít správný překladač. Například se pokusíme přeložit program `basic-utils/standards.c`, který vyžaduje SUSv3, na systému podporujícím SUSv3 (Solaris 10), ale překladačem, který podporuje pouze SUSv2 (SUSv3 překladač je `c99`). Pozor, že defaultní chování vašeho překladače může být klidně právě to z `c89`.

```
$ cat standards.c
#define _XOPEN_SOURCE   600
/* you must #include at least one header !!! */
#include <stdio.h>
int main(void)
{
        return (0);
}
$ c89 basic-utils/standards.c
"/usr/include/sys/feature_tests.h", line 336: #error: "Compiler or
options invalid; UNIX 03 and POSIX.1-2001 applications require
the use of c99"
cc: acomp failed for standards.c
```

- zdroj maker pro standard tedy může být již na straně 14 zmiňovaný hlavičkový soubor `feature_tests.h` na Solarisu.

- v dokumentaci konkrétního kompilátoru je možné najít, která další makra se používají. Množství maker je definováno také v systémových hlavičkových souborech.
- POSIX.1 v sobě zahrnuje ANSI C; tedy C89, ne C99 (o C standardech více na straně 16).
- co se týče maker k jednotlivým standardům, velmi dobrá je kapitola 1.5 v [Rochkind]. Také doporučuji C program `basic-tools/suvreq.c` k této kapitole, který je možné najít i na mých stránkách v sekci ukázkových příkladů.
- malý příklad na podmíněný překlad:

```
int
main(void)
{
#ifdef unix
        printf("yeah\n");
#else
        printf("grr\n");
#endif
        return (0);
}
```

- příklad na použití `__LINE__` viz `basic-tools/main__LINE__.c`

Link editor (linker)

- Volání:
  `ld [`*options*`] `*soubor* `...`
  `cc [`*options*`] `*soubor* `...`
- Nejdůležitější přepínače:

`-o` *soubor*	jméno výsledného souboru (default `a.out`)
`-l`*lib*	linkuj s knihovnou `lib`*lib*`.so` nebo `lib`*lib*`.a`
`-L`*path*	cesta pro knihovny (`-l`*lib*)
`-shared`	vytvořit sdílenou knihovnu
`-non_shared`	vytvořit statický program

- linker je program, který vezme typicky více objektů vygenerovaných překladačem a vytvoří z nich binární program, knihovnu nebo další objekt vhodný pro další fázi linkování.
- pozor na to, že na různých systémech se některé přepínače mohou lišit, například `ld` na Solarisu nezná přepínače `-shared` a `-non_shared`, je nutné použít jiné.
- existuje přepínač `-R`, který umožňuje specifikovat cestu pro hledání knihoven za běhu (runtime). Tato cesta se může lišit od cesty specifikované přepínačem `-L`.
- u malých programů (v jednom souboru) lze provést překlad a linkování jedním příkazem `cc`. U větších programů skládajících se z mnoha zdrojových souborů a knihoven se obvykle odděluje překlad a linkování a celý proces je řízen utilitou `make` (strana 30).

Řízení překladu a linkování (`make`)

- **zdrojové texty**

main.c

```
#include "util.h"
main()
{
  msg();
}
```

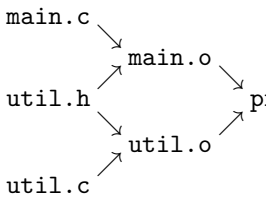
util.h

```
void msg();
```

util.c

```
#include "util.h"
msg()
{
  puts();
}
```

- **závislosti**



- **soubor `Makefile`**

```
prog : main.o util.o
        cc -o prog main.o util.o
main.o : main.c util.h
        cc -c main.c
util.o : util.c util.h
        cc -c util.c
```

.

- program je možné také přeložit a slinkovat jedním voláním kompilátoru, nebo definovat postup překladu a linkování pomocí shellového skriptu. Důvodem pro použití `make` je to, že vyhodnocuje závislosti mezi soubory a po změně některého zdrojového souboru překládá jenom to, co na něm závisí. Častý způsob překladu softwaru po aplikování změn způsobem "`make clean; make all`" je v situaci, kdy celý překlad trvá minuty (desítky minut, hodiny...), trochu nevhodný – právě proto je důležité mít dobře napsaný `Makefile`.
- řádek "`prog : main.o util.o`" definuje, že se má nejprve rekurzivně zajistit existence a aktuálnost souborů `main.o` a `util.o`. Pak se zkontroluje, zda

soubor (cíl) `prog` existuje a je aktuální (datum poslední modifikace souboru je mladší než `main.o` a `util.o`). Pokud ano, nedělá se nic. Když ne, provede se příkaz na následujícím řádku.

- `make` se spouští typicky s parametrem určující příšlušný cíl (*target*); při spuštění bez parametrů se vezme první target. To bývá `all`, což většinou podle unixové konvence přeloží vše, co se přeložit má. Následuje pak třeba spuštění `make` s parametrem `install` apod.
- `make` je samozřejmě univerzální nástroj, použitelný i jinde než u překladů zdrojových kódů
- příklad: `basic-utils/Makefile01`. Pozor na to, že pro nestandardní jméno vstupního souboru je nutné použít přepínač `-f`: "`make -f Makefile01`".

Syntaxe vstupního souboru (`make`)

- popis závislostí cíle: *targets* `:` [*files*]
- prováděné příkazy: <Tab>*command*
- komentář: `#`*comment*
- pokračovací řádek: *line-begin*`\`
  line-continuation

- **Pozor na to, že řádek s příkazem začíná tabulátorem, nikoliv mezerami.** Každý příkazový řádek se provádí samostatným shellem, pokud je potřeba provést více řádků pomocí jednoho shellu, musí se všechny až na poslední ukončit backslashem (shell je dostane jako jeden řádek). Viz příklad, ve kterém dva poslední `echo` příkazy jsou součástí jednoho `if` příkazu, který je spuštěn samostatným shellem. Dále si všimněte, že pokud máte příkaz na více řádků, musíte každý řádek ukončit zpětným lomítkem.

```
$ cat basic-utils/Makefile02
all:
        @echo $$$$
```

```
        @echo $$$$
        @if true; then \
                echo $$$$; \
                echo $$$$; \
        fi
$ make -f Makefile02
5513
5514
5515
5515
```

- co se týká použití zpětného lomítka, tak to funguje jako oddělovač slov, a `make` místo něj vloží mezeru. Příklad: `basic-utils/Makefile07`.
- zdvojením `$` se potlačí speciální význam dolaru (viz následující slajd)
- znak `@` na začátku řádku potlačí jeho výpis – `make` jinak standardně vypisuje nejdříve to, co bude vykonávat.
- vypsání všech přikazů které bude make provádět bez toho aby je skutečně provedl lze dosáhnout použitím přepínače `-n`.
- znak `-` na začátku řádku způsobí ignorování nenulové návratové hodnoty; jinak `make` vždy v takové situaci zahlásí chyby a okamžitě skončí. Příklad: `basic-utils/Makefile04`.
- ```
  test1:
          false
          echo "OK"

  test2:
          -false
          echo "OK"
  ```
  ```

Makra (make)

- definice makra:
  `name = string`
- pokračování vkládá mezeru
- nedefinovaná makra jsou prázdná
- nezáleží na pořadí definic různých maker
- definice na příkazové řádce:
  `make target name=string`
- vyvolání makra:
  `$name` (pouze jednoznakové *name*),
  `${name}` nebo `$(name)`
- systémové proměnné jsou přístupné jako makra

- když je stejné makro definováno vícekrát, platí jeho poslední definice, viz příklad `basic-utils/Makefile03`.
- makra není možné definovat rekurzivně, viz `basic-utils/Makefile05`:

```
$ cat basic-utils/Makefile05
M=value1
M=$(M) value2
all:
        echo $(M)
$ make -f Makefile05
Variable M is recursive.
```

- často se používají různé rozšířené verze `make` (např. GNU, BSD), které umí, podmíněné sekce v `Makefile`, redefinice proměnných, apod.
- napsat `Makefile` který bude fungovat najednou pro různé verze `make` nemusí být jednoduché, proto existují projekty jako je např. GNU automake. Pro jednoduchý podmínečný překlad v závislosti na systému a kde se dají očekávat různé verze příkazu make, je možné použít například následující kód, který mi fungoval na všech v něm zmíněných systémech (znak ‘ je zpětný apostrof, a ’ je normální apostrof):

```
CFLAGS=‘x=\‘uname\‘; \
        if [ $${x} = FreeBSD ]; then \
```

```
          echo '-Wall'; \
        elif [ $${x} = SunOS ]; then \
          echo '-v'; \
        elif [ $${x} = Linux ]; then \
          echo '-Wall -g'; \
        fi`

all:
        @echo "$(CFLAGS)"
```

- v ostatních situacích je vhodné, případně nezbytné použít programy typu `autoconf` nebo `automake`. Některé `make` implementace mají přímo direktivy pro podmíněné zpracování vstupního souboru, například make na BSD. Příklad: `basic-utils/Makefile08.bsd`.
- přepínačem `-e` můžeme `make` donutit, aby ignoroval nastavení proměnných ve vstupním souboru v případě, že je definována proměnná prostředí stejného jména. Normálně `make` přebírá aktuální nastavení proměnných prostředí jen v případě, že dané proměnné nejsou nastavené ve vstupním souboru. Příklad: `basic-utils/Makefile06`.
- `make` je velmi silný nástroj, stačí se podívat do systémových `Makefile` souborů jakéhokoli unix-like systému (najděte si na Wikipedii, co to znamená "unix-like", pokud to nevíte). Typickou společnou vlastností je to, že neexistuje dokumentace, jak je daný makefile framework postaven.

Dynamický linker (loader)

- překlad vyžaduje všechny potřebné dynamické knihovny pro kontrolu dosažitelnosti použitých symbolů
- **sestavení kompletního programu se ale provede až při spuštění**. To je úkol pro **dynamický linker** (*run-time linker*, *loader*)
- seznam dynamických knihoven zjistí ze sekce `.dynamic`
- systém má nastaveno několik cest, kde se automaticky tyto knihovny hledají
- v sekci `.dynamic` je možné další cesty ke knihovnám přidat pomocí tagů `RUNPATH`/`RPATH`
- nalezené knihovny se připojí do paměťového procesu pomocí volání **mmap**() (bude později)

- proces spuštění dynamicky slinkovaného programu probíhá takto:
  - kernel ve volání `exec` namapuje program do paměti a zjistí, jaký dynamický linker se má použít (viz dále)
  - kernel namapuje linker do paměťového prostoru spouštěného programu a pak linkeru předá kontrolu. Linker je program sám o sobě – na Solarisu je ho možné normálně spustit (má tedy funkci `main`) a jako parametr mu dát jméno programu. Možnost spouštět dynamický linker z příkazového řádku je ale hlavně pro experimentovaní s linkerem při jeho vývoji.
  - linker z hlavičky programu zjistí, jaké dynamické knihovny program používá, namapuje je do paměti a zavolá jejich inicializační funkce, pokud existují. Mapují se všechny nalezené závislosti které nejsou nastavené jako *lazy* (strana 144), rekurzívně prohledáváním do šířky. V tomto pořadí se pak také objekty prohledávají při hledání jednotlivých symbolů.
  - linker programu předá řízení (tj. zavolá funkci `main`)
  - proces může i za běhu dále využívat dynamický linker pomocí volání `dlopen` a spol.; k tomu se dostaneme na straně 144
- je potřeba si uvědomit, že dynamický linker zde nepracuje jako samostatný proces přestože má svoji vlastní `main` funkci, jeho kód se používá v rámci paměťového prostoru procesu; **program, linker a knihovny dohromady tvoří jeden proces**.
- **následující příkazy a příklady se týkají Solarisu**. Pokud to nebude fungovat na jiných systémech, tak mají ekvivaletní nástroje s podobnou funkcionalitou.
  - seznam sekcí se zjistí pomocí `elfdump -c` (GNU má příkazy `objdump` a `elfread`). O programových sekcích bude více na straně 133.
  - to, jaký dynamický linker se použije, kernel zjistí ze sekce `.interp`, viz "`elfdump -i`" a "`ld -I`". To znamená, že si můžete napsat vlastní linker a pomocí `-I` pro `ld` ho pak nastavit jako dynamický linker pro váš program.
  - dynamická sekce se vypíše pomocí `elfdump -d`, dynamické knihovny jsou označené tagem `NEEDED`
  - závislosti na dynamických knihovnách je možné pohodlně zjistit pomocí příkazu `ldd` (Solaris, Linux, BSD), který zjistí konkrétní cesty ke knihovnám. Tento příkaz řeší závislosti rekurzívně a uvidíte tedy i nepřímé závislosti - tj. takové knihovny, které jsou použité knihovnami, které příslušný program používá přímo. Zjistit co je přesně závislé na čem je možné pomocí volby `-v`. Na OS X je ekvivalentem `ldd` příkaz `otool -L`.
  - jaké knihovny byly při spuštění nakonec použity může být jiné než co ukáže příkaz `ldd`, a to třeba díky mechanismu `LD_PRELOAD`. Na Solarisu proto existuje příkaz `pldd`, který pomocí čísla procesu ukáže závislosti konkrétního procesu. Příklad na `LD_PRELOAD`: použijte již zmíněný `Makefile01`, a přeložte `basic-utils/preload.c` takto: "`cc -shared -o libpreload.so preload.c`". Pak spusťte program, který zachytí systémové volání `close`: "`LD_PRELOAD=./libpreload.so ./a.out`".

  - většinu zde uvedených informací naleznete v manuálové stránce pro dynamický linker v Solarisu, `ld.so.1(1)`, a více pak v *Linkers and Libraries Guide* na `docs.oracle.com`. Na FreeBSD se dynamický linker nazývá `ld-elf.so.1`, v linuxových distribucích většinou `ld-linux.so.1`, na IRIXu `rld` atd.
  - dynamický linker se typicky dá konfigurovat pomocí nastavení proměnných, například si zkuste na Solarisu spustit toto:

    ```
    LD_LIBRARY_PATH=/tmp LD_DEBUG=libs,detail date
    ```

    a pro zjištění všech možností jak debugovat dynamický linker použijte:

    ```
    LD_DEBUG=help date
    ```

  - příklad na to, kdy je potřeba, aby linker hledal knihovny i jinde než v defaultních adresářích (adresáře specifikované proměnnou `LD_LIBRARY_PATH` se prohledávají jako první):

    ```
    $ cp /lib/libc.so.1 /tmp
    $ LD_LIBRARY_PATH=/tmp sleep 100 &
    [1] 104547
    $ pldd 104547
    104547: sleep 100
    /tmp/libc.so.1
    /usr/lib/locale/cs_CZ.ISO8859-2/cs_CZ.ISO8859-2.so.3
    ```

  - Solaris má velmi zajímavý příkaz `elfedit(1)`, pomocí kterého můžete editovat metadata ELF objektu, například přepsat jméno závislé knihovny, změnit nastavení `RUNPATH` atd.

- Obecně platí, že používat `LD_LIBRARY_PATH` k ovlivnění běhu dynamického linkeru pro něco jiného než ladění dynamických knihoven při vývoji nebo přesunech knihoven mezi adresáři není dobrý nápad. Na internetu lze najít množství článků typu "why is `LD_LIBRARY_PATH` evil ?"apod., např. http://xahlee.org/UnixResource_dir/_/ldpath.html.

  Tato proměnná je typicky zneužívaná ve startovacích skriptech programů, aby předsunuly alternativní seznam adresářů kde hledat dynamické knihovny na kterých program závisí. To většinou proto, že byl program nesprávně slinkován a dynamický linker by podle informací v ELFu nedokázal jinak knihovny najít. Typický nechtěný side effect je že program spustí jiný program který používá knihovnu stejného jména, ale "díky"tomu, že se proměnné prostředí dědí, tak dynamický linker najde tuto knihovnu jako první v aresáři specifikovaném pomocí `LD_LIBRARY_PATH`. Tato knihovna může být ale jiné verze, a pak snadno může dojít k něčemu co se většinou nazývá "nedefinované chování"(viz příklad u dalšího slide o ABI).

API versus ABI

API – Application Programming Interface

- rozhraní použité pouze ve zdrojovém kódu
- rozhraní **zdrojáku** vůči systému, knihovně či vlastnímu kódu, tj. např. `exit(1)`, `printf("hello\n")` nebo `my_function(1, 2)`
- ... aby se stejný **zdrojový kód** mohl přeložit na všech systémech podporující dané API

ABI – Application Binary Interface

- low-level rozhraní **aplikace** vůči systému, knihovně či jiné části sama sebe
- ... aby se **objektový modul** mohl použít všude tam, kde je podporováno stejné ABI

- Příkladem API je třeba API definované normou POSIX.1.
- ABI definuje konvenci volání (to jak program předá parametry funkci a jak od ní převezme návratovou hodnotu), jaká jsou čísla systémových volání, jak se systémové volání provede či formát objektového modulu a přijímaných argumentů, viz příklad dole.
- API knihovny pak definuje mimo jiné množinu volání která jsou knihovnou definována, jejich parametry a typy těchto parametrů.
- následná ukázka je příklad na to, kdy vývojář změní velikost argumentů v bajtech (tj. změní ABI knihovny), a nahradí novou verzí tu starou. Všimněte si, že dynamický linker toto nezjistí; nemá totiž jak, řídí se podle jména knihovny v dynamické sekci programu, a to se nezměnilo. Uvedená změna je sice i změna v API a problém by se odstranil, kdybychom `main.c` znovu přeložili se změněným řádkem deklarace funkce `add`. To je ale často problém (překládejte celý systém jen kvůli tomu), proto je tak důležité dodržovat zpětnou kompatibilitu v ABI u knihoven.

  Výsledek následujícího překladu knihovny, programu a jeho spuštění je jak bychom očekávali (použit `cc` ze SunStudio, pro `gcc` použijte místo `-G` volbu `-shared`; novější `gcc` navíc neznají `-R` a je místo toho nutné použít `-Xlinker -R .`:

```
$ cat main.c
int my_add(int a, int b);
```

```
int
main(void)
{
        printf("%d\n", my_add(1, 2));
        return (0);
}

$ cat add.c
int
my_add(int a, int b)
{
        return (a + b);
}

$ cc -G -o libadd.so add.c
$ cc -L. -ladd -R. main.c
$ ./a.out
3
```

Nyní ale přišla další verze knihovny se stejným jménem, a ve funkci `my_add` nastala změna v typu argumentů, kde místo 4-bajtového integeru se použije 64-bitový celočíselný typ. Program ale o ničem neví, nechá se spustit a vrátí chybnou hodnotu:

```
$ cat add2.c
int64_t
my_add(int64_t a, int64_t b)
{
        return (a + b);
}

$ cc -G -o libadd.so add2.c
$ ./a.out
-1077941135
```

- příklad: `lib-abi/abi-main.c` (komentář v souboru napoví jak použít ostatní soubory ve stejném adresáři)
- zde pak přichází ke slovu verzování knihoven, tj. je nutné "něco" změnit tak, aby po instalaci nové knihovny nešlo program spustit bez jeho rekompilace.
- binární nekompatibilita je například problém u OpenSSL. Větve 0.9.x a 0.9.y nejsou ABI kompatibilní. Konkrétně verze 0.9.7 a 0.9.8, v roce 2009 stále obě používané. Verze rozlišené písmeny, tj. například 0.9.8a a 0.9.8g, jsou ABI kompatibilní. Některé systémy stále používají pouze 0.9.7 (FreeBSD 6.x, Solaris 10), jiné jen 0.9.8 (Solaris 11 Express), další integrují obě větve (různé Linuxové distribuce). Problém je, máte-li například program pro Solaris 10 používající `libcrypto.so` knihovnu, který chcete používat i na Solaris 11 Express (to je jinak díky zpětné binární kompatibilitě striktně dodržované mezi "major"verzemi Solarisu možné - např. program který běžel na Solarisu 2.6 z roku 1997 může běžet na Solarisu 10 z roku 2009 bez nutnosti rekompilace - to se týká systému a knihoven s ním dodávaných). Jediné správné rešení je zkompilovat pro nový systém, případně manuálně zkopírovat potřebné verze

knihoven, což ale zdaleka není ideální – program nebude fungovat s nově nainstalovaným systémem, ale nikdo najednou neví, proč to funguje na stejném systému vedle, a když se to zjistí tak je opět potřeba manuální zásah, a pochybuji o tom, že autor "řešení" bude instalovat opravené verze při výskytu bezpečnostních chyb. Nekompatibilita 0.9.x verzí je důvodem, proč je v dynamické sekci knihovny i její celé číslo (bez písmen, ta jak již víme nejsou pro ABI kompatibilitu u OpenSSL důležitá), a díky tomu je pak toto číslo uvedeno i v každém programu proti knihovně slinkovanému:

```
$ elfdump -d /usr/sfw/lib/libcrypto.so.0.9.8 | grep SONAME
   [7]  SONAME            0x1                libcrypto.so.0.9.8

$ elfdump -d /usr/bin/ssh | grep NEEDED
   [1]  NEEDED            0x3c99             libsocket.so.1
   [3]  NEEDED            0x3cb1             libnsl.so.1
   [5]  NEEDED            0x3cc6             libz.so.1
   [7]  NEEDED            0x3d12             libcrypto.so.0.9.8
   [9]  NEEDED            0x3cd9             libgss.so.1
  [10]  NEEDED            0x3cfe             libc.so.1
```

- příčinou zpětné nekompatibility OpenSSL verzí je to, že z historických důvodů jsou některé používané struktury v hlavičkových souborech. Tyto struktury je ale někdy nutné rozšířit, například při vývoji nové funkcionality. Tím nastane situace, že program přeložený s verzí 0.9.7 by předal novější knihovně "menší" strukturu, respektive nová knihovna by přistupovala ve staré struktuře na položky, které neexistují – a tedy by přistupovala k paměti, která programu nebyla přidělena. To může způsobit pád programu (přístup na nenamapovanou stránku), může to fungovat dále (v dané paměti je to, co se tam typicky očekává, například nula), nebo se to začne chovat "podivně" (v paměti bylo něco, co v dané situaci očekávané nebylo). Problém v OpenSSL je, že nyní již není technicky jednoduché z těchto struktur udělat interní a navenek pracovat jen s transparentními referencemi objektovým přístupem, což by umožnilo dělat libovolné změny ve strukturách, aniž by to program ovlivnilo.
- běžně viděné řešení způsobené neznalostí věci je vytvořit symbolický link, například na Solarisu 11 udělat 0.9.7 symlink na existující knihovnu verze 0.9.8. Častý výsledek je pak pád programu a údiv autora symlinku. Někdy to naopak funguje, protože program náhodou nepoužívá příslušné struktury, a to je jasným důkazem pro aktéra, že řešení musí být správné. Může se ale stát, že program struktury nepoužívá při konkrétním provedeném testu, ale začne dělat problémy až při živém nasazení. Tady je jediná rada – pokud si opravdu nejste jisti že víte, co děláte a nejste si jistí svoji detailní znalostí kódu programu i knihoven, vyhněte se tomu. Nebo riskujte, ale již víte jak to může skončit.
- zdánlivě jednoduché řešení dodávat více verzí OpenSSL s jedním systémem přináší zase jiné problémy – obtížnější vývoj systému, obtížnější správu systému (při výskytu bezpečnostní chyby je často nutné patchovat všechny instalované verze), problémy s nepřímými závislostmi obsahující více verzí dané knihovny apod.
- upgrade verze OpenSSL v existujícím systému je také věc, které je dobré se vyhnout, respektive toto těžko vyřešíte vydáním patche pro existující systémy – uvažte že zákazník používá své nebo jím koupené programy,

které závisí na existující verzi. A tu byste mu najednou upgradovali na verzi vyšší, ABI nekompatibilní.

– typickým příkladem, kdy se používá transparentní typ jako reference, což umožňuje další rozšiřování pod ní ležící struktury bez rizika výše uvedených problémů, je typ POSIX vláken. Struktura typu `pthread_t` (strana 194) je interní záležitostí knihovny. Typicky je to integer, ale to by programátora nemělo vůbec zajímat. Samozřejmě souborový deskriptor či číslo procesu jsou podobné případy, ale na příkladu vláken je to lépe vidět.

Debugger `dbx`

- Volání:
  `dbx [ options ] [ program [ core ] ]`
- Nejběžnější příkazy:

`run [arglist]`	start programu
`where`	vypiš zásobník
`print expr`	vypiš výraz
`set var = expr`	změň hodnotu proměnné
`cont`	pokračování běhu programu
`next`, `step`	proveď řádku (bez/s vnořením do funkce)
`stop condition`	nastavení breakpointu
`trace condition`	nastavení tracepointu
`command n`	akce na breakpointu (příkazy následují)
`help [name]`	nápověda
`quit`	ukončení debuggeru

- základní řádkový symbolický debugger, aby bylo možné ho plně využít, musí být program přeložen s ladicími informacemi (`cc -g`). Laděný program se startuje z debuggeru příkazem `run`, nebo se debugger připojí k již běžícímu procesu. Pomocí `dbx` lze analyzovat i havarovaný program, který vygeneroval soubor `core`.
- je možné ho najít např. na Solarisu, avšak na Linuxu a FreeBSD defaultně není.
- pro debugging se zdrojovými kódy nestačí použít volbu `-g`, je zároveň nutné mít i zdrojáky a objektové moduly tam, kde byly při překladu. To je typicky běžná situace, protože ladíte na stroji, kde zároveň i vyvíjíte. Pokud tomu tak není, je nutné si zdrojáky a objektové moduly zajistit, pokud k nim vede jiná cesta, lze použít dbx příkaz `pathmap`.
- `gdb`-kompatibilní mód se spustí přes `gdb on`. Pokud vás zajímá, jaký má

`dbx` ekvivalentní příkaz ke konkrétnímu `gdb` příkazu, pomůže vám `help FAQ`; hned první otázka je "A.1 Gdb does <something>; how do I do it in dbx?"

- pokud nepoužijete přepínač -g, bude vám dbx na Solarisu stále platný, protože zobrazí argumenty funkcí. U BSD systémů a linuxových distribucí -g použít musíte, jinak vám debuggery moc nepomohou. Kdy je to na Solarisu platné i bez -g je v vidět v příkladu `debug/dbx.c`. Při kompilaci s gcc a použití gdb neukáže příkaz `where` parametry funkce `crash()` zatímco se Solaris Studio kompilerem a debuggerem dbx se parametry funkce vypíší, takže je vidět hodnota která se přiřazovala.

- příklad: `debug/coredump.c`. Po přeložení a spuštění program spadne a zanechá core dump.

```
$ cc coredump.c
$ ./a.out
Segmentation Fault (core dumped)
$ dbx ./a.out core
Reading a.out
core file header read successfully
Reading ld.so.1
Reading libc.so.1
program terminated by signal SEGV (no mapping at the fault address)
0x08050a05: bad_memory_access+0x0015:   movb     %al,0x00000000(%edx)
(dbx) where
=>[1] bad_memory_access(0x8047ae8, 0x8047a44, ...
  [2] main(0x1, 0x8047a50, 0x8047a58, 0x8047a0c), at 0x8050a1b
```

Vidíme, ve které funkci to spadlo a je možné se vypsat zásobník. Nevidíme ale přesně řádku kódu, kde nastal problém. Pro to je nutné přeložit program s ladícími symboly, tj. "`cc -g coredump.c`". Zájemce o více informací o debugging pod unixem odkazuji na navazující přednášku "Programování v UNIXu II." (NSWI138).

```
$ cc -g coredump.c
$ dbx ./a.out core
Reading a.out
core file header read successfully
Reading ld.so.1
Reading libc.so.1
program terminated by signal SEGV (no mapping at the fault address)
Current function is bad_memory_access
    8           x[0] = '\0';
(dbx)
```

GNU debugger gdb

- Volání:
  `gdb [ options ] [ program [ core ] ]`
- Nejběžnější příkazy:

`run [arglist]`	start programu
`bt`	vypiš zásobník
`print expr`	vypiš výraz
`set var = expr`	změň hodnotu proměnné
`cont`	pokračování běhu programu
`next`, `step`	proveď řádku (bez/s vnořením do funkce)
`break condition`	nastavení breakpointu
`help [name]`	nápověda
`quit`	ukončení debuggeru

- GNU obdoba `dbx`. Mód kompatibilní s `dbx` spustíte přes `-dbx`.
- na různých platformách existují i debuggery s grafickým rozhraním, např. `workshop` (Solaris), `cvd` (IRIX), `xxgdb` (GNU), `ddd` (GNU). Často fungují jako nadstavby nad `dbx`, `gdb`.
- ```
  #include <stdio.h>
  int main(void) {
    printf("hello, world\n");
    return (0);
  }
  $ cc -g main.c
  $ gdb -q a.out
  (gdb) break main
  Breakpoint 1 at 0x8048548: file main.c, line 4.
  (gdb) run
  Starting program: /share/home/jp/src/gdb/a.out

  Breakpoint 1, main () at main.c:4
  4         printf("hello, world\n");
  (gdb) next
  hello, world
  5         return (0);
  (gdb) c
  Continuing.
  Program exited normally.
  ```
  ```

```
(gdb) q
```

- debuggery jsou výbornými pomocníky pokud váš program končí na chyby typu "segmentation error" – tj. když zkusíte nekorektně přistoupit do paměti, například tam kde nemáte co dělat. Když při překladu použijete option `-g`, ukáže vám pak debugger přesně číslo řádku, kde nastal problém. Konkrétní příklad (proč se vlastně tento program chová jak se chová??? Hint: zkuste přeložit na Solarisu překladačem `cc` a spustit):

```
$ cat -n main.c
     1  int
     2  main(void)
     3  {
     4          char *c = "hey world";
     5          c[0] = '\0';
     6          return (0);
     7  }
}
$ gcc -g main.c
$ ./a.out
Bus error (core dumped)
$ gdb a.out a.out.core
...
Core was generated by `a.out'.
Program terminated with signal 10, Bus error.
...
#0  0x080484e6 in main () at main.c:5
5               c[0] = '\0';
```

Obsah

- úvod, vývoj UNIXu a C, programátorské nástroje
- **základní pojmy a konvence UNIXu a jeho API**
- přístupová práva, periferní zařízení, systém souborů
- manipulace s procesy, spouštění programů
- signály
- synchronizace a komunikace procesů
- síťová komunikace
- vlákna, synchronizace vláken
- ??? - bude definováno později, podle toho kolik zbyde času

Standardní hlavičkové soubory (ANSI C)

`stdlib.h`	...	základní makra a funkce
`errno.h`	...	ošetření chyb
`stdio.h`	...	vstup a výstup
`ctype.h`	...	práce se znaky
`string.h`	...	práce s řetězci
`time.h`	...	práce s datem a časem
`math.h`	...	matematické funkce
`setjmp.h`	...	dlouhé skoky
`assert.h`	...	ladicí funkce
`stdarg.h`	...	práce s proměnným počtem parametrů
`limits.h`	...	implementačně závislé konstanty
`signal.h`	...	ošetření signálů

- hlavičkový soubor (*header file*) je soubor s deklaracemi funkcí (*forward declaration*), proměnných a maker. Z pohledu preprocesoru je to obyčejný soubor napsaný v jazyce C.
- **tyto hlavičkové soubory nejsou specifické pro UNIX. Jsou součástí standardu ANSI C, který jak již víme (strana 16), je zahrnut v POSIX.1. Je ale důležité si uvědomit, že tyto hlavičkové soubory musí podporovat každý systém, který podporuje ANSI C, ať již podporuje POSIX.1 nebo ne.**
- příslušný hlavičkový soubor pro konkrétní funkci najdete v manuálové stránce dané funkce, toto je začátek manuálové stránky na Solarisu pro `memcpy`:

```
Standard C Library Functions                          memory(3C)

NAME
   memory, memccpy, memchr, memcmp, memcpy, memmove,  memset  -
   memory operations

SYNOPSIS
   #include <string.h>
...
...
```

- jednotlivá makra obsažená v těchto souborech většinou nejsou vysvětlena, význam jednotlivých maker je ale možné si vyhledat v příslušných specifikacích, které jsou on-line. Na některých systémech (Solaris) mají jednotlivé hlavičkové soubory svoji vlastní manuálovou stránku (`man stdlib.h`).
- makro `assert` je možné z během kompilace odstranit pomocí makra `NDEBUG`. Příklad: `assert/assert.c`.

```
cat assert.c
#include <assert.h>

int
main(void)
{
        assert(1 == 0);
        return (13);
}
$ cc assert.c
$ ./a.out
Assertion failed: 1 == 0, file assert.c, line 6
Abort (core dumped)
$ cc -DNDEBUG assert.c
$ ./a.out
$ echo $?
13
```

Standardní hlavičkové soubory (2)

`unistd.h`	...	symbolické konstanty, typy a základní funkce
`sys/types.h`	...	datové typy
`fcntl.h`	...	řídící operace pro soubory
`sys/stat.h`	...	informace o souborech
`dirent.h`	...	procházení adresářů
`sys/wait.h`	...	čekání na synovské procesy
`sys/mman.h`	...	mapování paměti
`curses.h`	...	ovládání terminálu
`regex.h`	...	práce s regulárními výrazy

- tyto headery už patří do UNIXu.
- zajímavé může být podívat se do `sys/types.h`

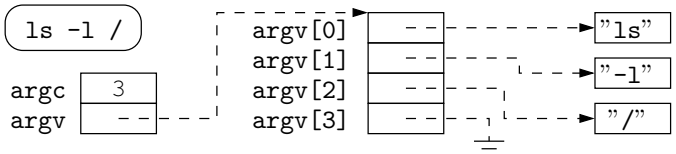
Standardní hlavičkové soubory (3)

`semaphore.h`	...	semafory (POSIX)
`pthread.h`	...	vlákna (POSIX threads)
`sys/socket.h`	...	síťová komunikace
`arpa/inet.h`	...	manipulace se síťovými adresami
`sys/ipc.h`	...	společné deklarace pro System V IPC
`sys/shm.h`	...	sdílená paměť (System V)
`sys/msg.h`	...	fronty zpráv (System V)
`sys/sem.h`	...	semafory (System V)

- dokončení nejdůležitějších UNIXových headerů. Existují samozřejmě ještě další.

Funkce `main()`

- při spuštění programu je předáno řízení funkci `main()`.
- `int` **`main`** `(int` *`argc`*`, char *`*`argv`*`[]);`
  - `argc` ... počet argumentů příkazové řádky
  - `argv` ... pole argumentů
    - podle konvence je `argv[0]` jméno programu (bez cesty)
    - poslední prvek je `argv[argc] == NULL`
  - návrat z `main()` nebo volání `exit()` ukončí program
  - standardní návratové hodnoty `EXIT_SUCCESS` (0) a `EXIT_FAILURE` (1)



- první parametr (typu `int`) udává počet argumentů na příkazovém řádku (včetně argumentu 0 – jména programu) a druhý parametr (typu `char**`) je pole ukazatelů na tyto řetězce. Za posledním řetězcem je ještě ukončovací `NULL` pointer – pozor na to, že to je něco jiného než prázdný řetězec.

- `argv[0]` někdy bývá důležitým zdrojem přidané informace. Na Solarisu jsou například příkazy `cp`, `mv` a `ln` nalinkované na stejný soubor. Hodnota `argv[0]` pak určuje, jakou funkci vlastně proces má. Jiný příklad – pokud má shell první znak z `argv[0]` nastaven na "-", znamená to, že se má chovat jako login shell (podívejte se třeba do manuálové stránky pro `bash` na sekci INVOCATION, pokud nevíte, co to je login shell). Ve výpisu procesů pak uvidíte "-bash". Toto není součást UNIX specifikace pro `sh`, ale používal to již Bourne shell na UNIXu V7 (1979) a ostatní shelly to převzaly.

- při spuštění programu předá kód dynamického linkeru řízení funkci `main`, viz strana 35. Staticky slinkovanému programu se řízení předá přímo. Nepřítomnost `main` v programu způsobí chybu při překladu na úrovni linkeru. Této funkci se předá jméno spuštěného programu, argumenty z příkazové řádky a případně i proměnné prostředí. Ukončení této funkce znamená konec programu a návratová hodnota se použije jako kód ukončení programu pro OS. Jinou možností ukončení programu je použití funkce `exit` nebo `_exit`, kterou lze použít kdykoliv, nejen ve funkci `main`. V C lze používat obě metody ukončení programu.

- předání proměnných prostředí třetím parametrem typu `char**` není součástí normativní části C standardu, pouze informativní. Překladače to ale typicky

podporují. Varianta `main` s proměnnými prostředí pak vypadá takto:
`int main(int argc, char *argv[], char *envp[]);`

- návratový typ funkce `main` by měl být vždy `int`; překladač si jinak bude stěžovat. **Z této hodnoty se ale použije pouze nejnižších 8 bitů**, a je to nezáporné číslo. Pozor na to, že na rozdíl od konvence jazyka C návratová hodnota 0 má v shellu význam true (úspěch) a nenula význam false (neúspěch). Typická konstrukce v shellu vypadá takto:

  ```
  if ! prog; then
          echo "failure"
  else
          echo "success"
  fi
  ```

  Příklad: `main/return-256.c`.

- nepoužívejte proto ve funkci `main` `return (-1)` a nikde pak ani `exit(-1)`, z toho vznikne návratová hodnota 255 a kód je matoucí. Je vůbec velmi vhodné používat pouze 0 a 1, pokud není zásadní důvod pro jiné hodnoty, třeba ten že jich potřebujete více – můžete se podívat například na manuálovou stránku pro `passwd` na Solarisu, sekce `EXIT STATUS`. Příklad: `main/return-negative-1.c`.

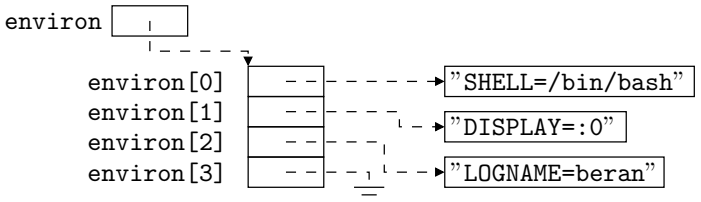
- rozdíl mezi funkcemi `exit` a `_exit` je v tom, že `exit` před ukončením programu ještě vyprázdní (pomocí knihovní funkce `fflush`) a zavře streamy a volá funkce zaregistrované pomocí `atexit`. V závislosti na systému to mohou být i další akce.

  Pro zajímavost, například ve FreeBSD je `_exit` systémové volání a `exit` knihovní funkce, ale na Solarisu jsou obě systémovými voláními. Příklad: `exit/exit.c`

- příklad: `main/print-argv.c`

Proměnné prostředí

- seznam všech proměnných prostředí (*environment variables*) se předává jako proměnná
  `extern char **environ;`
- je to pole ukazatelů (ukončené NULL) na řetězce ve tvaru:
  `proměnná=hodnota`



- shell předává spuštěnému programu ty proměnné, které jsou označeny jako exportované (Bourne-like shellech příkazem `export` *`variable`*). Po změně obsahu již jednou exportované proměnné samozřejmě není potřeba proměnnou znovu exportovat. Příkaz `env` vám vypíše aktuální proměnné prostředí. Abyste přidali proměnnou do prostředí spouštěného programu stačí provést přiřazení na příkazové řádce, aniž byste museli měnit prostředí vašeho shellu:

  ```
  $ date
  Sun Oct  7 13:13:58 PDT 2007
  $ LC_TIME=fr date
  dimanche  7 octobre 2007 13 h 14 PDT
  ```

  nedivte se, pokud to na vašem systému takto fungovat nebude, v tom případě nemáte instalovaný balík s francouzskou lokalizací (což je pravděpodobné).

- při nahrazení aktuálního obrazu procesu obrazem jiným se předává, pokud se neřekne jinak, synovským procesům celé pole `environ` automaticky. Je možné ve volání příslušné varianty funkce `exec` předat pole jiné.

- jaké proměnné prostředí konkrétní příkaz používá (a jak je používá) by mělo být v manuálové stránce. Typicky v sekci nazvané *ENVIRONMENT* nebo *ENVIRONMENT VARIABLES*

- `man` například používá `PAGER`, `vipw` pak proměnnou `EDITOR` apod.

- pokud je `envp` třetím parametrem funkce `main`, tak je to stejná hodnota co je v ukazateli `environ`.

- příklad: main/print-env.c (včetně použití `env` příkazu způsobem, který vyčistí zděděné proměnné prostředí).

```
$ cc print-env.c
$ env - XXX=yyy aaa=ABC ./a.out
aaa=ABC
XXX=yyy
```

Manipulace s proměnnými prostředí

- je možné přímo měnit proměnnou `environ`, SUSv3 to ale nedoporučuje
- `char *getenv (const char *name);`
  - vrátí hodnotu proměnné name
- `int putenv (char *string);`
  - vloží string ve tvaru *jméno=hodnota* do prostředí (přidá novou nebo modifikuje existující proměnnou)
- změny se přenášejí do synovských procesů
- změny v prostředí syna samozřejmě prostředí otce neovlivní
- existují i funkce **setenv**() a **unsetenv**()

- u `putenv` se vložený řetězec stane součástí prostředí (jeho pozdější změna tak změní prostředí) a nesmíte proto používat retězce v automatických proměnných, toto řeší `setenv`, který hodnotu proměnné zkopíruje. Viz příklad main/putenv.c.
- důležité je zapamatovat si, že synovský proces zdědí v okamžiku svého vzniku od rodiče všechny proměnné prostředí, ale jakákoliv manipulace s nimi v synovi je lokální a do otce se nepřenáší. Každý proces má svou kopii proměnných, proto ani následná změna prostředí otce nezmění proměnné již existujícího potomka.
- další rozdíl mezi `putenv` a `setenv` je ten, že v `setenv` mohu definovat, zda existující proměnnou chci nebo nechci přepsat. `putenv` vždy přepisuje.
- ```
  int
  main(void)
  {
          printf("%s\n", getenv("USER"));
  ```
  ```

```
        return (0);
}
$ ./a.out
jp
```

- jelikož proměnná `environ` je obyčejné pole ukazatelů na řetězce, není neběžné narazit na kód, který s tímto polem pracuje přímo. Pozor však na to, že v takovém případě pak již nesmíte používat zde uvedené funkce, jinak se můžete dostat do problémů s jejich konkrétní implementací. A hlavně, nový kód takto nepište, protože norma SUSv3 přímou práci s tímto polem nedoporučuje.
- příklad: `main/getenv.c`
- pozor na to, že je rozdíl mezi nastavením hodnoty proměnné na prázdný string a odmazáním proměnné ze seznamu proměnných (pomocí `unsetenv`).

Zpracování argumentů programu

- obvyklý zápis v shellu: `program -přepínače argumenty`
- přepínače tvaru `-x` nebo `-x` *`hodnota`*, kde `x` je jedno písmeno nebo číslice, *`hodnota`* je libovolný řetězec
- několik přepínačů lze sloučit dohromady: `ls -lRa`
- argument '`--`' nebo první argument nezačínající '`-`' ukončuje přepínače, následující argumenty nejsou považovány za přepínače, i když začínají znakem '`-`'.
- tento tvar argumentů požaduje norma a lze je zpracovávat automaticky funkcí `getopt`.

- argumenty lze samozřejmě zpracovávat vlastní funkcí, ale standardní funkce je pohodlnější.
- argumenty se typicky mohou opakovat, ale to má smysl jen v některých situacích
- pořadí přepínačů může být důležité a je na aplikaci, aby toto specifikovala
- UNIX norma definuje pomocí 13 pravidel velmi přesně, jak by měly vypadat názvy příkazů a formát přepínačů. Například jméno příkazu by mělo být pouze malými písmeny, dlouhé 2–9 znaků, z přenositelné znakové sady.

Přepínače bez argumentů by mělo být možné dát do skupiny za jedním znakem '–'. Atd.

- používat číslice jako přepínače je zastaralé; je to někde v normě SUSv3, i když já to v ní nenašel.
- pozor na Linux a jeho (poněkud zvláštní a nestandardní) permutování argumentů
- přepínač `-W` by měl být rezervovaný pro vendor options, tj. pro nepřenositelná rozšíření

Zpracování přepínačů: `getopt()`

```
int getopt(int argc, char *const argv[],
           const char *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
```

- funkce dostane parametry z příkazového řádku, při každém volání zpracuje a vrátí další přepínač. Pokud má přepínač hodnotu, vrátí ji v `optarg`.
- když jsou vyčerpány všechny přepínače, vrátí -1 a v `optind` je číslo prvního nezpracovaného argumentu.
- možné přepínače jsou zadány v `optstring`, když za znakem přepínače následuje '`:`', má přepínač povinnou hodnotu.
- při chybě (neznámý přepínač, chybí hodnota) vrátí '`?`', uloží znak přepínače do `optopt` a když `opterr` nebylo nastaveno na nulu, vypíše chybové hlášení.

- obvykle se nejprve pomocí `getopt` načtou přepínače a pak se vlastními prostředky zpracují ostatní argumenty; často jsou to jména souborů.
- je konvencí, že volby v parametru `optstring` jsou setříděné.

Příklad použití `getopt()`

```
struct {
    int a, b; char c[128];
} opts;
int opt; char *arg1;

while((opt = getopt(argc, argv, "abc:")) != -1)
    switch(opt) {
        case 'a': opts.a = 1; break;
        case 'b': opts.b = 1; break;
        case 'c': strcpy(opts.c, optarg); break;
        case '?': fprintf(stderr,
            "usage: %s [-ab] [-c Carg] arg1 arg2 ...\n",
            basename(argv[0])); break;
    }
arg1 = argv[optind];
```

- dobrým zvykem je při detekování neznámého přepínače nebo špatného zápisu parametrů programu vypsat stručnou nápovědu, případně s odkazem na podrobnější dokumentaci, a ukončit program s chybou, tj. s nenulovou návratovou hodnotou.
- příklad také ukazuje nebezpečné použití funkce `strcpy`
- z použití funkce `getopt` je vidět, že je stavová. Zpracovat další pole argumentů, případně začít opět od začátku, je možné nastavením externí proměnné `optreset` na 1.
- standardní `getopt` zachová pořadí přepínačů při zpracování
- při použití nedefinovaného přepínače funkce vypíše chybu; to lze potlačit nastavením `opterr` na 0.
- příklad: shellový skript `getopt/getopts.sh` přepsaný do jazyka C pomocí `getopt` funkce, `getopt/getopt.c`

Dlouhý tvar přepínačů

- poprvé se objevilo v GNU knihovně `libiberty`:

  `--jméno nebo --jméno=hodnota`

- argumenty se permutují tak, aby přepínače byly na začátku, např. `ls * -l` je totéž jako `ls -l *`, standardní chování lze docílit nastavením proměnné `POSIXLY_CORRECT`.
- zpracovávají se funkcí **getopt_long()**, která používá pole struktur popisujících jednotlivé přepínače:

```
struct option {
    const char *name; /* jméno přepínače */
    int has_arg; /* hodnota: ano, ne, volitelně */
    int *flag; /* když je NULL, funkce vrací val, jinak vrací 0
                  a dá val do *flag */
    int val; /* návratová hodnota */
};
```

verze jak se objevila ve FreeBSD (funkce getopt_long není standarizovaná), má následující vlastnosti:

- pokud všechny dlouhé přepínače mají nastaveny krátkou variantu ve `val`, je chování `getopt_long` kompatibilní s `getopt`
- je možné zadávat argument k dlouhému přepínači i s mezerou (například `--color green`)
- pokud je nastaven `flag`, tak `getopt_long` vrací 0, čímž se tyto dlouhé přepínače bez krátké varianty zpracují v jedné větvi příkazu `case`
- existuje i volání `getopt_long_only`, které povoluje i dlouhé přepínače uvozené jednou uvozovkou (`-option`)
- funkci `getopt_long` je možné používat dvěmi způsoby. První způsob je, že každý dlouhý přepínač má korespondující krátký – takto lze jednoduše přidat dlouhé přepínače do existujícího programu a je **kompatibilní s getopt**. Druhý způsob umožňuje mít samostatné dlouhé přepínače. V tom případě funkce vrací vždy 0 (nekompatibilita s `getopt`) a proměnná `*flag` se nastaví na `val`.
- na konkrétním příkladu na následující stránce je vidět, jak to celé funguje

Dlouhé přepínače (pokračování)

```
int getopt_long(int argc, char * const argv[],
                const char *optstring,
                const struct option *longopts,
                int *longindex);
```

- `optstring` obsahuje jednopísmenné přepínače, `longopts` obsahuje adresu pole struktur pro dlouhé přepínače (poslední záznam pole obsahuje samé nuly)
- pokud funkce narazí na dlouhý přepínač, vrací odpovídající `val` nebo nulu (pokud `flag` nebyl `NULL`), jinak je chování shodné s `getopt`.
- do `*longindex` (když není `NULL`) dá navíc index nalezeného přepínače v `longopts`.

- toto je upravený příklad z manuálové stránky na FreeBSD:

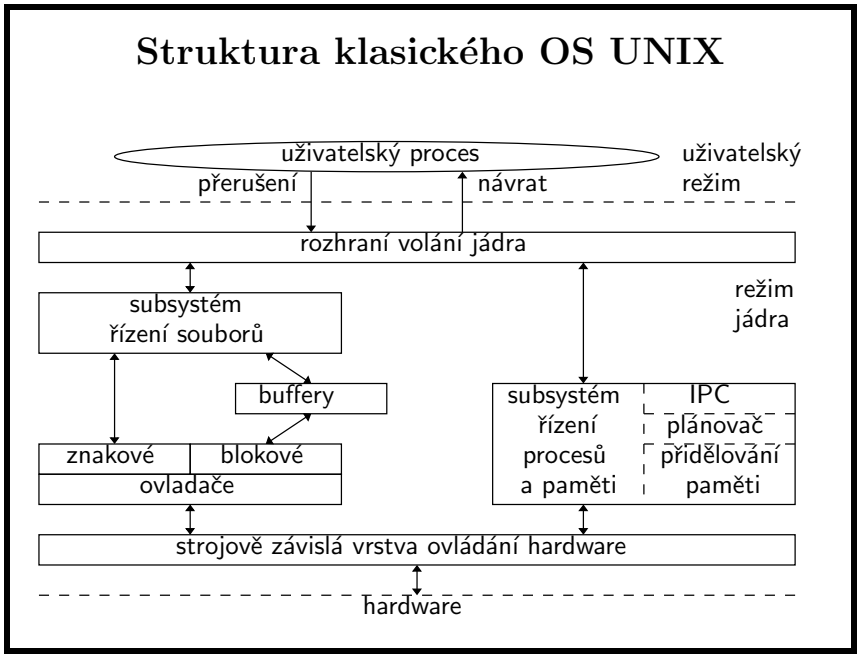
```
#include <stdio.h>
#include <getopt.h>
#include <fcntl.h>

int ch, fd, daggerset, bflag = 0;

static struct option longopts[] = {
       { "buffy",      no_argument,            NULL,           'b' },
       { "fluoride",   required_argument,      NULL,           'f' },
       { "daggerset",  no_argument,            &daggerset,     1 },
       { NULL,         0,                      NULL,           0 }};

int main(int argc, char **argv)
{
  while ((ch = getopt_long(argc, argv, "bf:", longopts, NULL)) != -1)
         switch (ch) {
         case 'b':
                 bflag = 1; break;
         case 'f':
                 if ((fd = open(optarg, O_RDONLY, 0)) == -1)
                         printf("unable to open %s", optarg);
                 break;
         case 0:
                 if (daggerset) {
                         printf("Buffy will use her dagger to "
                             "apply fluoride to dracula's teeth\n");
                 }
                 break;
         default: printf("usage: ...\n");
  }
```

```
  argc -= optind; argv += optind;
  return 0;
}
```



- toto schéma je převzato z [Bach86], viz literatura. Zdůrazňuje dva ústřední pojmy v modelu systému UNIX – soubory a procesy. **V dnešní době to vypadá velmi odlišně, ale pro nás stačí tato základní představa.**
- UNIX rozlišuje dva režimy běhu procesoru: *uživatelský režim* a *režim jádra*. V uživatelském režimu nejsou přístupné privilegované instrukce (např. mapování paměti, I/O, maskování přerušení). Tyto dva režimy musí být podporovány na hardwarové úrovni (procesorem).
- procesy běží obvykle v uživatelském režimu, do režimu jádra přechází buď instrukcí synchronního přerušení (trap) pro volání služby jádra, nebo na základě asynchronních přerušení (hodiny, I/O). Dále se v režimu jádra ošetřují výjimečné stavy procesoru (výpadek stránky, narušení ochrany paměti, neznámá instrukce apod.). Některé speciální akce jsou zajišťovány systémovými procesy, které běží celou dobu v režimu jádra.
- klasické UNIXové jádro je tvořeno monolitickým kódem. Původně bylo potřeba vygenerovat (tj. přeložit ze zdrojových textů a slinkovat) jádro při změně některého parametru nebo přidání ovladače zařízení. V novějších implementacích je možno nastavovat parametry jádra, někdy i za běhu, pomocí systémových utilit bez nutnosti rekompilace jádra. Moderní unixové systémy umožňují rozšiřovat kód jádra za běhu pomocí tzv. modulů jádra (*loadable kernel modules*). Například systém FreeBSD 5.4-RELEASE má 392 takových modulů.

- existují dva způsoby práce s periférie mi: bloková (*block devices*) a znaková zařízení (*character, raw devices*). Data z blokových zařízení (např. disky) procházejí přes vyrovnávací paměti (*buffers*) po blocích, znaková zařízení (např. terminály) umožňují pracovat s jednotlivými bajty a nepoužívají vyrovnávací paměť.
- **jádro není samostatný proces**, ale je částí každého uživatelského procesu. Když jádro něco vykonává, tak vlastně proces, běžící v režimu jádra, něco provádí.

Procesy, vlákna, programy

- **proces** je systémový objekt charakterizovaný svým kontextem, identifikovaný jednoznačným číslem (**process ID**, **PID**); jinými slovy ,,kód a data v paměti"
- **vlákno (thread)** je systémový objekt, který existuje uvnitř procesu a je charakterizován svým stavem. Všechna vlákna jednoho procesu sdílí stejný paměťový prostor kromě registrů procesoru a zásobníku; ,,linie výpočtu", ,,to, co běží"
- **program** ... soubor přesně definovaného formátu obsahující instrukce, data a služební informace nutné ke spuštění; ,,spustitelný soubor na disku"
  - **paměť** se přiděluje **procesům**.
  - **procesory** se přidělují **vláknům**.
  - vlákna jednoho procesu mohou běžet na různých procesorech.

- ***kontext*** je paměťový prostor procesu, obsah registrů a datové struktury jádra týkající se daného procesu
- jinak řečeno – kontext procesu je jeho stav. Když systém vykonává proces, říká se, že běží v kontextu procesu. Jádro (klasické) obsluhuje přerušení v kontextu přerušeného procesu.
- vlákna se dostala do UNIXu až později, původně v něm existovaly pouze procesy, které měly z dnešního pohledu pouze jedno vlákno. Možnost použít v procesu více vláken byla zavedena, protože se ukázalo, že je vhodné mít více paralelních linií výpočtu nad sdílenými daty.
- paměťové prostory procesů jsou navzájem izolované, ale procesy spolu mohou komunikovat. Později se dozvíme, že mohou i částečně sdílet paměť.
- procesy jsou entity na úrovni jádra, ale vlákna mohou být částečně nebo zcela implementována knihovními funkcemi. V případě implementace pomocí

knihovních fukncí to znamená, že vlákna nemusí jádro vůbec podporovat. S vlákny je spojena menší režie než s procesy.

- systémový proces, který běží na pozadí obvykle po celou dobu běhu systému a zajišťuje některé systémové služby (`inetd`, `cron`, `sendmail`...) se nazývá *démon* (angl. *daemon*). Systém BSD tedy nemá ve znaku čerta, ale démona.

Jádro, režimy, přerušení (klasický UNIX)

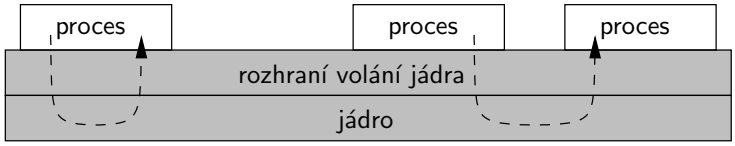
- procesy typicky běží v uživatelském režimu
- systémové volání způsobí přepnutí do režimu jádra
- proces má pro každý režim samostatný zásobník
- jádro je částí každého uživatelského procesu, není to samostný proces (procesy)
- přepnutí na jiný proces se nazývá *přepnutí kontextu*
- obsluha přerušení se provádí v kontextu přerušeného procesu
- klasické jádro je nepreemptivní

- **jádro není oddělená množina procesů, běžících paralelně s uživatelskými procesy, ale je částí každého uživatelského procesu.**
- přechod mezi uživatelským režimem a režimem jádra není přepnutí kontextu – proces běží pořád v tom samém
- přerušený proces nemusel přerušení vůbec způsobit
- v režimu jádra může proces přistupovat i k adresám jádra, která z uživatelského režimu přístupná nejsou; taktéž může přistupovat k instrukcím (např. instrukce manipulující se stavovým registrem), jejichž vykonání v uživatelském režimu vede k chybě
- přerušovací rutina se nemůže zablokovat, protože tím by zablokovala proces; proces se totiž může zablokovat jen ze své vlastní vůle. Moderní unixy dnes používají interrupt vlákna, v jejichž kontextu se **mohou** drivery zablokovat.
- to, že klasické unixové jádro je nepreemptivní znamená, že **jeden proces nemůže zablokovat jiný proces**

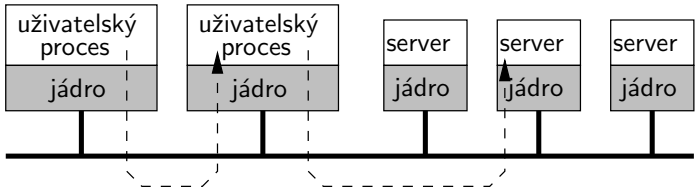
- při obsluze přerušení se může stát, že nastane další přerušení. Pokud je jeho priorita větší, je procesorem přijmuto. Posloupnost přijmutých přerušení je uchována v *zásobníku kontextových vrstev*.
- **u moderních kernelů je situace často velmi rozdílná – obsluha přerušení, preemptivnost kernelu atd.; k některým věcem se možná dostaneme později během semestru**

Volání služeb a komunikace mezi procesy

- UNIX



- distribuovaný OS



- pokud unixový proces vyžaduje provedení systémové služby, pomocí systémového volání předá řízení jádru. Jádro je kus kódu sdílený všemi procesy (ovšem přístupný jen pro ty, které jsou právě v režimu jádra). Jádro tedy není samostatný privilegovaný proces, ale vždy běží v rámci některého procesu (toho, který požádal jádro o službu, nebo toho, který běžel v okamžiku příchodu přerušení).
- komunikace mezi procesy v UNIXu je řešena pomocí systémových volání, je tedy zprostředkovaná jádrem.
- aby to nebylo tak jednoduché, mohou existovat systémové procesy (označované jako *kernel threads*), které běží celou dobu v režimu jádra. Naprostá většina systémových procesů však běží v uživatelském režimu a liší se jen tím, že mají větší přístupová práva. Plánovač procesů přepíná mezi procesy a tím umožňuje běh více procesů současně i na jednom procesoru. Na multiprocesorových počítačích pak funguje skutečný paralelismus procesů a vláken (dokonce se proces může při přeplánování dostat i na jiný procesor).
- v distribuovaném operačním systému má jádro obvykle formu mikrojádra, tj. zajišťuje pouze nejzákladnější služby řízení procesoru, přidělování paměti

a komunikace mezi procesy. Vyšší systémové služby, které jsou v UNIXu součástí jádra (např. přístup k systému souborů) jsou realizovány speciálními procesy (servery) běžícími v uživatelském režimu procesoru. Jádro předá požadavek uživatelského procesu příslušnému serveru, který může běžet i na jiném uzlu sítě.

- dostupných mikrokernelů je v dnešní době mnoho. Můžete zkusit například Minix (unix-like výukový systém), případně systém HURD, který běží nad mikrojádrem Mach.

Systémová volání, funkce

- v UNIXu se rozlišují **systémová volání** a **knihovní funkce**. Toto rozlišení dodržují i manuálové stránky: sekce **2** obsahuje systémová volání (*syscalls*), sekce **3** knihovní funkce (*library functions*).
  - knihovní funkce se vykonávají v uživatelském režimu, stejně jako ostatní kód programu.
  - systémová volání mají také tvar volání funkce. Příslušná funkce ale pouze zpracuje argumenty volání a předá řízení jádru pomocí instrukce synchronního přerušení. Po návratu z jádra funkce upraví výsledek a předá ho volajícímu.
- standardy tyto kategorie nerozlišují – z hlediska programátora je jedno, zda určitou funkci provede jádro nebo knihovna.

- zjednodušeně lze říci, že systémové volání je funkce, která jen upraví své argumenty do vhodné podoby, přepne režim procesoru a skutečnou práci nechá na jádru. Nakonec zase upraví výsledek. **Knihovní funkce může a nemusí volat jádro, ale vždy sama dělá nějakou netriviální činnost v uživatelském režimu.**
- v assembleru je možné zavolat volání jádra přímo
- API jádra je definované na úrovni volání funkcí standardní knihovny, nikoliv na úrovni přerušení a datových struktur používaných těmito funkcemi pro předání řízení jádru. Mechanismus přepnutí mezi uživatelským režimem a režimem jádra se totiž může lišit nejen v závislosti na hardwarové platformě, ale i mezi různými verzemi systému na stejném hardwaru.

Návratové hodnoty systémových volání

- celočíselná návratová hodnota (`int`, `pid_t`, `off_t`, apod.)
  - `>= 0` ... operace úspěšně provedena
  - `== -1` ... chyba
- návratová hodnota typu ukazatel
  - `!= NULL` ... operace úspěšně provedena
  - `== NULL` ... chyba
- po neúspěšném systémovém volání je kód chyby v globální proměnné `extern int errno;`
- úspěšné volání nemění hodnotu v `errno`! Je tedy třeba nejprve otestovat návratovou hodnotu a pak teprve `errno`.
- chybové hlášení podle hodnoty v `errno` vypíše funkce `void perror(const char *s);`
- textový popis chyby s daným číslem vrátí funkce `char *strerror(int errnum);`

- v Solarisu je hodnota `errno` ve skutečnosti knihovnou `libc` definovaná jako dereferencovaný pointer na integer (specifický pro daný userland thread) a hodnota se nastavuje ihned po výstupu z instrukce pro systémové volání. Např. na i386 architektuře je hodnota `errno` po návratu z kernelu (po dokončení instrukce `sysenter`) uložena v registru `eax` (před voláním v ní bylo číslo syscallu). Je to tedy knihovna `libc` kdo je zodpovědný za to, že program uvidí správnou hodnotu `errno`.
- funkce pro práci s vlákny `pthread_*` nenastavují `errno`, ale vrací buď nulu (úspěch) nebo přímo kód chyby.
- pro některá volání může mít smysl i návratová hodnota `-1`. Pak je třeba nejprve nastavit `errno = 0` a po návratu zkontrolovat, zda se `errno` změnilo. Např. funkce `strtol` vrací při chybě 0, což je platná hodnota i pro správný výsledek (a -1 je samozřejmě platný výsledek také).
- je tedy vždy nutné si přečíst manuálovou stránku pro příslušné volání nebo knihovní funkci
- pozn.: úspěšnost funkcí ze `stdio.h` je třeba testovat pomocí `int ferror(FILE *stream)`, protože jinak nelze rozlišit mezi chybou a koncem streamu. Vzhledem k tomu, že tyto funkce nepoužíváme (kromě `printf` a `fprintf` na `stdout`, resp. `stderr`), neměli byste ji potřebovat.

Skupina funkcí z `err(3)`

- pomocné funkce pro výpis chybových hlášení a případné ukončení programu
- místo `perror()` a `exit()` vám stačí jedna funkce
- `void` **`err`**`(int` *`status`*`, const char *`*`fmt`*`, ...);`
  - vypíše jméno programu, formátovaný řetězec, a chybu podle aktuální hodnoty `errno`
  - ukončí program s návratovou hodnotou ze *`status`*
- `void` **`warn`**`(const char *`*`fmt`*`, ...);`
  - stejné jako **err**(), ale program neukončí
- existují další podobné funkce, viz manuálová stránka
- funkce pocházejí ze 4.4BSD

- větší programy podobné funkce často nepoužívají, například proto, že stejně potřebují provést různé ukončovací práce před skončením programu, takže mají podobné funkce vlastní. Pro jiné programy jsou ale tyto funkce ideální, protože mají rozumný výstup a šetří řádky vašeho kódu.
- tyto funkce nemusí být všude, například nejsou v `libc.so` pro Solaris 10 (ale jsou v Solarisu 11). Jsou na BSD systémech a v linuxových distribucích. Tyto funkce nejsou součástí SUS, takže ani na certifikovaných UNIX systémech se nelze spolehnout na jejich přítomnost. Řešením je kontrolovat jejich existenci v předkompilačním skriptu a případně mít jejich verzi součástí zdrojových kódů programu pro případ že nebudou na cílovém systému nalezeny.
- příklad (viz také `err/err.c`):

```
#include <errno.h>
#include <err.h>

int
main(void)
{
        errno = 13;
        err(3, "ggr %s", "GRR");
        printf("after err()\n");
        return (0);
}
```

```
$ ./a.out
a.out: ggr GRR: Permission denied
$ echo $?
3
```

Obsah

- úvod, vývoj UNIXu a C, programátorské nástroje
- základní pojmy a konvence UNIXu a jeho API
- **přístupová práva, periferní zařízení, systém souborů**
- manipulace s procesy, spouštění programů
- signály
- synchronizace a komunikace procesů
- síťová komunikace
- vlákna, synchronizace vláken
- ??? - bude definováno později, podle toho kolik zbyde času

Uživatelé a skupiny

```
beran:x:1205:106:Martin Beran:/home/beran:/bin/bash
```

význam jednotlivých polí: uživatelské jméno, zahašované heslo (dnes už v `/etc/shadow`), číslo uživatele (UID); superuživatel (root) má UID 0, číslo primární skupiny (GID), plné jméno, domovský adresář, login-shell

```
sisal:*:106:forst,beran
```

význam jednotlivých polí: jméno skupiny, heslo pro přepnutí do skupiny, číslo skupiny (GID), seznam členů skupiny

- informace o uživatelích v souborech `/etc/passwd`, a `/etc/group` jsou zpracovávány různými systémovými programy, např. `login` (přihlášení do systému na základě uživatelského jména a hesla) nebo `su` (změna identity). **Jádro o těchto souborech nic neví, používá pouze numerickou identifikaci uživatele a skupiny.**
- dnes již hesla nejsou z bezpečnostních důvodů přímo v `/etc/passwd`, ale například v `/etc/shadow`, který běžnému uživateli přístupný není, pouze uživatel `root` má právo pro čtení a zápis. Tedy pouze privilegované programy jako `login` nebo `sshd` mohou z tohoto souboru číst nebo zapisovat (program `passwd` běží pod uživatelem `root` díky *setuid* bitu nastaveném na souboru programu, viz poznámky na straně 71). Takto jsou hesla separovaná od veřejně přístupných informací, navíc soubor který hesla obsahuje, je ukládá v zahašované/zašifrované formě, takže nejsou přímo čitelná. Na BSD systémech (např. FreeBSD, Mac OS X) se místo `/etc/shadow` používá soubor `/etc/master.passwd`.
- Soubor `/etc/shadow` je podobně strukturovaný jako `/etc/passwd`. Jeden záznam obsahuje většinou následující položky (Solaris): uživatelské jméno, zahašované heslo (spolu s indikátorem že daný účet je zablokovaný), poslední modifikace hesla, minimum dní požadovaných mezi změnou hesla, maximum dní po které je heslo platné, počet dní po kterých je uživatel varován o expiraci hesla, počet povolených dní neaktivity uživatele, absolutní čas expirace uživatele, počet neúspěšých pokusů o nalogování tohoto uživatele.
- Hesla v `/etc/shadow` jsou uložena takovým způsobem, aby pokud se povede někomu soubor získat, měl ztíženou práci při odhadování hesel. Původní

(cleartext) heslo se prožene jednosměrnou kryptografickou funkcí (která je navíc parametrizovatelná, čímž se výpočetní a prostorová složitost pro útok hrubou silou ještě zvýší) a uloží se do `/etc/shadow` v této formě. Ověřování hesla pak funguje tak že, se na cleartext heslo aplikuje daná funkce s danými parametry a výsledek se porovná s polem hesla v `/etc/shadow`. Pokud jsou stejné, proběhla autentizace úspěšně. Původně navržená funkce přestala s rozvojem procesorů stačit, takže se dnes používají funkce MD5, SHA1, Blowfish a další. Položka hesla v `/etc/shadow` je pak vnitřně strukturovaná pomocí speciálních znaků, takže programy ověřující heslo ví, jakou funkci a s jakými parametry použít. Většina systémů umožňuje globální konfiguraci použitých funkcí a jejich parametrů.

- existují i jiné systémy, které (nejen) pro autentizaci `/etc/passwd` nemusí vůbec používat, například NIS (Network Information Service) nebo LDAP (Lightweight Directory Access Protocol).

- skupina uživatele uvedená v `/etc/passwd` se nazývá **primární**. Tuto skupinovou identifikaci dostanou např. soubory vytvořené procesy uživatele. Další skupiny, ve kterých je uživatel uveden v souboru `/etc/group`, jsou doplňkové (*supplementary*) a rozšiřují přístupová práva uživatele: skupinový přístup je povolen ke všem objektům, jejichž skupinový vlastník je roven buď primární, nebo jedné z doplňkových skupin.

- původně měl v UNIXu každý uživatel vždy aktivní pouze jednu skupinovou identitu. Po nalogování byl ve své primární skupině, pro získání práv jiné skupiny bylo třeba se do ní přepnout příkazem `newgrp` (skupinová obdoba `su`, řídí se obsahem souboru `/etc/group`), který spustil nový shell.

- v novějších UNIXech není třeba pro přístup k souborům měnit primární skupinovou identitu procesu, pokud uživatel patří do potřebné skupiny. Změna identity je nutná, pouze když chceme vytvářet soubory s jinou skupinovou identitou, než je primární skupina uživatele. Lokálně pro určitý adresář toho lze dosáhnout nastavením skupinového vlastníka adresáře na požadovanou skupinu a nastavením bitu SGID v přístupových právech adresáře – to platí pro systémy založené na System V. U BSD stačí změnit požadovanou skupinu u adresáře. Příklad vytvoření takového adresáře na Solarisu (příkaz `chown` musí být proveden pod uživatelem `root`):

```
# mkdir mydir
# chown :lidi mydir
# ls -ald mydir
drwxr-xr-x   2 root     lidi           4 Nov  2 20:01 mydir
# cd mydir/
mydir # touch foo
mydir # ls -ald foo
-rw-r--r--   1 root     root           0 Nov  2 20:01 foo
mydir # chmod g+s .
mydir # ls -ald .
drwxr-sr-x   2 root     lidi           4 Nov  2 20:01 .
mydir # touch bar
mydir # ls -ald bar
-rw-r--r--   1 root     lidi           0 Nov  2 20:01 bar
mydir #
```

- druhá položka v řádcích `/etc/group` obsahuje zakódované skupinové heslo používané příkazem `newgrp`, to se již dnes nepoužívá. Například na FreeBSD je příkaz `newgrp` přístupný už jen superuživateli (kvůli volání `setgroups`).

Name service switch

- dnešní systémy nejsou omezeny na používání `/etc/passwd` a `/etc/groups`
- systém používá *databáze* (passwd, groups, protocols, ...)
- data databází pocházejí ze *zdrojů* (soubory, DNS, NIS, LDAP, ...)
- soubor `nsswitch.conf` definuje jaké databáze používají jaké zdroje
- knihovní funkce toto samozřejmě musí explicitně podporovat
- je možné některé zdroje kombinovat, například uživatel se nejdříve může hledat v `/etc/passwd` a poté v NISu
- poprvé se objevilo v Solarisu, poté převzato dalšími systémy

- systémy mají typicky manuálovou stránku `nsswitch.conf(4)`, kde lze nalézt podrobnosti v závislosti na konkrétním operačním systému, včetně API, pomocí kterého se pracuje s jednotlivými databázemi. Například, s databází `passwd` pracují standardní volání `getpwnam(3)`, `getpwent(3)` a další - není proto potřeba zpracovávat tyto soubory (databáze) sami.
- zde je část skutečného souboru `nsswitch.conf` ze stroje `u-us`:

```
passwd:     files ldap
group:      files ldap

# You must also set up the /etc/resolv.conf file for DNS name
# server lookup.  See resolv.conf(4).
hosts:      files dns

# Note that IPv4 addresses are searched for in all of the
# ipnodes databases before searching the hosts databases.
ipnodes:   files dns

networks:   files
protocols:  files
rpc:        files
ethers:     files
```

Získávání údajů o uživatelích/skupinách

- `struct passwd *`**getpwnam**`(const char *name)`
  vrací strukturu reprezentující uživatele.
- `struct passwd *`**getpwuid**`(uid_t uid)`
  vrací strukturu reprezentující uživatele podle UID.
- `void` **setpwent**`(void)`
- `void` **endpwent**`(void)`
- `struct passwd *`**getpwent**`(void)`
  slouží k procházení položek v databázi uživatelů. **setpwent** nastaví pozici na začátek, **getpwent** získá další položku, **endpwent** dealokuje prostředky použité pro procházení seznamu.

- tyto funkce fungují nezávisle na tom jak z jaké databáze byly získány informace o daném uživateli.
- všechny tyto funkce jsou součástí POSIX 1003.1-2008 (sekce XSH)
- **setpwent** je třeba zavolat před prvním voláním **getpwent**
- analogicky exitují funkce **getgrnam** a **getgrent** které získávají informace o skupinách.
- pro prohledávání a výpis databází lze použít program `getent`. Např. k nalezení záznamu uživatele a skupiny `root`:

```
$ getent passwd root
root:x:0:0:Super-User:/root:/sbin/sh
$ getent group root
root::0:
```

Testování přístupových práv

- uživatel je identifikován číslem uživatele (**UID**) a čísly skupin, do kterých patří (**primary GID**, **supplementary GIDs**).
- tuto identifikaci dědí každý proces daného uživatele.
- soubor S má vlastníka (UID_S) a skupinového vlastníka (GID_S).
- algoritmus testování přístupových práv pro proces $P(UID_P, GID_P, SUPG)$ a soubor $S(UID_S, GID_S)$:

Jestliže	pak *P*roces má vůči *S*ouboru
`if(`UID_P `== 0)`	... všechna práva
`else if(`UID_P `==` UID_S`)`	... práva vlastníka
`else if(`GID_P `==` GID_S `||` $GID_S \in SUPG$`)`	... práva člena skupiny
`else`	... práva ostatních

- procesy superuživatele `root` mohou měnit svoji uživatelskou a skupinovou identitu. Toho využívá např. proces `login`, který běží jako `root` a po zkontrolování jména a hesla spustí shell s uživatelskou identitou (pomocí volání `setuid` – viz další slajdy).
- z algoritmu plyne, že pro `roota` není relevantní nastavení práv (má vždy neomezený přístup). Pokud se shoduje uživatel, nepoužijí se nikdy práva skupiny nebo ostatních, i když povolují více než uživatelská práva. Podobně práva ostatních se nepoužijí, jestliže se shoduje skupinová identita. **Tedy pokud má můj soubor nastaveny práva `---rwxrwx`, nemohu ho číst, zapisovat ani spustit, dokud nastavení práv nezměním.**
- čím dál víc systémů se odklání od klasického modelu, kdy mnoho procesů běželo pod uživatelem s UID 0 a při bezpečnostní chybě v takové aplikaci často útočník získal vládu nad celým systémem a zavádějí modely jako je *least privilege model* v Solarisu nebo *privilege separation* a *systrace* v OpenBSD.
- opakování z prvního ročníku – aby uživatel mohl smazat soubor, musí mít právo zápisu do daného **adresáře**, protože to je ten "soubor", co se mění. **Práva k mazanému souboru nejsou podstatná**; to že vás shell upozorní, že mažete soubor, ke kterému nemáte právo zápisu, je pouze věc toho shellu. Je to logické – pokud si nastavíte soubor jako read-only, shell usuzuje, že ho asi normálně mazat nechcete. Viz příklad pod tímto odstavcem. **Unixové systémy nemají delete-like operaci na soubor**, smazání souboru nastane automaticky tehdy, když na soubor není žádný odkaz z adresářové struktury, a nikdo soubor již nemá otevřený.

```
$ whoami
janp
$ ls -ld janp-dir
drwx------   2 janp  staff  512 Mar 23 12:12 janp-dir/
$ ls -l janp-dir
total 0
-rw-r--r--   1 root  root     0 Mar 23 12:11 root_wuz_here.txt
$ rm janp-dir/root_wuz_here.txt
rm: janp-dir/root_wuz_here.txt: override protection 644 (yes/no)? yes
$ ls janp-dir/root_wuz_here.txt
janp-dir/root_wuz_here.txt: No such file or directory
```

- pokud ale `root` vytvoří v adresáři `janp-dir` svůj podadresář a tam vloží svůj soubor, uživatel `janp` už nemůže adresář `janp-dir` a jeho obsah smazat, protože:
  - podadresář nelze smazat protože není prázdný
  - a daný soubor nelze smazat z toho důvodu, že `janp` není vlastníkem podadresáře.
- Pokud odeberu adresáři read bit, není možné číst jeho obsah, tedy provádět výpis souborů v něm obsažených. Pokud ale znám jméno souboru v adresáři a execute bit je nastaven, mohu soubor přečíst:

```
$ mkdir foo
$ ls -ald foo
drwxr-xr-x  2 vladimirkotal  staff  68 Nov  5 14:37 foo
$ touch foo/bar
$ file foo/bar
foo/bar: empty
$ ls foo
bar
$ chmod u-r foo
$ ls foo
ls: foo: Permission denied
$ file foo/bar
foo/bar: empty
```

- existuje situace, kdy ani právo zápisu (a execute) pro adresář nestačí. To se používá u `tmp` adresářů, do kterých může každý psát, ale není žádoucí situace, kdy by si uživatelé navzájem mazali soubory. K tomu se používá tzv. *sticky bit* (01000). Systémy mají vetšinou manuálovou stránku `sticky`, kde je funkce sticky bitu popsaná. Na výpisu `ls` je označovaný jako `t`:

```
$ ls -ld /tmp
drwxrwxrwt   7 root     root         515 Mar 23 12:22 /tmp
```

Reálné a efektivní UID/GID

- u každého procesu se rozlišuje:
  - **reálné UID** (RUID) – skutečný vlastník procesu
  - **efektivní UID** (EUID) – uživatel, jehož práva proces používá
  - **uschované UID** (saved SUID) – původní efektivní UID
- podobně se rozlišuje reálné, efektivní a uschované GID procesu.
- obvykle platí `RUID==EUID && RGID==EGID`.
- **propůjčování práv** ... spuštění programu s nastaveným SUID (**set user ID**) bitem změní EUID a saved UID procesu na UID vlastníka programu, RUID se nezmění.
- podobně SGID bit ovlivňuje EGID procesu.
- **při kontrole přístupových práv se používají vždy EUID, EGID a supplementary GIDs**

- bity SUID a SGID se používají u programů, které potřebují větší přístupová práva, než má uživatel, jenž je spouští. Příkladem je program `passwd`, který musí aktualizovat soubory `/etc/passwd` a `/etc/shadow`, kde ten první nemůže běžný uživatel měnit a druhý z nich ani číst. Další příklad je program `su`. Ten musí mít právo libovolně změnit uživatelskou a skupinovou identitu, což je privilegium procesů s UID 0.
- SUID a SGID programy by měly být pečlivě naprogramovány, aby dovolily pouze ty operace, pro které jsou určeny, a neumožnily zneužít jejich privilegia pro neoprávněné akce (např. spuštění rootovského shellu). Zkušenost ukazuje, že tyto programy jsou jednou z nejčastějších příčin bezpečnostních problémů UNIXových systémů.
- základním pravidlem pro SUID programy je: **nepište je** pokud to není opravdu nezbytné. Je to typické místo pro generování bezpečnostních chyb protože dobře, tj. bezpečně, napsat složitější SUID program není jednoduché.
- **toto jsou pravidla pro změny:**
  - bežný uživatel nemůže změnit své RUID nebo uschované SUID (vyjímka je při volání `exec`, viz strana 131)
  - proces může vždy změnit své EUID na to z RUID nebo z uschovaného UID. Toto zaručuje, že v SUID programu je možné libovolně měnit EUID mezi tím původním kterým proces získal práva vlastníka a mezi UID skutečného uživatele který daný proces spustil.

– **root může všechno**, a když změní RUID, tak se zároveň změní i uchované UID – nemělo by smysl měnit jen jedno z nich když kterékoli můžete použít pro nastavení EUID.

Identifikace vlastníka procesu

- `uid_t` **`getuid`**`(void)`
  vrací reálné user ID volajícího procesu.
- `uid_t` **`geteuid`**`(void)`
  vrací efektivní user ID volajícího procesu.
- `gid_t` **`getgid`**`(void)`
  vrací reálné group ID volajícího procesu.
- `gid_t` **`getegid`**`(void)`
  vrací efektivní group ID volajícího procesu.
- `int` **`getgroups`**`(int` *`gidsz`*`, gid_t` *`glist`*`[])`
  – do `glist` dá nejvýše `gidsz` supplementary group IDs volajícího procesu a vrátí počet všech GIDs procesu.

- pro reálné UID je volání `getuid`, volání `getruid` neexistuje
- `getgroups`: když `gidsz == 0`, jen vrátí počet skupin. Když `0 < gidsz < #skupin`, vrátí `-1`.
- v UNIXu je mnoho typů jako `uid_t`, `gid_t`, `size_t`, apod. Vesměs jsou to celočíselné typy, často je najdete v `/usr/include/sys/types.h`
- Solaris má příkaz `pcred`, který jednoduše zobrazí informace o identifikaci procesu:

```
$ pcred 5464
5464:   e/r/suid=1993  e/r/sgid=110
        groups: 33541 41331 110
```

Změna vlastníka procesu

- `int setuid(uid_t uid);`
  - v procesu s EUID == 0 nastaví RUID, EUID i saved-SUID na `uid`
  - pro ostatní procesy nastavuje jen EUID, a `uid` musí být buď rovné RUID nebo uschovanému SUID
- `int setgid(gid_t gid);`
  obdoba `setuid`, nastavuje group-IDs procesu.
- `int setgroups(int ngroups, gid_t *gidset)`
  nastavuje supplementary GIDs procesu, může být použito jen superuživatelským procesem.

- o nastavení UID pro proces s EUID 0 viz také poznámky na straně 71.
- co výše uvedené tedy znamená: proces s efektivními právy superuživatele může libovolně měnit identitu. Ostatní procesory mohou pouze střídat svá reálná a efektivní práva.
- program *login* využívá volání `setuid`
- pokud chce process s UID == 0 změnit svou identitu, musí nejprve volat `setgid` a `setgroups`. Teprve pak lze zavolat `setuid`. Při opačném pořadí volání by proces po provedení `setuid` už neměl práva na `setgid` a `setgroups`.
- `setgroups` není uvedeno v UNIX 98 ani UNIX 03.
- RUID/EUID jsou uložené v záznamu tabulky procesů pro příslušný proces a zároveň v tzv. *u-area* (viz například [Bach]). EUID v tabulce procesů se nazývá již zmíněné uschované UID, neboli *saved UID*. Jak již bylo řečeno, uschované UID se používá pro kontrolu, když se proces chce vrátit k EUID, se kterým byl spuštěn (po té, co dočasně nastavil své EUID na UID uživatele, který proces spustil, tj. na RUID).
- pokud tedy jako root vytvoříte SUID program a v něm zavoláte `setuid` pro jakéholi UID mimo 0, již se v programu k EUID==0 nemůžete vrátit (je to logické – představte si situaci, kdy se uživatel loguje do systému). V tom případě byste museli použít volání `seteuid`, které nastavuje pouze EUID.
- příklad: `setuid/screate-file.c`

Systém souborů

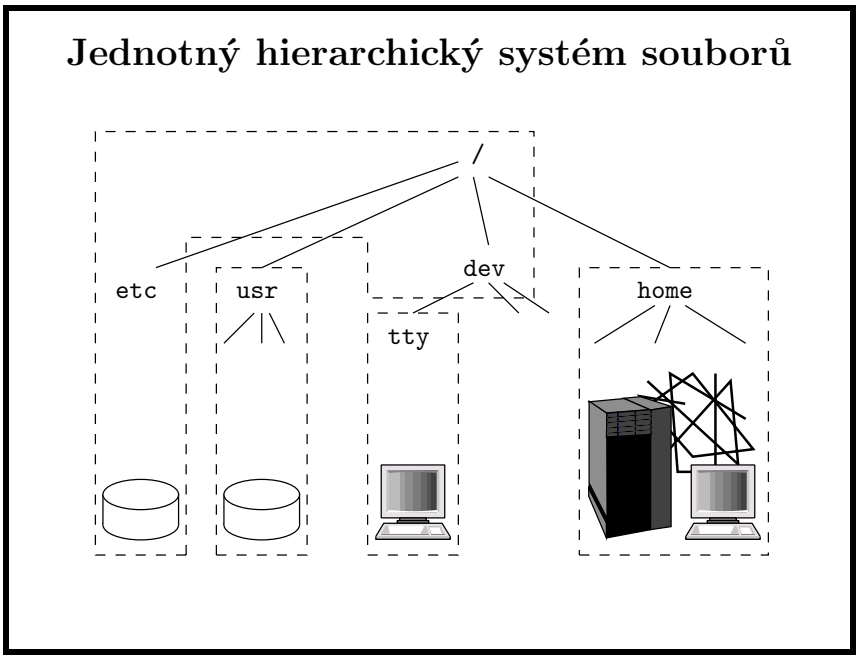
- adresáře tvoří strom, spolu se soubory acyklický graf (na jeden soubor může existovat více odkazů).
- každý adresář navíc obsahuje odkaz na sebe '`.`' (tečka) a na nadřazený adresář '`..`' (dvě tečky).
- pomocí rozhraní systému souborů se přistupuje i k dalším entitám v systému:
  - periferní zařízení
  - pojmenované roury
  - sokety
  - procesy (`/proc`)
  - paměť (`/dev/mem`, `/dev/kmem`)
  - pseudosoubory (`/dev/tty`, `/dev/fd/0`,...)
- z pohledu jádra je každý obyčejný soubor pole bajtů.
- všechny (i síťové) disky jsou zapojeny do jednoho stromu.

- zařízení, soubory v `/proc`, terminály, paměť atd. jsou všechno jeden typ souborů - speciální soubory. Další typy souborů - regulární soubor (hardlink), adresář, pojmenovaná roura, socket, symbolický link.
- nově vytvořený adresář má na sebe 2 odkazy – jeden z nadřazeného adresáře, a jeden sám na sebe, '`.`':

```
$ mkdir test
$ ls -ld test
drwx------   2 janp     staff        512 Mar 23 12:46 test
```

- root může v některých systémech strukturu adresářů zacyklit, ale tím zmate utility pro procházení filesystému; moc se cyklické struktury nepoužívají. Symbolické linky na adresáře fungují všude.
- pojmenované roury (viz strana 94) lze použít i mezi procesy, které nejsou příbuzensky spřízněné. Jinak fungují stejně jako nepojmenované roury.
- zmiňované sokety jsou v doméně UNIX, tj. slouží pro komunikaci v rámci jednoho systému. Sokety z domény INET, přes které probíhá síťová komunikace, se v systému souborů neobjevují. Síťová komunikace začíná na straně 179.
- debuggery používají paměťové obrazy procesů dostupné v `/proc`. Ve většině unix-like systémů obsahuje podstrom `/proc` údaje o jádru systému a běžících procesech ve formě textových souborů.

- dnešní moderní unixy mívají speciální filesystém *devfs*, jehož obsah odráží aktuální konfiguraci systému co se týče připojených zařízení. Tj. např. při připojení USB sticku se v `/dev` objeví příslušné diskové zařízení. Po fyzickém odpojení zařízení odkaz z adresářové struktury opět zmizí.



- svazek (angl. *file system*) je část souborového systému, kterou lze samostatně vytvořit, připojit, zrušit... Každý filesystém může mít jinou vnitřní strukturu (`s5`, `ufs`, `ext2`, `xfs`, atd.) a může být uložen na lokálním disku nebo na jiném počítači a přístupný po síti (`nfs`, `afs`).
- po startu jádra je připojený jen kořenový filesystém, další filesystémy se zapojují do hierarchie na místa adresářů příkazem `mount`. Tento příkaz je možné spustit ručně (uživatel root libovolně, ostatní pouze na některých systémech a s omezeními) nebo automaticky během inicializace systému, kdy se typicky řídí obsahem souboru `/etc/fstab`. Před zastavením systému se filesystémy odpojují příkazem `umount`.
- další možnost je připojení filesystému na žádost při prvním přístupu a jeho odpojení po určité době nečinnosti. Tuto funkci zajišťuje démon *automounter* (`autofs`, `automount`, `amd`).
- UNIX nemá žádné A, B, C, D... disky apod.

Typická skladba adresářů

`/bin`	...	základní systémové příkazy
`/dev`	...	speciální soubory (zařízení, devices)
`/etc`	...	konfigurační adresář
`/lib`	...	základní systémové knihovny
`/tmp`	...	veřejný adresář pro dočasné soubory
`/home`	...	kořen domovských adresářů
`/var/adm`	...	administrativní soubory (ne na BSD)
`/usr/include`	...	hlavičkové soubory pro C
`/usr/local`	...	lokálně instalovaný software
`/usr/man`	...	manuálové stránky
`/var/spool`	...	spool (pošta, tisk,...)

- v `/bin`, `/lib`, `/sbin` jsou příkazy a knihovny potřebné při startu systému, kdy je připojen pouze kořenový filesystém. Ostatní příkazy a knihovny jsou typicky v `/usr/bin`, `/usr/lib` a `/usr/sbin`. `/usr` bývá často samostatný filesystém, čímž jeho obsah není dostupný během startu systému.
- `s` v `sbin` značí „system", ne SUID. Jsou to binárky, u kterých se předpokládá, že je běžný uživatel nebude typicky potřebovat.
- podstrom `/usr` obsahuje soubory, které se nemění při běžném provozu a nejsou závislé na konkrétním počítači. Proto by měl jít sdílet read-only. Na své stanici doma ho ale většinou budete mít read-write.
- `/lib` typicky obsahuje knihovny potřebné pro programy z kořenového svazku. Pokud by všechny knihovny byly v `/usr/lib` a `/usr` byl samostatný svazek, problém s připojením `/usr` by kompletně ochromil celý systém, protože by nešlo spustit nic. Někdy to systémy řeší tak, že kořenový svazek obsahuje sadu základních programů, které jsou staticky slinkované. Například FreeBSD má takové binárky v adresáři `/rescue`, má ale i samostatné adresáře `/lib` a `/usr/lib`.
- v podstromu `/var` jsou data, která se za provozu mění a jsou specifická pro každý počítač.
- různé systémy i instalace jednoho systému se často liší.
- `hier(7)` na FreeBSD a Linuxu popisuje adresářovou hierarchii tohoto systému, Solaris má `filesystem(5)`.

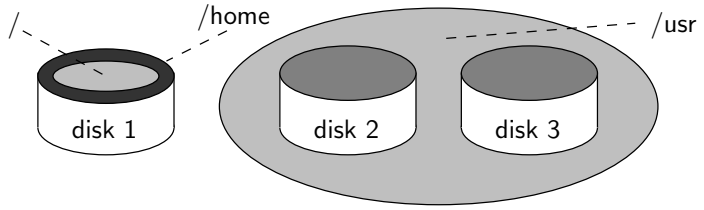
Přístup k periferním zařízením

- adresář `/dev` obsahuje speciální soubory zařízení. Proces otevře speciální soubor systémovým voláním `open()` a dále komunikuje se zařízením pomocí volání `read()`, `write()`, `ioctl()`, apod.
- speciální soubory se dělí na
  - **znakové** ... data se přenáší přímo mezi procesem a ovladačem zařízení, např. sériové porty
  - **blokové** ... data prochází systémovou vyrovnávací pamětí (buffer cache) po blocích pevně dané velikosti, např. disky
- speciální soubor identifikuje zařízení dvěma čísly
  - **hlavní (major) číslo** ... číslo ovladače v jádru
  - **vedlejší (minor) číslo** ... číslo v rámci jednoho ovladače

- vyrovnávací paměti (cache) urychlují periferní operace. Při čtení se data hledají nejprve v bufferu. Teprve když nejsou k dispozici, tak se čtou z disku. Při příštím čtení stejného bloku jsou data v bufferu. Při zápisu se data uloží do bufferu. Na disk je systém přepíše později. Lze si vynutit i okamžitý zápis dat na disk. Dnes již cache typicky není samostatná struktura, vše je zastřešeno modulem vitruální paměti.
- disky v UNIXu jsou obvykle přístupné přes znakové (používané při `mkfs` – vytvoření svazku – a `fsck` – kontrola konzistence) i blokové rozhraní (používané při normálním provozu systému souborů). Některé systémy (FreeBSD) ale už v `/dev` vůbec soubory pro bloková zařízení nemají, pouze znaková.
- dříve musel administrátor systému po změně hardwarové konfigurace upravit obsah adresáře `/dev` skriptem `MAKEDEV` nebo ručně. Dnes (Linux, IRIX, FreeBSD, Solaris, ...) již speciální soubory dynamicky vznikají a zanikají podle toho, jak jádro detekuje přidání nebo odebrání hardwarových komponent (viz *devfs* na straně 74).
- okamžitý zápis na disk lze vynutit přes `O_DIRECT` command ve volání `fcntl`.

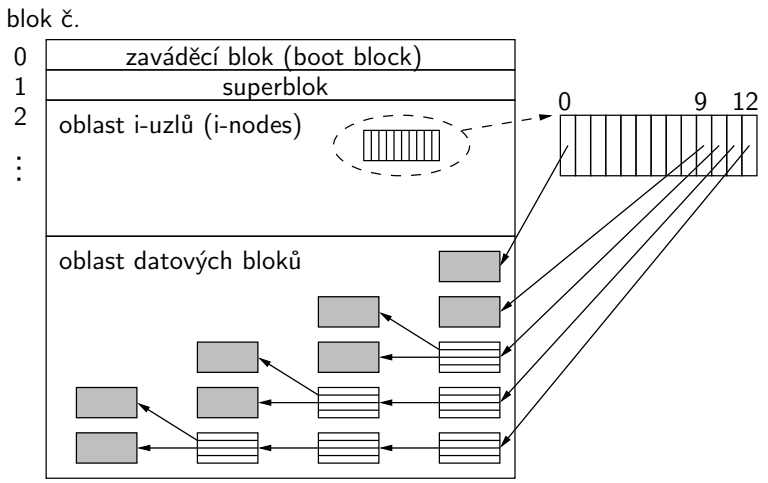
Fyzické uložení systému souborů

- **systém souborů** (svazek, **filesystem**) lze vytvořit na:
  - **oddílu disku** (**partition**) – část disku, na jednom disku může být více oddílů
  - **logickém oddílu** (**logical volume**) – takto lze spojit více oddílů, které mohou být i na několika discích, do jednoho svazku.
- další možnosti: striping, mirroring, RAID



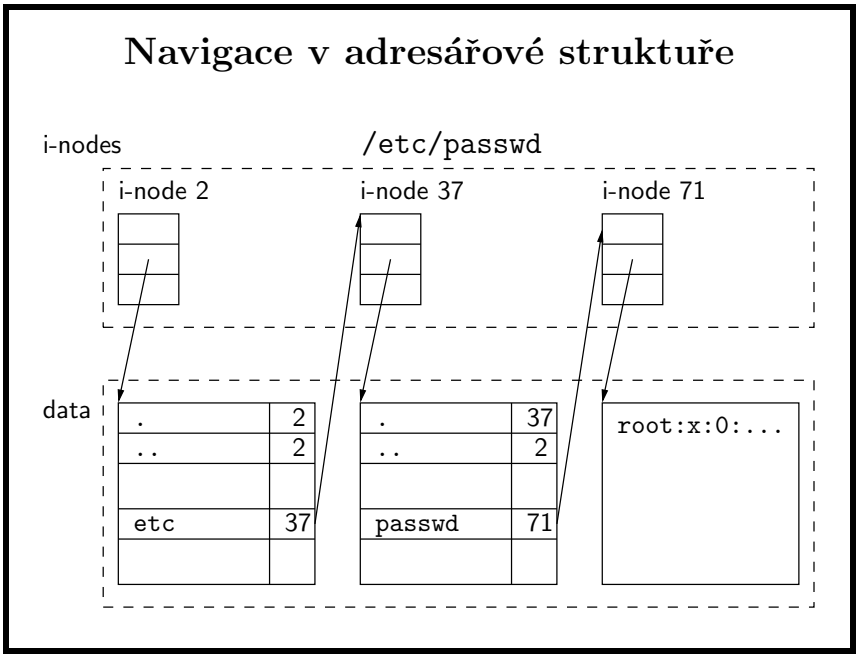
- výraz systém souborů se používá v několika významech:
  - jeden filesystém, tj. to, co vyrobí příkaz `mkfs`
  - celá hierarchie připojených svazků v systému (výstup příkazu `mount`)
  - způsob organizace svazku (tj. typ fs) a tomu odpovídající modul jádra, který s daty manipuluje (UFS2, Ext3, XFS, ...)
- striping je od slova stripe, ne strip; podle toho se také vyslovuje. Znamená, že za sebou následující bloky dat se ukládají paralelně na různé disky a tím se zvyšuje přenosová rychlost.
- mirroring ukládá kopie dat více disků pro případ havárie některého z nich.
- paritní disky: data se ukládají na dva disky, na třetí se ukládá XOR prvních dvou, po havárii libovolného disku jsou všechna data stále čitelná.
- jednotlivé úrovně RAID (Redundant Array of Inexpensive Disks) zahrnují striping, mirroring a využití paritních disků.
- na terminologii je třeba dát velký pozor. Např. to, co se v DOS světě nazývá *partition*, se v BSD nazývá *slice*. Tam jsou pak partitions definovány v rámci jednoho slice a v nich se vytvářejí filesystémy.
- pokud vás to zajímá nebo se s tím setkáváte v praxi, doporučuji vaší pozornosti ZFS, což je filesystém a manažer logických oddílů v jednom. Ze Solarisu se již dostal do FreeBSD od verze 7.0. Můžete okamžitě zapomenout na jednotlivé disky, co je důležité je pouze celková disková kapacita v systému.

Organizace systému souborů s5



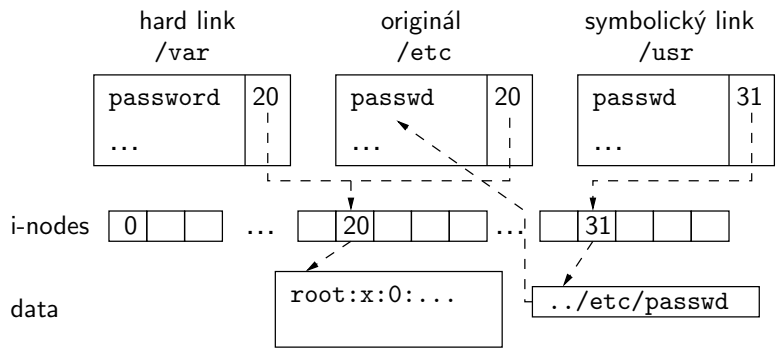
- původní UNIXový systém souborů standardně používaný do verze System V Release 3; v BSD se primárně používal do verze 4.1
- vlastnosti:
  - bloky délky 512, 1024 nebo 2048 bajtů
  - jediný (neduplikovaný) superblok
  - datový prostor pevně rozdělený na oblast *i-uzlů* a oblast *datových bloků*
  - při velikosti bloku 1024 bajtů byla teoretická velikost filesystému přes 16 GB
- *boot block* – pro uložení zavaděče OSu
- *superblok* – základní informace o svazku: počet bloků pro i-uzly, počet bloků svazku, seznam volných bloků (pokračuje ve volných blocích), seznam volných i-uzlů (po vyčerpání se prohledává tabulka i-uzlů), zámky pro seznamy volných bloků a i-uzlů, příznak modifikace používaný pro kontrolu korektního odpojení svazku, čas poslední aktualizace, informace o zařízení
- *i-uzel* – typ souboru, přístupová práva, vlastník, skupina, časy posledního přístupu, modifikace dat a modifikace i-uzlu, počet odkazů na soubor, velikost souboru, 10 odkazů na datové bloky a 3 odkazy na nepřímé bloky. Pozor na to, že čas vytvoření souboru není uložen.
- maximální velikost souboru: 2113674 bloků, tj. přibližně 1 GB při použití bloků velikosti 512 B

- jména souborů – max. 14 znaků (14 + 2 = 16, tedy mocnina dvou a tedy bezproblémové uložení adresářových položek do bloků)
- při použití tohoto filesystému byla výkonnost disků využita jen cca na 2% a rychlost čtení byla v řádu jen několika desítek kilobajtů za sekundu (!!!)
- pro srovnání – MS-DOS 2.0 z roku 1983 podporoval pouze FAT12, počítající s maximální velikostí filesystému 16 MB. Velikost svazku do 2 GB byla umožněna až ve verzi 4.0 (1988); tato verze zároveň zavedla diskové vyrovnávací paměti, tedy to, co UNIX má od svého vzniku v roce 1970.



- když cesta začíná znakem '/', začíná navigace v kořenovém adresáři, jinak začne v pracovním adresáři procesu.
- kořenový adresář má typicky číslo 2. 0 je pro označení prázdného uzlu a 1 byla dříve používaná pro soubor, do kterého se vkládaly vadné bloky, aby je systém už dále nepoužíval.
- cesta ve které je více lomítek za sebou je stále platná, tj. `///a///b///c` je ekvivaletní `/a/b/c`.

Linky



Hard linky lze vytvářet pouze v rámci jednoho (logického) filesystému.

- to co "normálně" vidíte při výpisu adresářů jsou většinou hardlinky, takže rozdělení není na soubory, hardlinky a symbolické linky. Co se týče souborů, jsou **pouze** hardlinky a symlinky.

  hardlink

  - odkaz na stejný i-uzel
  - vlastně druhé jméno souboru
  - není rozdíl mezi originálem a hardlinkem
  - lze vytvářet jen v rámci filesystému
  - nelze vytvářet pro adresáře

  symbolický link (symlink, softlink)

  - pouze odkaz na skutečnou cestu k souboru jiného typu (`ls -l` jej označuje '`l`'), tj. symbolický link je typem odlišný od běžného souboru a jeho data obsahují obyčejný řetězec – jméno cesty, ať již relativní nebo absolutní
  - odlišné chování pro originál a link (např. při mazání)
  - pozor na relativní a absolutní cesty při přesouvání symbolického linku
  - může ukazovat i na adresář nebo na neexistující soubor

- nejjednodušší způsob jak si ověřit, zda dané dva linky ukazují na stejný soubor na disku je použít `-i` přepínač příkazu `ls`, který vám ukáže číslo indexového uzlu.

```
$ ls -i /etc/passwd
172789 /etc/passwd
```

Vylepšení systému souborů

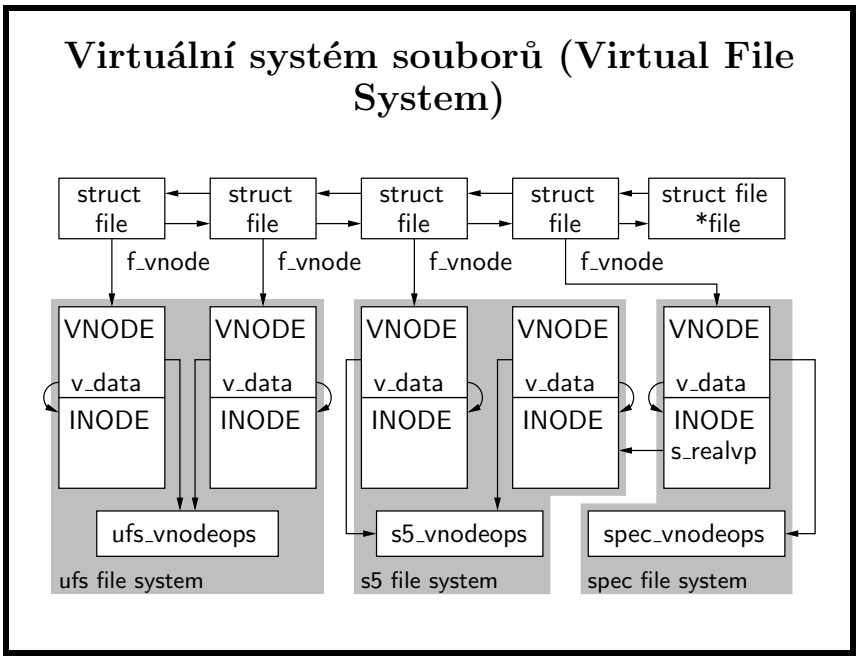
- cíl: snížení fragmentace souborů, omezení pohybu hlav disku umístěním i-uzlů a datových bloků blíž k sobě
- UFS (Unix File System), původně Berkeley FFS (Fast File System)
- členění na skupiny cylindrů, každá skupina obsahuje
  - kopii superbloku
  - řídicí blok skupiny
  - tabulku i-uzlů
  - bitmapy volných i-uzlů a datových bloků
  - datové bloky
- bloky velikosti 4 až 8 kB, menší části do fragmentů bloků
- jména dlouhá 255 znaků

- superblok v každé cylinder skupině posunut tak, aby superbloky nebyly na stejné plotně.
- další typy filesystémů: UFS2, Ext3, ReiserFS, XFS, ZFS aj.
- v http://mff.devnull.cz/pvu/common/docs/filesystems.ps je porovnání osmi různých filesystémů podle různých implementačních kritérií; nezahrnuje v sobě ale vývoj posledních let.
- UFS byl stále 32-bitový, což se odráželo na maximální délce souboru i na maximální velikosti filesystému. (Označení file systému jako 32-bitového znamená, že čísla i-uzlů jsou reprezentována jako 32-bitová. To pak dává teoretický limit pro velikost filesystému.)
- žurnálování (XFS, Ext3, ReiserFS) – snaha o zmenšení nebezpečí ztráty dat v případě havárie, urychlení zotavení po havárii
- ZFS – moderní 128-bitový filesystém vyvinutý v Sun Microsystems, od Solarisu 10, teď již také v FreeBSD 7.

Vývoj ve správě adresářových položek

- maximální délka jména souboru 14 znaků nebyla dostačující
- FFS – délka až 255; každá položka zároveň obsahuje i její délku
- nové filesystémy používají pro vnitřní strukturu adresářů různé varianty B-stromů
  - výrazně zrychluje práci s adresáři obsahující velké množství souborů
  - XFS, JFS, ReiserFS, ...
- UFS2 zavádí zpětně kompatibilní tzv. *dirhash* pro zrychlení přístupu k adresářům s velkým počtem souborů

- *dirhash* pracuje tak, že při prvním přečtení adresáře se vytvoří v paměti hash struktura, následné přístupy do adresáře jsou pak srovnatelné se systémy používající B-stromy. Tímto způsobem je dosaženo zlepšení bez toho, aby se měnila struktura filesystému na disku. Taková vylepšení ale nelze v implementaci filesysému dělat do nekonečna, nakonec je typicky nutná změna on-disk formátu pro adaptaci na výkonostní požadavky (nebo přechod na jiný filesystém).
- malé soubory se často ukládají v i-nodech, čímž se ušetří další přístupy na disk



- *FFS* uvedený ve 4.2BSD byl historicky druhý unixový filesystém. Někteří dodavatelé unixových systémů ho začali preferovat vzhledem k jeho lepšímu výkonu a novým možnostem, jiný zůstávali dále u *s5fs* z důvodu zpětné kompatibility. To dále prohlubovalo problém již tak nedostatečné interoperability mezi různými unixovými systémy. Některým aplikacím navíc plně nevyhovoval ani jeden z těchto filesystémů. Postupně se také objevovala potřeba pracovat s ne-unixovými filesystémy, např. *FAT*. A s rostoucí popularitou počítačových sítí se zvyšovala poptávka po sdílení souborů mezi jednotlivými systémy, což mělo za následek vznik distribuovaných filesystémů – např. *NFS* (Network File System).

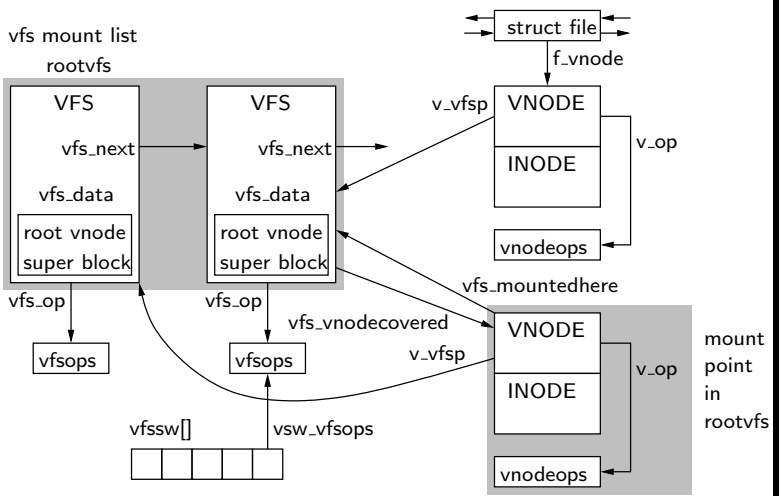
- vzhledem k výše popsané situaci bylo jen otázkou času, kdy dojde k fundamentálním změnám v infrastruktuře systému souborů, aby současně podporoval více typů filesystémů. Vzniklo několik různých implementací od různých výrobců, až se nakonec de facto standardem stala *VFS/vnode* architektura od firmy Sun Microsystems. V dnešní době prakticky všechny unixové a unix-like systémy podporují VFS, i když často se vzájemně nekompatibilními úpravami. VFS se poprvé objevilo v roce 1985 v Solarisu 2.0; brzy bylo převzato BSD – FFS s podporou VFS se začal nazývat UFS.

- hlavní myšlenka: každému otevřenému souboru v systému přísluší struktura `file`; to by vlastně byl jeden slot v námi již známé systémové tabulce otevřených souborů. Ta ukazuje na *vnode* (*virtual node*). Vnode obsahuje část nezávislou na konkrétním systému souborů a část závislou, což může být například struktura *inode*. Ta je specifická pro každý typ souborového systému. **Každý typ filesystému implementuje pevně danou**

sadu funkcí pro jednotlivé operace nad soubory, na kterou se odkazují všechny virtuální uzly odpovídající danému typu filesystému. **Tato sada funkcí tedy definuje vnode interface.** Když tedy zavoláte například `open`, jádro zavolá příslušnou implementaci v závislosti na typu filesystému (např. z modulu *ext2fs*). Implementačně závislá část struktury vnode je přístupná pouze z funkcí příslušného typu filesystému; jádro do ní tedy „nevidí" přímo. Jak uvidíte na dalším slajdu, existuje ještě jedna sada funkcí, která se týka práce s filesystémy jako takovými. **Ta pak definuje VFS interface.** Tyto **dvě sady společně** tvoří vnode/VFS rozhraní, kterému se běžně říká jen VFS.

- (nebudu zkoušet) – u speciálních souborů je situace složitější, v SVR4 struktura `file` ukazuje na *snode* (*shadow-special-vnode*), který definuje operace se zařízením (pomocí souborového systému *spec*) a prostřednictvím ukazatele `s_realvp` se odkazuje na reálný vnode pro operace se speciálním souborem; ten je potřeba například pro kontrolu práv přístupu. Každému zařízení může odpovídat více speciálních souborů, a tedy více snodes a příslušných reálných vnodes. Všechny takové snodes pro jedno zařízení mají ukazatel `s_commonvp` na jeden společný snode, to ale není na obrázku zachyceno. Při otevření speciálního souboru se hledá v hash tabulce snodes otevřených zařízení položka odpovídající speciálnímu souboru (podle major a minor čísla zařízení). Když snode není nalezen, vytvoří se nový. Tento snode se pak používá při operacích se zařízením. Více viz například [Vahalia].

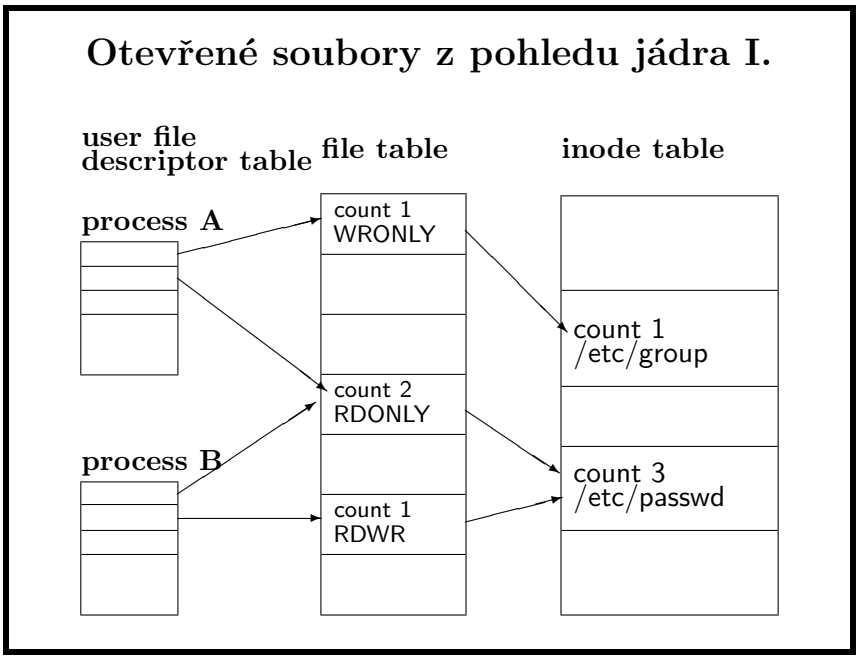
Hierarchie souborových systémů



- struktura `vfs` obsahuje implementačně nezávislé informace o filesystému, nezávislé na konkrétním typu filesystému, tedy podobně jako vnode funguje

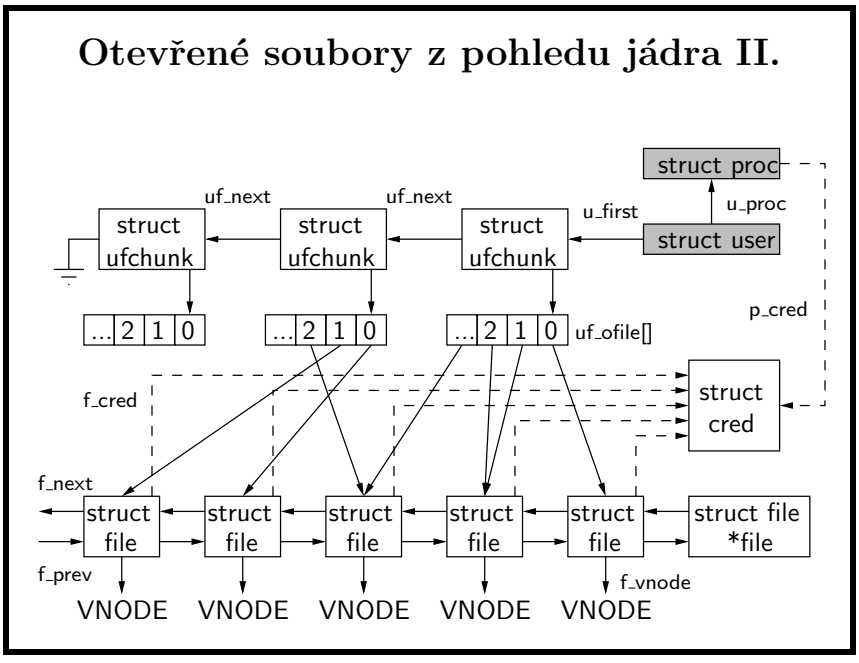
pro soubory. Tato struktura reprezentuje jeden konkrétní fyzický filesystém, aktuálně přimontovaný do hierarchie souborů. V tomto vázaném seznamu tedy může být více struktur stejného typu souborového systému.

- `rootvfs` – odkaz na root file system
- `vfsops` – tabulka funkcí pro konkrétní typ systému souborů
- `vfssw[]` – pole odkazů na tabulky `vfsops` pro všechny systémem podporované typy filesystémů, z tabulky se vybírá při připojování svazku podle typu filesystému zadaného při volání `mount`
- `v_vfsp` – odkaz z vnode na filesystém (strukturu `vfs`), na kterém leží soubor reprezentovaný pomocí vnode
- `v_vfsmountedhere` – pouze ve vnode, který reprezentuje mount point (adresář, na kterém je připojen kořen jiného filesystému); odkazuje na strukturu `vfs` reprezentující připojený filesystém
- `v_vnodecovered` – odkaz na vnode adresáře, na kterém je filesystém připojen



- toto je nejjednodušší pohled na tabulky v jádře které se týkají souborů, je to převzato z [Bach]; dnes to je o něco složitější, myšlenka je ale pořád stejná. Realitě více se podobající obrázek je na příštím slajdu.
- každý proces má svoji tabulku souborových deskriptorů (*user file descriptor table*)

- z této tabulky je odkazovnáno na systémovou tabulku otevřených souborů systému (*file table*; ano, tato tabulka je pouze jedna). Zde je mód otevření souboru a také **aktuální pozice v souboru.**
- z tabulky otevřených souborů je odkazováno do tabulky načtených inodů v paměti. Dnes jsou to tzv. *vnodes – virtual nodes*, ale to v této chvíli není relevantní.
- tabulka otevřených souborů systému, která vlastně vytváří o jednu úroveň odkazů navíc, je zde proto, aby různé procesy mohly sdílet stejnou aktuální pozici v souboru.
- při otevření souboru pomocí volání `open` se vždy alokuje nový slot v tabulce deskriptorů a také v systémové tabulce otevřených souborů (to je důležité!). Sdílení se pak v rámci jednoho procesu dosáhne duplikací deskriptorů, kdy více deskriptorů sdílí stejný slot v tabulce otevřených souborů systému nebo v případě různých procesů pak pomocí vytvoření nového procesu pomocí volání `fork`(), viz strana 138.



- struktury `proc` a `user` vytváří jádro pro každý proces a drží v nich služební informace o procesu.
- struktura `ufchunk` obsahuje `NFPCHUNK` (obvykle 24) *deskriptorů souborů*, po zaplnění se alokuje další `ufchunk`.
- struktura `file` (*otevření souboru*) obsahuje mód souboru (otevřen pro čtení, zápis, atd.), počet deskriptorů, které se na ni odkazují, ukazatel na `vnode` a

pozici v souboru. Jedno otevření souboru může být sdíleno více deskriptory, jestliže byl původní deskriptor zkopírován, např. voláním `fork()` nebo `dup()`.

- struktura `cred` obsahuje uživatelskou a skupinovou identitu procesu, který otevřel soubor.
- jeden vnode odpovídající jednomu souboru může být sdílen několika strukturami file, pokud byl daný soubor vícekrát otevřen.
- ne všechny vnodes jsou asociovány s tabulkou otevřených souborů. Např. při spuštění programu je potřeba přistupovat do spustitelného souboru a proto se alokuje vnode.

Oprava konzistence souborového systému

- pokud není filesystém před zastavením systému korektně odpojen, mohou být data v nekonzistentním stavu.
- ke kontrole a opravě svazku slouží příkaz `fsck`. Postupně testuje možné nekonzistence:
  - vícenásobné odkazy na stejný blok
  - odkazy na bloky mimo rozsah datové oblasti systému souborů
  - špatný počet odkazů na i-uzly
  - nesprávná velikost souborů a adresářů
  - neplatný formát i-uzlů
  - bloky které nejsou obsazené ani volné
  - chybný obsah adresářů
  - neplatný obsah superbloku
- operace `fsck` je časově náročná.
- žurnálové (např. XFS v IRIXu, Ext3 v Linuxu) a transakční (ZFS) systémy souborů nepotřebují `fsck`.

- data se přepisují na disky z vyrovnávacích pamětí se zpožděním. Uložení všech vyrovnávacích pamětí lze vynutit systémovým voláním `sync()`. Periodicky vyrovnávací paměti ukládá zvláštní systémový proces (démon).
- `fsck` kontroluje pouze metadata. pokud došlo k poruše dat samotných, nepozná to, natož aby s tím něco mohl udělat.
- zde je ukázka `fsck` na **odpojený** filesystém:

```
toor@shewolf:~# fsck /dev/ad0a
** /dev/ad0a
** Last Mounted on /mnt/flashcard
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
```

```
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
** Phase 5 - Check Cyl groups
24 files, 8848 used, 12951 free (7 frags, 1618 blocks, 0.0% fragmentation)
```

Další způsoby zajištění konzistence filesystému

- tradiční UFS – synchronní zápis metadat
  - aplikace vytvářející nový soubor čeká na inicializaci inode na disku; tyto operace pracují rychlostí disku a ne rychlostí CPU
  - asynchronní zápis ale častěji způsobí nekontistenci metadat
- řešení problémů s nekonzistencí metadat na disku:
  - *journalling* – skupina na sobě závislých operací se nejdříve atomicky uloží do žurnálu; při problémech se pak žurnál může "přehrát"
  - bloky metadat se nejdříve zapíší do non-volatile paměti
  - *soft-updates* – sleduje závislosti mezi ukazately na diskové struktury a zapisuje data na disk metodou *write-back* tak, že data na disku jsou vždy konzistentní
  - *ZFS* je nový filesystém v Solarisu, který používá *copy-on-write* transakční model

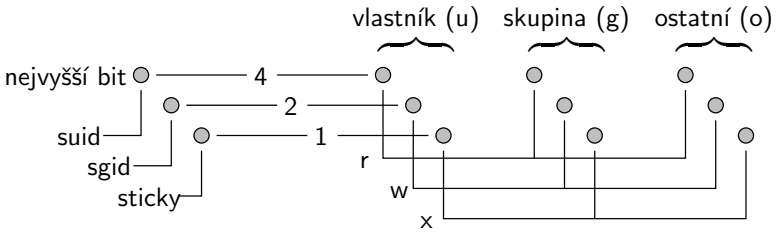
- filesystem *metadata* = inodes, directories, free block maps
- *ext2* dokonce defaultně používá asynchronní zápis metadat a je při použití synchronního zápisu výrazně pomalejší než UFS
- závislé operace jsou například smazání položky z adresáře a smazání diskového inode. Pokud by se stalo, že se nejdříve smaže diskový inode a pak teprve položka v adresáři, při výpadku mezi těmito dvěmi operacemi vniká nekonzistence – link ukazuje na diskový soubor, který neexistuje. Není problém se tomuto vyhnout při synchronním zápisu metadat (víme kdy a co zapisujeme, určujeme tedy pořadí zápisu), ale při metodě write-back je již nutné řešit závislosti jednotlivých bloků na sebe, protože při klasické synchronizaci vyrovnávacích pamětí na disk jádro nezajímá, který blok se zapíše dřív a který později.
- často jsou bloky na sobě závislé ve smyčce. Soft updates dokáže takovou smyčku rozbít tím, že provede *roll-back* a po zápisu pak *roll-forward*
- výkon soft updates je srovnatelný výkonu UFS filesystému s asynchronním zápisem metadat
- teoreticky soft updates zaručují, že po rebootu není potřeba použít `fsck`, tj. že filesystém bude v takovém stavu, že je možné nabootovat. Je však nutné

použít tzv. *background fsck* pro opravy nezávažných chyb – to je považováno stále za velkou nevýhodu soft updates zvláště s tím, jak rostou velikosti běžně používaných disků. Takovou chybou, která nebrání nabootování, ale je nutné ji odstranit je například blok, který je označen jako použitý, ale žádný soubor ho nepoužívá.

- soft updates nejsou vždy doporučovány pro root filesystém. Problém je to, že ztráta metadat na root filesystému (viz 30-ti sekundová perioda zápisu) může být výrazně větší hrozbou zde než na `/usr`, `/home` atd. Další nevýhodou může být i to, že u soft updates mi smazání velkého souboru hned neuvolní místo.

- příklad: na bezpečnou operaci rename potřebuju při synchronním zápisu metadat 4 zápisy – zvýšení počtu odkazů v inode, vytvoření nové adresářové položky, smazání staré, a opětné snížení počtu odkazů v inode. Kdykoli by systém spadnul, nenastane nebezpečná situace. Například 2 odkazy z adresářů na inode s referenčním počtem 1 je problém, protože po zrušení jednoho odkazu to bude vypadat, že soubor je stále na disku, když už byla jeho data dávno smazána. Není težké si asi představit, co by to mohlo znamenat v případě, že soubor obsahoval opravdu důležitá data – například zálohu. Opačná situace, tj. jeden odkaz na inode s referencí 2 sice také není správná situace, ale neohrožuje to možnost filesystém namontovat a normálně používat. V nejhorším se stane, že soubor vypadá že již na disku není a přitom stále existuje. U soft updates operace rename vytvoří kružnici, protože nejdříve je potřeba zapsat zvýšení počtu referencí, pak adresářové bloky a poté snížení referencí. A protože zvýšení/snížení se týká stejného inode, tak je při zápisu třeba udělat roll-back na (řekněme) 2, zapsat inode na disk, zapsat bloky adresárů a pak roll-forward na počet referencí 1. Při této akci je nad inodem držen zámek, takže nikdo nenačte starší data. Je jednoduché ukázat, že nelze zapsat žádný z těch tří bloků v kružnici tak, jak to je na konci operace rename, že je opravdu nutný ten roll-back – mohli bychom uvažovat, že inode se vlastně nezměnil a není třeba řešit zda se může/nemůže zapsat; zápis nového adresářového odkazu bez zvýšení počtu odkazů v inodu by nás totiž mohl dostat přesně do situace, která je popsána o pár řádků výše.

Přístupová práva



- **SGID** pro soubor bez práva spuštění pro skupinu v System V: kontrola zámků při každém přístupu (**mandatory locking**)
- **sticky bit** pro adresáře: právo mazat a přejmenovávat soubory mají jen vlastníci souborů
- **SGID** pro adresář: nové soubory budou mít stejnou skupinu jako adresář (System V; u BSD systémů to funguje jinak, viz poznámky)

- SGID pro adresáře u BSD systémů způsobí, že soubory a podadresáře vytvořené v tomto adresáři budou mít stejného majitele jako je majitel daného adresáře. Nutným předpokladem je dále to, že daný UFS filesystém musí být namontován s suiddir příznakem a v jádru je option SUIDDIR (a to není default). Navíc to nefunguje pro roota. Tato možnost existuje kvůli Sambě a Nettalku.
- sticky bit pro adresáře: přejmenovat nebo smazat soubor může jen jeho vlastník (v některých implementacích stačí i právo zápisu do souboru), nestačí právo zápisu do adresáře. Toto nastavení se používá pro veřejné adresáře (např. `/tmp`).
- původně měl sticky bit význam i pro spustitelné soubory: program s nastaveným sticky bitem zůstal po ukončení v paměti a jeho opětovné spuštění bylo rychlejší. Dnes se sticky bit v tomto významu už nepoužívá.
- některé filesystémy (XFS, AFS, UFS2, ZFS) mají tzv. access control lists (ACLs), které dovolují jemnější přidělování práv jednotlivým uživatelům a skupinám.

API pro soubory

- před použitím musí proces každý soubor nejprve otevřít voláním `open()` nebo `creat()`.
- otevřené soubory jsou dostupné přes **deskriptory souborů** (file descriptors), číslované od 0, více deskriptorů může sdílet jedno **otevření souboru** (mód čtení/zápis, ukazatel pozice)
- standardní deskriptory:
  - 0 ... standardní vstup (jen pro čtení)
  - 1 ... standardní výstup (jen pro zápis)
  - 2 ... chybový výstup (jen pro zápis)
- čtení a zápis z/do souboru: `read()`, `write()`
- změna pozice: `lseek()`, zavření: `close()`, informace: `stat()`, řídicí funkce: `fcntl()`, práva: `chmod()`, ...

- každá funkce, která alokuje deskriptory (nejen `open` a `creat`, ale např. i `pipe`, `dup`, `socket`) alokuje vždy volné deskriptory s nejnižším číslem.
- proces dědí otevřené soubory od rodiče, tyto soubory nemusí znovu otvírat. Obvykle (ale ne vždy) proces dostane otevřené alespoň deskriptory 0, 1 a 2.
- funkce ze souboru `stdio.h` (např. `fopen`, `fprintf`, `fscanf`) a odkazy na soubory pomocí ukazatele na `FILE` jsou definovány ve standardní knihovně a pro svojí činnost používají volání jádra (např. `open`, `write`, `read`). My se nebudeme knihovnou `stdio` zabývat.

Otevření souboru: `open()`

`int` **`open`**`(const char *`*`path`*`, int `*`oflag`*`, ... );`

- otevře soubor daný jménem (cestou) `path`, vrátí číslo jeho deskriptoru (použije první volné), `oflag` je OR-kombinace příznaků
  - `O_RDONLY`/`O_WRONLY`/`O_RDWR` ... otevřít pouze pro čtení / pouze pro zápis / pro čtení i zápis
  - `O_APPEND` ... připojování na konec
  - `O_CREAT` ... vytvořit, když neexistuje
  - `O_EXCL` ... chyba, když existuje (použití s `O_CREATE`)
  - `O_TRUNC` ... zrušit předchozí obsah (právo zápisu nutné)
  - ...
- při `O_CREAT` definuje třetí parametr `mode` přístupová práva

- při použití `O_CREAT` se `mode` ještě modifikuje podle nastavení aktuální masky, které se mění voláním `umask`. Defaultní hodnota je typicky `022`. Doporučuji se zamyslet, zda ji ve vašem shell profile skriptu nenastavit na `077`. To ale nikdy nedělejte pro roota, jinak skončíte s podivně se chovajícím systémem – nainstalovaný software nebude moct spustit pod běžným uživatelem, to co dříve fungovalo fungovat přestane apod.
- `mode` nemá defaultní hodnotu, tj. vezme se to, co je na zásobníku i když tento parametr není přítomen! Příznaky i mód jsou uloženy v systémové tabulce otevřených souborů.
- pokud se znovu použije dříve využívaná položka v tabulce deskriptorů nebo v tabulce otevřených souborů, vše potřebné se vynuluje (pozice v souboru, flagy deskriptoru, ...)
- existují ještě další nastavitelné příznaky:
  - `O_SYNC` (`O_DSYNC`, `O_RSYNC` – není na BSD) ... operace se souborem skončí až po fyzickém uložení dat (synchronized I/O)
  - `O_NOCTTY` ... při otvírání terminálu procesem, který nemá řídicí terminál, se tento terminál nestane řídícím terminálem procesu
  - `O_NONBLOCK` ... pokud nelze čtení nebo zápis provést okamžitě, volání `read`/`write` skončí s chybou místo zablokování procesu
- v přístupových právech se vynulují ty bity, které jsou nastavené pomocí `umask`.

- pro čtení a zápis nelze použít `O_RDONLY | O_WRONLY`, protože implementace použily 0 pro read-only flag. Norma proto definuje, že aplikace musí použít právě jeden z těchto tří flagů.
- je možné otevřít a zároveň vytvořit soubor pro zápis tak, že jeho mód zápis nedovoluje. Při příštím otevření souboru se ale již tento mód uplatní při kontrole přístupu a pro zápis jej otevřít nepůjde. Pro `O_TRUNC` také platí, že musíte mít pro daný soubor právo zápisu.
- Chování `O_EXCL` bez současného použití `O_CREAT` není definováno. Pro zamykání souborů se používá `fcntl`, viz strana 104.

Vytvoření souboru

`int` **creat**`(const char *`*path*`, mode_t` *mode*`);`

- `open()` s příznakem `O_CREAT` vytvoří soubor, pokud ještě neexistuje. V zadané hodnotě přístupových práv se vynulují bity, které byly nastaveny pomocí funkce
  `mode_t` **umask**`(mode_t` *cmask*`);`
- funkce je ekvivalentní volání
  `open(path, O_WRONLY|O_CREAT|O_TRUNC, mode);`

`int` **mknod**`(const char *`*path*`, mode_t` *mode*`, dev_t` *dev*`);`

- vytvoří speciální soubor zařízení.

`int` **mkfifo**`(const char *`*path*`, mode_t` *mode*`);`

- vytvoří pojmenovanou rouru.

- Volání vrací nejnižší deskriptor, který v dané chvíli nebyl otevřený pro proces
- `open` dokáže otevřít regulární soubor, zařízení i pojmenovanou rouru, ale vytvořit dokáže jen regulární soubor; pro ostatní typy souborů je nutné použít zde uvedená speciální volání.
- Test pomocí příznaku `O_EXCL` na to, zda soubor existuje, a jeho případné vytvoření je atomická operace. Toho se využívá při používání lock souborů. Ze slajdu je vidět, že tento test je možné provést jen pomocí volání `open`, ne `creat`.
- Speciální soubory může vytvářet pouze root, protože se pomocí nich definují přístupová práva k periferním zařízením.
- Povolené hodnoty pro `mode` je možné nalézt většinou v manuálové stránkce pro `chmod(2)`, a určitě je naleznete i v hlavičkovém souboru `stat.h`, kde musí být podle normy definované.

Čtení a zápis souborů: `read()`, `write()`

`ssize_t` **`read`**`(int` *`fildes`*`, void *`*`buf`*`, size_t` *`nbyte`*`);`

- z otevřeného souboru s číslem deskriptoru `fildes` přečte od aktuální pozice max. `nbyte` bajtů dat a uloží je od adresy `buf`.
- vrací počet skutečně přečtených bajtů (`<= nbyte`), 0 znamená konec souboru.

`ssize_t` **`write`**`(int` *`fildes`*`, const void *`*`buf`*`, size_t` *`nbyte`*`);`

- do otevřeného souboru s číslem deskriptoru `fildes` zapíše na aktuální pozici max. `nbyte` bajtů dat uložených od adresy `buf`.
- vrací velikost skutečně zapsaných dat (`<= nbyte`).

- pro UNIX je každý soubor posloupnost bajtů bez další vnitřní struktury.
- **chování `read` a `write` závisí na typu souboru** (regulární, zařízení, roura, soket) a na tom, zda je soubor v blokujícím nebo neblokujícím módu (flag `O_NONBLOCK` při otevření souboru, viz strana 93).
- pro obě volání existuje několik zásadních rohových případů a následující informace jsou výňatkem z manuálových stránek. Pokud si nejste jisti, doporučuji se podívat přímo do normy (POSIX 1003.1, sekce XSH, část System Interfaces).
- volání `read` vrátí nenulový počet bajtů menší než `nbyte`, pokud v souboru zbývá méně než `nbyte` bajtů, nebo volání bylo přerušeno signálem, nebo soubor je roura, zařízení či soket a aktuálně je k dispozici méně než `nbyte` bajtů. Při neexistenci dat se blokující `read` zablokuje, dokud se nějaká data neobjeví, neblokující `read` vrátí `-1` a nastaví `errno` na `EAGAIN`.
- volání `write` vrátí nenulový počet bajtů menší než `nbyte`, jestliže se do souboru nevejde víc dat (např. zaplněný disk), zápis je přerušen signálem nebo je nastaveno `O_NONBLOCK` a do roury, soketu nebo zařízení se vejde pouze část zapisovaných dat. Bez `O_NONBLOCK` se čeká, dokud se nepodaří zapsat vše. Pokud nelze aktuálně zapsat nic, blokující `write` se zablokuje, dokud není možné zapisovat, neblokující `write` vrátí `-1` a nastaví `errno` na `EAGAIN`.
- **důležité výjimky vzhledem k rourám** jsou uvedeny na straně 98.
- když `read` nebo `write` vrátí méně než `nbyte` z důvodu chyby, opakované volání téže funkce vrátí `-1` a nastaví kód chyby v `errno`.

- přerušení `read`, `write` signálem dřív, než se podaří přečíst, resp. zapsat aspoň jeden bajt, způsobí návrat s hodnotou `-1` a nastavení `errno` na `EINTR`. Pozor na to, že zde je rozdíl proti situaci, kdy před doručením signálu podaří přečíst nebo zapsat alespoň jeden bajt (viz výše).
- příznak otevření souboru `O_APPEND` zajistí atomický zápis na konec souboru (pouze na lokálním disku), tj. **každý** zápis se provede na konec souboru (tzv. *append-only* soubor).

Uzavření souboru: `close()`

`int` **close**`(int` *`fildes`*`);`

- uvolní deskriptor `fildes`, pokud to byl poslední deskriptor, který odkazoval na otevření souboru, zavře soubor a uvolní záznam o otevření souboru.
- když je počet odkazů na soubor 0, jádro uvolní data souboru. Tedy i po zrušení všech odkazů (jmen) mohou se souborem pracovat procesy, které ho mají otevřený. Soubor se smaže, až když ho zavře poslední proces.
- když se zavře poslední deskriptor roury, všechna zbývající data v rouře se zruší.
- při skončení procesu se automaticky provede `close()` na všechny deskriptory.

- Pokud proces potřebuje dočasný soubor, může ho vytvořit, ihned smazat a pak s ním pracovat přes existující deskriptor (tento deskriptor lze předat synovským procesům). Když je takový soubor zavřen všemi procesy, jádro smaže jeho data z disku.
- I operace `close` může selhat. Např. některé filesystémy zapisují data na disk až v okamžiku zavření souboru, když zápis skončí chybou, `close` vrátí `-1`.
- Chbějící zavírání deskriptorů při ukončení práce s nimi vede k situaci, které se říká *file descriptor leak*. Není to úplně přesně to, jak se typicky chápe *memory leak* (pamět, na kterou jsme ztratili ukazatel), ale následky jsou podobné.
- Příklad na jednoduchý `cat(1)` program: `read/cat.c`

Příklad: kopírování souborů

```
#include <sys/types.h>
#include <fcntl.h>
int main(int argc, char *argv[])
{
    char buf[4096];
    int inf, outf;
    ssize_t ilen;

    inf = open(argv[1], O_RDONLY);
    outf = creat(argv[2], 0666);
    while ((ilen = read(inf, buf, 4096)) > 0)
            write(outf, buf, ilen);

    close(inf); close(outf);
    return (0);
}
```

- Tento příklad je kompletní! Stačí provést cut-and-paste, přeložit a spustit. Nutno poznamenat, že pro účely použití pouze jednoho slajdu se netestují žádné chyby, včetně ignorování toho, že uživatel může program spustit bez parametrů, což způsobí pokus o přístup do paměti na místo, které neobsahuje očekávaná data. To pak na některých systémech může způsobit core dump.
- Je neefektivní číst a zapisovat soubor po jednotlivých bajtech, protože každé systémové volání má jistý overhead, nezávislý na velikosti zpracovávaného bloku. Lepší je proto najednou zpracovávat rozumně velké bloky, například 8-64KB. Přechozí příklad `read/cat.c` má přepínač `-b` pro nastavení velikosti bloku pro čtení; zkuste s (a bez) `-b 1` a změřte pomocí `time(1)`. Uvidíte velký rozdíl.
- Pokud potřebujete pracovat s malými částmi/buffery, je lepší použít stream orintované knihovní funkce `fopen`, `fread`, ... které data vnitřně bufferují.
- Všimněte si, že vždy zapisujeme jen tolik bajtů, kolik jsme jich načetli.

Práce s pojmenovanou rourou

- nemusí být možné vytvořit FIFO na síťovém filesystému (NFS, AFS)
- je nutné znát sémantiku otevírání pojmenované roury:
  - otevření roury pouze pro čtení se zablokuje do té doby, dokud se neobjeví zapisovatel (pokud již neexistuje)
  - otevření roury pouze pro zápis se zablokuje do té doby, dokud se neobjeví čtenář (pokud již neexistuje)
  - toto chování je možné ovlivnit flagem `O_NONBLOCK`
- sémantika čtení a zápisu je ještě o něco složitější, věnujte velkou pozornost poznámkám pod tímto slajdem
  - a je to stejné jako u nepojmenované roury

- Co se týče vytvoření pojmenované roury, mluvíme zde o volání `mkfifo` ze strany 94. Použití nepojmenované roury je popsané na straně 137.
- Čtenář je proces, který otevře soubor (i) pro čtení, zapisovatel je proces, který otevře soubor (i) pro zápis.
- Je možné otevřít rouru pro zápis i čtení stejným procesem najednou, aniž by předtím existoval čtenář nebo zapisovatel. Proces pak může do roury zapisovat a z opačného konce číst, co do roury napsal.
- Pokud rouru nemá žádný proces otevřenou pro zápis, pro okamžité vrácení deskriptoru pouze pro čtení je nutné **otevřít** rouru s flagem `O_NONBLOCK`, proces se jinak na rouře zablokuje čekáním na zapisovatele. Pozor ale na to, že pokus o **čtení** z roury bez zapisovatele okamžitě vrátí 0 jako indikaci konce souboru; **proces se nezablokuje čekáním na zapisovatele**, bez ohledu na to zda byl soubor otevřen v blokovacím či neblokovacím módu. Při **zápisu** do roury bez čtenáře (tj. zapisovatel otevřel rouru ještě v době, kdy čtenář existoval, viz následující odstavec) pošle kernel procesu signál `SIGPIPE` ("broken pipe"):

```
bash$ dd if=/dev/zero | date
Sun Mar  2 01:03:38 CET 2008
bash$ echo ${PIPESTATUS[0]}
141
bash$ kill -l 141
PIPE
```

- v případě otevírání pouze pro zápis s `O_NONBLOCK` bez existujícího čtenáře se vrátí chyba a `errno` se nastaví na `ENXIO`. Tato asymetrie je snahou, aby v rouře nebyla data, která nebudou v krátké době přečtena – systém nemá způsob, jak uschovávat data v rouře bez časového omezení. Bez `O_NONBLOCK` flagu se proces zablokuje čekáním na čtenáře. Asymetrií je míněno to, že systému nevadí čtenáři bez zapisovatelů, ale nechce mít zapisovatele bez čtenářů. I tato situace se může stát, viz předchozí odstavec, ale tam to jinak řešit nelze.
- z uvedeného tedy vyplývá, že pokud chcete vytvořit proces, který čeká na pojmenované rouře a vykonává požadavky od různých procesů-zapisovatelů (od každého jeden), musíte ji otevřít s flagem `O_RDWR` i když do roury nehodláte zapisovat; jinak po prvním zablokování v `open` při čekání na otevření roury zapisovatelem by další volání `read` pro akceptování dalšího požadavku typicky vrátilo 0, pokud by náhodou roura nebyla mezitím pro zápis otevřena dalším procesem.
- zápis maximální velikosti `PIPE_BUF` (`limits.h`) je zaručený jako atomický, tj. data nesmí být proložena daty jiných zapisujících procesů. Např. v Solarisu 11 to je to 5120 bajtů, ve FreeBSD 5.4 je to 512 bajtů. Pokud zapisujete více, může se to stát. Zároveň platí, že pokud se zapisuje méně než `PIPE_BUF` bajtů, zapíšou se vždy najednou, a pokud je nastaveno `O_NONBLOCK` a nelze toho dosáhnout, vrátí se chyba.
- roura nemá pozici v souboru, zápis tak vždy přidává na konec.
- stejně se vzhledem ke čtení a zápisu chová i nepojmenovaná roura, viz strana 137.

Nastavení pozice: `lseek()`

`off_t` **lseek**`(int` *`fildes`*`, off_t` *`offset`*`, int` *`whence`*`);`

- nastaví pozici pro čtení a zápis v otevřeném souboru daném číslem deskriptoru `fildes` na hodnotu `offset`.
- podle hodnoty `whence` se `offset` počítá:
  - `SEEK_SET` ... od začátku souboru
  - `SEEK_CUR` ... od aktuální pozice
  - `SEEK_END` ... od konce souboru
- vrací výslednou pozici počítanou od začátku souboru.
- `lseek(fildes, 0, SEEK_CUR)` pouze vrátí aktuální pozici.

- Počítá se od 0, tj. offset 0 je první bajt souboru. Tam, kde to má smysl, je možné pro `offset` použít i záporné číslo. Příklad: `read/lseek.c`.
- Lze se přesunout i na pozici za koncem souboru. Pokud se pak provede zápis, soubor se prodlouží a v přeskočené části budou samé nuly (samotné `lseek` nestačí). Některé filesystémy takové bloky celých nul pro úsporu místa neukládají.
- Velikost souboru je možné zjistit pomocí `lseek(fildes, 0, SEEK_END)`.
- Nejčastější operace s `lseek` jsou **tři**: nastavení konkrétní pozice od začátku souboru, nastavení pozice na konec souboru a zjištění aktuální pozice v souboru (0 společně se `SEEK_CUR`)
- Při použití `lseek` se žádné I/O neprovede, tj. žádný příkaz se nepošle na řadič disku.
- `lseek` nemusí sloužit jen pro operace `read` a `write`, ale také pro následnou operaci `lseek`
- Seekování a zápis může vést k problémům se zálohami. Příklad: `read/big-file.c` demonstruje, že přesun souboru s "dírami"(tzv. sparse file) může vést k nárůstu velikosti souboru co se týče bloků alokovaných file systémem. Chování záleží na kombinaci operačního systému, archivního programu a file systému (a jejich verzí). Některé programy mají přepínače, které umožní díry zachovat (např. `dd` s `conv=sparse`, `tar` s `-S`, `rsync` s `--sparse`, atd.).
- Pozor na přehození parametrů. Druhá řádka samostatně vypadá OK, ale má přehozené parametry. `SEEK_SET` je navíc 0 a `SEEK_CUR` je 1, takže vás to nikam neposune a nic špatného se nestane, a o to je horší to pak najít.

```
lseek(fd, 1, SEEK_SET)
lseek(fd, SEEK_SET, 1)
```

Změna velikosti: `truncate()`

```
int truncate(const char *path, off_t length);
int ftruncate(int fildes, off_t length);
```

- změní délku souboru zadaného cestou nebo číslem deskriptoru na požadovanou hodnotu.
- při zkrácení souboru zruší nadbytečná data.
- standard ponechává nespecifikované, jestli funguje prodloužení souboru (s vyplněním přidaného úseku nulami). Proto je lepší k prodloužení souboru použít:

```
char buf = '\0';
lseek(fildes, length - 1, SEEK_SET);
write(fildes, &buf, 1);
```

- zrušit veškerý obsah souboru při otevření se dá příznakem `O_TRUNC` ve funkci `open`.
- podrobnosti je třeba hledat v manuálové stránce, např. ve FreeBSD v sekci BUGS nalezneme toto: *Use of `truncate` to extend a file is not portable.*

Duplikace deskriptoru: `dup()`, `dup2()`

`int` **`dup`**`(int` *`fildes`*`);`

- duplikuje deskriptor `fildes` na první volný deskriptor, vrátí nový deskriptor, který odkazuje na stejné otevření souboru.
- ekvivalent `fcntl(fildes, F_DUPFD, 0);`

`int` **`dup2`**`(int` *`fildes`*`, int` *`fildes2`*`);`

- duplikuje deskriptor `fildes` do deskriptoru `fildes2`.
- ekvivalent
  ```
  close(fildes2);
  fcntl(fildes, F_DUPFD, fildes2);
  ```

- první volný deskriptor se použije i u otevírání a vytváření souborů, viz strany 93 a 94.
- duplikovaný a původní deskriptor sdílí stejné otevření souboru a tedy i aktuální pozici a mód čtení/zápis.
- ekvivalent pro `dup2` není zcela ekvivalentní, např. pokud je `fildes` rovný `fildes2`, tak se neprovede `close(fildes2)` a `dup2` rovnou vrátí `fildes2`. Více viz POSIX 1003.1 (část XSH, System interfaces).

Příklad: implementace shellového přesměrování

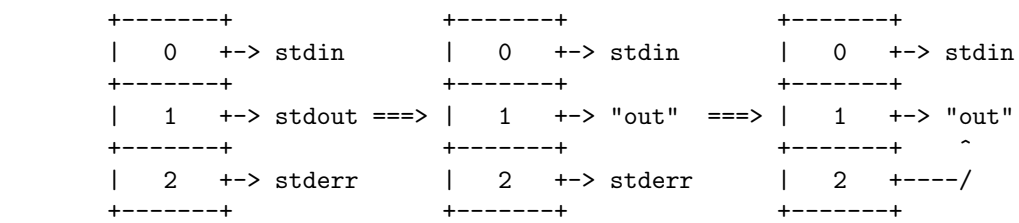
- `$ program < in > out 2>> err`

```
close(0);
open("in", O_RDONLY);
close(1);
open("out", O_WRONLY | O_CREAT | O_TRUNC, 0666);
close(2);
open("err", O_WRONLY | O_CREAT | O_APPEND, 0666);
```

- `$ program > out 2>&1`

```
close(1);
open("out", O_WRONLY | O_CREAT | O_TRUNC, 0666);
close(2);
dup(1);
```

- všimněte si korespondence flagu `O_APPEND` s `>>`
- Další příklad použití `dup` uvidíme, až se budeme zabývat rourami. První přesměrování (bez `stderr`) je v `read/redirect.c`. Volání `execl` v tomto příkladu nahradí aktuální obraz běžícího procesu obrazem programu předaného jako první argument. Více o volání `execl` je na straně 131.
- pro pochopení toho jak funguje přesměrování je dobré si nakreslit tabulku deskriptorů v čase a kam "ukazují". např. pro druhý příklad (iniciální stav, stav po `close(1) open("out", ...)`, konečný stav):



- Je potřeba si dát pozor na stav deskriptorů. Druhý příklad nebude fungovat, když bude deskriptor 0 uzavřen, protože `open` vrátí deskriptor 0 (první volný) a `dup` vrátí chybu (pokus o duplikaci uzavřeného deskriptoru). Možné řešení:

```
close(1);
if((fd = open("out", O_WRONLY | O_CREAT | O_TRUNC, 0666)) == 0)
    dup(0);
```

```
close(2);
dup(1);
if(fd == 0)
    close(0);
```

nebo

```
fd = open("out", O_WRONLY | O_CREAT | O_TRUNC, 0666);
if(fd != 1) {
    dup2(fd, 1);
    close(fd);
}
dup2(1, 2);
```

Řídicí funkce souborů a zařízení: `fcntl()`, `ioctl()`

`int` **`fcntl`**`(int` *`fildes`*`, int` *`cmd`*`, ...);`

- slouží pro duplikaci deskriptorů, nastavování zámků, testování a nastavování různých příznaků souboru.

  příklad: zavření standardního vstupu při spuštění programu (volání typu exec)

  ```
  fcntl(0, F_SETFD, FD_CLOEXEC);
  ```

`int` **`ioctl`**`(int` *`fildes`*`, int` *`request`*`, ... );`

- rozhraní pro řídicí funkce periferních zařízení
- používá se jako univerzální rozhraní pro ovládání zařízení, každé zařízení definuje množinu příkazů, kterým rozumí.

- jak později uvidíme, i k socketům se přistupuje přes souborové deskriptory, a je tedy možné `fnctl` použít i na ně, třeba pro nastavení neblokujícího socketu. Více informací viz strana 192.
- možné hodnoty `cmd` ve funkci `fcntl`:
  - `F_DUPFD` ... duplikace deskriptoru
  - `F_GETFD` ... zjištění příznaků deskriptoru (`FD_CLOEXEC` – uzavření při exec). `FD_CLOEXEC` je jediný flag pro deskriptory, definovaný v normě UNIX 03.
  - `F_SETFD` ... nastavení příznaků deskriptoru

  - `F_GETFL` ... zjištění módu čtení/zápis a příznaků otevření souboru (jako u `open`)
  - `F_SETFL` ... nastavení příznaků otevření souboru (`O_APPEND`, `O_DSYNC`, `O_NONBLOCK`, `O_RSYNC`, `O_SYNC`). Nemohu nastavit příznaky pro RO/RW a ani příznaky pro vytvoření, zkrácení nebo exkluzivní přístup k souboru.
  - `F_GETLK`, `F_SETLK`, `F_SETLKW` ... nastavování zámků

- je důležité si uvědomit, že jsou dva druhy příznaků – příznak(y) pro souborový deskriptor a příznaky pro jedno konkrétní otevření souboru – tj. příznaky jsou uložené ve dvou různých tabulkách.

- periferní zařízení podporují čtení a zápis dat pomocí `read`, `write` a mapování dat do paměti (`mmap`), veškeré další operace se zařízením (např. nastavení parametrů, zamčení nebo eject) se dělají funkcí `ioctl`.

- při nastavování příznaků nejdřív vždy zjistěte, jaké byly předtím. I když jste si jisti, že v dané chvíli jsou nulové a je tedy možné provést `fcntl(fd, O_APPEND)`, nemůžete vědět, co se může změnit (například o pár řádků výše nějaký flag přidáte, aniž byste věděli, že jste ovlivněni kódem dole). Tedy vždy použijte například toto:

  ```
  flags = fcntl(fd, F_GETFL);
  if (fcntl(fd, F_SETFL, flags | O_APPEND) == -1)
    ...
  ```

  ... a podobně pro odebrání flagu – je špatné řešení nastavit hodnotů flagů na nulu, místo toho je třeba použít bitového jedničkového doplňku příslušného flagu (`flags & ~O_APPEND`).

Informace o souboru: `stat()`

```
int stat(const char *path, struct stat *buf);
int fstat(int fildes, struct stat *buf);
```

- pro soubor zadaný cestou, resp. číslem deskriptoru, vrátí strukturu obsahující informace o souboru, např.:
  - `st_ino` ... číslo i-uzlu
  - `st_dev` ... číslo zařízení obsahujícího soubor
  - `st_uid`, `st_gid` ... vlastník a skupina souboru
  - `st_mode` ... typ souboru a jeho přístupová práva
  - `st_size`, `st_blksize`, `st_blocks` ... velikost souboru v bajtech, preferovaná velikost bloku pro I/O a počet bloků
  - `st_atime`, `st_mtime`, `st_ctime` ... časy posledního přístupu, modifikace souboru a modifikace i-uzlu
  - `st_nlink` ... počet odkazů na soubor

- update času posledního přístupu k souboru lze na běžných file systémech vypnout pomocí optionu `noatime` příkazu `mount`. Hodit se to může pro urychlení, pokud se provádí čtení velkého množství souborů a nezáleží na času přístupu na těchto souborech (např. při kompilaci).
- *Metadata* jsou informace o souboru – tedy mód, časy přístupu, délka, vlastník a skupina atd. Nepatří mezi ně skutečná data souboru, a ani jméno, které není uloženo v rámci daného souboru, ale v adresáři či v adresářích.
- Metadata je možné přečíst, i když proces nemá práva pro čtení obsahu souboru.
- Touto funkcí nezískám flagy deskriptoru ani flagy z pole tabulky otevřených souborů v systému, zde jde o informace ohledně souboru uloženého na paměťovém médiu.
- Co se stane když se zavolá `fstat` na deskriptory 0,1,2 ? (předpokládáme že nejsou přesměrované z normálního stavu)
- `st_ctime` není čas vytvoření souboru (creation time), ale čas změny indexového uzlu (change time)
- Norma nespecifikuje pořadí položek ve struktuře ani nezakazuje přidat další.
- Příklad: `stat/stat.c`

Informace o souboru (2)

- pro typ souboru jsou v `<sys/stat.h>` definovány konstanty `S_IFMT` (maska pro typ), `S_IFBLK` (blokový speciální), `S_IFCHR` (znakový speciální), `S_IFIFO` (FIFO), `S_IFREG` (obyčejný), `S_IFDIR` (adresář), `S_IFLNK` (symlink).
- typ lze testovat pomocí maker `S_ISBLK(m)`, `S_ISCHR(m)`, `S_ISFIFO(m)`, `S_ISREG(m)`, `S_ISDIR(m)`, `S_ISLNK(m)`.
- konstanty pro přístupová práva: `S_IRUSR` (čtení pro vlastníka), `S_IWGRP` (zápis pro skupinu), atd.

`int` **lstat**`(const char *`*path*`, struct stat *`*buf*`);`

- když je zkoumaný soubor symlink, `stat()` vrátí informace o souboru, na který ukazuje. Tato funkce vrací informace o symlinku.

- typ a práva souboru jsou uložena společně v `st_mode`, proto existují zmiňovaná makra.
- S_IFMT specifikuje tu část bitů, které jsou věnované typu souboru, makra pro jednotlivé typy pak nejsou masky, ale hodnoty, takže test na typ souboru je nutné udělat takto: `(st_mode & S_IFMT == S_IFREG)`. Všechna makra jsou v normě, takže jejich používáním zaručíme přenositelný kód.
- Příklad: `stat/filetype.c`

Nastavení časů souboru

`int` **utime**`(const char *`*path*`, const struct utimbuf *`*times*`);`

- nastaví čas poslední modifikace souboru a čas posledního přístupu k souboru.
- nelze změnit čas poslední modifikace i-uzlu.
- volající proces musí mít právo zápisu pro soubor.

- Tuto funkci používají hlavně kopírovací a archivační programy, aby zajistily stejné časy kopie a originálu (např. `tar` nebo `rsync`).
- Shellové rozhraní pro funkci `utime` představuje příkaz `touch`. Měnit takto čas modifikace i-uzlu ale nelze.

Test přístupových práv: `access()`

```
int access(const char *path, int amode);
```

- otestuje, zda volající proces má k souboru `path` práva daná OR-kombinací konstant v `amode`:
  - `R_OK` ... test práva na čtení
  - `W_OK` ... test práva na zápis
  - `X_OK` ... test práva na spuštění
  - `F_OK` ... test existence souboru
- na rozdíl od `stat()`, výsledek závisí na RUID a RGID procesu
- toto volání se nedá použít bezpečně, proto ho nikdy nepoužívejte procesu

- Volání `access` aplikuje mechanismus testování přístupových práv k zadanému souboru pro volající proces a vrátí výsledek.
- Funkce `access` volání byla pro setuid proces, aby si mohl ověřit, zda uživatel běžící daný setuid proces by měl za normálních okolností k příslušnému souboru přístup. Z toho vyplývá, že toto volání je **security hole** – mezi testem a následnou akcí se soubor může změnit třeba tak, že původní (můj) soubor smažu a vytvořím symbolický link se stejným jménem na soubor, ke kterému bych jinak neměl přístup. Toto proces nemůže nikdy ošetřit, takže vesele zmodifikuje soubor, který bych jako uživatel nikdy nemohl změnit. Správným řešením je vrátit se zpátky k reálným UID/GID a přístup rovnou vyzkoušet. Pokud například daný soubor otevřeme, už nám ho nikdo pod rukama "nevymění".

Nastavení přístupových práv

`int` **chmod**`(const char *`*`path`*`, mode_t `*`mode`*`);`

- změní přístupová práva souboru `path` na hodnotu `mode`.
- tuto službu může volat pouze vlastník souboru nebo superuživatel (root).

`int` **chown**`(const char *`*`path`*`, uid_t `*`owner`*`, gid_t `*`group`*`);`

- změní vlastníka a skupinu souboru `path`. Hodnota `-1` znamená zachovat vlastníka, resp. skupinu.
- měnit vlastníka může jen superuživatel, aby uživatelé nemohli obcházet nastavené quoty tím, že své soubory předají někomu jinému.
- běžný uživatel může měnit skupinu svých souborů a musí přitom patřit do cílové skupiny.

- parametr `mode` zde samozřejmě neobsahuje typ souboru, jako tomu je například u volání `stat`. Hodnoty `mode` viz `chmod`(2).
- pokud nejsem vlastník, nemohu celkem logicky změnit mód ani u souboru s nastaveným přístupem `rw-rw-rw-`
- v některých implementacích může vlastník souboru předat vlastnictví někomu jinému, např. v IRIXu je chování `chown` nastavitelné jako parametr jádra.
- není běžné volat funkci `chmod` z uživatelských aplikací, na to se používají flagy u volání `open`, hlavní použití je v aplikaci `chmod(1)`

Manipulace se jmény souborů

`int` **link**`(const char *`*path1*`, const char *`*path2*`);`

- vytvoří nový odkaz (položku adresáře) `path2` na soubor `path1`. Funguje pouze v rámci jednoho svazku.

`int` **unlink**`(const char *`*path*`);`

- zruší odkaz na soubor. Po zrušení posledního odkazu na soubor a uzavření souboru všemi procesy je soubor smazán.

`int` **rename**`(const char *`*old*`, const char *`*new*`);`

- změní jméno souboru (přesně odkazu na soubor) z `old` na `new`. Funguje pouze v rámci jednoho svazku.

- opět zdůrazňuji, že **unix nemá volání typu delete na soubory**. Podrobněji viz strana 69.
- volání `link` vytváří hardlinky, tj. zobrazení ze jména souboru na číslo i-uzlu. Čísla i-uzlů jsou jednoznačná pouze v rámci svazku, proto pro linky mezi filesystémy je nutné použít symlinky.
- parametr `path2` nesmí existovat, tedy nelze takto přejmenovávat
- `unlink` nefunguje na adresáře
- shellový příkaz `mv` používá `rename` pro přesuny v rámci jednoho svazku. Přesun souboru mezi filesystémy vyžaduje nejprve soubor zkopírovat a pak smazat originál voláním `unlink`.
- `rename` funguje nad symlinky, ne nad soubory, na které symlink ukazuje
- existuje i volání `remove`, viz strana 114.

Symbolické linky

`int` **symlink**`(const char *`*path1*`, const char *`*path2*`);`

- vytvoří symbolický link `path2` → `path1`.
- cíl symbolického linku může být i na jiném svazku, popřípadě nemusí vůbec existovat.

`int` **readlink**`(const char *`*path*`, char *`*buf*`, size_t` *bufsize*`);`

- do `buf` dá max. `bufsize` znaků z cesty, na kterou ukazuje symlink path.
- vrátí počet znaků uložených do `buf`.
- obsah buf není zakončen nulou (znakem `'\0'`)!

- Shellový příkaz `ln` volá `symlink` nebo `link`, podle toho, jestli je použit přepínač `-s` nebo ne.
- Smazání hardlinku nesmaže soubor, pokud na něj vede ještě jiný hardlink. Naopak soubor (položku adresáře i data) je možné smazat, i když na něj ukazují nějaké symlinky.
- `readlink` je použitelný v situaci, pokud chci smazat soubor, na který daný symlink ukazuje
- `bufsize` se typicky dává o 1 menší než velikost bufferu, to pro ukončení znakem `'\0'`

Manipulace s adresáři

`int` **`mkdir`**`(const char *`*`path`*`, mode_t `*`mode`*`);`

- vytvoří nový prázdný adresář, který bude obsahovat pouze položky '.' a '..'

`int` **`rmdir`**`(const char *`*`path`*`);`

- smaže adresář `path`. Adresář musí být prázdný.

`DIR *`**`opendir`**`(const char *`*`dirname`*`);`
`struct dirent *`**`readdir`**`(DIR *`*`dirp`*`);`
`int` **`closedir`**`(DIR *`*`dirp`*`);`

- slouží k sekvenčnímu procházení adresářů.
- struktura dirent obsahuje položky
  - `d_ino` ... číslo i-uzlu
  - `d_name` ... jméno souboru

- Položky adresáře nejsou nijak uspořádány, `readdir` je může vracet v libovolném pořadí. V případě chyby se vrací NULL jako předtím a `errno` je nastaveno. Konec adresáře se signalizuje tím, že `readdir` vrátí NULL a `errno` není změněno.
- `readdir` je stavová funkce. Pro vrácení se na začátek je možné použít funkci `rewinddir`. Pokud tuto funkci voláte z více vláken je nutné používat reentrantní `readdir_r`, protože struktura `dirent` je statická.
- V některých implementacích (např. FreeBSD) lze adresář otevřít pro čtení (ne pro zápis) jako normální soubor a číst ho pomocí `read`, ale je třeba znát jeho vnitřní organizaci. Proto je `readdir` pro zpracování obsahu adresáře lepší než `read`, který vrací raw data adresáře. Kromě toho, norma nevyžaduje, aby adresář bylo možné číst funkcí `read`, Linux to například nedovolí.
- `d_ino` není moc užitečné, protože v případě, kdy daný adresář je mount point, tak ukazuje na adresář, na který je další filesystém namontován, ne na kořen namontovaného filesystému
- Některé systémy mají ve struktuře `dirent` položku `d_type`. Ta může nabývat hodnot `DT_REG`, `DT_DIR`, `DT_FIFO` atd., viz manuálová stránka pro `dirent`. Byla to věc specifická pro BSD a následně převzatá i jinými systémy, například Linuxem. Není to součástí normy a tedy to **není přenositelné.** Přenositelný způsob je na každou položku zavolat `stat`.
- `rmdir` nefunguje na neprázdný adresář, musíte sami smazat jeho obsah před smazáním adresáře. Někdy lze v kódu nalézt zoufalé pokusy typu `system("rm`

`-r xxx")` což má svoje nevýhody (např. závislost programu na shellu. Dále pozor na sanitizaci proměnných prostředí a jména adresáře).

- Norma specifikuje i volání `remove`, které se chová jako `unlink` pro regulární soubory, a jako `rmdir` pro adresáře.

Příklad: procházení adresáře

```
int main(int argc, char *argv[])
{
    int i;
    DIR *d;
    struct dirent *de;
    for(i = 1; i < argc; i++) {
        d = opendir(argv[i]);
        while(de = readdir(d))
            printf("%s\n", de->d_name);
        closedir(d);
    }
    return (0);
}
```

- příkaz `ls` je založen na takovéto smyčce, navíc provádí např. třídění jmen souborů a zjišťování dalších informací pomocí `stat`.
- konec adresáře se pozná tak, že `readdir` vrátí `NULL`. To však vrátí i v případě, pokud nastala chyba. V takovém případě je kód chyby v proměnné `errno`, v případě prvním je `errno` nezměněna. Proto by `errno` mělo být vždy nastavené na nulu před voláním `readdir`. V příkladu tomu tak není, protože z důvodu málo místa nekontrolujeme `errno` vůbec.
- příklad: `readdir/readdir.c`

Aktuální adresář procesu

- každý proces má svůj aktuální (pracovní) adresář, vůči kterému jsou udávány relativní cesty k souborům. Počáteční nastavení pracovního adresáře se dědí od otce při vzniku procesu.

```
int chdir(const char *path);
int fchdir(int fildes);
```

- nastaví nový pracovní adresář procesu.

```
char *getcwd(char *buf, size_t size);
```

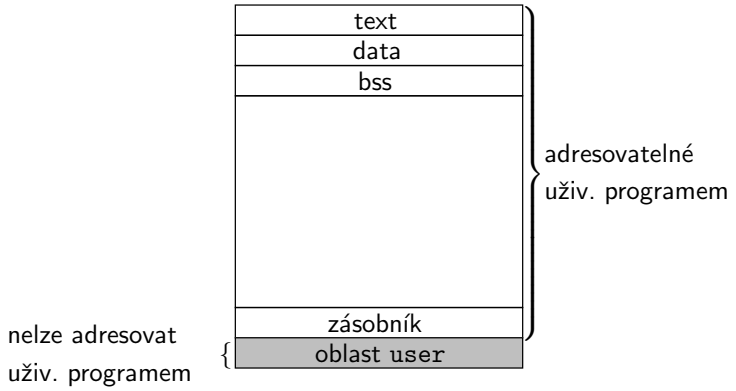
- uloží absolutní cestu k aktuálnímu adresáři do pole `buf`, jeho délka (`size`) musí být aspoň o 1 větší než délka cesty.

- funkci `fchdir` se předává deskriptor získaný voláním `open` na adresář.
- funkce `getcwd` může vrátit jinou cestu než tu, po které jsme se do aktuálního adresáře dostali, jesliže část cesty v `chdir` byl symlink nebo došlo k přesunu (přejmenování) některého adresáře na cestě od kořene. U současných unixů by se to ale už stát nemělo.
- na většině moderních unixových systémů existuje funkce `chroot`, která umožňuje změnit kořenový adresář volajícího procesu na daný adresář. Je často používaná v implementací serverů k omezení přístupu pouze na daný podstrom file systému (např. FTP servery). Pokud se používá pro zapezpečení, vyžaduje to zvláštní opatrnost, protože existují cesty jak se dostat do původního stromu. Pokud je nutné z chroot prostředí spouštět programy, nastávají další komplikace s nutností replikace kritických systémových souborů do chroot prostředí (pokud není program upraven aby měl vše zabudované v sobě). Tato funkce není součástí aktuální verze standardu (POSIX 1003.1-2008).
  - další, mnohem obecnější, způsoby izolace procesů se dají nalézt v unixových systémech (jails v FreeBSD, zones v Solarisu, sandboxing v Mac OS X). Zpravidla se velmi liší tím jaké hranice dokáží vytyčit, k čemu jsou určeny, způsobem administrace, ...

Obsah

- úvod, vývoj UNIXu a C, programátorské nástroje
- základní pojmy a konvence UNIXu a jeho API
- přístupová práva, periferní zařízení, systém souborů
- **manipulace s procesy, spouštění programů**
- signály
- synchronizace a komunikace procesů
- síťová komunikace
- vlákna, synchronizace vláken
- ??? - bude definováno později, podle toho kolik zbyde času

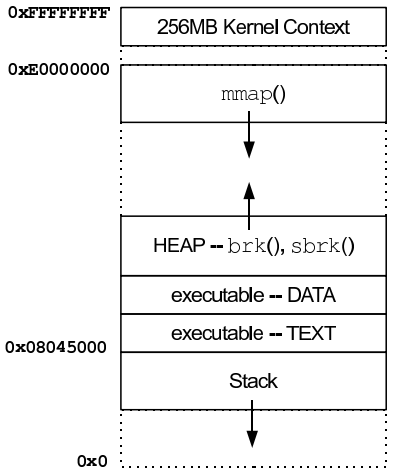
Paměť procesu v uživatelském režimu



- každý proces má tři základní segmenty (paměťové segmenty, nemluvíme o hardwarových segmentech):
  - text ... kód programu
  - data ... inicializované proměnné
  - zásobník
- sekce `text` a `data` jsou uloženy ve spustitelném souboru
- sekce pro inicializované i neinicializované proměnné a heap jsou brány dohromady jako data
- dále lze do adresového prostoru připojit segmenty sdílené paměti (`shmat`) nebo soubory (`mmap`).
- text je sdílen všemi procesy, které provádí stejný kód. Datový segment a zásobník jsou privátní pro každý proces.
- každý systém může používat zcela jiné rozdělení adresového prostoru procesu (a typicky tomu tak je). Konkrétní příklad je na následujícím slajdu, a zobrazuje i sekce pro `mmap` a *heap*.
- *bss* ... neinicializované proměnné (`bss` pochází z assembleru IBM 7090 a znamená „block started by symbol"). Za běhu programu tvoří sekce `data`, `bss` a heap (není na obrázku) dohromady datové segmenty procesu. Velikost heapu lze měnit pomocí systémových volání `brk` a `sbrk`.

- poznámka – neinicializované proměnné jsou statické proměnné, které (překvapivě) nejsou inicializované – tj. globální proměnné nebo proměnné definované jako `static` ve funkcích i mimo funkce. Jak víte z přednášek o jazyku C, všechny tyto proměnné jsou při startu programu automaticky inicializované nulami. Proto není nutné mít jejich hodnotu v binárce. Jakmile ale nějakou z takových proměnných inicializujete, bude již součástí datového segmentu programu na disku.
- *(uživatelský) zásobník* ... lokální nestatické proměnné, parametry funkcí (na určitých architekturách v daných modech - 32-bit x86), návratové adresy. Každý proces má dva zásobníky, jeden pro uživatelský režim a jeden pro režim jádra. Uživatelský zásobník procesu automaticky roste podle potřeby (neplatí, pokud se používají vlákna, tam má navíc každé vlákno zásobník svůj).
- *oblast user (u-area)* ... obsahuje informace o procesu používané jádrem, které nejsou potřebné, když je proces odložen na disku (počet otevřených souborů, nastavení ošetření signálů, počet segmentů sdílené paměti, argumenty programu, proměnné prostředí, aktuální adresář, atd.). Tato oblast je přístupná pouze pro jádro, které vždy vidí právě jednu u-oblast patřící právě běžícímu procesu. Další informace o procesu, které jádro může potřebovat i pro jiný než právě běžící proces, nebo i když je proces odložen, jsou ve struktuře `proc`. Struktury `proc` pro všechny procesy jsou stále rezidentní v paměti a viditelné v režimu jádra.

Příklad: Solaris 11 x86 32-bit



- z obrázku lze vyčíst několik dalších věcí:

  - maximální velikost kernelu pro Solaris 11 x86 32-bit je 256 megabajtů
  - mezi namapováním kernelu a pamětí vyhrazenou pro `mmap` je volné místo
  - zásobník roste směrem k nižším adresám a jeho velikost je omezena na 128 megabajtů

- *heap* je část paměti, kterou si proces může zvětšovat pomocí volání `brk` a `sbrk`, a je často používaná funkcí `malloc`. Funguje to tak, že `malloc` si postupně zvětšuje heap podle potřeby, a přiřazenou paměť interně spravuje a přiděluje ji procesu po částech. Když tedy voláte `free`, neznamená to, že vracíte paměť systému, ale pouze internímu alokátoru.

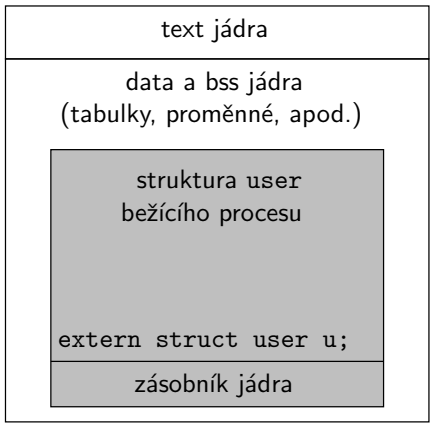
- oblast pro `mmap` se používá pro mapování souborů, tedy i sdílených knihoven. Některé alokátory používají interně i tuto paměť, například pro větší kusy paměti žádané najednou. Je možné exkluzivně používat pouze `mmap`, pro aplikaci je to zcela transparentní. Při použití `mmap` je možné paměť systému vracet (voláním `munmap`), na rozdíl od implementace pomocí `brk`/`sbrk`.

- obrázek byl převzat z [McDougall-Mauro], a není na něm oblast pro neinicializované proměnné. Pokud si ale na tomto systému necháte vypsat adresu jedné z nich, zjistíte, že inicializované i neinicializované proměnné sdílí společný datový segment, na obrázku označený jako „executable – DATA". Příklad: `pmap/proc-addr-space.c`

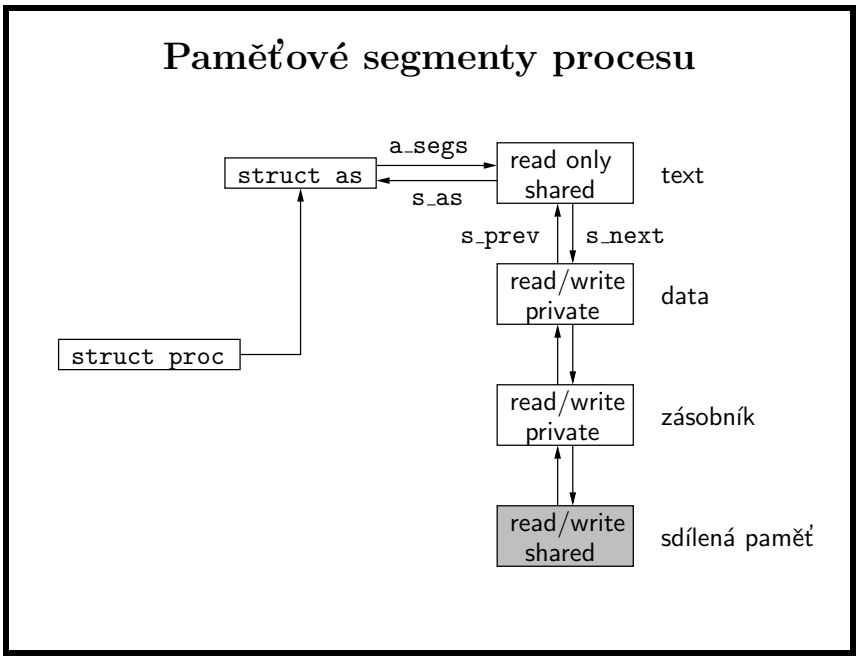
- mapování kernelu není nutné, například u Solarisu na *amd64* architektuře (tj. 64-bit) už kernel do uživatelského prostoru procesu mapován není.

- `brk` ani `sbrk` nejsou součástí normy, přenositelné aplikace by proto měly používat, pokud podobnou funkcionalitu potřebují, volání `mmap`, viz strana 140.

Paměť procesu v režimu jádra

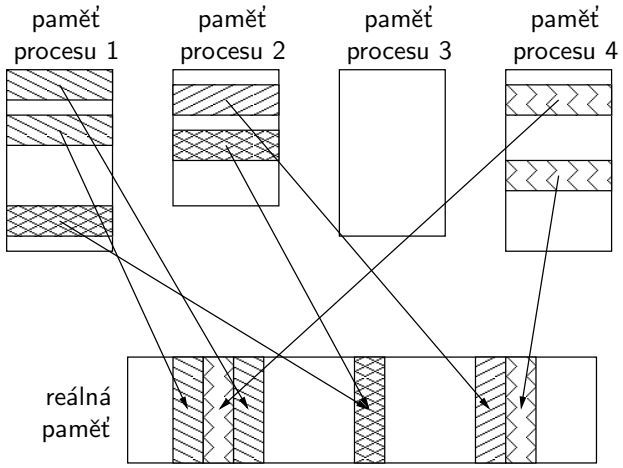


- proces se dostane do režimu jádra buď příchodem *přerušení vyvolaného procesorem* (výpadek stránky, neznámá instrukce,...), *časovačem* (v pravidelných intervalech je potřeba aktivovat plánovač procesů), *periferním zařízením*, nebo instrukcí synchronního přerušení (standardní knihovna takto předává řízení jádru, aby obsloužilo *systémové volání*).
- v paměti je pouze jedna kopie kódu a dat jádra, sdílená všemi procesy. Kód jádra je vždy celý rezidentní v paměti, není odkládán na disk.
- *text jádra* ... kód jádra operačního systému, zavedený při startu systému a rezidentní v paměti po celou dobu běhu systému. Některé implementace umožňují přidávat funkční moduly do jádra za běhu (např. při přidání nového zařízení se do jádra dynamicky přidá nový ovladač), není proto třeba kvůli každé změně regenerovat jádro a restartovat systém.
- *data a bss jádra* ... datové struktury používané jádrem, součástí je i u-oblast právě běžícího procesu.
- *zásobník jádra* ... samostatný pro každý proces, je prázdný, jestliže je proces v uživatelském režimu (a tedy používá uživatelský zásobník).



- takto vypadá většinou reprezentace paměťových segmentů procesu v jádře.
- základním rysem této architektury je tzv. *memory object*, což je abstrakce mapování mezi kusem paměti a místem, kde jsou data normálně uložena (tzv. *backing store* nebo *data object*). Takové místo uložení může být například swap nebo soubor. Adresový prostor procesu je pak množina mapování na různé datové objekty. Existuje i *anonymní objekt*, který nemá místo trvalého uložení (používá se například pro zásobník). Fyzická paměť pak slouží jako cache pro data těchto namapovaných datových objektů.
- tato zde velmi hrubě popsaná architektura se nazývá VM (od *Virtual Memory*), a objevila se v SunOS 4.0. Na této architektuře je založena architektura viruální paměti v SVR4. Více informací viz [Vahalia], původní článek z roku 1987 představující tuto architekturu: Gingell, R. A., Moran J. P., Shannon, W. A. – *Virtual Memory Architecture in SunOS* nebo přednáška o operačních systémech na MFF (NSWI004 - Operační systémy).
- z jakých segmentů se skládá paměťový prostor konkrétního procesu lze zjistit pomocí příkazu `pmap(1)` (na Solarisu, na NetBSD a v některých Linuxových distribucích) nebo `procmap(1)` (v OpenBSD) nebo `vmmap(1)` v Mac OS X.

Virtuální paměť



- každý proces vidí svůj adresový prostor jako souvislý interval (virtuálních) adres od nuly po nějakou maximální hodnotu. Přístupné jsou pouze ty adresy, na kterých je namapován některý segment procesu (to je právě to mapování, o kterém se mluví na předchozím slajdu).
- jádro dále rozděluje paměť procesu na stránky. Každá stránka má své umístění v rámci fyzické paměti. Toto umístění je dáno stránkovacími tabulkami jádra a stránky mohou být v rámcích libovolně promíchány vůči jejich pořadí ve virtuální adresovém prostoru.
- pokud není stránka právě používána, může být také odložena na disk.
- paměťový manager jádra zajišťuje mapování mezi virtuálními adresami používanými kódem uživatelských procesů i jádra na fyzické adresy a načtení odložených stránek z disku při výpadku stránky.

Implementace virtuální paměti

- procesy v UNIXu používají k přístupu do paměti virtuální adresy, které na fyzické adresy převádí hardware ve spolupráci s jádrem systému.
- při nedostatku volné paměti se odkládají nepoužívané úseky paměti do odkládací oblasti (**swap**) na disk.
- před verzí SVR2 se procesem `swapper` (nyní `sched`) odkládaly celé procesy.
- od verze SVR2 se používá stránkování na žádost (**demand paging**) a **copy-on-write**. Stránky se alokují až při prvním použití a privátní stránky se kopírují při první modifikaci. Uvolňování a odkládání jednotlivých stránek provádí proces `pageout`, odkládání celých procesů nastupuje až při kritickém nedostatku paměti.

překlad adres: přístup na neplatnou adresu nebo pokus o zápis do paměti pouze pro čtení vyvolá signál `SIGSEGV`.

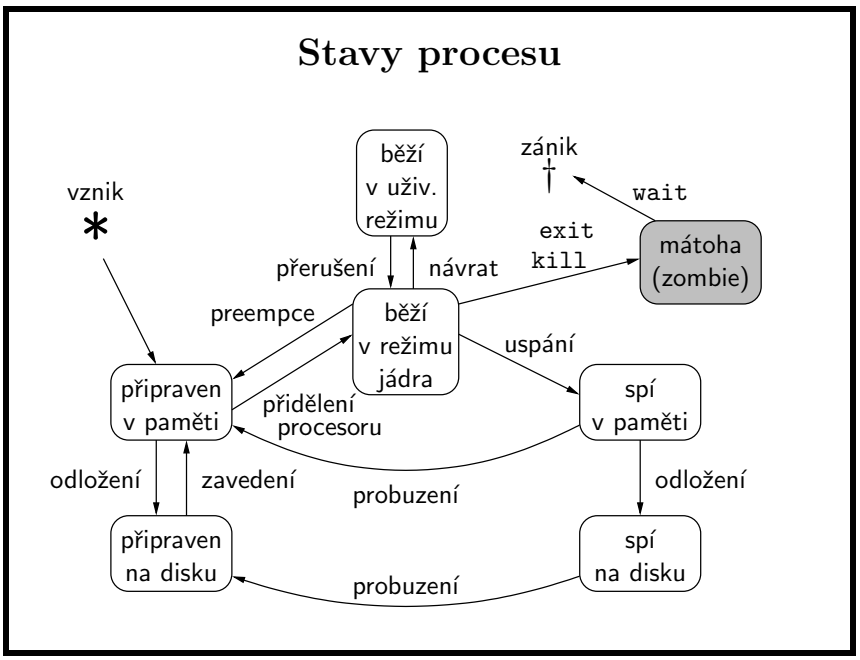
swap: odkládací prostor se vytváří na samostatném oddílu disku, od SVR4 může být i v souboru.

swapper: proces `swapper` se snaží odložit na disk nějaký proces, který není zamčen v paměti, a na uvolněné místo zavést dříve odložený proces.

demand paging: při žádosti procesu o paměť se pouze upraví tabulka stránek. První instrukce adresující obsah stránky vyvolá výjimku. Jádro ji ošetří tím, že alokuje stránku.

copy-on-write: více procesů může sdílet zapisovatelnou fyzickou stránku, která je ale logicky privátní pro každý proces (tato situace nastane např. po vytvoření procesu voláním `fork`). Dokud procesy z paměti pouze čtou, přistupují ke sdílené stránce. Pokud se proces pokusí obsah stránky změnit, vyvolá výjimku. Jádro zkopíruje stránku, přidělí procesu kopii, která už je privátní a proces ji může dále libovolně měnit. Ostatní procesy používají stále nezměněnou původní stránku.

stránky k odložení se hledají algoritmem *NRU* (not recently used): každá stránka má příznaky `referenced` a `modified`, na začátku vynulované. Při prvním přístupu se nastaví `referenced`, při změně `modified`. Oba příznaky se periodicky nulují. Přednostně se uvolňují stránky, které nejsou modifikované ani použité. Stránky kódu programu a mapovaných souborů se neukládají do odkládacího prostoru, ale obnovují se z příslušného souboru.



- po ukončení procesu voláním `exit` nebo v reakci na signál přechází proces do stavu mátoha (zombie), protože jádro si musí pamatovat k číslu procesu jeho návratovou hodnotu. Celá paměť procesu je uvolněna, zbývá pouze struktura `proc`. Proces lze definitivně zrušit, až když se jeho rodič zeptá na návratovou hodnotu voláním typu `wait`. Když nebude původní rodič k dispozici, provede volání `wait` proces `init`, který převzal rodičovství.
- v dnešních UNIXech se obvykle do odkládací oblasti na disku (swap area) neodkládají celé procesy, ale jednotlivé stránky paměti.
- proces je *uspán*, když o to sám požádá, např. začne čekat na dokončení periferní operace. *Preempce* je naopak nedobrovolné odebrání procesoru plánovačem.

Plánování procesů

- *preemptivní plánování* – jestliže se proces nevzdá procesoru (neuspí se čekáním na nějakou událost), je mu odebrán procesor po uplynutí časového kvanta.
- procesy jsou zařazeny do front podle priority, procesor je přidělen vždy prvnímu připravenému procesu z fronty, která má nejvyšší prioritu.
- v SVR4 byly zavedeny prioritní třídy a podpora procesů reálného času (real-time) s garantovanou maximální dobou odezvy.
- na rozdíl od předchozích verzí znamená v SVR4 vyšší číslo vyšší prioritu.

- **základem preemptivního plánování jsou pravidelná přerušení od časovače**, která odeberou procesor běžícímu procesu a předají řízení jádru (aktivuje se plánovač procesů).
- jiná varianta je nepreemptivní (kooperativní) plánování, kdy proces běží, dokud se sám nevzdá procesoru, tj. dokud nezavolá takovou systémovou funkci, která přepne kontext na jiný proces. Nevýhodou kooperativního plánování je, že jeden proces může stále blokovat procesor a ostatní procesy se nikdy nedostanou na řadu.
- UNIX používá pro uživatelské procesy pouze preemptivní plánování.
- tradiční UNIXové **jádro** funguje kooperativním způsobem, tj. proces běžící v režimu jádra není přeplánován, dokud se sám nevzdá procesoru. **Jádra moderních UNIXů jsou již preemtivní** – je to hlavně kvůli real-time systémům; tam je potřeba mít možnost běžící proces zbavit procesoru okamžitě, nečekat na to, až se vrátí z režimu jádra nebo se sám uspí. Pozor na ať to nepletete – UNIX byl od samého začátku preemptivní systém, ale jeho jádro bylo nepreemptivní.
- při preemptivním plánování může být proces kdykoliv přerušen a řízení předáno jinému procesu. Proces si proto nikdy nemůže být jistý, že určitou operaci (více než jednu intstrukci, kromě systémových volání se zaručenou atomičností) provede atomicky, bez ovlivnění ostatními procesy. Pokud je třeba zajistit atomičnost nějaké akce, musí se procesy navzájem synchronizovat. Při kooperativním plánování problém synchronizace odpadá (atomická posloupnost operací se zajistí tím, že se proces během ní nevzdá procesoru).

Prioritní třídy

- **systémová**
  - priorita 60 až 99
  - rezervována pro systémové procesy (`pageout`, `sched`, ...)
  - pevná priorita
- **real-time**
  - priorita 100 až 159
  - pevná priorita
  - pro každou hodnotu priority definováno časové kvantum
- **sdílení času (time-shared)**
  - priorita 0 až 59
  - proměnná dvousložková priorita, pevná uživatelská a proměnná systémová část – pokud proces hodně využívá procesor, je mu snižována priorita (a zvětšováno časové kvantum)

- systémová třída je používána pouze jádrem, uživatelský proces běžící v režimu jádra si ponechává svou plánovací charakteristiku.
- procesy ve třídě reálného času mají nejvyšší prioritu, proto musí být správně nakonfigurovány, aby nezablokovaly zbytek systému.
- jestliže je proces ve třídě sdílení času uspán a čeká na nějakou událost, je mu dočasně přiřazena systémová priorita. Po probuzení se takový proces dostane na procesor dříve, než ostatní procesy, které nespí.
- pevná část priority procesu ve třídě sdílení času se dá nastavit pomocí
  `int` **setpriority**`(int` *which*`, id_t` *who*`, int` *prio*`);`
  nebo
  `int` **nice**`(int` *incr*`);`
  Hodnota *which* udává co bude v argumentu *who*. Pokud je např. *which* *PRIO_PGRP*, bude v *who* číslo skupiny procesů. Pozor na volání **nice** které vrací novou hodnotu nice. Protože je -1 validní hodnota, je potřeba před volání vyčistit `errno` a pokud funkce vrátí -1 tak ji zkontrolovat.
- prioritní třídu a hodnotu nice daného procesu lze zobrazit přepínačem -l programu ps(1) nebo pomocí specifikace hodnot které se mají vypsat.
- Příklad: hodnoty priority mají na různých systémech různé škály. Např. na Mac OS X 10.9 má spuštěný proces hodnotu priority 30, po zvýšení hodnoty nice (proces je "hodnější"na ostatní procesy a dobrovolně tedy snížil svoji prioritu) se mu hodnota prorita sníží na 21 což na Mac OS X znamená i snížení priority:

```
$ sleep 200 &
[1] 36877
$ ps -O pri,nice -p $!
  PID PRI NI   TT  STAT      TIME COMMAND
36877  31  0 s003  S      0:00.00 sleep 200
$ renice 10 -p $!
$ ps -O pri,nice -p $!
  PID PRI NI   TT  STAT      TIME COMMAND
36877  21 10 s003  SN     0:00.00 sleep 200
```

Na Linuxu 3.10 to ale bude vypadat jinak - po zvýšení hodnoty nice se zvýší i hodnota priority, to ale v této verzi znamená, že proces bude běžet s prioritou nižší.

Skupiny procesů, řízení terminálů

- každý proces patří do skupiny procesů, tzv. *process group*
- každá skupina může mít vedoucí proces, tzv. *group leader*
- každý proces může mít řídící terminál (je to obvykle login terminál), tzv. *controlling terminal*
- speciální soubor `/dev/tty` je asociován s řídícím terminálem každého procesu
- každý terminál je asociován se skupinou procesů, tato skupina se nazývá řídící skupina (*controlling group*)
- kontrola jobů (*job control*) je mechanizmus, jak pozastavovat a probouzet skupiny procesů a řídit jejich přístup k terminálům
- *session* (relace) je kolekce skupin procesů vytvořená pro účely řízení jobů

- když se uživatel přihlásí do systému, je vytvořená nová relace, která se skládá z jedné skupiny procesů, ve které je jeden proces – ten který vykonává uživatelův shell. Tento proces je zároveň vedoucí této jediné skupiny procesů a také je vedoucí relace. V případě, že job control je povolen, každý příkaz nebo kolona příkazů vytvoří novou skupinu procesů, jeden z procesů v každé skupině se vždy stane vedoucím procesem dané skupiny. Jedna ze skupin může běžet na popředí, ostatní běží na pozadí. Signály, které jsou generované z klávesnice (tj. stiskem kombinace kláves, nemyslí se tím spuštění příkazu `kill`!), jsou zaslány pouze skupině, která běží na popředí.
- pokud job control není zapnut, znamená spustění příkazu na pozadí pouze to, že shell nečeká na jeho ukončení. Existuje pouze jedna skupina procesů,

signály z klávesnice se posílají všem procesům bežícím na popředí i na pozadí. Nelze přesouvat procesy z pozadí na popředí a naopak.

- když proces, který má kontrolní terminál, otevře soubor `/dev/tty`, tak se asociuje se svým kontrolním terminálem. Tj. pokud dva různé procesy z různých relací otevřou tento soubor, přistupují oba k různým terminálům.
- v bashi se skupina procesů (job) pozastaví pomocí `Ctrl-Z`, a rozběhne přes „`fg %N`" kde `N` je číslo jobu podle výpisu příkazu `jobs`. Více informací viz sekce „JOB CONTROL" v manuálové stránce pro bash.

Identifikace procesu

`pid_t` **`getpid`**`(void);`

- vrací proces ID volajícího procesu.

`pid_t` **`getpgrp`**`(void);`

- vrací ID skupiny procesů, do které patří volající proces.

`pid_t` **`getppid`**`(void);`

- vrací proces ID rodiče.

`pid_t` **`getsid`**`(pid_t` *`pid`*`);`

- vrací group ID vedoucího procesu session (sezení, terminálové relace) pro proces `pid` (0 znamená pro volající proces)

skupiny procesů umožňují posílat signály najednou celé skupině.

session (relace, sezení) je kolekce procesů vytvořená pro účely řízení prací (*job control*). Procesy sezení sdílejí jeden *řídící terminál*. Session zahrnuje jednu nebo více skupin procesů. Max. jedna skupina v rámci sezení běží na popředí (*foreground process group*) a má přístup k řídicímu terminálu pro vstup i výstup, ostatní běží na pozadí (*background process groups*) a mají k řídicímu terminálu přístup volitelně jen pro výstup nebo vůbec (nepovolená operace s terminálem pozastaví proces).

rodičovský proces: Každý proces (kromě `swapper`u, pid == 0) má rodiče, tj. proces, který ho stvořil voláním `fork`. Jestliže rodič skončí dříve než dítě, adoptivním rodičem se stává proces `init`, který se také postará o uklizení zombie po skončení procesu.

- Programatické zjišťování informací o ostatních procesech lze pomocí nestandardních rozhraní (např. knihovna `libproc` na Solarisu postavená na filesystému `procfs`, který je připojen pod `/proc`).

- Pozor na to, že zjišťovat že rodič skončil pomocí kontroly hodnoty vrácené z `getppid` na hodnotu 1 (což je běžně proces `init` resp. jeho ekvivalent), není portabilní.
  V různých virtualizovaných prostředích (PID namespaces v Linux, Zones v Solarisu) to nemusí platit. Viz příklad `session/getppid.c`.

Nastavení skupiny/sezení

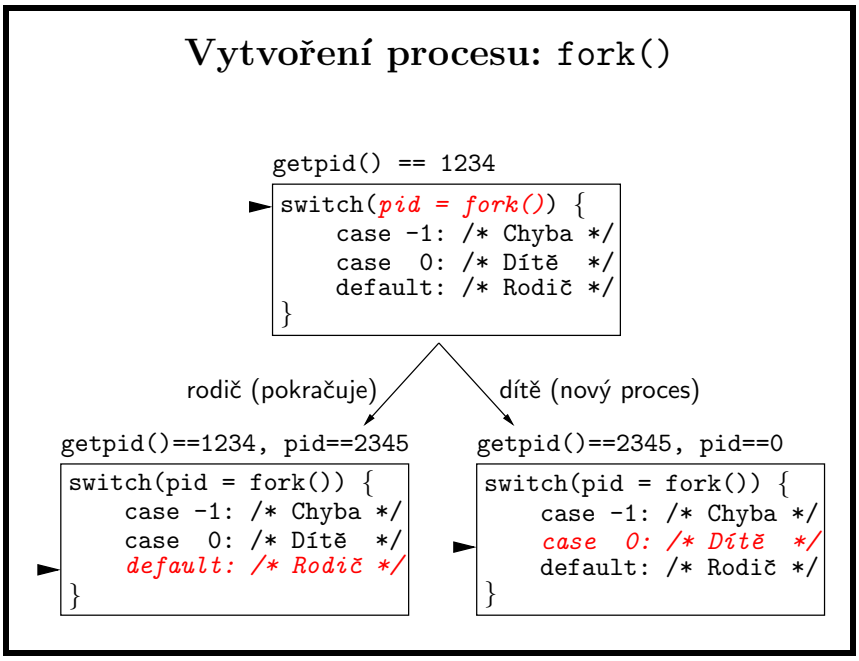
`int` **`setpgid`**`(pid_t` *`pid`*`, pid_t` *`pgid`*`);`

- nastaví process group ID procesu s pid na pgid.

`pid_t` **`setsid`**`(void);`

- vytvoří novou session, volající proces se stane vedoucím sezení i skupiny procesů.

- pro volání **setpgid** platí:
  1. pid == pgid : proces s *pid* se stane vedoucím skupiny procesů
  2. pid != pgid : proces s *pid* se stane členem skupiny procesů
- Proces, který ještě není vedoucím skupiny procesů, se může stát vedoucím sezení a zároveň skupiny procesů voláním `setsid`. Jestliže proces už je vedoucím skupiny, `setsid` selže, pak je třeba provést `fork` a `setsid` zavolat v synovském procesu. Takový proces nemá řídicí terminál, může ho získat otevřením terminálu, který ještě není řídicím terminálem sezení, když při `open` neuvede příznak `O_NOCTTY`, nebo jiným implementačně závislým způsobem.



- dítě je téměř přesnou kopií rodiče. Základní atributy, které se liší, jsou:
  - **PID** procesu a **parent PID**
  - pokud měl rodič více vláken, má syn pouze to, které `fork` zavolalo; o tom více na straně 223, až se dostaneme k vláknům. Tyto informace by každopádně měly být v manuálové stránce `fork(2)`.
  - účtovací časy se nastaví na 0
  - nedědí se nastavení `alarm` a zámky souborů
- co je důležité je, že tabulky deskriptorů jsou stejné v obou procesech, čímž více procesů může sdílet a posunovat společnou pozici v souboru, a také že masky signálů se nemění, viz strana 158.
- pro urychlení a menší spotřebu paměti se adresový prostor nekopíruje, ale používá se mechanizmus *copy-on-write*.
- je logické, že při úspěšném provedení volání otec dostane jako návratovou hodnotu volání `fork` PID syna a syn číslo 0; syn si totiž může velmi jednoduše svůj vlastní PID zjistit voláním `getpid`. Otec ale neumí jednoduše zjistit PID syna, navíc v situaci, pokud již dříve vytvořil syny jiné.
- příklad: `fork/fork.c`
- zajímavostí může být volání `vfork`, kterým se kdysi obcházel problém, kdy draze vytvořený proces byl okamžitě přepsán následným voláním `exec`. Tento problém byl dávno vyřešen již zmíněným použitím mechanizmu copy-on-write, ale příklad `fork/vfork.c` ukazuje, jak toto volání fungovalo.

Spuštění programu: `exec`

```
extern char **environ;
int execl(const char *path, const char *arg0, ... );
```

- spustí program definovaný celou cestou `path`, další argumenty se pak předají programu v parametrech `argc` a `argv` funkce `main`. Seznam argumentů je ukončen pomocí `(char *)0`, tj. `NULL`. `arg0` by měl obsahovat jméno programu (tj. ne celou cestu)
- úspěšné volání `execl` se nikdy nevrátí, protože spuštěný program zcela nahradí dosavadní adresový prostor procesu.
- program dědí proměnné prostředí, tj. obsah `environ`.
- handlery signálů se nahradí implicitní obsluhou
- zavřou se deskriptory souborů, které mají nastavený příznak `FD_CLOEXEC` (implicitně není nastaven).

- o signálech více na straně 147
- v `path` musí být celá (absolutní nebo relativní) cesta ke spustitelnému souboru. Obsah proměnné prostředí `PATH` se používá jen při voláních `execlp` a `execvp`, když argument `path` neobsahuje znak `'/'`.
- někdy se používá `argv[0]` různé od jména spustitelného souboru. Např. `login` vloží na začátek jména spouštěného shellu znak `'-'`. Shell podle toho pozná, že má fungovat jako login-shell, tj. například načíst `/etc/profile`.
- `exec` nepředá kontrolu načtenému programu v paměti přímo, ale přes dynamický linker (též nazývaný *loader*), poté co ho (= toho *loader*a) namapuje do adresového prostoru procesu. Loader následně namapuje potřebné dynamické objekty a teprve poté předá kontrolu aplikaci. Viz také strana 35. Pro jednoduché ověření stačí vytvořit program obsahující pouze třeba volání `open`. Tento program pak spusťte pomocí `truss(1)` takto: `truss ./a.out`. Uvidíte, která volání se použijí ještě předtím, než se zavolá `open`.
- `exec` nezmění hodnoty RUID a RGID. A pokud je to program s nastaveným SUID bitem, tak se EUID a uschované UID nastaví na UID majitele spustitelného souboru.
- dnešní unixové systémy umí spouštět i skripty, které začínají řádkem `#!`*`interpreter_path/interpreter_name [args]`*

Varianty služby `exec`

```
int execv(const char *path, char *const argv[]);
```

- obdoba `execl()`, ale argumenty jsou v poli `argv`, jehož poslední prvek je `NULL`.

```
int execle(const char *path, const char *arg0, ... ,
           char *const envp[]);
```

- obdoba `execl()`, ale místo `environ` se použije `envp`.

```
int execve(const char *path, char *const argv[],
           char *const envp[]);
```

- obdoba `execv()`, ale místo `environ` se použije `envp`.

```
int execlp(const char *file, const char *arg0, ...);
int execvp(const char *file, char *const argv[]);
```

- obdoby `execl()` a `execv()`, ale pro hledání spustitelného souboru se použije proměnná `PATH`.

- **l** = list (tj. lineární seznam argumentů), **v** = vector (tj. pole ukazatelů na řetězce), **e** = environment (tj. předávají se proměnné prostředí), **p** = PATH.
- kromě `execlp` a `execvp` je nutné jméno programu vždy zadávat celou cestu.
- všechny varianty kromě `execle` a `execve` předávají spouštěnému programu také své aktuální prostředí, tj. obsah pole `environ`.
- z mně neznámých historických důvodů neexistuje volání s **p** a **e** dohromady.
- příklad: `exec/exec-date.c`
- následující použití volání `execl` je špatně, protože mu chybí povinný parametr pro `argv[0]`:

  ```
  execl("/bin/ls", NULL);
  ```

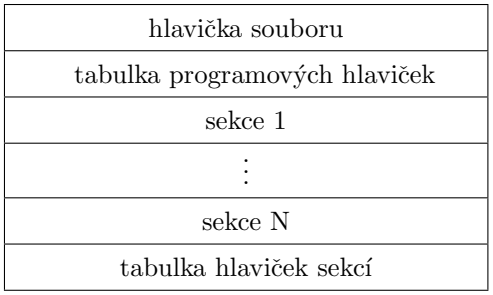
  Na některých systémech to má zajímavý efekt. Jelikož je `NULL` brán jako očekávaný `argv[0]`, další data na zásobníku jsou akceptována jako ukazatele na řetězce, dokud se nenalezne další `NULL`. V našem případě tak příkaz `ls` zkouší zobrazit informace o souborech, jako jejichž jména jsou použity řetězce nastavení proměnných prostředí z pole `environ`, o nichž již víme, že se voláním `execl` spouštěnému programu předávají:

  ```
  $ ./a.out
  : BLOCKSIZE=K: No such file or directory
  : FTP_PASSIVE_MODE=YES: No such file or directory
  ```

```
: HISTCONTROL=ignoredups: No such file or directory
: HISTSIZE=10000: No such file or directory
...
...
```

Formát spustitelného souboru

- **Common Object File Format (COFF)** – starší System V
- **Extensible Linking Format (ELF)** – nový v SVR4
- často se mluví o **a.out** formátu, protože tak se jmenuje (pokud není použit přepínač `-o`) výstup linkeru.
- Formát ELF:



- Standard nestanovuje formát spustitelného souboru. I když většina unixových systémů přešla na ELF, stále se najdou mainstreamové systémy používající jiné formáty. Např. Mac OS X (což je certifikovaný UNIX) používá tzv. *Mach-O* formát. Stejně jako ELF je strukturovaný, odlišuje se např. v tom, že za hlavičkou následují tzv. *load commands*, které se odkazují na sekce v další části. Za nimi následují samotné segmenty, které se skládají ze sekcí. Každý segment je úsek virtuální paměti, kterou dynamický linker mapuje do adresového prostoru procesu. Každá sekce obsahuje kód nebo data daného typu.
- Hlavička souboru (*ELF header*) obsahuje základní informace o souboru.
- Na Solarisu je k dispozici program `elfdump` umožňující zobrazení prakticky všech sekcí v ELF souboru a jejich obsahu v lidsky čitelné formě. V Linuxu lze použít program `readelf`, jeho funkcionalita je ovšem proti `elfdump` omezena.
- Tabulka programových hlaviček (*program header table*) je přítomna pouze u souborů obsahujících spustitelné programy, zobrazí se pomocí "`elfdump -p`". Obsahuje informaci o rozvržení virtuální paměti procesu.
- Sekce obsahují instrukce, data, tabulku symbolů, relokační data, apod.
- Tabulka hlaviček sekcí (*section header table*) obsahuje služební informace pro linker, "`elfdump -c`".

- Není to tak dávno co některé unixové systémy přešly na ELF. Např. OpenBSD přešlo z *a.out* formátu (což je další formát spustitelných souborů, viz a.out(5) man page na OpenBSD) na ELF ve verzi 3.4 v roce 2003. Proti ELFu je a.out mnohem jednodušší a méně obecný, hodí se tedy dobře na zkoumání fundamentálních principů zavádění spustitelných souborů do paměti. Základem hlavička (obsahující např. informace o tom zda spustitelný soubor vyžaduje služby run time linkeru nebo zda obsahuje Position Independent Code) a pak dalších nepovinných 6 sekcí (text, data, relokace v textovém segmentu, relokace v datovém segmentu, tabulka symbolů, tabulka stringů korespondujících s jednotlivými symboly).

- Dnes je naprosto běžné že virtuální adresy spustitelného souboru, namapovaných knihoven, zásobníku a heapu jsou náhodné, mění se při každém spuštění procesu. Tato technika (*Adress Space Layout Randomization*) slouží k znesnadnění útoků, které potřebují znát např. adresy symbolů v knihovně `libc` nebo zásobníku. Poprvé s touto ideou přišel Linux, OpenBSD byl první systém který ji začal používat pro všechny programy implicitně. Různé systémy aplikují tuto techniku s různými parametry a na různé části programu. Obecněji vzato lze vnášet náhodnost do dalších částí systému (např. ID procesů, iniciální sekvenční čísla pro TCP apod.).

Ukončení procesu

`void` **exit**`(int` *status*`);`

- ukončí proces s návratovým kódem `status`.
- nikdy se nevrátí na instrukci následující za voláním.

`pid_t` **wait**`(int *`*stat_loc*`);`

- počká, až skončí některý synovský proces, vrátí jeho PID a do `stat_loc` uloží návratový kód, který lze dále testovat:
  - `WIFEXITED(stat_val)` ... proces volal `exit()`
  - `WEXITSTATUS(stat_val)` ... argument `exit()`
  - `WIFSIGNALED(stat_val)` ... proces dostal signál
  - `WTERMSIG(stat_val)` ... číslo signálu
  - `WIFSTOPPED(stat_val)` ... proces pozastaven
  - `WSTOPSIG(stat_val)` ... číslo signálu

`pid_t` **waitpid**`(pid_t` *pid*`, int *`*stat_loc*`, int` *opts*`);`

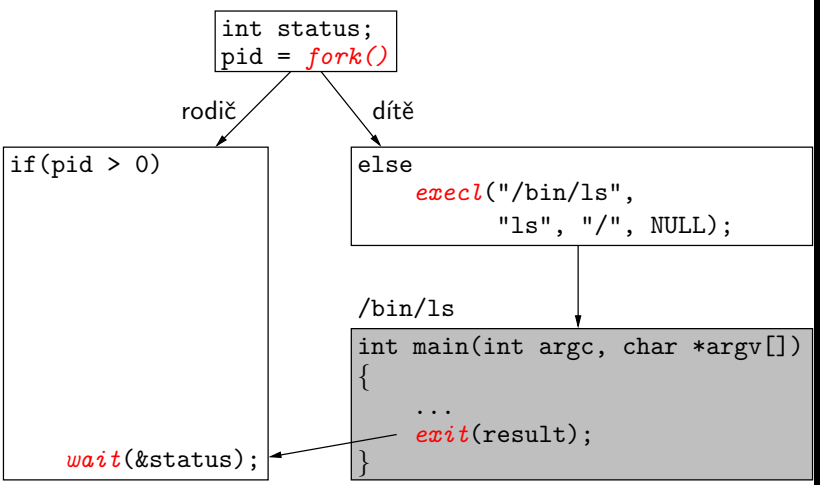
- čekání na jeden proces.

- `status_loc` roven `NULL` značí ignoraci statusu

- funkce `_exit` funguje jako `exit` s tím, že se neprovádí flush stdio streamů a nevolají se funkce nastavené pomocí `atexit`

- ve standardu je ještě `WIFCONTINUED(stat_val)`, což značí opětné rozběhnutí procesu po jeho předchozím zastavení, je to ale součást jistého rozšíření, které nemusí všechny systémy podporovat.
- poznámka – pozastavit proces lze pomocí „`kill -STOP <PID>`", rozběhnout pak přes „`kill -CONT <PID>`".
- `opts` ve `waitpid` jsou OR-kombinace následujících příznaků:
  - `WNOHANG` ... nečeká, pokud není status okamžitě k dispozici
  - `WUNTRACED` ... vrátí status i zastavených procesů, které ještě nebyly testovány po jejich zastavení. `WSTOPPED` může na některých systémech být synonymem pro `WUNTRACED`, není ale součástí normy.
  - `WCONTINUED` ... vrátí status i těch procesů, které ještě nebyly testovány od pokračování procesu po zastavení. Bez tohoto příznaku nejsou takové procesy reportovány. Součástí stejného rozšíření jako `WIFCONTINUED`.
  - pro `WUNTRACED` a `WCONTINUED` platí, že v přenositelných aplikacích byste tyto příznaky měli používat jen když je v `<unistd.h>` definováno makro `_POSIX_JOB_CONTROL`.
- `pid` ve `waitpid`:
  - `== -1` ... libovolné dítě
  - `> 0` ... jedno dítě
  - `== 0` ... dítě ve stejné skupině procesů jako volající proces
  - `< -1` ... dítě ve skupině `abs(pid)`
- rodič by měl vždy na své děti zavolat `wait` nebo `waitpid`, protože jinak se v systému hromadí *zombie* (ukončené procesy, které pouze čekají, až si rodič přečte jejich návratovou hodnotu). Hromadění těchto zombie procesů v tabulce procesů kernelu může skončit vyčerpáním veškeré volné paměti, takže pozor na to. Pokud takový rodič sám skončí, všechny jeho syny adoptuje process `init`, který se tak postará i o všechny zombie procesy.
- příklad: `wait/wait.c`

Příklad: spuštění programu a čekání



- toto je klasický způsob jak spustit nějaký program a po jeho skončení pokračovat. Rodič nemusí jen čekat na ukončení potomka, ale může vykonávat dál svůj kód.
- pozor na to, že ačkoli to na obrázku tak vypadá, návratová hodnota je pouze součást toho, co dostanete z volání `wait`. Na její získání je potřeba použít makra z předchozího slajdu.

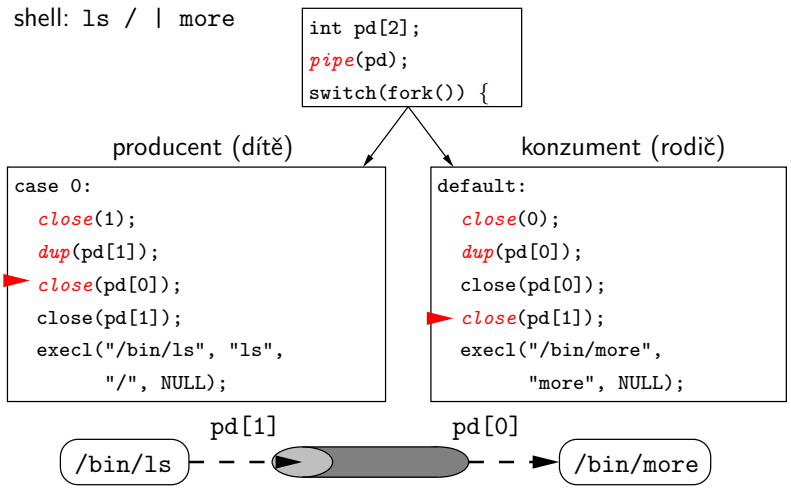
Roura: `pipe()`

`int` **`pipe`**`(int` *`fildes`*`[2]);`

- vytvoří rouru a dva deskriptory
  - `fildes[0]` ... čtení z roury
  - `fildes[1]` ... zápis do roury
- roura zajišťuje synchronizaci čtení a zápisu:
  - zapisující proces se zablokuje, když je roura plná,
  - čtoucí proces se zablokuje, když je roura prázdná.
- čtoucí proces přečte konec souboru (tj. `read()` vrátí `0`), pokud jsou uzavřeny všechny kopie `fildes[1]`.
- pojmenovaná roura (vytvořená voláním `mkfifo()`) funguje stejně, ale má přidělené jméno v systému souborů a mohou ji tedy používat libovolné procesy.

- nepojmenovanou rouru vytváří jeden proces a může ji předat pouze svým potomkům (pomocí deskriptorů zděděných při `fork`). Toto omezení se dá obejít pomocí předání otevřeného deskriptoru přes unix-domain socket.
- jestliže funkce `write` zapíše do roury nejvýše `PIPE_BUF` (systémová konstanta) bajtů, je zaručeno, že zápis bude atomický, tj. tato data nebudou proložena daty zapisovanými současně jinými procesy.
- v normě SUSv3 není specifikováno, zda `fildes[0]` je také otevřený pro zápis a zda `fildes[1]` je též otevřený pro čtení. Pozor na to, že například FreeBSD a Solaris mají roury obousměrné, ale Linux (když jsem se naposledy díval) ne. Je proto velmi vhodné počítat pouze s jednosměrnými rourami.
- **důležité:** pro čtení/zápis z/do roury platí stejné podmínky jako pro pojmenovanou rouru, jak jsou uvedeny na straně 98. To také znamená, že jediný způsob, jak čtenářovi roury "poslat" end-of-file je, že všichni zapisovatelé musí zavřít příslušný deskriptor pro zápis, případně všechny takové deskriptory, mají-li jich více.
- příklady: `pipe/broken-pipe.c`, `pipe/main.c`

Příklad: roura mezi dvěma procesy



- zavření zápisového deskriptoru `pd[1]` (ozn. ▷) v procesu konzumenta je nutné, protože jinak se na rouře nikdy nedetekuje konec souboru.
- čtecí deskriptor v procesu producenta `pd[0]` je také vhodné zavírat (ozn. ▷), protože když konzument předčasně skončí, dostane producent signál `SIGPIPE`. Kbyby deskriptor v producentovi nebyl zavřen, producent se nedozví, že konzument skončil, a po naplnění bufferu roury v jádru se zablokuje.
- pokud nemáme jistotu, že před voláním `pipe` byl otevřen deskriptor `0`, musíme v producentovi použít `dup2(pd[1], 1)`, protože `dup(pd[1])` by mohl vrátit deskriptor `0` místo požadovaného `1`. Také je třeba testovat, zda nenastala situace `pd[1] == 1`, abychom si nechtěně nezavřeli rouru. Podobně je třeba otestovat `pd[0] == 0` v konzumentovi.
- je lepší vytvářet rouru od syna k otci, protože typicky nejdřív skončí proces zapisující do roury, čtoucí proces přečte zbylá data, zpracuje je, něco vypíše a teprve pak skončí.

  Důvodem je to, že shell, který příslušnou kolonu příkazů spouští, čeká na ukončení otce a o jeho syny se nezajímá. Kdyby tedy směřovala roura od otce k synovi, otec jako producent dat skončí, shell vypíše prompt, ale pak ještě syn, který plní funkci konzumenta, může vypsat nějaký výstup. Možným řešením je čekání otce na skončení syna, jenže to se nedá zajistit, pokud otec provede exec.
- původní *Bourne shell* staví rouru tak, že poslední proces v rouře vytvoří předposlední jako svého syna, ten vytvoří předchozí proces a tak se postupuje až k začátku roury.

- v shellu `bash` jsou všechny procesy v rouře přímo potomky shellu (shell volá `fork` tolikrát, jak dlouhá je roura). Shell před vypsáním promptu čeká, až všechny procesy roury skončí.

Sdílená paměť – úvod

- pajpy a soubory jako metody meziprocesové komunikace vyžadují systémová volání
- výhoda: procesy nemohou poškodit adresový prostor jiného procesu
- nevýhoda: velká režie pro systémová volání, typicky **read**, **write**
- sdílená paměť je namapování části paměti do adresového prostoru více procesů
- odstranění nevýhody, ztráta dosavadní výhody
- synchronizace přístupu do sdílené paměti
  - System V semafory
  - POSIX semafory bez nutnosti systémového volání v běžném případě

- mapování souborů do paměti je jednou z implementací sdílené paměti. Pro popis úseku sdílené paměti používá soubor.
- takto implementovaná sdílená paměť je tedy pravidelně zapisována na disk
- pro sdílení bez režie zápisu změněných dat na disk je možné použít *memory based* filesystém, například *tmpfs* (Solaris, NetBSD – zde byl *tmpfs* napsán v roce 2005 jako součást *Summer of Code* sponzorovaného firmou Google, FreeBSD má podobnou vlastnost pod názvem *memory disk*). Jako tzv. *backing store* pro paměťové stránky patřící těmto filesystémům je obecně možné použít swap oblast na disku.
- pro synchronizaci přístupu do sdílené paměti se většinou používají semafory.

Mapování souborů do paměti (1)

`void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);`

- do paměťového prostoru procesu od adresy `addr` (`0` ... adresu přidělí jádro) namapuje úsek délky `len` začínající na pozici `off` souboru reprezentovaného deskriptorem `fildes`.
- vrací adresu namapovaného úseku nebo `MAP_FAILED`.
- v `prot` je OR-kombinace `PROT_READ` (lze číst), `PROT_WRITE` (lze zapisovat), `PROT_EXEC` (lze spouštět), nebo `PROT_NONE` (nelze k datům přistupovat).
- ve `flags` je OR-kombinace `MAP_PRIVATE` (změny jsou privátní pro proces, neukládají se do souboru), `MAP_SHARED` (změny se ukládají do souboru), `MAP_FIXED` (jádro nezmění addr).

- Příklady: `mmap/reverse.c`, `mmap/map-nocore.c`
- v příznacích musí být přítomen **právě jeden** z `MAP_PRIVATE` a `MAP_SHARED`
- mapování souborů do paměti je alternativou ke zpracování souborů pomocí `read`, `write`, `lseek`. Po namapování lze se souborem pracovat jako s datovou strukturou v paměti. Soubor se nekopíruje celý do paměti, alokují se pouze stránky na které se přistupuje. Pokud je potřeba stránku uvolnit, obsah se ukládá zpět do souboru (když je použit `MAP_SHARED` – tento způsob namapování je tedy ekvivalentní tomu, kdy program zapíše do stejného souboru pomocí `write(2)`) nebo do swapu – používá se mechanizmus copy-on-write (při `MAP_PRIVATE`).
- pro namapování souboru do paměti tedy potřebuji soubor nejdříve otevřít pomocí `open`. Mód v `prot` nemůže být "vyšší" než bylo specifikováno v módu pro `open`. Použití `MAP_FIXED` se nedoporučuje, protože to může být problém pro přenositelnost kódu.
- **varování:** toto se týká pouze `MAP_SHARED` – pokud je soubor jiným procesem zkrácen tak, že zkrácení se týká i právě namapované části, při přístupu do takové paměti je procesu zaslán signál `SIGBUS` resp. `SIGSEGV`. Řešením je použít mandatory locking, to ale není všude implementováno. V případě, že je namapovaná paměť použita jako parametr volání `write`, signál se nepošle, protože `write` vrátí -1 a `errno` je nastaveno na `EFAULT`.
- při použití `MAP_PRIVATE` vidím na Solarisu všechny změny provedené jinými procesy, které namapovaly sdíleně, až do té doby, kdy do stránky zapíšu

– v tom okamžiku se vytvoří kopie stránky a další takové změny již nevidím. Na FreeBSD tyto změny nevidím ani před zápisem. Viz specifikace: *„It is unspecified whether modifications to the underlying object done after the `MAP_PRIVATE` mapping is established are visible through the `MAP_PRIVATE` mapping."*

- hodnota `off+len` může překračovat aktuální velikost souboru, za konec souboru ale nelze zapisovat a soubor tak prodloužit - proces by obdržel signál `SIGBUS` resp. `SIGSEGV`, viz příklad `mmap/lseek.c`. Signál dostanu stejně tak v situaci, kdy do read-only namapovaného segmentu zkusím zapsat (je to logické, přiřazení nemá návratovou hodnotu kterou byste mohli otestovat).
- mapuje se vždy po celých stránkách, hodnoty `off` (a při `MAP_FIXED` i `addr`) musí být správně zarovnané. Poslední stránka je za koncem souboru doplněna nulami a tento úsek se nikdy nepřepisuje do souboru.
- Anonymní mapování lze sdílet mezi různými procesy pouze pomocí `fork()`. Jedinou alternativou je paměť sdílená pomocí systémového volání `shmat`.
- přístup do namapovaného úseku, ale za poslední existující stránku namapovaného objektu, způsobí signál `SIGBUS` resp. `SIGSEGV`. Neplatí to všude úplně přesně, viz příklad `mmap/sigbus.c`.
- Při použití `MAP_FIXED` namapování souboru nahradí případné předchozí mapování stránek v rozsahu. `addr` až `addr+len-1`, viz příklad `mmap/override.c`.
- existující rozšíření pro flagy (nejsou součástí SUSv3):
  - příznak `MAP_ANON` ve FreeBSD a Solarisu – vytvoření anonymního segmentu bez vazby na soubor, deskriptor musí být `-1`. Mapuje se tak anonymní objekt, který jak víme má místo fyzického uložení na swapu (tedy není trvalé). Linux má podobnou funkcionalitu přes `MAP_ANONYMOUS`. Tento příznak používají paměťové alokátory, které pracují s voláním `mmap`, viz také strana 118.
  - v IRIXu lze pomocí `MAP_AUTOGROW` automaticky zvětšit namapovaný objekt při přístupu za jeho stávající konec.
- běžným příkazem, který používá mapování souborů do paměti, je `cat(1)`. Číst z takové paměti je prostě rychlejší než opakovaně volat `read`, kde je nutné se pro každé takové volání přepnout z uživatelského režimu do režimu jádra a zpět.

Mapování souborů do paměti (2)

`int` **msync**`(void *`*addr*`, size_t` *len*`, int` *flags*`);`

- zapíše změněné stránky v úseku len bajtů od adresy addr do souboru. Hodnota flags je OR-kombinace
  - `MS_ASYNC` ... asynchronní zápis
  - `MS_SYNC` ... synchronní zápis
  - `MS_INVALIDATE` ... zrušit namapovaná data, která se liší od obsahu souboru

`int` **munmap**`(void *`*addr*`, size_t` *len*`);`

- zapíše změny, zruší mapování v délce `len` od adresy `addr`.

`int` **mprotect**`(void *`*addr*`, size_t` *len*`, int` *prot*`);`

- změní přístupová práva k namapovanému úseku souboru. Hodnoty prot jsou stejné jako u `mmap()`.

- uložení změn do souboru na disk je zaručené až po provedení `msync` nebo `munmap`, ale ostatní procesy, které mají soubor namapován, vidí změny hned.
- mapování paměti a nastavování přístupových práv používá např. knihovna Electric Fence, která slouží pro ladění chyb při práci s dynamickou pamětí.

Příklad: mapování souborů do paměti

```
int main(int argc, char *argv[])
{
    int fd, fsz; char *addr, *p1, *p2, c;

    fd = open(argv[1], O_RDWR);
    fsz = lseek(fd, 0, SEEK_END);
    p1 = addr = mmap(0, fsz, PROT_READ|PROT_WRITE,
                     MAP_SHARED, fd, 0);
    p2 = p1 + fsz - 1;
    while(p1<p2) {
        c = *p1; *p1++ = *p2; *p2-- = c;
    }
    munmap(addr, fsz);
    close(fd);
    return (0);
}
```

- Tento program otočí pořadí znaků v souboru (zapíše soubor od konce k začátku).
- Jedna z hlavních výhod sdílených segmentů je možnost pracovat s daty v souboru pomocí pointerové aritmetiky. Obecně je ale nutné dát pozor na zarovnání při dereferencích; např. na SPARCu dojde při takovém přístupu signál `SIGBUS`, viz příklad `mmap/aligned.c`.

Dynamický přístup ke knihovnám

`void *dlopen(const char *file, int mode);`

- zpřístupní knihovnu v souboru `file`, vrátí **handle** nebo `NULL`.
- v *mode* je OR-kombinace `RTLD_NOW` (okamžité relokace), `RTLD_LAZY` (odložené relokace), `RTLD_GLOBAL` (symboly budou globálně dostupné), `RTLD_LOCAL` (nebudou globálně dostupné).

`void *dlsym(void *handle, const char *name);`

- vrátí adresu symbolu zadaného jména z knihovny.

`int dlclose(void *handle);`

- ukončí přístup ke knihovně.

`char *dlerror(void);`

- vrátí textový popis chyby při práci s knihovnami.

- pomocí těchto funkcí lze implementovat dynamicky nahrávané plug-in moduly načítané aplikací podle potřeby (např. podle obsahu jejího konfiguračního souboru).
- dynamickým načítáním knihoven se také dá vyřešit situace, kdy potřebujeme využít několik knihoven, které definují symbol se stejným jménem. Jedna knihovna se přímo přilinkuje k programu, k ostatním se přistupuje pomocí `dlopen`.
- soubor musí být ve správném formátu (sdílená knihovna `.so` ve formátu *ELF* resp. formátu podporovaném na daném systému), například u `gcc` to znamená použít přepínač `-shared`, u `cc` na Solarisu (Sun Studio Compiler) je to přepínač `-G`. Na OS X (s formátem *Mach-O*) je to `-dynamiclib`, knihovny mají příponu `.dynlib`.
- pokud cesta obsahuje znak `/`, bere se podle tvaru jako globální nebo relativní. Pokud lomítko neobsahuje, použije se pro hledaní objektu defaultní nastavení dynamického linkeru, typicky `/lib` a `/usr/lib`, které se dá rozšířit pomocí proměnné `LD_LIBRARY_PATH`. Na její použití ovšem pozor, viz poznámky na straně 36.
- konstanty pro parametr *mode* funkce **dlopen**:
  - `RTLD_NOW` – všechny relokace (vyřešení všech odkazů) pro symboly nalezené v připojovaném objektu jsou provedeny okamžitě po natažení knihovny, aplikace má jistotu, že jsou všechny symboly přístupné

  - `RTLD_LAZY` – relokace mohou být odloženy až do chvíle použití symbolu. Co to v praxi znamená? Pokud otevřete objekt, který závisí na dalších objektech, dynamický linker tyto ostatní závislé objekty mapuje do paměti, až když jsou opravdu potřeba. Může se tak stát, že závislosti neexistují, ale volání `dlopen` stejně uspěje. U `RTLD_NOW` se závislé objekty mapují do paměti hned, a teprve pak `dlopen` vrátí příslušný handle. Na Solarisu můžete defaultní chování pro dynamický linker vynutit proměnnými prostředí `LD_BIND_NOW` a `LD_BIND_LAZY`. Při konfliktu nastavení má vždy přednost nastavení `NOW`, ať již je globální nebo jen v módu volání `dlopen` při mapování jednoho konkrétního objektu. Při spouštění aplikace jsou všechny závislé objekty defaultně mapované hned, ale je možné jednotlivé knihovny linkovat pro „lazy binding" pomocí `-z lazyload`, viz manuálové stránky pro `ld` a `ld.so.1`. Příklad: `dyn-lib/ld-lazy.c`.
  - `RTLD_GLOBAL` ... symboly z knihovny mohou být použity při zpracování relokací v ostatních knihovnách a jsou dostupné pomocí `dlopen(0, RTLD_GLOBAL)`. Toto je defaultní nastavení pro objekty mapované při spuštění programu. Pro `dlopen` je defaultní nastavení `RTLD_LOCAL`. To znamená, že je možné namapovat stejnou knihovnu několikrát a symboly se nebudou vzájemně překrývat. Pozor ale na to, když takové knihovny budou používat symboly jiného globální objektu - např. `errno` z `libc.so`. Takový symbol je dále společný pro všechny namapované objekty, včetně těch mapovaných pomocí `RTLD_LOCAL`.

- speciální handle `RTLD_NEXT` hledá symbol pouze v knihovnách nahraných po knihovně, ve které je volání `dlsym`. Hodí se pro předefinování existujících funkcí, pokud v redefinované funkci potřebujeme volat původní. Knihovna s novou funkcí se nahrává jako první (např. pomocí proměnné `LD_PRELOAD`), adresu původní funkce získá voláním `dlsym(RTLD_NEXT, fn_name)`. Příklad: `dyn-lib/rtld_next.c`.

- všechny tyto funkce jsou součástí dynamického linkeru, který má každá dynamicky slinkovaná aplikace namapovaný ve svém adresovém prostoru. Viz také strany 35 a 131.

Příklad: zpřístupnění knihovny

```
char *err;
void *handle;
double y, x = 1.3;
double (*fun)(double);
char *libname = "libm.so", *fn_name = "sin";

if ((handle = dlopen(libname, RTLD_NOW)) == NULL) {
        fprintf(stderr, "%s\n", dlerror()); exit(1);
}
fun = dlsym(handle, fn_name);
if ((err = dlerror()) != NULL)
        fprintf(stderr, "%s\n", err); exit(1);
y = fun(x);
dlclose(handle);
```

- zde se volá funkce `sin` z knihovny matematických funkcí `libm.so`.
- funkce `dlsym` vrátí adresu symbolu daného jména, ale vždy jako ukazatel na `void`, neprobíhá žádná typová kontrola ani není k dispozici žádná informace o typu symbolu. Ten, kdo tuto adresu používá, musí zajistit její správné přetypování.
- při použití knihoven v C++ je třeba si uvědomit, že C++ používá *name mangling*, tj. do jména funkce (metody) je zakódováno případné jméno třídy nebo namespace a typy parametrů.
- příklad: `dyn-lib/dlopen.c`

Obsah

- úvod, vývoj UNIXu a C, programátorské nástroje
- základní pojmy a konvence UNIXu a jeho API
- přístupová práva, periferní zařízení, systém souborů
- manipulace s procesy, spouštění programů
- **signály**
- synchronizace a komunikace procesů
- síťová komunikace
- vlákna, synchronizace vláken
- ??? - bude definováno později, podle toho kolik zbyde času

Signály

- informují proces o výskytu určité události
- na uživatelské úrovni zpřístupňují mechanizmy přerušení
- kategorie signálů:
  - **chybové události** generované běžícím procesem, např. pokus o přístup mimo přidělenou oblast paměti (`SIGSEGV`)
  - **asynchronní události** vznikající mimo proces, např. signál od jiného procesu, vypršení nastaveného času (`SIGALRM`), odpojení terminálu (`SIGHUP`), stisk `Ctrl-C` (`SIGINT`)
- nejjednodušší mechanizmus pro komunikaci mezi procesy – nesou pouze informaci o tom, že nastala nějaká událost.
- většinou se zpracovávají asynchronně – příchod signálu přeruší běh procesu a vyvolá se obslužná funkce, tzv. *handler signálu*

- se signálem není svázána žádná jiná informace než číslo signálu, pokud se nepoužije POSIX-1003.1b rozšíření (real-time), viz strana 155.
- po návratu z handleru (pokud k němu dojde) proces pokračuje od místa přerušení.
- historicky signály vznikly jako mechanizmus pro „násilné" ukončení procesu. Z toho vyplynul i název funkce `kill` pro poslání signálu.
- Signály lze zpacovávat i synchronně, viz `sigwait` na str. 159.

Poslání signálu

`int kill(pid_t pid, int sig);`

- pošle signál s číslem `sig` procesu (nebo skupině procesů) podle hodnoty `pid`:
  - `> 0` ... procesu s číslem pid
  - `== 0` ... všem procesům ve stejné skupině
  - `== -1` ... všem procesům, kromě systémových
  - `< -1` ... procesům ve skupině `abs(pid)`
- `sig == 0` znamená, že se pouze zkontroluje oprávnění poslat signál, ale žádný signál se nepošle.
- právo procesu poslat signál jinému procesu závisí na UID obou procesů.

- proces s EUID == 0 může poslat signál libovolnému procesu.
- ostatní procesy:
  - Linux, Solaris: RUID nebo EUID procesu, který poslal signál, se musí shodovat s reálným UID nebo saved set-user-ID cílového procesu.
  - FreeBSD: musí se shodovat EUID obou procesů.
  - IRIX: RUID nebo EUID procesu, který poslal signál, se musí shodovat s reálným nebo efektivním UID nebo saved set-user-ID cílového procesu.
- příklad (obsahuje i zachycení signálu, viz další slajdy) `signals/kill.c`
- nulový signál lze použít i pro jednoduchou kontrolu existence procesu s daným pid, viz `signals/check-existence.c`.

Ošetření signálů

- pokud proces neřekne jinak, provede se v závislosti na konkrétním signálu implicitní akce, tj. buď:
  - ukončení procesu (**exit**)
  - ukončení procesu plus coredump (**core**)
  - ignorování signálu (**ignore**)
  - pozastavení procesu (**stop**)
  - pokračování pozastaveného procesu (**continue**)
- proces také může nastavit ignorování signálu
- nebo signál ošetření uživatelsky definovanou funkcí (**handler**), po návratu z handleru proces pokračuje od místa přerušení

signály `SIGKILL` a `SIGSTOP` vždy vyvolají implicitní akci (zrušení, resp. pozastavení).

- vytvoření core dumpu znamená uložení kompletního obsahu paměti procesu do souboru, typicky se jménem `core`
- většina signálů implicitně ukončí proces, některé navíc vytvoří již zmiňovaný core dump, který je možné následně použít pro ladicí účely.
- důvod toho, proč při `exec` se všechny nastavené handlery signálů nahradí implicitní obsluhou (strana 131) je jasný – kód příslušných oblužných funkcí po volání `exec` přestane existovat.
- Čísla signálů a jejich jméno lze zjistit přepínače `-l` příkazu `kill`. Bez dalšího argumentu vypíše seznam všech signálů včetně čísla, při použití číselného argumentu vypíše název signálu:

```
$ kill -l SIGPIPE
13
```

- Implicitní nastavení akcí pro jednotlivé signály je většinou popsáno v manuálové stránce *signal* resp. *signal.h*.

Přehled signálů (1)

signály je možné logicky rozdělit do několika skupin...

detekované chyby:

`SIGBUS`	přístup k nedef. části paměťového objektu (core)
`SIGFPE`	chyba aritmetiky v pohyblivé čárce (core)
`SIGILL`	nepovolená instrukce (core)
`SIGPIPE`	zápis do roury, kterou nikdo nečte (exit)
`SIGSEGV`	použití nepovolené adresy v paměti (core)
`SIGSYS`	chybné systémové volání (core)
`SIGXCPU`	překročení časového limitu CPU (core)
`SIGXFSZ`	překročení limitu velikosti souboru (core)

- Generování těchto signálů vychází z chyb programu.
- Pro signály `SIGBUS`, `SIGFPE`, `SIGILL` a `SIGSEGV` není normou přesně definována příčina, ale obvykle jsou to chyby detekované hardwarem. Příklady `signals/sigsegv.c`, `signals/div-by-zero.c`.
- **Pro tyto čtyři signály také platí tato speciální pravidla** (podrobnosti viz kapitola *2.4 Signal Concepts* v normě SUSv3):
  - Pokud byly nastavené jako ignorované voláním `sigaction`, je chování programu po té, co je mu takový signál poslán, normou nedefinováno.
  - Návratová hodnota handleru není definována.
  - Následek situace, kdy jeden z těchto signálu je maskován v okamžiku jeho vygenerovaní je nedefinovaný.
- Jinými slovy – pokud je hardwarem detekovaná chyba reálná (signál není poslán přes `kill` a podobnými funkcemi), váš program se přes tuto chybu nemusí vůbec dostat. Není bezpečné chybu ignorovat, pokračovat v běhu po návratu z handleru nebo oddálit řešení pomocí zamaskování. Zachytit tyto signály lze, o tom, že by to bylo jinak, norma nemluví. Pokud tedy máte pro tyto signály handler, **je potřeba pořešit danou situaci jak uznáte za vhodné a pak ukončit program**. Můžete to vyzkoušet na příkladu `signals/catch-SIGSEGV.c`. Další informace včetně jiného příkladu na vyzkoušení je možné nalézt na straně 224.
- Poznámka: pokud je něco normou nedefinováno (*undefined*), obecně to znamená, že se neočekává, že by programátor potřeboval znát přesné chování v

takové situaci. Pokud je to potřeba, pravděpodobně je ve vašem programu něco špatně. Jako vždy, určite by se našly vyjímky potvrzující pravidlo.

Přehled signálů (2)

generované uživatelem nebo aplikací:

`SIGABRT`	ukončení procesu (core)
`SIGHUP`	odpojení terminálu (exit)
`SIGINT`	stisk speciální klávesy `Ctrl-C` (exit)
`SIGKILL`	zrušení procesu (exit, **nelze ošetřit ani ignorovat**)
`SIGQUIT`	stisk speciální klávesy `Ctrl-\` (core)
`SIGTERM`	zrušení procesu (exit)
`SIGUSR1`	uživatelsky definovaný signál 1 (exit)
`SIGUSR2`	uživatelsky definovaný signál 2 (exit)

- signál `SIGHUP` se často používá jako způsob, jak oznámit běžícímu démonu, že se změnil jeho konfigurační soubor a má si ho proto znovu načíst.
- `SIGINT` a `SIGQUIT` jsou obvykle generovány z terminálu (`Ctrl-C` a `Ctrl-\`) a lze je předefinovat příkazem `stty` nebo pomocí funkce `tcsetattr`. Pro to aby se mohl vygenerovat core file je zapotřebí mít to povolené v systémové konfiguraci a limitech, v shellu se toho dosáhne příkazem `ulimit`.
- vzhledem k tomu, že `SIGKILL` nelze zachytit, jej používejte jen v nutných případech; typickým případem je to, že běžící proces již nelze ukončit jiným signálem. Mnoho aplikací, hlavně démonů, spoléhá na to, že vynucené ukončení signálem je přes `SIGTERM`. Tento signál si zachytí a provede ukončovací operace – například uložení aktuální databáze na disk, smazání dočasných souborů apod. Používat rovnou `SIGKILL` proto, že proces to "vždycky zabije", je neznalost věci, která se vám může dost vymstít.
- ukázka na `SIGQUIT` na Solarisu:

```
$ sleep 10
^\Quit (core dumped)
$ mdb core
Loading modules: [ libc.so.1 ld.so.1 ]
> $c
libc.so.1`__nanosleep+0x15(8047900, 8047908)
```

```
libc.so.1‘sleep+0x35(a)
main+0xbc(2, 8047970, 804797c)
_start+0x7a(2, 8047a74, 8047a7a, 0, 8047a7d, 8047b91)
>
```

- `SIGTERM` je defaultní signál pro příkaz `kill(1)`
- `SIGUSR1` a `SIGUSR2` nejsou použity žádným systémovým voláním a jsou plně k dispozici uživateli

Přehled signálů (3)

job control:

`SIGCHLD`	změna stavu synovského procesu (ignore)
`SIGCONT`	pokračování pozastaveného procesu (continue)
`SIGSTOP`	pozastavení (stop, **nelze ošetřit ani ignorovat**)
`SIGTSTP`	pozastavení z terminálu `Ctrl-Z` (stop)
`SIGTTIN`	čtení z terminálu procesem na pozadí (stop)
`SIGTTOU`	zápis na terminál procesem na pozadí (stop)

- součástí nepovinného POSIX rozšíření, existují pouze když v `<unistd.h>` je definováno makro `_POSIX_JOB_CONTROL`

- platí, že nikdy není povoleno více procesům najednou číst z kontrolního terminálu, ale více procesů najednou může na terminál zapisovat.
- pozastavení skupiny procesů spustěné z terminálu (často přes `Ctrl-Z`) se provádí signálem `SIGTSTP`, ne `SIGSTOP`; aplikace tedy tento signál může zachytit.

Přehled signálů (4)

časovače:

`SIGALRM`	plánované časové přerušení (exit)
`SIGPROF`	vypršení profilujícího časovače (exit)
`SIGVTALRM`	vypršení virtuálního časovače (exit)

různé:

`SIGPOLL`	testovatelná událost (exit)
`SIGTRAP`	ladicí přerušení (core)
`SIGURG`	urgentní událost na soketu (ignore)

- `SIGALRM` a související funkce `alarm` se používají pro odměřování časových intervalů v uživatelském procesu (např. při implementaci timeoutů).

Nastavení obsluhy signálů

`int` **sigaction**`(int` *sig*`, const struct sigaction *`*act*`, struct sigaction *`*oact*`);`

- nastaví obsluhu signálu `sig` podle `act` a vrátí předchozí nastavení v `oact`.
- obsah struktury `sigaction`:
  - `void (*`*sa_handler*`)(int)` ... `SIG_DFL`, `SIG_IGN`, nebo adresa handleru
  - `sigset_t` *sa_mask* ... signály blokované v handleru, navíc je blokován signál `sig`
  - `int` *sa_flags* ... `SA_RESETHAND` (při vstupu do handleru nastavit `SIG_DFL`), `SA_RESTART` (restartovat přerušená systémová volání), `SA_NODEFER` (neblokovat signál `sig` během obsluhy)

- když je `act == NULL`, pouze se zjistí nastavení obsluhy, nemění se. Jestliže nás předchozí nastavení nezajímá, lze použít `oact == NULL`.
- pokud není nastaveno `SA_RESTART`, systémová volání aktivní v bodě příchodu signálu skončí s chybou `EINTR`. Restartování nemusí fungovat pro všechna systémová volání, např. na FreeBSD je `select` přerušen signálem vždy, i když je nastaveno `SA_RESTART` (pozn: nemusí být pravda u současných verzí, nezkoušel jsem to na nich).
- pozor na problém vzájemného vyloučení mezi procesem a handlerem, popř. mezi handlery pro různé signály. Jestliže je nastaveno `SA_NODEFER`, měl by být handler reentrantní.
- **v handleru signálu by se měly používat pouze funkce, které jsou pro takové použití bezpečné.** Musí buď být reentrantní, nebo je nutné zajistit, aby nepřišel signál v nevhodnou dobu (např. uvnitř funkce přijde signál, v jehož handleru se volá stejná funkce). Minimální skupina funkcí, které musí být tzv. ***async-signal-safe***, je vyjmenována v SUSv3 v sekci *System Interfaces: General Information ⇒ Signal Concepts ⇒ Signal Actions (2.4.3)*. Jednotlivé systémy mohou samozřejmě definovat i další takové funkce. Zda funkce je nebo není bezpečně použitelná v handleru by mělo být jasné z manuálové stránky; na Solarisu je tato informace vždy v sekci *ATTRIBUTES*.
- proč může nastat problém, když použijete v handleru signálu jinou funkci než async-signal-safe? Je to jednoduché – představte si, že když program vykonává funkci, která není async-signal-safe, přijde signál a v handleru se

vyvolá funkce stejná. Pokud funkce není pro takové použití napsaná, tak může dojít například k nekonzistenci statických dat ve funkci použitých, případně k uváznutí (dead lock) apod. Právě kvůli tomu, že v handlerech lze bezpečně použít jen podmnožinu existujících volání, se v handleru často pouze nastaví globální proměnná označující příchod příslušného signálu a ta se následně testuje, například v cyklu serveru, který vyřizuje požadavky nebo programu který zpracovává události. Zpomalení obsluhy signálu je minimální, protože funkce která čeká na další požadavek je typicky přerušitelná signálem a v takovém případě ihned vrací `EINTR`. Následuje kontrola globální(ch) proměnné(ých) na to, zda byl přijmut nějaký signál. Viz příklad `signals/event-loop.c`.

- funkce `sigaction` je obecnější než starší funkce `signal` a `sigset`, které zde ani nezmiňuji. Doporučuji používat pouze `sigaction`. Použití `signal` není například správné s vlákny, viz specifikace: "Use of this function is unspecified in a multi-threaded process."

- Pozor na to, že chování funkce `signal()` se může lišit podle systému. Například na FreeBSD zůstává handler stále nastaven, na Solarisu je z důvodu zachování zpětné kompatibility handler před jeho vyvoláním resetován na `SIG_DFL`. Funkce příznaku `SA_RESETHAND` je právě to, aby bylo možné simulovat původní chování funkce `signal()`, které bývá implementováno pomocí systémového volání `sigaction()`. Příklad na rozdíl mezi funkcemi `signal()` a `sigaction()`: `signal/signal-vs-sigaction.c`.

- (nebudete nejspíš potřebovat) pro výskok z handleru signálu jinam než na místo vzniku signálu se dají použít funkce `sigsetjmp` a `siglongjmp`. Pozor na to, že v tomto případě si musíme být jisti, že v okamžiku příchodu signálu není program uvnitř ne-reentrantní funkce. Výskokem z handleru do hlavního programu není vykonávání takové funkce ukončeno a mohou nastat stejné problémy jako při volání ne-reentrantní funkce přímo z handleru.

- pokud systém podporuje část POSIX.1b zvanou *Realtime Signals Extension* (RTS), je možné příznakem `SA_SIGINFO` toto rozšíření použít. V tom případě se použije jiná položka struktury `sigaction` pro ukazatel na handler, a tou je `sa_sigaction`. Tento handler má již 3 parametry a je možné zjistit například to, který proces signál poslal, pod jakým uživatelem proces běžel a mnoho dalších informací. Zájemce odkazuji na manuálovou stránku `signal.h(3HEAD)` na Solarisu, specifikaci tohoto hlavičkového souboru přímo v SUSv3 nebo na knihu [POSIX.4], strana 5. Další informace jsou také na stranách 15 a 159. Příklady: `signals/siginfo.c`, `signals/sigqueue.c`.

Příklad: časově omezený vstup

```
void handler(int sig)
{ write(2," !!! TIMEOUT !!! \n", 17); }

int main(void)
{
    char buf[1024]; struct sigaction act; int sz;
    act.sa_handler = handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGALRM, &act, NULL);
    alarm(5);
    sz = read(0, buf, 1024);
    alarm(0);
    if (sz > 0)
        write(1, buf, sz);
    return (0);
}
```

- lze používat i časovače s jemnějším rozlišením než 1 s. Nastavují a testují se funkcemi `setitimer` a `getitimer`. Při vypršení posílají signály procesu, který časovače nastavil podle prvního argumentu *which*:
  - `ITIMER_REAL` ... měří reálný čas, posílá `SIGALRM`
  - `ITIMER_VIRTUAL` ... měří virtuální čas (pouze čas, kdy proces běží), posílá `SIGVTALRM`
  - `ITIMER_PROF` ... měří virtuální čas a čas, kdy systém běží na konto procesu, posílá `SIGPROF`
- pozn: všiměte si, že ačkoliv by to svádělo použít pro tisk hlášky v příkladu funkci `fprintf` apod., nemusel by to být dobrý nápad, protože nemusí být bezpečná pro použití v handleru signálu, viz strana 154.
- příklad: `signals/alarm.c`

Blokování signálů

- blokované signály budou procesu doručeny a zpracovány až po odblokování.

```
int sigprocmask(int how, const sigset_t *set,
                sigset_t *oset);
```

- nastaví masku blokovaných signálů a vrátí starou masku.
- *how* – `SIG_BLOCK` pro přidání signálů co se mají blokovat, pro odebrání `SIG_UNBLOCK`, pro kompletní změnu masky `SIG_SETMASK`
- pro manipulaci s maskou signálů slouží funkce: `sigaddset()`, `sigdelset()`, `sigemptyset()`, `sigfillset()`, `sigismember()`

```
int sigpending(sigset_t *set);
```

- vrátí čekající zablokované signály.

- Je rozdíl mezi ignorováním a blokováním signálu. Ignorovaný signál jádro zahodí a proces ho nedostane, blokovaný signál proces dostane po jeho odblokování.
- Závisí na implementaci, zda při vícenásobném doručení stejného signálu procesu, který má tento signál zablokovaný, bude signál po odblokování ošetřen jednou nebo vícekrát.
- V případě rozšíření signálů z POSIX.4 (strana 155), tj. použití příznaku `SA_SIGINFO`, jsou signály doručované přes frontu a tedy se žádný násobný výskyt stejného signálu neztratí.
- Argument `oset` pro získání původní masky může být `NULL`, stejně jako může být nastavený na `NULL` i parametr druhý, tj. `set`. Ve speciálním případě, kdy jsou oba parametry nulové, funkce `sigprocmask` nedělá nic.

Příklad: blokování signálů

```
sigset_t sigs, osigs; struct sigaction sa;
sigfillset(&sigs); sigprocmask(SIG_BLOCK, &sigs, &osigs);
switch(cpid = fork()) {
    case -1: /* Chyba */
        sigprocmask(SIG_SETMASK, &osigs, NULL);
        ...
    case 0: /* Synovský proces */
        sa.sa_handler = h_cld; sigemptyset(&sa.sa_mask);
        sa.sa_flags = 0;
        sigaction(SIGINT, &sa, NULL);
        sigprocmask(SIG_SETMASK, &osigs, NULL);
        ...
    default: /* Rodičovský proces */
        sigprocmask(SIG_SETMASK, &osigs, NULL);
        ...
}
```

- příklad ukazuje situaci, kdy proces vytváří potomky pomocí `fork` a je potřeba, aby potomci měli jiný handler signálů než rodičovský proces. Funguje to proto, že volání `fork` nemění masky signálů, viz strana 130.
- pro jednoduchost v příkladu blokuji všechny signály, i když na stranách 150 a 224 je vysvětleno, proč to není správné použití maskování.
- blokování je vhodné použít tam, kde ošetření přerušení uprostřed posloupnosti operací by bylo příliš složité, nebo kde korektní ošetření není jinak možné. V uvedeném příkladě by bez blokování signálů mohl synovský proces dostat signál dřív, než stihne změnit handler.
- další příklad je proces, který při vypršení timeoutu přeruší prováděnou posloupnost operací voláním `siglongjmp` zevnitř handleru signálu. Je potřeba zablokovat signál `SIGALRM` během provádění atomických podposloupností (tj. takových, které se musí provést buď celé, nebo vůbec ne).

Čekání na signál

`int` **`pause`**`(void);`

- pozastaví volající proces do příchodu (neblokovaného) signálu

`int` **`sigsuspend`**`(const sigset_t *`*`sigmask`*`);`

- jako `pause()`, ale navíc po dobu čekání masku blokovaných signálů změní na `sigmask`

`int` **`sigwait`**`(const sigset_t *`*`set`*`, int *`*`sig`*`);`

- čeká na příchod signálu z množiny `set` (tyto signály musí být předtím zablokované), číslo signálu vrátí v `sig`. Vrací 0 nebo číslo chyby.
- nevolá se handler signálu (to ale není v normě jednoznačně definováno)

- Neblokovaný signál v `pause` a `sigsuspend` vyvolá handler a po jeho skončení program opustí signál zachycující funkci a pokračuje dále. Pokud má ale signál proces ukončit (např. nemaskovaný `SIGTERM` bez handleru), stane se tak.
- Pomocí těchto funkcí a blokování signálů se implementuje synchronní obsluha signálů. Proces nejprve zablokuje signály, které ho zajímají, a pak na ně ve vhodných chvílích buď čeká, nebo jen testuje (pomocí `sigpending`), zda signál přišel, a pokud ne, pokračuje dál.
- Funkce `sigwait` byla přidána s POSIX-1003.1c rozšířením (vlákna) a je to "jediný" správný způsob, jak obsluhovat asynchronní signály v multi-vláknové aplikaci. To že byla přidána s vlákny je potvrzeno i tím, že v případě problémů vrací přímo číslo chyby.
- Existují i příbuzné podobně se jmenující funkce `sigwaitinfo` a `sigtimedwait`, definované s rozšířením POSIX-1003.1**b** (real-time). Fungují na podobném principu, ale na rozdíl od `sigwait` pracují s `errno` a je z nich možné získat více informací díky struktuře `siginfo_t`, viz strana 155. Je tedy možné je použít místo `sigwait`.
- Příklad (signál se použije pro synchronizaci dvou procesů komunikujících přes sdílenou paměť): `signals/sigwait.c`
- **Pozor** na to, že byste neměli tento způsob obsluhy signálů používat pro signály synchronní jako jsou `SIGSEGV`, `SIGILL`, apod. Více se dočtete na stranách 150 a 224.

Obsah

- úvod, vývoj UNIXu a C, programátorské nástroje
- základní pojmy a konvence UNIXu a jeho API
- přístupová práva, periferní zařízení, systém souborů
- manipulace s procesy, spouštění programů
- signály
- **synchronizace a komunikace procesů**
- síťová komunikace
- vlákna, synchronizace vláken
- ??? - bude definováno později, podle toho kolik zbyde času

Problém: konflikt při sdílení dat

- máme strukturu `struct { int a, b; } shared;`
- ```
  for( ; ; ) {
      /* neatomická operace */
      a = shared.a; b = shared.b;
      if (a != b) printf("NEKONZISTENTNÍ STAV");
      /* neatomická operace */
      shared.a = val; shared.b = val;
  }
  ```
- jestliže tento cyklus spustíme ve dvou různých procesech (nebo vláknech), které obě sdílejí stejnou strukturu shared a mají různé hodnoty `val`, bude docházet ke konfliktům.
- příčina: operace na zvýrazněných řádcích nejsou atomické.

- ani operace, kterou lze v C zapsat jedním příkazem, nemusí být atomická. Př.: na RISCových procesorech se příkaz `a++` typicky přeloží jako sekvence:

  ```
  load reg,[a]
  inc reg
  store [a],reg
  ```

  a to z toho důvodu, že na této architektuře nelze inkrementovat číslo přímo v paměti. Pro tyto případy má například Solaris sadu funkcí `atomic_add(3c)`, jejichž použití je mnohem rychlejší než klasické zamykací mechanismy. Více viz strana 235.

- obecně obdobný problém nastává, když více procesů sdílí nějaký systémový zdroj.
- příklad: `race/race.c`
  ```

Scénář konfliktu

Procesy **A**(`val==1`) a **B**(`val==2`)

		`a`	`b`
1.	počáteční stav struktury	?	?
2.	proces **A** zapíše do položky `a`	1	?
3.	proces **B** zapíše do položky `a`	2	?
4.	proces **B** zapíše do položky `b`	2	2
5.	proces **A** zapíše do položky `b`	2	1
6.	struktura je v nekonzistentním stavu a jeden z procesů to zjistí.		

- další možnost:
  1. struktura je v konzistentním stavu, např. `(1, 1)`
  2. proces `B` zapíše `2` do položky `a`
  3. proces `A` přečte hodnotu struktury `(2, 1)` dříve, než proces `B` stihne zapsat položku `b`
- pozor na to, že synchronizační problémy se často projeví až na víceprocesorovém stroji, nebo na více procesorech než kolik jich máte při vývoji k dispozici apod. Je třeba na to myslet při testování.

Řešení: vzájemné vyloučení procesů

- je potřeba zajistit atomicitu operací nad strukturou, tzn. jeden proces provádí modifikaci a dokud neuvede strukturu do konzistentního stavu, druhý proces s ní nemůže manipulovat.

Procesy **A**`(val==1)` a **B**`(val==2)`		`a`	`b`
1.	počáteční stav struktury	?	?
2.	proces **A** zapíše do položky `a`	1	?
3.	proces **B** musí čekat	1	?
4.	proces **A** zapíše do položky `b`	1	1
5.	proces **B** zapíše do položky `a`	2	1
6.	proces **B** zapíše do položky `b`	2	2
7.	Struktura je v konzistentním stavu.		

- je třeba zajistit vzájemné vyloučení i při čtení, aby čtoucí proces nepřečetl nekonzistentní obsah struktury uprostřed změn. Při zápisu je nutné vyloučit všechny ostatní procesy, ale při čtení stačí vyloučit jen zápis, současné čtení více procesy nevadí.
- ***kritická sekce*** je kus kódu, který by měl provádět pouze jeden proces nebo vlákno, jinak může dojít k nekonzistencím; například špatně pospojovaný vázaný seznam, neodpovídající si indexy v databázi apod. Je možné to definovat i tak, že kritická sekce je kód, který přistupuje k nebo modifikuje blíže neurčený zdroj sdílený více procesy nebo vlákny a proto je nutné přístup k takovému kódu synchronizovat. Kritická sekce by měla být co nejkratší, aby ostatní procesy (vlákna) žádající o vstup do této sekce čekaly co nejkratší možnou dobu. Druhá definice je o něco obecnější, protože se do ní vejdou i situace, kdy pouze jeden proces nebo vlákno může stav měnit, ale pokud se tak neděje, může více procesů či vláken daný stav číst, to znamená že kód kritické sekce za jistých přesně definovaných podmínek může vykonávat více procesů nebo vláken najednou.

Problém: konflikt zapisovatelů a čtenářů

- několik běžících procesů zapisuje protokol o své činnosti do společného log-souboru. Nový záznam je připojen vždy na konec souboru.
- pokud zápis záznamu není proveden atomickou operací, může dojít k promíchání více současně zapisovaných záznamů.
- zapisovat smí vždy pouze jeden proces.
- další procesy čtou data z log-souboru.
- při přečtení právě zapisovaného záznamu obdržíme nesprávná (neúplná) data.
- během operace zápisu ze souboru nelze číst. Když nikdo nezapisuje, může více procesů číst současně.

- povoleny jsou tedy 2 situace: jeden zapisovatel nebo více čtenářů
- na lokálním disku lze pro synchronizaci zapisovatelů použít řešení pomocí příznaku `O_APPEND`, které ale nebude fungovat např. na vzdáleném disku přístupném přes NFS nebo v případě, že zápis jedné logovací zprávy je proveden více než jedním voláním `write()`. Navíc to neřeší synchronizaci čtenářů – lze číst i v průběhu zápisu.

Řešení: zamykání souborů

- zapisující proces zamkne soubor pro zápis. Ostatní procesy (zapisovatelé i čtenáři) se souborem nemohou pracovat a musí čekat na odemčení zámku.
- čtoucí proces zamkne soubor pro čtení. Zapisovatelé musí čekat na odemčení zámku, ale ostatní čtenáři mohou také zamknout soubor pro čtení a číst data.
- v jednom okamžiku může být na souboru aktivní nejvýše jeden zámek pro zápis nebo libovolně mnoho zámků pro čtení, ale ne oba typy zámků současně.
- z důvodu efektivity by každý proces měl držet zámek co nejkratší dobu a pokud možno nezamykat celý soubor, ale jen úsek, se kterým pracuje. Preferované je pasivní čekání, aktivní čekání je vhodné jen na velmi krátkou dobu.

- dva způsoby čekání:

  aktivní (busy waiting) – proces v cyklu testuje podmínku, na kterou čeká, dokud není splněna

  pasivní – proces se zaregistruje v jádru jako čekající na podmínku a pak se uspí, jádro ho probudí, když dojde ke splnění podmínky

- aktivní čekání je ospravedlnitelné pouze ve speciálních situacích. Na takové situace asi v zápočtovém programu nenarazíte a určitě ne v příkladu u zkoušky. **Použití aktivního čekání v takovém případě automaticky znamená nesplnění zadání písemky.**

Synchronizační mechanismy

- teoretické řešení – algoritmy vzájemného vyloučení (Dekker 1965, Peterson 1981)
- zákaz přerušení (na 1 CPU stroji), speciální *test-and-set* instrukce
- **lock-soubory**
- nástroje poskytované operačním systémem
  - **semafory** (součást System V IPC)
  - **zámky pro soubory** (`fcntl()`, `flock()`)
  - synchronizace vláken: **mutexy** (ohraničují kritické sekce, pouze jedno vlákno může držet mutex), **podmínkové proměnné** (zablokují vlákno, dokud jiné vlákno nesignalizuje změnu podmínky), **read-write zámky** (sdílené a exkluzivní zámky, podobně jako pro soubory)

- Dekker i Peterson potřebují k dosažení požadavaného výsledku **pouze sdílenou paměť**, tj. několik proměnných sdílených oběma procesy.
- Dekkerovo řešení se udává jako první řešení problému vzájemného vyloučení dvou procesů, aniž by bylo nutné aplikovat naivní algoritmus striktního střídání, tj. pokud druhý proces nevykazoval zájem o vstup do kritické sekce, mohl tam první (a naopak) proces vstoupit tolikrát za sebou, kolikrát chtěl. Dekkerovo řešení není vůbec triviální, porovnejte s o 16 let mladším řešením Petersonovým, například na `en.wikipedia.org` (hledejte "Dekker's algorithm", "Peterson's algorithm")
- my se nebudeme zabývat teoretickými algoritmy vzájemného vyloučení ani nebudeme popisovat hardwarové mechanismy používané jádrem. Zaměříme se pouze na použití lock souborů (které využívají atomičnosti některých souborových operací) a speciálních synchronizačních nástrojů poskytovaných jádrem.
- podmínkové proměnné = *conditional variables*

Lock-soubory

- pro každý sdílený zdroj existuje dohodnuté jméno souboru. Zamčení zdroje se provede vytvořením souboru, odemčení smazáním souboru. Každý proces musí otestovat, zda soubor existuje, a pokud ano, tak počkat.

```
void lock(char *lockfile) {
    while( (fd = open(lockfile,
                      O_RDWR|O_CREAT|O_EXCL, 0600)) == -1)
        sleep(1);  /* Čekáme ve smyčce na odemčení */
    close(fd);
}

void unlock(char *lockfile) {
    unlink(lockfile);
}
```

- klíčem k úspěchu je samozřejmě použití flagu `O_EXCL`
- příklad: `file-locking/lock-unlock.c`, a použijte společně se shellovým skriptem `file-locking/run.sh`.
- při havárii procesu nedojde ke zrušení případných zámků a ostatní procesy by čekaly věčně. Proto je vhodné si do lock-souboru poznamenat PID procesu, který zámek vytvořil. Proces, který čeká na odemčení, může testovat, zda proces, kterému zámek patří, stále běží. Když ne, lze zámek zrušit a znovu zkusit vytvořit. User level příkaz který toto umí a dovoluje používat lock soubory z shellových skriptů je například **shlock**(1) (na FreeBSD v `/usr/ports/sysutils/shlock`), teoreticky by však mohl způsobit situaci z následujícího odstavce.
- **pozor:** jestliže více procesů najednou zjistí, že proces držící zámek už neexistuje, může dojít k následující chybě. První proces smaže zámek a znovu ho vytvoří se svým PID. Další proces udělá totéž, protože operace přečtení obsahu souboru a jeho následného smazání není atomická. Teď si ale oba procesy myslí, že získaly zámek!
- **problém:** funkce `lock()` obsahuje aktivní čekání na uvolnění zámku. Lze řešit např. tak, že proces, který získá zámek, otevře pro zápis pojmenovanou rouru. Čekající procesy se uspí tím, že zkusí číst z roury. Součástí `unlock()` bude i zavření roury a tím uvolnění čekajících procesů. **Upozorňuji na to, že zkouškové příklady pro zadání obsahující jakoukoli synchronizaci nejsou akceptovány jako správné, pokud je jakkoli použito aktivní čekání, včetně ukázaného řešení typu `sleep(1)` volaného ve smyčce.**

- lock soubory se v praxi většinou používají pouze pro situace, kdy se například hlídá násobné spuštění téže aplikace. **Ze zkušenosti raději opět upozorňuji, že pokud je bude student na zkoušce používat například pro synchronizaci vláken, neuspěje.**

Zamykání souborů: `fcntl()`

`int` **`fcntl`**`(int` *`fildes`*`, int` *`cmd`*`, ...);`

- k nastavení zámků pro soubor `fildes` se používá `cmd`:
  - `F_GETLK` ... vezme popis zámku z třetího argumentu a nahradí ho popisem existujícího zámku, který s ním koliduje
  - `F_SETLK` ... nastaví nebo zruší zámek popsaný třetím argumentem, pokud nelze zámek nastavit, ihned vrací -1
  - `F_SETLKW` ... jako `F_SETLK`, ale uspí proces, dokud není možné nastavit zámek (`W` znamená "wait")
- třetí argument obsahuje popis zámku a je typu ukazatel na `struct flock`

- zamykání souborů sdílených přes NFS zajišťuje démon `lockd`.
- zámky jsou obecně dvou typů:

  advisory locks – pro správnou funkci musí všechny procesy pracující se zámčenými soubory kontrolovat zámky před čtením nebo zápisem souboru; jsou více používané

  mandatory locks – jestliže je na souboru zámek, budou čtecí a zápisové operace se souborem automaticky zablokovány, tj. zámek se uplatní i v procesech, které ho explicitně nekontrolují
  - nedoporučují se, ne vždy fungují (např. `lockd` implementuje pouze advisory locking)
  - pro určitý soubor se zapne nastavením bitu SGID a zrušením práva spuštění pro skupinu (tj. nastavení, které jinak nemá velký smysl - mít set GID executable bit na souboru, který není spustitelný). Funguje to tak, že jeden proces si vezme zámek na daném souboru (např. pomocí `fcntl`). Další procesy pak nemusí explicitně zamykat, protože každou operaci `open/read/write` s daným souborem kontroluje kernel proti zámkům na souboru a vynutí čekání až do chvíle kdy je zámek explicitně první procesem uvolněn.
    Systém, který mandatory locking implementuje, je například Solaris

nebo Linux, FreeBSD tuto vlastnost naopak nepodporuje. Manuálová stránka fcntl(2) na Linuxu (2013) nedoporučuje mandatory locking používat, protože Linuxová implementace obsahuje chyby které vedou k souběhu a mandatory zamykání nedokáže tedy zajistit konzistenci.

- Je důležité si uvědomit, že po skončení procesu se všechny zámky, které držel uvolní.

Zamykání souborů: `struct flock`

- `l_type` ... typ zámku
  - `F_RDLCK` ... sdílený zámek (pro čtení), více procesů
  - `F_WRLCK` ... exkluzivní zámek (pro zápis), jeden proces
  - `F_UNLCK` ... odemčení
- `l_whence` ... jako u `lseek()`, tj. `SEEK_SET`, `SEEK_CUR`, `SEEK_END`
- `l_start` ... začátek zamykaného úseku vzhledem k `l_whence`
- `l_len` ... délka úseku v bajtech, 0 znamená do konce souboru
- `l_pid` ... PID procesu držícího zámek, používá se jen pro `F_GETLK` při návratu

- soubory se dají zamykat po částech a dá se zjistit, který proces drží zámek. Při ukončení procesu se automaticky uvolní všechny jeho zámky.
- pokud při použití `F_GETLK` není příslušná část souboru zamčená, je struktura `flock` vrácena bez změny kromě první položky, která je nastavena na `F_UNLCK`.
- příklad: `file-locking/fcntl-locking.c`
- příklad na fcntl (je to pomocí fcntl „opravená" verze dřívejšího příkladu `race/race.c` ze strany 161): `race/fcntl-fixed-race.c`.
- zamykání přes `fcntl` i `lockf` má jednu důležitou vlastnost, kterou výstižně popisuje například manuálová stránka pro `fcntl` v systému FreeBSD:

  This interface follows the completely stupid semantics of System V and IEEE Std 1003.1-1988 ("POSIX.1") that require that all locks associated with a file for a given process are removed when any file descriptor for that file is closed by that process. This semantic means that applications must be aware of any

files that a subroutine library may access. For example if an application for updating the password file locks the password file database while making the update, and then calls ***getpwnam****(3) to retrieve a record, the lock will be lost because* ***getpwnam****(3) opens, reads, and closes the password database.*

- Funkce `lockf` (součástí SUSv3) je jednodušší variantou `fcntl`, specifikuje se pouze jak zamknout a kolik bajtů od současné pozice v souboru. Velmi často je implementována jako wrapper kolem `fcntl`.
- Příklad `file-locking/lockf.c` demonstruje jak funguje mandatory locking a ukazuje použití funkce `lockf`.

Deadlock (aka uváznutí)

- máme dva sdílené zdroje `res1` a `res2` chráněné zámky `lck1` a `lck2`. Procesy `p1` a `p2` chtějí každý výlučný přístup k oběma zdrojům.

p1	p2
`lock(lck1); /* OK */`	`lock(lck2); /* OK */`
`lock(lck2); /* Čeká na p2 */`	`lock(lck1); /* Čeká na p1 */`
Deadlock	
`use(res1, res2);`	`use(res1, res2);`
`unlock(lck2);`	`unlock(lck2);`
`unlock(lck1);`	`unlock(lck1);`

- **pozor na pořadí zamykání!**

- obecně deadlock vznikne, jestliže proces čeká na událost, která nemůže nastat. Zde např. na sebe dva procesy čekají navzájem, až ten druhý uvolní zámek, ale k tomu nikdy nedojde. Další možností je deadlock jednoho procesu, který čte z roury a předtím zapomněl uzavřít zápisový konec roury. Jestliže rouru nemá už nikdo další otevřenou, pokus o čtení ze zablokuje, protože nejsou zavřeny všechny kopie zápisového deskriptoru a tedy nenastane konec souboru na rouře, ale čtoucí proces svůj zápisový deskriptor nemůže zavřít, protože čeká ve volání `read`:

```
int main(void)
{
        int c, fd;
        mkfifo("test", 0666);
        fd = open("test", O_RDWR);
```

```
        read(fd, &c, sizeof(c));
        /* never reached */
        return (0);
}

$ ./a.out
^C
```

- `fcntl()` provádí kontrolu a při výskytu deadlocku vrátí chybu `EDEADLK`.
- je vhodné se deadlocku snažit vyvarovat správným naprogramováním a nespoléhat se na kontrolu systému.

System V IPC

- **IPC** je zkratka pro *Inter-Process Communication*
- komunikace mezi procesy **v rámci jednoho systému**, tj. nezahrnuje síťovou komunikaci
- **semafory** ... použití pro synchronizaci procesů
- **sdílená paměť** ... předávání dat mezi procesy, přináší podobné problémy jako sdílení souborů, k řešení lze použít semafory
- **fronty zpráv** ... spojují komunikaci (zpráva nese data) se synchronizací (čekání procesu na příchod zprávy)
- prostředky IPC mají podobně jako soubory definovaná **přístupová práva** (pro čtení a zápis) pro vlastníka, skupinu a ostatní.

- uvědomte si, že tyto prostředky se vztahují ke konkrétnímu systému, System V, kde se objevily jako první. Další systémy je pak převzaly. **Ze tří zde uvedených synchronizačních prostředků Systemu V se budeme zabývat pouze semafory. Pro sdílenou paměť je možné použít již probrané volání `mmap`, místo zasílání zpráv je možné použít sockety (budou v některé z příštích přednášek).**
- prostředky IPC existují i poté, kdy skončí proces, který je vytvořil. O jejich zrušení je nutno explicitně požádat (ze shellu lze zjistit seznam IPC prostředků příkazem `ipcs` a smazat je příkazem `ipcrm`). Stav a obsah existujících prostředků IPC zůstává v platnosti, i když s nimi právě nepracuje žádný proces (např. data ve sdílené paměti zůstávají, i když ji nemá žádný proces připojenou).

Semafory

- zavedl je E. Dijkstra
- semafor s je datová struktura obsahující
  - celé nezáporné číslo `i` (volná kapacita)
  - frontu procesů `q`, které čekají na uvolnění
- operace nad semaforem:

  init(s, n)
  vyprázdnit `s.q; s.i = n`

  P(s)
  ```
  if(s.i > 0) s.i-- else
  uspat volající proces a zařadit do s.q
  ```

  V(s)
  ```
  if(s.q prázdná) s.i++ else
  odstranit jeden proces z s.q a probudit ho
  ```

- **P** je z holandského ,,proberen te verlagen" – zkus dekrementovat, **V** pak ze slova ,,verhogen" – inkrementovat.
- operace `P(s)` a `V(s)` lze zobecnit: hodnotu semaforu je možné měnit o libovolnou hodnotu `n` ... `P(s, n)`, `V(s, n)`.
- Allen B. Downey: *The Little Book of Semaphores*, Second Edition, on-line na http://greenteapress.com/semaphores/
- *binární semafor* má pouze hodnotu 0 nebo 1

Vzájemné vyloučení pomocí semaforů

- jeden proces inicializuje semafor

```
sem s;
init(s, 1);
```

- kritická sekce se doplní o operace nad semaforem

```
...
P(s);
kritická sekce;
V(s);
...
```

- inicializace semaforu na hodnotu `n` dovolí vstoupit do kritické sekce `n` procesům. Zde semafor funguje jako zámek, vždy ho odemyká (zvyšuje hodnotu) stejný proces, který ho zamknul (snížil hodnotu).
- obecně ale může semafor zvednout jiný proces, než který ho snížil; jinak by to ani nemělo velký smysl. Je zde rozdíl oproti zámkům, viz strana 226.

API pro semafory

`int` **semget**`(key_t` *key*`, int` *nsems*`, int` *semflg*`);`

- vrátí identifikátor pole obsahujícího `nsems` semaforů asociovaný s klíčem `key` (klíč `IPC_PRIVATE` ... privátní semafory, při každém použití vrátí jiný identifikátor). `semflg` je OR-kombinace přístupových práv a konstant `IPC_CREAT` (vytvořit, pokud neexistuje), `IPC_EXCL` (chyba, pokud existuje).

`int` **semctl**`(int` *semid*`, int` *semnum*`, int` *cmd*`, ...);`

- řídicí funkce, volitelný čtvrtý parametr `arg` je typu `union semun`.

`int` **semop**`(int` *semid*`, struct sembuf *`*sops*`, size_t` *nsops*`);`

- zobecněné operace P a V.

- jak získat klíč pro `semget` je vysvětleno na strane 177.
- největší zajímavost na System V implementaci semaforů je skutečnost, že daný syscall neoperuje nad jedním semaforem, ale nad **polem semaforů**, a to atomicky. Většinou však budete potřebovat pouze jeden semafor, tj. pole o jednom prvku. Pro takové použití jsou System V semafory zbytečně složité.
- přístupová práva jsou jen pro čtení a zápis; bit execute zde nemá smysl.
- podobné schéma API funkcí (funkce na vytvoření, řízení a operace) dodržují i ostatní System V IPC mechanismy.
- jakmile je jednou pole semaforů jedním procesem vytvořeno, mohou i ostatní procesy použít `semctl()` a `semop()`, aniž by předtím volaly `semget()`. To platí i pro semafory vytvořené s klíčem `IPC_PRIVATE`, pro které nelze volat `semget()`, protože by se tím vytvořilo nové pole semaforů. Je to tak proto, aby i privátní semafory mohly být děděné v rámci `fork`.

API pro semafory: `semctl()`

- `semnum` ... číslo semaforu v poli
- možné hodnoty `cmd`:
  - `GETVAL` ... vrátí hodnotu semaforu
  - `SETVAL` ... **nastaví semafor na hodnotu** `arg.val`
  - `GETPID` ... PID procesu, který provedl poslední operaci
  - `GETNCNT` ... počet procesů čekajících na větší hodnotu
  - `GETZCNT` ... počet procesů čekajících na nulu
  - `GETALL` ... uloží hodnoty všech semaforů do pole `arg.array`
  - `SETALL` ... nastaví všechny semafory podle `arg.array`
  - `IPC_STAT` ... do `arg.buf` dá počet semaforů, přístupová práva a časy posledních `semctl()` a `semop()`
  - `IPC_SET` ... nastaví přístupová práva
  - `IPC_RMID` ... zruší pole semaforů

- volání `semctl(semid, semnum, SETVAL, arg)` odpovídá obecné semaforové inicializační operaci `init(s, n)`.

API pro semafory: `semop()`

- operace se provádí atomicky (tj. buď se povede pro všechny semafory, nebo pro žádný) na `nsops` semaforech podle pole `sops` struktur `struct sembuf`, která obsahuje:
  - `sem_num` ... číslo semaforu
  - `sem_op` ... operace
    - `P(sem_num, abs(sem_op))` pro `sem_op < 0`
    - `V(sem_num, sem_op)` pro `sem_op > 0`
    - čekání na nulovou hodnotu semaforu pro `sem_op == 0`
  - `sem_flg` ... OR-kombinace
    - `IPC_NOWAIT` ... když nelze operaci hned provést, nečeká a vrátí chybu
    - `SEM_UNDO` ... při ukončení procesu vrátit operace se semaforem

- atomičnost přes množinu semaforů zajistí, že nedojde k deadlocku v následující situaci: dva procesy A a B budou používat dva semafory k řízení přístupu (zamykání) ke dvěma systémovým zdrojům. Proces A je bude zamykat v pořadí $(0, 1)$ a proces B v pořadí $(1, 0)$. Ve chvíli, kdy proces A zamkne semafor 0 a B zamkne 1, dojde k deadlocku, protože ani jeden proces nemůže pokračovat (potřeboval by zamknout druhý semafor). Při použití atomické operace zamčení obou semaforů najednou bude úspěšný vždy právě jeden proces, který získá oba semafory, druhý bude čekat.
- `SEM_UNDO` zajistí, že při ukončení procesu dojde k odemčení semaforů (použitých jako zámky), které tento proces měl zamčené.

Vytváření prostředků IPC

- jeden proces prostředek vytvoří, ostatní se k němu připojí.
- po skončení používání je třeba prostředek IPC zrušit.
- funkce `semget()`, `shmget()` a `msgget()` mají jako první parametr klíč identifikující prostředek IPC. Skupina procesů, která chce komunikovat, se musí domluvit na společném klíči. Různé skupiny komunikujících procesů by měly mít různé klíče.

`key_t` **`ftok`**`(const char *`*`path`*`, int `*`id`*`);`

- vrátí klíč odvozený ze zadaného jména souboru `path` a čísla `id`. Pro stejné `id` a libovolnou cestu odkazující na stejný soubor vrátí stejný klíč. Pro různá `id` nebo různé soubory na stejném svazku vrátí různé klíče.

poznámky k `ftok()`:

- z `id` se použije jen nejnižších 8 bitů.
- není specifikováno, zda bude stejný klíč vrácen i po smazání a znovuvytvoření souboru. Většinou ne, protože v klíči se často odráží číslo indexového uzlu.
- různé klíče pro různé soubory nejsou vždy zaručené. Např. na Linuxu se klíč získá kombinací 16 bitů čísla i-uzlu, 8 bitů `id` a 8 bitů vedlejšího čísla zařízení. Stejný klíč pro různé soubory je vrácen, pokud se čísla i-uzlů shodují na spodních 16 bitech.
- pokud tedy nepříbuzné procesy chtějí používat stejný semafor, **musí být jmého souboru pro klíč domluveno předem.**
- příklad na semafory (je to pomocí semaforů „opravená" verze dřívejšího příkladu `race/race.c` ze strany 161): `race/sem-fixed-race.c`.

Další prostředky IPC

- POSIX a SUSv3 definují ještě další prostředky komunikace mezi procesy:
  - **signály** ... pro uživatelské účely lze využít signály `SIGUSR1` a `SIGUSR2`
  - **POSIXová sdílená paměť** přístupná pomocí `shm_open()` a `mmap()`
  - **POSIXové semafory** ... `sem_open()`, `sem_post()`, `sem_wait()`, ...
  - **POSIXové fronty zpráv** ... `mq_open()`, `mq_send()`, `mq_receive()`, ...
- Z BSD pochází **sokety (sockets)** umožňující komunikaci v doménách `AF_UNIX` (komunikace v rámci jednoho počítače) a `AF_INET` (komunikace na jednom počítači nebo po síti).

- POSIXové IPC používá pro pojmenování jednotlivých IPC objektů řetězce místo numerických identifikátorů, proto do značné míry odpadají problémy s identifikací známé ze System V IPC (kde se řeší např. funkcí `ftok()`).
- POSIXová rozhraní zde uvedená jsou součástí rozšíření 1003.1b (aka POSIX.4), viz strana 6.
- sokety se z BSD rozšířily i do ostatních UNIXových systémů a dostaly se i do normy SUSv2.
- existují další API specifické pro konkrétní systém, např. Solaris doors.

Obsah

- úvod, vývoj UNIXu a C, programátorské nástroje
- základní pojmy a konvence UNIXu a jeho API
- přístupová práva, periferní zařízení, systém souborů
- manipulace s procesy, spouštění programů
- signály
- synchronizace a komunikace procesů
- **síťová komunikace**
- vlákna, synchronizace vláken
- ??? - bude definováno později, podle toho kolik zbyde času

Síťová komunikace

UUCP (UNIX-to-UNIX Copy Program) – první aplikace pro komunikaci UNIXových systémů propojených přímo nebo přes modemy, součást Version 7 UNIX (1978)

sokety (sockets) – zavedeny ve 4.1aBSD (1982); soket je jeden konec obousměrného komunikačního kanálu vytvořeného mezi dvěma procesy buď lokálně na jednom počítači, nebo s využitím síťového spojení

TLI (Transport Layer Interface) – SVR3 (1987); knihovna zajišťující síťovou komunikaci na úrovni 4. vrstvy referenčního modelu ISO OSI

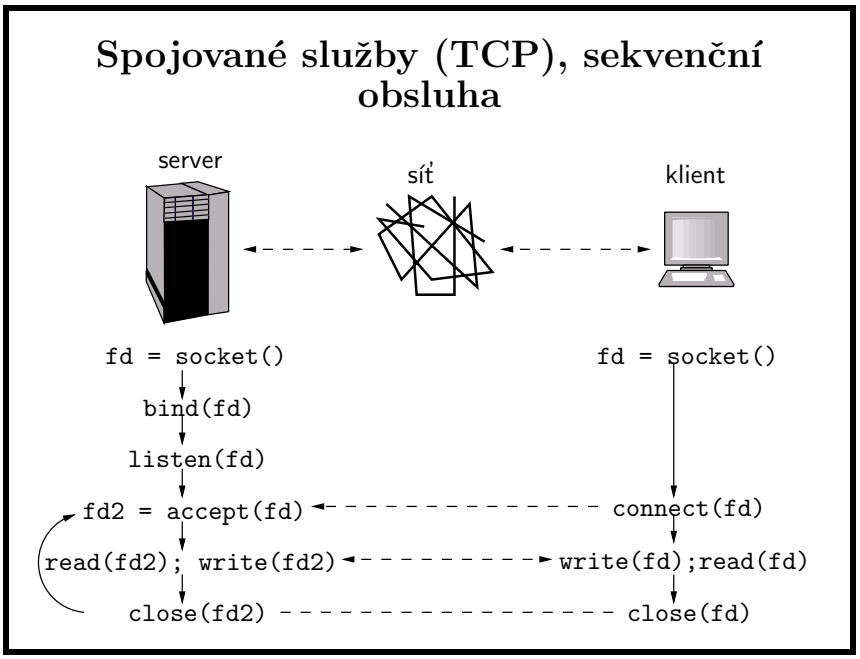
RPC (Remote Procedure Call) – SunOS (1984); protokol pro přístup ke službám na vzdáleném stroji, data přenášena ve tvaru XDR (External Data Representation)

- ISO (International Standards Organization) OSI (Open Systems Interconnect) – vrstvy (layers):
  1. fyzická (physical)
  2. linková (data link)
  3. síťová (network)
  4. transportní (transport)
  5. relační (session)
  6. prezentační (presentation)
  7. aplikační (application)
- UUCP je tvořeno aplikačními programy, nevyžaduje žádnou podporu v jádru. Implementace soketů a TLI jsou součástí jádra. TLI je ve verzi SVR4 implementováno s využitím mechanismu STREAMS. RPC existuje jako knihovna linkovaná k aplikacím, která využívá sokety (funguje nad protokoly TCP a UDP). RPC bylo vyvinuto jako komunikační protokol pro NFS (Networked File System).
- existuje více (vzájemně nekompatibilních) implementací RPC
- komunikační kanál je specifikován adresami dvou soketů.
- sokety pro komunikaci pouze v rámci jednoho počítače jsou v doméně `AF_UNIX` a jejich jména jsou jména speciálních souborů, které reprezentují sokety v systému souborů.
- sokety `AF_UNIX` jsou něco jiného než lokální TCP/IP komunikace přes loopback rozhraní `localhost` (`127.0.0.1`). Více o `AF_UNIX` na straně 186.

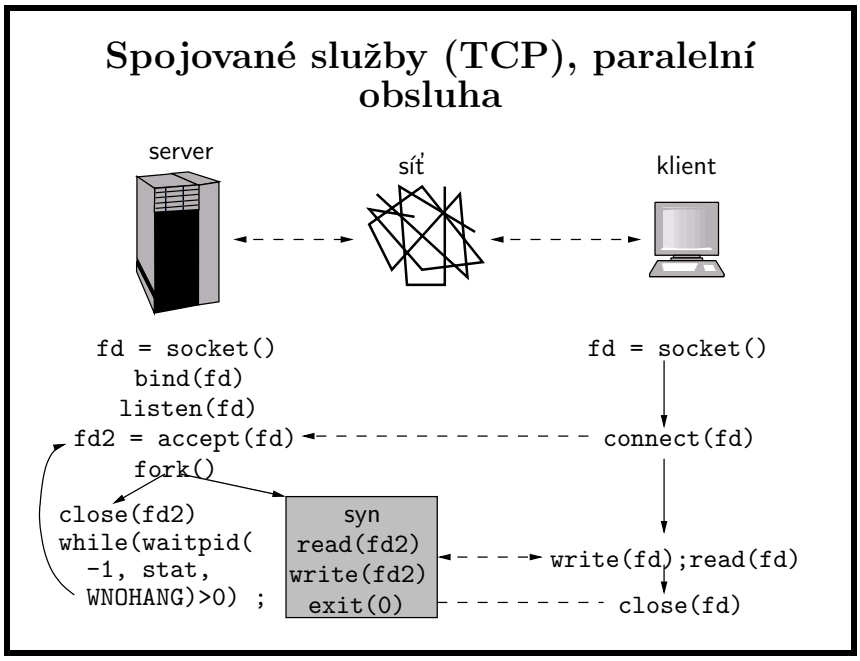
TCP/IP

- protokoly
  - **IP (Internet Protocol)** – přístupný jen pro uživatele `root`
  - **TCP (Transmission Control Protocol)** – streamový, spojovaný, spolehlivý
  - **UDP (User Datagram Protocol)** – datagramový, nespojovaný, nespolehlivý
- **IP adresa** – 4 bajty (IPv4) / 16 bajtů (IPv6), definuje síťové rozhraní, nikoliv počítač
- **port** – 2 bajty, rozlišení v rámci 1 IP adresy, porty s číslem menším než 1024 jsou rezervované (jejich použití vyžaduje práva uživatele `root`)
- **DNS (Domain Name System)** – převod mezi symbolickými jmény a numerickými IP adresami

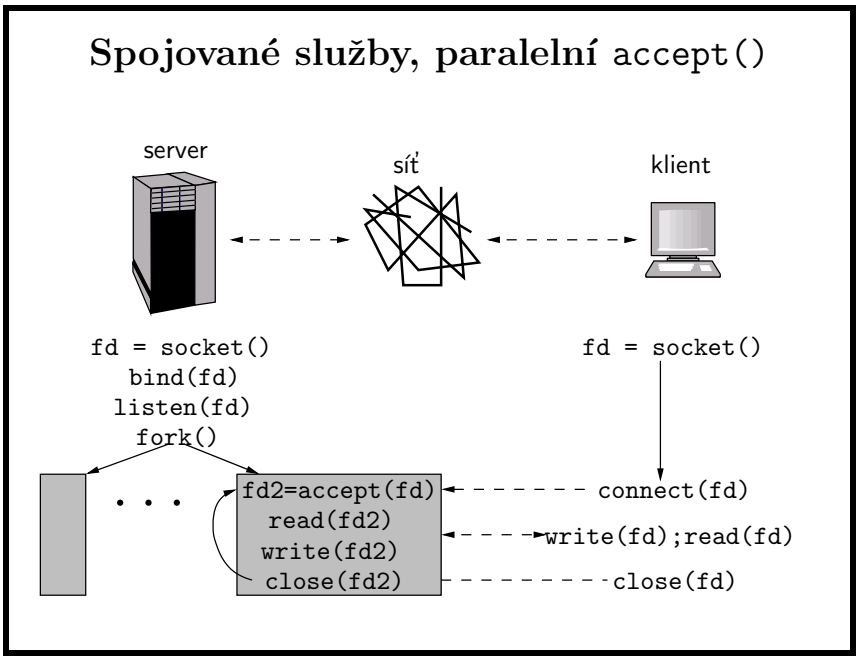
- pokud nevíte, o čem je řeč, doporučuji Peterkovy přednášky; jsou volně ke stažení na webu. Jsou to nejlepší materiály, které jsem k dané problematice viděl.
- UNIX používá pro síťovou komunikaci nejčastěji rodinu protokolů TCP/IP. Pro účely programování aplikací nás budou zajímat především protokoly TCP (spojovaná spolehlivá komunikace) a UDP (nespojovaná nespolehlivá komunikace). V obou protokolech je jeden konec komunikačního kanálu (odpovídá soketu) identifikován IP adresou síťového rozhraní a číslem portu (pomocí portů se rozlišují síťové služby běžící na jednom počítači). TCP spojení je pak jednoznačně definováno jedním párem soketů.
- *IP* – protokol 3. vrstvy, zajišťuje přenos paketů (datagramů) mezi rozhraními identifikovanými IP adresou; je nespolehlivý (nezaručuje doručení dat). Je definován v RFC 791. Nedílnou součástí IP je Internet Control Message Protocol (ICMP), RFC 792.
- *UDP* – jednoduchá nadstavba nad IP, přidává čísla portů, zůstává nespolehlivý a datagramově orientovaný. RFC 768.
- *TCP* – vytváří obousměrné spojení mezi dvěma body (IP+port), poskytuje tok dat (stream) bez rozdělení na zprávy, zajišťuje řízení toku dat a spolehlivé doručování. Pro vytvoření spojení je třeba provést tzv. *handshake*. RFC 793.
- *DNS* – hierarchicky organizovaná databáze, její struktura nemusí mít nic společného se strukturou IP adres



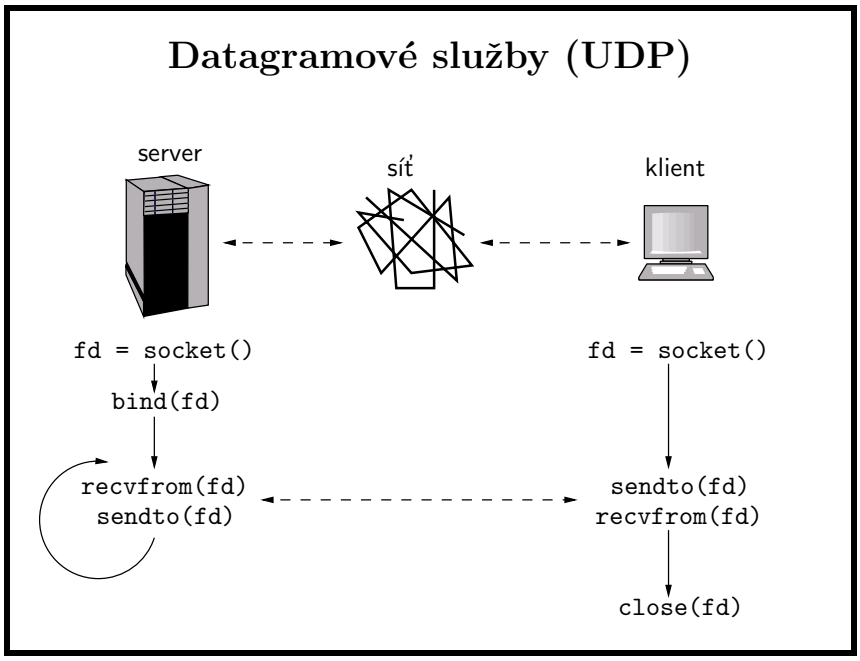
- uvědomte si, že se používají běžná `read` a `write` volání.
- server vytvoří jedno spojení, teprve po jeho ukončení akceptuje dalšího klienta.
- systémová volání:
  - `socket` – vytvoří soket, vrátí jeho deskriptor
  - `bind` – definuje adresu soketu (IP adresu a číslo portu), musí to být buď adresa jednoho ze síťových rozhraní počítače, na kterém je vytvořen soket; pak bude soket přijímat žádosti klientů pouze přes toto rozhraní, nebo je to speciální hodnota „libovolná adresa"; pak soket přijímá požadavky prostřednictvím všech síťových rozhraní (tzv. *wildcard socket*).
  - `listen` – oznámí jádru, že soket bude přijímat požadavky klientů
  - `accept` – uspí proces, dokud nebude k dispozici nějaká žádost klienta o spojení, vytvoří spojení a vrátí nový deskriptor, přes který bude probíhat další komunikace s klientem, původní deskriptor lze použít k novému volání `accept` pro obsloužení dalšího klienta
  - `close` – ukončí komunikaci
  - `connect` – žádost klienta o navázání spojení, IP adresa a číslo portu serveru se zadávají jako parametry, komunikace probíhá přes deskriptor `fd` (na rozdíl od `accept` nevytváří nový deskriptor)
- klient nemusí volat `bind`, v takovém případě mu jádro přidělí některý volný port. Existují služby (např. `rsh`), které vyžadují, aby se klient spojoval z privilegovaného portu (porty 0-1023). Takový klient pak musí provést `bind` (a navíc běžet s dostatečnými právy alespoň do okamžiku provedení `bind`; dostatečná práva mohou znamenat uživatele root nebo speciální privilegium/capability).



- server akceptuje spojení od klienta a na vlastní komunikaci vytvoří nový proces, který po uzavření spojení s klientem skončí. Rodičovský proces může mezitím akceptovat další klienty a spouštět pro ně obslužné procesy. Současně je tedy obsluhováno více klientů.
- po provedení `fork`, ale před zahájením obsluhy spojení, může synovský proces provést `exec` – takto funguje např. `inetd` (strana 208).
- volání `waitpid` v cyklu odstraňuje ze systému zombie. Jinou možností je využití signálu `SIGCHLD`, jehož explicitní ignorování zabrání vzniku živých mrtvých, popř. lze instalovat handler, v němž se volá `wait`.



- po `bind` a `listen` se vytvoří několik synovských procesů a každý v cyklu volá `accept` a obsluhuje klienty. Jádro, pro uživatele **nedeterministicky**, vybere pro každý požadavek jeden proces, v němž `accept` naváže spojení. Hlavní proces žádné požadavky nevyřizuje, ale čeká v některém z volání typu `wait` a vytváří procesy, když je potřeba.
- jednotlivé procesy serveru mezi sebou mohou komunikovat, aby v případě, že současných požadavků je více než serverových procesů, se tato skutečnost zjistila a hlavní server mohl dynamicky vytvořit další serverový proces.
- takto funguje např. webový server Apache. Je možné definovat, kolik požadavků má potomek zpracovat, než sám skončí. Tímto způsobem se výrazně omezí následky případných problémů, například neuvolňování paměti, tzv. *memory leaks*.
- **všechny tři uvedené způsoby činnosti serveru fungují se stejným klientem** – činnost klienta nezávisí na variantě serveru.



- z pohledu volaných síťových funkcí je funkce serveru a klienta shodná. Klient je zde ten, kdo pošle první datagram.
- stejně jako v případě TCP, klient nemusí dělat `bind`, jestliže mu nezáleží na tom, jaké číslo portu bude používat. Server zjistí port klienta z obsahu adresní části datagramu.
- výhodou nespojované služby je menší režie a to, že přes jeden soket lze komunikovat s více procesy (při spojované komunikaci je spojení vždy navázáno s jedním procesem).
- pro UDP je možné volat `connect`. Tím se nenaváže spojení, ale soket se nastaví tak, že může nadále komunikovat pouze s adresou a portem specifikovanými ve volání `connect`. Místo `sendto` a `recvfrom` se pak používají funkce `send` a `recv`.

Vytvoření soketu: `socket()`

`int` **`socket`**`(int` *`domain`*`, int` *`type`*`, int` *`protocol`*`);`

- vytvoří soket a vrátí jeho deskriptor.
- `domain` – „kde se bude komunikovat”:
  - `AF_UNIX` ... lokální komunikace, adresa je jméno souboru
  - `AF_INET`, `AF_INET6` ... síťová komunikace, adresa je dvojice (IP adresa, port)
- `type`:
  - `SOCK_STREAM` ... spojovaná spolehlivá služba, poskytuje obousměrný sekvenční proud dat
  - `SOCK_DGRAM` ... nespojovaná nespolehlivá služba, přenos datagramů
- `protocol`: `0` (default pro daný `type`) nebo platné číslo protokolu (např. `6` = TCP, `17` = UDP)

- funkce je deklarována v `<sys/socket.h>`, stejně jako funkce z dalších slajdů – `bind`, `listen`, a `accept`, a použité konstanty.
- **sokety jsou přístupné přes deskriptory souborů.** Po navázání spojení je (spojovaná) komunikace přes soket podobná komunikaci pomocí roury s tím rozdílem, že sokety jsou vždy **obousměrné**, což nemusí platit u rour, viz strana 137.
- často uvidíte konstanty začínající `PF_` (jako *protocol family*, např. `PF_INET`, `PF_UNIX`, nebo `PF_INET6`) a používané ve volání `socket`. Konstanty `AF_` (*address family*) jsou pak používané pouze při zadávání adres soketů. I když to snad dává větší smysl, norma specifikuje pouze `AF` konstanty pro použití pro volání `socket` i pro zadávání adres. Hodnoty odpovídajících konstant `PF` jsou pak stejné, tj. definované pomocí `AF` konstant. Doporučuji používat jen `AF`.
- existují další typy soketů pro plný přístup k danému protokolu (`SOCK_RAW`), nebo k informacím z routovací tabulky. Pro použití `SOCK_RAW` typicky potřebujete dodatečná privilegia – proto například příkaz `ping`, který vyplňuje ICMP hlavičky odesílaných paketů, bývá nastaven jako SUID:

```
$ ls -l /usr/sbin/ping
-r-sr-xr-x   1 root  bin  55680 Nov 14 19:01 /usr/sbin/ping
```

Pojmenování soketu: `bind()`

```
int bind(int socket, const struct sockaddr *address,
         socklen_t address_len);
```

- přiřadí soketu zadanému deskriptorem `socket` adresu
- obecná `struct sockaddr`, nepoužívá se pro vlastní zadávání adres:
  - `sa_family_t` *`sa_family`* ... doména
  - `char` *`sa_data`*`[]` ... adresa
- pro `AF_INET` se používá `struct sockaddr_in`:
  - `sa_family_t` *`sin_family`* ... doména (`AF_INET`)
  - `in_port_t` *`sin_port`* ... číslo portu (16 bitů)
  - `struct in_addr` *`sin_addr`* ... IP adresa (32 bitů)
  - `unsigned char` *`sin_zero`*`[8]` ... výplň (*padding*)

- volání `bind` přiřazuje soketu *lokální adresu*, tj. zdrojovou adresu odesílaných dat a cílovou adresu přijímaných dat. Vzdálená adresa, tj. adresa druhého konce komunikačního kanálu, se nastavuje pomocí `connect`.
- struktura `sockaddr` je obecný typ, používaný kernelem. Pro konkrétní nastavení adres pak podle zvolené domény a typu lze použít ,,konkrétnější" struktury k tomu definované (viz další slajd), které je pak nutné při použití v `bind` přetypovat na `struct sockaddr`. V naprosté většině případů je ale použití těchto struktur zbytečné a nedoporučené - program pak bude fungovat pouze pro jednu address family. Při použití obecných funkcí pro převod jmen na adresy se lze použití těchto struktur úplně vyhnout - viz funkce `getaddrinfo` na straně 199.
- Budete potřebovat i další hlavičkové soubory. Příklad je na straně 188.
- pro domény `AF_INET` a `AF_INET6` lze zadat číslo portu a speciální hodnotu IP adresy, která znamená libovolnou adresu na daném stroji. Na takový soket (*wildcard soket*) pak bude možné příjímat požadavky na kteroukoli IP adresu nastavenou na kterékoli síťové kartě. Je možné zadat i jednu konkrétní adresu, k tomu se dostaneme.
  - pro `AF_INET` je to hodnota `INADDR_ANY` (4 nulové bajty odpovídající adrese `0.0.0.0`)
  - u `AF_INET6` je situace komplikovanější, buďto lze použít konstantní proměnnou `in6addr_any` nebo konstantu `IN6ADDR_ANY_INIT`, kterou ale lze

použít pouze pro inicializaci proměnné (typu `struct in6_addr`), nikoliv pro přiřazení. Obě tyto hodnoty odpovídají adrese `::` (16 nulových bajtů).

- nelze spojit více soketů s jednou dvojicí (adresa, port).
- volání `bind` lze vynechat, jádro pak soketu (v případě TCP, UDP) přiřadí adresu a některý volný port. Obvykle `bind` volá pouze server, protože klienti očekávají, že bude poslouchat na pevném portu. Naopak klient pevný port nepotřebuje, server se port klienta dozví při navázání spojení nebo z prvního přijatého datagramu.
- **adresa i port musí být do struktury uloženy vždy v síťovém pořadí bajtů.** Pořadí bajtů bylo vysvětleno na straně 18, další informace pak jsou na straně 198.

Struktura pro IPv4 adresy

- každá adresní rodina má svoji strukturu a svůj hlavičkový soubor
- použitou strukturu pak ve volání `socket` přetypujete na `struct sockaddr`

```
#include <netinet/in.h>
struct sockaddr_in in; /* IPv4 */

bzero(&in, sizeof (in));
in.sin_family = AF_INET;
in.sin_port = htons(2222);
in.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(s, (struct sockaddr *)&in, sizeof (in)) == -1) ..
```

- Tento příklad slouží čistě pro demonstraci toho jak fungují struktury uvnitř. Pokud to není nezbytně nutné, neměl by kód který napíšete tyto konstrukce obsahovat - při použití obecných funkcí pro převod jména na adresy vyplňování struktur specifických pro danou address family odpadá. U většiny síťových programů se dnes předpokládá, že budou fungovat jak na IPv4, tak na IPv6.
- u IPv4 je položka `sin_addr` sama strukturou, typu `in_addr`. Tato struktura musí mít alespoň jednu položku, `s_addr`, jejíž typ musí být ekvivaletní 4-bajtovému integeru – pro čtyři bajty IPv4 adresy. To říká přímo UNIX norma pro soubor `netinet/in.h`.

- `sockaddr_in6` je o něco složitější. Je přístupný z `netinet/in.h` (buďto je v něm tato struktura přímo definována nebo soubor inkluduje `netinet6/in6.h` s její definicí).
- u `AF_INET` a `AF_INET6` musí být port i adresa v síťovém pořadí bajtů, viz strana 198.
- `INADDR_ANY` je definovaná jako 0, takže často uvidíte její použití bez `htonl`. Není to dobrý zvyk. Až místo tohoto makra vložíte IP adresu definovanou integerem a zapomenete `htonl` přidat, je hned problém. Když budete od začátku programovat jako slušně vychovaní lidé, toto se vám nemůže stát. A když budete psát síťové programy tak aby už od začátku byly agnostické k address family, tak se tomuto problému vyhnete úplně.
- v doméně `AF_UNIX` se používá adresová struktura `struct sockaddr_un`, definovaná v `<sys/un.h>`:
  - `sa_family_t` *`sun_family`* ... doména
  - `char` *`sun_path`*`[]` ... jméno soketu
  - délka jména není ve standardu definována, závisí na implementaci. Obvyklé hodnoty jsou mezi 92 a 108.

Čekání na spojení: `listen()`

`int` **listen**`(int` *`socket`*`, int` *`backlog`*`);`

- označí soket zadaný desktriptorem `socket` jako akceptující spojení a systém na soketu začne poslouchat.
- maximálně `backlog` žádostí o spojení může najednou čekat ve frontě na obsloužení (implementace může hodnotu `backlog` změnit, pokud není v podporovaném rozsahu). Žádosti, které se nevejdou do fronty, jsou odmítnuty (tj. volání `connect` skončí s chybou).
- soket čeká na spojení na adrese, která mu byla dříve přiřazena voláním `bind`.

- Wildcard sokety se používájí pro server nejčastěji. Konkrétní IP adresa serveru se zadává tehdy, jestliže je potřeba rozlišit, přes které síťové rozhraní přišel požadavek na spojení (pro každé rozhraní máme jeden soket). Tuto možnost využívaly web servery, které podle IP adresy rozlišovaly virtuální

servery. Obvykle se na takovém serveru jednomu fyzickému rozhraní přiřadilo několik IP adres (IP aliasing). To je ale již dávno minulostí – rozlišení virtuálních serverů podle HTTP hlavičky „`Host:`" už nepotřebuje IP aliasy. Podobně TLS protokol obsahuje rozšíření `ServerName`.

- To že systém začne na portu poslouchat znamená, že po připojení proběhne TCP handshake a systém začne přijímat data. Data se ukládají do bufferu s omezenou velikostí, a po dosažení jeho limitu je spojení sice stále aktivní, ale TCP window je nastaveno na 0 – systém další data přestal přijímat. Velikost bufferu bývá v řádu desítek kilobajtů. Příklad: `tcp/up-to-listen-only.c`.
- V příkladu je použito makro `SOMAXCONN`, vyžadované normou v souboru `sys/socket.h`. Specifikuje maximální povolenou délku fronty pro `listen()`. Co se dívám na různé verze systémů co mám k dispozici, tak Linux, FreeBSD, Mac OS X i Solaris používají pro toto makro hodnotu 128.

Akceptování spojení: `accept()`

```
int accept(int socket, struct sockaddr *address,
           socklen_t *address_len);
```

- vytvoří spojení mezi lokálním soketem `socket` (který dříve zavolal `listen`) a vzdáleným soketem, žádajícím o spojení pomocí `connect`. Vrátí deskriptor (nový soket), který lze používat pro komunikaci se vzdáleným procesem. Původní soket může hned přijímat další spojení pomocí `accept`.
- v `address` vrátí adresu vzdáleného soketu.
- `address_len` je velikost struktury pro uložení adresy v bajtech, po návratu obsahuje skutečnou délku adresy.

- Vytvoření druhého deskriptoru pro komunikaci umožňuje na tom původním ihned znovu volat `accept`.
- Nově vytvořený soket má stejné vlastnosti, jako má soket *socket*, tj. pokud je například neblokující, je i nový soket neblokující.
- Jestliže se více klientů ze stejného počítače najednou připojí k jednomu serveru (tj. na jednu serverovou IP adresu a jeden port), jsou jednotlivá spojení rozlišena jen číslem portu na klientské straně. Nezapomeňte, že TCP spojení je jednoznačně definováno dvěma sokety, tj. ((addr1, port1), (addr2, port2)).

- `address` může být zadána jako `NULL`, čímž oznamujeme, že nás konkrétní adresa našeho nového soketu nezajímá. V tomto případě by i `address_len` mělo být `0`.
- Pokud je program napsán správně a je tedy nezávislý na address family, měl by v druhém argumentu předávat adresu struktury `struct sockaddr_storage` (do které se vejde jakákoliv struktura specifická pro danou address family, tedy `struct sockaddr_in` nebo `struct sockaddr_in6`) a v třetím argumentu její délku.
- Příklad: `tcp/tcp-sink-server.c`

Práce s IPv4 a IPv6 adresami

- binární reprezentace adresy se nám špatně čte
- a reprezentaci adresy řetězcem nelze použít při práci se `sockaddr` strukturami

```
int inet_pton(int af, const char *src, void *dst);
```

- převede řetězec na binární adresu, tj. to co je možné použít v `in_addr` nebo `in6_addr` položkách `sockaddr` struktur
- vrací 1 (OK), 0 (chybně zadaná adresa) nebo -1 (a nastaví `errno`)

```
cont char *inet_ntop(int af, const void *src,
                     char *dst, size_t size);
```

- opak k `inet_pton`; vrací *`dst`* nebo `NULL` (a nastaví `errno`)
- pro obě volání platí, že `af` je `AF_INET` nebo `AF_INET6`

- Funkce jsou deklarované v `arpa/inet.h`.
- `inet_pton` vrací 1 pokud konverze proběhla, 0 pokud daný řetězec není adresa dané adresní rodiny, a -1 pokud *af* není podporováno (`EAFNOSUPPORT`). `inet_ntop` vrací `dst` pokud je vše OK a `NULL` s nastaveným `errno` pokud ne.
- Adresy i porty v binární podobě jsou v síťovém pořadí bajtů, jak lze očekávat.
- `dst` musí být dostatečně velké, protože zde není parametr specifikující velikost. To ale není problém, podle nastavení `af` použijete konkrétní adresní strukturu nebo řetězec. Pro maximální postačující velikost řetězců pro adresy norma definuje dvě makra, `INET_ADDRSTRLEN` (16) a `INET6_ADDRSTRLEN` (48). Je to včetně místa pro ukončující `\0`.
- `size` je velikost řetězce `dst`. Pokud není dostatečná, volání selže s chybou `ENOSPC`.

- `n` je `network`, `p` je `presentable`
- Pro IPv4 adresy se používala volání `inet_aton` a `inet_ntoa` (`a` jako `ascii`). Díky novým voláním již tyto funkce není potřeba používat. Všechna tato volání bývají zdokumentována v manuálové stránce `inet`.
- Uvědomte si, že pomocí těchto funkcí můžete převést najednou jen jeden typ, to je buď IPv4, anebo IPv6 adresu. Když nevíte, co čekat, zkusíte jednu a pokud to nevyjde, zkusíte druhou. Příklad: `resolving/addresses.c`.

Navázání spojení: `connect()`

```
int connect(int sock, struct sockaddr *address,
            socklen_t address_len);
```

- naváže spojení lokálního soketu `sock` se vzdáleným procesem, který pomocí `listen` a `accept` čeká na spojení na adrese `address` (o délce `address_len`).
- jestliže pro soket `sock` nebyla definována adresa voláním `bind`, je mu přiřazena nějaká nepoužitá adresa dle zvolené rodiny protokolů.
- pokud se spojení nepovede, je soket v nedefinovaném stavu. Před novým pokusem o spojení by aplikace měla zavřít deskriptor `sock` a vytvořit nový soket.

- Po úspěšném navázání spojení mohou server i klient pro komunikaci používat běžná souborová volání `write` a `read`, nebo funkce `send`, `recv`, `sendmsg`, `recvmsg`. Chování funkcí pro zápis a čtení dat je podobné jako `write` a `read` pro roury.
- Příklad: `tcp/connect.c`
- I pro nespojované služby (UDP) se dá volat `connect`, tím se nenaváže spojení, ale pouze se omezí adresa protistrany, se kterou je možné přes soket komunikovat. Pro posílání datagramů se pak používají funkce `send` a `recv`, které už nemají parametr pro adresu protistrany. Pro nespojované služby můžeme také volat `connect()` vícekrát, každé volání nově nastaví adresu komunikující strany. Pokud použijeme místo adresy `NULL`, nastavení protistrany je zresetováno.
- Pokud je socket nastaven jako neblokující, viz strana 104, `connect` se nezablokuje čekáním na spojení. Místo toho vrátí `-1` s `errno` nastavené na

`EINPROGRESS` (= "nebylo možné vytvořit spojení okamžitě"), a žádost o spojení se uloží do systémové fronty pro následné vyřízení. Do té doby, než je spojení připraveno, volání `connect` vrací `-1`, nyní ale s chybou `EALREADY`. Není ale takto vhodné testovat připravenost spojení, protože pokud `connect` skončí s chybou, další volání `connect` provede nový pokus o spojení a jsme opět tam, kde jsme byli... Je možné ale použít `select` nebo `poll` pro čekání na zápis (ne čtení) do socketu, viz strany 204, 207 které obsahují kompletní příklad na neblokující connect, viz strana 205.

Přijetí zprávy: `recvfrom()`

```
ssize_t recvfrom(int sock, void *buf, size_t len,
                 int flg, struct sockaddr *address,
                 socklen_t *address_len);
```

- přijme zprávu ze soketu `sock`, uloží ji do bufferu `buf` o velikosti `len`, do `address` dá adresu odesílatele zprávy, do `address_len` délku adresy. Vrátí délku zprávy. Když je zpráva delší než `len`, nadbytečná data se zahodí (`SOCK_STREAM` nedělí data na zprávy, data se nezahazují).
- ve `flg` mohou být příznaky:
  - `MSG_PEEK` ... zpráva se bere jako nepřečtená, další `recvfrom` ji vrátí znovu
  - `MSG_OOB` ... přečte urgentní (out-of-band) data
  - `MSG_WAITALL` ... čeká, dokud není načten plný objem dat, tj. `len` bajtů

- používá se hlavně pro sokety typu `SOCK_DGRAM`. V takové situaci opět čeká na celou zprávu, tj. nevrátí vám půlku datagramu. Opět je možné nastavit socket jako neblokující.
- `address_len` **musí** být inicializovaná velikostí bufferu, pokud není adresa `NULL`, čímž říkáte, že vás adresa protistrany nezajímá – tomu tak ale u datagramů vetšinou není.
- místo `recvfrom` se dá použít obecnější funkce `recvmsg`.
- pro sokety, na které bylo voláno `connect`, se místo funkce `recvfrom` volá `recv`.
- po úspešném návratu z `recvfrom` je možné `address` a `address_len` beze změny použít pro následné volání `sendto`.
- stejně jako `sendto`, je i `recvfrom` možné použít pro spojovanou službu. Získat adresu protistrany je ale asi jednodušší přes volání `getpeername`, viz strana 195.

- příklad: `udp/udp-server.c`

Poslání zprávy: `sendto()`

```
ssize_t sendto(int socket, void *msg, size_t len,
               int flags, struct sockaddr *addr,
               socklen_t addr_len);
```

- prostřednictvím soketu `socket` pošle zprávu `msg` o délce `len` na adresu `addr` (o délce `addr_len`).
- parametr `flags` může obsahovat příznaky:
  - `MSG_EOB` ... ukončení záznamu (pokud je podporováno protokolem)
  - `MSG_OOB` ... poslání urgentních (out-of-band) dat, jejichž význam je závislý na protokolu

- Používá se hlavně pro sokety typu `SOCK_DGRAM`, protože v této situaci máme pouze socket reprezentující naši stranu spojení; viz poznámka u slajdu k `accept`. Musíme proto specifikovat adresu protistrany, což voláním `write` nedokážeme. Pro **datagramovou** službu se navíc data berou jako nedělitelná, tj. buď se akceptují celá, nebo se volání zablokuje – neexistuje částečný zápis. Stejně jako je tomu u souborových deskriptorů, je i zde možné socket nastavit jako neblokující, viz strana 104.
- Místo `sendto` se dá použít obecnější funkce `sendmsg`.
- Pro sokety, na které bylo voláno `connect`, se místo `sendto` může použít `send`.
- Úspěšný návrat z libovolné funkce zapisující data do soketu **neznamená úspěšné doručení zprávy protistraně**, ale pouze uložení dat do lokálního bufferu odesílaných dat.
- Pokud použijete `sendto` na streamovanou službu, je to možné, ale adresa se ignoruje. Jediný důvod proč nepoužít přímo `write` by tak byla možnost použít flagy. V tom případě je ale jednodušší použít `send`.
- Příklad: `udp/udp-client.c`.

Další funkce pro sokety

```
int setsockopt(int socket, int level, int opt_name,
               const void *opt_value, socklen_t option_len);
```

- nastavení parametrů soketu

```
int getsockopt(int socket, int level, int opt_name,
               void *opt_value, socklen_t *option_len);
```

- přečtení parametrů soketu

```
int getsockname(int socket, struct sockaddr *address,
                socklen_t *address_len);
```

- zjištění (lokální) adresy soketu

```
int getpeername(int socket, struct sockaddr *address,
                socklen_t *address_len);
```

- zjištění adresy vzdáleného soketu (druhého konce spojení)

- hodnota `level` v `getsockopt` a `setsockopt` je obvykle `SOL_SOCKET`. U `getsockopt`, `option_len` **musí** být inicializována na velikost `opt_value`.
- funkce `getsockname` se používá, když nevoláme `bind` a potřebujeme zjistit, jaká (lokální!) adresa byla jádrem soketu přidělena.
- volání `getsockopt(sock, SOL_SOCKET, SO_ERROR, &val, &len)` vrátí (a vymaže) příznak chyby na soketu. Asi nejužitečnější je při zjišťování, jak dopadl neblokující `connect`, viz strana 192.
- při použití `SO_REUSEADDR` se dá po uzavření poslouchajícího serverového soketu znovu spustit server – volat `socket`, `bind`, `listen` a `accept` na stejné adrese a portu – i když ještě dobíhají spojení vytvořená předchozí instancí serveru:

```
int opt = 1;
setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
```

- Použití `setsockopt` s `SO_REUSEADDR` je vidět např. v příkladu `tcp/reuseaddr.c`. Pozor na to, že musíte udělat alespoň jedno spojení, jinak systém nemá na co čekat a opakovaný `bind` za sebou se podaří i tak.

Uzavření soketu: `close()`

`int` **close**`(int fildes);`

- zruší deskriptor, při zrušení posledního deskriptoru soketu zavře soket.
- pro `SOCK_STREAM` soket záleží na nastavení příznaku `SO_LINGER` (default je `l_onoff == 0`, mění se funkcí `setsockopt`).
  - `l_onoff == 0` ... volání `close` se vrátí, ale jádro se snaží dál přenést zbylá data
  - `l_onoff == 1 && l_linger != 0` ... jádro se snaží přenést zbylá data do vypršení timeoutu `l_linger` v sekundách, když se to nepovede, `close` vrátí chybu, jinak vrátí OK po přenesení dat.
  - `l_onoff == 1 && l_linger == 0` ... provede se reset spojení

- po uzavření může TCP soket zůstat po nějakou dobu v přechodném stavu, který je definován protokolem TCP při ukončování spojení. Než je soket zcela zrušen, nelze použít jiný soket na stejném portu, pokud toto nebylo předtím povoleno nastavením příznaku `SO_REUSEADDR` pomocí funkce `setsockopt`, viz strana 195.
- reset spojení je abnormální ukončení spojení. V TCP se použije paket s nastaveným příznakem `RST`. Na druhé straně se normální ukončení spojení projeví jako konec souboru (při čtení), reset způsobí chybu `ECONNRESET`. Příklad `tcp/linger.c`.

Uzavření soketu: `shutdown()`

`int` **shutdown**`(int` *socket*`, int` *how*`);`

- Uzavře soket (ale neruší deskriptor) podle hodnoty `how`:
  - `SHUT_RD` ... pro čtení
  - `SHUT_WR` ... pro zápis
  - `SHUT_RDWR` ... pro čtení i zápis

- Po uzavření soketu pomocí `shutdown` je nutné zavřít i deskriptor pomocí `close`.
- Při normálním ukončení spojení na úrovni protokolu TCP každá strana signalizuje, že už nebude nic zapisovat. To platí i v případě použití `close` nebo `shutdown(fd, SHUT_RDWR)`. Při použití `shutdown(fd, SHUT_WR)` lze ze soketu dál číst. Druhá strana dostane `EOF` při čtení, ale může dál zapisovat.

Pořadí bajtů

- síťové služby používají pořadí bajtů, které se může lišit od pořadí používaného na lokálním systému. Pro převod lze použít funkce (makra):
  - `uint32_t` **`htonl`**`(uint32_t` *`hostlong`*`);`
    host → síť, 32 bitů
  - `uint16_t` **`htons`**`(uint16_t` *`hostshort`*`);`
    host → síť, 16 bitů
  - `uint32_t` **`ntohl`**`(uint32_t` *`netlong`*`);`
    síť → host, 32 bitů
  - `uint16_t` **`ntohs`**`(uint16_t` *`netshort`*`);`
    síť → host, 16 bitů
- síťové pořadí bajtů je big-endian, tj. nejprve vyšší bajt. Používá se hlavně ve funkcích pracujících s adresami a čísly portů.

- pokud lokální systém používá stejné pořadí bajtů jako síť, nedělají převodní funkce nic.
- jednoduchý a přitom většinou postačující test na to, zda správně převádíte pořadí bajtů, je spustit vaš program proti sobě (pokud to tak lze) na architekturách s rozdílným uspořádáním bajtů.

Čísla protokolů a portů

```
struct protoent *getprotobyname(const char *name);
```

- v položce `p_proto` vrátí číslo protokolu se jménem *name* (např. pro `"tcp"` vrátí 6).
- čísla protokolů jsou uložena v souboru `/etc/protocols`.

```
struct servent *getservbyname(const char *name,
                              const char *proto);
```

- pro zadané jméno služby `name` a jméno protokolu `proto` vrátí v položce `s_port` číslo portu.
- čísla portů jsou uložena v souboru `/etc/services`.

funkce vrací `NULL`, když v databázi není odpovídající položka.

- výsledek **getprotobyname** se hodí pro volání `socket`, výsledek **getservbyname** pro volání `bind`.
- kromě `getservbyname` existuje ještě `getservbyport` umožňující hledat službu pomocí čísla portu (pozor, v network byte order !) a funkce `getservent` a spol. pro ruční procházení záznamů.
- všechny tyto funkce prohledávají pouze "oficiální"seznamy služeb a protokolů, které se nacházejí většinou v souborech zmíněných na slajdu.
- v uvedených souborech je definováno mapování mezi jmény a čísly pro standardní protokoly a služby.
- pozor na to, že protokolem zde nemyslíme HTTP, SSH, telnet nebo FTP – to jsou zde *služby*, reprezentované čísly portů. Protokol je TCP, UDP, OSPF, GRE apod., tedy to, co je přenášeno v IP paketu v položce `Protocol`, viz strany 11 a 14 v RFC 791.
- příklad: `resolving/getbyname.c`

Převod hostname na adresy: `getaddrinfo()`

- ze zadaných parametrů přímo generuje `sockaddr` struktury
- pracuje s adresami i s porty
- její chování je ovlivnitelné pomocí tzv. *hintů*

```
int getaddrinfo(const char *nodename,
                const char *servicename,
                const struct addrinfo *hint,
                struct addrinfo **res);
```

- položky struktury `addrinfo`: `ai_flags` (pro hinty), `ai_family` (address family), `ai_socktype`, `ai_protocol`, `ai_addrlen`, `struct sockaddr *ai_addr` (výsledné adresy), `char *ai_canonname`, `struct addrinfo *ai_next` (další prvek v seznamu)

- Při vyhodnocování dotazů na adresy a jména se používají jmenné služby podle konfigurace v souboru `/etc/nsswitch.conf`.
- struktura `addrinfo` je definována v `netdb.h`
- `getaddrinfo` umí do struktur `sockaddr` převádět jak řetězec který obsahuje hostname, tak řetězec obsahující IP adresu. Stejně je tomu i pro porty.
- funkce `getaddrinfo` vrací parametrem *res* seznam struktur `sockaddr`, ve kterých jsou uloženy adresy korespondující k danému vstupu.
- je rozdíl zda budou adresní struktury dále použity pro server nebo klient; pro server např. stačí jako *nodename* dát `NULL` (wildcard socket). Podobně pro pro položku `ai_flags` struktury `addrinfo` použité pro parametr *hint* a její hodnotu `AI_PASSIVE`.
- Po skončení práce s výsledky nezapomeňte zavolat funkci `freeaddrinfo`, která uvolní naalokovanou pamět.
- příklad: `resolving/getaddrinfo.c`
- Dříve se hodně používaly funkce **gethostbyname** a **gethostbyaddr**. Tyto funkce pracují pouze s IPv4 adresami a jmény, jsou tak **považovány za zastaralé** a jejich použití se nedoporučuje. Místo nich se doporučuje používat obecnější volání `getipnodebyname` a `getipnodebyaddr` (`getipnode*` funkce se nachází v různých systémech, nicméně byly nedávno vyjmuty z GNU libc, takže se na jejich přítomnost nelze univerzálně spolehnout. Navíc nejsou součástí UNIX standardu.) resp. `getaddrinfo`, `getnameinfo`. Z těchto funkcí pouze `getaddrinfo` a `getnameinfo` splňují standard (POSIX.1-2001).

- Pochopitelně, je nutné dobře uvážit, zda existující kód používající legacy/obsolete funkce je potřeba přepisovat. Pokud takovému programu bude IPv4 stačit i nadále, měnit existující a funkční kód nemusí být vždy rozumné. **Nový kód ale vždy pište pomocí nových funkcí**, které podporují IPv4 i IPv6, i pokud byste si mysleli, že váš program nebude IPv6 nikdy používat. Za použití `gethostbyname`, `gethostbyaddr` nebo struktur specifických pro IPv4 resp. IPv6 (bez vážného důvodu) půjde u zkoušky hodnocení dolů.

Převod adresy na hostname: `getnameinfo()`

- protějšek k funkci `getaddrinfo`
- jako vstup pracuje s celými `sockaddr` strukturami, tedy funguje s IPv4 i IPv6 adresami, narozdíl od funkce `gethostbyaddr`.

```
int getnameinfo(const struct sockaddr *sa,
                socklen_t *sa_len,
                char *nodename,
                socketlen_t *nodelen,
                char *servicename,
                socketlen_t *servicelen,
                unsigned flags);
```

- `getnameinfo` provádí v jednom volání funkce konverzi adresy a čísla portu uložených ve struktuře `sockaddr` na stringy v závislosti na namingových službách (tedy standardní cesta přes naming backendy v `/etc/nsswitch.conf`) a hodnotě `flags`. Tedy to, co by bylo postaru nutné provádět pomocí volání `gethostbyaddr` a `getservbyport`.
- Tato funkce je narozdíl od výše zmíněných legacy funkcí reentrantní a hodí se tedy pro použití v prostředí s thready (alespoň na většině systémů).
- manuálová stránka obsahuje seznam použitelných `NI_` flagů pro *`flags`* a jejich význam
- příklad: `resolving/getnameinfo.c`

Příklad: TCP server

```
struct sockaddr_storage ca; int nclients = 10, fd, nfd;
struct addrinfo *r, *rorig, hi;
memset(&hi, 0, sizeof (hi)); hi.ai_family = AF_UNSPEC;
hi.ai_socktype = SOCK_STREAM; hi.ai_flags = AI_PASSIVE;
getaddrinfo(NULL, portstr, &hi, &rorig);
for (r = rorig; r != NULL; r = r->ai_next) {
    fd = socket(r->ai_family, r->ai_socktype,
        r->ai_protocol);
    if (!bind(fd, r->ai_addr, r->ai_addrlen)) break;
}
freeaddrinfo(rorig); listen(fd, nclients);
for (;;) { sz = sizeof(ca);
    nfd = accept(fd, (struct sockaddr *)&ca, &sz);
    /* Komunikace s klientem */
    close(newsock);
}
```

- Toto je obecná kostra serveru. Argument `portstr` je jediný vstupní parametr. První úspěšný bind ukončí procházení seznamu adres.
- Takto napsaný server bude na systému, který podporuje IPv6 i IPv4 přijímat spojení v obou adresních rodinách na tomtéž socketu. Spojení z IPv4 klientů budou mít jako zdrojovou adresu tzv. *IPv4-mapped IPv6 adresu*, která v sobě obsahuje IPv4 adresu (např. `::FFFF:78.128.192.1`).
- Všimněte si, že není nutné téměř nic přetypovávat na pointer na strukturu `sockaddr`. Jedinou výjimkou je volání `accept`. Pro volání `accept` se používá struktura typu `sockaddr_storage` což je obecný "kontejner"schopný pojmout prvky struktury `sockaddr_in` i `sockaddr_in6`.
- `socket` a `bind` se volají pro všechny vrácené `sockaddr` struktury dokud pro jednu z nich `bind` neuspěje nebo se seznam nevyčerpá.
- Problém: nekontrolují se některé návratové hodnoty a neobsahuje volání uvolňující alokované entity, např. uzavření socketu v případě chyby volání `bind`. Stejně tak se neřeší situace kdy se vyčerpá seznam bez toho aby `bind` uspěl (to lze detekovat pomocí kontroly nenulovosti pointeru `r` za koncem prvního cyklu).

Příklad: TCP klient

```
int fd; struct addrinfo *r, *rorig, hi;
memset(&hi, 0, sizeof (hi)); hi.ai_family = AF_UNSPEC;
hi.ai_socktype = SOCK_STREAM;
getaddrinfo(hoststr, portstr, &hi, &r);
for (rorig = r; r != NULL; r = r->ai_next) {
    fd = socket(r->ai_family, r->ai_socktype,
        r->ai_protocol);
    if (connect(fd, (struct sockaddr *)r->ai_addr,
        r->ai_addrlen) == 0)
            break;
}
freeaddrinfo(resorig);
/* Komunikace se serverem */
close(fd);
```

- Toto je obecná kostra TCP klienta. Argumenty `hoststr` a `portstr` jsou jediné vstupní parametry. První úspěšný `connect` ukončí procházení adres pro daný host.
- V příkladu využíváme automatického přidělení volného portu systémem při volání `connect` (nepředcházelo mu volání `bind`).
- postupně se volá `connect` na všechny vrácené `sockaddr` struktury (v `addrinfo` strukturách) pro daný host do té doby, než se podaří spojit.
- Problém: neobsahuje kontrolu návratových hodnot volání. Stejně tak se neřeší situace kdy se vyčerpá seznam bez toho aby `connect` uspěl (to lze detekovat pomocí kontroly nenulovosti pointeru `r` za koncem prvního cyklu). Nebo se neuzavírá socket při neúspěšném volání `connect`, tj. je to file descriptor leak.

Čekání na data: `select()`

```
int select(int nfds, fd_set *readfds,
           fd_set *writefds, fd_set *errorfds,
           struct timeval *timeout);
```

- zjistí, které ze zadaných deskriptorů jsou připraveny pro čtení, zápis, nebo na kterých došlo k výjimečnému stavu. Pokud žádný takový deskriptor není, čeká do vypršení času `timeout` (`NULL` ... čeká se libovolně dlouho). Parametr `nfds` udává rozsah testovaných deskriptorů (`0, ..., nfds-1`).
- pro nastavení a test masek deskriptorů slouží funkce:
  - `void` **FD_ZERO**`(fd_set *fdset)` ... inicializace
  - `void` **FD_SET**`(int fd, fd_set *fdset)` ... nastavení
  - `void` **FD_CLR**`(int fd, fd_set *fdset)` ... zrušení
  - `int` **FD_ISSET**`(int fd, fd_set *fdset)` ... test

- **Motivace:** jestliže chceme číst data z více deskriptorů, je možné, pokud jde o soubor, rovnou jej otevřít s příznakem `O_NONBLOCK`, anebo tento příznak kdykoli nastavit na deskriptoru pomocí volání `fncnl` s druhým parametrem `O_SETFL` (ne `O_SETFD`, viz strana 104). Neblokujícím `read` pak střídavě testujeme jednotlivé deskriptory, a mezi každým kolem testů třeba použít `sleep(1)`. Nevýhody jsou aktivní čekání, režie přepínání mezi módem uživatelským a jádra, možná prodleva (až do délky vašeho čekání mezi jednotlivými koly), a také to, že nesložíte zkoušku (viz strana 167). Správné řešení této situace je použít například `select` a následně zavolat `read` na ty deskriptory, které volání `select` ohlásí jako *připravené*.
- Příklad na busy waiting: `select/busy-wait.c`. Všimněte si, že nově vytvořený soket v příkladu je bez flagu `O_NONBLOCK`, viz strana 190, takže je nutné flag nastavit.
- *Připravený* (*ready*) znamená, že `read` nebo `write` s vynulovaným příznakem `O_NONBLOCK` by se **ne**zablokovalo, tedy ne nutně že nějaká data jsou připravena (`read` např. může vrátit 0 pro end-of-file nebo -1 pro chybu)
- Množina `errorfds` je pro výjimky v závislosti na typu deskriptoru; pro socket to je například příchod urgentních dat (flag `URG` v TCP hlavičce). Neznamená to, že na daném deskriptoru došlo k chybě ! Chyba s nastaveným `errno` se zjistí z ostatních množin po návratovém kódu -1 provedeného volání, tj. například `read`.
- Při volání jsou v množinách deskriptory, které chceme testovat, po návratu zůstanou nastavené jen ty deskriptory, na kterých nastala testovaná událost.

Je nutné je tedy před dalším voláním `select` znovu nastavit. Typicky to jsou bitové masky, ale nemusí tomu být tak; z pozice programátora je to samozřejmě jedno. Procházení přes vrácené množiny je nutné dělat po jednom deskriptoru, přes `FD_ISSET`.

- Funkce `select` je použitelná i pro čekání na možnost zápisu do roury nebo soketu – čeká se, až druhá strana něco přečte a uvolní se místo v bufferu pro další data.
- Místo množiny pro deskriptory je možné uvést NULL, speciální případ při nastavení všech množin na NULL je volání, které se pouze zablokuje do příchodu signálu nebo do vypršení time-outu.
- Po návratu je nutné otestovat každý deskriptor zvlášť, není k dispozici volání, které by vám vytvořilo množinu připravených deskriptorů.
- Pokud obsluhuje síťový server více portů, může volat `select` na příslušné deskriptory soketů a následně `accept` na deskriptory, pro které `select` ohlásil příchod žádosti klienta (připravenost ke čtení).
- Volání `connect` na neblokujícím soketu se hned vrátí, navázání spojení ohlásí následný `select` jako připravenost k zápisu. Více viz strana 192.
- Další možnost použití `select` je síťový server, který v jednom procesu obsluhuje paralelně několik klientů. Pomocí `select` se testuje stav deskriptorů odpovídajících spojení s jednotlivými klienty a přes deskriptory připravené pro čtení/zápis se komunikuje. Aby se mohli připojovat noví klienti, testuje se i deskriptor soketu, který se používá pro `accept`. Využívá se toho, že `select` ohlásí deskriptor s čekající žádostí klienta o spojení jako připravený pro čtení. Na takový deskriptor je možné volat `accept`.
- Pozor na to, že `select` **může** změnit strukturu `timeval`, existuje nové volání `pselect`, které kromě dalších změn strukturu pro timeout nezmění.
- Pro **nfds** je možné použít `FD_SETSIZE`, což je systémová konstanta pro maximální počet deskriptorů. Není to ale nejlepší řešení, protože tato konstanta je sice většinou jen 1024 na 32-bitových systémech, na Solarisu to však pro 64-bitové architektury je už 65536. Předpokládám podobné chování i pro ostatní systémy.
- Pokud se čas nastaví na 0, tedy teď nemluvíme o nastavení ukazatele na `NULL`, `select` se dá použít pro tzv. *polling* – zjistí současný stav a hned se vrátí.
- Příklad: `select/select.c`
- `select` lze použít na zjišťování stavu po volání `connect` na neblokující socket. Zda spojení proběhlo úspěšně se dozvíte z funkce `getsockopt` s `opt_name` nastaveným na `SO_ERROR`, viz strana 195. Příklad: `tcp/non-blocking-connect.c`.

Příklad: použití `select()`

- Deskriptor `fd` odkazuje na soket, přepisuje síťovou komunikaci na terminál a naopak.

```
int sz; fd_set rfdset, efdset; char buf[BUFSZ];
for(;;) {
    FD_ZERO(&rfdset); FD_SET(0, &rfdset);
    FD_SET(fd, &rfdset); efdset = rfdset;
    select(fd+1, &rfdset, NULL, &efdset, NULL);
    if(FD_ISSET(0, &efdset))
        /* Výjimka na stdin */ ;
    if(FD_ISSET(fd, &efdset))
        /* Výjimka na fd */ ;
    if(FD_ISSET(0, &rfdset)) {
        sz = read(0, buf, BUFSZ); write(fd, buf, sz); }
    if(FD_ISSET(fd, &rfdset)) {
        sz = read(fd, buf, BUFSZ); write(1,buf,sz); }
}
```

- Zde je typické použití `select`, kdy je třeba číst data současně ze dvou zdrojů. Tento příklad předpokládá, že deskriptory `0` a `fd` jsou nastaveny jako neblokující.
- Před každým voláním `select` se musí znovu nastavit množiny deskriptorů.
- Lepší řešení je použít `select` i na zápis. Logika řízení je pak taková, že pro každý směr datové komunikace máme samostatný buffer. Příslušný čtecí deskriptor bude v množině pro čtení v `select`, právě když je buffer prázdný. Naopak zápisový deskriptor bude v množině pro zápis, právě když je buffer neprázdný.
- `select` uspí proces i při kontrole připravenosti k zápisu, pokud data nejsou z druhé strany čtena. To jde jednoduše nasimulovat pomocí programu, který se jen připojí (provede TCP handshake), ale nic nečte. Viz příklad a komentář v `select/write-select.c`.

Čekání na data: `poll()`

`int` **`poll`**`(struct pollfd` *`fds`*`[], nfds_t` *`nfds`*`, int` *`timeout`*`);`

- čeká na událost na některém z deskriptorů v poli `fds` o `nfds` prvcích po dobu `timeout` ms (`0` ... vrátí se hned, `-1` ... čeká se libovolně dlouho).
- prvky struktury `pollfd`:
  - `fd` ... číslo deskriptoru
  - `events` ... očekávané události, OR-kombinace `POLLIN` (lze číst), `POLLOUT` (lze psát), atd.
  - `revents` ... události, které nastaly, příznaky jako v `events`, navíc např. `POLLERR` (nastala chyba)

- Tato funkce je obdoba volání `select`.
- argument *timeout* je v milisekundách
- Existujících příznaků pro použití je mnohem více, viz manuálová stránka.
- Na Solarisu je `poll` systémové volání, `select` pak knihovní funkce implementovaná pomocí `poll`, a `poll` je tam preferováno. `poll` je nutné použít v případě, že chcete testovat deskriptor větší nebo rovný `FD_SETSIZE`. **To je hlavní rozdíl mezi voláními `select` a `poll`.** Další rozdíl je ten, že není třeba nastavovat deskriptory po každém volání `poll`, ani nulovat `revents`.
- Čas nastavený na -1 je to samé jako `NULL` u `select`.
- Pokud nastavíte počet deskriptorů na 0 (a měli byste pak pro `fds` použít `NULL`), můžete `poll` jednoduše využít pro čekání s menší granularitou než po sekundách nabízené voláním `sleep`. Příklad: `sleep/poll-sleep.c` Tento "trik"ovšem nefunguje na macOS, takže se na něj nelze spoléhat.
  - Mimochodem, jiný způsob, jak dosáhnout menší granularity než je jedna sekunda je volání `nanosleep`, které je ale definované rozšířením POSIX.4 a tedy nemusí být vždy k dispozici. Příklad: `sleep/nanosleep.c`.

Správa síťových služeb: `inetd`

- servery síťových služeb se spouští buď při startu systému, nebo je startuje démon `inetd` při připojení klienta.
- démon `inetd` čeká na portech definovaných v `/etc/inetd.conf` a když detekuje příchozí spojení/datagram, spustí příslušný server a přesměruje mu deskriptory.
- příklad obsahu `/etc/inetd.conf`:

```
ftp stream tcp nowait root /usr/etc/ftpd ftpd -l
shell stream tcp nowait root /usr/etc/rshd rshd -L
login stream tcp nowait root /usr/etc/rlogind rlogind
exec stream tcp nowait root /usr/etc/rexecd rexecd
finger stream tcp nowait guest /usr/etc/fingerd fingerd
ntalk dgram udp wait root /usr/etc/talkd talkd
tcpmux stream tcp nowait root internal
echo stream tcp nowait root internal
```

- `inetd` je v podstatě velký `poll` cyklus obhospodařující sadu socketů podle konfigurace.
- Start přes `inetd` šetří prostředky, protože příslušný server běží pouze po čas, kdy jsou jeho služby opravdu potřeba. Nehodí se tedy pro pro spoustění vytížených služeb (HTTP) nebo služeb kde může být velký overhead při inicializaci (např. SSH).
- Typicky se pomocí `inetd` spouští servery, které se používají málo nebo jejichž inicializace je relativně nenáročná (`telnetd`, `ftpd`). Silně vytížené a dlouho startující servery (`httpd`) se obvykle startují ze systémových inicializačních skriptů a běží stále.
- Často má cenu mít `inetd` vypnutý úplně. Pokud na vašem stroji běží např. pouze SSH, tak pro to se `inetd` ve většině případů nepoužívá, `inetd` by byl jen dalším serverem běžícím na stroji a zdroj potenciálního nebezpečí, pokud by se v něm nebo v jednom z něj spouštěných programů objevila bezpečnostní chyba. To by ostatně mělo platit pro všechny instalované programy poskytující síťové služby - buďto by měly implicitně poslouchat pouze na localhostu (resp. používat unixové sockety) nebo by implicitně neměly běžet a měly by být spuštěny (ve smyslu permanentně zapnuty) až tehdy když jsou skutečně třeba (tento přístup se označuje jako *secure by default*).

Formát souboru `/etc/inetd.conf`

`služba soket proto čekání uživ server argumenty`

- `služba` ... jméno síťové služby podle `/etc/services`
- `soket` ... `stream` nebo `dgram`
- `proto` ... komunikační protokol (`tcp`, `udp`)
- `čekání` ... `wait` (`inetd` čeká na ukončení serveru před akceptováním dalšího klienta), `nowait` (`inetd` akceptuje dalšího klienta hned)
- `uživatel` ... server poběží s identitou tohoto uživatele
- `server` ... úplná cesta k programu serveru nebo `internal` (službu zajišťuje `inetd`)
- `argumenty` ... příkazový řádek pro server, včetně `argv[0]`

- Soket typu `stream`:
  - `wait` ... server dostane soket, na který musí aspoň jednou zavolat `accept`. Teprve tím získá nový soket, přes který může komunikovat s klientem. Po skončení serveru přebírá řízení soketu zpět `inetd`.
  - `nowait` ... `inetd` zavolá `accept` a získaný soket předá serveru, server tedy může rovnou komunikovat (může používat standardní vstup a výstup) a nemusí vědět, že komunikuje po síti. Mezitím `inetd` čeká na další klienty a podle potřeby spouští další instance serveru.
- Jak pro `wait` tak pro `nowait` provedl `inetd` na soketu volání `bind` aby mu přiřadil číslo portu podle pole `služba`.
- V případě `wait` dostane server soket od `inetd` v podobě deskriptoru 0 (standarní vstup). `inetd` před voláním `fork` provedl `dup2(sock, 0)`, kde `sock` je nabindovaný socket. Viz příklad `inetd/accept-rw.c`.
- V případě `nowait` `inetd` zavolá `accept` (pro TCP spojení), přesměruje deskriptory 0, 1, a 2 do síťového socketu a spustí daný server. Viz příklad: `inetd/echo-server.sh`. Podívejte se do daného skriptu pro podrobné instrukce, jak ho použít.
- Pro soket typu `dgram` má smysl pouze `wait`. Server musí přečíst ze soketu aspoň jeden datagram.
- Jestliže `inetd` restartuje server (kromě `stream nowait`) příliš často (cca jednou za sekundu), usoudí, že nastala chyba a po určitou dobu (asi 10 minut) službu zablokuje (nespouští server a odmítá spojení). Ostatní servery spouští normálně dál.

- Pro zajímavost, ne všechny systémy musí nutně `inetd.conf` používat stejným způsobem. Například od Solarisu 10 se tento soubor použije pouze pro počáteční konverzi do interní databáze pomocí `inetconv(1M)` a pro zapínání a vypínání služeb se pak používá příkaz `inetadm(1M)`.

Obsah

- úvod, vývoj UNIXu a C, programátorské nástroje
- základní pojmy a konvence UNIXu a jeho API
- přístupová práva, periferní zařízení, systém souborů
- manipulace s procesy, spouštění programů
- signály
- synchronizace a komunikace procesů
- síťová komunikace
- **vlákna, synchronizace vláken**
- ??? - bude definováno později, podle toho kolik zbyde času

Vlákna

- vlákno (*thread*) = linie výpočtu (*thread of execution*)
- vlákna umožňují mít více linií výpočtu v rámci jednoho procesu
- klasický unixový model: jednovláknové procesy
- vlákna nejsou vhodná pro všechny aplikace
- výhody vláken:
  - zrychlení aplikace, typicky na víceprocesorech (vlákna jednoho procesu mohou běžet současně na různých procesorech)
  - modulární programování
- nevýhody vláken:
  - není jednoduché korektně napsat složitější kód s vlákny
  - obtížnější debugging

- Pro aplikace, kde každý krok závisí na kroku předcházejícím, nemají vlákna příliš velký smysl.
- Debuggery typicky mají podporu vláken, ale debugging změní timing, takže to co v reálu dělá problém se při debuggingu vůbec nemusí projevit. Toto většinou není problémem u klasických 1-vláknových procesů.
- Jak bylo uvedeno na slajdech s doporučenou literaturou, k vláknům existuje výborná kniha [Butenhof]. On-line pak je třeba dostupná obsáhlá kniha **Multithreaded Programming Guide** na http://docs.oracle.com.
- Další situací, kdy je potřeba zůstat u použití procesů, je pokud je nutné vytvořeným procesům měnit reálné a efektivní UID. To je třeba případ OpenSSH, kde se pro každé spojení vytvoří dva server procesy. Jeden proces běží s maximálními privilegii, typicky jako root, a poskytuje služby neprivilegovanému procesu běžícím pod přihlášeným uživatelem. Takovou službou je třeba alokace presudo terminálu, což pod běžným uživatelem nelze provést. Idea je taková, že většina kódu privilegia roota nepotřebuje, čímž se výrazně zmenší množství kódu, kde by chyba vedla k získání privilegií roota. Tento způsob se nazývá *privilege separation* a není možné ho dosáhnout pomocí vláken tak, že by různá vlákna běžela pod různými uživateli, protože všechna vlákna sdílejí stejný adresový prostor, a každé ho tak může měnit.

Implementace vláken

library-thread model

- vlákna jsou implementována v knihovnách, jádro je nevidí.
- run-time knihovna plánuje vlákna na procesy a jádro plánuje procesy na procesory.
- ⊕ menší režie přepínání kontextu
- ⊖ nemůže běžet více vláken stejného procesu najednou.

kernel-thread model

- vlákna jsou implementována přímo jádrem.
- ⊕ více vláken jednoho procesu může běžet najednou na různých procesorech.
- ⊖ plánování threadů používá systémová volání místo knihovních funkcí, tím více zatěžuje systém.

hybridní modely

- vlákna se multiplexují na několik jádrem plánovaných entit.

- původně UNIX s vlákny nepracoval a první implementace byly čistě knihovní, bez účasti jádra. Dnes se používá spíše implementace vláken v jádru nebo smíšený model.
- **zatímco při práci s více procesy je nutné vyvinout jisté úsilí proto, aby dané procesy mohly data sdílet, u vláken je naopak nutné řešit situaci, jak přirozené sdílení dat uhlídat.**
- vlákna implementována pouze v knihovně mohou být preemptivní i nepreemptivní. Pro preemptivnost je možné použít časovače a signály. Pokud *multithreading* (= použití vláken v aplikaci) není použit pro zvýšení výkonu aplikace, typicky není problém s použitím nepreemptivních vláken. Střídání vláken se automaticky dosáhne používáním blokujících systémových volání.
- pokud se volání v library-thread modelu zablokuje, zablokuje se celý proces, tj. žádné vlákno nemůže běžet. To vyplývá z toho, že jádro v tomto modelu o pojmu vlákno nic neví. Knihovní funkce jsou proto přepsány tak, že místo blokujících volání se použijí neblokující, aktuální kontext se uloží a následně přepne na jiné vlákno pomocí volání `setjmp` a `longjmp`. Příklad: `pthreads/setjmp.c`.

POSIX vlákna (pthreads)

- definované rozšířením POSIX.1c
- volání týkající se vláken začínají prefixem `pthread_`
- tyto funkce vrací 0 (OK) nebo přímo číslo chyby
  - ... a nenastavují `errno`!
  - nelze s nimi proto použít funkce `perror` nebo `err`
- POSIX.1c definuje i další funkce, například nové verze k těm, které nebylo možné upravit pro použití ve vláknech bez změny API, např `readdir_r`, `strtok_r`, atd.
  - `_r` znamená *reentrant*, tedy že funkci může volat více vláken najednou bez vedlejších efektů

- Obecné informace o POSIXu jsou na straně 15.
- Existují i další implementace vláken podporující rozdílná API, např. systémové volání `sproc()` v IRIXu, Cthreads, Solaris threads, ...
- API pro POSIX vlákna jsou na různých systémech k dispozici v různých knihovnách. Např. na Linuxu je nutné programy používající POSIX threads API linkovat s `-lpthread`, na Solarisu jsou funkce součástí `libc`.
- Implementace POSIX threads je většinou postavena nad nativní implementací threadů na daném systému, např. na Solarisu nad funkcemi s prefixem `thr_`.
- O reentrantních funkcích ve spojení s vlákny více na straně 239
- Co se týká ošetření návratových hodnot `pthread_` funkcí, tak vzhledem k tomu, že nenastavují `errno`, není následující kód napsán korektně:

```
if (pthread_create(&thr, NULL, thrfn, NULL) != 0)
        err(1, "pthread_create");
```

protože program vám při chybě vypíše něco jako:

- "`a.out: pthread_create: Error 0`" na Solarisu
- "`a.out: pthread_create: Success`" na Linuxových distribucích
- "`a.out: pthread_create: Unknown error: 0`" na FreeBSD

– nebo něco jiného, podle systému, který zrovna používáte

Trochu matoucí mi přijde přístup Linuxu, protože na první pohled není jasné, co se vlastně stalo a že `errno` bylo nulové. Nicméně, správně napsaný kód je například tento:

```
int e;
if ((e = pthread_create(&thr, NULL, thrfn, NULL)) != 0)
        errx(1, "pthread_create: %s", strerror(e));
```

Všimněte si nutnosti použít `errx` funkci, ne `err`, protože ta interně pracuje s `errno`. Pozor na to, že proměnná `errno` by vůbec nemusela být nulová, pokud by ji nastavila jiná funkce volaná před `pthread_create`, což by mohlo uživatele nebo programátora zmást ještě více.

- Ostatní funkce, které `errno` nastavují, fungují stejně i s vlákny, tj. každé vlákno má svojí `errno`. Pozor je ale třeba dát na to, že různé systémy se chovají různě. Na Linuxu je `errno` automaticky thread-safe protože tam musíte specifikovat vláknovou knihovnu, na Solarisu ale musíte u Sun Studia použít přepínač `-mt` nebo `-D_REENTRANT`, viz manuálová stránka `threads(5)`. Pokud tak neuděláte, nebude `errno` nastaveno korektně. Podívejte se na definici `errno` v `/usr/include/errno.h` a pochopíte, jak to funguje. Další rozdíl je ten, že Solaris má vlákna přímo v `libc`, u ostatních systémů musíte většinout specifikovat vláknovou knihovnu. U `gcc` stačí na jakémkoli systému použít přepínač `-pthreads`, potřebnou knihovnu si překladač už najde sám, a zároveň nastaví `errno` jako thread-safe. Stejně jako u `cc` je ale na Solarisu možné použít jen `-D_REENTRANT`.

Vytvoření vlákna

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_fun)(void*), void *arg);
```

- vytvoří nové vlákno, do `thread` uloží jeho ID.
- podle `attr` nastaví jeho atributy, např. velikost zásobníku či plánovací politiku. `NULL` znamená použít implicitní atributy.
- ve vytvořeném vláknu spustí funkci `start_fun`() s argumentem `arg`. Po návratu z této funkce se zruší vlákno.
- s objekty `pthread_attr_t` lze manipulovat funkcemi `pthread_attr_init()`, `pthread_attr_destroy()`, `pthread_attr_setstackaddr()`, atd...

- Pozor na konstrukce typu:

  ```
  for (i = 0; i < N; i++)
      pthread_create(&tid, attr, start_routine, &i);
  ```

  Na první pohled takto předáme každému vláknu jeho index. Jenže plánovač může způsobit to, že než si nově spuštěné vlákno stačí přečíst hodnotu `i`, příkaz `for` provede další iteraci a hodnota se změní. Obecně vlákno může dostat místo správné hodnoty `i` libovolnou větší.

- Příklady: `pthreads/wrong-use-of-arg.c`, `pthreads/correct-use-of-arg.c`.

- Co je možné použít, pokud potřebujeme předat pouze jednu hodnotu (**podle C standardu je to ale implementačně závislé a tedy nepřenositelné**):

  ```
  assert(sizeof (void *) >= sizeof (int));
  for (i = 0; i < N; i++)
      pthread_create(&tid, attr, start_routine, (void *)(intptr_t)i);
  ```

  ...a ve funkci `void *start_routine(void *arg)` pak přetypovat ukazatel zpátky na integer a máme potřebný identifikátor vlákna:

  ```
  printf("thread %d started\n", (int)arg);
  ```

  Příklad: `pthreads/int-as-arg.c`

- Pokud potřebujeme předat více bajtů než je velikost ukazatele, tak už opravdu musíme předat ukazatel na paměť s příslušnými předávanými daty nebo použít globální proměnné; přístup k nim je pak ale samozřejmě nutné synchronizovat.

- `pthread_t` je "průhledný" typ, do kterého program nevidí a ani by neměl, jeho implementace se může lišit systém od systému; nicméně většinou jde o celočíselný typ, který se používá k mapování na nativní thready na daném systému. Při vytváření několika threadů v cyklu je tedy nutné předat funkci `pthread_create` pokaždé adresu jiné proměnné `pthread_t`, jinak už nepůjde s těmito vlákny nadále manipulovat z hlavního vlákna, např. čekat na jejich dokončení (ačkoliv běžet budou normálně), jejich identifikace se ztratí. Toto je řešitelné např. přes pole `pthread_t` nebo dynamickou alokací.

Soukromé atributy vláken

- čítač instrukcí
- zásobník (automatické proměnné)
- thread ID, dostupné funkcí
  `pthread_t pthread_self(void);`
- plánovací priorita a politika
- hodnota `errno`
- klíčované hodnoty – dvojice `(pthread_key_t key, void *ptr)`
  - klíč vytvořený voláním `pthread_key_create()` je viditelný ve všech vláknech procesu.
  - v každém vláknu může být s klíčem asociována jiná hodnota voláním `pthread_setspecific()`.

- Každé vlákno má zásobník pevné velikosti, **který se automaticky nezvětšuje.** Bývá to kolem 64-512 KB, takže pokud si ve funkci alokujete pole o velikosti 256KB a použijete ho, je dost možné, že skončíte s core dumpem. Pokud chcete zásobník větší než je systémový default, musíte použít argument *attr* při vytvoření vlákna. Příklad: `pthreads/pthread-stack-overflow.c`
- O soukromých klíčovaných atributech vlákna více na straně 221.
- Každé vlákno má ještě vlastní signálovou masku, k tomu se také dostaneme, viz strana 224.

Ukončení vlákna

`void` **`pthread_exit`**`(void *`*`val_ptr`*`);`

- ukončí volající vlákno, je to obdoba `exit` pro proces

`int` **`pthread_join`**`(pthread_t `*`thr`*`, void **`*`val_ptr`*`);`

- počká na ukončení vlákna `thr` a ve `val_ptr` vrátí hodnotu ukazatele z `pthread_exit` nebo návratovou hodnotu vláknové funkce. Pokud vlákno skončilo dříve, funkce hned vrací příslušně nastavené `val_ptr`.
- obdoba čekání na synovský proces pomocí `wait`

`int` **`pthread_detach`**`(pthread_t `*`thr`*`);`

- nastaví okamžité uvolnění paměti po ukončení vlákna, na vlákno nelze použít `pthread_join`.

- Pokud se nepoužije `pthread_exit`, vyvolá se tato funkce při skončení vlákna implicitně, s hodnotou použitou pro `return`
- Norma specifikuje, že stav zásobníku ukončovaného vlákna není definovaný, a proto by se v `pthread_exit` neměly používat odkazy na lokální proměnné ukončovaného vlákna pro parametr *`val_ptr`*
- Pokud nemáme v úmyslu po skončení vlákna volat `pthread_join`, je třeba zavolat `pthread_detach`. Jinak po ukončeném vláknu zůstanou v paměti data nutná pro zjištění jeho výsledku pomocí `pthread_join`. To je podobná situace, jako když rodičovský proces nepoužívá `wait` a v systému se hromadí zombie. Ve funkci pro takové vlákno je možné jednoduše použít toto:

  ```
  pthread_detach(pthread_self());
  ```

- Jiná možnost jak nastavit, že se na vlákna nemusí čekat, je zavolat funkci `pthread_attr_setdetachstate` s hodnotou `PTHREAD_CREATE_DETACHED` nad strukturou atributů a tu pak použít ve voláních `pthread_create`. Příklad: `pthreads/set-detachstate.c`
- Nastavením `NULL` do argumentu *`val_ptr`* systému sdělujeme, že nás návratová hodnota ukončeného vlákna nezajímá.
- Čekat na ukončení vlákna může libovolné jiné vlákno, nejen to, které ho spustilo.

- Doporučuji vždy kontrolovat návratovou hodnotu `pthread_join`, tím si budete jisti, že čekáte správně, při použití špatného ID vlákna funkce hned vrátí, což váš program nutně nemusí negativně ovlivnit co se týká funkčnosti, ale můžete pak narazit na limit počtu vláken nebo vyplýtvat paměť.
- Na rozdíl od procesů **nelze čekat na ukončení kteréhokoli vlákna**. Je to z toho důvodu, že vlákna nemají vztah rodič–potomek, a proto to nebylo považováno za potřebné. Pro zajímavost, Solaris vlákna toto umožní (jako ID vlákna ve funkci `thr_join` se použije 0). Pokud byste tuto funkčnost potřebovali s POSIXovými vlákny, je jednoduché nastavit vlákna jako *detached*, použít podmínkové proměnné, a předávat potřebné informace přes globální proměnnou pod ochranou zámku, který je s podmínkovou proměnnou svázán. Více viz strana 229.
- Příklady: `pthreads/pthread-join.c`, `pthreads/pthread-detach-join.c`

Inicializace

```
int pthread_once(pthread_once_t *once_control,
                 void (*init_routine)(void));
```

- V parametru `once_control` se předává ukazatel na staticky inicializovanou proměnnou
  `pthread_once_t once_control = PTHREAD_ONCE_INIT;`
- První vlákno, které zavolá `pthread_once()`, provede inicializační funkci `init_routine()`. Ostatní vlákna už tuto funkci neprovádějí, navíc, pokud inicializační funkce ještě neskončila, zablokují se a čekají na její dokončení.
- Lze použít např. na dynamickou inicializaci globálních dat v knihovnách, jejichž kód může zavolat více vláken současně, ale je třeba zajistit, že inicializace proběhne jen jednou.

- V programu samotném tuto funkci asi potřebovat nebudete. Místo použití `pthread_once` stačí danou inicializační funkci zavolat před tím, než vytvoříte vlákna...
- Není definováno, co se má stát, pokud je `once_control` automatická proměnná nebo nemá požadovanou hodnotu.

Zrušení vlákna

`int` **pthread_cancel**`(pthread_t` *thread*`);`

- požádá o zrušení vlákna `thread`. Závisí na nastavení

`int` **pthread_setcancelstate**`(int` *state*`, int *`*old*`);`

- nastaví nový stav a vrátí starý:
  - `PTHREAD_CANCEL_ENABLE` ... povoleno zrušit
  - `PTHREAD_CANCEL_DISABLE` ... nelze zrušit, žádost bude čekat na povolení

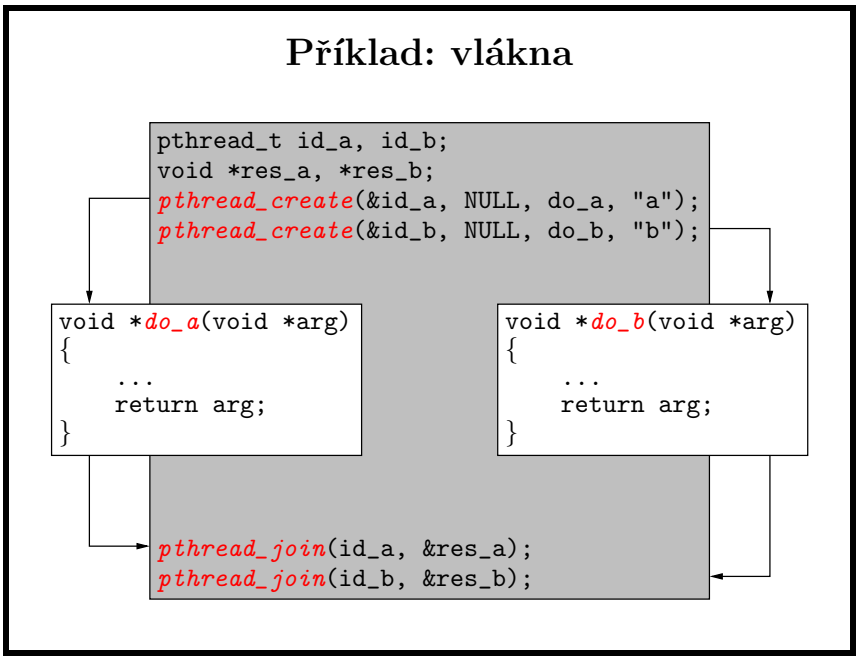
`int` **pthread_setcanceltype**`(int` *type*`, int *`*old*`);`

- `PTHREAD_CANCEL_ASYNCHRONOUS` ... okamžité zrušení
- `PTHREAD_CANCEL_DEFERRED` ... žádost čeká na vstup do určitých funkcí (např. `open()`, `read()`, `wait()`), nebo na volání

`void` **pthread_testcancel**`(void);`

- Vlákna je možné zvenku „násilně" rušit (obdoba ukončení procesu pomocí signálu) buď okamžitě, nebo jen v určitých voláních (tzv. *cancellation points*). To znamená, že v takovém případě je možné vlákno zrušit v čase, kdy je vlákno vykonává danou funkci. Pokud vlákno zrovna takovou funkci nevykonává, informace o zrušení se "doručí" během vykonání první takové funkce od té doby.
- Funkce `pthread_cancel` se podobá zrušení procesu signálem poslaným voláním `kill`.
- Při zrušení vlákna se zavolají úklidové handlery, viz strana 222. Pokud se rozhodnete rušení vláken používat, **buďte velmi opatrní**. Například, pokud budete rušit vlákno, které má zamknutý mutex, musíte mutex odemknout v úklidových handlerech.
- Funkce `pthread_setcancelstate` a `pthread_setcanceltype` jsou obdobou zakázání a povolení zrušení procesu signálem pomocí manipulace s maskou blokovaných signálů (`sigprocmask`).
- Příklad: `pthreads/pthread-cancel.c`
- Při rušení vlákna může nastat mnoho možností. Solaris má třeba samostatnou manuálnovou stránku `cancellation(5)`, která se tomu věnuje. FreeBSD definuje *cancellation points* v manuálové stránce `pthread_setcanceltype(3)`.

Příklad: vlákna



- Toto je triviální příklad, kdy proces (hlavní vlákno) vytvoří dvě další vlákna a počká na jejich ukončení.

Globální proměnné pro vlákno

```
int pthread_key_create(pthread_key_t *key,
                       void (*destructor)(void *));
```

- vytvoří klíč, který lze asociovat s hodnotou typu `(void *)`. Funkce `destructor()` se volají opakovaně pro všechny klíče, jejichž hodnota není `NULL`, při ukončení vlákna.

```
int pthread_key_delete(pthread_key_t key);
```

- zruší klíč, nemění asociovaná data.

```
int pthread_setspecific(pthread_key_t key,
                        const void *value);
```

- přiřadí ukazatel `value` ke klíči `key`.

```
void *pthread_getspecific(pthread_key_t key);
```

- vrátí hodnotu ukazatele příslušného ke klíči `key`.

- Běžné globální proměnné (a také dynamicky alokovaná data) jsou společné pro všechna vlákna. Klíčované hodnoty představují způsob, jak vytvořit globální proměnné v rámci vláken. Uvědomte si rozdíl proti lokální proměnné definované ve vláknové funkci - taková proměnná není vidět v dalších volaných funkcích ve stejném vlákně. Využitelnost soukromých atributů vláken se může zdát malá, ale občas se to velmi dobře hodí. Já to jednou použil v existujícím kódu, kde jsem potřeboval nahradit práci s globálním spojovým seznamem na lokální seznamy specifické pro každé vlákno. Nejjednodušší, tj. nejméně změn v existujícím kódu bylo převést práci s globální proměnnou na práci se soukromým atributem vlákna.
- Při vytvoření klíče je s ním asociována hodnota `NULL`. Ukazatel na destruktor funkci není povinný, pokud není potřeba, použijte `NULL`.
- Při ukončení nebo zrušení vlákna se volají destruktory (v nespecifikovaném pořadí) pro všechny klíče s hodnotou různou od `NULL`. Aktuální hodnota je destruktoru předána v parametru. Jestliže po skončení všech destruktorů zbývají klíče s hodnotou různou od `NULL`, znovu se volají destruktory. Implementace může (ale nemusí) zastavit volání destruktorů po `PTHREAD_DESTRUCTOR_ITERATIONS` iteracích. Destruktor by tedy měl nastavit hodnotu na `NULL`, jinak hrozí nekonečný cyklus.
- Destruktor si musí sám zjistit klíč položky, ke které patří, a zrušit hodnotu voláním `pthread_setspecific(key, NULL)`.
- SUSv3 tento nesmyslný požadavek odstraňuje, protože definuje, že před vstupem do destruktoru je hodnota automaticky nastavená na `NULL`; destruktor se následně vyvolá s předchozí hodnotou klíče.

Úklid při ukončení/zrušení vlákna

- vlákno má zásobník úklidových handlerů, které se volají při ukončení nebo zrušení vlákna funkcemi `pthread_exit` a `pthread_cancel` (ale ne při `return`). Handlery se spouští v opačném pořadí než byly vkládány do zásobníku.
- po provedení handlerů se volají destruktory privátních klíčovaných dat vlákna (pořadí není specifikované)

```
void pthread_cleanup_push(void (*routine)(void *),
                          void *arg);
```

- vloží handler do zásobníku.

```
void pthread_cleanup_pop(int execute);
```

- vyjme naposledy vložený handler ze zásobníku. Provede ho, pokud je `execute` nenulové.

- Handlery se volají jako `routine(arg)`.
- Tyto handlery se dají používat např. na úklid lokálních dat funkcí (obdoba volání destruktorů pro automatické proměnné v C++).

`fork()` a vlákna (POSIX)

- je nutné definovat sémantiku volání `fork` v aplikacích používajících vlákna. Norma definuje, že:
  - nový proces obsahuje přesnou kopii volajícího vlákna, včetně případných stavů mutexů a jiných zdrojů
  - ostatní vlákna v synovském procesu neexistují
  - pokud taková vlákna měla naalokovanou paměť, zůstane tato paměť naalokovaná (= ztracená)
  - obdobně to platí pro zamčený mutex již neexistujícího vlákna
- vytvoření nového procesu z multivláknové aplikace má smysl pro následné volání `exec` (tj. včetně volání `popen`, `system` apod.)

- Pokud mělo již neexistující vlákno zamčený mutex, tak je přístup k příslušným sdíleným datům ztracen, protože zamčený mutex může odemknout pouze to vlákno, které ho zamknulo. Zde ale trochu předbíhám, mutexy jsou představené až na straně 226.
- Ostatní vlákna v novém procesu přestanou existovat, nevolají se žádné rutiny jako při volání `pthread_exit`, `pthread_cancel` nebo destruktory klíčovaných dat.
- Pozor na to, že chování `fork` také závisí na použité knihovně a verzi systému, například na Solarisu před verzí 10 znamenalo `fork` v knihovně `libthread` (jiná knihovna než `libpthread`) to samé co `forkall`.
- Příklady: `pthreads/fork.c`, `pthreads/fork-not-in-main.c`, `pthreads/forkall.c`
- Pomocí funkce `pthread_atfork` je možné nastavit handlery které se automaticky vykonají před použitím `fork` v rodiči a po návratu z `fork` v rodiči a synovském procesu. Toto volání se velmi hodí pro všechny multithreadové programy které volají `fork` a nepoužívají ho pouze jako jednoduchý wrapper pro `exec`. Po `fork` se totiž v synovském procesu nacházejí všechny proměnné ve stejném stavu jako byly v rodiči ve chvíli kdy se zavolal `fork` a tedy pokud mělo nějaké jiné vlákno (než které zavolalo `fork`) např. zamčený mutex (zamykání pomocí mutexů viz strana 226) tak bude tento mutex zamčený i v synovském procesu. Pokud se pak vlákno ze synovského procesu tento mutex pokusí zamknout, dojde k deadlocku. Pokud se v *pre-fork* handleru provede zamčení všech mutexů a v obou (rodič i syn) *post-fork* handlerů odemčení všech mutexů, tato situace nenastane. Tento mechanismus funguje díky tomu,

že *pre-fork* handler se zavolá ve vláknu, které zavolalo fork, ještě předtím než se provede samotný fork syscall; ostatní vlákna mezitím dál běží a mohou tedy uvolnit mutexy (jednoduše tak že každé vlákno v rozumně napsaném programu časem opustí kritickou sekci), na které se čeká v handleru. Toto samozřejmě předpokládá, že zamykání a odemykání v handlerech dodržuje zamykací pravidla ("protokol") stanovený pro celý program a nedojde tedy k deadlocku. `pthreads/atfork.c` Více např. v [Butenhof].

Signály a vlákna

- signály mohou být generovány pro proces (voláním `kill`), nebo pro vlákno (chybové události, volání `pthread_kill`).
- nastavení obsluhy signálů je stejné pro všechna vlákna procesu, ale masku blokovaných signálů má každé vlákno vlastní, nastavuje se funkcí

```
int pthread_sigmask(int how, const sigset_t *set,
                    sigset_t *oset);
```

- signál určený pro proces ošetří vždy právě jedno vlákno, které nemá tento signál zablokovaný.
- lze vyhradit jedno vlákno pro synchronní příjem signálů pomocí volání `sigwait`. Ve všech vláknech se signály zablokují.

- Jestliže je pro signál nastaveno ukončení procesu, skončí celý proces, nejen jedno vlákno.
- Vytvořené vlákno dědí nastavení signálové masky od vlákna, které ho vytvořilo
- Analogicky k použití `sigwait` s procesy (strana 159) – zablokujte příslušné signály ve všech vláknech, včetně vlákna, ve kterém chcete zpracovávat signály pomocí `sigwait`. **Tento způsob zpracování signálů bývá často jediný opravdu doporučovaný pro vlákna**, a navíc je i nejsnáze implementovatelný. Jak vyplývá z předchozí poznámky, stačí zamaskovat signály pouze jednou, a to v hlavním vlákně, protože maska se pak podědí při každém volání `pthread_create`.
- V prostředí vláken nepoužívejte `sigprocmask` (strana 157), protože chování tohoto volání není v takovém prostředí normou specifikováno. Může to fungovat, a také nemusí.
- Příklad: `pthreads/sigwait.c`.

- **Pozor** na to, že byste neměli tento způsob obsluhy signálů používat pro signály synchronní jako jsou `SIGSEGV`, `SIGILL`, apod. Tyto signály jsou generované přímo pro vlákno, takže pokud je zablokujete, vlákno určené pro synchronní příjem signálů je nemusí "vidět", jedině pokud by ten signál způsobilo samo, samozřejmě. Další věc ale je, že specifikací není definováno, zda blokování takových signálů tyto signály skutečně zablokuje, jak již bylo zmíněno na straně 150. Některé systémy tyto signály normálně doručí, čímž proces ukončí, viz přesné znění:

  > *If any of the SIGFPE, SIGILL, SIGSEGV, or SIGBUS signals are generated while they are blocked, the result is undefined, unless the signal was generated by the kill() function, the sigqueue() function, or the raise() function.*

  Příklad: `pthreads/sigwait-with-sync-signals.c`. Tento příklad ukazuje, že na Solarisu 10, Solarisu 11, FreeBSD 7.2 a Linux distribuci, která se hlásí jako "Gentoo Base System release 1.12.13", je signál `SIGSEGV` doručen a proces zabit bez ohledu na to, zda je maskován. Našel jsem ale i systém, který po zamaskování signál nedoručí – FreeBSD 6.0. Synchronní signály by ale mělo být vždy možné zachytit (před zavoláním `exit`), viz strana 150, kde je i příklad.

Synchronizace vláken obecně

- většina programů používajících vlákna mezi nimi sdílí data
- nebo vyžaduje, aby různé akce byly prováděny v jistém pořadí
- ... toto vše potřebuje **synchronizovat** aktivitu běžících vláken
- jak bylo zmíněno u implementace vláken, u procesů je pro sdílení dat nutné vynaložit jisté úsilí, u vláken naopak musíme investovat do toho, abychom přirozené sdílení dat uhlídali
- my se vzhledem k vláknům budeme zabývat:
  - mutexy
  - podmínkovými proměnnými
  - read-write zámky

- pro připomenutí, synchronizací procesů jsme se již zabývali na stranách 161 až 178.
- pomocí mutexů a podmínkových proměnných je možné postavit jakýkoli jiný synchronizační model.

- na tom jakým způsobem budou fungovat synchronizační primitiva se velkou částí podílí scheduler, který např. rozhoduje které z vláken čekajících na odemčení bude po odemčení probuzeno. S tím souvisí klasické problémy, např. *thundering horde* (na odemčení čeká velké množství vláken) nebo *priority inversion* (vlákno, které drží zámek, má menší prioritu než vlákno, které čeká na tento zámek).

Synchronizace vláken: mutexy (1)

- nejjednodušší způsob zajištění synchronizovaného přístupu ke sdíleným datům mezi vlákny je použitím mutexu
- inicializace staticky definovaného mutexu:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- inicializace dynamicky alokovaného mutexu `mx` s atributy `attr` (nastavují se pomocí `pthread_mutexattr_...`; a je-li místo `attr` použit `NULL`, vezmou se defaultní atributy)

```
int pthread_mutex_init(pthread_mutex_t *mx,
                       const pthread_mutexattr_t *attr);
```

- po skončení používání mutexu je možné ho zrušit:

```
int pthread_mutex_destroy(pthread_mutex_t *mx);
```

- Mutex = *mutual exclusion* (vzájemné vyloučení)
- Je to speciální forma Dijkstrových semaforů – rozdíl mezi mutexy a binárními semafory je ten, že **mutex má majitele a zamčený mutex musí odemknout pouze to vlákno, které ho zamknulo.** To u semaforů neplatí. Pozor na to, že abyste zjistili, že například zkoušíte odemknout mutex, který dané vlákno nezamknulo, je nutné jednak testovat návratovou hodnotu volání `pthread_mutex_lock`, a také je nutné ověřit, že máte nastavenou kontrolu zamykání; viz dále.
- Mutexy jsou určené pouze ke krátkodobému držení a měly by fungovat rychle. Slouží pro implementaci kritických sekcí (definice je na straně 163), podobně jako lock-soubory nebo semafory (použité jako zámky).
- Kontrola zamykání u mutexů – defaultně je typ mutexu `PTHREAD_MUTEX_DEFAULT`. V tomto nastavení nejsou normou definovány výsledky zamknutí již zamknutého mutexu, odemknutí mutexu zamknutého jiným vláknem ani odemknutí odemknutého mutexu. Implementace si namapují toto makro na normou definované `PTHREAD_MUTEX_NORMAL` nebo `PTHREAD_MUTEX_ERRORCHECK`. Může se vám proto v závislosti na konkrétní implementaci stát, že pokud

v defaultní konfiguraci zamknete již jednou zamknutý mutex, nastane deadlock (normal), a nebo také ne (errorcheck). V tom druhém případě dostanete návratovou hodnotu o chybě a pokud ji netestujete, budete se mylně domnívat, že jste mutex zamknuli. U normal není výsledek zbylých dvou situací definován, u errorcheck se vám vrátí chyba. "Nedefinováno" u normal znamená, že vlákno odemykající mutex, který nezamknulo, ho klidně může odemknout, nebo také ne – vše závisí na konkrétní implementaci. "Nedefinováno" ale také znamená, že skutečný výsledek takovýchto situací by vás neměl zajímat, protože byste se jim vždy měli vyhnout. Více informací viz norma nebo manuálová stránka k `pthread_mutexattr_settype`. Mně osobně přijde testování mutexových funkcí jako nadbytečné a trochu znepřehledňující kód, ale může být dobrý nápad je používat při vývoji kódu, a před odevzdáním finální verze pak testy odstranit. Solaris a Linux defaultně používají nastavení normal, FreeBSD používá errorcheck. Příklad: `mutexes/not-my-lock.c`.

- další typ je `PTHREAD_MUTEX_RECURSIVE` které drží počet zamčení daným threadem. Ostatní thready dostanou přístup jedině tehdy když počet dosáhne 0. Tento typ mutexu nelze sdílet mezi procesy.

- K čemu jsou dobré rekurzivní mutexy ? Např. máme-li knihovní funkci A:foo(), která získá mutex a zavolá B:bar() která ale může zavolat A:bar(), která se pokusí získat ten samý mutex. Bez rekurzivních zámků by došlo k deadlocku. S rekurzivními mutexy je to ok pokud tyto volání provádí ten samý thread (jiný thread by se zablokoval), pokud si tedy A:foo() a A:bar() jsou vědomy, že ten samý thread už může být v kritické sekci.

- Chování v jednotlivých případech podle typu mutexů shrnuje tabulka:

	`NORMAL`	`ERRORCHECK`	`RECURSIVE`
detekuje deadlock	N	Y	N/A
vícenásobné zamykání	deadlock	error	success
odemčení jiným vláknem	undefined	error	error
odemykání odemčeného	undefined	error	error
lze sdílet mezi procesy	Y	Y	N

- Inicializace statického mutexu pomocí zmíněného makra nastaví pro mutex jeho defaultní atributy. Je možné použít inicializační funkci i pro staticky alokované mutexy. Pokud je mutex alokován dynamicky, je vždy nutné použít `pthread_mutex_init` funkci, ať již budeme používat defaultní atributy nebo ne.

- Dynamické mutexy můžeme potřebovat například v situaci, kdy dynamicky alokujeme datovou strukturu, jejíž součástí je i mutex, který sdílená data struktury chrání. I zde, před zavoláním funkce `free` na datovou strukturu, je potřeba použít volání pro zrušení mutexu, protože mutex sám může mít například alokovanou nějakou paměť. Výsledek zrušení zamknutého mutexu není normou definován.

- Kopírovat mutexy není normou definováno – výsledek takové operace závisí na implementaci. Je možné samozřejmě zkopírovat ukazatel na mutex a s tímto ukazatelem pak dále pracovat.

- Zrušení mutexu znamená jeho deinicializaci.

Mutexy (2)

- pro zamčení a odemčení mutexu použijeme volání:

```
int pthread_mutex_lock(pthread_mutex_t *mx);
```

a

```
int pthread_mutex_unlock(pthread_mutex_t *mx);
```

- pokud je mutex již zamčený, pokus o zamknutí vyústí v zablokování vlákna. Je možné použít i volání:

```
int pthread_mutex_trylock(pthread_mutex_t *mx);
```

... které se pokusí zamknout mutex, a pokud to nelze provést, skončí s chybou

- zamknout mutex, pokud ho dané vlákno již zamčené má, není korektní. Někdy může dojít i k self dead-locku, viz předchozí strana. Pokud potřebujete odemknout mutex, který zamknulo jiné vlákno, použijte místo toho binární semafor; opět viz předchozí strana.
- při vytváření aplikace, kde je efektivita klíčovým faktorem, je nutné rozmyslet, jak, kde a kolik mutexů používat. Z knihovny, která nebyla napsaná pro použití v aplikacích používající vlákna, je možné udělat thread-safe (viz také strana 239) knihovnu tak, že před zavoláním jakékoli funkce knihovny zamknu jeden, tzv. "giant" mutex a po skončení funkce ho odemknu. Mám tedy pouze jeden mutex, ale vlákna používající tuto knihovnu často spí při čekání na přístup do knihovny. Na druhou stranu, pokud zamykám přístup ke konkrétním, malým sekcím, mohu potřebovat mnoho mutexů a značné množství času tak mohu strávit ve funkcích, které mutexy implementují. Je proto vhodné podle situace zvolit rozumný kompromis. Také nezapomeňte, že je možné explicitně nastavit, zda chcete error checking u mutexů nebo ne. Může být dobré řešení kontrolu zamykání/odemykání vypnout tehdy, když už máte vše otestované a věříte, že error checking kód dále nepotřebujete.
- příklady: `mutexes/race.c` a `mutexes/race-fixed.c`
- mutexy je možné nastavit i pro fungovaní mezi vlákny různých procesů. Funguje to tak, že funkce implementující mutexy využijí sdílenou paměť, jejíž odkaz se nastaví jako jeden z atributů mutexu. Více viz manuálová stránka pro `pthread_mutexattr_setpshared`.

Podmínkové proměnné (1)

- mutexy slouží pro synchronizaci přístupu ke sdíleným datům
- podmínkové proměnné pak k předání informací o těchto sdílených datech – například že se hodnota dat změnila
- ... a umožní tak vlákna podle potřeby uspávat a probouzet
- z toho plyne, že **každá podmínková proměnná je vždy asociována s právě jedním mutexem**
- jeden mutex ale může být asociován s více podmínkovými proměnnými
- společně pomocí mutexů a podmínkových proměnných je možné vytvářet další synchronizační primitiva – semafory, bariéry, ...

- jinými slovy – podmínkové proměnné se používají v situaci, kdy vlákno potřebuje otestovat stav **sdílených** dat (např. počet zpráv ve frontě), a dobrovolně se uspat, když hledaného stavu nebylo dosaženo. Spící vlákno je pak probuzeno jiným vláknem, které změnilo stav dat tak, že nastala situace, na kterou první vlákno čeká (třeba vložením prvku do fronty). Druhé vlákno vzbudí první vlákno zavoláním k tomu určené funkce. Pokud žádné vlákno v dané chvíli nespí, probouzecí funkce nemá žádný efekt – nic se nikde neuloží, prostě jako by se to nikdy nestalo.
- není to tak, že při deklaraci podmínkové proměnné, což je pro programátora zcela transparentní typ, s ní asociujete podmínku např. "`n` *je větší než 7*". Podmínkovou proměnnou totiž můžete přirovnat k praporu nějaké barvy, a pokud jej zvednete, znamená to, že ta vlákna, která čekají až touto vlajkou někdo zamává nad hlavou, jsou o této situaci informována (= vzbuzena) a mohou se podle toho zařídit. Některá vlákna tak mohou čekat na to, až `n` bude větší než 7, jiná mohou čekat pouze na to, až se `n` jakkoli změní. Je pouze na programátorovi, zda pro každou konkrétní situaci použije jednu podmínkovou proměnnou, nebo jednu pro všechny situace dohromady. Pro druhou situaci platí, že vlákna čekající na `n == 7` pak musí vždy `n` otestovat, protože ví, že je vzbuzeno při kazdé změně čítače `n`. Pokud není čítač roven sedmi, znovu se dobrovolně uspí. Jak je však uvedeno dále, **test je nutné po probuzení provést vždy**, i když pro něj používáte samostatnou podmínkovou proměnnou – může se stát, že systém z různých implementačních důvodů vzbudí vlákno uspané nad podmínkovou proměnnou, aniž by to způsobilo jiné vlákno a tedy aniž by stav na který vlákno čeká opravdu nastal.

Podmínkové proměnné (2)

```
int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
```

- Inicializuje podmínkovou proměnnou `cond` s atributy `attr` (nastavují je funkce `pthread_condattr_...()`), `NULL` = default.

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

- zruší podmínkovou proměnnou.

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

- čeká na podmínkové proměnné dokud jiné vlákno nezavolá **pthread_cond_signal()** nebo **pthread_cond_broadcast()**.

- je nutné, aby po té, co vlákno zamkne mutex a dříve, než vlákno zavolá `pthread_cond_wait`, otestovat podmínku. Pokud tuhle operaci vlákno neprovede, mohlo by se uspat na neurčitou dobu, protože hláška o splnění podmínky od jiného vlákna by prošla "bez povšimnutí". Jinak řečeno, nesmím se uspat při čekání na situaci, která už mezitím nastala. Nefunguje to jako signály, které pro vás systém drží, pokud je máte například zablokované. Co je důležité je to, že ten test provádíte pod ochranou mutexu, tedy si opravdu můžete být jistí daným stavem věcí při zavolání `pthread_cond_wait`.
- to, že podmínkové proměnné opravdu fungují je způsobeno tím, že při vstupu do kritické sekce vlákno zamkne mutex a **funkce `pthread_cond_wait` před uspáním vlákna mutex odemkne**. Před návratem z této funkce se pak mutex opět zamkne. Může se tedy stát, že vlákno je probuzeno z čekání nad podmínkovou proměnnou, ale nějaký čas se pak ještě uspí při pokusu o zamknutí mutexu. Nehledejte v tom nic složitého, jde pouze o klasické vzájemné vyloučení procesů nad kritickou sekcí.

Podmínkové proměnné (3)

```
int pthread_cond_signal(pthread_cond_t *cond);
```

- probudí jeden proces čekající na podmínkové proměnné `cond`.

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- probudí všechny procesy čekající na podmínkové proměnné `cond`.

```
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           const struct timespec *atime);
```

- čeká na `pthread_cond_signal()` nebo `pthread_cond_broadcast()`, ale maximálně do doby než systémový čas dosáhne hodnoty dané `atime`.

- jak již bylo řečeno, jedna podmínková proměnná může být použita pro hlášení několika rozdílných situací najednou – například při vložení prvku do fronty i při jeho vyjmutí. Z toho důvodu je nutné po probuzení otestovat podmínku, na kterou se čeká. Další věc, která z toho vychází je ta, že v takovém případě musíte použít broadcast. Stačí si představit následující situaci – čekáte na podmínku "změnil se stav fronty", na které čekají čtenáři i zapisovatelé (předpokládejme, že jednu frontu používá více čtenářů i více zapisovatelů). Pokud po vložení zprávy pouze vypustíte jednu signalizaci, tak se může stát, že tato signalizace probudí jiného zapisovatele, který ale samozřejmě čeká na situaci, kdy čtenář z fronty zprávu odebere. Tím se stane, že ve frontě zůstane zpráva, která může být vytlačena až další signalizací.

- vlákno může být probuzeno jiným vláknem i v případě, že je podmínková proměnná svázána pouze s jednou konkrétní událostí, která však po probuzení vlákna již neplatí. Představte si, že těsně po té, kdy jiné vlákno zahlásí splnění podmínky, může další vlákno zamknout mutex a nějakou akcí, např. vyjmutím prvku z fronty, zrušit platnost podmínky "ve frontě je zpráva". To vlákno, které je probuzeno, tak najde frontu prázdnou. Další důvod pro to, že podmínkové proměnné je nutné **vždy** testovat v cyklu.

- v řídkých případech je možné, že se vlákno probudí a podmínka není platná i díky konkrétní implementaci. Z toho opět vyplývá nutnost použití cyklu.

- v parametru `abstime` funkce `pthread_cond_timedwait` se předává absolutní čas, tj. timeout vyprší, když systémový čas dosáhne hodnoty větší nebo rovné `abstime`. Pro absolutní čas bylo rozhodnuto proto, aby programátor nemusel

při případných probuzeních, kdy následně zjistí, že daná podmínka neplatí, přepočítávat časový rozdíl.

Použití podmínkových proměnných

```
pthread_cond_t cond; pthread_mutex_t mutex;
...
pthread_mutex_lock(&mutex);
while (!podminka(data))
    pthread_cond_wait(&cond, &mutex);
process_data(data, ...);
pthread_mutex_unlock(&mutex);
...
pthread_mutex_lock(&mutex);
produce_data(data, ...);
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mutex);
```

- První kus kódu čeká na změnu podmínky. Pokud k ní dojde, data změnila a tedy mohou být zpracována. Druhý kus kódu (spuštěný v jiném vlákně) data připravuje ke zpracování. Jakmile jsou připravena, zasignalizuje konzumentovi.
- pro zopakování – ke každé podmínkové proměnné je potřeba mít ještě mutex.
- funkce `pthread_cond_wait` atomicky odemkne mutex a uspí vlákno. Když je vlákno probuzeno, nejprve se znovu zamkne mutex (tj. toto zamknutí se provede v rámci příslušné implementace podmínkových proměnných!) a teprve pak se volání `pthread_cond_wait` vrátí.
- když signalizujeme, že se něco změnilo, neznamená to ještě, že po změně bude platit podmínka. Navíc, jak bylo několikrát zdůrazněno, `pthread_cond_wait` může vrátit, i když žádné jiné vlákno nezavolalo `pthread_cond_signal` ani `pthread_cond_broadcast`. Další důvod proč je potřeba znovu otestovat podmínku a případně obnovit čekání.
- Odemknutí zámku následuje v příkladu na slajdu až po signalizaci podmínky, ale není to nutné. Můžete signalizovat až po odemknutí a v takovém případě to může být v závislosti na konkrétním systému i efektivnější, protože probuzené vlákno se hned neuspí na zámku, který jinak ještě držíte, pokud signalizujete v rámci kritické sekce.
- příklad: `cond-variables/queue-simulation.c`

Read-write zámky (1)

```
int pthread_rwlock_init(pthread_rwlock_t *l,
                        const pthread_rwlockattr_t *attr);
```

- vytvoří zámek s atributy podle `attr` (nastavují se funkcemi `pthread_rwlockattr_...()`, *NULL* = default)

```
int pthread_rwlock_destroy(pthread_rwlock_t *l);
```

- zruší zámek

```
int pthread_rwlock_rdlock(pthread_rwlock_t *l);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

- zamkne zámek pro čtení (více vláken může držet zámek pro čtení), pokud má někdo zámek pro zápis, uspí volající vlákno (`rdlock()`) resp. vrátí chybu (`tryrdlock()`).

- není součástí POSIXových vláken z POSIX.1c, ale POSIX.1j rozšíření, nazvané "advanced realtime extensions".
- najednou může mít zámek buď několik vláken zamčeno pro čtení nebo maximálně jedno vlákno zamčeno pro zápis (a nikdo pro čtení).
- read-write zámky jsou semanticky podobné zamykání souborů pomocí funkce `fcntl`.
- Je běžné, že implementace preferuje vlákna, která chtějí zapisovat před vlákny, která chtějí číst. Např. pokud je zámek vlastněný zapisovatelem a nějaké další vlákno zavolá **pthread_rwlock_rdlock** a existuje aspoň jedno vlákno čekající v **pthread_rwlock_wrlock**, dá čtenář přednost zapisovateli.
- Existuje maximální počet zamčení pro čtení daný implementací (velikostí typu reprezentujícího počet zamčení), po dosažení maxima vrací **pthread_rwlock_rdlock** hodnotu `EAGAIN`.

Read-write zámky (2)

`int` **pthread_rwlock_wrlock**`(pthread_rwlock_t *`*rwlock*`);`

- zamkne zámek pro zápis; pokud má někdo zámek pro čtení nebo zápis, čeká.

`int` **pthread_rwlock_trywrlock**`(pthread_rwlock_t *`*rwlock*`);`

- jako `pthread_rwlock_wrlock()`, ale když nemůže zamknout, vrátí chybu.

`int` **pthread_rwlock_unlock**`(pthread_rwlock_t *`*rwlock*`);`

- odemkne zámek

- *zvláštnost*: pokud vlákno čekající na zámek dostane signál, po návratu z handleru se vždy pokračuje v čekání, tj. nenastane chyba `EINTR`. Takto se chovají i mutexy a podmínkové proměnné.

Atomické aritmetické operace

- pro architektury, kde operace sčítání není atomická
- výrazně rychlejší než jiné mechanismy pro získání exkluzivního přístupu díky použití instrukcí na dané platformě zajišťujících atomicitu.
- některé systémy dodávají funkce pro atomické operace, (např. `atomic_add(3c)` v Solarisu), obecně je lepší použít podporu v C11 standardu přes *stdatomic.h*.
- sada volání pro různé typy a operace, např. sčítání:

```
#include <stdatomic.h>

atomic_int acnt;
atomic_fetch_add(&acnt, 1);
```

- Příklad `race/atomic-add.c` demonstruje problém se souběhem při sčítání a jeho možná řešení. Program spustí dvě vlákna, každé vlákno pracuje se stejnou globální proměnnou **x**, do které v cyklu postupně přičte čísla od jedné do velikosti parametru **arg**, který program dostane na příkazové řádce. Vlákna běží paralelně, každé z nich provádí toto:

```
for (i = 1; i < arg; ++i)
        x = x + i;
```

  Poté se sčítání pro kontrolu provede v hlavním vláknu, a dvojnásobek (měli jsme dva thready) se porovná s hodnotou globální proměnné **x**. Pokud nejsou výsledky stejné, došlo k souběhu (přetečení proměnné můžeme v této situaci zcela ignorovat).

  Výsledky a časy běhu se markantně liší pro situace, kdy program použil obyčejné sčítání, funkci pro atomickou aritmetiku a zamykání pomocí mutexů. Je vidět několikanásobný rozdíl v době běhu mezi použitím funkce pro atomickou aritmetiku a mutexů, zejména na procesorech hardwarovou podporou paralelismu.

- Podobně existují další atomické rutiny pro odečítání, bitové operace AND a OR, pro přiřazení hodnoty atd.

Bariéra

- bariéra (*barrier*) je způsob, jak udržet členy skupiny pohromadě
- všechna vlákna čekají na bariéře, dokud ji nedosáhne poslední vlákno; pak mohou pokračovat
- typické použití je paralelní zpracování dat

`int` **`pthread_barrier_init`**`(pthread_barrier_t *`*`barrier`*`, `*`attr`*`, unsigned `*`count`*`);`

- inicializuje bariéru pro *count* vstupů do ní

`int` **`pthread_barrier_wait`**`(pthread_barrier_t *`*`barrier`*`);`

- zablokuje se dokud není zavolána *count*-krát

`int` **`pthread_barrier_destroy`**`(pthread_barrier_t *`*`barrier`*`);`

- zruší bariéru

- API pro bariéry je definované od SUSv3, podívejte se například na `pthread_barrier_init`, je možné je ale jednoduše vytvořit pomocí mutexů a podmínkových proměnných.
- Pozor na to, že bariéry jsou nepovinnou částí POSIXu (patří do Advanced realtime threads) a tedy i systém certifikovaný na SUS je nemusí implementovat, což je případ Mac OS X (10.10).
- bariéru můžete využít např. v situaci, kdy mezi jednotlivými fázemi zpracování je potřeba provést jistou inicializaci, vlákna před ní tedy na sebe vždy musí počkat, protože inicializace další fáze může začít až tehdy, kdy skončí fáze předchozí. Příklad `pthreads/pthread-barrier.c` ukazuje použití bariéry pro několik fází zpracování dat.
- podmínka pro bariéru je například hodnota čítače rovnající se nule. Každé vlákno, které dosáhne bariéry, sníží čítač, který je na začátku inicializován na počet vláken. Pokud vlákno po dekrementování čítače zjistí, že ještě není roven nule, uspí se na podmínkové proměnné. Pokud dané vlákno je tím vláknem, které sníží čítač na nulu, místo zavolání `pthread_cond_wait` pošle broadcast, který následně probudí všechna vlákna spící na bariéře (`pthread_cond_signal` zde nestačí, chcete probudit všechna vlákna, ne jen jedno!). Před spuštěním další fáze zpracování se čítač reinicializuje na původní hodnotu. I zde je nutné řešit různé problémy, například není možné jen tak reinicializovat čítač poté, co bariéry dosáhne poslední vlákno, protože jak již víme, vlákna po probuzení z `pthread_cond_wait` musí vždy otestovat, zda čítač je opravdu nulový a pokud není, opět se uspí. Takže by se vám mohlo

stát, že by se probudila jen některá vlákna, nebo taky žádná. Je nutné zresetovat čítač až po probuzení posledního vlákna. Jak byste to řešili?

Semafory

- semafory pochází z POSIX-1003.1b (real-time extensions)
- jména funkcí nezačínají **pthread_**, ale **sem_** (`sem_init`, `sem_post`, `sem_wait`, ...)
- je možné je použít s vlákny

- funkce pro semafory se drží klasické UNIXové sémantiky – při chybě vracejí -1 a nastaví `errno`

Typické použití vláken

- **pipeline**
  - každé z vláken provádí svoji operaci nad daty, která se postupně předávají mezi vlákny
  - každé vlákno typicky provádí jinou operaci

  ... zpracování obrázku, kde každé vlákno provede jiný filtr
- **work crew**
  - vlákna provádějí stejnou operaci, ale nad jinými daty

  ... zpracování obrázku dekompozicí – každé vlákno zpracovává jinou část obrázku, výsledkem je spojení zpracovaných dat ze všech vláken; zde se hodí řešení s bariérou
- **client – server**

- dané rozdělení je jen orientační, použití vláken je samozřejmě neomezené, toto jsou asi ty tři nejčastější použití
- V případě modelu klient – server zpracovává každé vlákno jeden požadavek od jednoho klienta.

Thread-safe versus reentrantní

- *thead-safe* znamená, že kód může být volán z více vláken najednou bez destruktivních následků
  - do funkce, která nebyla navržena jako thread-safe, je možné přidat jeden zámek – na začátku funkce se zamkne, na konci odemkne
  - tento způsob ale samozřejmě není příliš efektivní
- slovem *reentrantní* se typicky myslí, že daná funkce byla navržena s přihlédnutím na existenci vláken
  - ... tedy že funkce pracuje efektivně i ve vícevláknovém prostředí
  - taková funkce by se měla vyvarovat použití statických dat a pokud možno i prostředků pro synchronizaci vláken, protože běh aplikace zpomalují

- z výše uvedeného vyplývá, že thread-safe je slabší vlastnost než reentrantní. Napsat thread-safe funkci lze s použitím synchronizačních primitiv; přepsání existující funkci tak, aby byla reentrantní vyžaduje mnohem více invence.
- reentrantní funkce jsou také jediné funkce bezpečně použitelné v signal handlerech.
- v dnešní době thread-safe většinou znamená reentrantní, tj. funkce jsou přepsány tak, aby pracovaly efektivně i s vlákny, je ale dobré vědět, že někdy to může vyjadřovat rozdíl.
- o zamykání knihoven viz také strana 228.
- existuje množství funkcí, které mohou být thread-safe, ale nikoliv reentrant, např. `gethostbyname`. Běžně tato funkce používá statickou proměnnou, která se znovu použije při každém volání, takže je pro použití v multithreadovém prostředí nevhodné - není thread-safe. Nicméně, na FreeBSD 6.0 je tato funkce implementovaná tak, že používá implicitně thread-local storage pro uložení výstupních dat a tím pádem je thread safe. To ji ještě ale nečiní úplně bezpečnou k použití (nemluvě o tom, že program , který se spoléhá na takové chování není portabilní), viz příklad `reentrant/gethostbyname.c`. O něco lepší je použít reentrantní verzi této funkce `gethostbyname_r` (pokud je na daném systému k dispozici), u které lze specifikovat adresu, kam má ukládat svůj výstup, čímž se stává reentrantní. Daleko nejlepší řešení je použít standardní funkci `getaddrinfo` (viz strana 200), která je sama o sobě reentrantní.

- příklad: `reentrant/inet_ntoa.c` - tady je vidět, že ani takto napsaná funkce vám nepomůže pokud je volaná dvakrát v rámci jednoho volání `printf`. Pokaždé vrací pointer se stejnou adresou (v jednom threadu), kterou si `printf` pouze poznamená a při finálním tisku tedy vypíše reprezentaci poslední adresy, se kterou byla `inet_ntoa` volána. Na Solarisu je to vidět pomocí:

  ```
  truss -t\!all -u libnsl::inet_ntoa ./a.out
  ```

- Na Solarisu obsahují manuálové stránky knihovních funkcí položku `MT-level` v sekci `ATTRIBUTES`, která udává zda je možné funkci použít v multithreadovém prostředí a případně s jakými omezeními. Tyto úrovně jsou popsány v manuálové stránce attributes(5).

Nepřenositelná volání

- nepřenositelná volání končí řetězcem `_np` (*non-portable*) a jednotlivé systémy si takto definují vlastní volání
- FreeBSD
  - **`pthread_set_name_np`**`(pthread_t tid, const char *name)`

  → umožňuje pojmenovat vlákno
- Solaris
  - **`pthread_cond_reltimedwait_np`**`(...)`

  → jako timedwait, ale časový timeout je relativní
- OpenBSD
  - `int` **`pthread_main_np`**`(void)`

  → umožňuje zjistit, zda volající vlákno je hlavní (= `main()`)

- Tyto informace jsou pro zajímavost, abyste věděli, že se s podobnými věcmi můžete setkat. Nepřenositelná volání by se měla používat spíše pro ladící účely. Nikdy nevíte, kdo bude potřebovat váš kód spustit na jiném systému, což se stane typicky a nečekaně po té, co zrovna opustíte vaši společnost a nemáte již čas to opravit.
- Zjistit, jaká nepřenositelná volání váš systém poskytuje je jednoduché, třeba pomocí `apropos _np`, nebo o něco hruběji (aplikujte na svůj systém podle lokace manuálových stránek):

  ```
  $ cd /usr/share/man
  $ find . -name '*_np\.*'
  ./man3c/mq_reltimedreceive_np.3c
  ```

```
./man3c/mq_reltimedsend_np.3c
./man3c/posix_spawnattr_getsigignore_np.3c
./man3c/posix_spawnattr_setsigignore_np.3c
./man3c/pthread_cond_reltimedwait_np.3c
./man3c/pthread_key_create_once_np.3c
./man3c/pthread_mutex_reltimedlock_np.3c
./man3c/pthread_rwlock_reltimedrdlock_np.3c
./man3c/pthread_rwlock_reltimedwrlock_np.3c
./man3c/sem_reltimedwait_np.3c
```

ChangeLog

2017-01-01 oprava textu o tom, že je možné signalizovat podmínkovou proměnnou mimo kritickou sekci, p210. Původní text chybně vysvětloval, že takový kód není správně napsán.

2016-12-12 poznámka o C11 a `stdatomic.h`

2016-11-28 přidán příklad `tcp/linger.c`

2016-11-07 přidán příklad `signals/check-existence.c`

2016-02-25 poznámka k použití `O_EXCL` bez `O_CREAT`, p93

2016-01-13 poznámky k implementací rwlock zámků, p233

2015-12-15 přidán příklad `signals/event-loop.c`

2015-12-12 serie malých oprav

2015-10-22 přidán příklad `main/putenv.c`

2014-12-17 poznámky k rekurzivním mutexům, p226

2014-12-16 přidán příklad na velikost stacku u pthreadů `pthreads/pthread-stack-overflow.c`, p215

2014-12-16 úprava kódu TCP klienta a rozšířeny poznámky o hybách pro TCP server a klient, p201, p202

2014-12-15 zobecněn slajd na straně 235 o atomických aritmetických operacích, příklad `race/atomic-add.c` přepsán aby fungoval jak na Solarisu, tak OS X.

2014-12-15 poznámka k *IPv4-mapped IPv6 adresách* u příkladu TCP serveru, p201.

2014-12-15 vyjmut slajd o legacy/obsolete funkcích **gethostbyname** a **gethostbyaddr**.

2014-12-12 různé poznámky k socket API

2014-02-13 přidán příklad `pthreads/pthread-barrier.c` a vysvětlení API k POSIX bariérám, p236.

2013-11-28 přidán příklad `file-locking/lockf.c` a poznámky k mandatory lockingu, p168.

2013-11-18 více poznámek k OS X, příklad `session/getppid.c`

2013-11-12 více k mmap(), příklad `mmap/aligned.c`

2013-11-11 podrobnější vysvětlení k prioritám procesů

2013-10-21 poznámka k rekurzivnímu mazání adresáře u knihovního volání `rmdir`

2013-05-31 přidán příklad `inetd/accept-rw.c` a více informací k `inetd`, p208.

2013-01-01 více detailů k fungování k pthread_atfork, `pthreads/atfork.c`, p223.

2012-12-18 poznámka k vytváření threadů v cyklu, p215

2012-12-05 přidán příklad `resolving/getbyname.c`, p199 a komentář k funkcím prohledávajícím databáze služeb a protokolů

2012-11-28 přidán příklad `race/fcntl-fixed-race.c`, p169

2012-11-21 přidán příklad `signals/sigqueue.c`

2012-11-14 přidán příklad `mmap/lseek.c` k mmap

2012-01-30 priklady jsou nyni odkazy na web

2012-01-22 přidány poznámky k tread-safe/reentrant funkcím a odkazy na příklady `reentrant/gethostbyname.c`, `reentrant/inet_ntoa.c`, p239.

2012-01-16 slajd o bariéře a semaforech rozdělen na dva, p236

2012-01-16 přidána tabulka shrnující chování jednotlivých typů mutexů, p227.

2012-01-13 poznámka o způsobu implementace POSIX threads a knihovnách, které toto API nabízejí, p213

2012-01-13 poznámka o důležitosti funkce scheduleru při implementaci synchronizačních primitiv

2012-01-11 přidán odkaz na příklad `pthreads/set-detachstate.c`

2012-01-11 rozveden popis použití funkce `pthread_atfork`, p223.

2012-01-10 přeformulován komentář k slajdu s příkladem pro `atomic_add`, p235.

2012-01-10 příklad na neblokující `connect` přesunut za volání `getsockopt` a `select`.

2012-01-05 přepsány slajdy s příklady pro TCP klient a server tak aby nebyly specifické pro žádnou address family

2012-01-04 přidán příklad `resolving/getnameinfo.c`, doplněny slajdy pro funkce `getnameinfo` a `getaddrinfo` na stránkách 201 a 200.

2012-01-03 příklady pro vyšší granularitu sleep byly přesunuty do samostatného adresáře `resolving`

2011-12-22 upřesnění k wildcard soket adresám u volání `bind`

2011-12-21 přidán příklad `tcp/reuseaddr.c` pro demonstraci parametru `SO_REUSEADDR` volání `setsockopt`

2011-11-24 nový slajd o **setpgid** a **setsid**

2011-11-23 poznámky k formátu spustitelných souborů `a.out`

2011-11-09 poznámky k přesměrování a tabulce deskriptorů

2011-11-07 Přidány PDF bookmarks druhého řádu k definicím funkcí a důležitým slajdům.

2011-11-07 Otázka k `fstat`

2011-11-02 Doplněny informace o `/etc/passwd` a `/etc/shadow`. Přidán slajd p 68 s funkcemi pro získávání údajů o uživatelích/skupinách (**getpwent** apod.)

2011-10-26 Zmíněn `chroot`, kosmetické opravy v sekci o API pro soubory.

2011-10-19 Přidán příklad `exit/exit.c` a informace o proměnné prostředí `LD_LIBRARY_PATH`.

2011-10-18 Aktualizace informací o současných unixových systémech, přidáno pár detailů k historii.

2011-10-17 Úpravy stylu (linky, příklady, PDF bookmarks).

2011-01-08 Nový příklad `mutexes/not-my-lock.c`, p227.

2010-10-23 Poznámka o funkci znaku '-' ve jméně shellu v `argv[0]`, p48.

2010-10-21 Nové příklady `read/lseek.c`, p100, `read/big-file.c`, p100, `read/redirect.c`, p103, `stat/filetype.c` a `stat/stat.c`, p106.

2010-10-20 Nový příklad `read/cat.c`, p96.

2010-10-15 Nový příklad `err/err.c`, p63.

2010-03-07 Upřesnění informací ohledně zachytávání synchronních signálů, p150 a p224, a nový příklad `signals/catch-SIGSEGV.c`.

2010-03-06 Přidány informace o `sigwait` volání vzhledem k synchronním signálům, p224. Nový příklad `pthreads/sigwait-with-sync-signals.c`.

2010-02-28 Drobné úpravy.

2010-01-11 Více o `errno` při použití s vlákny, p214.

2009-12-13 Nové příklady `select/busy-waiting.c`, p204, `select/write-select.c`, p206 a `tcp/addresses.c`, p192.

2009-12-06 Nový příklad `signal/signal-vs-sigaction.c`, p155, zmíněno makro `SOMAXCONN`.

2009-10-11 Nové příklady odkazované z části pro první přednášku.

2009-10-06 Drobné fixy, přidání Vládi Kotala jako vyučujícího tohoto předmětu pro letošní rok.

2009-02-13 Zmíněno volání `remove(3)`, p114.

2009-02-06 Nový příklad `tcp/non-blocking-connect.c`, p205.

2009-01-14 Nové příklady z adresáře `main`, p49, `lib-abi/abi-main.c`, p38, `exec/exec-date.c`, p132, `getaddrinfo/gethostbyname.c`, `getopt/getopts.sh` a `getopt/getopt.c`, p54.

2009-01-13 Nové slajdy `getaddrinfo`, p200, `getnameinfo`, p201, nové příklady `cond-variables/queue-simulation.c`, p232, `getaddrinfo/getaddrinfo.c`, p200.

2009-01-10 Nový slajd, který uvádí celou sekci o vláknech, p213, a obsahuje některé obecné informace, které by jinak nebyly zdůrazněny. Nové příklady `mutexes/race.c` a `mutexes/race-fixed.c`, p228.

2009-01-06 Přidána zmínka o *privilege separation*, p211, přidány příklady `pthreads/setjmp.c`, p212, `pthreads/wrong-use-of-arg.c` a `pthreads/correct-use-of-arg.c`, p215, `pthreads/int-as-arg.c`, p215, `pthreads/pthread-join.c` a `pthread-detach-join.c`, p218, `pthreads/pthread-cancel.c`, p219, `pthreads/fork.c`, `pthreads/fork-not-in-main.c`, a `pthreads/forkall.c`, p223, a `pthreads/sigwait.c`, p224.

2008-12-16 Nové slajdy o adresních strukturách pro síťovou komunikaci, p188, a o převodech adres z binární reprezentace na řetězce a naopak, p191. Přidány příklady na TCP, `tcp/up-to-listen-only.c`, p190, `tcp/tcp-sink-server.c`, p191, `tcp/connect.c`, p192, a UDP, `udp/udpclient.c`, p194, `udp/udp-server.c`, p194. Příklad pro práci s více deskriptory, `select/select.c`, p205, a alternativní použití volání `poll` v `select/poll-sleep.c`, p207.

2008-12-09 Příklady `file-locking/lock-unlock.c`, p167, `file-locking/fcntl-locking.c`, p169, `semaphores/sem-fixed-race.c`, p177.

2008-12-01 Příklad `dyn-lib/rtld_next.c`, p145

2008-11-30 Příklady `signals/killme.c`, p148, `signals/catch-SIGKILL.c`, p??, `signals/ctrl-z.c`, p152, `signals/sa_sigaction.c`, p155, `race/race.c`, p161.

2008-11-28 Poznámka k `return (-1)`, p49. Doplnění poznámek k dynamickému linkování. Nové příklady: `dyn-lib/ld-lazy.c` a `dyn-lib/dlopen.c`, p144, `signals/alarm.c`, p156, `signals/sigwait.c`, p159.

2008-11-22 `d_type`, nepřenositelná položka struktury `dirent`, p113. Nový slajd s ukázkou konkrétního adresového prostoru procesu, p118, a k tomu příklad `pmap/proc-addr-space.c`.

2008-11-19 Nový příklad `wait/wait.c`, p135, a doplnění poznámek ke stejnému slajdu. Nové příklady `fork/fork.c` a `fork/vfork.c`, p130.

2008-11-11 Začal jsem postupně přidávat odkazy na příklady k jednotlivým slajdům. Pokud je tedy v poznámkách uvedeno ,,příklad: `readdir/readdir.c`", znamená to, že příklad najdete zde: http://mff.devnull.cz/pvu/src/readdir/readdir.c

2008-11-06 Přidány různé poznámky k API pro práci se soubory. Nový slajd představující funkce `err(3)`, p63.

2008-10-21 Přidán příklad binární nekompatibility OpenSSL knihoven různých verzí v dnešních systémech, p38.

2008-10-04 Přidány různé poznámky a doplnění při kontrole před první přednáškou nového semestru, například ukázka na nesprávné použití `execl` volání, p132, nebo upozornění na to, že jakékoli řešení pro synchronizaci pomocí aktivního čekání není při zkoušce akceptováno, p167, 204.

2008-09-05 Zmíněna volání `setjmp` a `longjmp`, p212

2008-03-30 Přidány informace k módům `RTLD_LAZY` a `RTLD_NOW` pro volání `dlopen`, p144.

2008-03-23 Přidána poznámka k mazání souborů a sticky bitu, p69, více informací k `SIGKILL`, p151, k async-signal-safe funkcím, p154, k aktivnímu čekání při synchronizaci, p165, příklad na deadlock, p170.

2008-03-01 Drobná doplnění k signálům. Přidána poznámka o funkcích `sigwaitinfo(3)` a `sigtimedwait(3)`, p159.

2008-02-29 Přidán samostatný slajd o POSIXu, p15.

2008-02-17 Mnoho oprav různých chyb a jiných nepřesností. Přidána informace o POSIX.4, p155. Děkuju Martinovi Horčičkovi (martin at horcicka dot eu) za feedback.

2008-01-25 Malé opravy v sazbě textu, přidání křížových odkazů, drobná doplnění poznámek na různých místech apod.

2007-12-29 Doplnění definice kritické sekce (p163), přidán úvodní slajd k synchronizaci vláken (p225), doplněny informace o různých typech mutexů (p226), doplněny poznámky k podmínkovým proměnným.

2007-12-12 Přidáno mnoho nových poznámek ke slajdům o síťové komunikaci.

2007-12-09 Přidán slajd pro `atomic_add(3c)` v sekci synchronizace vláken, p235.

2007-11-21 Různá doplnění k signálům na slajdech i v poznámkách.

2007-11-18 Opraveny další výskyty znaku '0' místo řetězce 'len', tentokrát na slajdech k `mmap(2)` volání.

2007-11-07 Přidán konkrétní příklad k poznámkám o soft updates.

2007-10-23 Nový slajd „API vers ABI", p37, a doplnění poznámek k zápisu do pojmenované roury, p98.

2007-10-15 Doplněny poznámky ke slajdům o proměnných prostředí a čtení/zápisu z/do roury, kde byly i faktické chyby.

2007-10-07 Částečně přepsány poznámky ke slajdům o historii unixu, standardech a současných systémech; doplněny poznámky k dynamickému linkeru, debuggerům a C.

2007-09-16 Vyřazena kapitola o administraci, bude nahrazeno něčím jiným, pravděpodobně aplikací `autoconf`.

2007-09-16 Opravy překlepů, gramatických chyb a pod. Díky Honzovi Friedlovi (frido at devnull dot cz)

2007-07-18 Do těchto materiálů byl přidán ChangeLog pro snadnější sledování změn. Generování tohoto textu bylo zároveň přesunuto z FreeBSD na Solaris a v souvislosti s tím jsem přešel z CsLaTeX na Babel. Může to ovlivnit některé fonty a tím i sazbu textu.

The End.