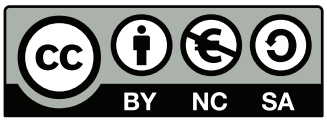
NSWI 0138: Advanced Unix programming

(c) 2011-2016 Vladimír Kotal
(c) 2009-2010 Jan Pechanec, Vladimír Kotal

SISAL MFF UK, Malostranské nám. 25, 118 00 Praha 1
Charles University
Czech Republic

Vladimír Kotal
vlada@devnull.cz

March 10, 2016



Contents

- Officially, this is "Programming in Unix II" (NSWI138) but "Advanced Unix Programming" is a more convenient name. It is supposed to pick up where "Programming in Unix" (NSWI015) left off, and show some areas outside of the scope of the 1st lecture that (not just) a C programmer is usually going to hit sooner or later anyway.
- All information you will need should be on http://mff.devnull.cz/, including the last version of this material.
- Source code examples are on http://mff.devnull.cz/pvu2/src/
- Assumptions:
  - NSWI015 passed ("Programming in Unix"). Materials for that are on-line on http://mff.devnull.cz/pvu/slides/ but they are only in Czech.
  - Good knowledge of the C language.
  - Operating Systems theory basics.
- This text and source code examples are under construction, see ChangeLog on page 87 for more information.

Terms of Use

- Source code: standard 3-clause BSD license (see http://blogs.oracle.com/chandan/entry/copyrights_licenses_and_cddl_illustrated for illustration of what it allows)
- This material: Creative Commons: Attribution + Non-commercial + Share Alike

1 Overview

1.1 What is this lecture about?

- this lecture should extend the knowledge gained in Programming in UNIX (SWI015)
- covers more advanced areas a common UNIX C programmer usually hits sooner or later

Obviously, it cannot cover everything, however what is discussed is tightly connected with source code examples and hands-on experience.

1.2 The lecture will cover...

not neccessarily in the following order...

- a few notes on testing your code
- how to efficiently debug user level programs
- working with terminals and pseudo terminals, writing terminal applications
- advanced network programming
- advanced thread programming, using a non-POSIX thread API
- advanced IPC and I/O
- secure programming or how to write your programs in a more secure way and how to avoid common pitfalls

1.3 A few notes on source code files

- most of them should work on Solaris, Linux, BSD/OS X
- with POSIX standard in mind but sometimes we show system specific features, e.g. Solaris Threads API

2 Testing

2.1 Why ?

- To deliver good quality product (otherwise you will lose customers, money or lifes). Read the article "History's Worst Software Bugs" on http://www.wired.com/software/coolapps/news/2005/11/69355?currentPage=all in Wired (by Simson Garfinkel) about the nature and dreadful consequences of some of the bugs in software.
- To gain good level of confidence that the code works as expected (so you can sleep without unrest)
- Not to harm your reputation (by delivering crap product)
- Not to break the existing functionality when adding new feature or substantially changing current implementation: http://en.wikipedia.org/wiki/Regression_testing
- To test stuff others have produced, e.g. when doing assesment of the product, when taking over the code someone else has written, etc.
- etc.

2.2 When ?

- During development (to make sure you're actually progressing, not going sideways or regressing) (1)
- During code review (to be sure your changes done to satisfy code reviewer's comments are sound)
- Before integration (after merging with mainstream gate)
- After integration (to be sure someone else's change will not break your integration)

(1) To write pscp (parallel scp(1) script) pscp-test was used: http://blogs.oracle.com/janp/entry/speeding_up_ssh_data_transfer

  – otherwise it would not be possible to complete the script in such a short time and catch most of the corner cases.

Task: *Structure shape*

Source code: `testing/structure-shape/`
Comment: This source code is based on a real world code from a daemon implementing IKE protocol (see RFC 2409). During protocol exchange, the 2 sides need to agree on common set of properties. The key established during the exchange is then used to protect internet traffic. Given the properties of symmetric ciphers (see birthday attack) and limitation of exposure in case of key compromise, the rekeying is performed every once in a while. This rekeying should not disrupt the traffic flow, so it happens in

steps. There are two rekeying times - *soft* and *hard*. When the soft timer expires, new key negotiation is kicked off. When the hard timer expires, the old key is deleted. Thus, there should be ample time between soft and hard timer values for the rekeying to succeed. There are two sets of timers - one for time and one for the amount of transferred data. Each timer has its default value. The program tries to produce a set of 4 values which comply to the set of rules.

Steps:

1. Compile: `make clean && make`
2. See the code and try to run with different arguments:
   `./shapeup 50 60 0 0`
   `./shapeup 0 0 0 0` , etc.
3. Find how many bugs are there in the code (there are at least 5 however I am pretty sure you will find even more).
4. Modify the program slightly to check for the 4 conditions using `assert()` and write a shell script which will perform fuzz testing, i.e. run the program with random (integer) values. The crashes induced by the `assert()`s will make the problems apparent.
5. Now you know what types of errors are there in the program so you can design test cases to prevent regressions.
6. Create public project on http://github.com and populate it with Makefile, `struct.c` files.
7. Modify `struct.c` code so that when specifying `-t` option on the command line, it will check whether the 4 conditions are met. If not, the program will return 1, otherwise it will return 0. The modification of the program should be done using refactoring so that the previous functionality is preserved, i.e. running `./shapeup <num> <num> <num> <num>` will still work and will always return 0.
8. Create couple of unit tests, each unit test will run the program with the `-t` option with multiple different sets of values. Add `test` target to the Makefile which will run the tests. Even if one of the tests fails, the rest of tests should be still executed with overall result as fail.
9. Integrate Travis CI with your GitHub repo. With each push to the repository the `test` target will be invoked. Carefully note the limitations of building in the Travis CI environment (operating system choices, compiler modes and versions, etc.)
10. Start fixing the program so that it works. Proceed with small steps, commit each change and observe Travis CI builds.
11. Once you are confident that all of the bugs are fixed, ask one of your fellow students to review your code. Fix any bugs he may find.

2.3 Types of testing

- How ?

  - unit test (for given problem)
    * test just the specific change, e.g. to make sure the fix really addresses the problem.
  - stress test (3)
  - automated test suite (2)
  - Test bed - deploy the product in near-real-life scenario (1)

(1) The "eat our own food/fly our own planes" spirit - install the product and use it internally/by yourself/by your friends (better motivation for good quality). Gives additional code coverage (http://en.wikipedia.org/wiki/Code_coverage).

(2) Automated test suite should be written during development and after that used on a regular basis - (at least before integration of a change and before release is shipped), can be run on multitude of hardware platforms, with different settings - this will give a matrix of what should/could be tested. Test suite can contain both functional and stress tests.

(3) Example: when writing multithreaded application/library it's useful to write a stress test which would try to break it by issuing many valid requests/calls in unpredictable fashion. Useful for detecting boundary conditions, race conditions etc. This also gives more testing coverage.

3 Debugging

3.1 Debuging in general

Why to (know how to) debug (huh ?)

- eventually, every programmer will get to a point where there is a bug in his program which is non-obvious
  - even with loads of testing (cannot simulate all real-life situations, hard to get 100% code coverage in testing) (3)
  - how to approach this problem in effective way ? (to find a root cause and correct fix)
- debugging knowledge will make you to write the program with debugging in mind → leads to faster debug-fix-develop cycle (2)
- debugging is understimated, everyone seems to focus on programming techniques but not on how to find bugs in programs (1)

(1) and, some jobs are just about finding+fixing bugs. In pure development, surprisingly debugging skills often are not needed that much but when sustaining software it is crucial.

(2) it is too late when developer realizes that the program contains horrendous amount of bugs but there is no debugging framework and logging capabilities are poor.

(3) This is valid even for semi-real-life testing with eat-your-own-food test beds.

3.2 Observing

- What to observe:
  - system calls
  - library calls
  - transactions
  - input/output data (e.g. network traffic)
  - all of the above (ideally correlated/connected together)
- How to observe:
  - capture the events
  - stop and run `(1)`

Heisenbug
is special kind of bug named after Heisenberg's Uncertainty Principle - altering the way program runs by observing it might make it harder to spot a bug which depends on timing (race conditions etc.) http://en.wikipedia.org/wiki/Unusual_software_bug

Special business (not covered):

- kernel debugging
- performance debugging

3.3 Helper tools

- code browsing/exploring:
  - useful for complex projects or areas you're not familiar with
  - there are many tools, everyone has his own preference and work style (similar to code editors)
  - web-based/terminal-based
- getting information from binaries
- peeking into transient objects (processes, lwps, memory, ...)

3.3.1 ctags

- easy to find definitions of functions/macros/variables
- very simple to use
- original ctags vs. exuberant ctags (http://ctags.sourceforge.net/)
  - limited number of supported languages (original: C, Pascal, Fortran)
  - ectags better integrated with vim
- how does it work: scans source code for definitions and generates $tag + file + regexp$ (`ex` command) for each definition found into `tags` file:

```
load_config  reload.c  /^int load_config(void) {$/
```

- ctags examples:
  - Emacs ctags:

    ```
    etags --declarations --defines --globals --typedefs *.[ch]
    ```

    original
  - original ctags:

    ```
    ctags *.[ch]
    ctags -R
    ```

- it will create output file called 'tags' and can immediately work with it
- vim setup: add the following into `/.vimrc`:
  `set tags=./tags,./TAGS,tags,TAGS,/usr/include/tags, /Source/...`
- open vim and jump to the symbol:
  `vim -t known_symbol`
- after that tab-complete via:
  `:ts symbolprefix`
- jump backwards via 'Ctrl-t'
- see tags stacks via `:tags`

3.3.2 cscope

- great for caller/callee searching
- great for learning by observing code flow
  - e.g. capture truss output (with multiple instances of -u) and go through interesting functions
- how does it work: generates index file (binary format) to `cscope.out` by going through the code (including header files)
- how-to for vim+cscope setup: http://cscope.sourceforge.net/cscope_vim_tutorial.html
- References:
  - cscope+ctags+vim setup: http://www.fsl.cs.sunysb.edu/~rick/cscope.html
  - cscope+vim: http://cscope.sourceforge.net/cscope_vim_tutorial.html
- Examples:
  1. find symbol definition:
     (a) produce `cscope.out` file: `cd $SRC; cscope -b -R`
         - some old implementations lack the -R recursive option:
           `find .  -type f -name '*.[ch]' > /tmp/list; cscope -b -i /tmp/list`
     (b) in `vim` position the cursor over a symbol

(c) hit Ctrl-@ and 'g'

2. search for a symbol
   (a) in `vim` position the cursor over a variable/function/definition
   (b) hit Ctrl-@ and 's'
3. find more about a function:
   (a) find a function definition via ctags `:ts func<tab>`
   (b) see who calls this function 'Ctrl-@ + c'

3.3.3 OpenGrok

- source code search engine: http://opengrok.github.io/OpenGrok/
- cross-reference + syntax highlighting
  - kind of similar to cscope but web-based
- indexes most common languages (C, Java, ...) and SCMs (CVS, SVN, Mercurial, ...)
- allows to search based on definitions, symbols, paths, history, full search
- really fast (even with full search across many projects)

3.3.4 Other tools

- `vimdiff`
  - useful for comparing changes in source files or log files produced by different versions of the program in question
  - even useful for watching changes in behavior (`truss` output)
  - can be very useful in cases when we don't know what exactly has changed
  - enable colored view via `:color on`
- number of small tools like `bc, od, hexdump`, ...
- combinations
  - e.g. combine dtrace/truss with graphviz and you'll get great call graphs: http://thermalnoise.wordpress.com/2007/11/14/visual-call-graph-using-dtrace/

3.4 Debugging data

formats for embedding debugging information into ELF binaries

3.4.1 stabs

symbol table strings

- largely historical
- debugging information is stored as special entries in symbol table

3.4.2 DWARF

- modern de-facto standard for representing debug data
- large space overhead, difficult to process

XXX

- References:
  - DWARF paper: http://www.dwarfstd.org/Debugging%20using%20DWARF.pdf

3.4.3 CTF (Compact C Type Format)

similar to DWARF in function but small space overhead (normally used for all userland binaries and also kernel-land). Originally introduced in Solaris, currently (2016) it is being added to FreeBSD.

- stored in ELF section header called `.SUNW_ctf`
- describes types, unions, typedefs, structures, function prototypes
- no binary to source code mapping
- manipulated by `ctfdump, ctfmerge, ctfconvert`, consumed by mdb(1), dtrace(1M)
- generated via -g compiler option from the stabs/DWARF data and replaces the data in the process (ctfconvert)
- Usually CTF is created from DWARF data

References:

- On Solaris, CTF tools are available in the `onbld` package

```
pfexec pkg install developer/build/onbld
export PATH=$PATH:/opt/onbld/bin:/opt/onbld/bin/`uname -p`
```

- http://blogs.oracle.com/levon/entry/generating_assembly_structure_offset_values
- http://blogs.oracle.com/levon/entry/reducing_ctf_overhead
- https://blogs.oracle.com/jmcp/entry/getting_started_with_your_own1

Task: *Construct library with CTF data*

Source code: `debugging/ctf`

Comment: The data types supplied by the library can be described by CTF. The program using the library does not necessarily have to support CTF for its data structures. This task is Solaris/FreeBSD specific.

Steps:

- Use `ctfconvert` and `ctfmerge` commands to supply CTF data to the library.
- Link a program with the library, call the `fillit()` function from the program. Use `ctfdump`, `elfdump` commands to verify the library indeed contains CTF data.
- Set a breakpoint at the entry to `fillit()` and verify the entry has been set

- Run the command in debugger, this time with a breakpoint set to the entry of the function `fillit()`
- print the contents of the structure with types and addresses/offsets

3.5 Resource leaks

- loose definition: resource leak is an event which happens when reference to a resource is removed without actually freeing the resource
- Most common: http://en.wikipedia.org/wiki/Memory_leak
  - memory leak (1) is just one kind of resource leak - a process can also leak descriptors, objects/structures, files, ...
- Why it happens (in environment without garbage collector):
  - oversight (insufficient analysis) / careless programming
  - unexcersized code path or unexpected combination of configuration options (testing/-code coverage again), most commonly error paths (2)

(1) by memory we mean the heap segment of a process.

(2) "Snowball effect problem" (smallest possible case, imagine similar case with more allocations and complex dependencies):

```
/* pseudo-code */
foo = malloc(sizeof (struct foo_t));
/* foo needs to be processed first */
if (process(foo) < 0) { /* failure */
      free(foo);
      return (-1);
}

bar = malloc(sizeof (struct bar_t));
/* bar will be a clone of foo */
if (clone(foo, bar)) < 0) { /* failure */
      /* GAH, something is missing here */
      free(bar);
      return (-1);
}
```

3.6 libumem

- full-blown memory allocator (alternative to the allocator from libc)
- suitable for debugging memory leaks (1)
- minimal overhead (can be even used in production)
- has pedigree of the slab allocator (3) (but in userland) (2)
- no application modification necessary

  - just LD_PRELOAD the libumem.so library and set UMEM_DEBUG environment variable

- scalable allocator (slabs) with memory debugging
  - can diagnose memory leaks, double free's, used free'd memory etc.

(1) umem_debug(3MALLOC) man page

(2) Per object type caches from which objects are allocated. Freeing of an object is a matter of returning it to the cache. If object constructors are used, the object is returned from the cache in initialized state and also must be returned to the cached in that state. For more details see the umem_cache_create(3MALLOC) man page.

(3) The Slab Allocator: An Object-Caching Kernel Memory Allocator (1994, USENIX), Jeff Bonwick http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.4759

- References:
  - umem_debug(3MALLOC)
  - http://blogs.oracle.com/ahl/entry/solaris_10_top_11_20
  - http://blogs.oracle.com/jwadams/entry/debugging_with_libumem_and_mdb
  - http://blogs.oracle.com/roller/page/jmcp?entry=on_findleaks_wot_is_it
  - http://blogs.oracle.com/amith/entry/detecting_memory_corruption_with_libumem

3.6.1 How does libumem work

- replacing default heap allocator in libc (preloading or directly linking against `libumem.so`)
- protecting the buffer with patterns which are checked inside libumem's malloc/free
  - patterns:
    * `0xbaddcafe` is free'd memory
    * `0xdeadbeef` is uninitialized variables
    * `0xfeedface` redzone (after the actual buffer contents)
- recording the log of stack traces for each allocated buffer `(1)`
- last piece: debugger going through global variables, constructing reachability graph and finding unreachable addresses at the end

(1) makes it easy to identify who allocated that buffer

3.6.2 Using libumem+mdb to find memory leaks

- generic approach:
  - run the program with preloaded `libumem.so` and environment variables which trigger special behavior of the library
    `LD_PRELOAD=libumem.so UMEM_DEBUG=default ./prog`
  - attach debugger and use `::findleaks` to get memory leaks: `(2)`

```
mdb -p <pid_of_prog>
::findleaks
```

  - get the stack trace history of leaked buffers via `::bufctl_audit (3)`

- the view of the memory has to be consistent → need to have the program in quiescent state (1)

(2) alternatively, do it in one step: set the environment variables and run the program from the debugger, load libumem and stop it in a function:

```
export LD_PRELOAD=libumem.so
export UMEM_DEBUG=default
mdb ./prog
::load libumem.so.1
::bp foobar
:r <program_arguments>
```

(3) can get stack trace easily:

```
> ::findleaks
CACHE     LEAKED   BUFCTL CALLER
00076508       1 00097158 libc.so.1`strdup+0xc
------------------------------------------------------------------------
   Total       1 buffer, 24 bytes
> 00097158::bufctl_audit
            ADDR          BUFADDR        TIMESTAMP           THREAD
                             CACHE          LASTLOG         CONTENTS
            97158            91f48    6696e428fce34                1
                             76508                0                0
                  libumem.so.1`umem_cache_alloc+0x144
                  libumem.so.1`umem_alloc+0x58
                  libumem.so.1`malloc+0x28
                  libc.so.1`strdup+0xc
                  get_alg_parts+4
                  pluck_out_low_high+0x10
                  yyparse+0xe80
                  config_update+0x58
                  config_load+0x4c
                  main+0x474
                  _start+0x108
```

(1) Possible solutions:

  - stop the program temporarily (`SIGSTOP` / Ctrl+Z)
  - check for the leaks on the way to exit:

```
$ LD_PRELOAD=libumem.so
$ export LD_PRELOAD
$ UMEM_DEBUG=default
$ export UMEM_DEBUG
$ /usr/bin/mdb ./my_leaky_program
> ::sysbp _exit
> ::run
```

```
mdb: stop on entry to _exit
mdb: target stopped at:
libc.so.1`exit+0x14:    ta        8
mdb: You've got symbols!
mdb: You've got symbols!
Loading modules: [ ld.so.1 libumem.so.1 libc.so.1 ]
> ::findleaks
```

- use gcore(1)

3.6.3 How does `::findleaks` work

- why is `::findleaks` needed at all ? wouldn't it be sufficient just to match `malloc()/free()` calls ? 2 disadvantages:
  - space - necessary to log all the calls
  - need to watch the process during whole life cycle `(1)`
- `::findleaks` process:
  1. start with static data
  2. grep memory for references to allocated blocks

(1) assumes the process has a cleanup routine which is run on the way to exit (e.g. server process which accumulates data over time).

- How does ::findleaks work http://blogs.oracle.com/jwadams/entry/the_implementation_of_findleaks
- More tricks:
- `::umem_debug` - enable debugging in libumem (umem_debug())
- `::findleaks -f` - reinit full leak search (normally the results are cached)
- `::umem_status` - shows heap corruptions in detail

Task: *Resource leak debugging*

Source code: `debugging/resource-leaks/`
Comment: This piece of code resembles configuration reloading done in system daemons, including errors.
Steps:

1. See the source of `reload.c` and find all resource leaks.
2. compile
   `make clean && make`
3. There are resource leaks happening during restart
   `kill -HUP 'pgrep leaky'`
4. Use your favorite tools to discover the resource leaks.
5. Switch to different operating system and try to find the leaks with tools available there (preferably completely different tool chain). Compare the differences between various tools and approaches.

6. Fix the leaks, use the tool to verify they are indeed gone.

3.7 watchmalloc

- Debugging library in Solaris similar to libumem, actually enforces the boundaries and correct memory allocation (i.e. detects double `free()`, write past the end of the buffer, etc.)
- has a mode which uses `/proc` watchpoint facility (`SIGTRAP` on invalid access)
  - `WATCH`: free'd blocks are write protected
  - `RW`: the area before and after the buffer (called *red zone*) and free'd blocks are read/write protected

The cost of debugging with watchmalloc

has significant overhead (even without watchpoints)

Task: *watchmalloc experimentation and inspiration*

Source code: `debugging/watchmalloc/`

Comment: The watchmalloc library can be used as inspiration for writing your own heap allocator with detection capabilities.

Steps:

1. compile
   ```
   make clean && make
   ```
2. run with malloc from libc
   ```
   ./underoverflow 10 -2 16
   ```
3. run with watchmalloc and no write
   ```
   LD_PRELOAD=watchmalloc.so.1 MALLOC_DEBUG=WATCH
   ./underoverflow 10 -2 16
   ```
4. run with watchmalloc with read guards and write
   ```
   LD_PRELOAD=watchmalloc.so.1 MALLOC_DEBUG=RW  ./underoverflow
   10 -2 16 40
   ```
5. Experiment some more, try to read/write before the beginning of the buffer, etc. What is the granularity of detection ? Can watchmalloc detect off-by-one reads/writes ?
6. test double free with watchmalloc
   ```
   LD_PRELOAD=watchmalloc.so.1 MALLOC_DEBUG=WATCH ./badfree 10
   ```
7. Write your own simple heap allocator (using either `brk()` or `mmap()` with checking. Use `mprotect()` to protect metadata of each buffer. Implement double free detection.
8. Produce unit tests for your allocator.

- References:

  - http://blogs.oracle.com/wfiveash/entry/playing_with_solaris_memory_debuggers
  - http://blogs.oracle.com/peteh/entry/hidden_features_of_libumem_firewalls
  - http://blogs.oracle.com/eschrock/entry/watchpoints_features_in_solaris_10

3.8 Call tracing

- syscalls: truss(1)/strace(1)
  - both work across fork(2) with `-f`
- library calls: truss(1)/ltrace(1) `(3)` `(4)`
  - to observe calls to functions in the executable `(1)`
- invasive observation: both truss and strace stop the program for a while
- how does it work: manipulate with lwp via the `/proc` interface `(2)`

(1) Example: get all calls done by functions in `nc` binary and also by functions from `libc.so` library which have `get` prefix:

```
truss -u a.out -u libc::get* nc www.mff.cuni.cz 80
```

(2) truss uses `libproc.so` to access `/proc`.

(3) truss can trace also library calls:

```
truss -f -ulibcrypto::pk11_* -upkcs11_softtoken:: \
    `pgrep traced-app` > /tmp/app.truss 2>\&1
```

(4) trace calls in the program itself: `truss -u a.out`

- note on `truss` and its data representation. If you truss a "hello world" program, you will see something like this at the end:

```
Hello world.
write(1, " H e l l o   w o r l d .".., 13)      = 13
_exit(0)
```

Ie., you see spaces within the strings. The thing is that `truss` needs 2 characters to represent a byte. It uses hexadecimal representation for non-printable characters, ascii for printables chars, and escapes where possible (`\n`, `\r`, etc.). So, if you print three characters: 0x01, the new line character (0x0a), and X (0x58), you will see this:

```
write(1, "01\n X", 3)                           = 3
```

Task: *Construct your own strace*

Source code: `debugging/ptrace/strace.c`

Comment: `strace/truss` basic functionality can be constructed using `/proc` or `ptrace()` interfaces. Getting the syscall number and return code is architecture dependent, it is necessary to find out where (e.g. in which registers) the data is stored.

Steps:

1. Adapt the source code to trace system calls of a program. The output should include syscall number and return code, i.e. something like this:

```
        syscall #21 called
        syscall #21 returned with 0
```

3.9 Using /proc

- p-commands (proc(1)):
  - pstack
  - pfiles
  - pcred
  - pldd
  - pmap
- Exmaple: check for file descriptor leaks:
  1. stop the process `(1)` via pstop(1)
  2. watch the open descriptors via pfiles(1)
  3. see the stack via pstack(1)
  4. continue the run via prun(1)
  5. jump to step nr. 1

(1) p-commands can operate on LWPs and core files

Example:

```
    $ sleep 20 \&
=====> remember the PID
    [1] 7407
=====> stop the process
    $ pstop 7407
=====> wait a bit too see it's really stopped
    $ sleep 20
=====> see the userland stack
    $ pstack 7407
=====> unbrake the process again
    $ prun 7407
```

- How does it work ?
  - via proc(1). Use the above example with:
    `truss pstop 7407`
  - NOTE: pstop(1) can even stop a single lwp
- References:
  - pfiles can display filenames: http://blogs.oracle.com/eschrock/date/20040712#nuts_and_bolts_of_pfiles
  - http://blogs.oracle.com/ahl/entry/the_solaris_10_top_111
  - pmap can display libraries and regions for thread stacks: http://blogs.oracle.com/ahl/entry/number_18_of_20_pmap

3.10 Debugging dynamic libraries

- ld.so.1(1)
- Run-time linker and link-editor have shared built-in debugging capability:
  `LD_DEBUG=help /bin/ls`

Example: using `LD_DEBUG`:

```
LD_DEBUG=symbols,bindings /usr/bin/openssl speed \
  rsa1024 -engine pkcs11
```

- Note: can use ld.so.1 for running a program:

```
/lib/ld.so.1
/lib/ld.so.1 /bin/ls
/lib/64/ld.so.1 /bin/ls
/lib/64/ld.so.1 /usr/bin/amd64/openssl version
```

3.11 Debuggers

- Kinds of debuggers:
  - generic - able to debug both userland/kernel (gdb/mdb/adb)
  - in-kernel (kmdb)
  - userland (dbx)
- How userland debugger works:
  - ptrace(3C) / proc(4) `(1)`
    - start/stop: `PCSTOP` et al., `PCRUN`
    - single stepping: `PRUN` + `PRSTEP`
    - breakpoints: `PCWRITE` with trap instruction `(2)`
    - watch points: `PCWATCH`
    - read/write process memory: `PCREAD/PCWRITE`

(1) ptrace() is just a proc(4) wrapper in Solaris, regular syscall in Linux (ptrace(2))

(2) `0x91d02001` for SPARC, `0xcc` for i386/AMD64,

- References: Crash dump analysis lecture (NPRG050) http://dsrg.mff.cuni.cz/teaching/nprg050/

3.12 Symbol search and interposition

- default model symbol search can be quite complicated
- Symbol search:
  - each reference to the symbol by given object leads to a search
  - first found instance wins
  - ldd(1) reports the order in which the search will be done
  - each symbol search starts from the application

- Interposition:
  - foo() definition in bigfoo.so.1 interposes on foo() definition in smallfoo.so.1

Example: `debugging/symbol-search/`

1. see thee source files and try to guess:
   - which functions from which files will be called ?
   - what will happen ?
   - how many times we lookup symbol 'bar' ?
2. compile
   `make clean && make`
3. observe the library dependencies:

   ```
   ldd ./prog
   ldd ./bigbar.so.1
   ```

4. run:
   `./prog`
5. see what happened:
   `LD_DEBUG=symbols,bindings ./prog`

3.13 dtrace

- dynamic instrumentation
- ...
- Available on: Solaris, FreeBSD, Mac OS X, QNX
- performance/lock contention debugging
- References:
  - dtrace guide
  - The Solaris Dynamic Tracing (DTrace) Guide:
  - dtrace(1M) man page

4 Terminals

4.1 Terminal I/O Overview

- Terminal is an input/output device.
- You can read from and write to it.
- Historicaly, it was a hardware device.
  - Connected to the computer via a serial line.
  - There was no stderr as we know it now.

  - A keyboard and a paper roll for displaying text (no video displays at the time).
  - Then, video displays came in (AKA "glass TTYs").

- Whatever is displayed on the terminal comes from the terminal driver, ie. what you type goes right over the serial line to the computer, the input might be processed in a few different ways, and the output goes back to the terminal to be displayed – unless the echo is "off". **The terminal does not display directly what you type, it always goes through the terminal driver code running on the computer.**
- Also note that RS-232, a commonly used standard for serial communication, is a fully duplex protocol.

4.2 Terminal I/O Overview (cont.)

- today, a *terminal* usually means a *text terminal*, not a *physical terminal*
  - a text terminal usually runs under a graphical environment
- some byte sequences have special meaning
  - moving the cursor, clearing the screen, etc.
  - try `vim >output` and see what is in `output`
    - do not forget to type ":q" as well
- different terminals have different capabilities
  - some cannot clear the screen, for example
  - types of terminals: ansi, vt100, xterm, ...
- You probably know that we can also have virtual terminals on a console. Most unix-like system support that. Usually you can switch between virtual terminals by hitting `Alt + Fx` where `Fx` is a function key.
  - Virtual terminal configuration varies. On BSD systems, you have file `/etc/ttys` to set what virtual (and network) terminals you have.
- UNIX windowing (ie., graphical) environments did not eradicate the terminal interface since it is very convenient to deal with.
- Some applications might even refuse to run without a terminal – screen oriented editors, for example.
- When running a terminal in the graphical environment, `xterm`, for example, then that is the application that gets the keyboard input. It then processes the input and sends the filtered data to the shell through a pseudo terminal. Do not worry about pseudo terminals now, will be explained later.
- You have to distinguish what is supported by the terminal, the terminal driver itself, and what is done in the shell. For example, the terminal driver offers by default ^D for "End-Of-File", meaning that it will close its output (usually, shell's input) when ^D (binary `04`) is read from the terminal, or ^S which suspends the output from the driver. However, `bash` itself, for example, accepts ^L to clear the screen. That means that ^L goes through the terminal and the terminal driver to the shell which interprets it and sends a "clear the screen" sequence back to the terminal, if the terminal itself supports the feature. ^L is not interpreted by the terminal driver in contrast to ^D or ^S. More information is on page 40.

- When you hit ^C, that character is sent by the terminal application (eg. `xterm`) which is interpreted by the terminal driver which generates the `SIGINT` signal to the application. That means that neither the terminal application nor the shell generates that signal. You can verify it like this – start `cat` in your shell. Then, `truss` your `xterm` (not shell! Remember, `ptree` is great on Solaris for finding out child-parent relationships. On Linux distros, try "`ps --forest`") or any other terminal you use. Move your mouse to the terminal again. You can press `Ctrl` itself several times to see that there is some communication between the terminal and X over the X11 socket; use `pfiles` to check which file descriptor is the X11 socket. Then, type ^C. You should see something like this:

```
write(4, "03", 1)                                  = 1
ioctl(3, FIONREAD, 0x08047A50)                     = 0
pollsys(0x080479D0, 2, 0x00000000, 0x00000000)     = 1
read(4, " ^ C", 1024)                              = 2
read(4, 0x08074C4A, 1022)                          Err#11 EAGAIN
```

  The terminal writes `03` to the terminal because that's what ^C is. `03` means 1 byte of value 3. Note that ^E is `05` then, ^L is `12` etc. Be careful that `truss` shows some characters using escapes so ^J is `\n` (`0A`, new line), ^L is `\f` (`0C`, form feed), and ^M is `\r` (`0D`, carriage return). What you get from the terminal is literaly "^C" in two characters, and that's what you see in your terminal, literaly. Shell does not send it, that's the (pseudo) terminal driver. More on this later.

4.3 Physical (Hardware) Terminal

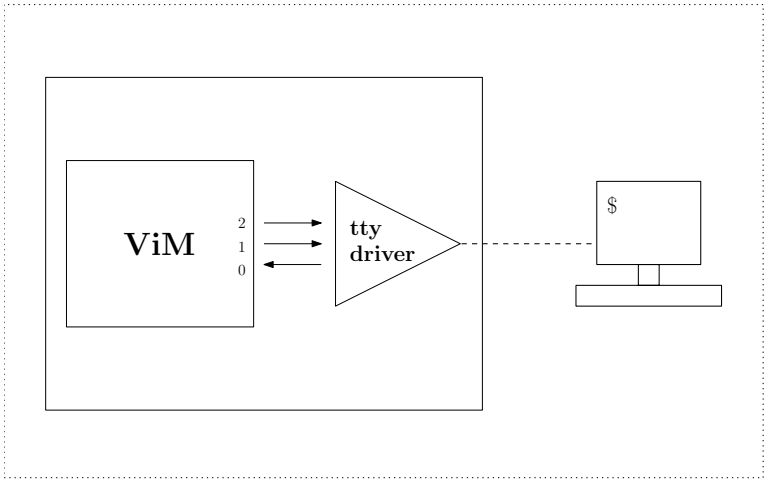
- Picture in public domain downloaded from Wikipedia.
- Note that while it looks like a "normal" computer it is not. It just communicates with the "real" computer over a serial line.
- As already mention, remember that echo (ie. what you see when you type) on the terminal display is not done by the terminal itself. The terminal can output only what it gets from the other side, meaning that the "remote" system (= the terminal driver) itself does that echo for you. This can be switched off via the `tcsetattr` function, which will be introduced later. You can use "`stty -echo`" from the command line to switch off the echo, and "`stty echo`" to switch it on again.
- Even these days, you might need to use your computer to emulate a physical terminal. Some devices like Cisco routers and switches might still need to be configured with the IP address and the gateway using the serial console before you can log it to it. You can use `tip(1)` (usually shipped by default with your UNIX or unix-like system) or myriad of other comms like `Minicom` etc.

4.4 `stty`(1) command

- changes and prints the terminal line settings

- `stty -a`
  - lists the settings in a human readable form
- example
  - `stty -echo` disables echo
  - `stty echo` enables echo
- maybe most useful example of this command
  - `stty sane`
  - for example, if you paste some rubbish by mistake, and your terminal starts acting weird, the `sane` option often helps.
    - not every time though
- We will use this command in other examples.

4.5 TTY Driver Connected To a Phy Terminal

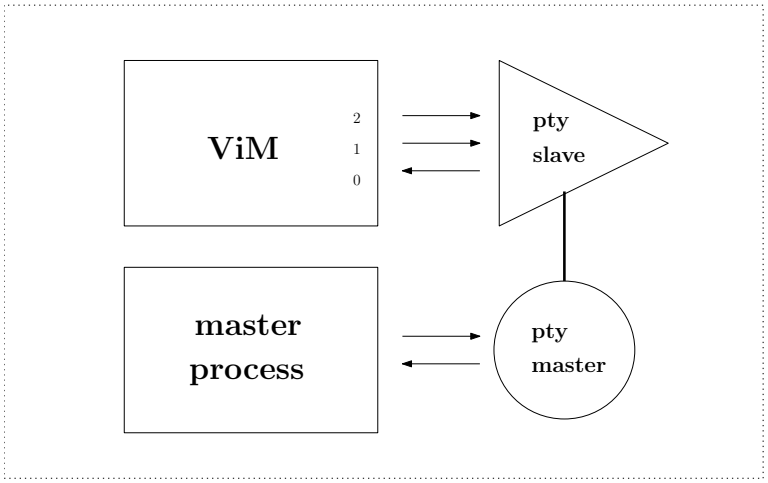


- The ViM editor communicates with the terminal driver, the driver takes care of the communication with the actual physical terminal.
- This is how it also looks when you run ViM from the virtual console.

4.6 Pseudo terminals Overview

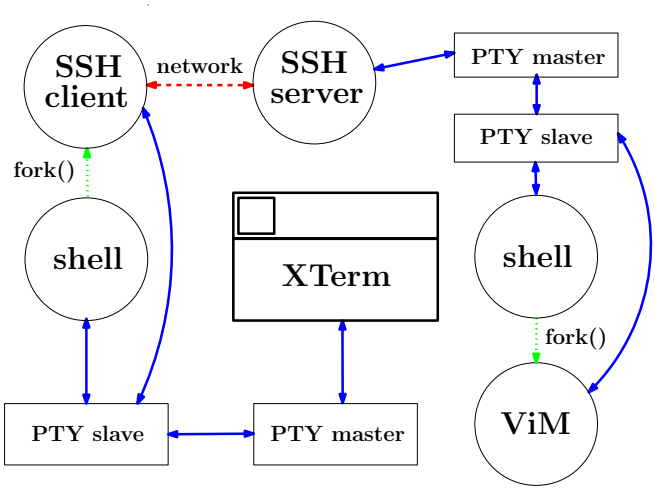
- pseudo terminal is an emulation of a physical terminal
- **allows one process to control another process that is written for a terminal**
  - remotely running ViM over an SSH connection, for example
  - such application (SSH) must support it in order this can work
- you can have multiple (text) terminals on your desktop
- an application written for a terminal does not care (more or less) whether it communicates with a physical or a pseudo terminal
- The pseudo terminal device driver acts just like a terminal as far as the interactive (slave) process is concerned. However, the other end is connected to a master process, not to the physical device. The master process can read from the pty master what the application wrote to it, and can write to the pty master what it wants the application to read from the slave pty.
- In other words, the master side of the pseudo terminal represents the physical terminal, the slave part represents the special device end point.
- The pseudo terminal can not be just one point used by both processes. It is the same as a (bidirectional) pipe - you need 2 ends so that the processes can communicate with each other.

4.7 PTY Driver Connected To a Process



- The master process can be `xterm`, for example, running a shell which started the ViM editor. More processes can work with the same terminal, as we will see later.
- **The master process and the master PTY together emulates the physical terminal.**
- How to get the pseudo terminal and how to use it in your program will be explained later in the chapter.
- How exactly is the slave and the master part "connected" together in the kernel does not have to be of a programmer's concern.

4.8 More Complex Example



- The picture shows a situation where an SSH client running from a shell in a terminal window is connected to the remote side, and running ViM there.
- Note that ViM is not communicating over the shell. It is reading and writing to the same terminal as the shell is. It is about process groups and the controlling terminal as we will see later in the chapter.
- The communication between the 2 pseudo terminal ends is bidirectional (in other words, it's full duplex).
- Question: when user runs ViM over SSH, what is the settings of the slave pseudo terminal devices on remote and local side?
  - Answer: will be the same because SSH transmits all terminal settings to the other side, see RFC 4254 above.
- Example:

```
# Get the local pseudo terminal name, remotely log in using
# SSH, get the remote pseudo terminal name there as well,
# and then run ViM.
```

```
$ tty
/dev/pts/5
$ ssh localhost
Password:
$ tty
/dev/pts/8
$ vim

# And now from another console, check the settings of both
# slave pty's.
$ /usr/gnu/bin/stty --file=/dev/pts/5
speed 38400 baud;
eol2 = M-^?; swtch = <undef>; min = 1; time = 0;
-icrnl
-opost
-isig -icanon -iexten -echo -echoe -echok

$ /usr/gnu/bin/stty --file=/dev/pts/8
speed 38400 baud;
eol2 = M-^?; swtch = <undef>; min = 1; time = 0;
-icrnl ixany
-onlcr tab3
-isig -icanon -iexten -echo -echoe
```

4.9 Reading From a Terminal

- normally, the application uses descriptors 0, 1, and 2
  - those may or may not be connected to a terminal
  - if not, user probably redirected those, eg.
    `cat /etc/passwd > my_passwd`
- an application can explicitly open `/dev/tty`
  - a synonym for a controlling terminal for the process
  - the reason to do that might be to read a password and bypassing any redirections
- an app can (try to) open actual terminal name (eg., `/dev/tty02`)
- Applications usually do not allow to read a password from a redirected file for security reasons. However, they might provide a way to do it via a separate program. For example, look for `SSH_ASKPASS` in `ssh(1)` manual page.
- See `terminals/getpassphrase.c`, read the comment inside. The program will use the terminal no matter how you redirect the standard input. And, if you have no terminal at all, it will fail the way as documented in the manual page for `getpassphrase(3C)`.
- Shell command `tty(1)` prints the controlling terminal:

```
$ tty
/dev/pts/15
$ pfiles $$
```

```
2256:   bash
  Current rlimit: 256 file descriptors
   0: S_IFCHR mode:0620 dev:342,0 ino:453881557 uid:1629480 gid:7 rdev:24,15
      O_RDWR|O_NOCTTY
      /dev/pts/15
   1: S_IFCHR mode:0620 dev:342,0 ino:453881557 uid:1629480 gid:7 rdev:24,15
      O_RDWR|O_NOCTTY
      /dev/pts/15
   2: S_IFCHR mode:0620 dev:342,0 ino:453881557 uid:1629480 gid:7 rdev:24,15
      O_RDWR|O_NOCTTY
      /dev/pts/15
   3: S_IFDOOR mode:0444 dev:351,0 ino:50 uid:0 gid:0 size:0
      O_RDONLY|O_LARGEFILE FD_CLOEXEC  door to nscd[328]
      /var/run/name_service_door
 255: S_IFCHR mode:0620 dev:342,0 ino:453881557 uid:1629480 gid:7 rdev:24,15
      O_RDWR|O_NOCTTY FD_CLOEXEC
      /dev/pts/15
```

4.10 Reading From a Terminal (cont.)

- normally, an application gets input in lines because the terminal driver does not send the data up until "Enter" is hit.
  - unless the input/output is redirected to/from a non-terminal
- no arrow keys, deletes, backspaces get through to the application. All that is handled in the driver.
  - it is said that terminal is in the *canonical mode*
- Ctrl+D at the beginning of the line means "End of File". `read()` then returns 0 to indicated it.
- For more information on terminal modes, see page 40.
- See examples (again, read the comments): `terminals/simple-read.c` and `terminals/simple-tty-read.c` The 2nd one does similar thing to what `getpassphrase(3C)` does. Try without a terminal to see that.
- `O_NONBLOCK` can be used as usual. Either with `fcntl()` on the file descriptor, or directly with `open()` in which case it also affects `read()` as well. See [unix-prog] for more info.
- Do not forget that `select()` and `poll()` calls are useful when there is a need to read from more file descriptors without busy waiting.

4.11 Sessions and Process Groups (Jobs)

- a new *session* is created when the user logs in
- that new session consists of one *process group*
- the only process in the process group is the login shell
  - that is the simple scenario with no windowing environment

- the process is a *process group leader* and a *session leader* as well
  - its PID is also a proces group ID and a session ID
- that's how it is today. The history of this was quite colorful and diverse.
- Some of that was in [unix-prog].
- Example to show that my login shell is both the session leader and a process group leader ("-j" prints session and process group ID as well):

```
$ ssh $(hostname)
Password:

$ ptree $$
503   /usr/lib/ssh/sshd
  4117  /usr/lib/ssh/sshd
    4118  /usr/lib/ssh/sshd
      4124  -bash
        4160  ptree 4124
$ ps -j -p 4124
  PID  PGID   SID TTY          TIME CMD
 4124  4124  4124 pts/10       0:00 bash
```

- The pseudo terminal slave, ie. the end used the by the terminal application (`bash`), is `/dev/pts/10`.
- And, if we use `pfiles` on the right `sshd` process, we can see that the pty master end of the pseudo terminal is also 10 (note "`rdev:23,10`").

```
  11: S_IFCHR mode:0000 dev:341,0 ino:56712 uid:0 gid:0 rdev:23,10
      O_RDWR|O_NONBLOCK|O_NOCTTY|O_LARGEFILE
      /devices/pseudo/clone@0:ptm
```

- Remember that you can kill a process group with `kill(2)` by using a negative number where the number is a process group PID. You can do the same with `kill(1)` command: "`kill -- -<PGPID>`". Note that you must not forget to use "`--`" to stop the command to interpret the process group PID as an option. Killing the whole process group may come in handy if you want to kill a big process group like a parallel system build with tens of processes, etc.
- Try to run "`sleep 991 | sleep 992 | sleep 993 | sleep 994`" and play with this process group. You can kill it as a group using the negative number, you can try to kill individual process and see what happens. Examining the behaviour of `bash`, for example, it will not notify you that the job has finished until all processes of the group have finished. Korn Shell 93 (`ksh93`), on the other hand, waits on the last process only. You can also observe that there is a difference in how processes are created. `bash` is the father of all processes and starts creating those from left to right, Korn shell creates the last process in the pipe line and all other processes in the job are created by it. Given that Korn shell creates just one process, the group leader, it must create the last one in the pipe since that assures that shell will wait for the whole pipe to get completed.

- **Exercise:** use Bash and Korn shell with the "sleep" job above and verify the above mentioned information. With the Korn shell, use `/bin/sleep` instead of `sleep` since the latter is a built-in command in KSH so you would see only `ksh` in the process listing. However, you need the seconds to distinguish between individual processes.

4.12 Sessions and Process Groups (cont.)

- the terminal the user logged in becomes a *controlling terminal* of the session
- the session leader is also a *controlling process*
- if *job control* is enabled, every command or a pipeline forms a new process group
  - all such groups have the same controlling terminal, unless the application explicitly changes that
- See this commented output:

```
# Let's see what is in our process tree.
$ ptree $$
503   /usr/lib/ssh/sshd
  4117  /usr/lib/ssh/sshd
    4118  /usr/lib/ssh/sshd
      4124  -bash
        4283  ptree 4124

# Let's run the 1st job, on the background.
$ while true; do sleep 1; echo X; done | wc -l &
[1] 4291

# Let's run the 2nd job, on the background as well.
$ sleep 999 &
[2] 4303

# Let's see what jobs we have.
$ jobs
[1]-  Running   while true; do sleep 1; echo X; done | wc -l &
[2]+  Running   sleep 999 &

# Now, let's see our process tree again.
$ ptree $$
503   /usr/lib/ssh/sshd
  4117  /usr/lib/ssh/sshd
    4118  /usr/lib/ssh/sshd
      4124  -bash
        4290  -bash
          4316  sleep 1
        4291  wc -l
        4303  sleep 999
        4317  ptree 4124
```

```
# The following bash process is the process group leader for
# the while loop (note that 'while' is a shell internal
# command so that's why we see 'bash', not 'while' there).
# Also note that all those processes share the same login
# session, with session ID 4124.
$ ps -j -p 4290
  PID  PGID   SID TTY          TIME CMD
 4290  4290  4124 pts/10   0:00 bash

# Note that 'wc' is a member of the above mentioned process
# group as well and it is NOT a process group leader (bash
# with PID 4290 is).
$ ps -j -p 4291
  PID  PGID   SID TTY          TIME CMD
 4291  4290  4124 pts/10   0:00 wc

# However, 'sleep 999' was run as its own job so it formed a
# new process group, with one process in it (sleep) and
# sleep is the process group leader as well. You can also
# see that from the ptree(1) output that we cannot assign
# the commands listed to their respective groups, we must
# use ps(1) to get that additional information.
$ ps -j -p 4303
  PID  PGID   SID TTY          TIME CMD
 4303  4303  4124 pts/10   0:00 sleep
```

4.13 Controlling Terminal

- a session can have one controlling terminal. Daemons usually do not have it.
- it is the **first terminal** that a session leader (and a session leader **only** can do that) opens after the new session was created
- the session leader that establishes the connection to the controlling terminal is called a controlling process
- terminal input and terminal generated signals go to the foreground group only
- if modem (or network) disconnect is detected by the terminal interface, `SIGHUP` is sent to the controlling process or group
- A new session is created via calling `setsid(2)`, see p38.
- As to differences between systems and the `SIGHUP` signal, see p35.
- The above says that there are sessions without a controlling terminal, those sessions usually contain daemons, server processes, etc.
- The above also says that in order to get rid of a controlling terminal, you have to create a new session. It is not enough to close the controlling terminal – which would usually mean to close stdin, stdout, and stderr. See `terminals/close-the-tty.c`. And, if you want to make sure you will never get a controlling terminal in the session, let the session leader fork and exit. Since only the session leader can acquire the controlling terminal, and

we have no session leader anymore (just its child), we are safe. Actually, that's how the `daemon` call usually works.

- If the session leader opens a terminal with `O_NOCTTY`, such terminal will not become the controlling terminal.
- You cannot open `/dev/tty` to get a controlling terminal, `/dev/tty` is a synonym for an already existing controlling terminal.
- How to check that if you Ctrl-C your application the signal will go to the foreground group and not to the shell? That's easy, get the PID of the shell, run "`sleep 999`", run "`truss -p <PID>`" on the shell, and then ^C the `sleep` process. You will see that the shell will get `SIGCLD`, not `SIGINT`.
- **Exercise:** write a very simple `tty` program. Read what exactly `tty(1)` does and then use `truss` to find out what function to call. It's basically a one-liner. The solution is in `terminals/tty.c`.

4.14 Job Control

- the fact that a process group runs in the background does not mean that it cannot use the terminal
  - however, it can only write to it, cannot read from it
- the "job control" means that certain signals are only sent to the foreground process group
  - `SIGINT` (`^C`), `SIGQUIT` (usually `^\`, quits and generates a core dump), and `SIGTSTP` (usually `^Z`, suspends the process)
- the "control" part of job control means that we can move jobs back and forth between foreground and background, kill them independently of other jobs, etc.
- job control as such needs a system support for process groups
- The *foreground process*, or the *foreground group* if there are more than one process in the group, is defined as process or a group that has an unlimited access to the controlling terminal.
- The *controlling group* is a group associated with the terminal. If job control is enabled, it's the same as the foreground group.
- Every shell may do that differently but usually internal commands for job control are "jobs", "fg", and "bg". Also, internal "kill" command can work with job IDs. See manual page for your shell.
- See the following bash example on `kill` shell internal command:

```
$ sleep 999 &
[1] 5760
$ jobs
[1]+  Running                 sleep 999 &
# use '%' with the job ID to kill a specific job
$ kill %1
[1]+  Terminated              sleep 999
```

4.15 What Happens If Job Control is Disabled?

- in other words, how it was before job control was introduced...
- you can still run processes in the background
  - but it means that the shell just does not wait for them
- all processes are part of the same process group as the login shell
- signals are sent to all processes
- note that choosing a job control might be a compile time option of your shell
  - bash has it like that
- Note that when bash is built with "`configure --disable-job-control`" (which is not the default setting, obviously), all commands started in the background will be spawned with `SIGINT` and `SIGQUIT` ignored. That means that ^C will not kill them and will kill only the pipeline in the foreground. That is what a user would most probably expect. Killing background processes with ^C could really suprise the user.
- **Exercise:** build Bash w/o the job control and verify the above stated information. Use `psig` command on Solaris to see what signal handlers are installed for commands started in the background.

4.16 `SIGHUP` Signal

- if the terminal hangs up or is disconnected, the controlling process (shell) gets a `SIGHUP` signal (SVR4)
  - or, the whole controlling group gets the signal (BSD)
- usually, `SIGHUP` terminates the shell
  - in SVR4 though, the shell itself is expected to send `SIGHUP` to to **foreground group** before it exits
- the shell **may** send `SIGHUP` to all processes in the session then
  - that's why you might need to start remote jobs with "`nohup <job>`" if you want them to survive the logout
  - `ksh` and `bash` do that, not sure about other shells
- On Solaris, try the following. Start two `sleep` processes, one in the foreground, one in the background:

```
$ echo $$
4104
$ sleep 888 &
[1] 4150
$ sleep 999
```

and see how the process tree looks like from another terminal:

```
$ ptree 4104
...
...
  2659  /usr/local/bin/rxvt -fn fixed
     4104  bash
       4150  sleep 888
       4154  sleep 999
```

Now, truss both `sleep` processes and the shell as well. When you kill the terminal window which simulates the terminal disconnect, you will see that the foreground `sleep` processes got `SIGHUP`s from the shell:

```
$ truss -p 4154
pause()                          (sleeping...)
    Stopped by signal #24, SIGTSTP, in pause()
    Received signal #15, SIGTERM, in pause() [default]
      siginfo: SIGTERM pid=4104 uid=1629480
pause()                                          Err#4 EINTR
```

The `sleep` process in the background continues to run. However, the shell gets the signal from the kernel (the terminal driver) since there is no PID, and then it sent the signals to all existing process groups:

```
$ truss -p 2881
waitid(P_ALL, 0, 0x08047AA0, WEXITED|WTRAPPED|WSTOPPED|WCONTINUED) (sleeping...)
    Received signal #1, SIGHUP, in waitid() [caught]
...
...
kill(-2716, SIGHUP)                              = 0
kill(-2735, SIGHUP)                              = 0
```

- Note that there is a difference between a shell getting a `SIGHUP` or a shell exiting. For example, if `bash` exits, it sends a `SIGHUP` to the controlling group only (if you type `exit`, the controlling group would be empty since no foreground process is running). However, if you set the `huponexit` option; it sends `SIGHUP` to all processes. You can verify it easily – start `sleep 888` in the background, truss it from another terminal, and exit the shell. The terminal window disappears but the `sleep` process continues to run without getting any signal.
- There are significant differences between systems, expecially between older systems, about how they deal with terminals, process groups, and such.
- SVR3 (1986)
  - the session usually consisted of 1 process group, and terminal could be associated with only one group. Thus, if a new group was created, the (new) process leader was disconnected from the controlling terminal and had to open another one.
  - however, it could still use it through already open file descriptors, no `SIGHUP` would be sent on death of the leader though (see below).

  - also, process group could not close the controlling terminal and later allocate another one (screen(1) would not work there)
  - on death of the group leader the kernel sends `SIGHUP` to all processes in the group, those processes lose the terminal and also the group ceases to exist (group ID of the processes is zeroed out)
  - when the terminal is disconnected the driver sends `SIGHUP` to all processes in the session
  - no job control possible
  - many other differencies from what we are used to today
- 4.3BSD (1986)
  - a process can change its group ID
    - thus we can have process groups without a leader
  - more groups can share the same controlling terminal
    - but the terminal can control just one group
  - background process gets `SIGTTIN` on read (will suspend it by default)
  - background process can write to a terminal (or get `SIGTTOUT` if the terminal is configured that way)
  - when the terminal is disconnected the driver sends `SIGHUP` to the controlling group (only)
  - on session end, `vhangup()` is used on the controlling terminal to traverse the process table, all the terminal entries are made unusable, then it `close()` the terminal, and `SIGHUP` is sent to the controlling group.
    - that's why on, for example, FreeBSD all background processes survives the logout.
  - however, there is no notion of a controlling process as it was in SVR3.
- SVR4 (1990)
  - have sessions and process groups
  - controlling terminal is associated with a session and a foreground process group
  - only session leader may allocate and deallocate a controlling terminal

4.17 So called hang-on-exit problem

```
$ ssh bash_user@hostname "sleep 10 &"
```

- the above hangs for 10 seconds on Solaris and Linux
- however, it does not on FreeBSD, OpenBSD, and NetBSD
  - in both cases though, the shell itself exits immediatelly
  - so, why the hang?
- SSH sleeps in `select()`, waiting on an event on the terminal
  - and on Solaris and Linux, it does not get the end-of-file when the shell exits. See notes for more info.

- bash's `huponexit` prevents the hang on Solaris and Linux but for the price of `SIGHUP` killing the process
- On BSD, `revoke()` is used to revoke all descriptors using the terminal, and SSH daemon thus gets 0 (end-of-file) on `read()`. And, `sleep(1)` still runs after the connection closes.
- On Solaris and Linux, that does not happen since there is no `revoke()`, so the `sleep(1)` command keeps the terminal open which is why the SSH daemon does not get end-of-file till `sleep(1)` exits.
- You should definitely try it yourself on those systems.
- See a discussion on OpenSSH mailing list. Look for "so-called-hang-on-exit" and especially for the subject "The complete answer (was Re: so-called-hang-on-exit)" by Nico Williams; it has a lengthy explanation of the problem:
  http://marc.info/?l=openssh-unix-dev&m=102878253003241&w=2
- On Solaris:

```
$ ssh localhost
$ sleep 99 &
[1] 4552
$ logout
<HANGS HERE UNTIL THE SLEEP CMD EXITS>
```

  However, if you use the `huponexit` option in `bash`:

```
$ ssh localhost
$ shopt -s huponexit
$ sleep 99 &
[1] 4552
$ logout
Connection to localhost closed.
```

4.18 Function calls for sessions and process groups

`int` **setsid**`(void);`

- creates a new session, with the current process as a session leader

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

- sets the process group ID `pgid` for process `pid`

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

- get the session or process group ID of process with PID `pid`
- The process calling `setsid` must not be a process group leader. `EPERM` is returned otherwise.
- New session has no controlling terminal. That's perfect for daemons since they do not need any.

- Normally, you can use `daemon()` call to daemonize your app which does the right thing. It might not be available on your system so the way how to do it is this:
  - fork()
  - the parent exits
  - the child calls setsid()
  - and redirects 0/1/2 file descriptors from/to `/dev/null` (ie., it opens `/dev/null`, `dup2()` it to 0/1/2, and closes it again).
- Source code example: `terminals/daemonize.c`.

4.19 Opening a Terminal

- On a classic terminal login each configured terminal is managed by `init`, followed by `getty` and `login`.
  - And `getty` is run with the terminal name as a parameter so it knows which terminal to open. It initializes the line, reads the user name and `exec()s login` with the username as a parameter.
- On networks logins and on terminals that are run through your windowing environment, pseudoterminals are used.
  - You ask for a new pseudoterminal then. More on that soon.
- By classic terminal login we mean sitting in front of a black screen with "Login:" only.
- By configured terminals we mean even virtual terminals used usually via Alt-Fxx keys from the text console.

4.20 Working with Pseudoterminals

- different on different systems
- however, POSIX defines functions that should be used
  - systems map their native calls to POSIX calls:

```
posix_openpt() - open a master pseudo terminal device
grantpt()      - changes the ownership of the corresponding
                 slave device to the calling UID
unlockpt()     - unlock the corresponding slave device so
                 that it can be read/written from/to
ptsname()      - gets the name of the corresponding slave
                 device
```

- `xterm` uses `posix_openpt()`, `grantpt()`, and `unlockpt()`.
- `xterm` forks, calls `pstname()` to get the slave device, and redirects 0/1/2 to the slave for the child.
- xterm's child `exec()s` the shell.
- For example, Linux gets the master pseudo terminal by calling `getpt(3)`, Solaris by opening `/dev/ptmx` etc. `posix_openpt()` is then implemented by those native function calls.

- On Solaris, the app needs to push `ptem` and `ldterm` modules onto the slave side to get terminal semantics.
  - pty's are implemented using STREAMS on Solaris
  - see the example code
- Source code example on how one process can communicate with another process over a newly allocated pseudo terminal: `terminals/pty.c`.

4.21 Setting Terminal Attributes

- various terminal attributes can be set
  - terminal speed, uppercase to lowercase conversion, remapping various control characters, echo switch off/on, etc.
- **tcgetattr**(), **tcsetattr**()
- canonical versus punctual mode
  - normally, input characters are queued until a line is complete, as indicated by pressing an Enter
  - in the punctual mode, there is no line queing.
- raw mode is punctual with many other attributes switched off, including ECHO
  - editors usually operate in raw mode
- The terminal attributes are set using the `termios` structure that is used as a parameter of a subsequent `tcsetattr()` call. The structure has input (eg. enable start/stop input control), output (eg. map CR to NL on output), control, and local modes (eg. enable echo on/off). Then it has an array of control characters (EOF, INTR, START, STOP, SUSPEND, ...) mapped to characters – usually ^D, ^C, ^Q, ^S, ^Z, respectively.
- So, if you want to be able to generate EOF with ^X instead of ^D, you would use "`stty eof ^X`". `stty(1)` would change `termios.c_cc[VEOF]` to `24` and then set the new structure via `tcsetattr()` with command `TCSANOW` (Change Attributes Now), for example. See `termios.h` in the UNIX spec for more information on all required terminal control characters.
- Raw mode with no ECHO is used when you type a password, for example, and want to print * for every character typed. You could not do that in the canonical mode. See the `ICANON` flag in `c_lflag` of the `termios` structure. Also, see `MIN` and `TIME` params. For more information, consult manual page for `termio(7I)`.
- There is no single attribute to set for the raw mode. See `-raw` option in `stty(1)` man page for the list of attributes switched off/on for that mode.
- Some old terminals could print only uppercase. To input/output an uppercase then, the letter is preceded by a backslash.
- Example on how to switch the echo off: `terminals/no-echo.c`. Note that it is independent on whether we use a physical terminal or a pseudo terminal. That means that we work with the terminal driver which is the slave part in case of pseudo terminals. That is the part that echos the characters read from the terminal.

- **Exercise:** you can modify the `no-echo.c` code so that you can read the password and print * for each character typed. It's fine to ignore all special characters like backspace, arrows, etc. You will have to switch off the canonical mode and use `c_cc` field of the `termio` structure to set those `MIN` and `TIME` parameters. Hint – when used as indexes, you prepend `V` to them. Solution: `terminals/type-password.c`. Read `termio` manual page for more information.

4.22 Terminfo database

- a database describing the capabilities of devices such as terminals and printers
- devices are described by specifying a set of capabilities and character sequences that trigger them
- screen oriented apps (eg., `vim`) and other commands (eg., `ls`) use **generic** capabilities without a need to know the exact terminal used
- terminfo source files are converted into a database by the `tic`(1) command
- terminfo source files are usually in `/usr/share/lib/terminfo`
- Terminfo capabilities:
  - Boolean capabilities
    - Whether the terminal has the feature or not.
    - Eg., can clear the screen.
  - Numeric capabilities:
    - Quantifying a particular feature of a device.
    - Eg., number of columns in a line
  - String capabilities:
    - Providing sequences used to perform particular operations on devices.
- terminfo is an System V equivalent of the original *termcap* system born in the BSD world. The termcap was invented by Bill Joy after he wrote the first version of the `vi` editor. After the ininitial version of the editor was written for a specific terminal, users started to want ports of the editor for different terminals. Instead of bending the editor for each available terminal, Bill Joy rewrote the editor with generic commands to manipulate the terminal, and introduced a *termcap* database containing capabilities of various terminals, and a library to query the database.
- termcap is a text based database, usually in `/etc/termcap`, see FreeBSD, for example. It is just one file. Terminfo is a compiled database, built from a separate files, one for each terminal. It might not come with the source files used to build the database. On Solaris, it is in `/usr/share/lib/terminfo`. It contains subdirectories named after letters to accomodate hundreds of terminals in an easy way.
- you can use `infocmp` to display the contents of the terminfo entry:

```
$ infocmp vt100
#       Reconstructed via infocmp from file:
#       /usr/share/lib/terminfo/v/vt100
```

```
vt100|vt100-am|dec vt100 (w/advanced video),
    am, mir, msgr, xenl, xon,
    cols#80, it#8, lines#24, vt#3,
    acsc=``aaffggjjkkllmmnnooppqqrrssttuuvvwwxxyyzz{{||}}~~,
    bel=^G, blink=\E[5m$<2>, bold=\E[1m$<2>,
...
...
```

- For example, `am` means "automargin", ie. when a line reaches the right edge of the screen, the terminal automatically continues on the next line. `cols#80` says that the terminal has 80 columns.
- Example: move the cursor to the specified position when the terminal is `xterm`. Solution is in `terminals/move-cursor-on-xterm.c`. You can set `PS1` to an empty string to see that the cursor will get to the absolute position then.
- **Exercise:** clear the screen and then move the cursor to the specified position on any supported terminal. Use low level terminfo routines to do that. Those routines are beyond the scope of this lecture, read apropriate documentation if you are interested. However, it is recommended to use *curses* which is a higher level interface. Anyway, the solution to experiment with is here: `terminals/move-cursor-with-terminfo.c`.

5 Advanced Network programming

beyond the "listen on a socket and accept requests"

5.1 Accessing low level structures

- get a list of interfaces on the system (ala ifconfig(1M)) (1)
  - OpenBSD: getifaddrs(3) - get list of IPs on the system
  - Solaris: ioctl() interface and `SIOCGLIFNUM`, `SIOCGLIFCONF` (2)

(1) e.g. to listen on each individual address in the system

- handy when the reply has to have the right source address (e.g. NTP protocol)
- need to track down the changes while running (see the `PF_ROUTE` socket slide)

(2) Generic approach of copying out sequence of structures from the kernel to userland:

1. open `AF_INET`/`SOCK_DGRAM` socket
2. fill the lifnum structure
3. issue the `SIOCGLIFNUM` ioctl to get number of interfaces
4. allocate buffer for number of entries acquired in previous step
5. fill the lifconf structure
6. issue the `SIOCGLIFCONF` ioctl to get interface structures
7. traverse the list of lifreq structures

- `lifreq` is defined in `usr/src/uts/common/net/if.h` and contains various interface specific data:

  - name
  - physical info
  - address info
  - MTU

- **Example**: `adv-net-prog/ifclist/`

  1. compile

     ```
     make clean && make
     ```

  2. run for both address families

     ```
     ./getlist 4
     ./getlist 6
     ```

  3. compare the output from the list of interfaces

     ```
     ifconfig -a
     ```

  Task: adjust `ifclist` to print out IPv4/IPv6 addresses on the interfaces

5.2 More setsockopt() options

- `SO_SNDBUF`/`SO_RCVBUF` - performance tuning (1)
- `SO_EXCLBIND` - exclusive binding (2)
- `SO_KEEPALIVE` (3)
- `SO_DONTROUTE` - append ifc name and send to the wire (4)
- `IPPROTO_IPV6` + `IPV6_V6ONLY`
- `IPPROTO_TCP` + `TCP_NODELAY`
- `IP_SEC_OPT` - use IPsec per-socket (per-port) policy, including policy bypass (5)
- some of the socket options are OS specific

(1) we can set snd,rcvbuf not only for TCP/UDP sockets but also for ICMP (raw ICMP socket)

(2) related to security (XXX find the SSH security bug with AF_INET6/AF_INET fallback)

(3) mention OpenSSH and the distinction of Keepalive messages on application layer

(4) gets full control over sending together with RAW sockets - see ping(1M) and -r option

(5)
  - **Example**: `adv-net-prog/setsockopt/policy-bypass.c`

    1. compile
    2. run the program w/out bypass

       ```
       ./bypass 0 20.0.0.1 80
       ```

    3. set policy

       ```
       echo "{ raddr 20.0.0.1 rport 80 } ipsec
         { encr_auth_alg sha1 encr_alg aes }" > /tmp/mypolicy.conf
       chmod 644 /tmp/mypolicy.conf
       ipsecconf -f -a /tmp/mypolicy.conf
       ```

4. run the program w/out bypass again
   ```
   ./bypass 0 20.0.0.1 80
   ```
5. run the program with bypass
   ```
   ./bypass 1 20.0.0.1 80
   ```

– **Example**: `adv-net-prog/setsockopt/auth.c`
  * similar to policy-bypass.c but provides traffic authentication
  1. setup IKE XXX
  2. initiate traffic

– References: http://blogs.oracle.com/danmcd/entry/put_ipsec_to_work_in

5.3 Raw sockets

- bypass packet encapsulation, i.e. push our own headers on the wire
  – this is for layer-3/network control
  – for layer-2/link access PF_PACKET interface is needed (1)
- `SOCK_RAW` is a type of socket (see socket(3socket))
  – can be used in various domains (will focus on PF_INET)
- need to know packet header format and protocol specifics (2)

(1) works on Linux, Solaris

(2) complete+correct IP header will be needed for most of the cases
  – this means we have to compute the checksum
  – we're pushing stuff to the wire, hence correct byte ordering is necessary (`htons()` etc, see byteorder(3socket))

(3) if we need application layer protocols we can let the stack to build IP header for us
  – ICMP
    ```
    socket(PF_INET, SOCK_RAW, IPPROTO_ICMP)
    ```
  – TCP
    ```
    socket(PF_INET, SOCK_RAW, IPPROTO_TCP)
    ```
  – UDP
    ```
    socket(PF_INET, SOCK_RAW, IPPROTO_UDP)
    ```

Example: construct ICMP packet header with arbitrary type and code `adv-net-prog/raw-socket`

1. compile
   ```
   make clean && make
   ```
2. construct ICMP echo request
   ```
   ./icmpraw 20.20.10.10 8 0
   ```

- libnet is a good C wrapper for raw socket manipulation
  – http://sourceforge.net/projects/libnet-dev/
  – http://www.securityfocus.com/infocus/1386

5.4 Layer-2 (link) access

- possible choices for link layer access:
  - Datalink Provider Interface (DLPI) - Solaris
  - `SOCK_PACKET` - Linux
  - Berkeley Packet Filter (BPF) - almost everywhere
- libpcap
  - high-level interface to packet capture
  - wrapper for all the 3 choices above - OS independent interface
  - pcap(3)
- Sequence of steps:
  1. find capturing device: pcap_lookupdev() (optional)
  2. get packet capture descriptor and open the device for capture: pcap_open_live() - alternatively open a file with stored traffic: pcap_open_offline()
  3. get network address and mask for the interface: pcap_lookupnet() - will be needed for compiling the filter
  4. compile the filter: pcap_compile()
  5. put the filter into effect: pcap_setfilter()
  6. read the data: pcap_loop()/pcap_next()
- **Example**: how to integrate pcap into an application `adv-net-prog/packet-capture` Capture and interpret TCP segements:
  1. compile

     ```
     make clean && make
     ```

  2. run

     ```
     ./tcptraffic host www.ms.mff.cuni.cz and port 80
     ```

  3. initiate some traffic

     ```
     echo "GET /" | nc -w 10 www.ms.mff.cuni.cz 80
     ```

  4. see the output
- Resources:
  - UNIX network programming, volume 1 (Richard W. Stevens), 29 Data link access

5.5 Porting applications to IPv6

- or alternatively - writing portable applications
- or better - address family agnostic applications
- AF-specific data structures (avoid if possible (1)):
  - `AF_INET` (IPv4): `sockaddr_in`, `in_addr_t`

  - `AF_INET6` (IPv6): `sockaddr_in6`, `in6_addr_t`

- AF-agnostic data structures:
  - `sockaddr`, `sockaddr_storage` (2)
- AF-specific functions (avoid):
  - `gethostbyname()`, `gethostbyaddr()`
- AF-agnostic functions:
  - `getaddrinfo()`, `getnameinfo()`

(1) Sometimes it is not possible to avoid AF-specifics

(2) define in `/usr/include/sys/socket_impl.h` (will be included with `<sys/socket.h>`)

Example: Watch out when listening on `AF_UNSPEC` sockets:

- CVE-2008-1483 in SunSSH (CR 6684003)
- http://src.opensolaris.org/source/history/onnv/onnv-gate/usr/src/cmd/ssh/libssh/common/channels.c
- `x11_create_display_inet()` does the setup for X11 forwarding over SSH channel
- it works like this: `adv-net-prog/ipv6-port-bind/cycles.c`

```
for port in (low, high); do
    list = resolve(port for all address faimilies)
    for family in list; do
        create socket for family
        bind the socket to the port and family
        if the bind fails
            if the list is non-empry
                continue
                  (try the same port number with different address family)
            else
                try next port number
            fi
        else
            record the port number
        fi
    done
done
```

- resolving will produce list with IPv6 family entry first and IPv4 entry second
- if someone binds the port to IPv6 *only* then `bind()` for IPv4 will succeed
  - the forwarder thinks everything is okay and passes the port number
- the app will try to connect to that port
  - some will try IPv6 first but that is hijacked by the attacker (who key steal keystrokes in case of X11 connection etc.)

5.6 PF_ROUTE socket

- exists on BSD, Linux, Solaris
  - PF_ROUTE exists since 4.3BSD-Reno
- usage: routing daemons, network server programs which need to observe changes in local addresses and logical interfaces (adding/removing server instances)
- Solaris man pages:
  - routing(7P)
  - route(7P) - describes `PF_ROUTE` interface
- working with the socket (passive mode):
  1. open new socket

     ```
     socket(PF_ROUTE, SOCK_RAW, 0);
     ```

  2. enter event loop watching for new messages: select/poll + read
  3. processing messages

Example: write something like 'route monitor' (see route(1M)) but only for watching interface additions/removals - useful for managing per-address server instances (e.g. UDP servers) `adv-net-prog/routing-socket/`

1. compile

   ```
   make clean && make
   ```

2. run

   ```
   ./rtwatch
   ```

3. create new interface (has to plumb IPv6 interface first, then add logical)

   ```
   ifconfig iprb0 inet6 plumb up
   ifconfig iprb0 inet6 addif 2001:0db8:3c4d:55:a00:20ff:fe8e:f3ad/64 up
   ```

4. observe the messages emitted by rtwatch

5.7 PF_KEY socket

- analogous to the PF_ROUTE socket
- available in Solaris, BSD
  - man page: pf_key(7P)
- manipulates IPsec Security Associations (SAs) (1) from userland
- app says ADD/DELETE/UPDATE SAs to the kernel
- `PF_KEY` interface described in RFC 2367
- there is also `PF_POLICY` socket on Solaris which manipulates security policy kernel entries

  - which decide whether packet will be allowed, denied or changed (2)

(1) SAs src/dst description, contain key material, timing information etc.

  - SA is unidirectional
  - they are used for computing authentication data and/or encrypt packets on the wire

(2) simple per-port (per-application) firewall

Example: flush IPsec AH SAs `adv-net-prog/key-socket/`

1. compile the program

```
make clean && make
```

2. Use Example `adv-net-prog/setsockopt/auth.c` to set IPsec SAs
3. observe SAs in the system and start watching

```
ipseckey dump
ipseckey monitor
```

4. flush the AH SAs using our program

```
./ahflush
```

5. check

```
ipseckey dump
```

- References: http://blogs.oracle.com/danmcd/entry/pf_key_in_solaris_or

6 Advanced Thread Programming

- Quick Recapitulation of Threads
- Other Existing Implementations of Threads
- `fork()` and Threads
- More on Thread Cancellation
- MT-Level Attributes
- Solaris Threads API
- Threads and Performance

6.1 Very Quick Recap of Threads

Multithreading is a programming and execution model that allows multiple threads to co-exist within the context of a single process. All threads share the process' resources but are able to execute independently. Threads provide developers with a useful abstraction of concurrent execution.

However, perhaps the most interesting application of the technology is when it is applied to a single process to enable parallel execution of threads on a multiprocessor system.

- Great introduction to threads including some of the history around this technology can be found in document "Multithreading in the Solaris Operating Environment".

6.2 Very Quick Recap of Threads (cont.)

- main differences between a thread and a process
- per thread attributes – instruction counter, stack, thread ID, ...
- when to use processes and when to use threads
- type of implementations of threads
  - user-level library (through `setjump/longjump` calls), inherently M:1
  - kernel (1:1)
  - hybrid (M:N)
- M:1 means that multiple threads are mapped to one process only. Thus, the kernel does not know about those threads at all, and in fact does not need to support any kernel threads. If one thread blocks, the whole process would block so a thread user-level library must replace blocking function calls with non-blocking ones.
- Hybrid M:N approach could be quite difficult to implement, you may be mapping, say, 23 threads used by your application to 5 kernel threads. The idea was to use as many threads as needed while not doing so many kernel switches between those. However, you need two schedulers, for example – one that schedules the kernel threads, and one that schedules the user threads. Solaris abandoned this approach, and NGPT (see page 51) was not chosen as the replacement LinuxThreads library, see below. It seems that much simpler 1:1 approach is generaly the winning one.
- The assumption that a kernel context switch must be inherently more expensive than a thread context switch in user level might be tricky. You must do a lot of things the same whether it is a switch in the kernel or in user space, and the user space switch typically involves several system calls anyway - you need to poll to see if the upcoming potentially blocking call is going to block or not, and if it is then you need to save the current state and use a long jump to switch to another thread. And there is more to it, see the reference below.
- Since FreeBSD 7, an alternative threading library there is `libkse(3)` which is M:N. The default threading library is still 1:1 `libthr(3)` though.
- References:

  - See [unix-prog] for more information on thread basics.
  - Very good discussion on M:N versus 1:1 mapping: http://xiao-feng.blogspot.com/2008/08/thread-mapping-11-vs-mn.html

6.3 What we know (from POSIX API)

- actions we already know how to perform
  - creating, destroying, joining, and terminating threads, and setting thread attributes when creating a thread
- synchronization primitives we know
  - mutexes, conditional variables, read-write locks
  - POSIX semaphores were not in [unix-prog] but it is just an analogy to SysV semaphores
  - barriers
- what else then do we need to know?
- [unix-prog] went through almost everything from [posix-threads].
- **Exercise:** to get your hands dirty with threads again, without use of any existing source code files write a simple program that creates two threads. Each thread writes a "Hello world"-like message on stdout to prove that the thread was really created, and then returns. The main thread waits on both threads to complete before it returns. Writing that should take 2 minutes and 20 lines of code :-).

6.4 Other Implementations

- Solaris Threads
- protothreads
  - low-overhead approach, non-preemptable threads with no stack
  - context is hold in global variables
- POSIX user-space library implementation in FreeBSD 3.0
- GNU Portable Threads (pth API)
  - highly portable user-space threading library (M:1) implementation
  - can emulate POSIX Thread API as well
- C Threads API (Mach OS)
- and many, many more...
- "Non-preemtable" in protothreads means that a context switch can take place on blocking operations only (`read()`, for example). No stack means no local variables. Protothreads are targeted at severely memory constrained systems, such as small embedded systems or wireless sensor network nodes but its use it not limited to just that.

- LinuxThreads was a partial implementation of POSIX Threads that has since been superseded by the Native POSIX Thread Library (NPTL). LinuxThreads had many problems owed directly to its implementation; each thread had its own PID, for example, which is a violation of the POSIX standard. NPTL is fully incorporated in the GNU C library now.
- Linux version 2.4 had no real kernel thread support.
- GNU Pth and NPTL are two distinct implementations. NPTL makes use of the Linux kernel threads. There was also NGPT M:N aproach intended to replace LinuxThreads but eventually NPTL was chosen (BTW, it is 1:1). Read more on history of Linux threads on the Onlamp link below.
- User-space implementation of threads, or "user threads", not relying on the system thread support and able to run on systems without such a support at all, is sometimes called "green threads". This seems to have come from the Java world:

  > *When Java 1.0 first came out on Solaris, it did not use the native Solaris library libthread.so to support threads. Instead it used runtime thread support that had been written in Java for an earlier project code-named "Green." That threading library came to be known as "green threads."*

- References:
  - http://en.wikipedia.org/wiki/Thread_(computer_science)
  - http://en.wikipedia.org/wiki/LinuxThreads
  - http://en.wikipedia.org/wiki/Native_POSIX_Thread_Library
  - http://onlamp.com/onlamp/2002/11/07/linux_threads.html
  - protothread implementation: http://www.sics.se/~adam/pt

6.5 Fork and Threads in POSIX

- discussed to some extent already in [unix-prog]
- `fork` duplicates only the calling thread in the child
  - was not true on Solaris 9 or earlier with Solaris Threads
  - it used the "fork all" semantics (`forkall(2)`), as opposed to the "fork one"
- fork-one safety issues
  - when `fork` is used, threads must be well behaved
  - imagine a thread that does not call `fork` but holds a mutex
    - the thread will not be duplicated, leaving the mutex locked for ever
    - the child will probably dead-lock sooner or later
- Remember that a locked mutex may be unlocked only by the thread that locked it before. That's the difference between a mutex and a binary semaphore.
- The class of problems that can happen on fork are in general called "fork safety issues".

6.6 Memory Leaks on `fork(2)`

- If a pointer to some memory dynamically allocated in each thread is on the thread's stack only, such memory will be leaked if the thread is not the one calling `fork(2)`. It happens in such a way because the thread, including its stack, is not duplicated in the new process.

6.7 Fork handlers - solution to fork-one safety issues

```
int pthread_atfork(void (*prepare)(void),
                   void (*parent)(void),
                   void (*child)(void));
```

- `prepare` is called before `fork` is started
- `parent`/`child` are called in parent/child after `fork`
- `prepare` acquires all mutexes making sure no thread can hold any lock
- `parent`/`child` functions will release all locks after the `fork` then
- Source: `adv-thread-prog/fork-one-issue.c` shows memory leaks in the child, `adv-thread-prog/at-fork.c` shows how to use the handlers in general.
- You may use fork handlers with a process with the `main()` thread only, of course, which make them quite a general tool to use.
- It is not just about mutexes and memory leaks. Imagine a library that has some kernel state (ie. a handler to some kernel table slot):
  - A `fork` will duplicate the user-level data.
  - Kernel state will stay the same but now we have 2 processes with a handler to the same slot in a kernel table (which holds a reference counter equal to 1, not 2).
  - When one process releases the handler the other thread one can no longer work with it since the reference counter dropped to 0, leading to the slot deallocation.
  - For example, PKCS#11 fork safety issues stem from the fact that according to the spec, the child should never use crypto sessions initialized in the parent.
    - At-fork handlers can help here as well. The sessions are closed and initialized again (= the child will get its own kernel state structures) in the atfork functions.
- `pthread_atfork()` is not a necessary solution for `fork-one-issue.c`. Given you cannot give any parameters to those functions, global memory would have to be used to store all the pointers anyway so that we could free those in the child. That means that (a) mdb would show no memory leaks and (b) we could free all the pointers in the child after the fork. However, we would have to do that manually which might not be possible – we could fork in a library function without knowing about it, for example.
- In a threaded application, you usually need to fork only to run an external command, either explicitly or implicitly through a library call. If you need to do something else you create a new thread, not a new process, so those problems with mutexes might seem quite unreal – the external command can not use those mutexes at all. However, the example above, about PKCS#11, shows a real example of a library that uses PKCS#11 API, and an application

that knows nothing about it while forking a child. That's how SunSSH uses the OpenSSL PKCS#11 engine to offload crypto operations to the hardware accelerator. You must be careful not to create a dead-lock in the library then.

- Source: another example showing what happens if we hold a mutex in a thread that has not called `fork`, `adv-thread-prog/mutex-with-atfork.c`.
- **Exercise:** fix `adv-thread-prog/fork-one-issue.c` with the fork handlers (ie. do not free the allocated memory in the child manually). Of course that you will have to use global memory to store the memory pointers anyway. Also remember that you should not free memory allocated in the thread that called `fork(2)` – it will probably need it.

6.8 Fork-Safe Library

Library can take care of the fork issues itself. On Solaris, `attributes(5)` defines a *Fork-Safe* library as the one that:

> When `fork()` is called, a Fork-Safe library arranges to have all of its internal locks held only by the thread performing the fork. This is usually accomplished with `pthread_atfork(3c)`, which is called when the library is initialized.

...which is exactly what was discussed on the previous slide.

- "Fork-Safe" is actually one of the categories of the "MT-Level" attribute. More information is also on page 55. See `attributes(5)` for more information on attributes in general.

6.9 More on thread cancellation

- cancellation is good for situations where, for example:
  - user is requesting to close or exit some running operation
  - number of threads are solving a problem, one thread finds the solution, and all the remaining threads can be cancelled
- you must be careful
  - calling a cancel-unsafe library might cause a problem in certain situations
    - the thread might be cancelled while in a library call which might cause dead locks, memory leaks, etc.
  - `pthread_setcancelstate()` can be called to temporarily disable cancellation
- *cancellation point* is a place at which cancellation can occur
- Source code file `adv-thread-prog/pthread-cancel.c` borrowed from [unix-prog] recaps how it works.
- The specification names the list of calls that must function as cancellation calls with another list of calls that may be cancellation points. See the link below. The actual set is then defined by the system and not suprisingly, each system documents it somewhere else. Solaris specifies the list in the `cancellation(5)` man page, FreeBSD uses `pthread_cancel(3)`, Linux distros `pthreads(7)`.
- References:
  - Cancellation points are defined in the "Thread Cancellation" section of http://opengroup.org/onlinepubs/007908775/xsh/threads.html

6.10 More on thread cancellation (cont.)

- use `pthread_testcancel()` to insert your own cancellation points
- any call that might wait long should be a cancellation point
- be careful with asynchronous cancellation
- `pthread_cleanup_push()` should be used when a thread changes some state and there is a possibility of cancellation
  - the handler makes sure the state is reverted should there be any cancellation
  - do not forget to pop the handler when the previously changed state has been restored
- Asynchronous cancellation:
  - Locked mutex in a cancelled thread can deadlock your application or cause memory leaks.
  - In general, **the problem is to cancel a thread that holds some resources**.
- Cancel-Safe library pushes handlers wherever cancellation can occur, and pops them when the state is restored. See `attributes(5)` man page on Solaris.
- **Exercise:** demonstrate a problem with a non-cancel-safe library. Write a library with one function which internally uses dynamically allocated memory. The memory is deallocated before the call returns. Use it the way that `mdb` will report some memory leaks (sleep in the library before freeing previously allocated memory and then cancel the thread from `main()`). Then fix with `pthread_cleanup_push()` and a handler that deallocates the memory should the thread be cancelled. Remember that shared library is created using `--shared` option with `gcc` or `-G` with Sun Studio (`cc`). Use `-R` and `-L` options properly.

6.11 Cancel-Safety in Libraries

On Solaris, `attributes(5)` defines a *Cancel-**Unsafe*** library as the one that:

> If the thread has not installed the appropriate cancellation cleanup handlers to release the resources appropriately (see `pthread_cancel(3c)`), the application is "cancel-unsafe", that is, it is not safe with respect to cancellation.

...and:

> All applications that use `pthread_cancel(3c)` should ensure that they operate in a Cancel-Safe environment.

There are two subcategories for libraries wrt cancel safety: "Asynchronous-Cancel-Safety" and "Deferred-Cancel-Safe".

- Obviously, Deferred-Cancel-Safety is easier to achieve than Asynchronous-Cancel-Safety. **Most applications and libraries are expected to always be Asynchronous-Cancel-Unsafe, unless explicitly specified otherwise.**

6.12 MT-Level Attribute in General

In Solaris, there are more categories for the MT-Level attribute assigned to libraries:

- *Safe* – can be used from multithreaded apps
- *Unsafe* – library contains unprotected global and static data. Make sure only 1 thread uses the library at a time
- *MT-Safe* – fully prepared for multithreaded apps and should provide reasonable concurrency
- and some more

This is per function:

- *Async-Signal-Safe* – the function is MT-Safe and it can be safely used from a signal handler
- Using a "Safe" library means that you will not dead-lock, crash or corrupt its internal data if called from multiple threads. We can make an Unsafe library a Safe one by surrounding an entire library with a mutex but such aproach does not provide any concurrency. Thus, the library can be called Safe but not MT-Safe.
- Remember, using a function in a signal handler that is not ready for that might result in a dead-lock. Imagine that another signal comes when the handler is being already processed – if the handler is used for that signal as well, we can dead-lock if async-unsafe function is used. Note that
- In Solaris, every manual page for a function call has an `ATTRIBUTES` section that also states a value of the MT-Level attribute. Example from the `read(2)` manual page:

```
ATTRIBUTES
     See attributes(5) for descriptions of the following attri-
     butes:
     ____________________________________________________________
    |       ATTRIBUTE TYPE      |       ATTRIBUTE VALUE           |
    |___________________________|__________________________________|
    | Interface Stability       | Committed                        |
    |___________________________|__________________________________|
    | MT-Level                  | read() is Async-Signal-Safe      |
    |___________________________|__________________________________|
    | Standard                  | See standards(5).                |
    |___________________________|__________________________________|
```

6.13 Solaris Threads

- in `libthread(3lib)` library shipped since Solaris 2.2 (1993)
  - POSIX thread API not available at that time
  - support for POSIX threads (pthreads) added in 2.5 (1995)
- Solaris API was also used in UNIX International spec
- there are differences between both APIs
  - but not major wrt functionality provided

- you can combine both APIs in one program
  - note that the same kernel threads are underneath, API is just a way to work with those kernel threads
- Basic information on the Solaris Threads API is in the `libthread(3lib)` manual page on Solaris.
- Source: `adv-thread-prog/posix-with-yield.c`. Read the opening comment for instructions on how to use the program.
- Differences are listed in the `threads(5)` manual page. Look for `THR_DAEMON`, for example, that sounds interesting.
- Both threading libraries were merged into `libc` since Solaris 10, see the source code example.
- References:
  - UNIX International: http://en.wikipedia.org/wiki/Unix_International
  - Multithreading in the Solaris Operating Environment: See "Reliable, Scalable Threads For All" section for more information on the `libthread` library.

6.14 Solaris Threads API

- no attribute objects as in POSIX threads – you must use `flags`
- use `<thread.h>` header file

```
int thr_create(void *stack_base, size_t stack_size,
               void *(*start_func) (void *), void *arg,
               long flags, thread_t *new_thread_ID);
```

- `start_func` is the thread function
- `arg` is the argument the function will be called with
- you can use flags: `THR_DETACHED`, `THR_SUSPENDED` (the thread is created suspended), `THR_DAEMON` (the thread will continue to operate after `main()` returned)
- As with `pthread_create()`, the new thread inherits the signal mask from the creating thread.
- Default thread creation (`NULL` for the stack base and `0` for the stack size makes the system use the default values) is like this:

```
thread_t tid;
void *start_func(void *), *arg;

thr_create(NULL, 0, start_func, arg, 0, &tid);
```

  It creates a joinable thread. The following piece of code creates a detached thread:

```
thr_create(NULL, 0, start_func, arg, THR_DETACHED, NULL);
```

- **Exercise:** remember the simple recap program on threads on page 50? Rewrite it using Solaris Threads API. Note that joining the thread is accomplished via `thr_join()`, suprisingly.

6.15 Differences between POSIX and Solaris Threads

- POSIX threads can be cancelled
  - this is a new thing introduced in POSIX threads
- Solaris threads can be suspended and resumed
  - POSIX also does not have a yield (`thr_yield()`)
- in Solaris threads, we can wait for any thread
- Solaris threads have no clean-up handlers for `fork(2)`
- Solaris threads offer daemon threads
- use POSIX threads on Solaris for new applications
  - because that is the portable way of doing things
- Waiting for any thread was intentionally not included in the POSIX standard.
  - There is no parent-parent relationship between threads.
  - All threads aside from main are equal.
  - So, there is no concept of "waiting for a child thread" as is in the process environment.
- In Solaris threads, waiting for any thread is possible if 0 is used as the thread ID:

```
if (thr_join(0, NULL, NULL) == 0) {
        ...
}
```

- In pthreads, you could easily simulate that using one thread (the main one, perhaps) for waiting on a conditional variable while each finishing thread would signal the variable right before returning from its function. Main could then join the thread, getting its id from a protected global variable, for example.

6.16 Combining both APIs

- works because those APIs "just" work with the same kernel entity – the thread
- probably does not have much sense
- maybe "fixing" old code with mechanisms provided in other thread API
  - anyway, note that thread types are different - we cannot join POSIX thread with Solaris thread API call
- Source: while we alread showed on page 56 that we can yield threads created with POSIX API, the following program even creates threads using both APIs: `adv-thread-prog/both-thread-APIs.c`.

6.17 Threads and Performance

- parallelism is one of the answers for a need to get higher performance
- great for threads
- however, threads bring inherent data sharing between them
- need to synchronize
- synchronization points, a shared structure for example, are candidates for bottlenecks in the program performance
- also, not everything is possible to parallelize – CBC (cipher block chaining) is one example
- On the wake of more and more multi-core CPUs emerging **it is always important to realize what we can expect there**. In general, if you do not know what to expect, you can not estimate various attributes of the project. If you can not estimate, you can not manage the project.
- With multiple cores, we can roughly expect linear increase in computation power, depending on the system, program, and the type of a problem being solved, of course. What's more, some cores are able to run more threads in parallel, without a need for a software context switch. Thus, such threads can represent something called *virtual CPUs*. This idea in general is called *Chip Multi-Processing*.
- One example of a CMP machine is UltraSPARC T1 or T2. T2 can have up to 8 cores with each core running up to 8 threads. T2 with 8 cores then have 64 virtual CPUs. However, how does that scale then? Take `u-us` in Malá Strana's lab. It's an UltraSPARC T1 machine with 6 cores (T1 can run up to 4 threads per core). So, `/usr/sbin/psrinfo` will show you 24 virtual CPUs. With a simple program that runs just looping threads, we can see that we can get 70% of the theoretical maximum and we even scale linearly up to 6 threads:

```
# We get 600% of a single thread performance if running with 6
# threads:
/a.out 6 10
1661374469
$ ./a.out 1 10
277845608
$ bc
6 * 277845608
1667073648

# With 24 threads we can see that virtual CPUs are not fully
# independent and that we get "only" 70% of the theoretical
# maximum:
$ ./a.out -g 24 10
Using global memory for counters.
4679205463
./a.out 1 10
277776481
$ bc
24 * 277776481
```

```
6666635544
scale=2
4679205463/6666635544
.70
```

- Source: `adv-thread-prog/parallel-computation.c`.

7 Advanced IPC and I/O

7.1 Advanced IPC and I/O

- (Slightly) More on the `ioctl(2)` System Call
- Doors Interface as a Lightweight RPC
- Passing File Descriptors Between Processes
- POSIX Real-time Signals
- Asynchronous POSIX I/O
- Existing Means of Creating a New Process, and new ones:
  - `popen()` with `pclose()`
  - `posix_spawn()`

7.2 Known Means of IPC

- oldest (pre-1980) UNIX systems
  - signals
  - process tracing
  - files and shared file offests
  - pipes
- then:
  - named pipes (FIFOs)
  - file locks
  - sockets
  - (System V IPC) semaphors, messages, shared memory
  - POSIX version of semaphores, messages, and shared memory
- ,,Known" means known to you from [unix-prog-I]. Those are **14 different mechanisms** of how processes can communicate with each other.
- And there are more:
  - doors
  - passing file descriptors between processes
  - and probably something else...

- Process tracing is done today using `/procfs`, which might be considered an IPC over files. However, there was, and sometimes still is, `ptrace()` call, usually a system call, that allowed one process to debug another one. Debugging means setting and clearing breakpoints, and reading and writing the other process's address space.
- [unix-prog-I] introduced System V IPC semaphores only, and just mentioned messages and shared memory, including its POSIX equivalents. However, it's recommended to use the POSIX calls. In general, mostly you do not need messages and shared memory since you can use pipes or sockets for data passing, and `mmap(2)` for memory sharing (which with `MAP_ANON` does not even need a file descriptor at all).

7.3 ioctl(2)

- a catchall for I/O operations
  - when there is no specific funtion for particular I/O, it usually ends up in `ioctl()`
- terminal I/O is (was) the biggest user of this function
  - however, POSIX introduced separate functions for terminal handling
- easy access to kernel through device drivers
- UNIX spec includes `ioctl()` only as an extension for dealing with STREAMS devices, but it is used heavily elsewhere as well
- and **why** is `ioctl()` so convenient? Because you give it a command (`int`) and then a variable number of other parameters.
- `int ioctl(int fd, int request, /* arg */ ...);`
- The command (request) is interpreted by the driver so if you write your own driver, you interpret the commands as you wish, and most probably you define your own macros for your commands, and put those macros in a public header file for the driver.
- STREAMS is a mechanism on some UNIX implementations that allow character device drivers to be implemented in a modular fashion. Some systems extended this idea. For example, pipes are implemented on top of STREAMS on Solaris. Anyway, users usually do not have to care about that. See page 63 for an example.
- We just cannot invent another system calls as we see fit so we use `ioctl()` for that.

7.4 Example: `/dev/crypto` on Solaris

- Solaris Cryptographics Framework provides crypto services to users and applications
- it has a user level and a kernel level part. You always link to `libpkcs11.so` library.
- `libpkcs11.so` can utilize HW crypto through the `pkcs11_kernel.so` library
- your application → `libpkcs11.so` → `pkcs11_kernel.so` → `ioctl()` on `/dev/crypto`
  - ie. through `/dev/crypto` pseudo driver we get to the kernel from the user space
- The kernel part of the Crypto Framework takes care of HW crypto cards, for example.

- In the user level part, you can utilize the software crypto providers working completely in user level. That means that all the implementation is provided by a software library (aka „software provider"). However, what we are interested here is the kernel part of the Crypto Framework.
- `CRYPTO_DEVICE` is defined as `/dev/crypto`:
  http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/lib/pkcs11/pkcs11_kernel/common/kernelGlobal.h
- It is opened here:
  http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/lib/pkcs11/pkcs11_kernel/common/kernelGeneral.c
- And it is used through `ioctl()` here, for example, to perform an encryption:
  http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/lib/pkcs11/pkcs11_kernel/common/kernelEncrypt.c
- Another example from Solaris – you know `poll()` and `select()` calls. While `poll()` is preferred (you should know why), it still has issues. For example, polling thousands of file descriptors will little activity on them means that we must pass all those structures to kernel for every call. It is better to use `/dev/poll` pseudo device which provides what `poll()` can do for you, and more. It keeps all the information about polled descriptors in kernel memory, for example. And you use `ioctl()` to manipulate `/dev/poll`, of course. You can read the following documents to get more information:
  http://developers.sun.com/solaris/articles/polling_efficient.html
  http://developers.sun.com/solaris/articles/using_devpoll.html

7.5 Doors Overview

- fast light-weight RPC mechanism for secure control transfer between processes on the same machine
- developed as part of Sun's experimental microkernel-based Spring operating system
- a thread in one process can issue a call using a door descriptor that causes code to be executed in another process
  - the doors server exports functions to be called through the door created via `door_create(3c)`
- In Solaris since version 2.5 as a private interface, changed to a public one and documented since 2.6.
- Spring was an experimental microkernel-based object oriented operating system developed at Sun Microsystems in the early 1990s. Development ceded in the mid-1990s but several ideas and some code from the project was later re-used in the Java programming language libraries and the Solaris operating system.
- In Solaris used in the name service cache daemon, `nscd(1M)`, for example. The reason why name service resolution is not fully done in libraries is that the daemon implements a system wide caching so that resolution of a name by one process can be later used by another one. That is not possible with libraries. It would be technically possible, for example, via a shared database on the disk and updated as needed by all but that could not be used in a production system due to inherent security problems.

- Doors implementation was in a separate `libdoor(3lib)` library, later merged to `libc`.
- Used as a client-server model within the same machine.
- Server itself does not have to use threads at all but the system will use it to implement the doors functionality.
  - that means that the server can be single threaded but the way it works is effectively multithreaded.
- Kernel can communicate back to userland through doors.
  - in which case the door server is a user process, the client is the kernel.

7.6 Doors - how it works

- server calls `door_create()` with the server function that will be called
- the call takes care of thread creation since all door calls are handled by threads
- **the server can sleep or do whatever it wants**
- the client calls `door_call()` to access the functionality
- a filename is used as the door identification, similarly to System V IPC
- use `fattach()` to attach the door to the id file
- There is also doors implementation for Linux, as a patch.
- See [Solaris-internals], section 4.8 for more information.
- Doors are used for various system stuff on Solaris:

```
$ find /var/run/ -name '*door'
/var/run/syseventconfd_door
/var/run/syseventconfd_door/reg_door
/var/run/name_service_door
/var/run/sysevent_channels/syseventd_channel/reg_door
/var/run/sysevent_door
/var/run/rcm_daemon_door
/var/run/picld_door
/var/run/rpc_door
/var/run/syslog_door
```

- `adv-ipc-and-io/door-server.c`
- `adv-ipc-and-io/door-client.c`
- **Exercise:** modify the code so that you can give the client 2 numbers and the server will return their sum, ie. compute `a + b` using `a`, `b`.
- There is more what you can do about doors. You can manage the thread pools on the server side via `door_bind()`, or get the door info from the client side via `door_info()`. See respective man pages if you are interested.
- Doors can also serve as a poor man's paralelization technique. You do not use threads directly in the parent but they are created and run in parallel for you.

7.7 Sending file descriptors between processes

- sometimes it is extremely useful to let another process open a file and send the descriptor to another process
  - for example, the 1st process has enough privileges to open the file but the 2nd one does not
  - `chroot()`ed process has no access to external resources (`/dev`) but may need a PTY. Used in OpenSSH.
- mechanisms for file descriptors passing
  - via STREAMS (→ pipes). On Solaris, where pipes are implemented via STREAMS.
  - via UNIX domain socket messages.
- According to the documentation, it should able to use `door_call()` to pass a file descriptor. I have not tried that. If you do, send us your example source code and we will update the `src` repository and this text.

7.8 Passing fd over a pipe (Solaris)

- pipes are built on top of STREAMS
  - and file descriptors can be sent over STREAMS
  - so, use `I_SENDFD` and `I_RECVFD` `ioctl()` command on the pipe descriptor
- code used:
  - sender: `ioctl(p[0], I_SENDFD, fd)`
  - receiver: `ioctl(p[0], I_RECVFD, &getfd_struct)`
    and then you use `getfd_struct.fd` to get the file descriptor
- Note that we get UID/GID of the sending process in `getfd_struct` as well, see the example code.
- `adv-ipc-and-io/fd-over-pipe.c`
- **Exercise:** is file descriptor passing out-of-band communication? Can we pass "normal" data over the pipe when the peer expects an `I_RECVFD` message? Try it out.

7.9 Passing file descriptor over a socket

- passing a file descriptor over a pipe is nice and easy
  - not supported on many systems though
- need a more portable way
- one can pass a file descriptor over a UNIX domain socket, using a special message with `sendmsg()` call
  - systems still do that differently but usually it is possible to use UNIX domain sockets for file descriptor passing
  - you will not get UID/GID as when used with pipes

- You can see that it is much more complicated than using a pipe. However, it is more portable.
- `adv-ipc-and-io/fd-over-socket.c`
- **Exercise:** port the source code to Linux.

7.10 Real-time signals - motivation

There are some serious problems with POSIX.1 signals:

- only 2 signals `SIGUSER(1|2)` left for application use
- no signal queuing
- no delivery order
- poor information content
- asynchronous delivery only
- low speed

POSIX.4 extension addresses those problems. Some are not solved completely – the problem of a low speed, for example.

- POSIX.4 signal extension was already mentioned in [unix-prog-I] lecture, and a source code example was provided.

7.11 Using real-time signals

- check `_POSIX_REALTIME_SIGNALS` macro in `unistd.h` during a compile time
- use `SA_SIGINFO` flag in `sigaction` structure to set the extension
- use `sa_sigaction`, not `sa_handler` for the handler:

```
void handler(int signum, siginfo_t *data, void *extra)
```

- ignore `extra`, it is there for compatibility reasons
- use `siginfo_t data` to get more information
- check `siginfo_t`'s union `sigval` for additional information (integer or pointer) if you expect it, and its `si_code` can give you more info on why the signal was generated.
- POSIX.4 signal extension defined a new set of signals and changed semantics for them. The new signals are numbered `SIGRTMIN` through `SIGRTMAX`. Those may not be macros but may change. There are at least `RTSIG_MAX` such signals, the macro is defined in `limits.h`. Use `SIGRTMIN + n` to specify signals.
- The `siginfo_t` structure contains at least the following members (plus some more also required by the POSIX signal extension):

```
typedef struct {
        ...
        int     si_signo;
        int     si_code;
        union   sigval si_value;
        pid_t   si_pid;
        uid_t   si_uid;
    ...
```

- Real-time signals are queued and delivered in order. The real-time signal extension says nothing about existing POSIX.1 signals. The "ordered delivery" means that lower-numbered signals are delivered before higher-numbered ones. Again, this says nothing about existing POSIX.1 signals.

7.12 Using real-time signals (continuted)

- a new function for sending a signal is needed
- because we can send the `sigval` union together with the signal:

```
int sigqueue(pid_t pid, int signo,
             const union sigval value);
```

- asynchrony can be supressed (ie., no handlers are called) using the `sigwaitinfo` function:

```
int sigwaitinfo(const sigset_t *restrict set,
                siginfo_t *restrict info);
```

- The `sigwaitinfo` function, **if signals are blocked**, will not cause the handler to be invoked. So, you can block waiting on the signal, and increase the speed of delivery since no handler is called. If signals are not blocked, the old-style handler delivery takes precedence.
- You should remember from [unix-prog-I] that we also have the `sigwait` function. That function was defined in the POSIX thread extension and while it works in a similar way there are some differencies – the `errno` value is returned directly by `sigwait`.
- Real-time signals are not generated by a user only. They can be generated as a result of a POSIX.4 timer or a completion of asynchronous I/O (and with messages but that's outside of the scope of this lecture). In that case, you give the system a `sigevent` structure beforehand so that the system knows what information to pass along with the signal later. We will see that with asynchronous POSIX I/O that begins on page 67.

7.13 Asynchronous versus Synchronous Programming

Many slightly different uses of those two adjectives in the real life.

- in programming, **asynchronous** events are those occurring independently of the main program flow, **allowing the main program flow to continue processing**
- a **synchronous** event takes place **while you wait**

POSIX defines these two terms as follows:

- a **synchronous I/O operation** causes the requesting process to be blocked until that I/O operation completes
- an **asynchronous I/O operation** does not cause the requesting process to be blocked
- In asynchronnous programming, you **do not block** waiting for completion.
- `select()` and `poll()` are still part of synchronous programming. While you can work with multiple descriptors, you block until one is ready, and then you wait until that action performed on it is complete, and you know that you are not going to be put to a sleep. I saw it also being called an ,,asynchronous blocking" scheme. That is not correct according to the POSIX definition. Nothing is happening until you initiate the `read()`, `write()`, `open()`, or another operation.
- Note that POSIX does not define how `O_NONBLOCK` works with regular files. It can have no effect on them. Using the flag on regular files would mean that if the data is to be get from the disk, that would mean blocking the process, if the data is in system's memory it would need no blocking.

7.14 Asynchronous I/O (AIO) – motivation

- **initiate an operation and do something else until it completes**
- with threads, you can reach the same objective – some threads are blocked and waiting but other threads use CPU cycles
- you may end up using a significant number of threads
  - which means a significant number of context switches
- threads means synchronization
  - not a trivial problem as such
  - synchonization (eg., mutexes) is not for free
- relatively small number of threads with asynchronous I/O can serve a several order of magnitudes more requests, clients, etc.
- Using threads assumes a different type of programming model. With AIO, you can work in a single thread and still achive notion of paralelism. It's similar to what you can do in a single thread with `select()`. However, you still block waiting for operations to finish in the latter case.
- In general, the need for AIO usually arises because an application has severe timing constraints.
- Also, the AIO framework was designed to support AIO transactions in a per-process manner, and thus it does not scale well for highly multithreaded applications – if you have hundreds of threads, sending a signal as a notification type definitely does not scale, remember how to handle signals with threads from [unix-prog-I]. See Solaris Event Ports for more information, p. 71.
- References
  - Thread Pools Using Solaris 8 Asynchronous I/O http://developers.sun.com/solaris/articles/thread_pools.html

7.15 Asynchronous POSIX I/O

- POSIX 1003.1b-1993
- one of those 9 optional parts of the 1b standard
- instead of blocking for completion, the system just queues I/O and the function returns right away
- on I/O completion a signal can be delivered (if you wish)
- normal `open()` and `close()` is used
- asynchronous I/O operations are submitted using an `aiocb` structure
  - `aiocb` = Asynchronous I/O Control Block
- Defined in POSIX.4, formally known as *IEEE Std 1003.1b-1993 Realtime Extension*, see [unix-prog-I] for more information on POSIX in general.
- `_POSIX_VERSION` must be greater or equal 199309L, and `_POSIX_ASYNCHRONOUS_IO` must be defined (remember from [unix-prog-I], POSIX.4 is just one small mandatory extension to signals in comparison to POSIX.1 from 1990 plus many optional extensions, including the asynchronous I/O). Saying ,,the system conforms to the POSIX.1b standard" without saying what optional parts are supported is very misleading. Do not forget to include `unistd.h` before checking those macros.
- References
  - GNU libc manual, section ,,13.10 Perform I/O Operations in Parallel" http://www.gnu.org/software/libc/manual/html_node/Asynchronous-I_002fO.html#Asynchronous-I_002fO

7.16 The `aiocb` structure

```
typedef struct aiocb {
    int              aio_fildes;     /* file descriptor */
    off_t            aio_offset;     /* file offset */
    volatile void    *aio_buf;       /* buffer location */
    size_t           aio_nbytes;     /* length of transfer */
    int              aio_reqprio;    /* request priority offset */
    struct sigevent  aio_sigevent;   /* notification type */
    int              aio_lio_opcode; /* listio operation */
} aiocb_t;
```

- Include `aio.h` header file.
- Using `aio_offset` is mandatory unless `O_APPEND` was used in `open()`. What's more, the file position is in an unspecified state after the operation so when performing normal `read()` or `write()` after that, you **must** call `lseek()` before that.
- `aio_sigevent.sigev_notify` can be one of `SIGEV_NONE`, `SIGEV_SIGNAL`, `SIGEV_THREAD`, or `SIGEV_PORT`, meaning that nothing is done when the operation finishes, a queued signal is sent, a thread is created, or event port notification is used, respectively.
- `aio_lio_opcode` is used only with `lio_listio()`.

7.17 Using POSIX Asynchronous I/O

```
int     aio_read(struct aiocb *aiocbp);
int     aio_write(struct aiocb *aiocbp);
int     aio_suspend(const struct aiocb *lacb[],
                    int num_acbs,
                    const struct timespec *timeout);
int     aio_cancel(int fd, struct aiocb *acbp);
int     lio_listio(int wait_or_not,
                   struct aiocb *cont lacb[], int num_acbs,
                   struct sigevent *notification);
ssize_t aio_return(const struct aiocb *acbp);
int     aio_error(const struct aiocb *acbp);
```

and some more...

- A very simple example. Let's read first 512 bytes of the file, and handle the completion in a signal handler. You should use the extended signal so that you get the address of a buffer in the handler's argument. Remember, if `sa_flags` is set to `SA_SIGINFO` in the `sigaction` structure, the handler is like this:

  ```
  void term_handler(int sig, siginfo_t *info, void *ignored);
  ```

  and in this situation with `SIGEV_SIGNAL` set in `aio_sigevent`, the handler is called like this:

  ```
  struct aiocb a;
  siginfo_t info;

  /* ... */

  signo = SIGXXX;
  info.si_signo = SIGXXX;
  info.si_value.sival_ptr = (void *)&a;
  term_handler(signo, &info, ignored);
  ```

  The source code for this: `adv-ipc-and-io/simple-aio.c`.

- **Exercise:** modify the code so that in one thread you call `aio_read()` to read the whole file and call `aio_suspend()` in another thread and print out data in the right order, simulating `cat(1)`. Do not forget to allocate `aiocb` structure dynamically or use other method to make sure you never reuse the structure before the operation is complete. Get the file size before you start calling reading the file so that you know how many `aio_read()` calls you are going to need.

- **Exercise 2:** use `lio_listio()` instead of `aio_read()`.

7.18 `popen()` and `pclose()`

- creates a pipe between the calling process and the command
- returns FILE*

- As you can see from the picture, `popen()` can be used both to read from and write data to the command. However, reading is probably the most common case. And while it explicitly says ,,`fork`", it does not have to be like that, of course. The implementation is system depended because the UNIX specification does not care about the actual implementation.
- Creating a pipe, forking a new process, closing and redirecting some descriptors – all that is managed by the `popen()` call.
- `FILE *popen(const char *command, const char *mode);`
  `int pclose(FILE *stream);`

7.19 `popen()` and `pclose()` example code

```
/* Echo the output of "ls" command to my standard
 * output. */
FILE *ptr;
char *cmd = "/usr/bin/ls -d *";

if ((ptr = popen(cmd, "r")) != NULL) {
        while (fgets(buf, BUFLEN, ptr) != NULL)
                printf("%s", buf);
} else
        err(1, "popen");

pclose(ptr);
```

- We silently assume in the code that you know how to work with a FILE type. This was not in the [unix-prog-I] lecture. See man page for `fopen()` and `fclose()` if needed.
- If you use threads, you should use `popen/pclose` instead of thread unsafe `system(3c)`.
- `adv-ipc-and-io/popen.c`
- **Exercise:** modify the code so that you feed your standard input through: `tr [[:alpha:]] [[:upper:]]` using `popen()`. Print the output of `tr` on the standard output of your program.

7.20 Mechanisms for Creating a Process

We know functions that somehow create a process for us:

- `fork(void)`
- `fork_all(void)`
  - forks and duplicates all existing threads. That was the former default behaviour of Solaris threads, `fork_all()` now exists to allow backward compatibility.
- `popen(const char *command, const char *mode)`
- `system(const char *string)`

Now we add a new one:

- `posix_spawn(...)`
- `fork_all()` was already mentioned in [unix-prog-I].

7.21 `posix_spawn()`, `posix_spawnp()`

- creating a new process from a process image in a single step
  - no need for a fork and exec in the program itself
  - HW architectures w/o dynamic address translation can not easily provide `fork()` since the addresses in the process are physical addresses
- it is part of POSIX real time extensions (POSIX.1d), see [posix.1d]
- note that there must be means to specify various attributes normally dealt with after `fork()` but before `exec()`
  - user/group ID changes, signal mask and file descriptor manipulations
- See [unix-prog-I] for more info on the POSIX standards.

7.22 Using `posix_spawn()`

int **posix_spawn**(pid_t *restrict *pid*,
const char *restrict *path*,
const posix_spawn_file_actions_t **file_actions*,
const posix_spawnattr_t *restrict *attrp*,
char *const *argv*[restrict],
char *const *envp*[restrict]);

- `posix_spawnp()` function call is the same as `posix_spawn()` but `PATH` is used if the `path` argument does not contain a slash
- you need `path` and `argv`, everything else can be set to `NULL`
- however, to read from a pipe, for example, you will need to use *file actions*
- The *pid* will be filled with the child's PID.
- *attrp* contains various attributes applied on the process like user/group IDs, signal masks, process group membership, ...
- File actions contains info on how to deal with individual file descriptors (if needed). If `NULL`, the behaviour will be the same as with `fork()`, including honoring the `FD_CLOEXEC` flag.
- *argv* is mandatory, with at least `argv[0]` (+ `argv[1]` containing NULL).

7.23 File actions

- if you want to duplicate the standard output to a pipe in the child, for example, you must use file actions

```
posix_spawn_file_actions_t factions;

pipe(p);
flags = fcntl(p[0], F_GETFL);
/* FD_CLOEXEC works together with file actions. */
fcntl(p[0], F_SETFL, flags | FD_CLOEXEC);
posix_spawn_file_actions_init(&factions);
posix_spawn_file_actions_adddup2(&factions, p[1], 1);
posix_spawn_file_actions_addclose(&factions, p[1]));
```

- File actions are performed in the child in the order in which they were added to the file action object. File actions are performed before file descriptors tagged with the `FD_CLOEXEC` flag are closed.
- One more file action you can use is `posix_spawn_file_actions_addclose`.
- `adv-ipc-and-io/posix_spawn.c`
- **Exercise:** change the code so that the new process blocks `SIGTERM` signal, then call „`sleep 99`" from it, and then send it a `SIGTERM` signal from the parent. After the parent exits check that the child is stil alive, verifying that the mask has been set properly. You will have to use the `attrp` argument of the `posix_spawn()`.

7.24 Solaris Event Ports

- existing "old" event notification frameworks
  - AIO
  - timers
  - `poll()`
- no unified way to reap an application's completion of events in Solaris before the event ports
- all of these frameworks built independently
- no unified mechanism to reap events
- varying performance and scalability of the existing frameworks
- **need for one common API**

- Solaris event ports first appeared in Solaris 10.
- Timers were not mentioned in [unix-prog-I] nor in this material. See `setitimer()` and `clock_gettime()` for more information.

7.25 Solaris Event Ports (continuted)

- the fundamental piece of the event completion framework is the *port*
- applications use ports to register and reap events on the objects of interest

```
/* Create a port. */
int portfd = port_create();

/* Register the events you are interested in. */
port_associate(portfd,  ... );

/* Block until an event appears on the port. */
port_get(portfd,  ... );
```

- Solaris event ports first appeared in Solaris 10.
- Source code example not provided yet. See manual pages for respective functions.

- References:
  - http://developers.sun.com/solaris/articles/event_completion.html
  - http://blogs.oracle.com/dap/entry/event_ports_and_performance
  - http://blogs.oracle.com/barts/entry/entry_2_event_ports

7.26 FreeBSD KQueue

- Kqueue is another event notification interface
- first appeared in FreeBSD 4.1
- Kqueue enables the user to receive alerts regarding events on specified targets very quickly
- ported to NetBSD and OpenBSD
- See `kqueue(2)` manual page on FreeBSD for more information.
- References:
  - http://people.freebsd.org/~jlemon/papers/kqueue.pdf

7.27 `libevent` – a portable aproach to event notifications

- asynchronous event notification software library
- can use `/dev/poll`, Solaris event ports, kqueue, `select()`, `poll()`, and `epoll(4)`
- if you need an event notification framework on several different platforms, `libevent` might be exactly what you are looking for
  - should compile on Linux, *BSD, Solaris, Max OSX, and Windows
- written by Niels Provos

- `epoll(4)` is an I/O event notification facility in the Linux kernel.
- We already mentioned `/dev/poll` on p. 61.
- References:
  - http://www.monkey.org/~provos/libevent
  - http://blogs.oracle.com/dap/entry/libevent_and_solaris_event_ports

8 Secure Programming

- Motivation/goal:
  - learn from the mistakes of others
  - know how to write secure code (approach, techniques)
  - be aware of what could go wrong/sensitive areas
- We are not dealing with:

  - secure use of cryptography primitives
  - side-channel attacks
  - robust secure network protocol design
  - exploit techniques and analysis (1)

(1) while technically interesting topic (e.g. for more in depth understanding of call procedures, assembly, HW architecture and internals in general) this is certainly out of scope and not in line with the main goal of this chapter

8.1 More secure programs

- Prevention:
  - use of secure functions (strlcpy/strlcat)
- Making attacker's life harder
  - random allocations (mmap, malloc, ld.so)
  - protection of memory segments (non-executable stack) (1)
- Impact minimization
  - privilege revocation (ping)
  - privilege separation (OpenSSH)
  - isolation (chroot, systrace, virtualization)
  - separate uids for each service (`_ntp`, `_snmpd`, ...)
  - buffer overflow detection and reaction (canary in the stack)

(1) non-exec stack can be worked around by e.g. overwriting return address pointer with address of `system()` routine in libc and constructing a frame which contains `"/bin/sh"` as an argument.

8.2 Generic rules

- Check input data thoroughly (and do the right thing if they do not fit)
- *Do not assume* (1) (but rather check to confirm)
- Focus on corner cases (and go through all of them in detail)
  - this includes handling failures properly (2)

(1) A quotation from a whiteboard in office of an engineer.

(2) I.e. returning failure code all the way up, freeing unneeded resources and reacting on it

  - e.g. memory leak in error path can lead to Denial Of Service (DoS) by injecting number of invalid requests. See CR XXX (n2cp leak).

Task:

1. Intro: KSSL is in-kernel SSL proxy in OpenSolaris/Solaris. It had a interoperability bug related to SSL/TLS protocol implementation.

2. problem statement: *"... we needlessly fail a client hello request that has other compression methods in addition to the mandatory null compression method. This should be allowed since the spec allows supporting only the mandatory CompressionMethod.null and ignoring any other methods in client hello."*

3. see the webrev `XXX/client_hello.webrev.1` with the proposed fix

   - focus on `kssl_handle_client_hello()`

4. input info:

   - m-block structure (`mblk_t *mp`) represents SSL Client Hello message
   - `mp->b_rptr` is the beginning (pointer to area where we can start reading), `b_wptr` is the end (pointer to area where we can start writing)
   - see the definition in `usr/src/uts/common/sys/stream.h`:

```
363 /*
364  * Message block descriptor
365  */
366 typedef struct msgb {
367  struct msgb *b_next;
368  struct  msgb *b_prev;
369  struct msgb *b_cont;
370  unsigned char *b_rptr;
371  unsigned char *b_wptr;
372  struct datab  *b_datap;
373  unsigned char b_band;
374  unsigned char b_tag;
375  unsigned short b_flag;
376  queue_t *b_queue; /* for sync queues */
377 } mblk_t;
```

5. the assignment:

   (a) find the problem with the proposed fix
   - hint: it's handy to look into the TLSv1 spec RFC 2246 (sections 4.3, 7.4.1.2)
   - use code review/inspection approach (reader role)
   - What would happen if someone exploited the problem ? (see first point of this Task)

   (b) once the problem is clear, propose correct solution
   - compare your solution with `XXX/client_hello.webrev.2`

8.3 Checking where necessary

- checking return values is not only useful for graceful handling of errors but can also have security impact
- assuming a function does always the right thing can later undermine sensitive area of code

Example: `rtld` local root exploit in FreeBSD 7.2

- intro: in FreeBSD, when loading setuid/setgid programs into memory, dynamic (run-time) loader scrubs the environment so it does not contain insecure (read user supplied) variables. (mainly `LD_LIBRARY_PATH` and `LD_PRELOAD`)
  - this is done via `unsetenv()` defined in http://www.freebsd.org/cgi/cvsweb.cgi/src/lib/libc/stdlib/getenv.c:unsetenv()
- see the code in: http://www.freebsd.org/cgi/cvsweb.cgi/src/libexec/rtld-elf/rtld.c
  - function `_rtld()` is the "main entry point for dynamic linking". It contains this section:

```
trust = !issetugid();

ld_bind_now = getenv(LD_ "BIND_NOW");
/*
 * If the process is tainted, then we un-set the dangerous environment
 * variables.  The process will be marked as tainted until setuid(2)
 * is called.  If any child process calls setuid(2) we do not want any
 * future processes to honor the potentially un-safe variables.
 */
if (!trust) {
    unsetenv(LD_ "PRELOAD");
    unsetenv(LD_ "LIBMAP");
    unsetenv(LD_ "LIBRARY_PATH");
    unsetenv(LD_ "LIBMAP_DISABLE");
    unsetenv(LD_ "DEBUG");
    unsetenv(LD_ "ELF_HINTS_PATH");
}
```

- let's look at GETENV(3) man page which describes the semantics of return values of `unsetenv()`:

```
     The unsetenv() function deletes all instances of the variable name
     pointed to by name from the list.

RETURN VALUES
...
     The setenv(), putenv(), and unsetenv() functions return the value 0 if
     successful; otherwise the value -1 is returned and the global variable
     errno is set to indicate the error.

     [EINVAL]           The function getenv(), setenv() or unsetenv() failed
                        because the name is a NULL pointer, points to an empty
                        string, or points to a string containing an ``=''
                        character.
...
     [EFAULT]           The functions setenv(), unsetenv() or putenv() failed
                        to make a valid copy of the environment due to the
                        environment being corrupt (i.e., a name without a
                        value).  A warning will be output to stderr with
                        information about the issue.
```

- the first case is no-variable/no-value, the latter is variable/no-value.

```
/*
 * Unset variable with the same name by flagging it as inactive.  No variable is
 * ever freed.
 */
int
unsetenv(const char *name)
{
        int envNdx;
        size_t nameLen;

        /* Check for malformed name. */
        if (name == NULL || (nameLen = __strleneq(name)) == 0) {
                errno = EINVAL;
                return (-1);
        }

        /* Initialize environment. */
        if (__merge_environ() == -1 || (envVars == NULL && __build_env() == -1))
                return (-1);

        /* Deactivate specified variable. */
        envNdx = envVarsTotal - 1;
        if (__findenv(name, nameLen, &envNdx, true) != NULL) {
                envVars[envNdx].active = false;
                if (envVars[envNdx].putenv)
                        __remove_putenv(envNdx);
                __rebuild_environ(envActive - 1);
        }

        return (0);
}
```

- it calls `__merge_environ()` which contains this code:

```
          if (origEnviron != NULL)
                for (env = origEnviron; *env != NULL; env++) {
                        if ((equals = strchr(*env, '=')) == NULL) {
                                __env_warnx(CorruptEnvValueMsg, *env,
                                    strlen(*env));
                                errno = EFAULT;
                                return (-1);
                        }
```

- the `*envp[]` array contains pointers to strings like: `HOME=/home/user` , `TERM=xterm` and such
  - so if we replace one of the env strings by value-less string `unsetenv()` will bail out as above and will not unset the variable

- compare the above with OpenSolaris approach of handling setuid/setgid programs: http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/cmd/sgs/rtld/common/setup.c:setup()
  - `setup()` which contains the main rtld functionality calls `security()` which performs the setuid/setgid check:

```
/*
 * Determine whether we have a secure executable.
 */
security(uid, euid, gid, egid, auxflags);
```

  - `security()` defined in http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/cmd/sgs/rtld/common/util.c#security only sets the per-process `RT_FL_SECURE` flag (lines 3464, 3474, 3481).

```
3445 /*
3446  * Determine whether we have a secure executable.  Uid and gid information
3447  * can be passed to us via the aux vector, however if these values are -1
3448  * then use the appropriate system call to obtain them.
3449  *
3450  *  -If the user is the root they can do anything
3451  *
3452  *  -If the real and effective uid's don't match, or the real and
3453  * effective gid's don't match then this is determined to be a 'secure'
3454  * application.
3455  *
3456  * This function is called prior to any dependency processing (see _setup.c
3457  * Any secure setting will remain in effect for the life of the process.
3458  */
3459 void
3460 security(uid_t uid, uid_t euid, gid_t gid, gid_t egid, int auxflags)
3461 {
3462         if (auxflags != -1) {
3463                 if ((auxflags & AF_SUN_SETUGID) != 0)
3464                         rtld_flags |= RT_FL_SECURE;
3465                 return;
3466         }
3467
3468         if (uid == (uid_t)-1)
3469                 uid = getuid();
3470         if (uid) {
3471                 if (euid == (uid_t)-1)
3472                         euid = geteuid();
3473                 if (uid != euid)
3474                         rtld_flags |= RT_FL_SECURE;
3475                 else {
3476                         if (gid == (gid_t)-1)
3477                                 gid = getgid();
3478                         if (egid == (gid_t)-1)
3479                                 egid = getegid();
3480                         if (gid != egid)
```

```
3481                                    rtld_flags |= RT_FL_SECURE;
3482                }
3483        }
3484 }
```

- `LD_PRELOAD` handling is done in `ld_preload()` defined in http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/cmd/sgs/rtld/common/setup.c#ld_preload:

```
/*
 * If this a secure application, then preload errors are
 * reduced to warnings, as the errors are non-fatal.
 */
if (rtld_flags & RT_FL_SECURE)
        rtld_flags2 |= RT_FL2_FTL2WARN;
if (expand_paths(*clmp, ptr, &palp, AL_CNT_NEEDED,
    PD_FLG_EXTLOAD, 0) != 0)
```

- `expand_paths()` defined in http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/cmd/sgs/rtld/common/paths.c#expand_paths contains the actual check:

```
/*
 * If this a secure application, validation of the expanded
 * path name may be necessary.
 */
if ((rtld_flags & RT_FL_SECURE) &&
    (is_path_secure(str, clmp, orig, tkns) == 0))
 continue;
```

  * this is different approach (set flag and check where needed) which allows bigger flexibility. (but the checks have to be in the right places as opposed to scrub-before-doing-anything approach used in FreeBSD)
    - ld.so.1(1) SECURITY section has more details, section FILES sums it up: `/lib/secure` and `/usr/lib/secure` are `LD_PRELOAD` locations for secure applications. `/lib/secure/64` and `/usr/lib/secure/64` are `LD_PRELOAD` locations for secure 64-bit applications.
    - the locations are empty by default and writable only by `root`.

- References:
  - http://xorl.wordpress.com/2009/12/01/freebsd-ld_preload-security-bypass/
  - http://stealth.openwall.net/xSports/fbsd-rtld-full-package

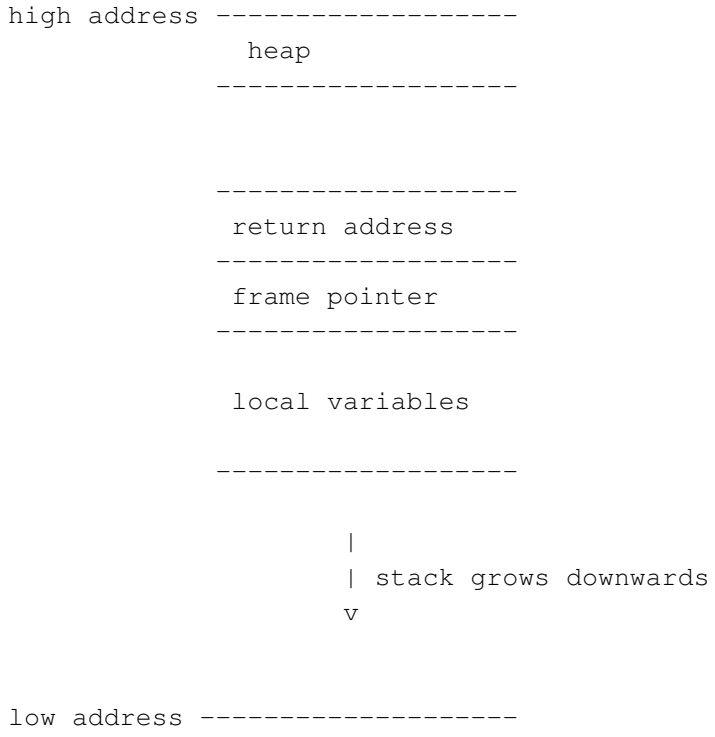
8.4 Function classification

- Library functions can be divided into several equivalence classes according to how secure/usable they are
  - Some functions are inherently insecure and cannot be safely used at all
  - Other functions require great deal of attention to get the code right (correct+secure code)
  - There are even some functions which offer consistent API/behavior and make it easier to work with corner cases

- categorized list of functions: http://hub.opensolaris.org/bin/view/Community+Group+security/funclist
- Secure programming guidance at OpenSolaris Security community http://www.opensolaris.org/os/community/security/library/secprog
- Secure programming presentation by Scott Rotondo http://opensolaris.org/os/community/security/library/secprog/secure_prog.pdf

8.5 Buffer overflows

- informal definition: writing past the end of a buffer
  - comon cases:
    - string buffers but can be any buffer space
    - buffer on the stack but applies also for on the heap buffers
- the danger:
  - the surroundings of the buffer being overflowed (stack)
    - function return address (redirecting code flow) (1)
    - frame pointer (altering code flow)
  - persistent system space (heap)
    - FILE I/O space
- What happens on overflow:



(1) Smashing The Stack For Fun And Profit, Phrack Volume Seven, Issue 49: http://www.phrack.com/issues.html?issue=49&id=14&mode=txt

8.6 Safe string manipulation 1/2

- no boundary check: gets(), strcpy(), strcat(), sprintf()/vsprintf() (1)
- watch out when using: strncpy()/strncat()

(1) unsafe functions:

  - gets() does not check boundaries at all

- potentionally unsafe / avoid:
  - strcpy(), strcat()
    - do not check boundaries at all
    - strcpy() does not zero terminate on overflow (but strcat() does)
    - possible to calculate the necessary dst size for strcat() but hard to follow the code
  - sprintf()/vsprintf()
    - no boundary check
    - possible approach: compute the buffer size + control the fmt string
- potentionally unsafe / use with caution:
  - scanf() family
  - scanf() should not be called without limit
  - `scanf("%s", str);` is asking for buffer overflow
  - `scanf("%10s", str);` needs 11 characters buffer (+ terminating `'\0'`)
- snprintf()/vsnprintf()
  - return the number of characters necessary (without the zero), not the actual number of characters written to the buffer - watch out for constructions like this: p += snprintf(p, lenp, "...");
- strncpy()
  - does not zero terminate on overflow
  - zero-fills the remainder of the dst buffer in no-overflow case (perf)
- strncat()
  - always terminates (even on overflow)
  - is not intuitive - some arithmetics for the size argument is almost always needed

8.7 Safe string manipulation 2/2

- overflow aware+helping: string(3C):
  `size_t` **strlcpy**`(char *`*dst*`, const char *`*src*`, size_t` *dstsize*`);`
  `size_t` **strlcat**`(char *`*dst*`, const char *`*src*`, size_t` *dstsize*`);`
  - always terminate with zero, even in case of buffer overflow
  - do not zero-out the rest of the dst string

  - easy truncation detection
  - the *dstsize* argument is the total space in *dst*
  - for `strlcat()` nothing happens if the dst string is longer than size
  - `strlcat()` returns `min{dstsize,strlen(dst)}+strlen(src)`

(1) unfortunately, `strlcpy()/strlcat()` are not available in Linux world

  - the main reasons for rejection (for inclusion into GNU libc) seem to be:
    1. the functions are non-standard
    2. risk of truncation is considered to be greater than risk of overflow

- the exact reasons are not very clear because of the strong language used by RedHat developers who are opposing the idea:
  - http://sources.redhat.com/ml/libc-alpha/2000-08/msg00061.html
  - http://sources.redhat.com/ml/libc-alpha/2000-08/msg00053.html (links from http://en.wikipedia.org/wiki/Strlcpy#Criticism)
- that's why the Task below is even more important when writing code which uses static arrays because to make the code work on Linux as well there are basically 2 choices:
  - either use the unsecure/inconsistent functions (not recommended)
  - attach strlcpy/strlcat implementation (just like many projects do)
- ensuring zero termination with `strncpy()/strncat()`:
  - pass `dstlen - 1` as length and terminate by hand:

```
dst[dstlen] = '\0';
```

    * if the dst variable is static or `calloc()`'ed the zero termination is not necessary but the code is then harder to inspect for overflow problems
- problems with using `strncat()`:
  - the size passed is the free space available in the buffer (not the total size of the buffer)
  - the size argument must not count the terminating zero (even though the function always terminates)
- check for buffer overflow/truncation:

```
        if (strlcpy(dst, src, dstsize) >= dstsize)
                return (-1);

        if (snprintf(buf, sizeof (buf), "%s", src) >= sizeof (buf))
return (-1);
```

  - the 'equal' case is where the string fits but without the terminating zero
- `strlcpy()` paper: http://www.openbsd.org/papers/strlcpy-paper.ps

Task: Write a secure (detects and prevents buffer overflow) function **foo()** which constructs a string using the folllowing rule. The size of the destination string is passed to the function. Use hard-coded value in the `main()` calling **foo()** (use e.g. `MAXPATHLEN` define from `<sys/param.h>`).

- prototype: `int` **foo**`(char *input, char *dst, size_t dstlen);`
- rule (+ means concatenation):

```
env("HOME") + "/" + argv[1] + "/.foorc"
```

  - which means `main()` will pass `argv[1]` to **foo()** as *input*.

- write using:
  1. `strcpy()`/`strcat()` (optional, only for the really patient)
  2. `strncpy()`/`strncat()`
  3. `strlcpy()`/`strlcat()`
- The function should return the length of the string or in case of overflow a negative number which if placed in `abs()` is equal to the number of missing characters in the destination string.
- Construct number of unit tests for all possible cases (most importantly for corner cases) and verify that each of the implementations works correctly.

8.8 Time of use versus time of check

- possible race conditions - a window between check and use where the actual object can be swapped/changed but the reference to the object remains the same.
- setuid/setgid program often needs to verify that once it switches to a user the user will be able to read a file
- access(2) does the check using real uid/gid
- Example of insecure approach (assume `path` points to a file in directory writable by others):

```
/* we're running with euid = 0, and ruid = userid != 0 */
if (access(path, R_OK|W_OK) < 0)
        return -1;

fd = open(path, O_RDWR);
...
```

  - access(2) resolves the path to a XXX and performs the check. After that the XXX object is free'd. open(2) does the lookup again XXX. Problem is that this time different object can be used - the operation of check and return is not atomic. Now the setuid application might be writing to different file.
  - correct approach: switch to the user and try to open the file -¿ no race condition

```
seteuid(getuid());
if ((fd = open(path, O_RDWR)) == -1)
    err(1, "open")
seteuid(0);
```

- similar problem with stat():

```
/* make sure "path" is a regular file before opening */
if (stat(path, &buf) == -1 || !S_ISREG(buf.st_mode))
   err(...);
fd = open(path, O_RDONLY);
```

  - the solution:

```
if ((fd = open(path, O_NONBLOCK|O_RDONLY)) == -1)
   err(...);
if (fstat(fd, &buf) == -1 || !S_ISREG(buf.st_mode))
   err(...);
```

  - `O_NONBLOCK` will prevent blocking when opening a device

- other functions which take file descriptor as argument instead of filename: `fchmod()` (chmod(2)), `fchdir()` (chdir(2)), `fchroot()` (chroot(2)), ...

8.9 Memory locking

- prevents a memory region from being swapped out to disk (1)
- mlock() is perfectly useful for password managers, for example
- it seems one must use mmap() on Solaris to ensure the correct memory alignment (2)
- plock() is more generic but less granular
  - allows locking of text/data segments
- mlockall() locks every current/future mapped page in the address space
  - the same semantics as mlock()
  - can do selective unlock with munlock()

(1) semantics:

- no fork() inheritance
  - unless `MAP_PRIVATE` mmap() flag is used
- locking of the same page by multiple processes is reference counted
  - the last process which unlocks results in the page being unlocked
- single process locking of the same page is not nested
  - i.e. single unlock of page locked multiple times does unlock

(2) see `secure-prog/mlock.c`

8.10 Temporary files handling

- mktemp(3C): `char *`**`mktemp`**`(char *`*`template`*`)`
  - should not be used because of race condition between check and create
- mkstemp(3C)/mkdtemp(3C): `int` **`mkstemp`**`(char *`*`template`*`)`
  - secure, race condition-less variants - return file descriptor of the temporary file and replaces the template argument with filename
- XXX following symlinks

Example: mktemp(1) in shell script

```
TMPFILE=`mktemp /tmp/example.XXXXXX`
if [[ -z "$TMPFILE" ]]; then exit 1; fi
print "program output" >> $TMPFILE
```

8.11 Off-by-one errors / Integer overflows

- `calloc()` can overflow internally (1)
- XXX off-by-one

(1) when computing the size of the number of bytes to be allocated XXX

8.12 Format string overflows

- XXX user-supplied format string can be used to overflow stack of the process - examples XXX:

8.13 Privilege separation

- the goal: minimize the amount of code running as root/with privileges
- basic principle - one unprivileged process for input data processing, another process for performing privileged tasks. they communicate with each other using set of well defined messages.

XXX picture

8.14 Examples of privilege separated programs

- OpenSSH:
  - master process: accepts new connections, then immediately forks new process group (1)
  - alternative privilege separation model (2)
- OpenBGPD
  - parent: runs as root, enters routes into kernel - works with routing socket (3)
- OpenNTPD

  - parent: runs as root, sets the time
  - ntp engine: runs as non-root uid/gid `_ntp:_ntp` and chrooted to `/var/empty`
  - simple communication (4)

(1) process group consists of:

  - monitor: privileged process, handles authentication
  - listener: SSH communication over the network

(2) OpenSSH: XXX privsep protocol with XXX messages - SunSSH: altprivsep with XXX messages

(3) children processes:

  - child #1 (session engine): manages sessions (TCP connections) - runs as unprivileged user `_bgpd`, chroots to `/var/empty`
  - child #2 (route decision engine): works with routing tables - runs as unprivileged user `_bgpd`, chroots to `/var/empty`

(4) `socketpair`, 2 message types (set the time, lookup hosts using the files in `/etc`)

8.15 Privilege revocation/bracketting

- use only the privileges which are needed (at the time they are needed)
- `seteuid()`
  - better: `setreuid(getuid(), getuid());`
- privilege awareness (privileges(5) on Solaris)

8.16 Non-executable stack

- XXX
- XXX OpenBSD approach for platforms lacking N-X

```
cc -M /usr/lib/ld/map.noexstk myprogram.c
```

8.17 Prevention techniques and tools

- check return values of functions (and interpret + act upon them correctly)
  - use `-Wall` and fix all warnings
- chroot(2) syscall
  - close unneeded descriptors **and `chdir()` and `chroot()`**
- automated checking
  - lint, Parfait, XXX
- stress testing (1)
- peer code reviews

- do the what-went-wrong case studies regularly (2)

(1) try to break the program (e.g. by feeding it all sorts of bogus data, injecting faults, etc.)

(2) Analyses of vulnerabilities/fixes: http://xorl.wordpress.com (be sure to double check/do your research, though)

- References:
  - CERT C secure coding standard: https://www.securecoding.cert.org/
  - Secure Coding in C and C++ (Robert C. Seacord) http://www.cert.org/books/secure-coding/

ChangeLog

2016-03-10 Removal of some OS specific sections from ,,Debugging" section

2016-03-08 Minor changes to testing, start using new tasks formatting.

2016-03-04 Overhaul of the formatting and style.

2010-10-25 More information on a difference between Bash and the Korn shell, p31.

2010-05-11 More info on the `SIGHUP` signal, p. 35.

2010-02-18 Some fixes in the terminal chapter.

2009-12-30 Bulk of changes for the ,,Secure programming" and ,,Advanced networking" chapters.

2010-01-03 Thread chapter.

2009-11-29 More work on the ,,Advanced IPC and IO" chapter.

2009-11-21 ,,Terminals" chapter more or less ready, initial ,,Advanced IPC and IO" section. It looks that the ,,Debugging" chapter is also quite ready.

2009-11-10 better ,,Testing" section, initial ,,Debugging" chapter.

2009-08-30 1st version of this material compiled.