CptS 360 (System Programming)
Unit 10: Process Control

Bob Lewis

School of Engineering and Applied Sciences
Washington State University

Spring, 2022

Motivation

- Processes are fundamental components of operating systems.
- “Where do processes come from?”
- Multiple processes take advantage of multi-core architectures trivially, but you need to control them.

References

- Stevens & Rago Ch. 8
- *man* pages

Process, User, and Group Identifiers

These are attributes of your process.

- the process and its parent IDs
  - *getpid(2)*
  - *getppid(2)*
- user IDs
  - *getuid(2)*
  - *geteuid(2)*
- group IDs
  - *getgid(2)*
  - *getegid(2)*

fork(2)

- only way to create a new process in POSIX
- near-clone (”child”) of parent process created
- first half of the famous “fork-exec” concept
  (which is also known as “spawn”)
- on Linux, *fork(2)* is implemented as a special case of *clone(2)*, which is
  - more flexible
  - less portable (being non-POSIX)
  - also used for threads

The Child's Inheritance

- UIDs (all 4)
- supplementary GIDs
- process group ID
- session ID (?)
- controlling terminal
- set-user-ID and set-group-ID flags
- current working directory
- root directory
- umask
- signal mask
- etc. (see Stevens & Rago)

Copying all of this stuff to the child process is part of *fork(2)*'s overhead.

Child vs. Parent

child and parent *differ* in

- return value from *fork(2)*
- process ID's
- parent PID's (duh)
- time usages of child are zero'd
- file locks not inherited
- pending alarms (SIGALRM) cleared for child
- pending signal set is cleared for child

The Child's Memory

- Child gets copy of all of parent's data space:
  - initialized
  - uninitialized
  - stack
  - heap
  - `environ` and `argv[]`
- Linux implements "copy-on-write"
  - reduces overhead (usually), especially if a *fork(2)* is following
  - child's pages may diverge from parent's (consider implications for low-level vs. stdio on parent/child I/O transfers.)

Uses for *fork(2)*:

- spawning a different process
- accumulating run statistics on child
- debugging child (*gdb(1)* uses this)
- parallel processing, e.g.
  - a network server
    (in general, threads would be better for this)
- On UNIX, *vfork(2)* guarantees that child executes first.
  (Don't count on this, though.)
- Under Linux, *vfork(2)* is *almost* a synonym for *fork(2)*.

(Run the `demos/dn_sr3e_fork1` demo.) (S & R Figure 8.1?)

The Semantics of *exit(3)*

- If parent terminates before child,
  - *init(1)* (on Linux, *systemd(1)*) process becomes new parent
- If child terminates before parent *wait(2)*s for it,
  - it's a "zombie". (One of the cooler UNIX concepts.)
- Avoid creating lots of zombies. (You've seen the movies!)
- Avoid zombies completely by forking twice.
  (see S & R Figure 8.8)

Race Conditions

- Q: What's a race condition?
  A: Output depends on the arbitrary order in which processes run.
  - Example: Child used to create a file that the parent is going to read.
  - If child creates, writes, and closes the file first, everything okay.
  - If parent calls *open(2)* first,
    - child's *open(2)* might fail
    - parent's *read(2)* might contain only partial data
- Especially annoying: It might work *most* of the time, so test for it is unreliable.
- Solution: Parental "wait".
- Other situations may not be so easy.
- General solution: use IPC (e.g., signals) to coordinate.

wait(2)

When parent has nothing else to do ...

- *wait(2)*
  wait for any child to exit
- *waitpid(2)*
  wait for a particular child to exit
- *waitid(2)*
  waits for specific conditions on specific children
- Use handy macros if you want to find out *why* child exited.

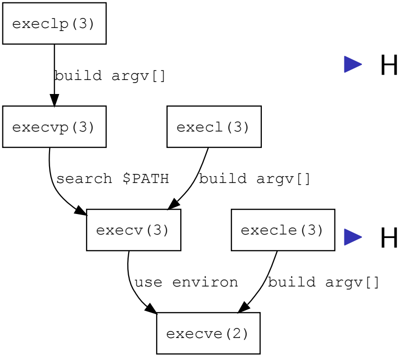
(Run the `demos/dn_sr3e_wait1` demo.) (S & R Figures 8.5)

execve(2) and Friends

- *execve(2)*
  - the basic system call
  - *replaces* caller's address space with that contained in an executable file
  - only returns if there's an error
- second half of the famous "fork-exec" dynamic duo
- friends:
  - *execl(3)*
  - *execv(3)*
  - *execle(3)*
  - *execlp(3)*
  - *execvp(3)*

These are all related...

exec* Suffixes Determine...



- What is the child's enviroment?
  - "e": the caller provides an environment
  - otherwise, it inherits the caller's
- How does parent pass the arguments?
  - "l": it wants successive "char *" arguments
  - "v": it takes a single, NULL-terminated list (like argv[])
- How is the path argument resolved?
  - "p": it searches your $PATH (note potential security hole). path can be absolute.
  - otherwise, path *must* be absolute.

Changing User IDs and Group IDs

- use the least privilege model to minimize security risk
- *setuid(2)*
  - if you're root, this sets real, effective, and saved set-user-id
  - if you're setting your uid to your real or saved uid, this works, but only changes your effective uid.
- *setgid(2)*
  - likewise for gid's

How do Scripts Work?

- The file you *exec(3)* doesn't need to be a binary file.
- It does, however, need the proper execute bit set.
- "`#!`" at the start of a file is special to `exec*`'s:
  - The rest of the line is the name of a program to run.
  - The rest of the *file* is sent to that program on standard input.
- This works for any executable *exec(3)* finds, even your own.

(Run the `demos/dn_scripts` demo.)

system(3)

- fork-execs a (child) shell
- shell runs command (“`/bin/sh -c` *command*”)
- parent waits for child
- some blocked signals can lead to trouble, see suggested fix in man page

(Run the `demos/d`*n*`_system_call` demo.)

Process Accounting

- needs kernel compiled with accounting feature (I don't think that applies to WSUTC.)
- *acct(2)*
- logs accounting information to a log file
- format of file described in *acct(5)*
- use `/usr/include/sys/acct.h` to get struct format (`struct acct`)

User Identification

- These get your process's login:
  - *getlogin(3)*
  - *getlogin_r(3)*
- other possibilities:
  - `getpwuid(getuid())`
    - possibility of multiple entries for given UID
    - choose shell or home directory by login
    - unusual, but legal
  - `getenv("LOGNAME")`
    - manpage recommends it
    - S & R say not for authentication

Process Times

times(2) fills in a `struct` with

- CPU time spent in user mode by calling process
- CPU time spent in system mode by calling process
- CPU time spent in user mode by all descendents
- CPU time spent in system mode by all descendents

Units are "clock ticks". To convert them to seconds, divide by `sysconf(_SC_CLK_TCK)` (number of clock ticks per second). (See *sysconf(3)*.) Note: `clock_t` is the same type returned by *clock(3)*, but in that case the units are defined differently.