CS503A Lecture #4 2019-06-10

Administrivia

- shell pipeline: top-ten wordcount
- filter programs

Our Story So Far

- Process: (essentially) running program
  - More precise definition: container for running program
    - ↳ might be bad choice of terminology because we use "container" for light-weight VMs (e.g. docker)

- in Posix-like systems, new process is created by fork() system call - clones current running program

- new program can replace current program in existing process with exec* system call
  - ↳ being sloppy here: one system call with several library wrappers
  - If exec. is successful it does not return

- Separating program initiation from process creation was a design choice
  - pros: original program can do some housekeeping (e.g. reopen fd 0, 1, 2)
    can divide & conquer
  - cons: more complicated to do the most common use pattern

OSSF (cont.)

- open files stay open across fork, exec
  - unless special system call to set close-on-exec flag (for individual files)
  - especially fd 0, 1, 2, but applies to all open files in process

- program termination: parent collects child exit status
  - what is preserved (always)
  - unless parent dies first
  - exit status is held in process table until collected
    - zombie process: almost dead
  - no problem if zombie status is short-lived
    - parent is waiting, but collects it soon
    - parent is ephemeral
  - overlong zombie processes is a resource leak

OSSF (cont.)

- Pipes: producer-consumer model
  - 2 endpoints, one-way communication
  - pipe(2) system call:
    - pass in array [2] ints
      - 0 - reader (consumer)
      - 1 - writer (producer)

      } file descriptors
  - file descriptors are files opened by a process, but fds stay open across fork/exec

  ⇒ two processes can communicate via pipe only if the pipe was created by their common ancestor process
  - typically, { producer closes read fd
    consumer closes write fd
  - multiple processes can share a pipe
    - but it's a bit tricky
    - likely to be brittle (error-prone)
      in practice — educated guess

- popen(3) C std lib only slightly abstracts
  - forks a process that runs a shell
  - passes your string to shell as cmd
  - returns FILE* object that wraps pipe
  - like system(3), but allows parent to collect output

USF (cont.)

- file & pipe file descriptors use same read/write system call
  - works like virtual function

- everything is a file
  - devices / pseudo devices / network connections
  - can write program so it doesn't care what it's looking to

- system calls to determine nature of open file (regular / device / etc)
  - device specific operations
  - ioctl(2) - general interface

most programs should never do this: much more flexible / better system design if your program doesn't (are)

OSSF (cont.)

- shell does string processing to determine which programs to run, argv, fd 0, 1, 2 (and collects exit status)

- shell syntax `>` `>>` `<` `|` } redirect input/output

  `|` — limited use of pipe(2) capabilities

- pipelines: data-flow style
  - filter programs

OSSP (cont.)

- Ass 1: Convex hull

Signals

- Software analog of hardware interrupts

- Signal is small integer with system-defined semantics

man 7 signal

SIGUSR1	10	} user-defined
SIGUSR2	12	} user-defined
SIGHUP	1	"hangup"
SIGINT	2	keyboard interrupt (control-C)
SIGSEGV	11	Segmentation violation
SIGALRM	14	timer
SIGCHLD	17	child process stopped/terminated

- Default action depends on specific signal
  - terminate
  - terminate & dump core
  - ignore
  - continue (SIGCONT)
  - Stop (SIGSTOP/SIGTSTP)

- process control block : Signal dispatch table

Signals (cont.)

- signal(2): system call to specify action on signal
  - prefer sigaction(2) — system gets more complex because it has to support legacy applications — backwards compatibility
- specify action
  - SIG_DFL - default behavior (for this signal)
  - SIG_IGN - ignore signal
  - function pointer
- SigKill - #9 - terminates program
  - cannot be caught (trapped)

Signals (cont.)

- Signals may be due to an event (e.g. program executes illegal instruction, or child process terminates) or explicitly sent (by another process)

- kill(2)
  - misnamed (YAMSC)
    (yet another mis-named system call
  - the guys who did the original design were figuring this stuff out for the first time

  kill sends signal, does not kill process
  - process's response to signal (SIG_DFL) is often to terminate
  - chalk the name up to "seemed like a good idea at the time"

- kill(1): thin wrapper around kill(2)

Signals (cont.)

- to dispatch signal, kernel manipulates the process's kernel data structure while the process is not running
  - push current value of program counter onto stack
  - set program counter to address of first instruction in signal handler routine
  - when function is finished, it returns to the next instruction that was supposed to run before the signal occurred

Interprocess Communication

- Cooperating processes
  - Can use machine resources more efficiently
  - Can simplify programming solutions

- already discussed some ways for 2 process to communicate
  - shared file - kludgy
  - pipe
    - useful for producer-consumer (one-way) debate
      - more efficient than file, process can proceed concurrently (uses kernel memory buffer)
    - pipe must be created by common ancestor process
      - pass around open fds
  - Program exit status
    - slightly more useful than Boolean value
    - Shell can use exit status in loops & conditionals
    - only works between parent & child
      - child must finish first
      - parent must explicitly call wait(2)
  - Signals
    - "wake up and check something"

Process

- executable file (on disk): passive data
  - can manipulate with any program that can read/write files - including text editor
  - binary file format: contents in text editor: unintelligible
    - data structure
      - #include <elf.h>
      - man 5 elf
        - section 5: file formats

executing and interpreting formats

  - "sections"
    - text (code)
    - initialized data
    - bss (uninitialized data): zero
      - doesn't take up much space on disk
      - run length encoding: compression
      - it's all zeros, just tell me how big it is
    - relocation symbol table
    - debug symbols

Process (cont.)

- Running Program
  - registers (esp. program counter, stack pointer)
  - text (code)
  - Static data (initialized / uninitialized) → Memory initialized to 0
  - Stack ("call stack")
  - Heap (arena) - dynamic memory allocation
  - open files (file descriptors)
  - argv / envp
  - exit status (on program termination)

Processes (cont.)

- Process in kernel: data structure representing process
  - you may see mechanisms former of art like process control blocks

- register save space (when process is suspended)
- memory map
- table of open files
- other info
  - pid
  - uid
  - ppid
  - gid
  - priority (for scheduling)
  - nice value → only root (superuser, admin) can have negative nicenes
- signal vector

Processes (cont.)

- process status
  - new
  - waiting / not-ready / suspended
  - ready / runnable
  - running
  - exited / zombie

⇒ Scheduler picks ready process and runs it until syscall, interrupt, timer (timeslice)

- refinements
  - short-term waiting vs. long-term waiting
    - → e.g. slow I/O device
    - → e.g. fast system request

Interprocess Communication (cont.)

- 2 basic paradigms for communication
  - message passing (aka "shared nothing")
    - e.g. pipes
  - shared memory (stay tuned)

- named pipes
  - create "special" entry in filesystem
  - open like any other file (by name) read or write mode
  - works just like pipe, except processes do not require common ancestor
  - more awkward than regular pipes for common use case: pipelines

- unix-domain sockets
  - like pipes, but uses networking interface
    - on same machine only
    - it's an API thing

- Networking
  - message passing between machines (essentially) — which means separate processes on those machines
  - separate topic (rich)
  - "cluster OS" - treat entire datacenter as system that requires resource management / abstractions