CSS503A

Lecture #2

2019-04-03

Administrivia

- questions about shell / user-level stuff
  - noodle around UW
  - CSS 390
  - LinuxFest NW 2013
  - 1e9 tutorials on the internets
  - tutorial Sunday pm & Monday after class
- Lab 1 skeleton: was locked out → commenting style → overly verbose ↓ don't do that so much ... as I say, not as I do
  - fixed & apologies for anyone inconvenienced
- 2-4 not quite available yet

I will take marks off for too little or too much commentary

Our Story So Far

- abstractions let us build large-scale systems

- choice of algorithm, implementation language, compiler flags, & low-level implementation details can make dramatic differences in performance
  - in extreme cases months-long computation may be reduced to seconds
    - no exaggeration!
  - millions of dollars of computing can be reduced to thousands

  ⇒ not always possible: some problems are intrinsically hard (computationally expensive)

- most of the time, programmer time is more expensive than computer time

OSSF (cont.)

(not exhaustive)

- Some cases where performance matters
  - large-scale systems
  - tiny-scale systems (resource-constrained)
  - real-time systems
  - operating systems
    - performance bottlenecks in OS affects everything

- real-time systems: hard deadline on computation
  - late result may be worse than no result at all

  e.g. oil refinery: not closing valve on time → boom!
  rocket: veer off course or boom!
  self-driving car: splat!

  - Zavie story: ~1980s weather models produced decent 10-day forecasts – 3 days late (hard deadline, but not what we traditionally think of as real-time)

  → also: payroll

OSSF (cont.)

- history: how we got to where we are
  - card deck

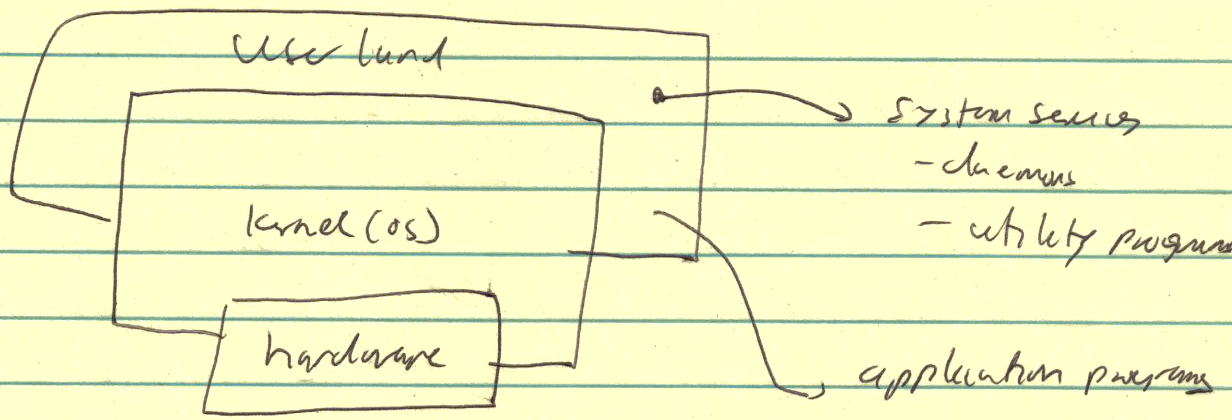
- Operating systems: no standard definition
  - kernel + system support programs or just kernel?
  - resource allocator (traffic cop) or hardware mediator?

- key concerns
  1) efficient use of hardware resources
     - device drivers
     - user-level access via system request (sys req) calls
  2) process management
     - create / tear down
     - scheduling (time slices) - manage concurrency
  3) file systems (data structure on top of storage)
  4) networking
     - low-level protocols

=> Security / safety & convenience

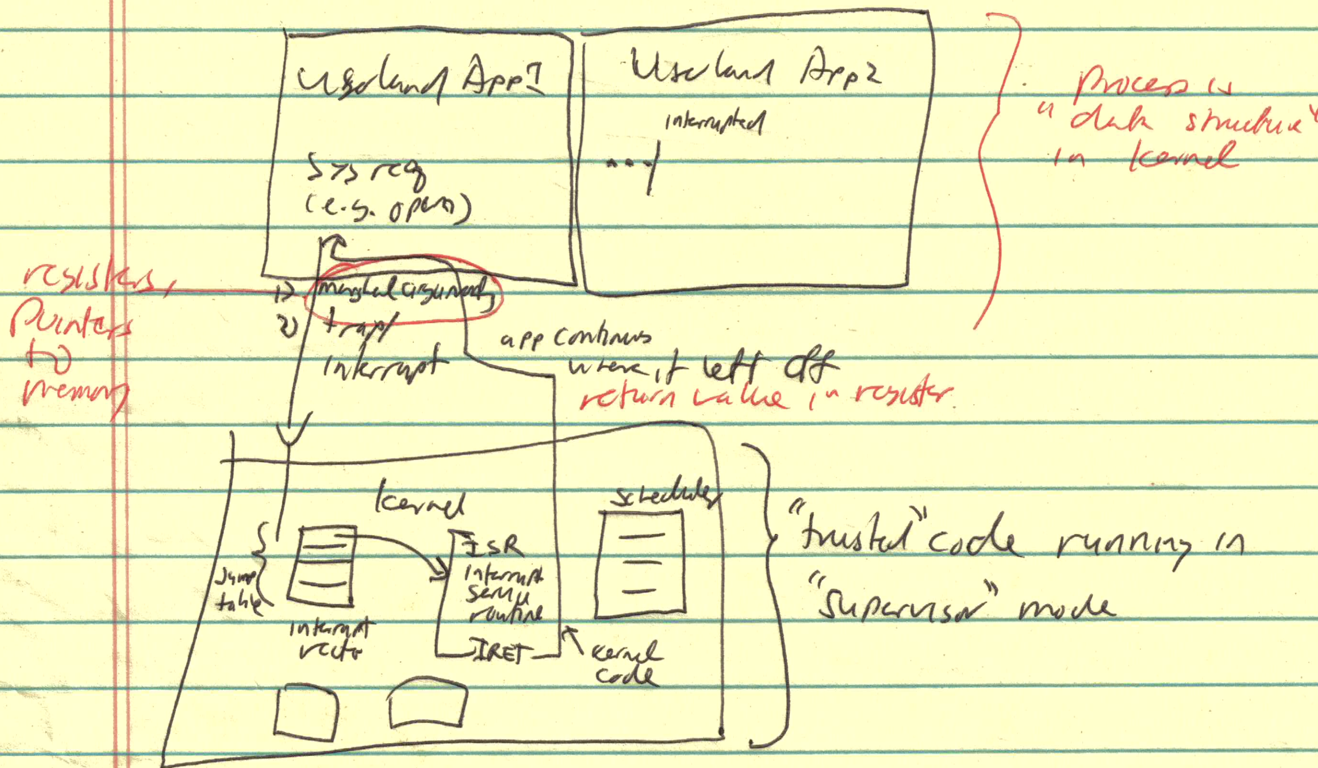
OSSF (cont.)

- layered model



- processor mode
  - normal / user (restricted)
    - no direct access to hardware
    - MMU: segmented or virtual memory
  - privileged
    - direct access to hardware,
    - MMU: real memory mode
- processes make system calls (sys req) to ask kernel for something
  - context switch (someone asked last night after class)

Layered Model (cont.)



Kernel vs User Mode

- processor (CPU) operates in 2 (or more) modes
  - user (normal) mode aka restricted
  - privileged / supervisor / kernel mode

- user mode:
  - non-privileged operations only
  - no direct access to hardware
  - typical: segmented / (OS) virtual memory (MMU)

- kernel mode
  - all operations
  - direct access to hardware
  - real memory mode (MMU)

Kernel vs User Mode (cont.)

- on system startup: processor in supervisor mode
- kernel boots up, initiates 1st process
    - loads program in memory
    - constructs internal data structures

  { Process is "data structure" in kernel view }
- kernel transfers control to process #1 via special instruction (e.g. IRET) → context switch (to user)

- userland process (so far, only one process) executes until either → context switch (to kernel)
    1) Process makes system request (trap)
    2) interrupt occurs (e.g. hardware event)
    3) (special case of 2): timer interrupt
        - time slice

- interrupt / trap transfers control back to kernel & switches processor back to supervisor mode
    - kernel handles interrupt or user request (e.g. initiate disk I/O operation)
    - kernel then checks its process table for "runnable" processes & chooses (schedules) one
    - kernel transfers control back to scheduled process

Switching Between User & Supervisor Mode

- Power on / reset: } begin executing instruction at fixed address → boot loader (boot strapping)
  - initially in supervisor mode
- Special instruction to switch to user mode
- User mode → supervisor
  - trap instruction
  - illegal instruction
  - access illegal address
    segv / bus error
  - misc other stuff
  - interrupt
    - hardware signal
      - I/O device, timer

Switching Processor Mode (cont)

trap/interrupt:

- program counter → interrupt vector
  - → jump to Interrupt Service Routine

ISR may disable (mask) interrupts while servicing current interrupt

- we want to keep this brief
- critical section

Non-maskable / Priority interrupt

kernel → usr mode:

- special instruction, eg IRET
- set register value

What's a kernel?

- Core program, always running, manages the hardware, filesystem(s), processes, etc...

- Leaky abstraction (muddying the model):
  - may be supported by userland "daemon" processes

- userland process ~~interact~~ interact with kernel via "system calls" ("traps")
  - section 2 vs section 3 of Unix manual (good interview question)
  - both look like function calls

→ want kernel to be small & fast { minimal measure set of (orthogonal) operations

~> I/O — Reality: not so small anymore: Unix had dozens of system calls, linux has 100s
- minimal ("orthogonal") set of operations
- 4 syscalls: open, close, read n bytes, write n bytes

- formatted I/O: printf, <<, etc
  ⇒ userland library functions (does not deal with hardware)

Kernel

- process management
  - create, schedule, terminate (userland) processes
- filesystem
- device drivers: hardware interface (I/O)
- networking (protocols)
- memory management
- accounting
  - there was a time when systems were billed by the CPU-second
  - these times have returned: cloud services
- protection / security
  - prevent one program from reading/writing another process's memory
    1) malicious code
    2) bugs (wild pointers)
- error detection
  - kernel panic
  - BSOD

Kernel (cont.)

- design considerations
  - always running (resident in memory)
    - → lowest possible memory footprint
  - fast performance
    - time spent deciding which task to run is time spent not running *any* task

- traditionally: written in assembly
  - now generally written in C (with critical sections in assembly)
  - C is "high-level assembler"
    - more productive for programming
    - more portable (to different computer architectures)

- compiler support: nonstandard features
  - "intrinsics"
  - inline assembly
  - ⇒ must compile windows with visual studio
    must compile linux with gcc

Userland Processes

- process: essentially a running program

- kernel starts up process #1
  - where do other processes come from?
  - spoiler alert: process #1 is called init {initial? initializer?}

- init reads config file(s), makes system calls to ask kernel to create other processes
  - system services (daemons) eg print server, cron
  - window system
  - login program (getty)

- init → getty → login → bash (shell)

Process Management

- process: (essentially) running program
  - ⇒ how to create (& tear down) processes

- program
  - executable file
  - argument vector (list of strings)
    - no intrinsic meaning to system
      - up to program to interpret
    - conventions { -f --flag --key=value
      filename ...
    - exceptions - specific programs
  - pre-opened files — magic: set to it manually
    - convention (cin, cout, cerr)
      0 1 2
  - program exit status
    - { 0 = success ("true" in shell)
      non-zero = failure ("false" in shell)
      - C/C++ std defines { EXIT_SUCCESS 0
        EXIT_FAILURE 1
      - unix allows byte (0..255)
    - some programs use non-zero exit status to communicate results
      - { grep < found / not-found
        test boolean expression
- plus other stuff not important right now

Process Management (cont.)

- Unix solution: separate process creation from program invocation

- fork(): clone current process (the one that called fork)
  - original process: parent
  - new process: child

  - both processes are identical (including open files) except for process id # and return value of fork

  - fork returns int value
    - like all system calls

  - negative value indicates failure
    - like all (?) system calls

  - child process: return 0

  - parent process: pid of child

⟹ tree of processes
process 1 is root
↳ (init)

Process Management (cont.)

- when child process terminates, kernel holds the child's exit status until parent collects it

- parent collects child exit status via wait/wait-pid system call
  - poor choice of name, but we're stuck with it
  - indefinite wait or polling (return immediately)
    - wait() doesn't necessarily wait

- optionally, parent may ask kernel to send signal (software analog of interrupt) when child terminates

- until parent calls wait(), kernel maintains child's entry in kernel's process table
  - all other child resources are freed up
  - child is in zombie state (interview question)

- if parent dies first, child becomes orphan
  - orphan is adopted by process 1 (init)
  - init's other function is to reap orphan zombies

System Calls (Linux)

- design principles:
  - minimal set of orthogonal operations *
  - uniform file interface **

* original (1970s) Unix had dozens of system calls, modern Linux has 100s

** mostly

- in C (API layer), system calls look like ordinary function calls

System Calls (Linux) cont.

- examples
  - process control: fork, exec, wait, signal, kill
  - file management: rename, unlink, chmod, chown
  - file ops: open, close, read, write
  - device ops: open, close, read, write, ioctl
  - networking: bind, listen, connect, send, recv, read, write
  - inter-process communication: signal, kill, popen, shmget, mmap
  - misc: umask, getpid, getppid, getcwd, chdir

System Calls (Linux) cont.

- arguments to / results from system call stored in registers (typically)
  - register: integer / pointer

- return values from system calls are integers
  - negative for failure
  - 0 - ok (if no other meaningful result)
  - positive (or non-negative) value for specific calls
    - e.g. file descriptor (index into table)
      process ID (index into table)

File Descriptors

- file descriptor: small integer representing an open file
  - literally: index into kernel data structure (table - array)
  - handle / token into opaque data structure

- convention:

		formatted buffered	
0	standard input	stdin	cin
1	standard output	stdout	cout
2	standard error	stderr	cerr

- most programs expect files 0, 1, 2 to be pre-opened for them (cin, cout, cerr)

- shell reads command, calls fork
  - child copy replaces files 0, 1, 2 if necessary, then calls exec
  - parent calls wait to get child exit status
    - user can access exit status in shell variable $? (sh/bash)

File Descriptors (cont.)

- C Library
  - fopen() returns FILE* object (handle into library object) — userland
    - internally, calls open(2) which returns file descriptor
  - fileno() returns descriptor for FILE* object
  - fdopen() takes open file descriptor & wraps new FILE* object around it

Pipes

- common paradigm : producer - consumer
  - output of one program is input to another

- program 1 : write file
  program 2 : read file
  } • program 2 has to wait for program 1 to finish
  • inefficient to write to disk

- Solution pipe : write to / read from kernel buffer
  - producer blocks when pipe is full
  - consumer blocks when pipe is empty
  - otherwise, both processes operate concurrently

- ~~pipe()~~ popen system call returns pair of open file descriptors for producer (writer) & consumer (reader)
  - open files (descriptors) stay open across fork/exec
  ⇒ this means pipes only work for descendants of process that opened the pipes