CSS503A   # Lecture #5   2019-04-15

Administrivia

- solution to lab #1

- Canvas

- top-ten (revisited)
  - same as shell script

- labs: artist's study for assignments

- lab 2 posted

Previously on CSS 503A...

- key kernel function: managing processes
  - using fork, exec, close, open, pipe, wait, we know enough to write a rudimentary shell
  - userland functions to determine programs to run, system calls to obtain desired behavior

- open files stay open across fork/exec * (unless you ask kernel not to)

- everything is a file
  - programs are simpler, more flexible if they read stdin, write to stdout
    → design principle: separate mechanism & policy
  - simpler if your program does not have to do anything different if reading from keyboard vs reading from file

- shell syntax to specify fd 0, 1, 2

- dataflow pipelines: simple "filter" programs can be combined to perform complex operations
  - similar to functional style $f(g(h(x)))$
    or collection.map{ }.reduce{ }.filter{ }...
    ↳ fold

Previously ... (cont.)

- executable file:
  - passive data (structured binary data)
  - can be manipulated by any program that can read/write files (i.e. ~~all~~ many programs)
- system loader (exec): loads "data" (your program) into process memory
  - text (code)
  - data
  - space for dynamic memory allocation (new/malloc) → heap
  - call stack
- running state of program includes
  - registers (esp. PC, SP)

Processes (cont.)

- running program
  - registers (esp. program counter, stack pointer)
  - text (code)
  - static data (initialized / uninitialized) → misnomer: initialized to Ø
  - stack ("call stack")
  - heap (arena) - dynamic memory allocation
  - open files (file descriptors)
  - argv / envp
  - exit status (on program termination)

Previously... (cont.)

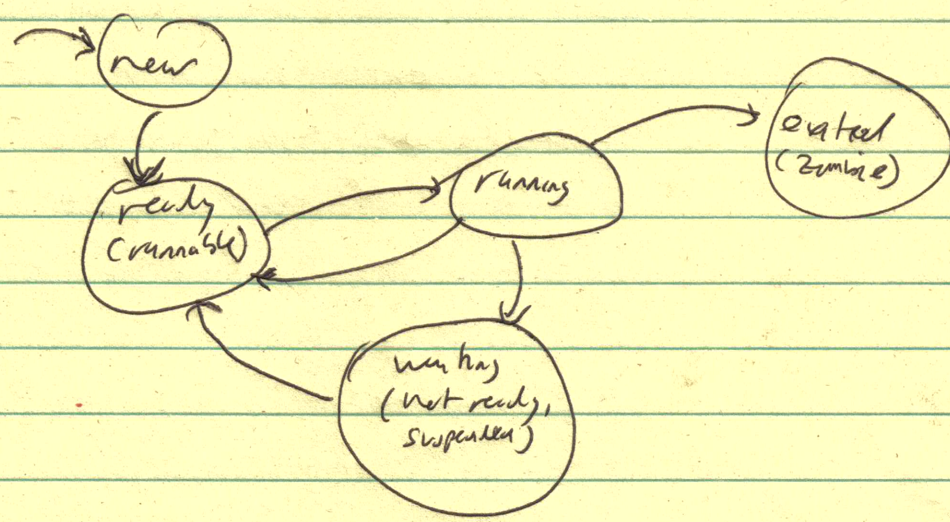
- Kernel view of process: data structure

Process Control Block

- register save space
- memory map
- table of open files
- pid, ppid, uid, gid, nice, ...
- signal vector

Previously ... (cont.)

- process state: running, not running
  → more complicated



- system scheduler picks ready process & runs it until system call, interrupt, timer (time slice)

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

Previously ... (cont.)

- Signals: software analog of hardware interrupts
  - H/W interrupt goes to kernel
  - software interrupt: kernel manipulates userland state (data & registers)

- Signal number: small (system-defined integer)
  - each symbol has semantics & default action
  - user makes system call to override (or restore) default behavior
    - signal 9 cannot be caught

- trap/catch signal to perform cleanup action before program termination
  - grace period between SIGINT & SIGTERM

- signals to notify process
  - I/O is available (async I/O)
  - child state change (call wait to reap zombie)

- user-defined signals: SIGUSR1, SIGUSR2

- kill(2): send signal [kill(1) is then wrapper]
  - poor choice of names
  - many signals' default behavior

Previously... (cont.)

- cooperating processes
  - simpler, more modular/flexible designs
  - latest incarnation: microservices
    - we keep "rediscovering" basic principles

- cooperating processes need to communicate

- argv / exit status
  - only exchange information at program startup/termination
  - limited bandwidth, but useful
  - shell uses exit status as Boolean value (loops & conditionals)

- signals (wake up and do something!)

- files: awkward & inefficient (clunky)
  - but it's been done
  - system calls to support this

Previously (cont.)

- pipes
  - file-like: use read/write system calls
  - read/write: copy data from/to kernel buffer
  - one-way communication (producer–consumer)
    - extremely common paradigm
      (but doesn't cover all cases)
  - must be created by common ancestor process (usually: shell)

- named pipes
  - create "special" entry entry in filesystem
  - open by name (like any other file on filesystem)
    - use read/write system calls
  - works just like regular pipe, except processes do not require common ancestor
  - more awkward than regular pipes for common use cases
  - probably used today only for legacy applications & work-arounds
    → only time I used a named pipe was back in the '90s and it was exactly that: workaround

Previously... (cont.)

- 2 basic paradigms for communication
  - message passing (aka "shared nothing")
    - everything we've seen so far (esp. pipes)
  - shared memory (stay tuned)

Message Passing

- safer than shared memory
- higher-level than mutexes

- Send / receive
  - Go-language: channels
  - Erlang: mail boxes
  - CAR Hoare: Communicating Sequential processes

- Pipes: unstructured
  - pass sequence of bytes
  - requires application-level structuring & synchronization

- issues: buffering, synchronization

- blocking vs. non-blocking send

Interprocess Communication (cont.)

- Posix IPC
  - same goals as SysV IPC
  - specifies API only
    - may be implemented as userland library that translates operations into native system calls
  - goal is portability of programs

IPC (cont.)

- Message passing (cont.)
  - unix-domain sockets
    - like pipes but uses networking infrastructure
    - on same machine only
    - different protocol from TCP/IP & UDP/IP
  - TCP/IP & UDP/IP
    - networking: can communicate with processes on other machines
    - loopback interface: use networking to talk to self
    - separate topic (rich)
    - "cluster OS" – ~~treat~~ entire datacenter as system that requires resource management / abstractions

Interprocess Communication (cont.)

- System V IPC (original AT&T Unix, version 5)
  - because pipes are not always the most convenient/efficient mechanism
  - developed ~ early 1980s
  - predates Posix IPC, same basic goals
  - predates rise of threading
    - which is the next topic

- Message queue
  - like structured pipe: can handle higher-level protocol ↑ Protocol
  - guarantees: no partial reads

- Semaphores
  - protect against concurrent updates
  - more later

- Shared memory

System V Message Queues

```
int msgid = msgget(key_t key, int msgflg)

int msgsnd(int msgid,
           const void* — (struct msgbuf) * msgp,
           size_t — (int) msgsz
           int msgflg)

struct msgbuf {
    long mtype,
    char mtext[]
}
```

C-style casting wizardry: superimpose msgbuf onto your own data structure

mtype: use pattern matching on receive

System V Message Queues (cont.)

```
ssize_t msgrcv (
        int msgid,
        void * msgp,      ← pointer to user allocated space
        size_t msgsz,
        long msgtype,     ← pattern matching:
        int msgflg );       select message with
                            this mtype
```

System V Semaphores

- semaphore (in real life): flag
  - flag up: resource available
  - flag down: wait for resource

} mutex

- semaphore: more complicated
  - Counter
    - decrement counter, get token
    - increment counter, release token
    - counter = 0 (no tokens available): wait

```
int semid = semget( key_t key,
                    int nsems,
                    int semflag)
```

- create / open semaphore set

System V Semaphores (cont.)

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

struct sembuf {
    unsigned short sem_num;
    short  sem_op
    short  sem_flag)
```

sem_op :

- \>0 increment (always successful)
- <0 decrement or wait
- =0 wait for zero

sem_flag : IPC_NOWAIT

- return (fail) immediately

* race condition → atomic operation

Shared Memory

- processes must explicitly request this
  - via system calls

- handled by kernel memory management
  - memory map: data structure

- 2 virtual memory segments / pages mapped to the same real memory (frames)
  - kernel's doing this mapping anyway
  - about the fastest you can do IPC
    - no double buffering

- kernel data structure shmid_ds
  - permissions } like file
  - timestamps }
  - size (# pages)
  - ptrs to frames (array of pointers)
  - etc

Shared Memory (cont.)

```
#include <sys/ipc.h>
#include <sys/shm.h>
```

- create/access shared memory

```
int shmid = shmget (
        key_t key,
        size_t size,
        int shmflg);
```

- shmid : small nonnegative integer - like file descriptor
    - -1 on failure

- key_t : typedef for long
    - "identifier" - like file name
        - must be shared across all processes wishing to share that piece of memory

- ftok(3) : take pathname (& proj_id)
    → convert to key_t
    (hash ?)

Shared Memory (cont.)

- shmflg: bit-mapped flags
  - symbolic constants, or'ed together
  - owner/group/other permissions: like file
  - IPC_CREAT | IPC_EXCL
    - create if it doesn't exist & fail if it does

- So, we have a shmid (descriptor), now what?
  - shmctl : get/set flags
  - shmat } attach/detach
    shmdt }

void* shmat(int shmid,
            const void* shmaddr,
            int shmflg)

- Better: shmaddr → null
  - more portable
  - let system decide where to put it
- otherwise: shmaddr must be page-aligned

e.g. flag SHM_RDONLY

Read-only segment: consumer in producer-consumer arrangement (or subscriber in Pub-Sub model?)

Shared memory (cont.)

int shmdt(const void* shmaddr)
- shmaddr: returned from shmat

* now you have a (void *) pointer
  - can use like any other pointer
  - except: be careful you don't scribble over the memory at the same time another process is scribbling over it

* need synchronization mechanism to avoid clobbering data

Shared Memory (cont.)

- Usage:

  application requests pointer to block of n bytes

  → application must cast memory into appropriate data structures

  - applications must synchronize concurrent access

    e.g. via semaphore / mutex

- most common use case: same program

  → why not just use threads

- JVM

  - multiple programs in single VM session

  → multithreading

- Sample code on slide (bad example)

  - spin lock (busy wait)
  - assumes count ++/ count -- is atomic (we'll revisit this)
  -

Shared libraries

- mmap() : map file to memory
  - can use in lieu of read write
  - kernel still needs to manage block I/O
- shared libraries : .so
  - all programs do I/O
    - stdlib : buffered & formatted I/O
- dynamic linking
  - link at load time
    - that's why linkers & loaders is often conflated

Threads

→ main advantage of processes
→ won't clobber other process

- processes are (protected) against stomping on / clobbering each other
  - separate memory spaces
  - hardware controls set up by kernel
- kernel has total access to process's address space
  - context switch: kernel saves state of registers
- have you ever written a (C++) program with a wild pointer?
  - imagine the debugging problem if your wild pointer scribbled all over someone else's process — or vice versa

Threads (cont.)

- more "light weight" vs process
  - more responsive
  - lower resource consumption
  - faster context switching (between threads)
    - just reload the registers
  - useful for scaling algorithms
    - better solution to assignment I
      ⇒ see my Go solution
  - deadlock avoidance
    - YMMV (slides show 2 pipes for bidirectional communication)
  ⇒ IME deadlocks / race conditions
    <u>more</u> likely

- process: single executable program
  ⇒ all threads in process running same program
    - at different execution points

Threads

- similar to processes (but different)

- concurrent execution paths within a single process
  - common memory / resources (e.g. open files)
  - each thread has its own stack (local variables)
    - smaller than single-threaded system stack
  - data structure to manage threads (e.g. save registers)

- lightweight vs. process
  - more responsive
  - deadlock avoidance
    - ymmv
  - scalability (parallelization of algorithms)
  - lower resource consumption
  - faster context switching

- process is a single executable program
  i.e. all threads are executing same program
  - at different execution points

- interprocess (interthread) communication:
  - both easier & harder
  → shared memory

Threads (cont.)

- the downsides of process
  - fairly expensive to set up (machine resources)
  - interprocess communication is klunky

- solution: abandon what makes process great & have multiple "threads of control" within single process
  - concurrent execution paths within single process

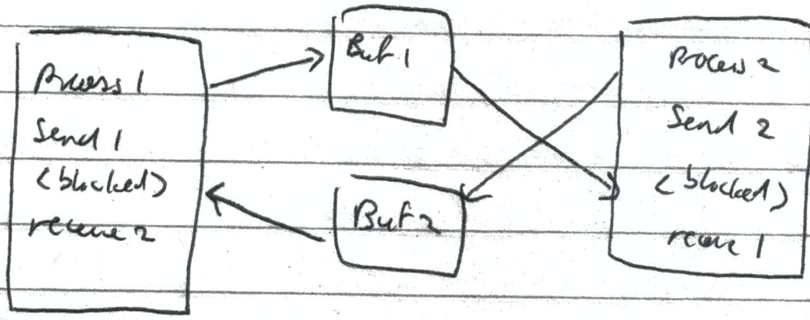
- common memory/resources
  - each thread has its own (call) stack
    - smaller than single-threaded system stack
  - data structure to manage threads (e.g. save registers)

- may be implemented in
  - kernel (system support)
  - userland (library support)
  - both

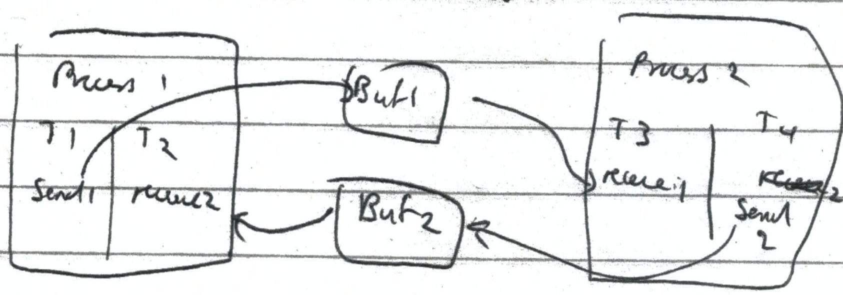
)

Deadlock

- bounded buffer (e.g. pipe)



- avoid deadlock: process 1 & 2 are multithreaded



- IMHO: example looks a bit farfetched

green threads

User Threads vs Kernel Threads

- can implement threads via
  - kernel
    - e.g. linux clone(2)
      - extended kernel process data structure
  - user library
    - Java
    - pthreads (Posix)
    - Erlang (processes : just to be confusing)
    - go (go routines)

- user library: may or may not have kernel support
  - full support: 1-1 mapping between library threads & kernel threads
    - library may be more convenient / more portable
      - e.g. pthreads
  - no support: library has total access to its own address space (it's part of the process)
    - create application-level process control block
    - handle scheduling, etc
      - non-preemptive (cooperative): yield
      - sigalrm (timer)
    - textbook calls this Many : 1

- hybrid model: multiplexing (many : many)

Threads (cont.)

- communication between threads (within process):
  - both easier & harder
  - → shared memory
    (must take care to avoid race conditions & deadlocks)

- message passing is becoming more popular
  - Erlang / Elixir
  - Go - Channels
  - Scala / Akka
  - → functional style
    - → immutable data

} easier to reason about

Threading Issues

- interaction with fork() / exec()

- thread cancellation:
    - what happens to shared resources (e.g. locks)?
    - async / deferred

- signal handling
    - synchronous (e.g. illegal memory access)
        - received by same thread
    - asynchronous (e.g. SIGINT - control - C)
        - which thread?

Thread Pools

{ relatively expensive to create threads: pay overhead cost once

- create n threads at process startup
  → "worker pool"
  - send unit-of-work to worker

Java Threads

- method 1:
  - subclass Thread
  - override run() method
- method 2:
  - implement Runnable interface

```
public interface Runnable {
    public abstract void run()
}
```

  - Thread t = new Thread(new RunnableImpl())
    t.start()
    t.join()
    - wrap join in try/catch block in case InterruptedException

- low-overhead threads (e.g. Go, Erlang):
  no need for worker pools
    - just create as many threads as needed, use other mechanism for rate limiting (e.g. Go buffered channels)

- other abstractions for concurrency
    - futures / promises
    - parallel container classes

Posix Threads (pthreads)

```
#include <pthread.h>
#include <unistd.h>
```

- user-defined thread function

  `void * thread_func (void * param);`

  - takes pointer arg
  - returns pointer value

  } - most general API - user-defined data
  - requires typecasting (not type-safe)

- int pthread_create (

```
    pthread_t * thread,
    const pthread_attr_t * attr,
    void * (* start_routine) (void *),
    void * arg )
```

Pthreads (cont.)

- thread termination
  - thread calls pthread_exit()
    - exit status passed to pthread_join()
  - start_routine() returns
  - pthread_cancel()
  - exit()

- thread may be joinable or detached
  - pthread_join(): like wait(2)

- pthread attributes