CSL373: Operating Systems
Lecture 2: threads & processes

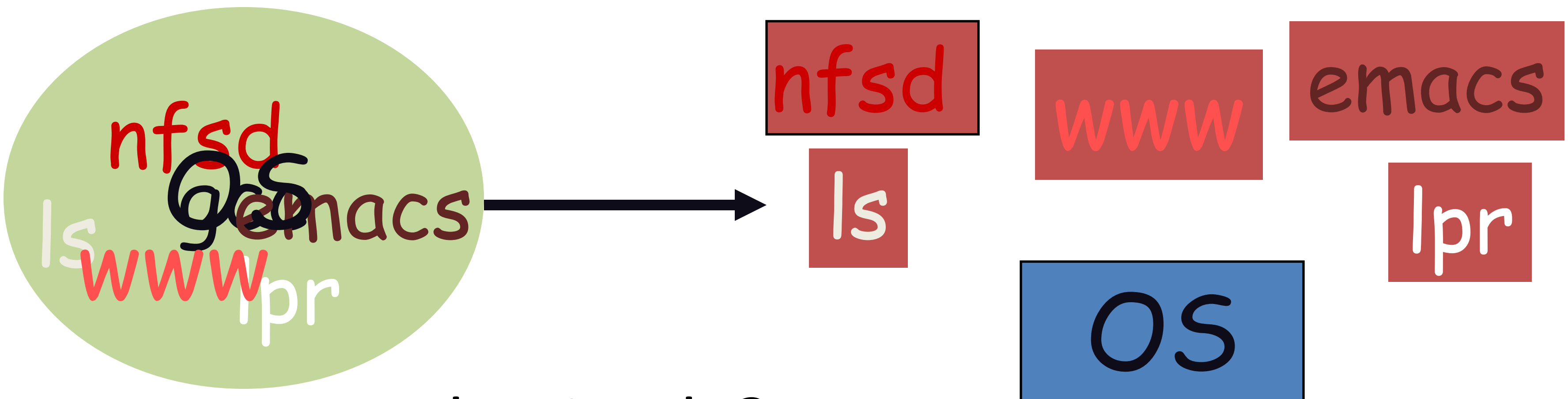
Sorav Bansal

Today’s big adventure

- What are processes, threads?
- What are they for?
- How do they work?
- Threads vs processes?
- Readings: Silberschatz/Galvin: Ch 4 (skip 4.6)

Why processes? Simplicity

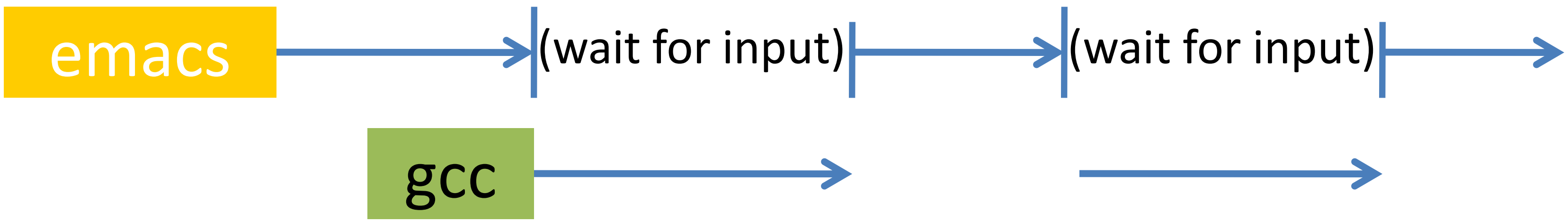
- Hundreds of things going on in the system



- How to make simple?
  - Separate each in isolated process. OS deals with one thing at a time, they just deal with OS
  - *THE* universal trick for managing complexity: decomposition (“reductionism”)

Why processes? Speed

- I/O parallelism:

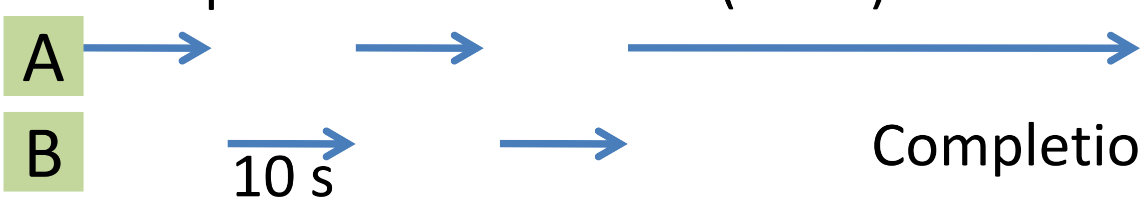


overlap execution: make 1 CPU into many

(Real parallelism: > 1 CPU (multiprocessing))

- Completion time:

B’s completion time = 100s (A + B). So overlap



Completion time for B? A?

Processes in the real world

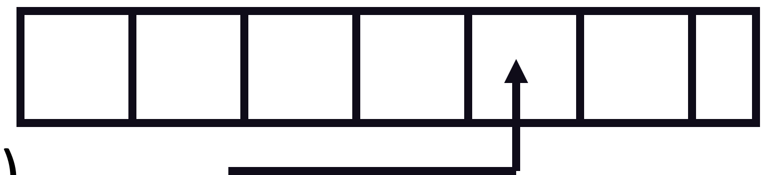
- Processes, parallelism fact of life much longer than OSes have been around
  - Companies use parallelism for more throughput: 1 worker = 100 widgets? Hire 100 to make 10,000.

- Can you always partition work to speed up job?
  - Ideal: N-fold speedup
  - Reality: bottlenecks + coordination overhead
  - Example: Will class size=1000 work? Or will project group size=30 work? (Similar problem in programs.)

(More abstractly: easy to increase throughput, reducing latency more difficult)

What is a thread?

- In theory: turing machine
  - tape(state), tape head(position)



- In practice: What’s needed to run code on CPU
  - “execution stream in an execution context”
  - Execution stream: sequential sequence of instructions
- CPU execution context (1 thread)
  - state: stack, heap, registers
  - position: program counter register

```
add r1, r2, r3
sub r2, r3, r10
st r2, 0(r1)
...
```

- OS execution context (n threads):
  - Identity + open file descriptors, page table, …

What is a process?

- Process: thread + address space
  - or, abstraction representing what you need to run thread on OS (open files, etc)
- Address space: encapsulates protection
  - Address state passive, threads active
- Why separate thread, process?
  - Many situations where you want multiple threads per address space (servers, OS, parallel program)

Process != Program

- Program: code + data

  passive

```
int a;
int main() {
    printf("hello");
}
```

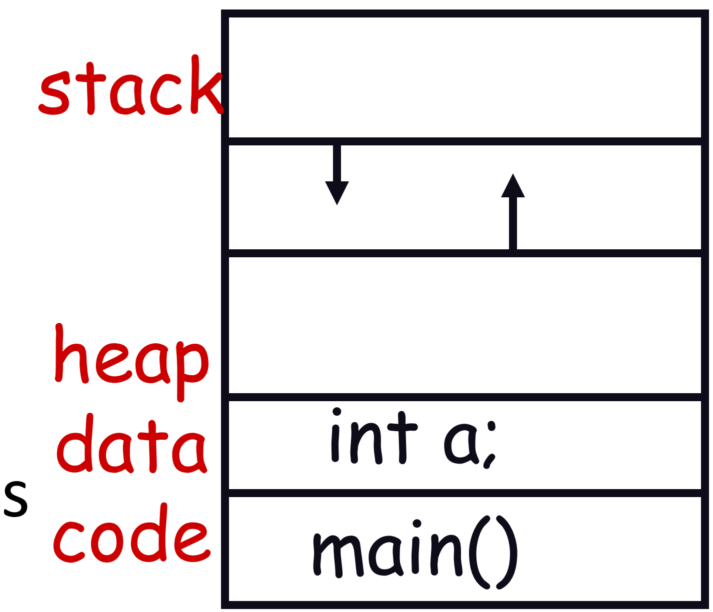
- Process: running program

  state: registers, stack, heap…

  position: program counter

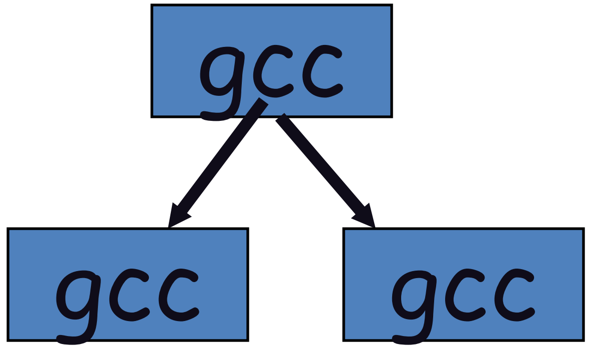
- We both run netscape:

  Same program, different process



How to make one?

- Creation:
  - Load code and data into memory; create empty call stack
  - Initialize state to same as after a process switch
  - Put on OS's list of processes
- Clone:
  - Stop current process and save state
  - Make copy of current code, data, stack and OS state
  - Add new process to OS's list of processes



Example: Unix

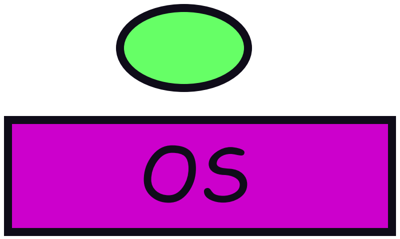
- How to make processes:
  - fork() clones a process
  - exec() overlays the current process
  - No create! Fork then exec.

```
if ((pid = fork()) == 0) {
        /* child process */
        exec(“foo”);    /* exec does not return */
} else {
       /* parent */
       wait(pid); /* wait for child to finish */
}
```

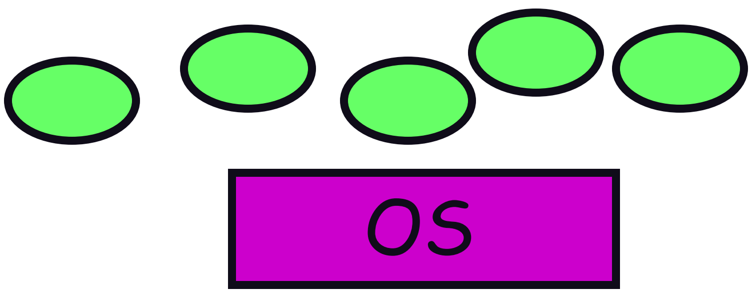
- Pros: Simple, clean. Con: duplicate operations
- Note: fork() and exec() are “system calls”
    - system calls = functions implemented by the OS and exposed to the application)
    - Look just like a normal procedure call, but implemented differently. Other examples: open(), read(), write(), …

Process environments

- Uniprogramming: 1 process at a time
  “Cooperative timesharing”: vintage OSes
  Easy for OS, hard for user (generally)
  Violates isolation: Infinite loops? When should process yield?



- Multiprogramming: >1 process at a time
  Time-sharing: CTSS, Multics, Unix, VMS, NT

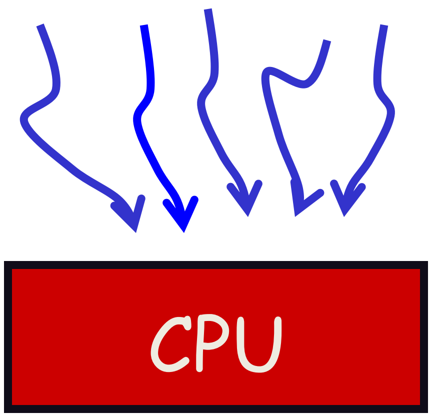


multiprogramming != multiprocessing

The multithreading illusion

- Each thread has its illusion of own CPU
  - yet on a uni-processor, all threads share the same physical CPU!

How does this work?



- Two key pieces:
  - thread control block: one per thread, holds execution state
  - dispatching loop:

```
while(1)
    interrupt thread
    save state
    get next thread
    load state, jump to it
```

The multiprogramming problems

- Track state? PCB (process control block)
  - Thread state, plus OS state: identify, accounting, …

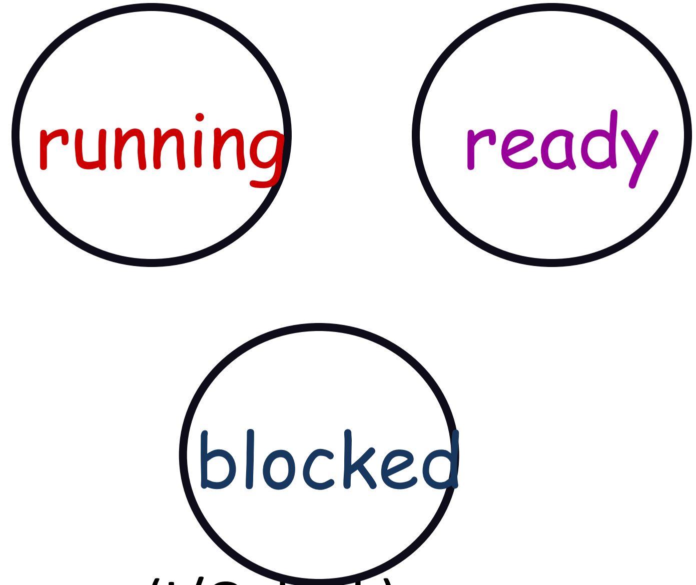
pcb	Priority	registers	open file descriptors, …

- N processes? Who to run? ("Scheduling")

  Need to schedule whenever 1 resource and many requestors (disk, net, CPU, classroom, …)
- Protection? Need two things
  - Prevent process from getting at another's state
  - Fairness: make sure each process gets to run
  - No protection? System crashes ~ O(# of processes)

Process states

- Processes in three states



  - Running: executing now
  - Ready: waiting for CPU
  - Blocked: waiting for another event (I/O, lock)

- Which ready process to pick?

  0 ready processes: run idle loop

  1 ready process: easy!

  >1: what to do?

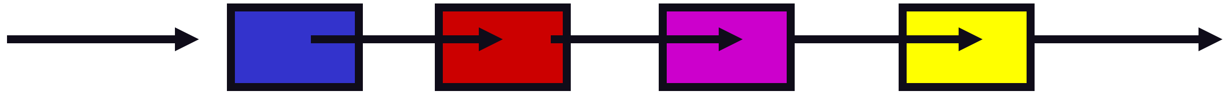
Picking a process to run

- Scan process table for first runnable?

  Expensive. Weird priorities (small pid's better)

  Divide into runnable and blocked processes

- FIFO?
  - Put threads on back of list, pull them off from front



  (pintos does this)

- Priority?

  give some threads a better shot at the CPU problem?

  (you are required to implement this in Assignment 1)

Scheduling policies

- Scheduling issues
  fairness: don't starve process
  prioritize: more important first
  deadlines: must do by time 'x' (car brakes)
  optimization: some schedules >> faster than others

- No universal policy:
  Many variables, can't maximize them all
  conflicting goals
    - more important jobs vs starving others
    - I want my job to run first, you want yours.

- Given some policy, how to get control? Switch?

How to get control?

- Traps: events generated by current process
    - System calls
    - Errors (illegal instructions)
    - Page faults
- Interrupts: events external to the process
    - I/O interrupt
    - Timer interrupt (every 100 milliseconds or so)
- Process perspective
    - Explicit: process yields processor to another
    - Implicit: causes an expensive blocking event, gets switched

How to "context switch"?

- Very machine dependent. Must save:
  - general-purpose & floating point registers, any co-processor state, shadow registers (Alpha, sparc)
- Tricky:
  - OS code must save state without changing any state
  - How to run without touching any registers??
    - Some CISC machines have single instruction to save all registers on stack
    - RISC: reserve registers for kernel (MIPS) or have way to carefully save one and then continue
- How expensive? Direct cost of saving; opportunity cost of flushing useful caches (cache, TLB, etc.)

Fundamentals of process switching

- "execution" *THE* grand theme of CS:
  procedure calls, threads, processes just variations

- What's the minimum to execute code?
  - Position (pointer to current instruction)
  - State (captures result of computation)

- Minimum to switch from one to another?
  - Save old instruction pointer and load new one

- What about state?
  - If per-thread state, have to save and restore
  - In practice, can save everything, nothing or combination.

Switching between procedures

- Procedure call:

  save active caller registers

  call foo

  saves used callee registers

  ...do stuff...

  restores callee registers

  jumps back to pc

  restore caller regs

- How is state saved?

  saved proactively? saved lazily? not saved?

Threads vs procedures

- threads may resume out of order
  - cannot use LIFO stack to save state
  - general solution: duplicate stack
- threads switch less often
  - don’t partition registers (why?)
- threads involuntarily interrupted:
  - synchronous: proc call can use compiler to save state
  - asynchronous: thread switch code saves all registers
- more than one thread can run
  - scheduling: what to overlay on CPU next?
  - proc call scheduling obvious: run called procedure.

~Synchronous thread switching

```
# called by scheduler:  a0 holds ptr to old thread blk,
#  a1 ptr to new thread blk
cswitch:
        add     sp, sp, -128
        st      s0, 0(sp)  # save callee registers
        st      s1, 4(sp)
        …
        st      ra, 124(sp)    # save return addr
        st      sp,  0(a0)     # save stack
        ld      sp,  0(a1)     # load up in reverse
        ld      s0, 0(sp)
        …
        add     sp, sp, 128
        j       ra
```

~Asynch thread switching

Assume ~MIPS, k0 = reserved reg

```
# save current state:
# triggered by interrupt
save_state:
   add sp, sp, -128
   st s0, 0(sp)  # save callee regs
    ...
   st  t0,  64(sp)  # save caller regs
    ...
   st  epc,  132(sp) # interrupt pc
   ld  k0, current_thread
   st  sp, 0(k0)
   ld  sp,  scheduler_stack
   j  scheduler
```

```
# restore current state
# called by scheduler
restore_state:
        ld k0, current_thread
        ld sp, 0(k0)
        ld s0, 0(sp)
          ...
        ld t0, 64(sp)
        ...
        add sp, sp, 128
        ld k0, 132(sp) # old pc
        j k0
```

Process vs threads

- Different address space:
  switch page table, etc.
  Problems: How to share data? How to communicate?

- Different process have different privileges:
  switch OS's idea of who's running

- Protection:
  have to save state in safe place (OS)
  need support to forcibly revoke processor
  Prevent imposters

- Different than procedures?
  OS, not compiler, manages state saving

Real OS permutations

- One or many address spaces
- One or many threads per address space

# of address spaces	1	many
# of threads/space 1	MS/DOS Macintosh	Traditional UNIX
many	Embedded systems, Pilot	VMS, Mach, OS/2, Win/NT, Solaris, HP-UX, Linux

Generic abstraction template

- Abstraction: how OS abstracts underlying resource
- Virtualization: how OS makes small number of resources seem like an "infinite" number
- Partitioning: how OS divides resource
- Protection: how OS prevents bad people from using pieces they shouldn't
- Sharing: how different instances are shared
- Speed: how OS reduces management overhead

How CPU abstracted

- CPU state represented as process
- Virtualization: processes interleaved transparently (run ~1/n slower than real CPU)
- Partitioning: CPU shared across time
- Protection: (1)pigs: forcibly interrupted; (2) corruption: process' state saved in OS; (3) imposter: cannot assume another's identity
- Sharing: yield your CPU time slice to another process
- Speed: (1) large scheduling quanta; (2) minimize state needed to switch; (3) share common state (code); (4) duplicate state lazily

Summary

- Thread = pointer to instruction + state
- Process = thread + address space
- Key aspects:
    - Per-thread state
    - Picking a thread to run
    - Switching between threads

- The future:
    – How to share state among threads?