CS 423
Operating System Design: Midterm Review

Tianyin Xu

Overview: OS Stack

OS Runs on Multiple Platforms while presenting the same Interface:

Application Software

- Web Server
- Browser
- Slack
- Pop Mail

Standard Operating System Interface

Operating System (machine independent part)

- Read/Write
- Standard Output
- Device Control
- File System
- Communication

Machine specific part

Hardware Abstraction Layer

Hardware

Network

Overview: OS Roles

Role #1: Referee

- Manage resource allocation between users and applications
- Isolate different users and applications from one another
- Facilitate and mediate communication between different users and applications

Role #2: Illusionist

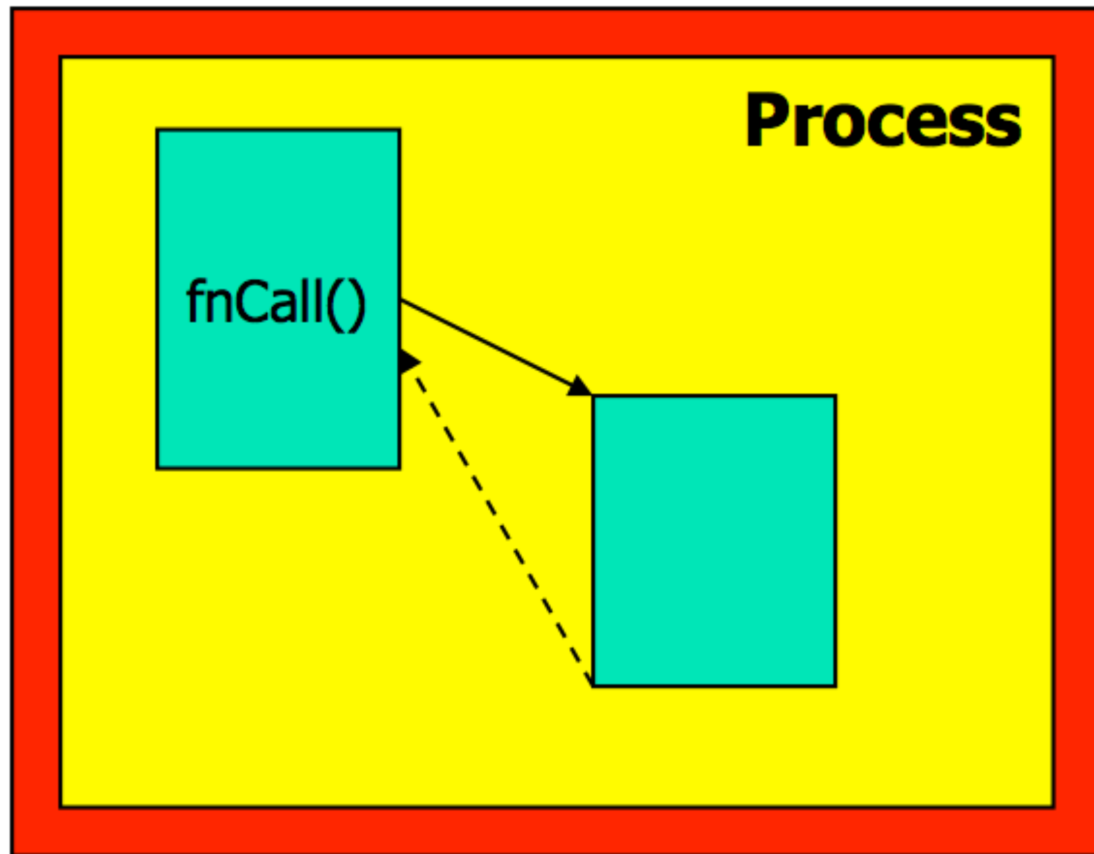
- Allow each application to believe it has the entire machine to itself
- Create the appearance of an Infinite number of processors, (near) infinite memory
- Abstract away complexity of reliability, storage, network communication…

Role #3: Glue

- Manage hardware so applications can be machine-agnostic
- Provide a set of common services that facilitate sharing among applications
- **Examples of “Glue” OS Services?**

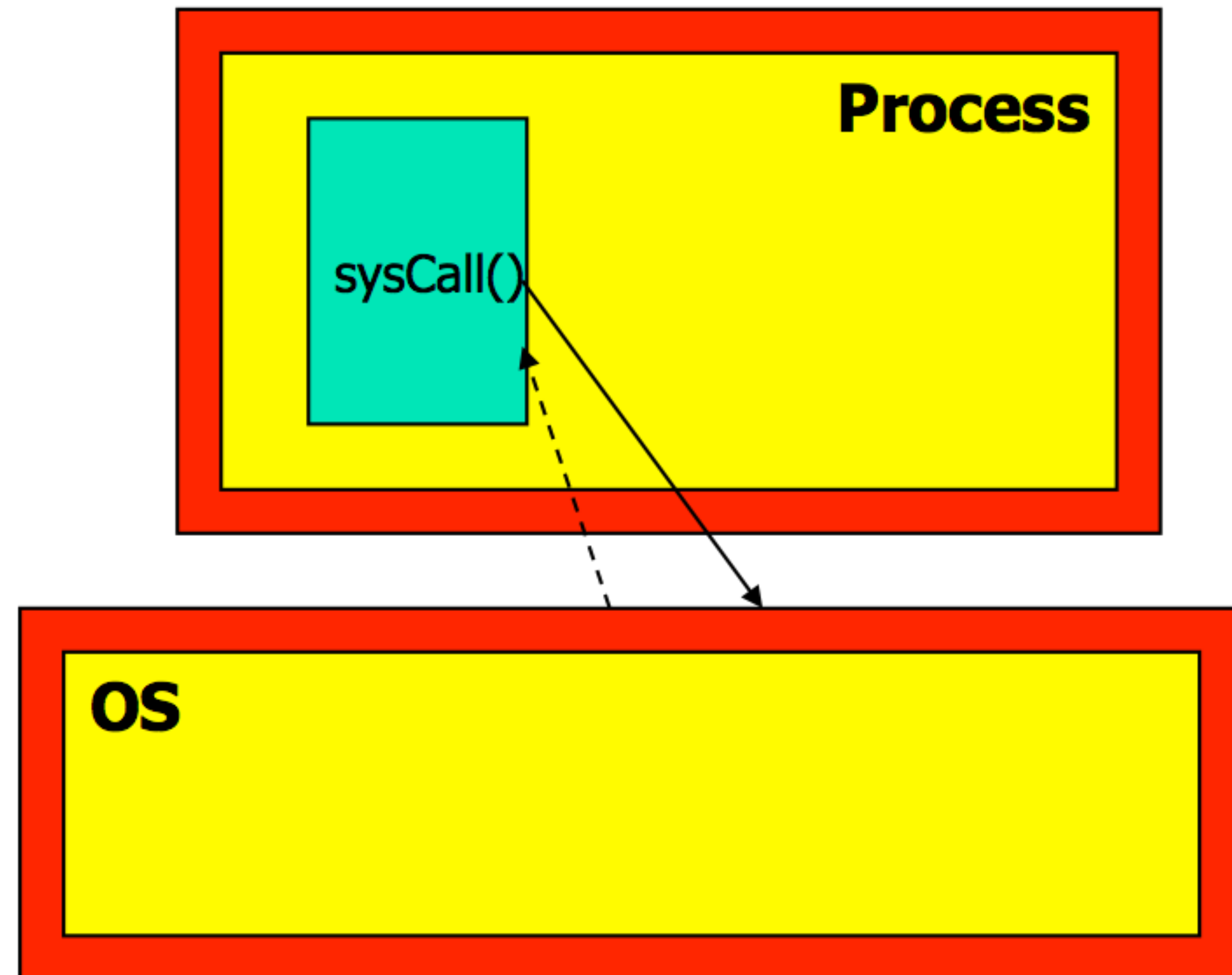
Review: System Calls

Function Calls



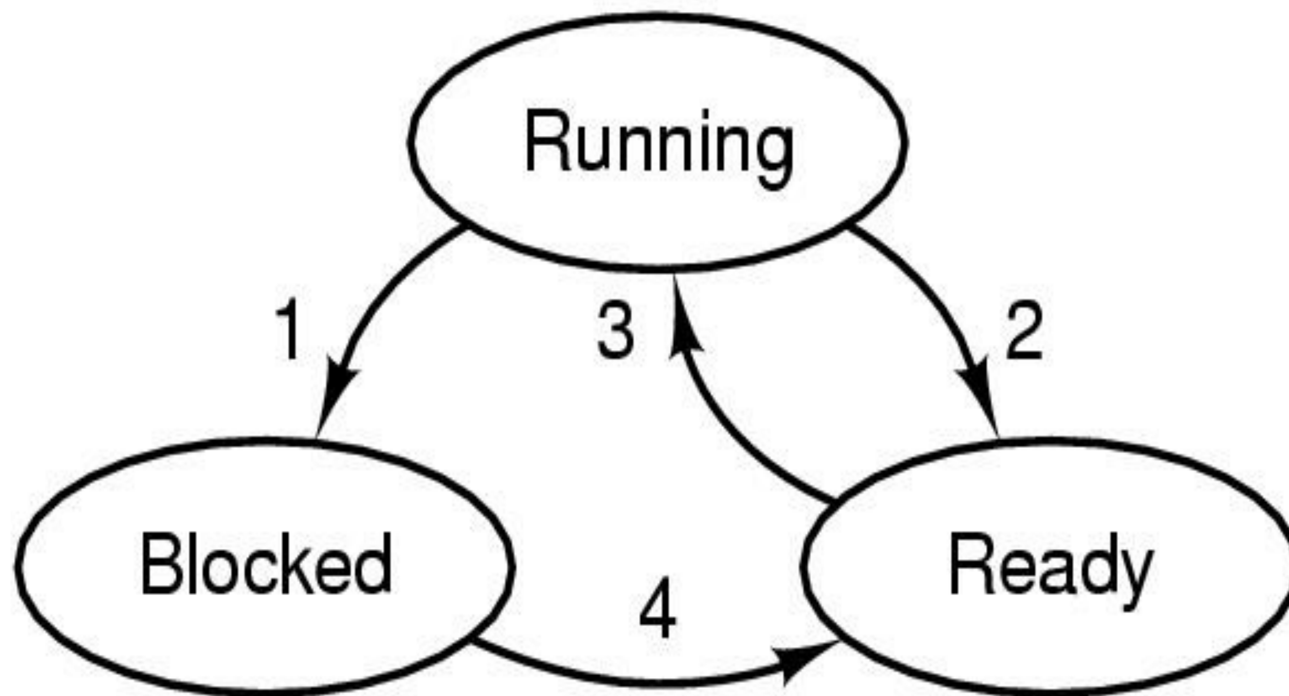
Caller and callee are in the same Process
- Same user
- Same "domain of trust"

System Calls



- OS is trusted; user is not.
- OS has super-privileges; user does not
- Must take measures to prevent abuse

Review: Process Abstraction



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- Possible process states
  - Running (occupy CPU)
  - Blocked
  - Ready (does not occupy CPU)
  - Other states: suspended, terminated

Question: in a single processor machine, how many process can be in running state?

Review: Threads

- (a) Three processes each with one thread
- (b) One process with three threads

Kernel Abstraction: HW Support

Branch Address

+4

Handler PC

Select PC

New PC

Program Counter

CPU Instructions Fetch and Execute

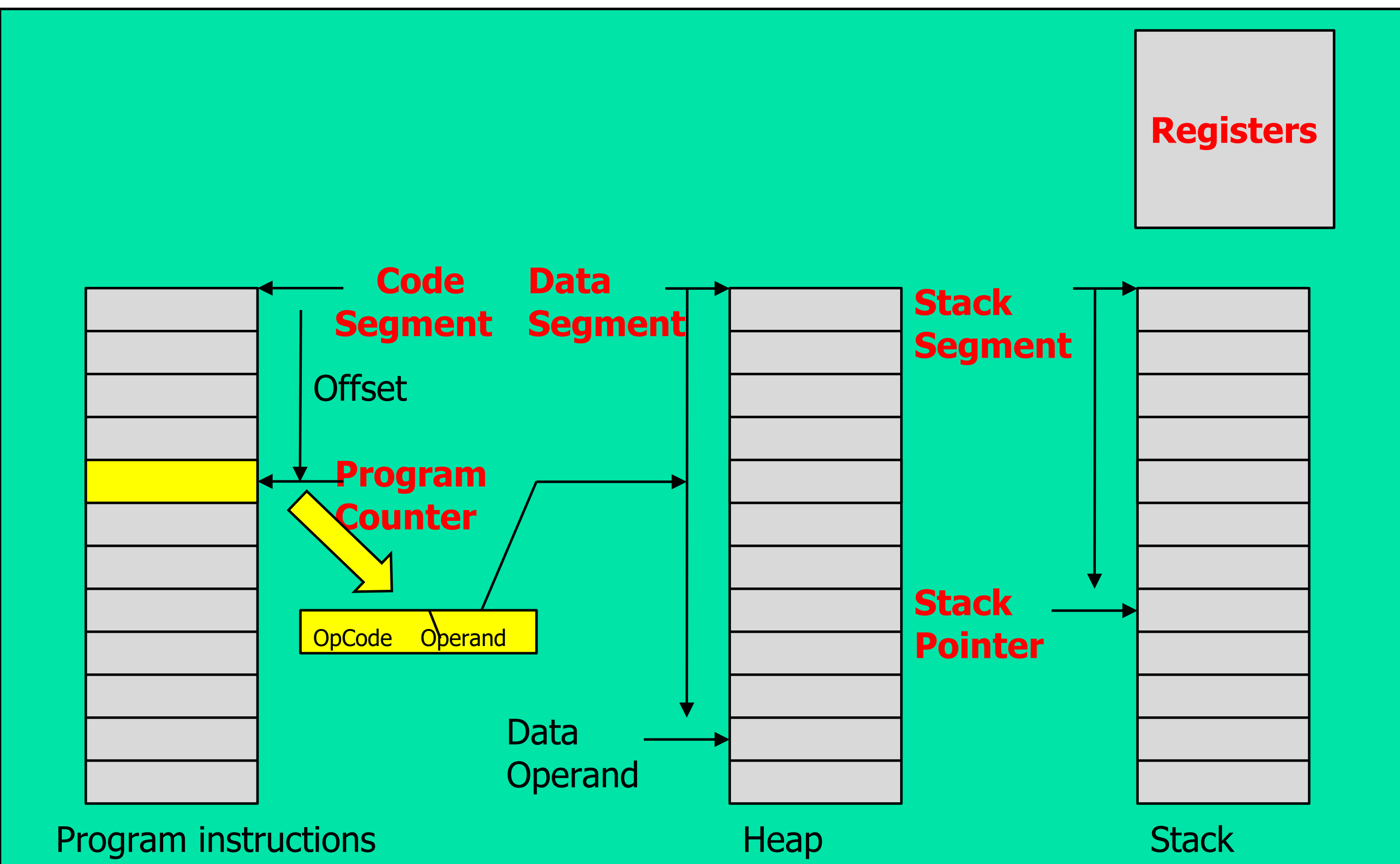
Select Mode

New Mode

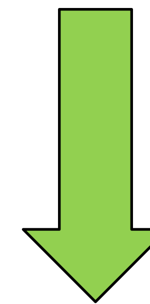
Mode

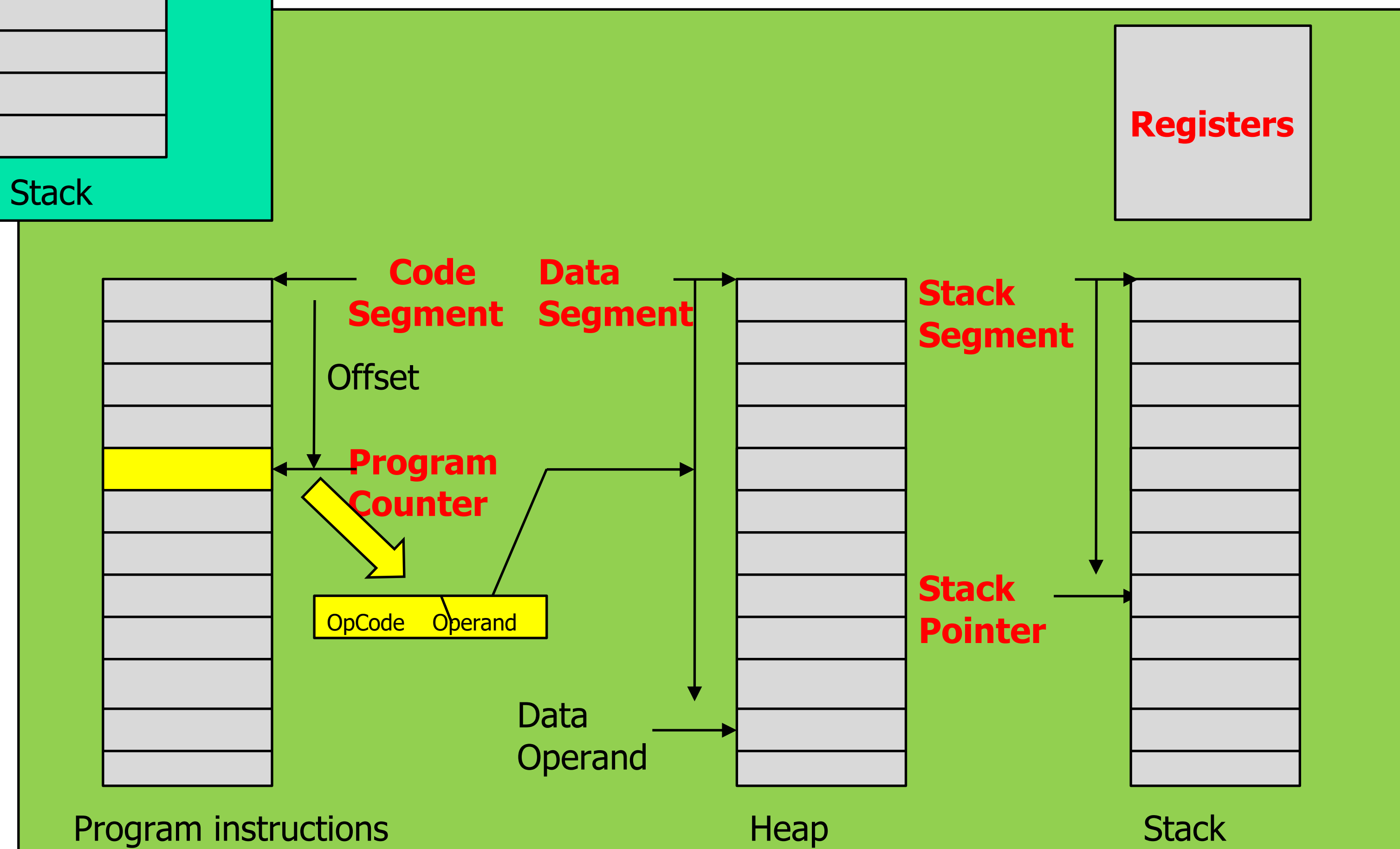
opcode

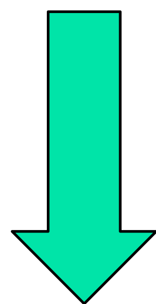
Kernel Abstraction: CTX Switch



Load State (Context)



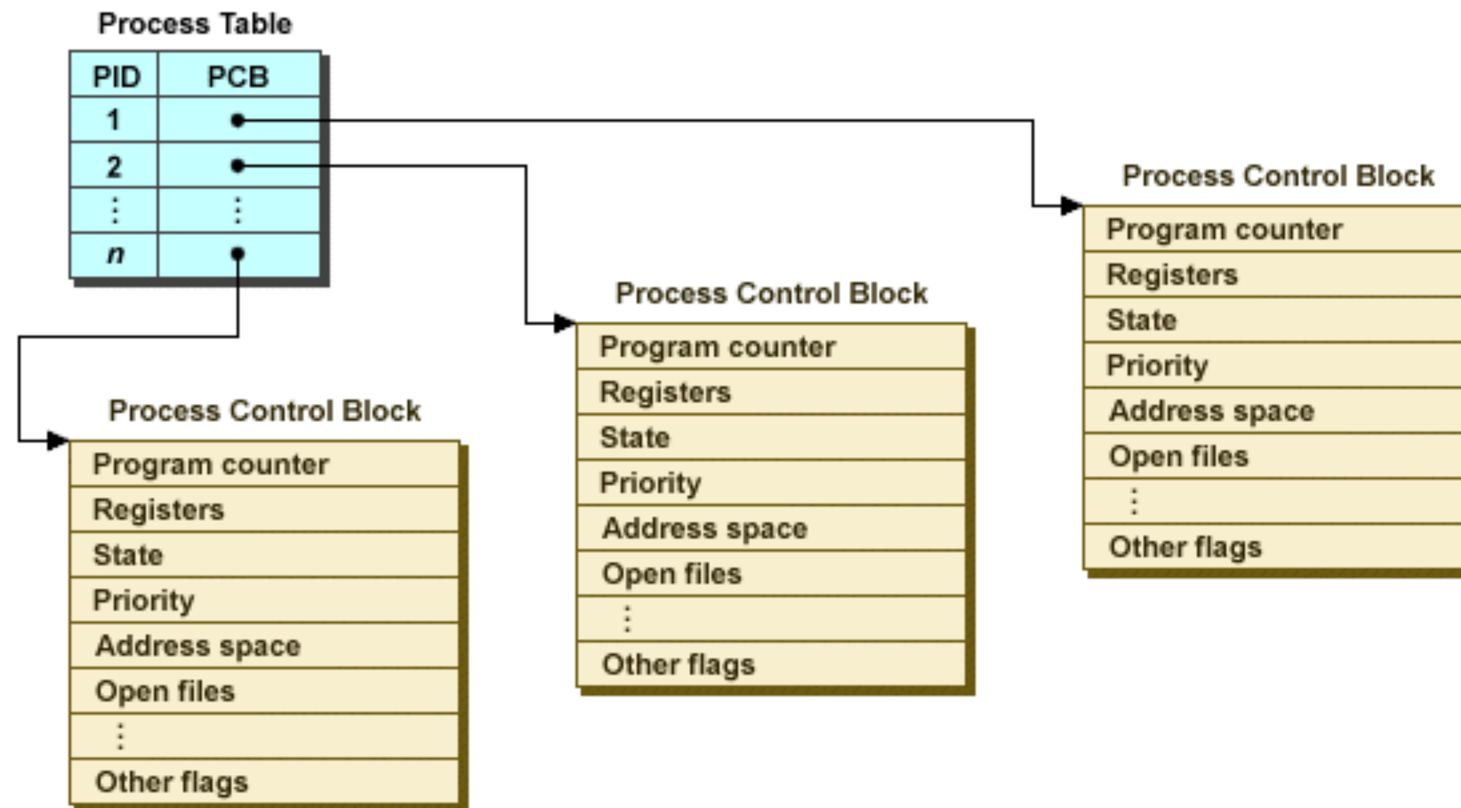




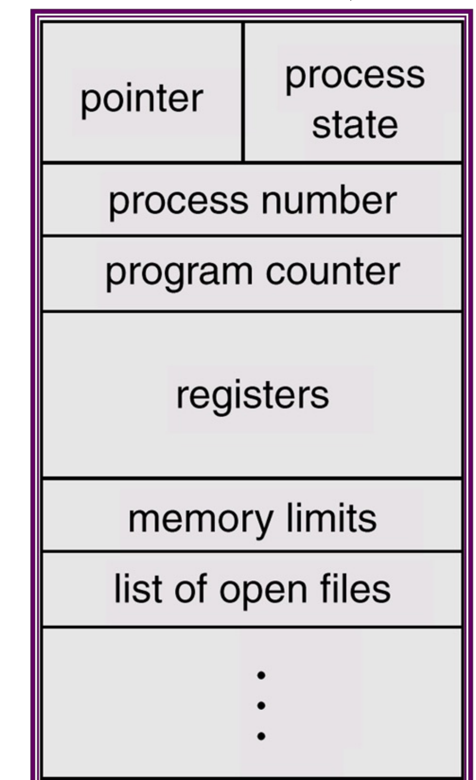
Save State (Context)

Kernel Abstraction: PCBs

The state for processes that are not running on the CPU are maintained in the Process Control Block (PCB) data structure



Updated during context switch



An alternate PCB diagram

Interrupts: Model

Interrupts to drive scheduling decisions!

Interrupt handlers are also tasks that share the CPU.

"Virtual" CPU | "Virtual" CPU | ... | "Virtual" CPU | Interrupt Handler

Context Switching
+ Scheduling

The Hardware (CPU)

External Devices

Interrupts: Handling

How does interrupt handling change the instruction cycle?

Interrupts: Handling

Table set up by OS kernel; pointers to code to run on different events

System Calls: Under the Hood

read (fd, buffer, nbytes)

Address
0xFFFFFFFF

User space

Return to caller
Trap to the kernel
5 Put code for read in register

Library procedure read

10

4

Increment SP 11
Call read
3 Push fd
2 Push &buffer
1 Push nbytes

User program calling read

6

9

Kernel space (Operating system)

Dispatch 7 8 Sys call handler

0

Concurrency: Thread Lifecycle

Init --(Thread Creation, sthread_create())--> Ready

Ready --(Scheduler Resumes Thread)--> Running

Running --(Thread Yield/Scheduler Suspends Thread, sthread_yield())--> Ready

Running --(Thread Exit, sthread_exit())--> Finished

Running --(Thread Waits for Event, sthread_join())--> Waiting

Waiting --(Event Occurs, Other Thread Calls sthread_join())--> Ready

Concurrency: Thread State

Kernel

Code | Globals | Heap

Kernel Thread 1	Kernel Thread 2	Kernel Thread 3	Process 1	Process 2
TCB 1	TCB 2	TCB 3	PCB 1	PCB 2
Stack	Stack	Stack	Stack	Stack

User-Level Processes

Process 1	Process 2
Thread	Thread
Stack	Stack
Code	Code
Globals	Globals
Heap	Heap

Synchronization: Principals

Concurrent Applications

Shared Objects

Bounded Buffer　　Barrier

Synchronization Variables

Semaphores　　Locks　　Condition Variables

Atomic Instructions

Interrupt Disable　　Test-and-Set

Hardware

Multiple Processors　　Hardware Interrupts

Queueing Lock Implementation (1 Proc)

```
Lock::acquire() {
    disableInterrupts();
    if (value == BUSY) {
        waiting.add(myTCB);
        myTCB->state = WAITING;
        next = readyList.remove();
        switch(myTCB, next);
        myTCB->state = RUNNING;
    } else {
        value = BUSY;
    }
    enableInterrupts();
}
```

```
Lock::release() {
    disableInterrupts();
    if (!waiting.Empty()) {
        next = waiting.remove();
        next->state = READY;
        readyList.add(next);
    } else {
     value = FREE;
    }
    enableInterrupts();
}
```

Multiprocessor Sync Tool!

- Read-modify-write (RMW) instructions
  - Atomically read a value from memory, operate on it, and then write it back to memory
  - Intervening instructions prevented in hardware
- Examples
  - Test and set
  - Intel: xchgb, lock prefix
  - Compare and swap
- Any of these can be used for implementing locks and condition variables!

Test-and-set

- The **test-and-set** instruction is an instruction used to write 1 (set) to a memory location and return its old value as a single **atomic** (i.e., non-interruptible) operation. If multiple processes may access the same memory location, and if a process is currently performing a test-and-set, no other process may begin another test-and-set until the first process's test-and-set is finished.
- Please implement a lock using test-and-set (5 minutes)

```
lock:acquire() {

}

lock:release() {

}
```

Synchronization: Locks

- Lock::acquire
  – wait until lock is free, then take it
- Lock::release
  – release lock, waking up anyone waiting for it

1. At most one lock holder at a time (safety)
2. If no one holding, acquire gets lock (progress)
3. If all lock holders finish and no higher priority waiters, waiter eventually gets lock (progress)

Synchronization: Condition Variables

- Waiting inside a critical section
  - Called only when holding a lock
- CV::Wait — atomically release lock and relinquish processor
  - Reacquire the lock when wakened
- CV::Signal — wake up a waiter, if any
- CV::Broadcast — wake up all waiters, if any

Synchronization: Spinlocks

- A spinlock is a lock where the processor waits in a loop for the lock to become free
  - Assumes lock will be held for a short time
  - Used to protect the CPU scheduler and to implement locks

```
Spinlock::acquire() {
   while (testAndSet(&lockValue) == BUSY)
      ;
}

Spinlock::release() {
   lockValue = FREE;
  memorybarrier();
}
```

Semaphores

- Semaphore has a non-negative integer value
  - P() atomically waits for value to become > 0, then decrements
  - V() atomically increments value (waking up waiter if needed)
- Semaphores are like integers except:
  - Only operations are P and V
  - Operations are atomic
    - If value is 1, two P's will result in value 0 and one waiter

Scheduling: Principals

- Basic scheduling algorithms
  - FIFO (FCFS)
  - Shortest job first
  - Round Robin
- What is an optimal algorithm in the sense of maximizing the number of jobs finished (i.e., minimizing average response time)?

Scheduling: Mixed Workloads??

Tasks

I/O Bound

Issues I/O Request

I/O Completes

Issues I/O Request

I/O Completes

CPU Bound

CPU Bound

Time

Scheduling: MFQ

Priority	Time Slice (ms)	Round Robin Queues
1	10	New or I/O Bound Task
2	20	Time Slice Expiration
3	40	
4	80	

Scheduling: Early Linux

- Linux 1.2: circular queue w/ round-robin policy.
  - Simple and minimal.
  - Did not meet many of the aforementioned goals

- Linux 2.2: introduced scheduling classes (real-time, non-real-time).

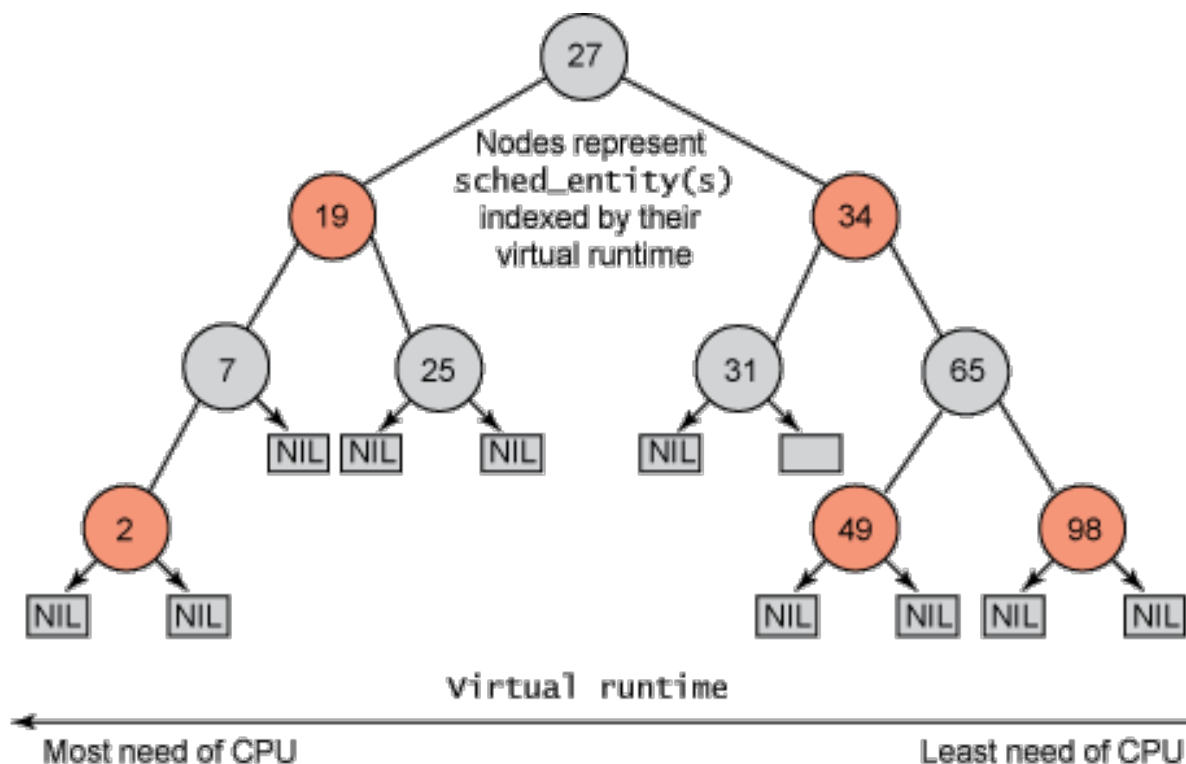
```
/* Scheduling Policies
*/
#define SCHED_OTHER  0 // Normal user tasks (default)
#define SCHED_FIFO   1 // RT: Will almost never be preempted
#define SCHED_RR     2 // RT: Prioritized RR queues
```

Scheduling: CFS

- Merged into the 2.6.23 release of the Linux kernel and is the default scheduler.
- Scheduler maintains a red-black tree where nodes are ordered according to received virtual execution time
- Node with smallest virtual received execution time is picked next
- Priorities determine accumulation rate of virtual execution time
  - Higher priority → slower accumulation rate

Scheduling: Red-Black Trees

- CFS dispenses with a run queue and instead maintains a time-ordered **red-black tree**. Why?



An RB tree is a BST w/ the constraints:

1. Each node is red or black
2. Root node is black
3. All leaves (NIL) are black
4. If node is red, both children are black
5. Every path from a given node to its descendent NIL leaves contains the same number of black nodes

Takeaway: In an RB Tree, the path from the root to the farthest leaf is no more than twice as long as the path from the root to the nearest leaf.

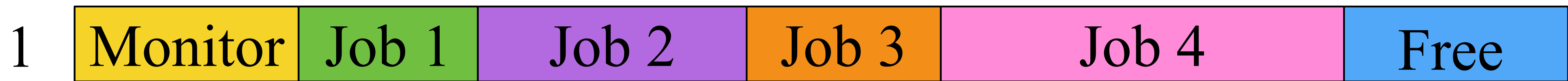
Scheduling: Multi-Processor

- CPU affinity would seem to necessitate a multi-queue approach to scheduling… but how?
- Asymmetric Multiprocessing (AMP): One processor (e.g., CPU 0) handles all scheduling decisions and I/O processing, other processes execute only user code.
- Symmetric Multiprocessing (SMP): Each processor is self-scheduling. Could work with a single queue, but also works with private queues.
  - Potential problems?

Virtual Memory

- Provide user with virtual memory that is as big as user needs
- Store virtual memory on disk
- Cache parts of virtual memory being used in real memory
- Load and store cached virtual memory without user program intervention

Virtual Memory Systems

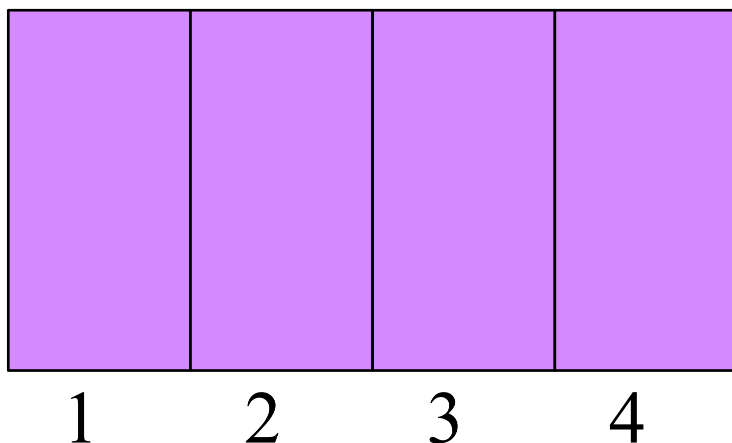
1 

- Fixed partitions
  - Internal fragmentation
- Segmentation (variable partition)
  - External fragmentation
- Paging

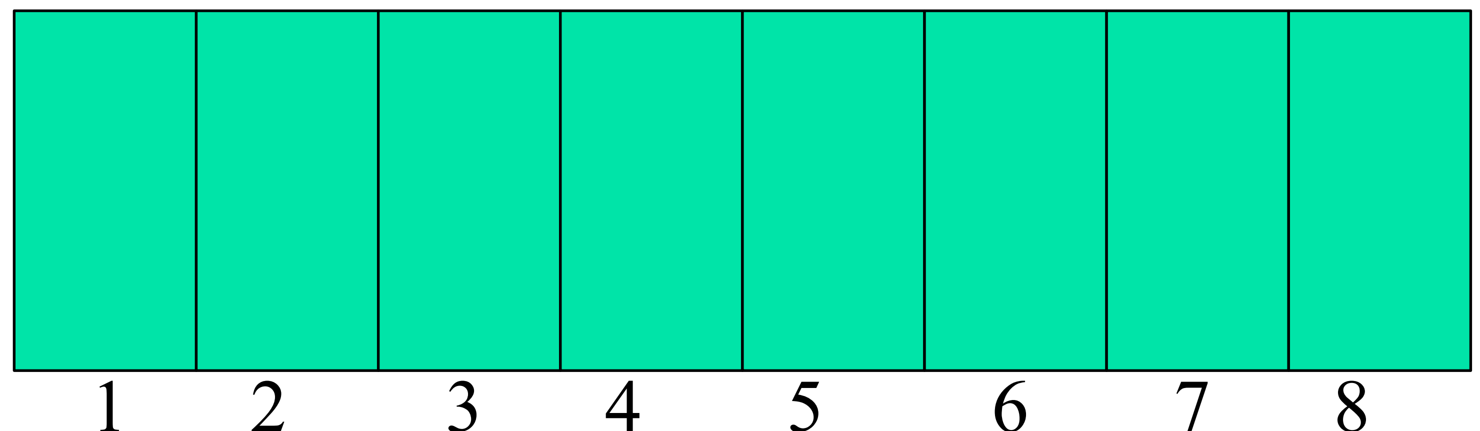
Page Table

	1
	2
	3
	4

Memory



Virtual Memory Stored on Disk



Page Faults

- Occur when we access a virtual page that is not mapped into any physical page
  - A fault is triggered by hardware
- Page fault handler (in OS's VM subsystem)
  - Find if there is any free physical page available
    - If no, evict some resident page to disk (swapping space)
  - Allocate a free physical page
  - Load the faulted virtual page to the prepared physical page
  - Modify the page table

More Q&A