Lecture 8: Scheduling

Professor G. Sandoval

Also some Slides by Brendan Dolan-Gavitt and Bryant and O'Hallaron, Computer Systems: A programmer's Perspective, Third Edition

Fork()

- *Exact* copy of current process

- Only difference between parent and child is return value

Fork() example

```
int main(void)
{
   pid_t pid = fork();

   if (pid == -1) {
      perror("fork failed");
      exit(EXIT_FAILURE);
   }
   else if (pid == 0) {
      printf("Hello from the child process!\n");
      _exit(EXIT_SUCCESS);
   }
   else {
      int status;
      (void)waitpid(pid, &status, 0);
   }
   return EXIT_SUCCESS;
}
```

In-Class Exercise

How many times

is "foo" printed?

```c
#include <stdio.h>
#include <unistd.h>

int main(void) {
    int i = 0;
    for (i = 0; i < 4; i++) {
        fork();
        printf("foo\n");
    }

    return 0;
}
```

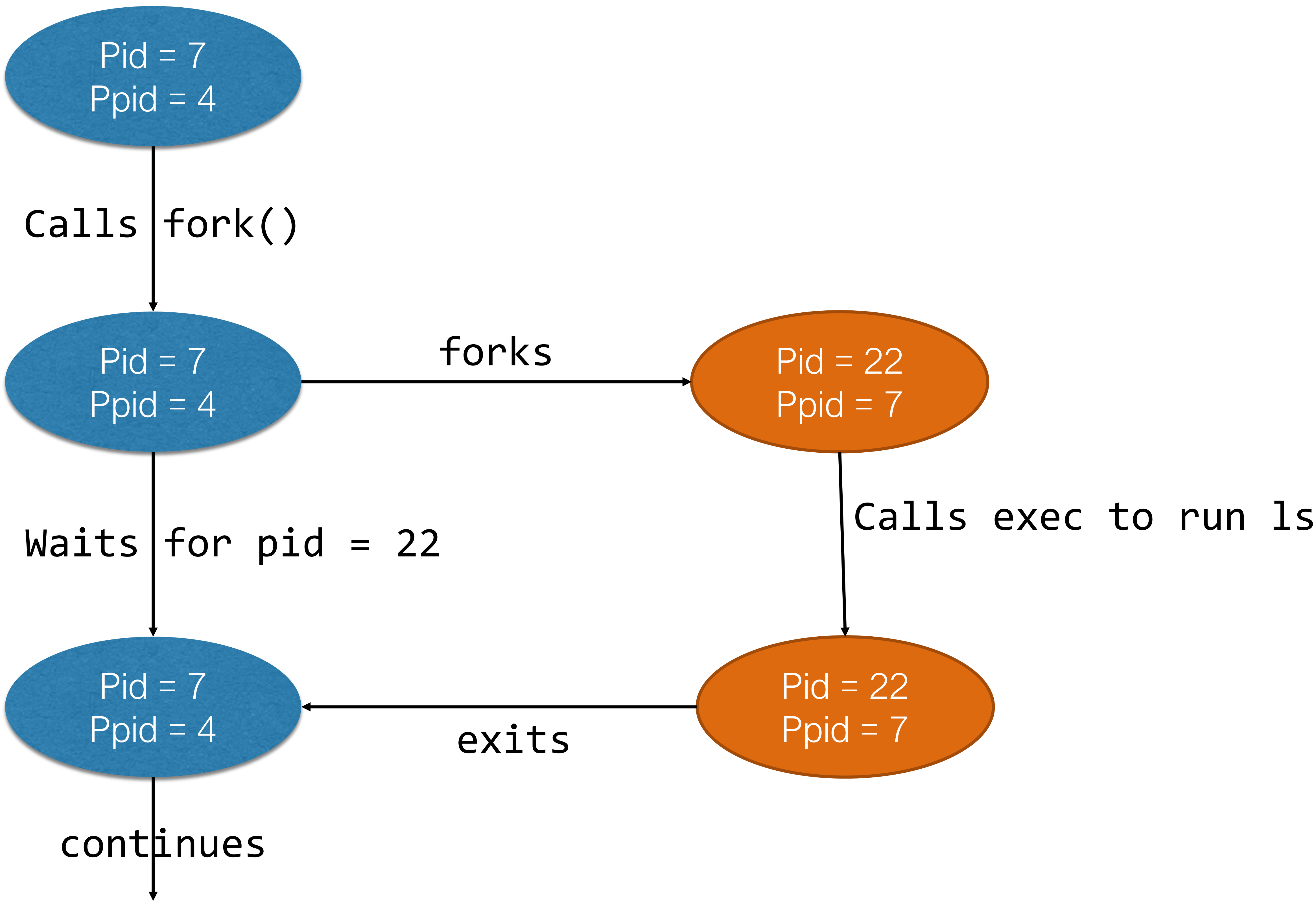
The Role of Exec

- If all we had was fork(), we couldn't launch new programs (why?)
- So we also have exec(), which allows you to *replace* the code and data in the current process with new code and data
- In modern UNIX systems, exec has many variants that provide different features:
  - Passing a variable number of arguments
  - Searching the directories listed in the PATH environment variable for the program

Bash example

- How would you use fork/exec in a shell?
- For example when the shell calls `ls()`

Bash shell example



Today

- Process Scheduling
- Scheduling Algorithms
- Threads
- In Real Life: Windows
- XV6 Process Scheduling Implementation

Metrics

- **Throughput** – jobs completed / time interval
- **Turnaround** – average time between when a job is submitted and when it completes
- **Response time** – time between when a user issues a command and gets the result

Tradeoffs

- Improving on one metric can hurt another
- For example:
  - We want to improve *throughput*, so we decide to only schedule short jobs
  - But now longer jobs never get run, so their *turnaround time* is effectively **infinite**

Today

- Process Scheduling
- Scheduling Algorithms
- Threads
- In Real Life: Windows
- XV6 Process Scheduling Implementation

First Come First Served

- Single queue of ready processes
- The process at the head of the queue runs as long as it likes or until it blocks
- After it runs, you add it to the back of the queue and let the next one in line run

FCFS

- Very easy to program! Can just use a linked list.
- But I/O may suffer
  - CPU-bound process can take up lots of time
  - But I/O bound process will have to block and then wait until it gets back to the head of the queue before it can issue another I/O

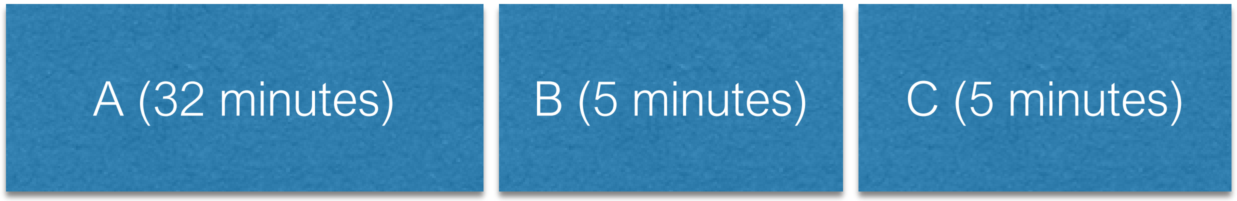
Analyzing First Come First Served

- To measure turnaround time:

  - $\text{Time}_{(\text{completed})} – \text{Time}_{(\text{submitted})}$ / N

Analyzing First Come First Served

- Turnaround time depends on order we pick jobs
  Assuming jobs arrive at time 0:



Turnaround time: (32 + 37 + 42) / 3 = 37 minutes

Analyzing First Come First Served

- Turnaround time depends on order we pick jobs
  Assuming jobs arrive at time 0:

B (5 minutes)	C (5 minutes)	A (32 minutes)

Turnaround time: (5 + 10 + 42) / 3 = 19 minutes

Shortest Job First

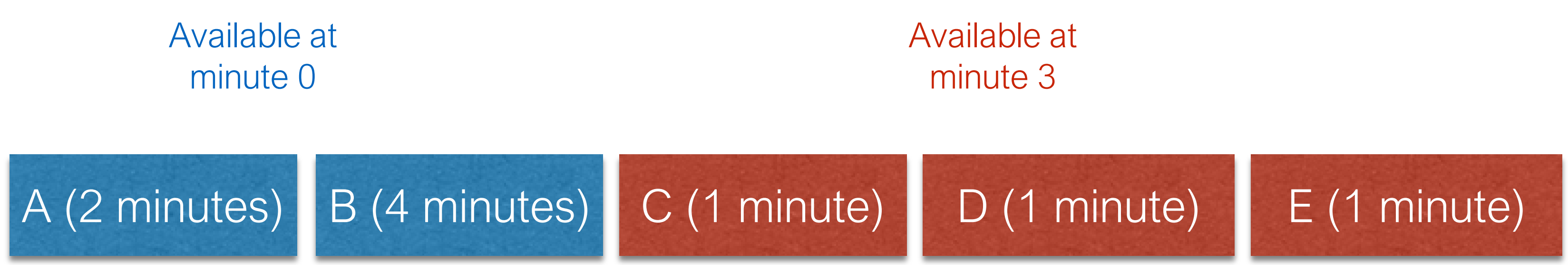
- Batch, non-preemptive
- Assumes we can predict how long each job will take to execute
- Always picks the job with the shortest time to execute to run

Analyzing Shortest Job First

- Using SJF provably minimizes turnaround time
- To see why, consider scheduling 3 jobs with runtimes a, b, and c
- Turnaround time
  $= (a + (a + b) + (a + b + c)) / 3$
  $= (3a + 2b + c) / 3$
- So to minimize turnaround time, we want a to be as small as possible (since it has the largest effect on the average)

Counterexample

- The optimality proof only applies when all jobs are available at time 0
- Suppose we have instead:



Then turnaround time is
(2 + 6 + (7 - 3) + (8 - 3) + (9 - 3)) / 5 = 4.6

- But if we run them in the order B, C, D, E, A, time is:
(4 + (5 - 3) + (6 - 3) + (7 - 3) + 9) / 5 = 4.4

Interactive Scheduling

- In an interactive system, scheduling algorithms are generally *preemptive*
- Time is divided up into slices called *quanta*
- Each process runs for 1 *quantum* and then the scheduler runs again

Round Robin Scheduling

- Again, simple algorithm
- Run first process until its quantum is used up
- Move that process to the end and run the next process until its quantum is used up
- Simple, fair

Design Considerations with Round Robin

- The length of the quantum is typically determined by a hardware timer interval
- This is generally configurable
- So: how long should we make a quantum?

Design Considerations with Round Robin

- Context switching takes some amount of time (swap out CPU registers, change address space)
- Consider:
    - context switching = 1ms,
    - quantum = 4ms,
    - => 20% of time spent just switching

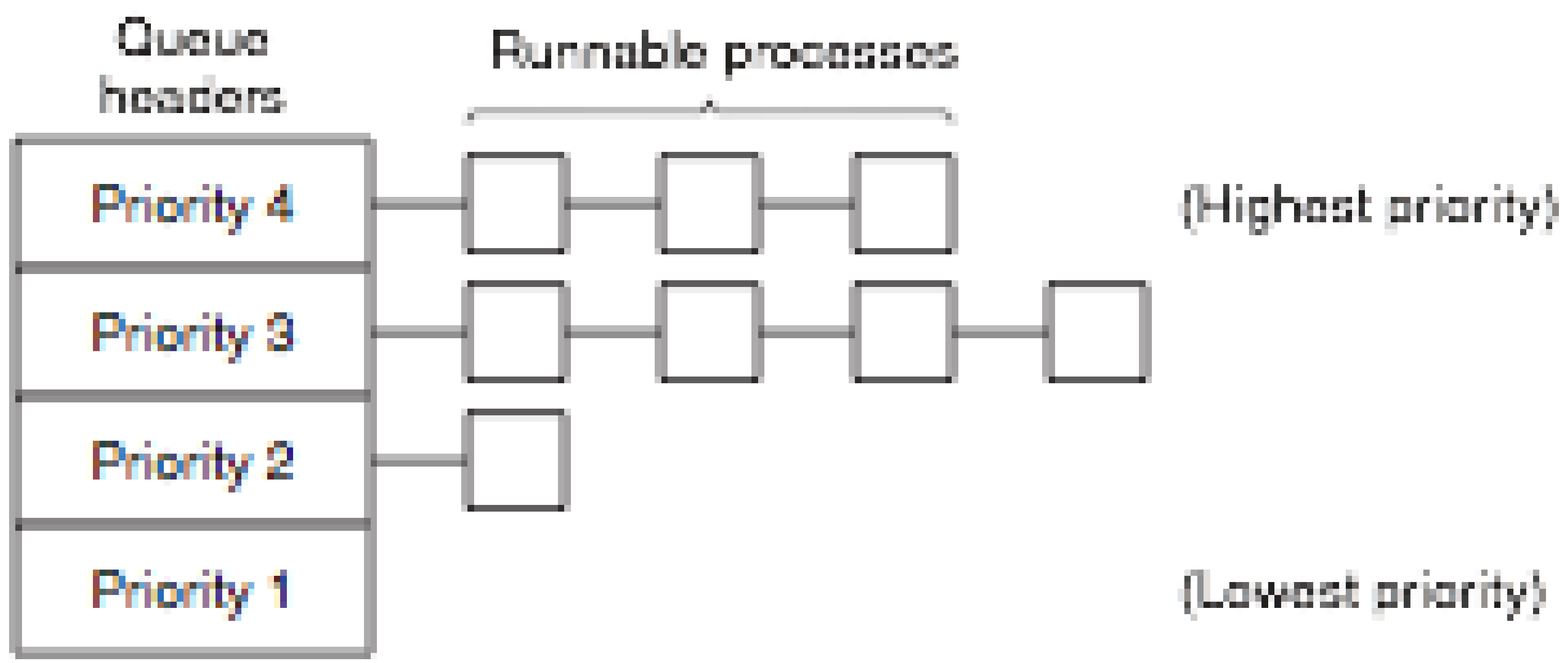
Design Considerations with Round Robin

- What if:
  - Large quantum = 100ms ?
  - => 5 seconds before a process gets to run on a system with 50 active processes
- Tuning this is a balancing act. Values around 10-50ms are common
- xv6 uses a 10ms quantum

Priorities

- Round robin scheduling assumes that every process has the same **priority**
- In reality, we may always want some processes to get scheduled before others
- Possible policies:
  - Safety-critical code should run first
  - Users who pay more should get higher priority
  - Interactive processes should get priority over daemons

Priority Scheduling



Dynamic Priority

- We can also adjust the priority of processes *dynamically*
- One nice way to use this is to dynamically give I/O bound processes more chances to run
- We can set the priority as a function of the fraction of the last quantum the process actually used:

  priority = 1 / f   f = fraction of quantum used

- This would give *higher* priority to processes that used a smaller fraction of their quantum – i.e., processes that waited for I/O

Shortest Process Next

- *Latency* in an interactive system is analogous to *turnaround time* in a batch system
- Unfortunately in an interactive system we don't necessarily know how long a command will take
- But we can make *estimates,* and then update our estimates over time with real data

Process Aging

- Set some initial estimate T_0
- Run the process and measure to get T_1
- Fix *a*, the aging parameter; $0 \le a \le 1$
- After running a process, update the estimate as:
  $T_i = aT_{i-1} + (1-a)T_{i-2}$

Process Aging

- So with a = 1/2, the estimates become:
  - T_0,
  - $T_0/2 + T_1/2$,
  - $T_0/4 + T_1/4 + T_2/2$,
  - $T_0/8 + T_1/8 + T_2/4 + T_3/2$
- Over time, our initial estimate is weighted less and less and more recent events have greater weight

Remember:
$T_i = aT_{i-1} + (1-a)T_{i-2}$

Guaranteed Scheduling

- Guarantee to each of n processes that they will get 1/n about 1/n of the CPU
- As processes run, keep track of how much CPU time they have actually used
- Now we can schedule processes based on how "unfair" we have been to them up to this point
- Easy to say, hard to implement!

Lottery Scheduling

- We can get a probabilistic version of guaranteed scheduling that is much easier to implement
- Give each process a fixed number of *lottery tickets*
- When it comes time to schedule, pick a random number between 1 and the number of tickets
- Schedule the process that won the lottery

Lottery Scheduling

- We can give each process a proportion of the CPU by just giving it that proportion of the tickets
- We can also guarantee that every process *eventually* gets to run as long as it gets at least one ticket
- It's not a true guarantee, however – only a probabilistic one

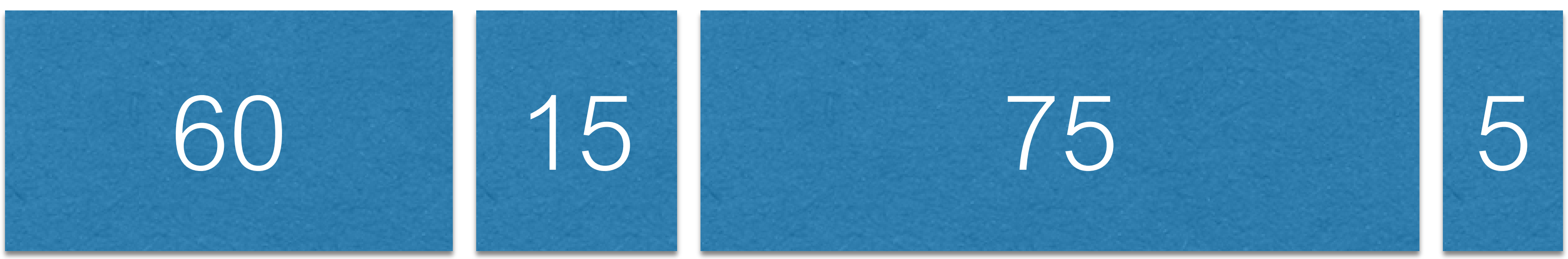
Lottery Scheduling

“All processes are equal, but some are more equal” – George Orwell

Implementing Lottery Scheduling

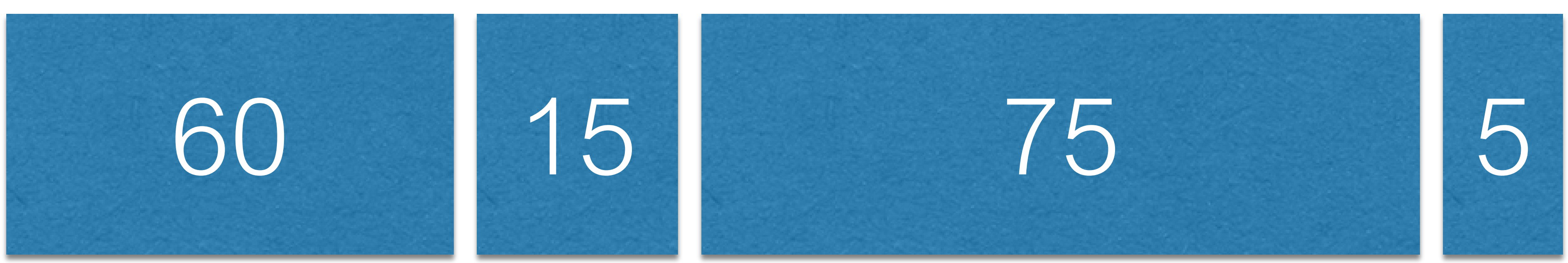
- Take our existing process control block (PCB) data structure and augment it with a **num_tickets** field
- At scheduling time:
  - Generate a random ticket number *winner*
  - Loop over processes, keeping a *counter*
  - If *counter* ≥ *winner* then pick that process
  - Otherwise, add the process's tickets to *counter* and continue

Lottery Scheduling



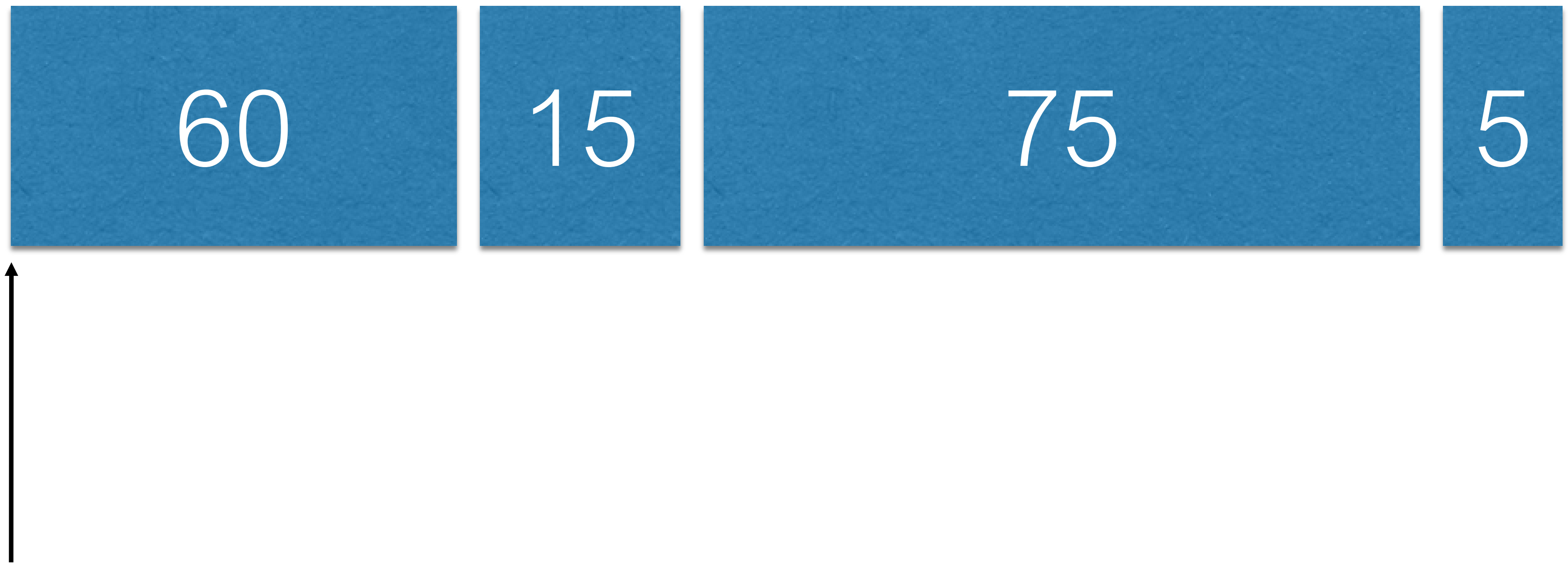
Lottery Scheduling

Winner: 83



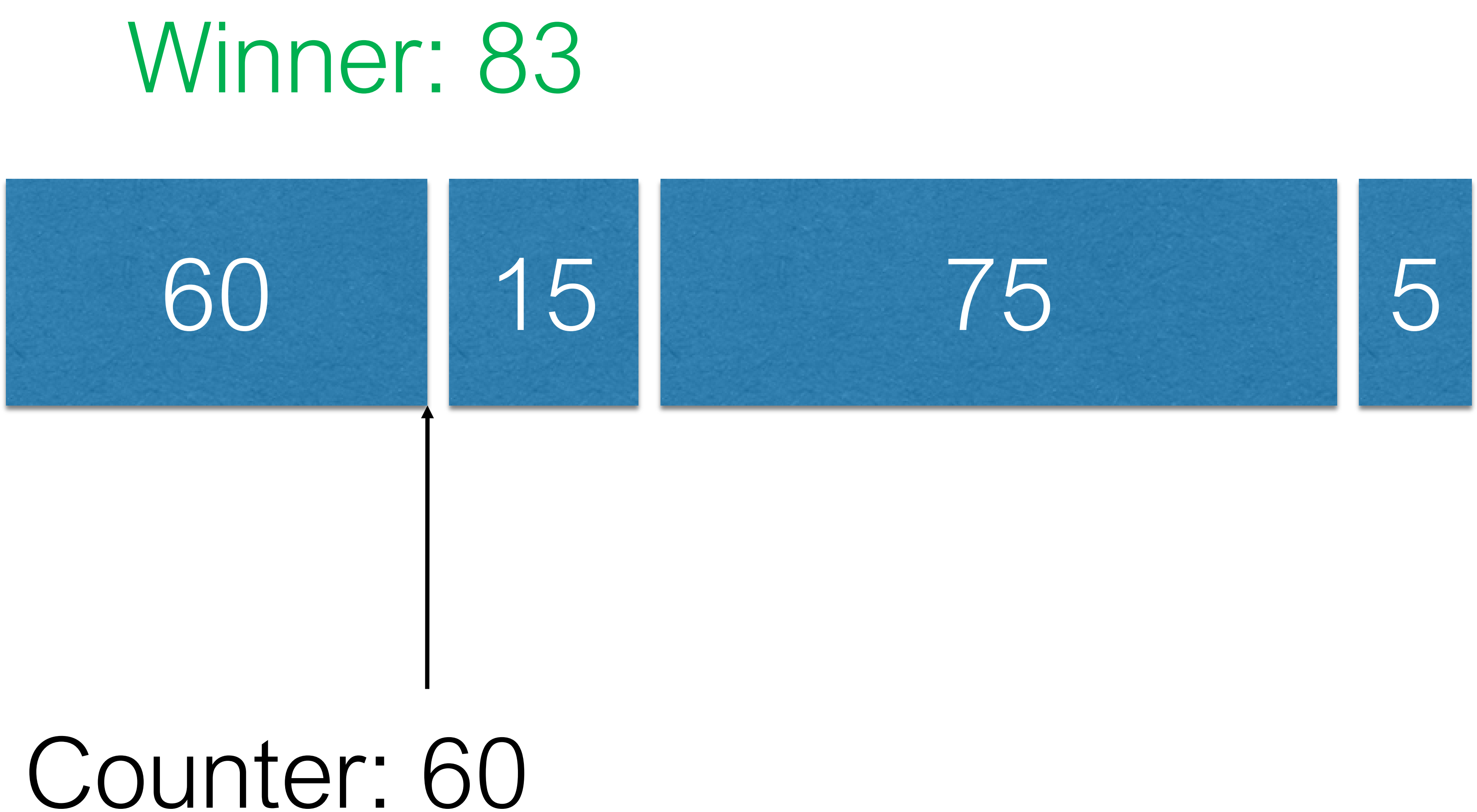
Lottery Scheduling

Winner: 83

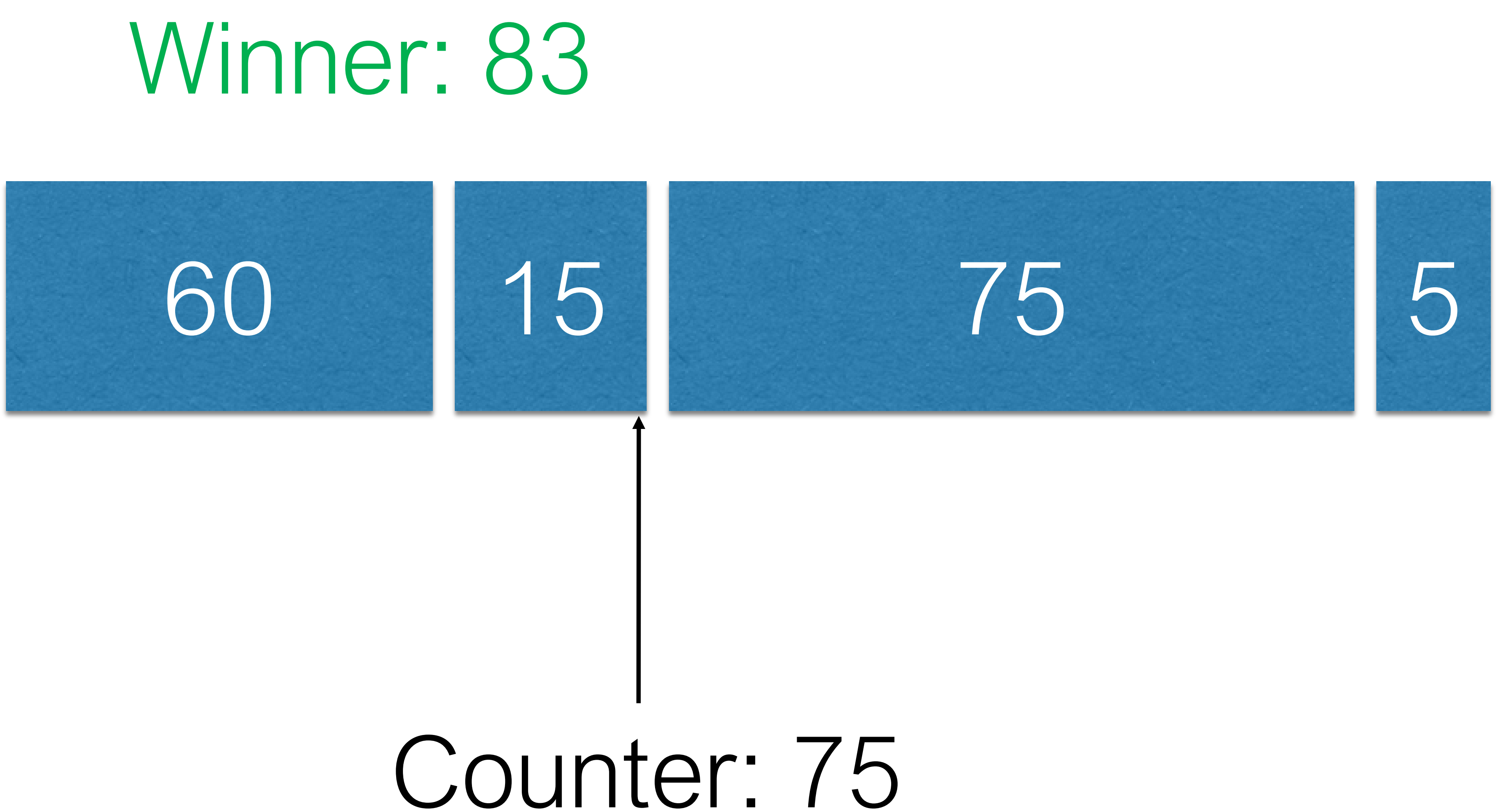


Counter: 0

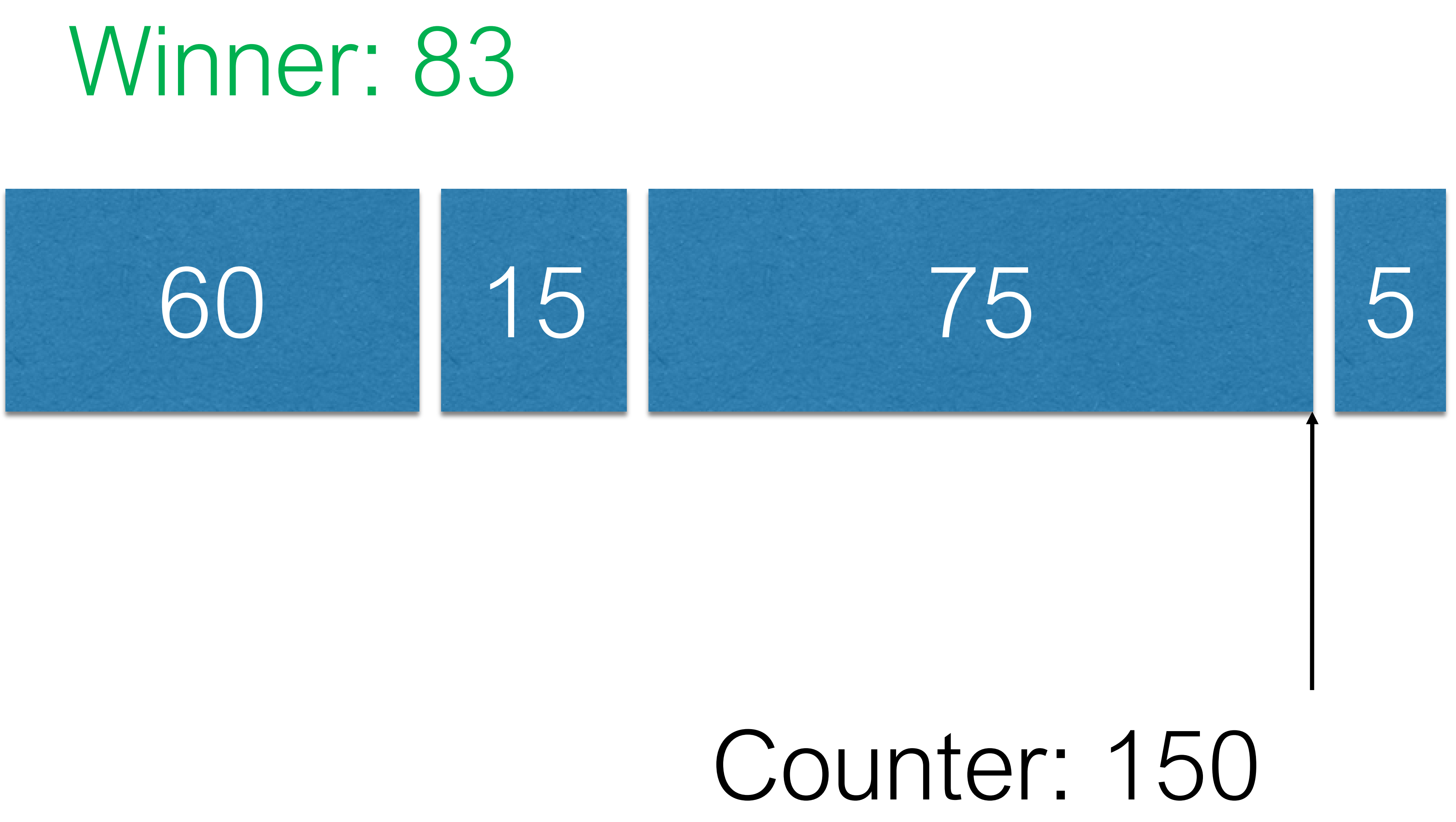
Lottery Scheduling



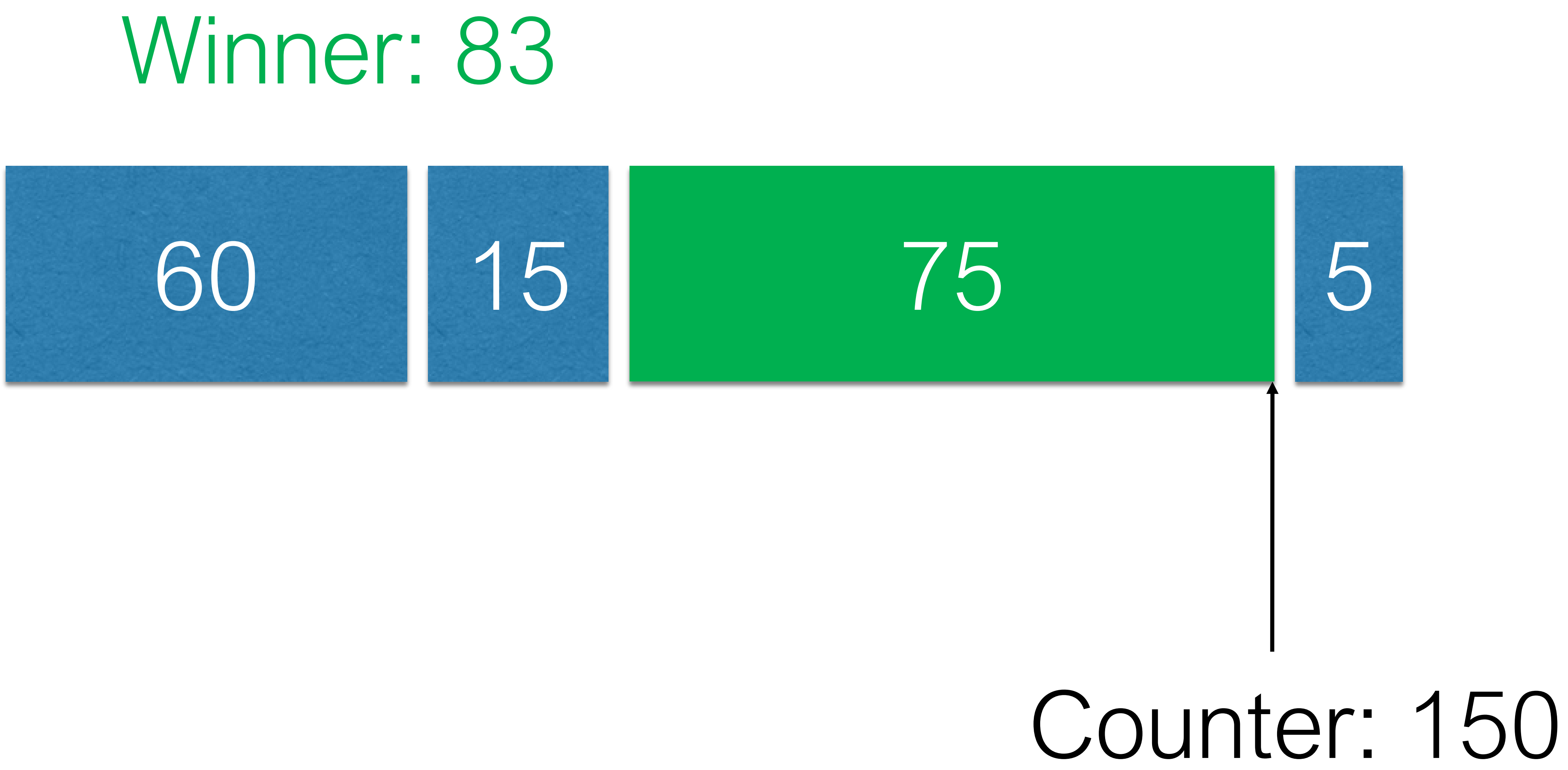
Lottery Scheduling



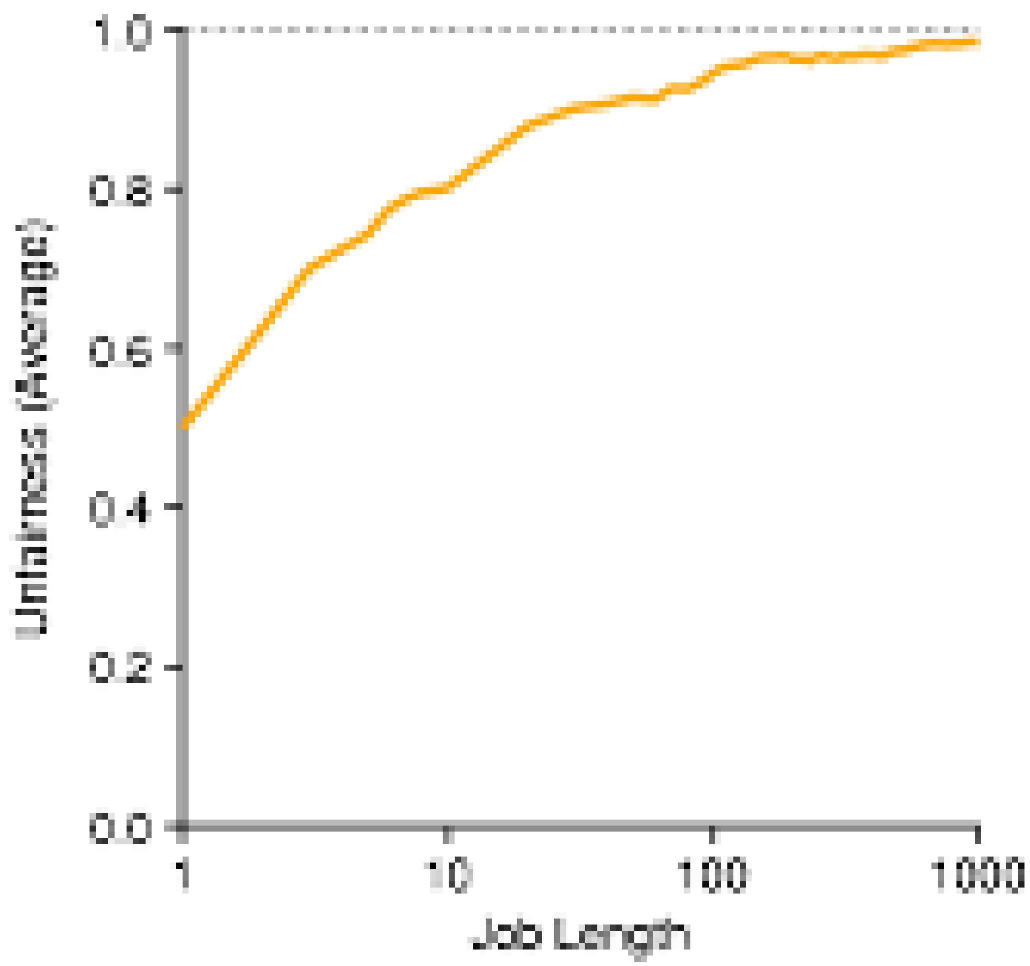
Lottery Scheduling



Lottery Scheduling



Analyzing Lottery Scheduling



Today

- Process Scheduling
- Scheduling Algorithms
- Threads
- In Real Life: Windows
- XV6 Process Scheduling Implementation

THREADS

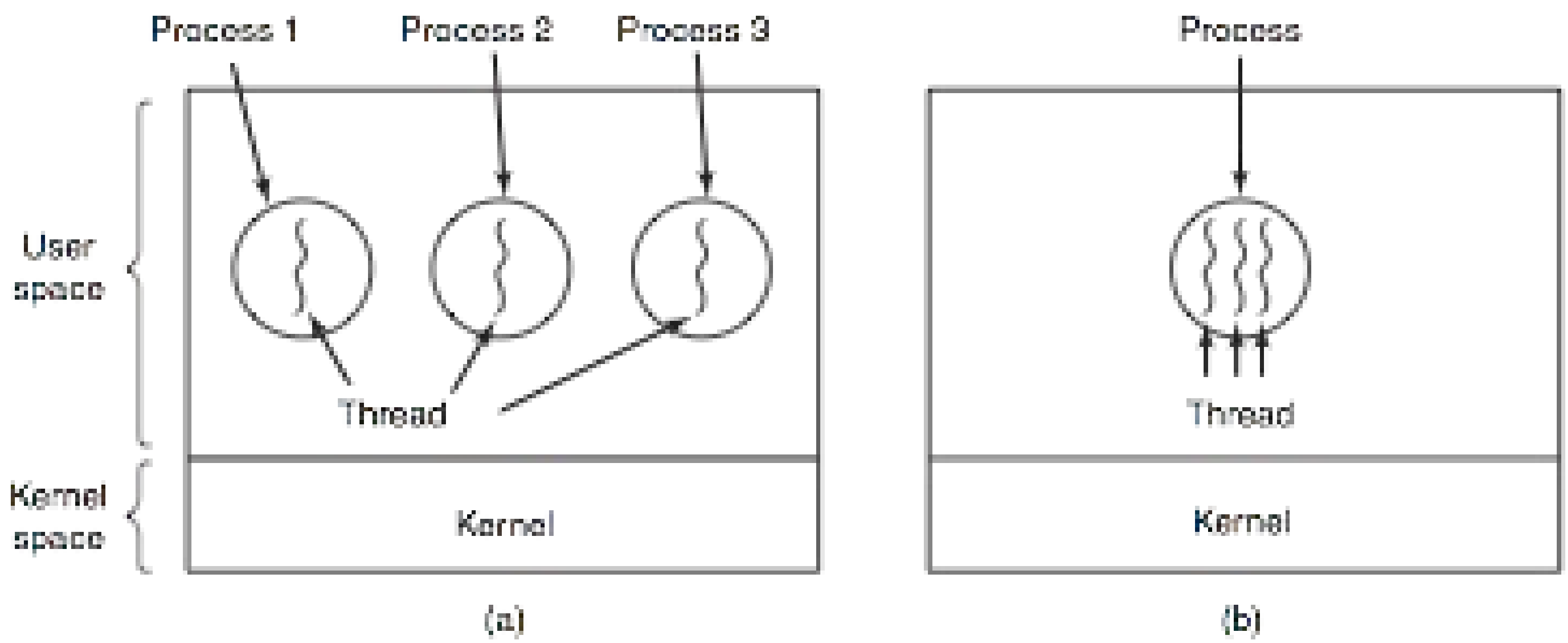
- In many operating systems, we can have multiple *threads* of execution inside a single process
- As with processes, each thread has its own program counter & CPU state
- Unlike processes, multiple threads within a process **share their address space and memory**

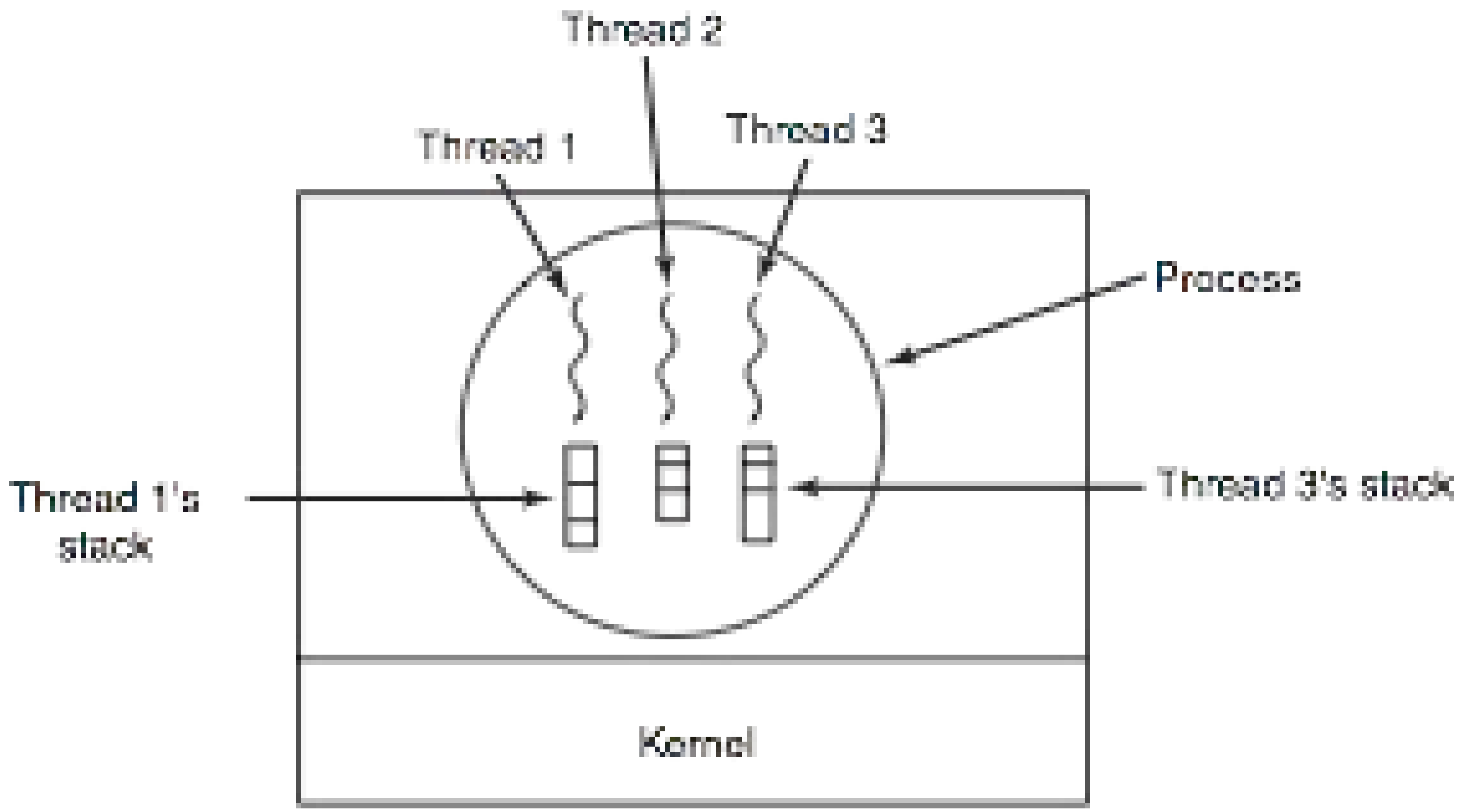
Threading Benefits

- Same benefits as for processes – can have multiple things going on at once – but still share data easily
  - In particular, multiple threads are useful when there are multiple I/O tasks to be done
- Can allow a single application to take advantage of multiple processors

Threading Benefits

- Threads are more lightweight – faster to switch between threads than between processes
- Example: in a web browser, can have threads for responding to user input, loading data from network, rendering HTML


Process 1
Process 2
Process 3
User space
Thread
Kernel space
Kernel
(a)
Process
Thread
Kernel
(b)



Thread 2
Thread 1
Thread 3
Process
Thread 1's stack
Thread 3's stack
Kernel

Threading – Cooperation

- Unlike processes, no *protection* between threads
- Thus, threads are assumed to always be mutually cooperating
- Threads share:
  - Address space
  - Global variables
  - Open Files

Threading Model

- Typically start with a single thread
- API calls then allow:
  - Thread creation
  - Thread exit (without exiting the process)
  - Waiting for another thread to finish (*join*)
  - Giving up the CPU voluntarily (*yield*)

Threading Implementation: User Space

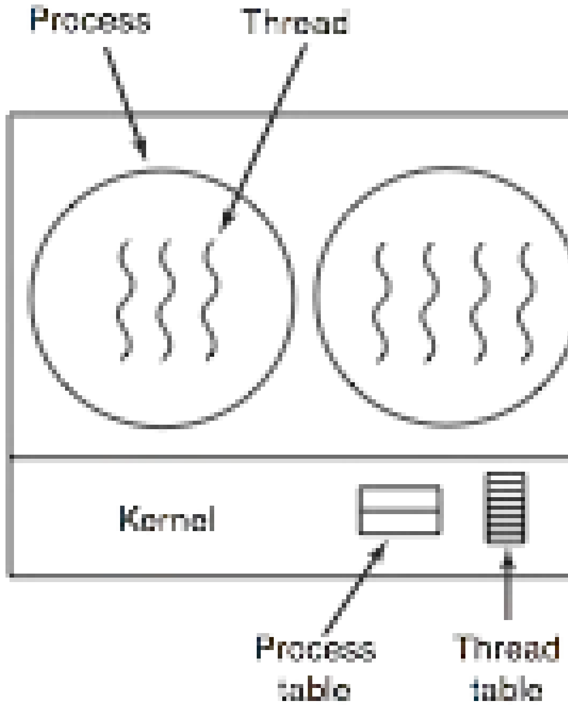
- We can implement threading without any changes to the underlying operating system
- Each process can maintain a table of threads, keeping track of its CPU context & stack
- The process can implement a *thread scheduler* that chooses the next thread to run when one exits or yields

User Space Threading Downsides

- Suppose we have a thread that wants to make a *blocking* system call (e.g., *recv* to wait for a network packet)
- With user-space threading, we can't just make the call, or else all threads will stop (defeating the point of multithreading)
- If the OS supports non-blocking versions of system calls, it's possible to work around this
  - Force the programmer to always use non-blocking versions
  - Write a wrapper library that converts blocking calls to their non-blocking versions and then yields to another thread

Kernel Threads

- Much more common to implement threading in the kernel



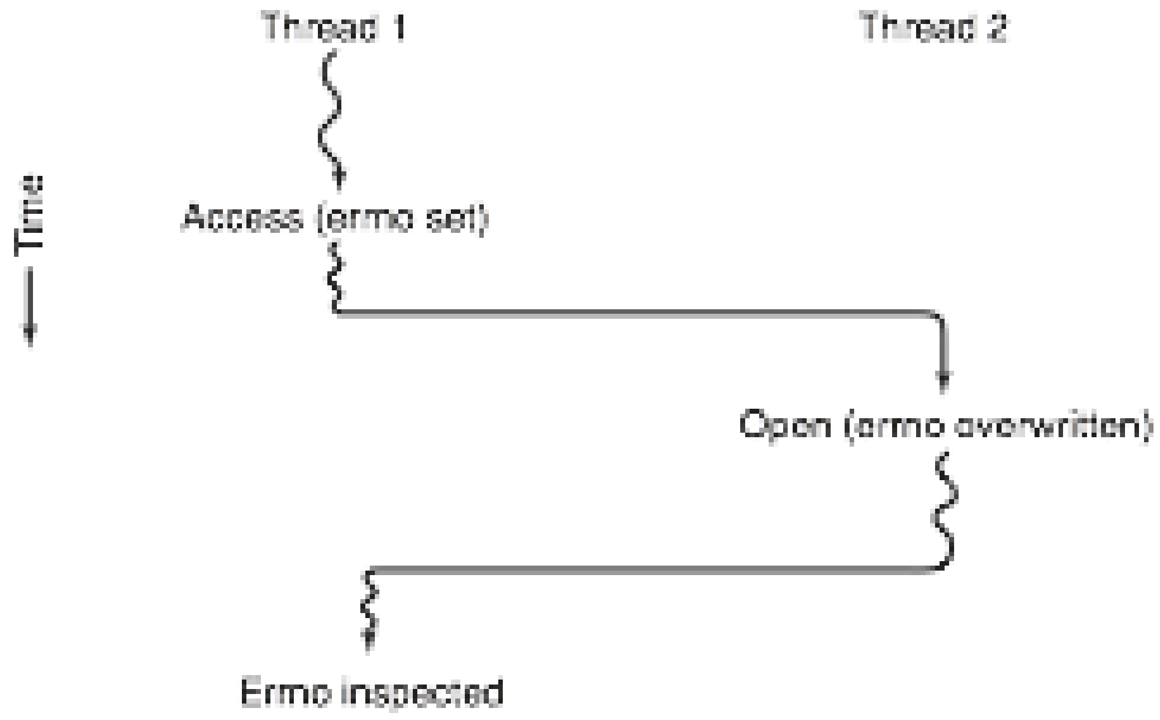
Kernel Threads

- Now threading is handled by the OS scheduler
- No different from switching between processes, but we don't need to change address space
- Downside: switching between user and kernel mode still has some overhead, so it is inherently slower than user-space threading
    - Faster system call mechanisms like sysenter/sysexit are designed to mitigate this problem

Threading: Pitfalls

- Global variables are shared between all threads
- One thread's use of a global may interfere with another
- Example:
  - The UNIX C library provides a global variable, *errno*, which holds the error code of the last API call
  - If two threads try to use the C library and check the status of *errno*, one may get incorrect results

errno Conflict



Note: on modern UNIX systems (e.g. Linux) *errno* is actually replaced by a call to a function __errno_location() that *is* thread-safe

Thread-Local Storage

- Instead of global variables, we can have an API that lets us have separate "globals" for each thread
- Example – Windows API:
  - TlsAlloc() – sets up a thread-local storage area
  - TlsSetValue(TlsIndex, Value) – puts Value into the thread's local storage slot
  - TlsGetValue(TlsIndex, *Value) – retrieves Value from the thread's local storage

Thread-Local Storage

- TLS can also be implemented as a compiler extension
- In GCC, we can declare a variable as
  __thread int i;
- At runtime, when we reference *i*, it will automatically retrieve the version for the current thread

Threading Takeaways

- Having multiple threads can be extremely beneficial for responsiveness, taking advantage of multiple processors, etc.
- Converting single-threaded applications or libraries to use multiple threads is not trivial
- We will cover these issues in more detail when we talk about *concurrency*

Today

- Process Scheduling
- Scheduling Algorithms
- Threads
- In Real Life: Windows
- XV6 Process Scheduling Implementation

History

- MS-DOS & Original Windows were **not** multitasking => **no scheduler**.
- Windows 3.1 used cooperative multitasking.
- Windows 95 introduced a preemptive scheduler but still supported legacy 16bit apps without preemption.
- Windows NT-based OSs use a multilevel feedback queue with 32 pri levels. 0 -15 normal. 16 – 31 real-time.

Windows Scheduling

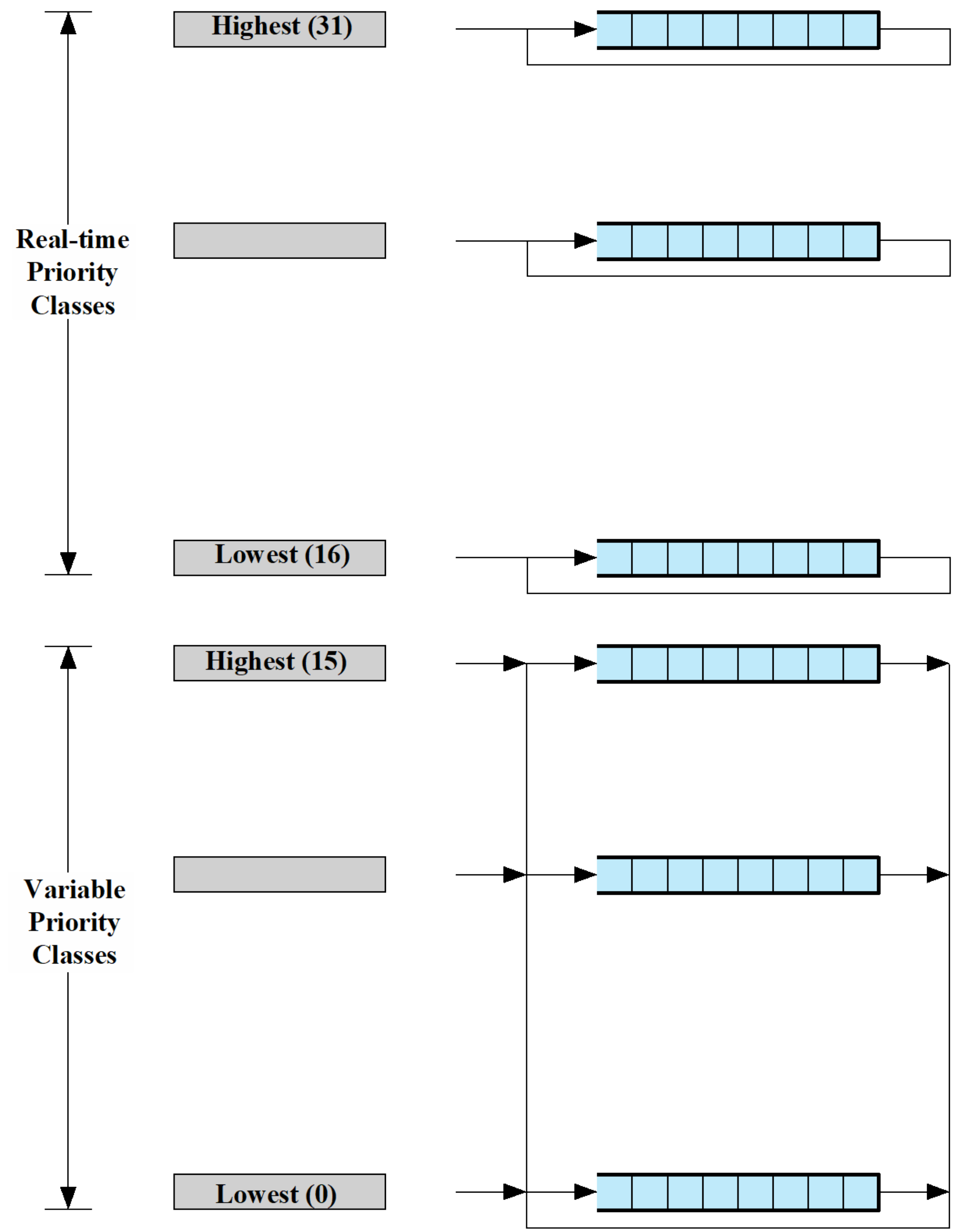
- Priorities organized into two classes, each with 16 priority levels:

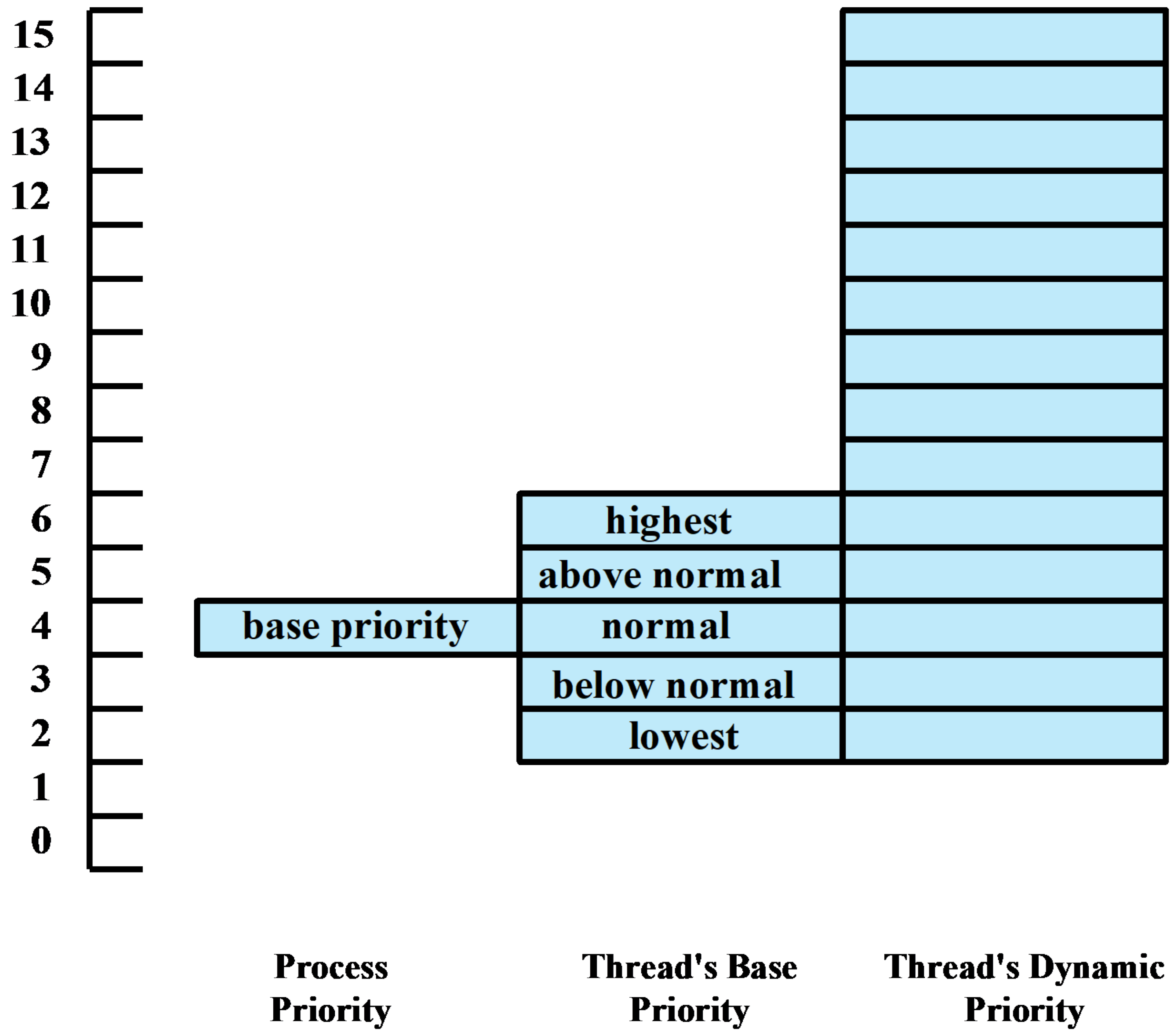
real time priority class

- all threads have a fixed priority that never changes
- all of the active threads at a given priority level are in a round-robin queue
- Threads requiring immediate attention: comms & real time

variable priority class

- a thread's priority begins an initial priority value and then may be temporarily boosted during the thread's lifetime


Highest (31)
Real-time
Priority
Classes
Lowest (16)
Highest (15)
Variable
Priority
Classes
Lowest (0)


15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
highest
above normal
base priority
normal
below normal
lowest
Process Priority
Thread's Base Priority
Thread's Dynamic Priority

Multiprocessor Scheduling

- Windows supports multiprocessor and multicore
- Threads of any process can run on any processor
- If no affinity then kernel assigns ready thread to the next available processor
- Multiple threads from the same process can be executing simultaneously on multiple processors

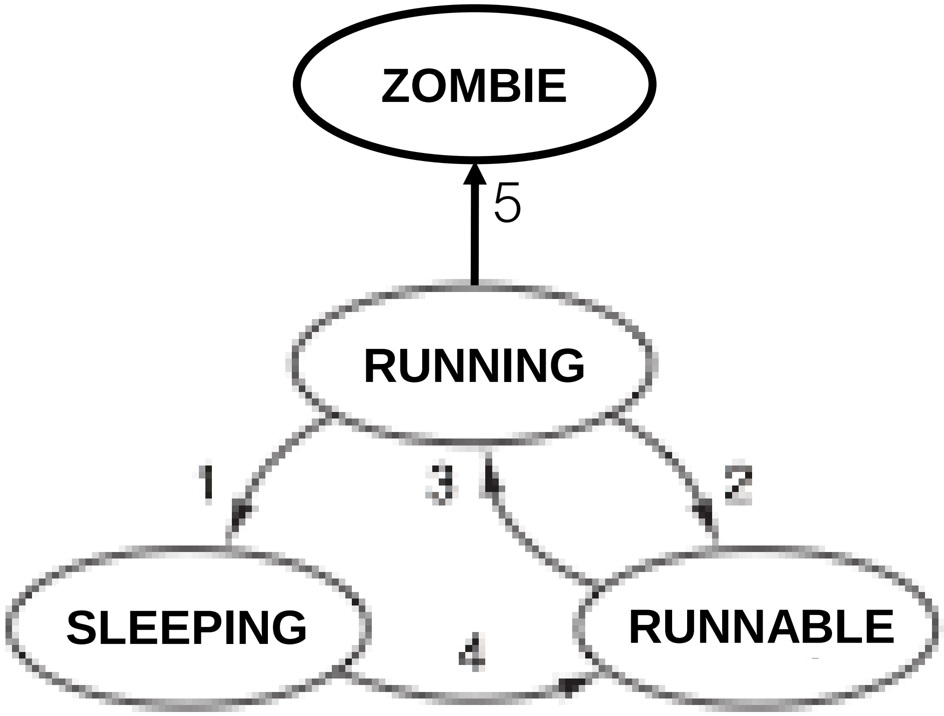
Multiprocessor Scheduling

- Soft affinity
  - used as a default by the kernel dispatcher
  - dispatcher tries to assign a ready thread to the same processor it last ran on
- Hard affinity
  - application restricts thread execution only to certain processors

Today

- Process Scheduling
- Scheduling Algorithms
- Threads
- In Real Life: Windows
- XV6 Process Scheduling Implementation

xv6: State Transitions



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

1. Process Blocks for Input

RUNNING → SLEEPING

- Example: waiting for the user to type something
- In `console.c`, `consoleread` calls **`sleep()`** to put any process waiting for keyboard input to sleep until a key is pressed

```c
int
consoleread(struct inode *ip, char *dst, int n)
{
  uint target;
  int c;

  iunlock(ip);
  target = n;
  acquire(&input.lock);
  while(n > 0){
    while(input.r == input.w){
      if(proc->killed){
        release(&input.lock);
        ilock(ip);
        return -1;
      }
      sleep(&input.r, &input.lock);
    }
```

```
// Atomically release lock and sleep on chan.
// Reacquires lock when awakened.
void
sleep(void *chan, struct spinlock *lk)
{
[... locking code omitted ...]
  // Go to sleep.
  proc->chan = chan;
  proc->state = SLEEPING;
  sched();

  // Tidy up.
  proc->chan = 0;

[... code omitted ...]
}
```

2. Scheduler Picks Another Process

RUNNING → **RUNNABLE**

- Example: time slice is up (timer interrupt fires)
- In trap.c, the trap function will forcibly give up the CPU on a timer interrupt

Trap

```
void
trap(struct trapframe *tf)
{

[... other trap handling omitted ...]

  // Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check nlock.
  if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();

  // Check if the process has been killed since we yielded
  if(proc && proc->killed && (tf->cs&3) == DPL_USER)
    exit();
}
```

Yielding

```
// Give up the CPU for one
//  scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  proc->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

3. Scheduler Picks This Process

RUNNABLE → **RUNNING**

- Example: the main scheduler() function in proc.c

```c
Void scheduler(void)
{
  [ . . . ]
  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);
  }
}
```

4. Input Becomes Available

SLEEPING ⟶ RUNNABLE

- Example: user presses a key
- In trap.c, the interrupt handler recognizes that a keyboard interrupt has occurred and calls the keyboard handler, which calls the console handler
- Console handler finally calls wakeup() to notify any processes waiting for keyboard input

trap.c

```
void
trap(struct trapframe *tf)
{
[...]
  switch(tf->trapno){
[...]
  case T_IRQ0 + IRQ_KBD:
    kbdintr();
[...]
  }
}
```

kbd.c

```
void
kbdintr(void)
{
  consoleintr(kbdgetc);
}
```

console.c

```
void
consoleintr(int (*getc)(void))
{
[...]
        if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
          input.w = input.e;
          wakeup(&input.r);
        }
[...]
}
```

proc.c

```c
// Wake up all processes sleeping on chan.
void
wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}

// Wake up all processes sleeping on chan.
// The ptable lock must be held.
static void
wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}
```

5. Process Exits

RUNNING $\longrightarrow$ **ZOMBIE**

- Happens at process exit (i.e. the exit() system call)
- Process gets marked as a zombie, and anyone that might be waiting for its exit status gets woken up
- Process doesn't actually get destroyed until someone calls wait()
- If the parent isn't around to call wait(), the zombie gets reparented (assigned to init)

proc.c

```
// Exit the current process.  Does not return.
// An exited process remains in the zombie state
// until its parent calls wait() to find out it exited.
void
exit(void)
{
  struct proc *p;
[...]
  // Parent might be sleeping in wait().
  wakeup1(proc->parent);

  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == proc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  // Jump into the scheduler, never to return.
  proc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

Who Cleans Up?

```
// Wait for a child process to exit and return its pid.
// Return -1 if this process has no children.
int
wait(void)
{
  struct proc *p;
  int havekids, pid;

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for zombie children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != proc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->state = UNUSED;
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        release(&ptable.lock);
        return pid;
      }
    }

[...]

    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(proc, &ptable.lock);  //DOC: wait-sleep
  }
}
```

The Role of Init

- Init is the first process created
- It spawns the system shell (sh)
- After starting the shell, sits in a loop calling wait() in case any zombies get assigned to it

init.c

```
int
main(void)
{
  int pid, wpid;
[...]
  for(;;){
[...]
    while((wpid=wait()) >= 0 && wpid != pid)
      printf(1, "zombie!\n");
  }
}
```

Question

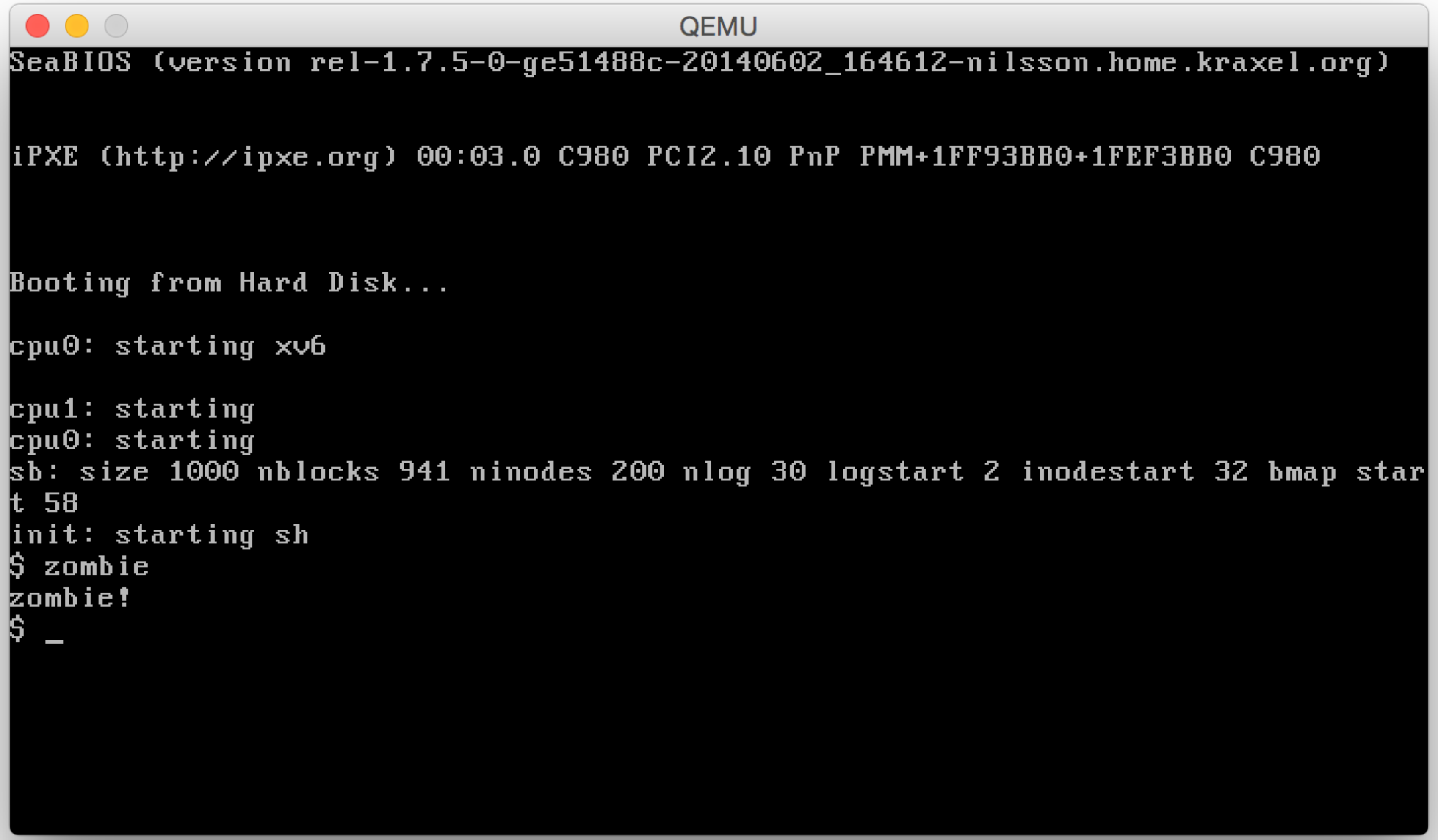
- How can we create an orphan zombie that must be adopted by init?
- Or, restated – how do we make init.c reach the printf that prints "zombie!" ?

zombie.c

```c
// Create a zombie process that
// must be reparented at exit.

#include "types.h"
#include "stat.h"
#include "user.h"

int
main(void)
{
  if(fork() > 0)
    sleep(5);  // Let child exit before parent.
  exit();
}
```


QEMU
SeaBIOS (version rel-1.7.5-0-ge51488c-20140602_164612-nilsson.home.kraxel.org)
iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF93BB0+1FEF3BB0 C980
Booting from Hard Disk...
cpu0: starting xv6
cpu1: starting
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ zombie
zombie!
$ _