Lecture 5: Scheduling (cont.)

Instructor: Mitch Neilsen

Office: N219D

Outline

- Reading:
  - Ch. 4 - Threads
  - Ch. 5 - CPU Scheduling
  - Ch. 6 - Synchronization (next week)
- Project 1: Scheduling and Synchronization
  - Alarm Clock
  - Priority-based Scheduler
  - Synchronization and Priority Inheritance
  - [Extra Credit] MLFQ Scheduler

Quote of the Day

"The man who doesn’t read good books has no advantage over the man who can’t read them. "

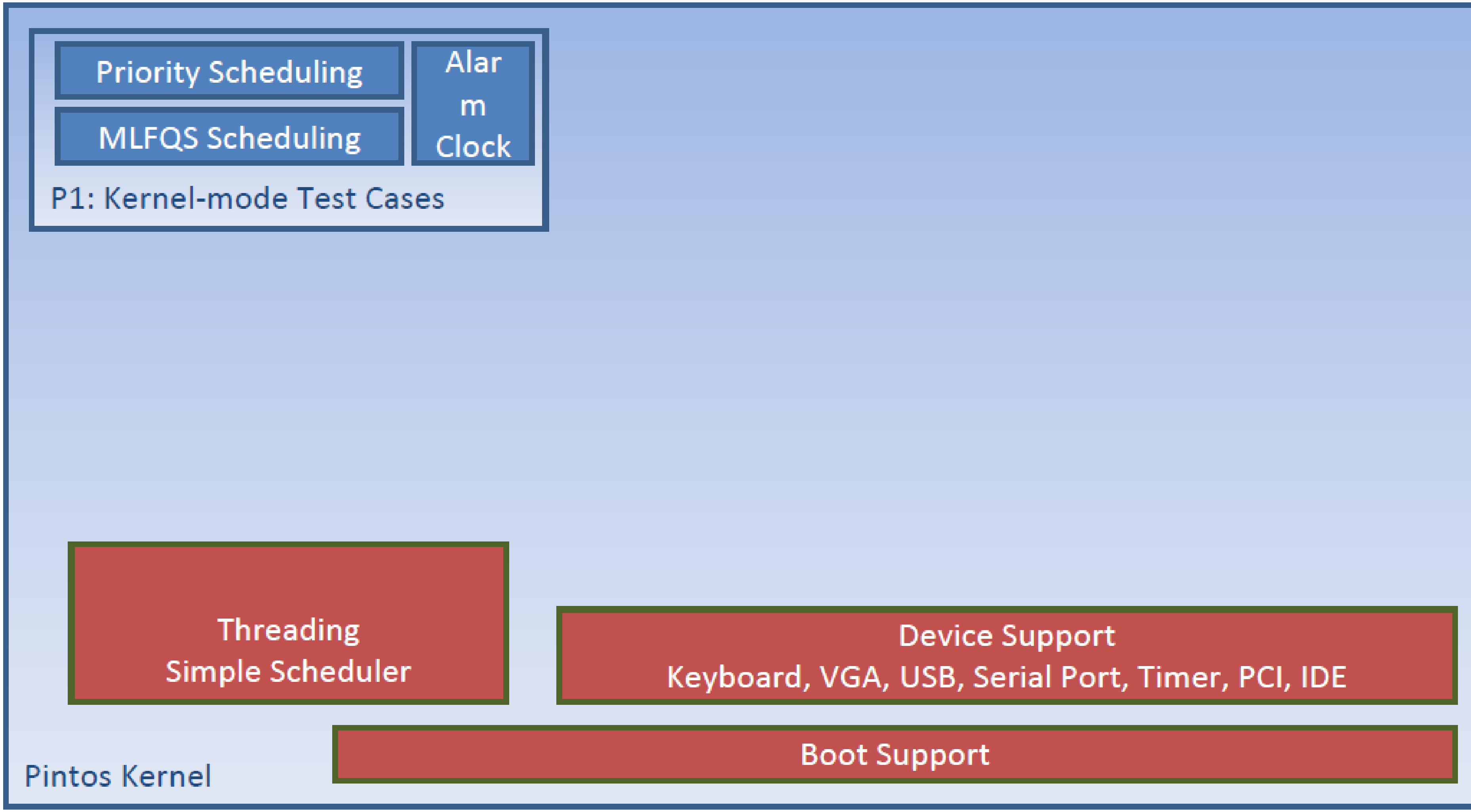
-- Mark Twain

Project 1: Thread Scheduling

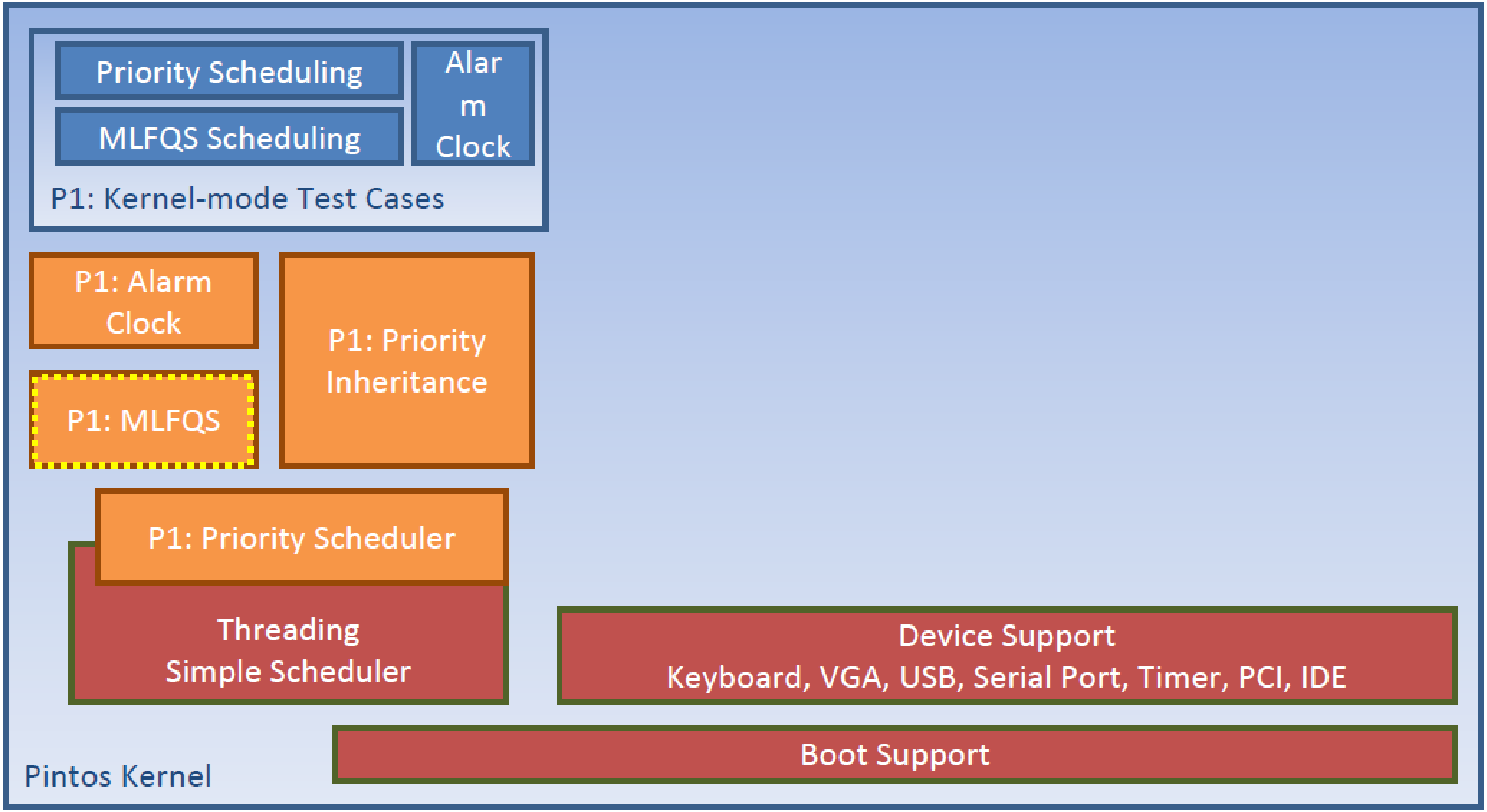
Project 1 Overview

- Extend the functionality of a minimally functional thread system
- Implement
  - Alarm Clock
  - Priority Scheduling
    - Including priority inheritance (priority donation)
  - Advanced MLFQ Scheduler [Extra Credit]

Project 1: Components



Project 1: Components

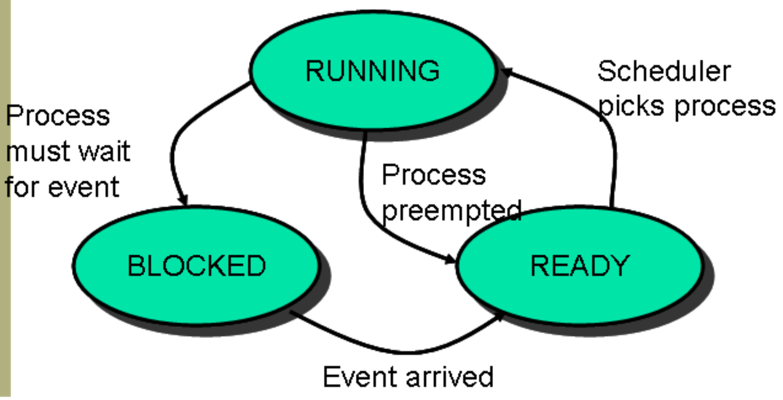


Pintos Thread System

src/threads/thread.h

```
struct thread
 {
   tid_t tid;              /* Thread identifier. */
   enum thread_status status;          /* Thread state. */
   char name[16];   /* Name (for debugging purposes). */
   uint8_t *stack;    /* Saved stack pointer. */
   int priority;         /* Priority. */
   struct list_elem allelem;  /* List element for all-threads list.*/
   /* Shared between thread.c and synch.c. */
   struct list_elem elem;              /* List element. */

You add more fields here as you need them.

#ifdef USERPROG
   /* Owned by userprog/process.c. */
   uint32_t *pagedir;                  /* Page directory. */
#endif
   /* Owned by thread.c. */
   unsigned magic;  /* Detects stack overflow. */
 };
```

```
4 kB +---------------------------------+
     |          kernel stack           |
     |                |                |
     |                |                |
     |                V                |
     |         grows downward          |
     |                                 |
     |                                 |
     |                                 |
     |                                 |
     |                                 |
     |                                 |
     |                                 |
     |                                 |
     +---------------------------------+
     |              magic              |
     |                :                |
     |                :                |
     |               name              |
     |              status             |
0 kB +---------------------------------+
```



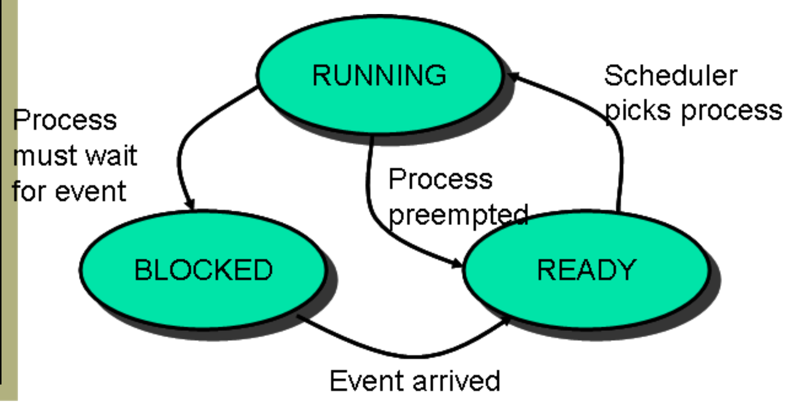
Pintos Thread System

src/threads/thread.c

```
..
  /* Random value for struct thread's `magic' member.
   Used to detect stack overflow.  See the big comment at the top
   of thread.h for details. */
#define THREAD_MAGIC 0xcd6abf4b

/* List of processes in THREAD_READY state, that is, processes
   that are ready to run but not actually running. */
static struct list ready_list;

/* List of all processes.  Processes are added to this list
   when they are first scheduled and removed when they exit. */
static struct list all_list;

/* Idle thread. */
static struct thread *idle_thread;
```

See src/lib/kernel/list.c for list handling functions

```
4 kB +---------------------------------+
     |          kernel stack           |
     |                |                |
     |                |                |
     |                V                |
     |         grows downward          |
     |                                 |
     |                                 |
     |                                 |
     |                                 |
     |                                 |
     |                                 |
     |                                 |
     |                                 |
     +---------------------------------+
     |              magic              |
     |                :                |
     |                :                |
     |               name              |
     |              status             |
0 kB +---------------------------------+
```



Pintos Thread System (contd…)

- Read threads/thread.c, threads/switch.S, and threads/synch.c to understand:
  - How the switching between threads occur
  - How the provided scheduler works
  - How the various synchronizations primitives work

Alarm Clock

- Reimplement timer_sleep( ) in devices/timer.c without busy waiting

```
/* Suspends execution for approximately TICKS timer ticks. */
void timer_sleep (int64_t ticks){
  int64_t start = timer_ticks ();
  ASSERT (intr_get_level () == INTR_ON);
  while (timer_elapsed (start) < ticks)
    thread_yield ();
}
```

- Implementation details
  - Remove thread from ready list and put it back after sufficient ticks have elapsed
  - Use semaphore to block thread on semaphore associated with thread calling timer_sleep

Semaphore [Dijkstra]

A semaphore is a structure consisting of 2 parts:

```
struct semaphore {
        int count;  // number of resources available
        queue Q;  // queue of process/thread ids of blocked
}
```

Shorthand notation:

semaphore S = 1 → S.count = 1, S.Q = { }

Operations on Semaphores

There are two basic semaphore operations:

sem_wait(S):

if (S.count > 0) then S.count = S.count -1;

else block calling process in S.Q;

sem_signal(S):

if (S.Q is non-empty) then wakeup a process in S.Q;

else S.count = S.count + 1;

Semaphore Example: Mutual Exclusion

Semaphore S = 1;

Thread A:

```
sem_wait(S);
   (do work in critical section CS);
sem_signal(S);
```

Thread B:

```
sem_wait(S);
   (do work in CS);
sem_signal(S);
```

Semaphore Example: Order Execution

Semaphore S = 0;

Thread A → Thread B:

Thread A:	Thread B:
(do work);	
sem_signal(S);	sem_wait(S);
	(do work);

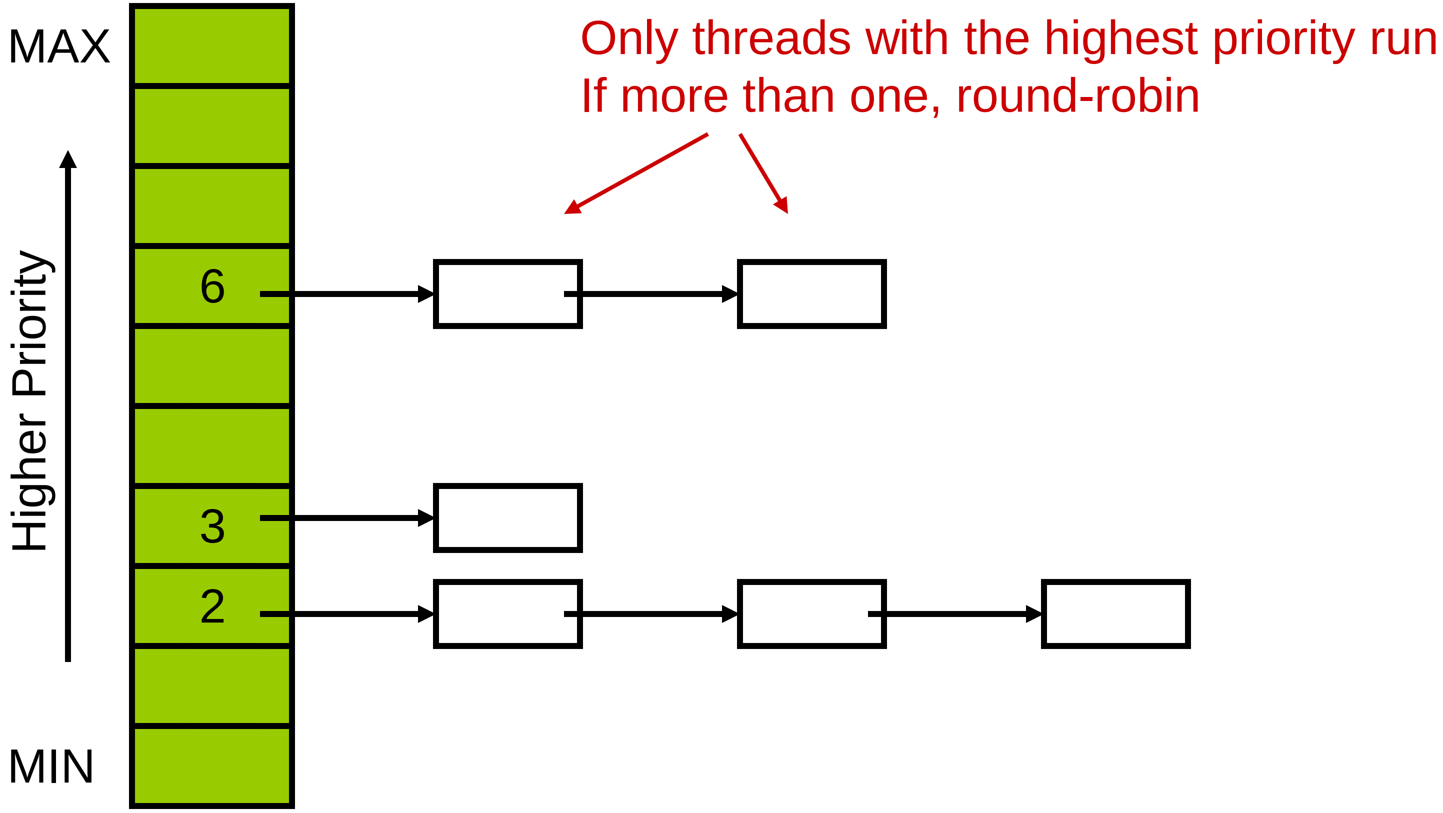
Pintos Semaphores

- struct semaphore s;
- sema_init(&s, 1);
- sema_down(&s);
- sema_up(&s);

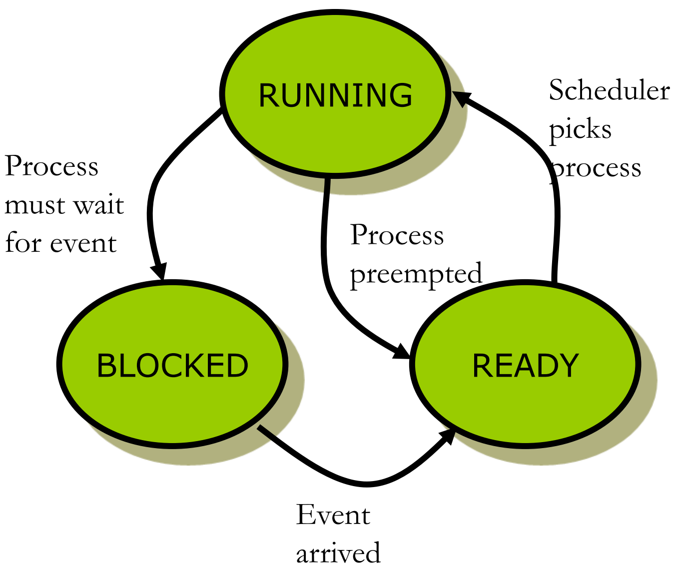
Priority Scheduler

- Ready thread with highest priority gets the processor
- When a thread is added to the ready list that has a higher priority than the currently running thread, immediately yield the processor to the new thread
- When threads are waiting for a lock (e.g., on a semaphore or condition variable), the highest priority waiting thread should be woken up first
- Implementation details
  - compare priority of the thread being added to the ready list with that of the running thread
  - select next thread to run based on priorities
  - compare priorities of waiting threads when releasing locks, semaphores, condition variables

Priority Based Scheduling



Using thread_yield() to implement preemption



- Current thread ("RUNNING") is moved to READY state, added to READY list.
- Then scheduler is invoked. Picks a new READY thread from READY list.
- Case a): there's only 1 READY thread. Thread is rescheduled right away
- Case b): there are other READY thread(s)
  - b.1) another thread has higher priority – it is scheduled
  - b.2) another thread has same priority – it is scheduled provided the previously running thread was inserted in tail of ready list.
- "thread_yield()" is a call you can use whenever you identify a need to preempt current thread.
- Exception: inside an interrupt handler, use "intr_yield_on_return()" instead – don't yield until the interrupt service routine returns

Priority Inversion

- Strict priority scheduling can lead to a phenomenon called "**priority inversion**"
- Supplemental reading:
  - What really happened on the Mars Pathfinder?
- Consider the following example where prio(H) > prio(M) > prio(L)

  H needs a lock currently held by L, so H blocks

  M that was already on the ready list gets the processor before L

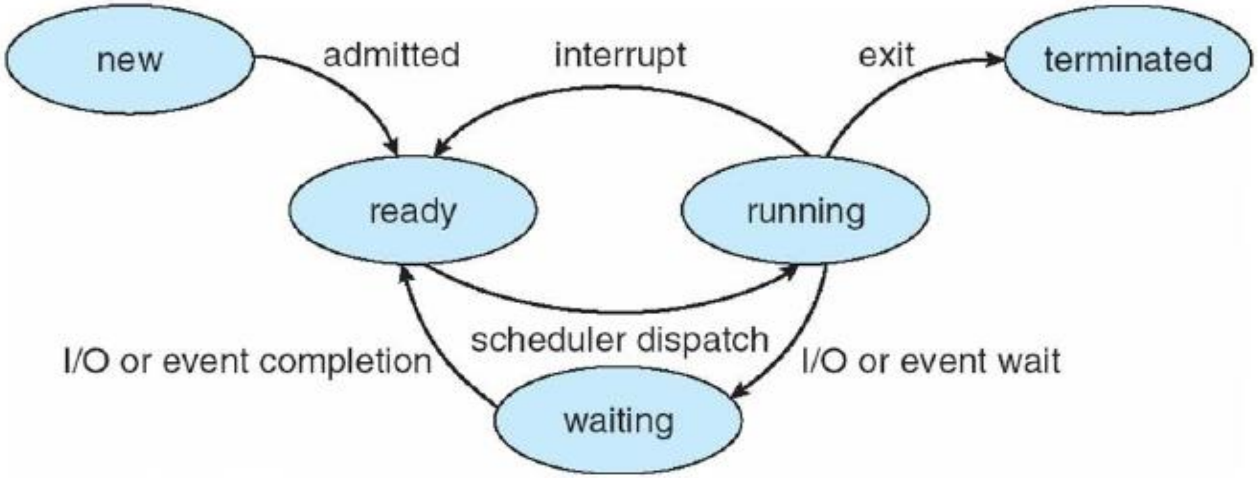
  H indirectly waits for M

  - (on Path Finder, a watchdog timer noticed that H failed to run for some time, and continuously reset the system)

Priority Donation

- When a high priority thread H waits on a lock held by a lower priority thread L, donate H’s priority to L and recall the donation once L releases the lock
- Implement priority donation for locks
- Handle the cases of multiple donations and nested donations

CPU Scheduling



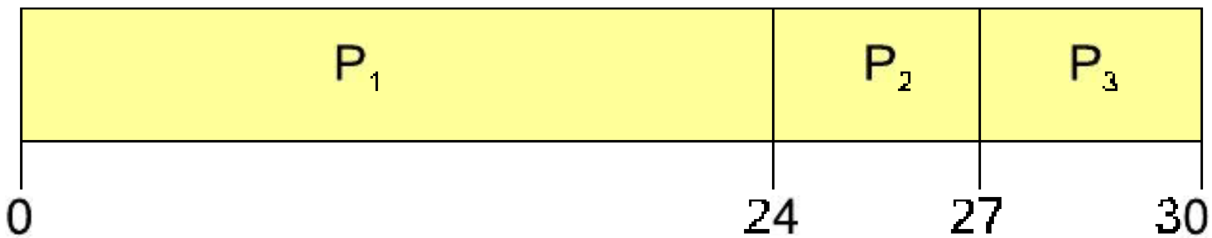
- **Scheduling decisions may take place when a process:**
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Exits
- **Non-preemptive schedules use 1 & 4 only**
- **Preemptive schedulers run at all four points**

Scheduling criteria

- **Why do we care?**
  - What goals should we have for a scheduling algorithm?
- ***Throughput* – # of procs that complete per unit time**
  - Higher is better
- ***Turnaround time* – time for each proc to complete**
  - Lower is better
- ***Response time* – time from request to first response (e.g., key press to character echo, not launch to exit)**
  - Lower is better
- **Above criteria are affected by secondary criteria**
  - *CPU utilization* – keep the CPU as busy as possible
  - *Waiting time* – time each proc waits in ready queue

Example: FCFS Scheduling

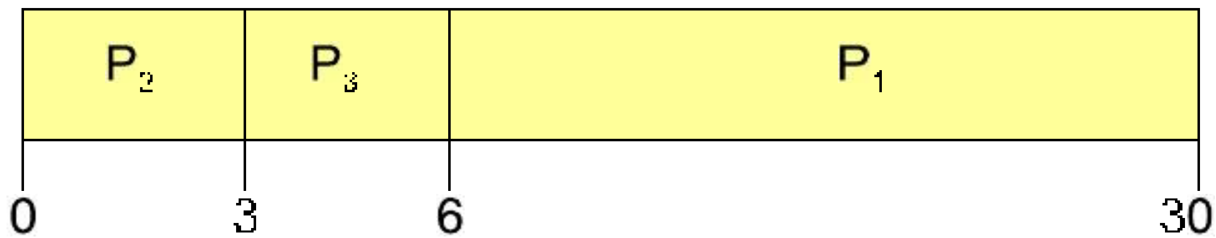
- **Run jobs in order that they arrive**
  - Called "*First-come first-served*" (FCFS)
  - E.g.., Say P_1 needs 24 sec, while P_2 and P_3 need 3.
  - Say P_2 , P_3 arrived immediately after P_1 , get:



- **Dirt simple to implement—how good is it?**
- **Throughput: 3 jobs / 30 sec = 0.1 jobs/sec**
- **Turnaround Time:** P_1 : 24**,** P_2 : 27**,** P_3 : 30
  - Average TT: (24 + 27 + 30)/3 = 27
- **Can we do better?**

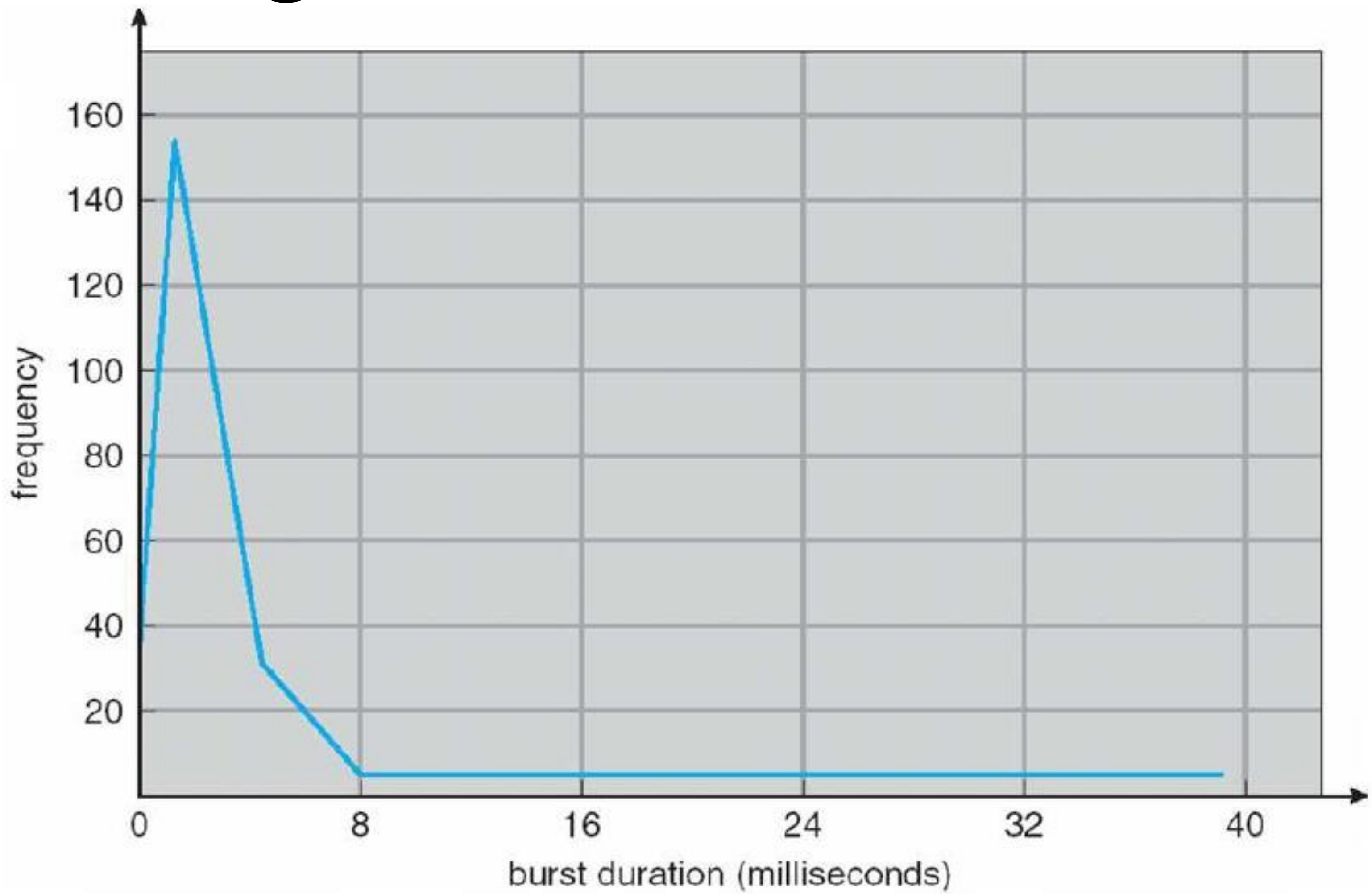
FCFS continued

- **Suppose we scheduled** P_2, P_3**, then** P_1
  - Would get:



- **Throughput: 3 jobs / 30 sec = 0.1 jobs/sec**
- **Turnaround time:** $P_1 : 30$, $P_2 : 3$, $P_3 : 6$
  - Average TT: (30 + 3 + 6)/3 = 13 – much less than 27
- **Lesson: scheduling algorithm can reduce TT**
  - Minimizing waiting time can improve RT and TT
- **What about throughput?** Still the same, 0.1 jobs/sec

Histogram of CPU-burst times



- **What does this mean for FCFS?**

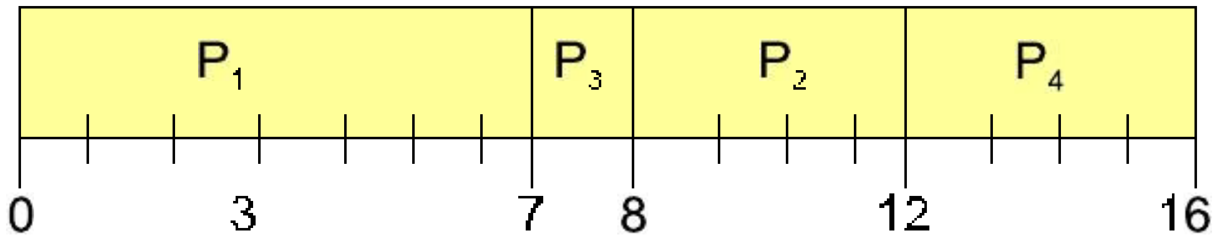
SJF Scheduling

- ***Shortest-job first* (SJF) attempts to minimize TT**
  - Schedule the job whose next CPU burst is the shortest
- **Two schemes:**
  - *Non-preemptive* – once CPU given to the process it cannot be preempted until completes its CPU burst
  - *Preemptive* – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt (Known as the *Shortest-Remaining-Time-First* or SRTF)
- **What does SJF optimize?**
  - Gives minimum average *waiting time* for a given set of processes

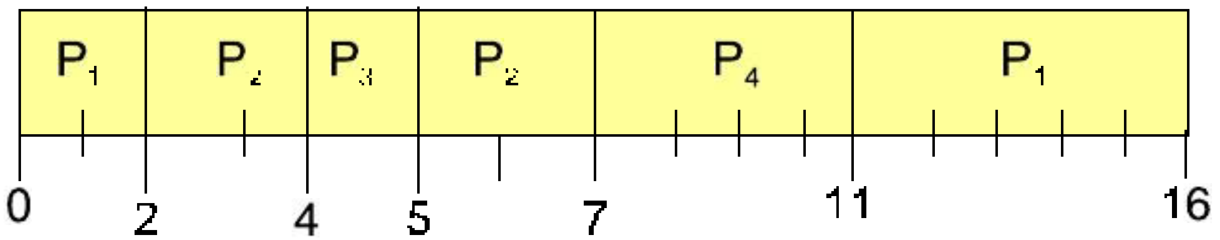
Examples

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- **Non-preemptive**



- **Preemptive**

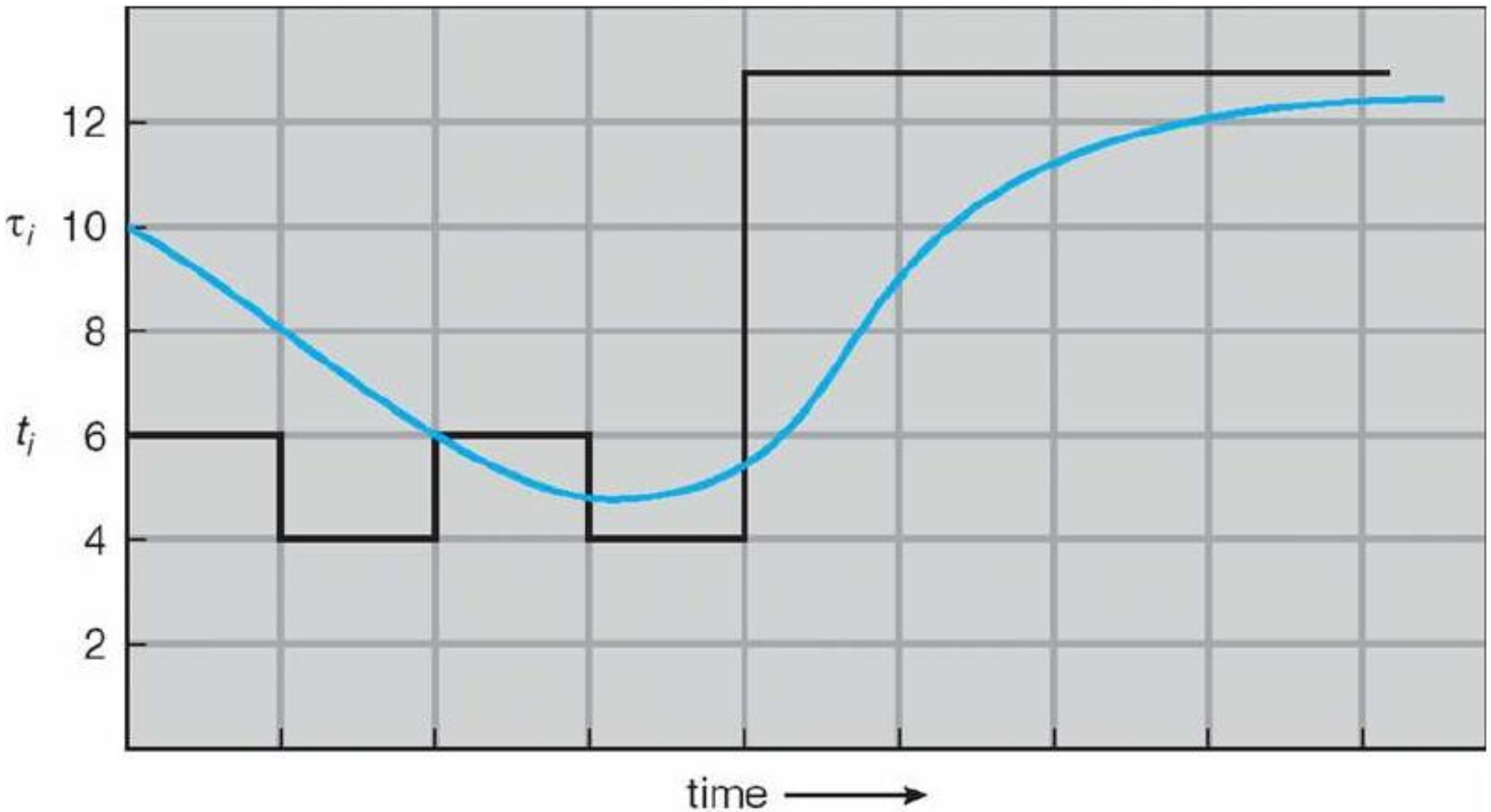


- **Drawbacks?**

SJF limitations

- **Doesn't always minimize average turnaround time**
  - Only minimizes waiting time, which minimizes response time
  - Example where turnaround time might be suboptimal?
- **Can lead to unfairness or starvation**
- **In practice, can't actually predict the future**
- **But can estimate CPU burst length based on past**
  - Exponentially weighted average a good idea
  - t_n actual length of proc's n^{th} CPU burst
  - τ_{n+1} estimated length of proc's $n + 1^{\text{st}}$
  - Choose parameter α where $0 < \alpha \leq 1$
  - Let $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$

Exp. weighted average example



CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Round robin (RR) scheduling

- **Solution to fairness and starvation**
  - Preempt job after some time slice or ***quantum***
  - When preempted, move to back of FIFO queue
  - (Most systems do some flavor of this)
- **Advantages:**
  - Fair allocation of CPU across jobs
  - Low average waiting time when job lengths vary
  - Good for responsiveness if small number of jobs
- **Disadvantages?**
  - Performance depends on size of time quantum used

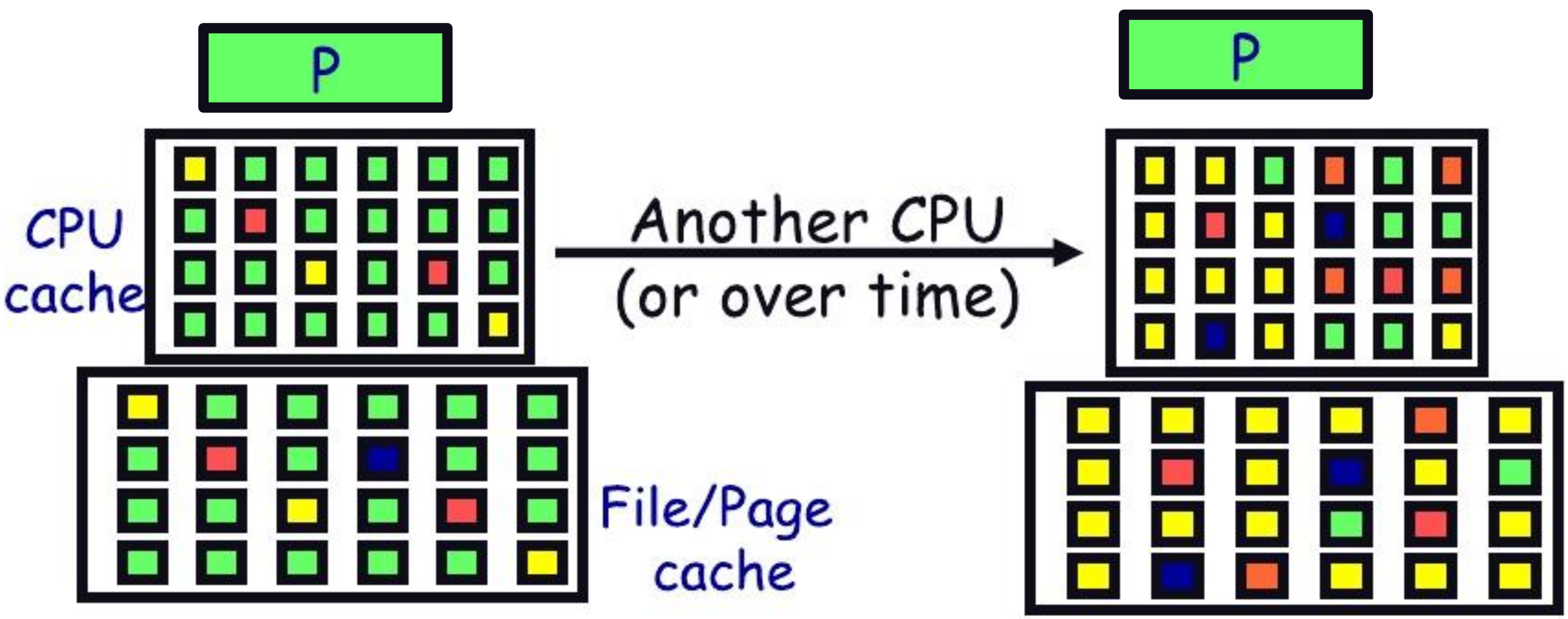
RR disadvantages

- **Varying sized jobs are good**
  . . . but what about same-sized jobs?
- **Assume 2 jobs of time=100 each:**

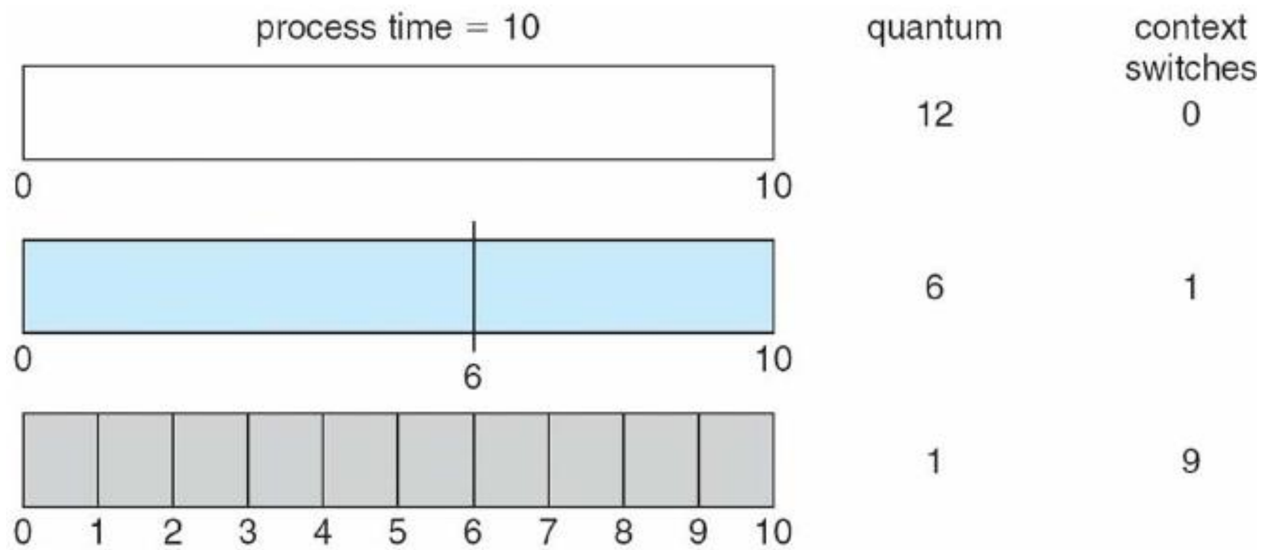
  - What is average completion time? (199+200)/2 = 199.5
  - How does that compare to FCFS? (100+200)/2 = 150.0

Context switch costs

- **What is the cost of a context switch?**
- **Brute CPU time cost in kernel**
  - Save and restore resisters, etc.
  - Switch address spaces (expensive instructions)
- **Indirect costs: cache, buffer cache, & TLB misses**

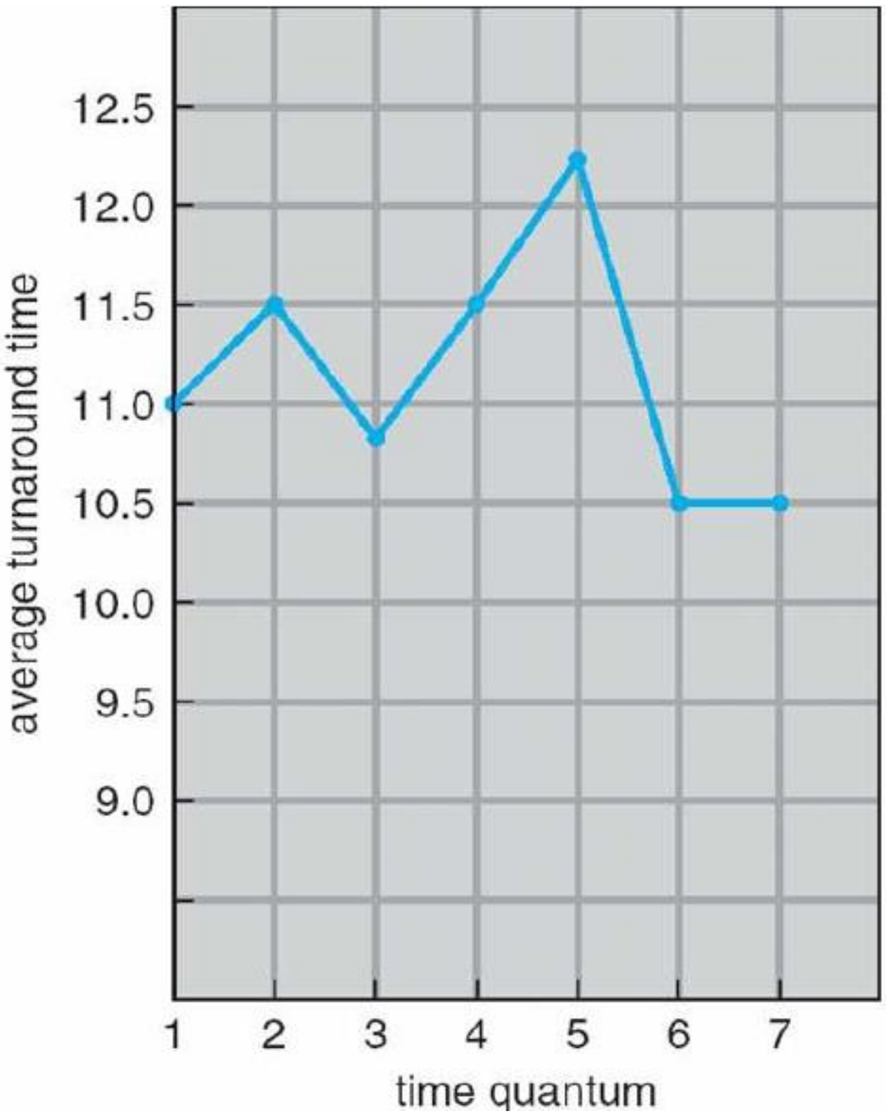


Time quantum



- **How to pick quantum?**
  - Want much larger than context switch cost
  - Majority of bursts should be less than quantum
  - But not so large system reverts to FCFS
- **Typical values: 10–100 msec**

Turnaround time vs. quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

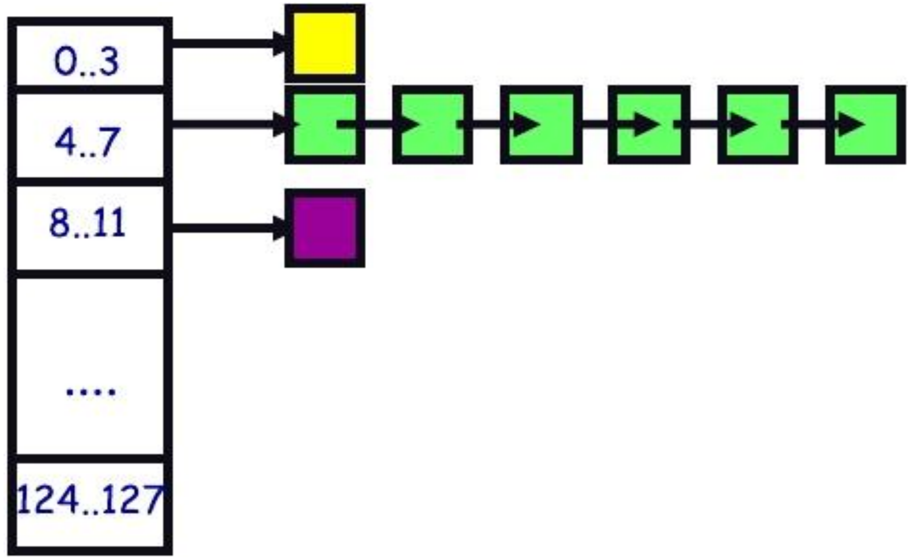
Two-level scheduling

- **Switching to swapped out process very expensive**
  - Swapped out process has most pages on disk
  - Will have to fault them all in while running
  - One disk access costs ~10ms. On 1GHz machine, 10ms = 10 million cycles!
- **Context-switch-cost aware scheduling**
  - Run in-core subset for "a while"
  - Then swap some between disk and memory
- **How to pick subset? How to define "a while"?**
  - View as scheduling *memory* before CPU
  - Swapping in process is cost of memory "context switch"
  - So want "memory quantum" much larger than swapping cost

Priority scheduling

- **Associate a numeric priority with each process**
  - E.g., smaller number means higher priority (Unix/BSD)
  - Or smaller number means lower priority (Pintos)
- **Give CPU to the process with highest priority**
  - Can be done preemptively or non-preemptively
- **Note SJF is a priority scheduling where priority is the predicted next CPU burst time**
- **Starvation – low priority processes may never execute**
- **Solution?**
  - **Aging - increase a process's priority as it waits**

Multilevel feeedback queues (BSD)



- **Every runnable process on one of 32 run queues**
  - Kernel runs process on highest-priority non-empty queue
  - Round-robins among processes on same queue
- **Process priorities dynamically computed**
  - Processes moved between queues to reflect priority changes
  - If a process gets higher priority than running process, run it
- **Idea: Favor interactive jobs that use less CPU**

Process priority (BSD model)

- p_nice – **user-settable weighting factor**
- p_estcpu – **per-process estimated CPU usage**
  - Incremented whenever timer interrupt found proc. running
  - Decayed every second when process runnable

$$\text{p\_estcpu} \leftarrow \left(\frac{2 \cdot \text{load}}{2 \cdot \text{load} + 1}\right) \text{p\_estcpu}$$

  - Load is sampled average of length of run queue plus short-term sleep queue over last minute
- **Set process priority by** **(lower p_usrpri = higher priority)**

$$\text{p\_usrpri} \leftarrow 50 + \left(\frac{\text{p\_estcpu}}{4}\right) + 2 \cdot \text{p\_nice}$$

(value clipped if over 127)

Sleeping process increases priority

- p_estcpu **not updated while asleep**
  - Instead p_slptime keeps count of sleep time
- **When process becomes runnable**

$$\text{p\_estcpu} \leftarrow \left(\frac{2 \cdot \text{load}}{2 \cdot \text{load} + 1}\right)^{\text{p\_slptime}} \cdot \text{p\_estcpu}$$

  - Approximates decay ignoring nice and past loads
- **Previous description based on [McKusick]**[a]

[a] See library.stanford.edu for off-campus access

Pintos notes

- **Same basic idea for second half of project 1**
  - But 64 priorities, not 128
  - Higher numbers mean higher priority
  - Okay to have only one run queue if you prefer (less efficient, but we won't deduct points for it)
- **Have to negate priority equation:**

$$\text{priority} = 63 - \left(\frac{\text{recent\_cpu}}{4}\right) - 2 \cdot \text{nice}$$

Limitations of BSD scheduler

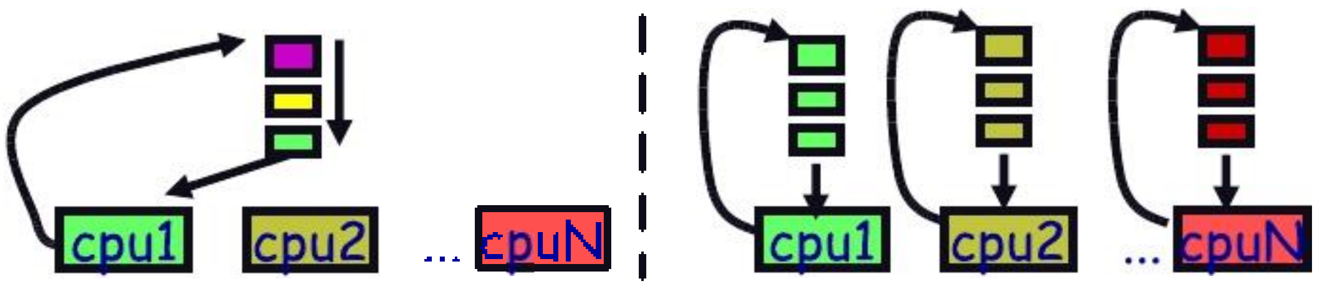
- **Hard to have isolation / prevent interference**
  - Priorities are absolute
- **Can't donate priority (e.g., to server on RPC)**
- **No flexible control**
  - E.g., In Monte Carlo simulations, error is 1/sqrt(N) after N trials
  - Want to get quick estimate from new computation
  - Leave a bunch running for a while to get more accurate results
- **Multimedia applications**
  - Often fall back to degraded quality levels depending on resources
  - Want to control quality of different streams

Real-time scheduling

- **Two categories:**
  - *Soft real time*—miss deadline and CD will sound funny
  - *Hard real time*—miss deadline and plane will crash
- **System must handle periodic and aperiodic events**
  - E.g., procs A, B, C must be scheduled every 100, 200, 500 msec, require 50, 30, 100 msec respectively
  - *Schedulable* if $\sum \frac{\mathit{CPU}}{\text{period}} \leq 1$ (not counting switch time)
- **Variety of scheduling strategies**
  - E.g., earliest deadline first (EDF) (works if schedulable)

Multiprocessor scheduling issues

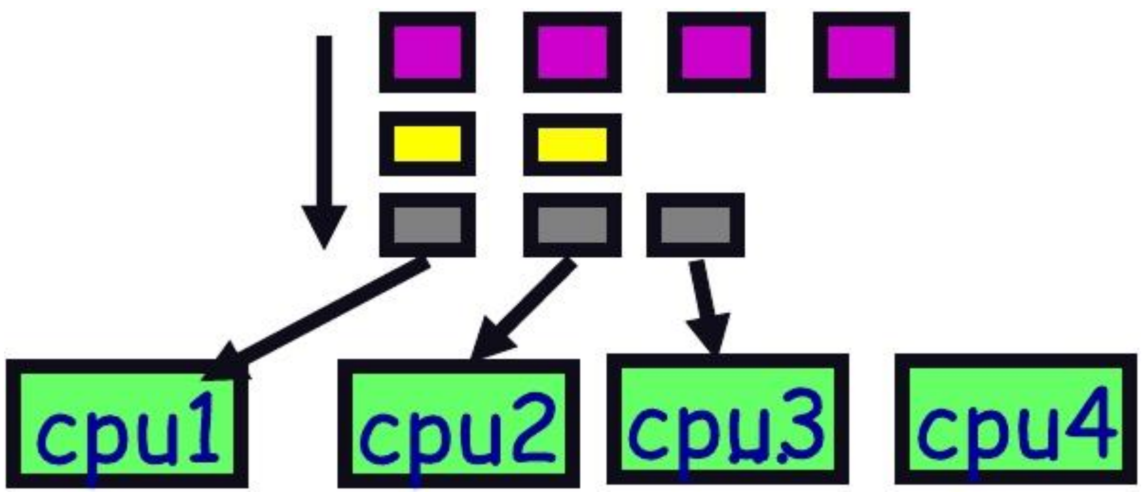
- **Must decide on more than which processes to run**
  - Must decide on which CPU to run which process
- **Moving between CPUs has costs**
  - More cache misses, depending on arch more TLB misses too
- ***Affinity scheduling*—try to keep threads on same CPU**



  - But also prevent load imbalances
  - Do *cost-benefit* analysis when deciding to migrate

Multiprocessor scheduling (cont)

- **Want related processes scheduled together**
  - Good if threads access same resources (e.g., cached files)
  - Even more important if threads communicate often, otherwise must context switch to communicate
- ***Gang scheduling*—schedule all CPUs synchronously**
  - With synchronized quanta, easier to schedule related processes/threads together



Thread scheduling

- **With thread library, have two scheduling decisions:**
  - *Local Scheduling* – Thread library decides which user thread to put onto an available kernel thread
  - *Global Scheduling* – Kernel decides which kernel thread to run next
- **Can expose to the user**
  - E.g., pthread_attr_setscope allows two choices
  - PTHREAD_SCOPE_SYSTEM – thread scheduled like a process (effectively one kernel thread bound to user thread – Will return ENOTSUP in user-level pthreads implementation)
  - PTHREAD_SCOPE_PROCESS – thread scheduled within the current process (may have multiple user threads multiplexed onto kernel threads)

Thread dependencies

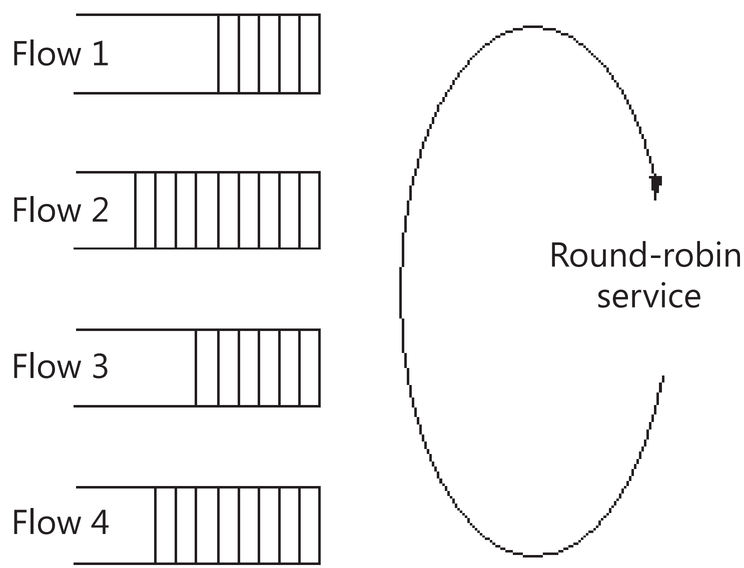
- **Say *H* at high priority, *L* at low priority**
  - *L* acquires lock *l*.
  - Scene 1: *H* tries to acquire *l*, fails, spins. *L* never gets to run.
  - Scene 2: *H* tries to acquire *l*, fails, blocks. *M* enters system at medium priority. *L* never gets to run.
  - Both scenes are examples of ***priority inversion***
- **Scheduling = deciding who should make progress**
  - Obvious: a thread's importance should increase with the importance of those that depend on it.
  - Naive priority schemes violate this.

Priority donation

- **Say higher number = higher priority**
- **Example 1:** L **(prio 2),** M **(prio 4),** H **(prio 8)**
  - L holds lock l
  - M waits on l, L's priority raised to $L' = \max(M, L) = 4$
  - Then H waits on l, L's priority raised to $\max(H, L') = 8$
- **Example 2: Same threads**
  - L holds lock l, M holds lock l_2
  - M waits on l, L's priority now $L' = 4$ (as before)
  - Then H waits on l_2. M's priority goes to $M' = \max(H, M) = 8$, *and* L's priority raised to $\max(M', L') = 8$
- **Example 3:** L **(prio 2),** $M_1, \ldots M_{1000}$ **(all prio 4)**
  - L has l, and $M_1, \ldots, M_{1000}$ all block on l. L's priority is $\max(L, M_1, \ldots, M_{1000}) = 4$.

Fair Queuing (FQ)

- **Digression: packet scheduling problem**
  - Which network packet should router send next over a link?
  - Problem inspired some algorithms we will see next week
  - Plus good to reinforce concepts in a different domain. . .
- **For ideal fairness, would send one bit from each flow**
  - In weighted fair queuing (WFQ), more bits from some flows



Packet scheduling

- **Differences from CPU scheduling**
  - No preemption or yielding—must send whole packets
    - Thus, *can't* send one bit at a time
  - But know how many bits are in each packet
    - Can see the future and know how long packet needs link
- **What scheduling algorithm does this suggest?**

Packet scheduling

- **Differences from CPU scheduling**
  - No preemption or yielding—must send whole packets
    - Thus, *can't* send one bit at a time
  - But know how many bits are in each packet
    - Can see the future and know how long packet needs link
- **What scheduling algorithm does this suggest? SJF**
- **Recall limitations of SJF:**
  - Can't see the future
    - solved by packet length
  - Optimizes response time, not turnaround time
    - but these are the same when sending whole packets
  - Not fair
- **Kind of want fair SJF for networking**

FQ Algorithm

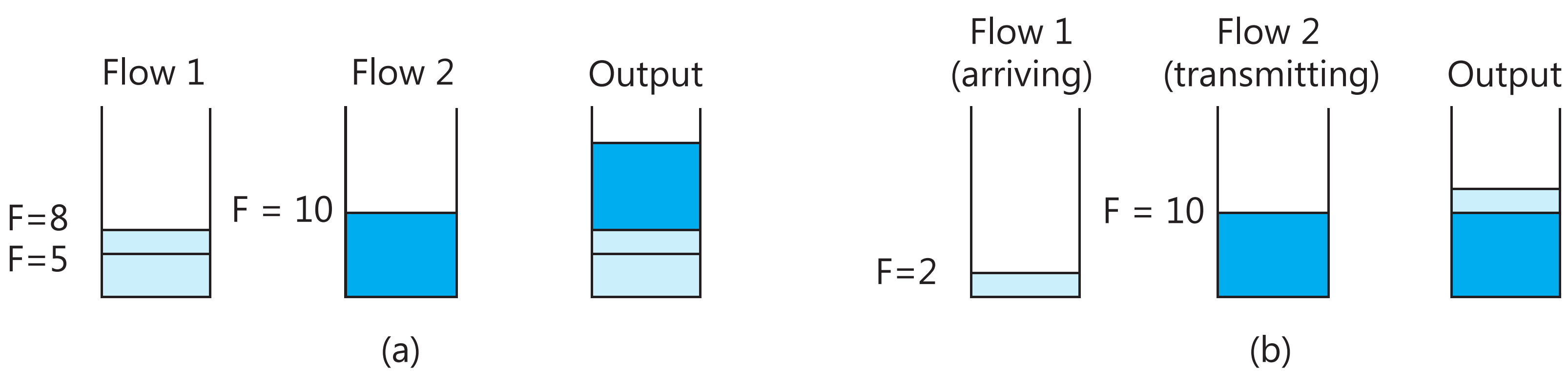
- **Suppose clock ticks each time a bit is transmitted**
- **Let** P_i **denote the length of packet** i
- **Let** S_i **denote the time when start to transmit packet** i
- **Let** F_i **denote the time when finish transmitting packet** i

- $F_i = S_i + P_i$
- **When does router start transmitting packet** i**?**
  - If arrived before router finished packet $i - 1$ from this flow, then immediately after last bit of $i - 1$ (F_{i-1})
  - If no current packets for this flow, then start transmitting when arrives (call this A_i)

- **Thus:** $F_i = \max(F_{i-1}, A_i) + P_i$

FQ Algorithm (cont)

- **For multiple flows**
  - Calculate F_i for each packet that arrives on each flow
  - Treat all F_i s as timestamps
  - Next packet to transmit is one with lowest timestamp
- **Example:**



Summary

- Read Ch. 1-6
- Processes and Threads (Ch. 4)
- Process Scheduling (Ch. 5)
- Synchronization (Ch. 6)
- Project 1 – Scheduling and Synchronization