CSE 120
Operating Systems Principles

Spring 2025

Lecture 8: CPU Scheduling

Amy Ousterhout

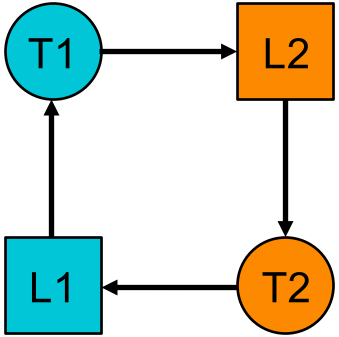
Today's Outline

- Deadlock
  - What can go wrong with concurrency?
  - What can we do about it?
- CPU Scheduling
  - What are our goals with scheduling?
  - What scheduling algorithms can we use?

Deadlock

- Deadlock exists among a set of threads if every thread is waiting for an event that can only be caused by another thread in the set

Dining Philosophers



threads holding locks

deadlocked traffic

Conditions for Deadlock

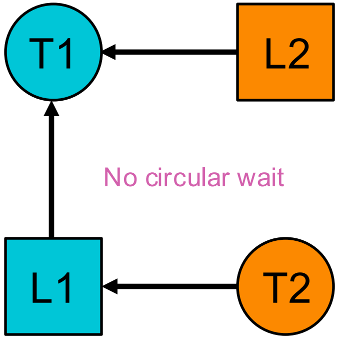
- Deadlock can exist if an only if the following conditions hold simultaneously:
  - Mutual exclusion: a resource is assigned to at most one thread at once
  - Hold and wait: threads holding resources can request new resources while continuing to hold old resources
  - No preemption: resources cannot be taken away once obtained
  - Circular wait: one thread waits for another in a circular fashion
- Eliminating any condition eliminates deadlock!

Strategies for Dealing with Deadlock

- Ignore the problem
  - Ostrich algorithm
- Prevention
  - Make it impossible for deadlock to happen
- Avoidance
  - Control allocation of resources
- Detection and Recovery
  - Look for a cycle in dependencies

Deadlock Prevention

- If we ensure that at least one of the conditions cannot occur, then deadlock is impossible
  - No mutual exclusion
    - » Make resources sharable (not always possible)
  - No hold and wait
    - » Threads cannot hold one resource while requesting another
    - » Threads try to lock all resources at once at the beginning
  - Preemption
    - » OS can preempt resources
  - No circular wait
    - » Impose an order on all resources, request in order
    - » Popular OS implementation technique when using multiple locks



Deadlock Avoidance

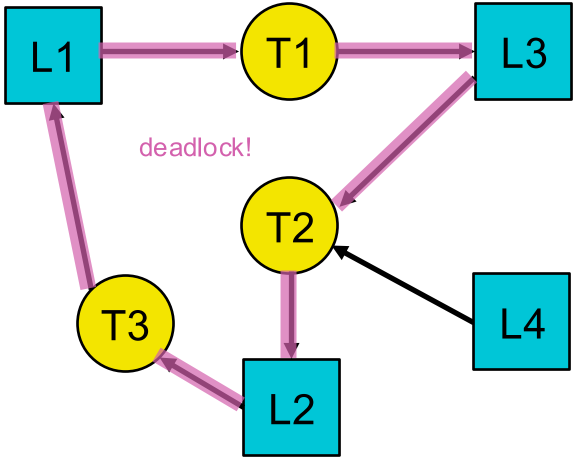
- Avoidance
  - Threads indicate in advance what resources they will need
  - System carefully schedules threads to ensure that deadlock is not possible
  - Avoids circular dependencies
- Banker's Algorithm
  - Only allocates resources if there is some scheduling order in which every thread can complete
- Avoidance is tough
  - Hard to determine all resources needed in advance
  - Fine theoretical problem, not as practical to use

Deadlock Detection and Recovery

- Detection and recovery
    - Allow deadlocks to happen but detect them and recover
- To do this, we need two algorithms:
    - One to determine whether a deadlock has occurred
    - Another to recover from the deadlock

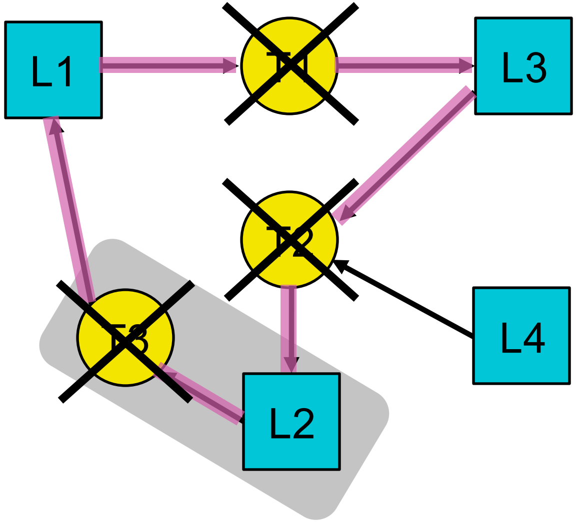
Deadlock Detection

- Detection
  - Traverse the resource graph looking for cycles
- Expensive
  - Many threads and resources to traverse
- Invoke detection algorithm depending on:
  - How often or likely deadlock is
  - How many threads are likely to be affected when it occurs



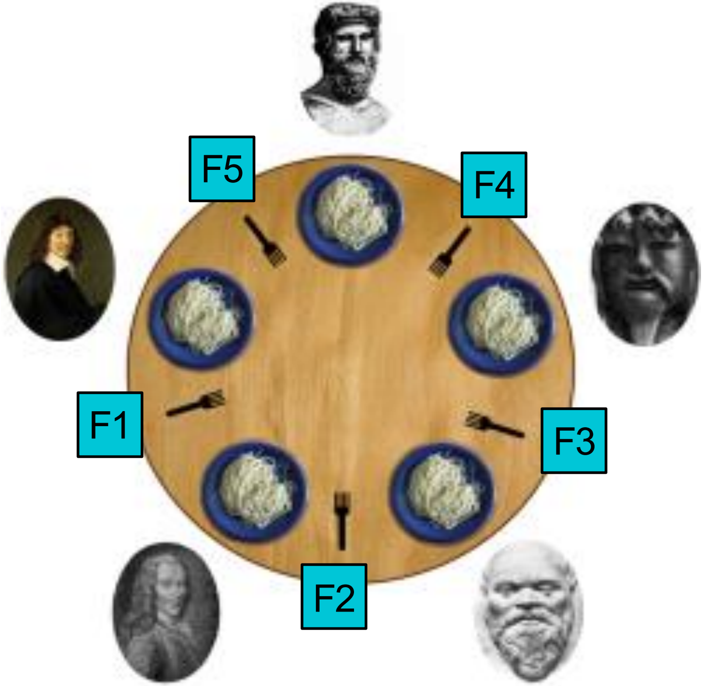
Deadlock Recovery

- Once a deadlock is detected, we have two options:
  - Abort threads
    - » Abort all deadlocked threads – threads need to start over again
    - » Abort one thread at a time until the cycle is eliminated – system needs to rerun detection after each abort
  - Preempt resources (force their release)
    - » Need to select thread and resource to preempt
    - » Need to roll back thread to previous state



Dining Philosophers' Problem

- How can we solve this problem?
- Which of the 4 approaches should we take?
- One solution:
  - Prevention
  - Ensure no circular wait
  - Assign a number to each fork
  - Acquire forks in increasing order



Deadlock Summary

- Deadlock occurs when threads are waiting on each other and cannot make progress
  - Cycles in the Resource Allocation Graph
- Deadlock requires 4 conditions:
  - Mutual exclusion, hold and wait, no resource preemption, circular wait
- Four approaches to dealing with deadlock:
  - Ignore it – live life on the edge
  - Prevention – make one of the 4 conditions impossible (by programmer, usually)
  - Avoidance – carefully control allocation (by the OS with programmer help)
  - Detection and Recovery – look for a cycle, then preempt or abort (by the OS)

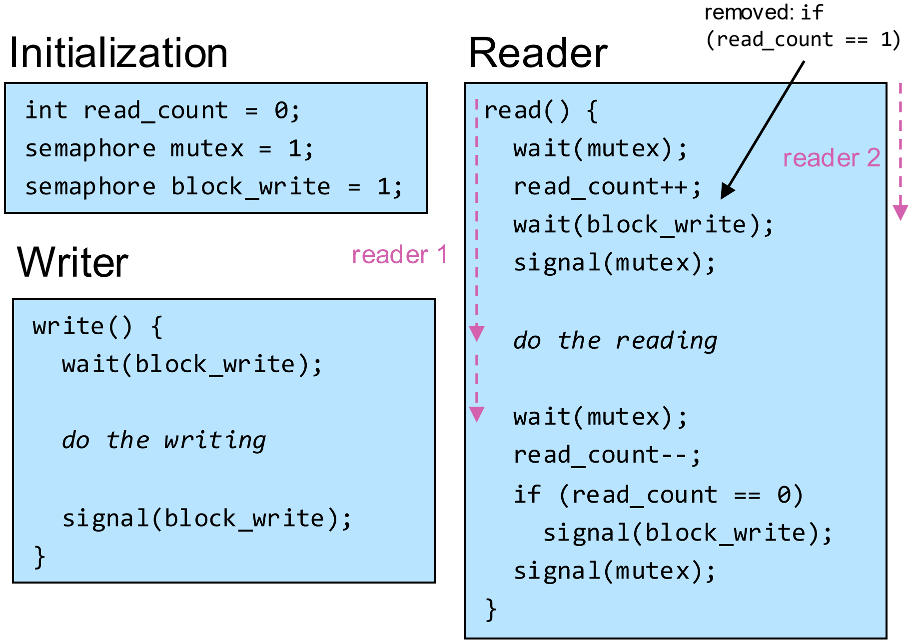
Poll – Atomicity

- Which of the following operations is not atomic?
  - A: `x++` not atomic – typically implemented with three instructions (load, increment, store)
  - B: `test_and_set` atomic instruction that both reads and sets a value
  - C: `MOV` instruction on x86 (used to load from or store to memory) atomic – single instruction
  - D: calling `signal()` on a semaphore atomically wakes up a queued thread and increments the counter
  - E: disabling interrupts atomic – done with a single instruction (e.g., `CLI` in x86)

Atomic: happens completely or not at all,
can't observe an in-between state

Poll – Readers-Writers with Semaphores

- Consider the solution to the Readers-Writers problem at right.
- What could happen if we removed `` `if (read_count == 1)` `` so that readers always called `wait(block_write)`?
  - A: multiple writers could write at once
  - B: a reader and writer could access the data concurrently
  - C: there could be a race condition when updating `read_count`
  - D: deadlock
  - E: the code would work as intended



Initialization

```
int read_count = 0;
semaphore mutex = 1;
semaphore block_write = 1;
```

Writer

```
write() {
   wait(block_write);

   do the writing

   signal(block_write);
}
```

Reader

removed: if (read_count == 1)

```
read() {
   wait(mutex);
   read_count++;
   wait(block_write);
   signal(mutex);

   do the reading

   wait(mutex);
   read_count--;
   if (read_count == 0)
     signal(block_write);
   signal(mutex);
}
```

Today's Outline

- Deadlock
    - What can go wrong with concurrency?
    - What can we do about it?
- CPU Scheduling
    - What are our goals with scheduling?
    - What scheduling algorithms can we use?

Separation of Policy and Mechanism

- Mechanism: tool that achieves some effect
- Policy: decision about what effect should be achieved
- Example: card keys instead of physical keys
- Separation leads to flexibility!

CPU Scheduling

- Share CPU resources (across processes or threads) by time-slicing the CPU
- Processing illusion: every process thinks it owns the CPU

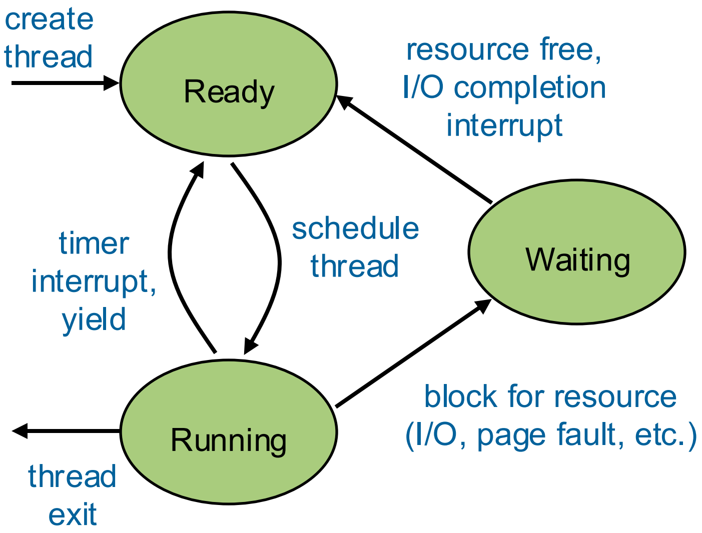
CPU Scheduling – Policy vs. Mechanism

```
yield() {
    thread_t old_thread = current_thread;
    current_thread = get_next_thread();  ←— policy
    append_to_queue(ready_queue, old_thread);
    context_switch(old_thread, current_thread);  ←— mechanism
    return;
}
```

- CPU scheduling mechanisms
  - Context switching – saving state of old thread and restoring state of new thread
  - Thread queues and thread states
  - Timer interrupts
- CPU scheduling policies
  - Which thread should we run next and for how long?

CPU Scheduler

- The scheduler (or dispatcher) is the module that moves threads between queues and states
  - Let a thread run for a while
  - Save its execution state
  - Load state of another thread
  - Let it run…
- When does the scheduler run? When…
  - A thread switches from running to waiting or ready
  - A thread is terminated
  - An interrupt or exception occurs



CPU Scheduling Policies

- The scheduling algorithm (aka policy) determines which thread to run
    - Which thread should we run next?
    - How long should we run it for?
- Today we’ll discuss:
    - Goals of CPU scheduling
    - Well-known CPU scheduling algorithms (or policies)
- We’ll refer to schedulable entities as jobs
    - These could be processes, threads, people, etc.

Scheduling Goals

- Scheduling algorithms can have many different goals:
  - Minimize average turnaround time
    - Time to complete a job: $T_{turnaround} = T_{completion} - T_{arrival}$
  - Maximize throughput
    - Jobs per second
    - Minimize overhead (e.g., of context switches)
    - Use system resources efficiently (CPU, memory, disk, etc.)
  - Minimize average response time
    - Time until a job starts: $T_{response} = T_{firstrun} - T_{arrival}$
  - Fairness
    - No starvation, no deadlock, fair access to the CPU

Application Goals

- Different applications may have different goals
- Batch applications
  - E.g., training machine learning models, large scientific simulations
  - Strive for high job throughput, low turnaround time
- Interactive applications
  - E.g., Zoom, your browser
  - Strive for low response time

Starvation: A Non-Goal

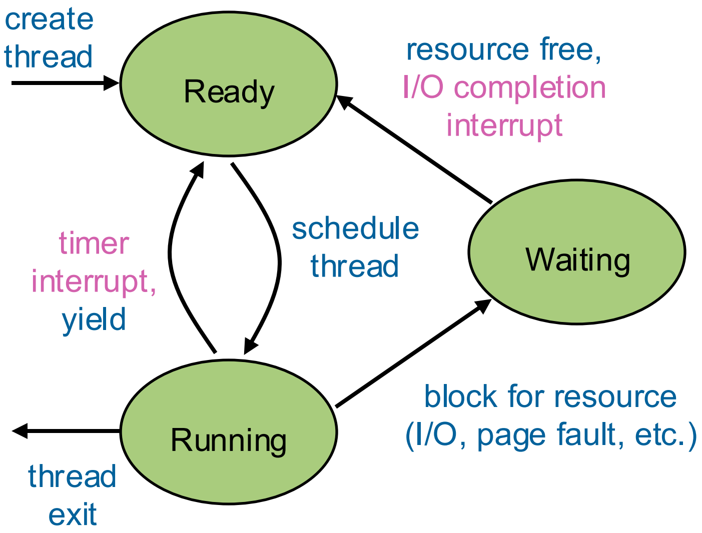
- Starvation: a situation in which a job is prevented from making progress because some other job has the resource it requires
  - Resource could be the CPU or a lock
- Starvation is usually a side effect of the scheduling algorithm
  - E.g., a high priority process always prevents a low priority process from running
- Starvation can be a side effect of synchronization
  - E.g., constant supply of readers blocks out any writers

Scheduling Challenges

- Jobs can have different run times
- Jobs can arrive at different times
- The scheduler can interrupt jobs
- Jobs can use other resources besides the CPU (e.g., I/O)
- The run time of each job may not be known ahead of time

Preemptive vs. Non-Preemptive Scheduling

- Jobs can be scheduled preemptively or non-preemptively
  - Preemptive: the scheduler can interrupt a running job
  - Non-preemptive: the scheduler waits for a job to explicitly block

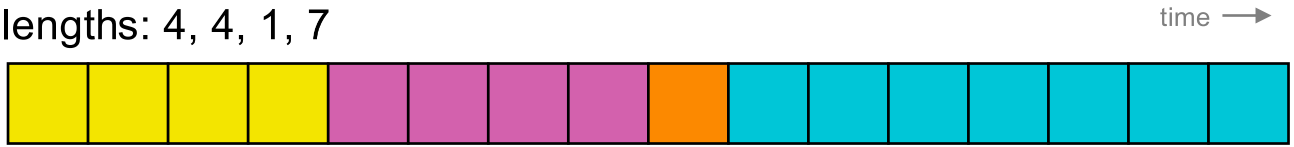


Scheduling Policies

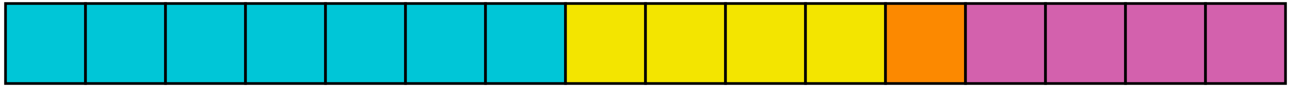
- First-come first-served (FCFS) or first-in first-out (FIFO)
- Shortest job first (SJF)
- Shortest remaining time to completion first (SRTCF)
- Round robin
- Priority scheduling
- Multi-level feedback queues (MLFQ)

First-Come First-Served (FCFS) Policy

- First-come first-served (FCFS) or first-in first-out (FIFO)
  - Schedule jobs in the order they arrive
  - Non-preemptive – run them until completion or they block or yield
- Jobs all arrive at the beginning
  - Job lengths: 4, 4, 1, 7



  - Job lengths: 7, 4, 1, 4



run jobs in whatever order they arrive in

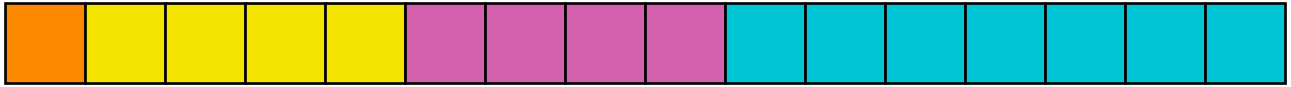
- Pros: simplicity, jobs treated equally, no starvation
- Con: average waiting time can be large if short jobs wait behind long jobs

Shortest Job First (SJF)

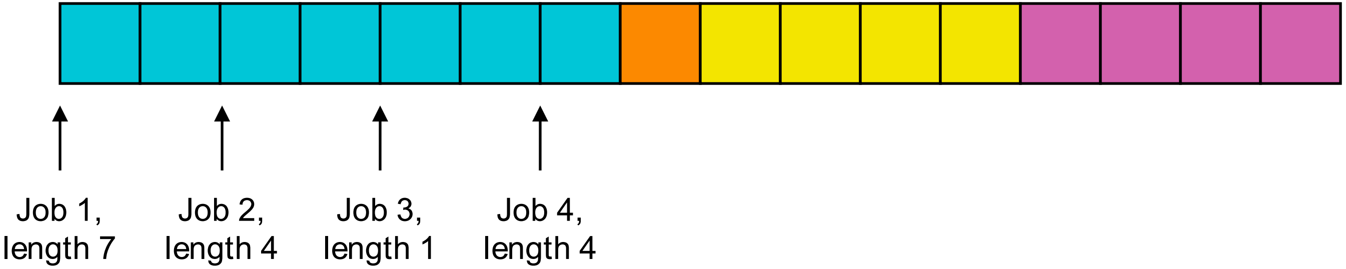
- Shortest job first (SJF)
    - Run the job with the shortest run time first
    - Non-preemptive

Shortest Job First (SJF) Examples

- Jobs arrive all at the beginning
  - Job lengths: 7, 4, 1, 4



- Jobs arrive over time
  - Average turnaround time = (7 + 10 + 4 + 10) / 4 = 7.75s

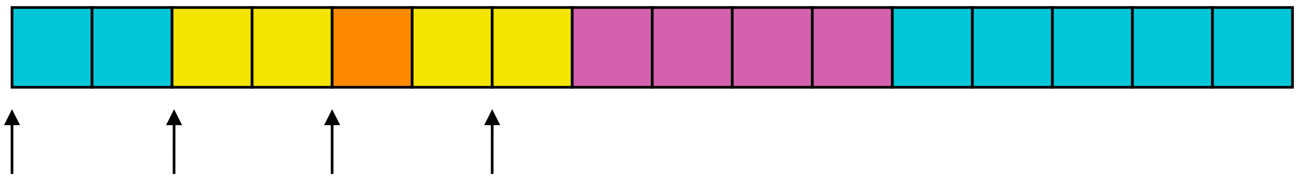


Shortest Job First (SJF)

- Shortest job first (SJF)
  - Run the job with the shortest run time first
  - Non-preemptive
- How do we know how long a job runs for?
- Pro: minimizes average turnaround time if all jobs arrive at the beginning
- Cons:
  - Difficult to predict run times
  - Can’t preempt long jobs
  - Can potentially starve long jobs

Shortest Remaining Time to Completion First (SRTCF)

- Shortest remaining time to completion first (SRTCF)
  - Run the job with the shortest remaining run time first
  - Preemptive
- Jobs arrive over time
  - Average turnaround time = (16 + 5 + 1 + 5) / 4 = 6.75s



compared to 7.75s without preemption (SJF)

Job 1, length 7

Job 2, length 4

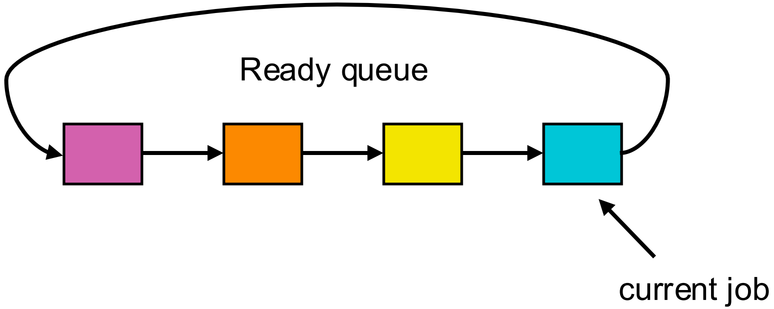
Job 3, length 1

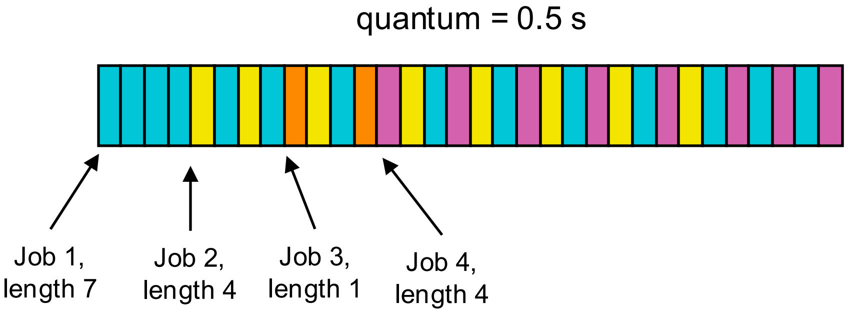
Job 4, length 4

- Pro: provably optimal – minimizes average turnaround time
- Cons: difficult to predict run times, can potentially starve long jobs

Round Robin

- Round robin
    - Preemptive
    - Each job runs for a time slice or quantum (or until it blocks or is interrupted)
    - Ready queue is treated as a circular queue





- Pros: short response time, fair, no starvation
- Cons: context switches are frequent and can add overhead

FCFS vs. Round Robin – Example 1

- Jobs with equal run times
    - 10 jobs, each takes 100 seconds
- Which policy will result in lower average turnaround time?
- FCFS
    - Job 1: 100s, job 2: 200s, … , job 10: 1000s
    - Average turnaround time = (100 + 200 + … + 1000) / 10 = 550s
- Round robin
    - Time slice 1 second and no overhead
    - Job 1: 991s, job2: 992s, … , job 10: 1000s
    - Average turnaround time = (991 + 992 + … + 999 + 1000) / 10 = 995.5s
- Round robin slows down all (but one) of the jobs!

FCFS vs. Round Robin – Example 2

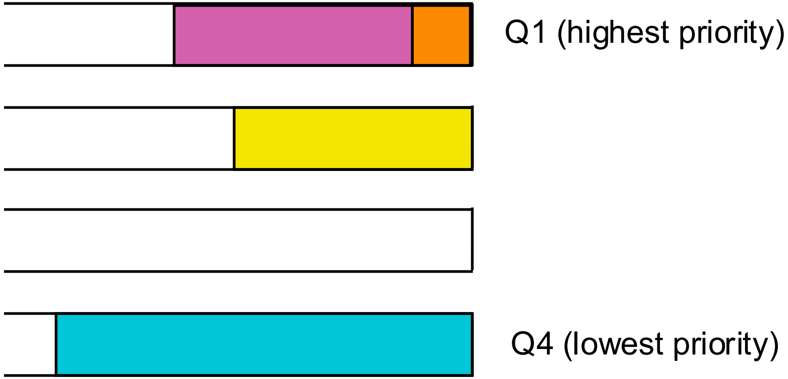
- When would round robin be a better choice?
- Jobs have different run times
  - 1 job takes 100 seconds, 9 jobs take 10 seconds
- FCFS
  - Job 1: 100s, job 2: 110s, … , job 10: 190s
  - Average turnaround time = (100 + 110 + … + 190) / 10 = 145s
- Round robin
  - Time slice 1 second and no overhead
  - Job 1: 190s, job 2: 92s, … , job 9: 99s, job 10: 100s
  - Average turnaround time = (190 + 92 + … + 99 + 100) / 10 = 105.4s
- Round robin is faster on average in this example

Priority Scheduling

- Priority scheduling
  - Assign each job a priority
  - Run the job with the highest priority first
    - Use FIFO for jobs with equal priority
  - Can be preemptive or non-preemptive
- Pros: flexibility
- Cons:
  - Starvation – low priority jobs can wait indefinitely
  - Who sets the priorities?
    - Internally by the OS
    - Externally by users or an administrator

Multi-level Feedback Queues (MLFQ)

- Multi-level feedback queues (MLFQ)
  - Multiple queues, each with a different priority
  - Jobs start at highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn't expire (or a job doesn't run), stay or move up one level
- Pros:
  - Dynamically adapts priorities
  - No starvation
- Cons: more complex, parameters to tune



Handling I/O

- Modern time-sharing OSes (Unix, Windows, …) time slice threads on the ready list
    - A CPU-bound thread may use its entire quantum (e.g., 1 ms)
    - An IO-bound thread might only use part (e.g., 100 μs) then issue IO
    - The IO-bound thread will go on a wait queue, goes back on the ready list when the IO completes

Scheduling Overhead

- Operating system aims to minimize overhead
    - Context switching it not doing useful work, it’s just overhead
    - Overhead includes making a scheduling decision + context switch
- Typical scheduling quantum: 1 ms
- Typical context-switch time: 1 μs

CPU Utilization

- CPU Utilization is the fraction of time the system spends doing useful work
    - Time doing useful work / total time
- Quantum of 1 ms + context-switch overhead of 1 µs
- Example: 3 CPU-bound jobs
    - Steady state: 1 ms + 1 µs + 1 ms + 1 µs + 1 ms + 1 µs…
    - CPU utilization: (3 * 1ms) / (3 * 1 ms + 3 * 1 µs) = 99.9%
- Example: 3 IO-bound jobs
    - IO-bound jobs don't use the full quantum
    - Steady state: 20 µs + 1 µs + 20 µs + 1 µs + 20 µs + 1 µs…
    - CPU utilization: (3 * 20 µs) / (3 * 20 µs + 3 * 1 µs) = 95.2%

Scheduling in Practice

- Additional challenges
  - Multiple CPU cores – should we schedule them together or independently?
  - Scheduling over groups of threads or processes
  - Generality – supporting many different kinds of workloads
- MacOS – Multilevel Feedback Queue
- Windows – Multilevel Feedback Queue
- Linux – Completely Fair Scheduler

Scheduling Summary

- Scheduler (dispatcher) gets invoked to handle context switches
  - Policy: which thread/process to run next
  - Mechanism: how to switch between threads/processes
- Many potential goals of scheduling algorithms
  - Utilization, throughput, turnaround time, response time, fairness
- Many possible policies
  - FCFS, SJF, SRTCF, Round robin, Priority, MLFQ

Midterm

- In class on Thursday 5/1
- Includes all the material so far (including today)
  - Lectures, homework, and programming projects
- An example exam is on the course website
- You may bring one 8.5”x11” double-sided sheet of notes to the exam
  - Handwritten or printed
- You must bring your ID to the exam
  - UCSD or government-issued ID
- Review session tomorrow
  - 4:00-5:30 pm in Peterson 110

Upcoming Tasks

- Midterm Review
  - Friday 4/25, 4:00-5:30 pm in Peterson 110
- Homework 2
  - Due Friday 4/25 at 11:59 pm
- Project 1
  - Due Tuesday 4/29 at 11:59 pm
- Midterm
  - In class on 5/1

- Next week:
  - Tuesday 4/29: no class
  - Tuesday 4/29: virtual office hours
    - 4:00-5:00 pm
  - Wednesday 4/30: no office hours
  - Thursday 5/1: office hours and midterm
  - See Piazza for details