CS111ACE Week 4
Scheduling & Deadlock

Attendance: https://tinyurl.com/CS111A-coin
Pollev for Anon Q+A: https://pollev.com/tripmaster419
Game Plan

- Scheduling
- Deadlock
Icebreaker

What is one thing you’re grateful for this quarter?
Scheduling Algorithms

- What’s the difference between the **dispatcher** and the **scheduler**?
Scheduling Algorithms

- What’s the difference between the **dispatcher** and the **scheduler**?
  - The **scheduler** is the part of the OS that decides which thread to run next. The **dispatcher** is the part of the OS that actually performs the context switch.
Scheduling Algorithms (Single Core)

- There are a few different ways we can schedule threads:
Scheduling Algorithms (Single Core)

- There are a few different ways we can schedule threads:
  - Round Robin: All threads run for the same amount of time. We run threads for a small (but even) “time slice,” and forcibly remove them when the time is up.

We call this kind of scheduling algorithm “preemptive” because the OS must interrupt switch (preempt) the running thread every time slice.
Scheduling Algorithms (Single Core)

- There are a few different ways we can schedule threads:
  - FIFO (First-In-First-Out): Queue up all “ready” threads in a First-In-First-Out (queue) structure. When we dispatch a thread, let it run to completion (or yield)

We call this kind of scheduling algorithm "cooperative" because threads are responsible for giving up CPU time to other threads (i.e., the OS won’t ensure that we have fairness, so threads need to cooperate)
Scheduling Algorithms (Single Core)

- There are a few different ways we can schedule threads:
  - SRPT (Shortest Remaining Processing Time): Always dispatch the thread with the minimum required processing time.
Scheduling Algorithms (Single Core)

- There are a few different ways we can schedule threads:
  - SRPT (Shortest Remaining Processing Time): Always dispatch the thread with the minimum required processing time.

This looks great! What might be problematic about this approach?
There are a few different ways we can schedule threads:

- SRPT (Shortest Remaining Processing Time): Always dispatch the thread with the minimum required processing time.

This looks great! What might be problematic about this approach?

It’s not actually possible to implement perfect SRPT. Instead, the OS can monitor the blocking/running patterns of threads to determine which threads will need more or less time on the CPU!
Scheduling Algorithms (Single Core)

- There are a few different ways we can schedule threads:
  - Round Robin: All threads run for the same amount of time. We run threads for a small (but even) “time slice,” and forcibly remove them when the time is up.
  - FIFO (First-In-First-Out): Queue up all “ready” threads in a First-In-First-Out (queue) structure. When we dispatch a thread, let it run to completion (or yield)
  - SRPT (Shortest Remaining Processing Time): Always dispatch the thread with the minimum required processing time.

Questions?
Scheduling Algorithms (Single Core)

- There are a few different ways we can schedule threads:
  - Round Robin: All threads run for the same amount of time. We run threads for a small (but even) “time slice,” and forcibly remove them when the time is up.
  - FIFO (First-In-First-Out): Queue up all “ready” threads in a First-In-First-Out (queue) structure. When we dispatch a thread, let it run to completion (or yield)
  - SRPT (Shortest Remaining Processing Time): Always dispatch the thread with the minimum required processing time.

Quiz Time: Given the following scenarios, which scheduling algorithm yield the optimal average completion time? There can be multiple answers for each!
Scheduling Algorithms (Single Core)

- There are a few different ways we can schedule threads:
  - Round Robin: All threads run for the same amount of time. We run threads for a small (but even) “time slice,” and forcibly remove them when the time is up.
  - FIFO (First-In-First-Out): Queue up all “ready” threads in a First-In-First-Out (queue) structure. When we dispatch a thread, let it run to completion (or yield)
  - SRPT (Shortest Remaining Processing Time): Always dispatch the thread with the minimum required processing time.

1. You have 100 incoming tasks, all with roughly the same amount of required CPU time
2. You have 100 incoming tasks, all with various required CPU times
3. You have 100 incoming tasks which have LOTS of I/O’s and not much computation
4. You have a constant stream of tasks. Half require very little CPU time, half require lots.
Scheduling Algorithms (Priority)

- Modern Schedulers keep track of thread priorities. Thread priorities indicate how urgently these threads need to run.
Scheduling Algorithms (Priority)

- Modern Schedulers keep track of thread priorities. Thread priorities indicate how urgently these threads need to run.
  - The OS determines priorities based on past behavior (trying to manifest SRPT). Threads with equal priorities will run in Round Robin.
Priority Scheduling Take 1: Multi-Level-Feedback-Queue

Q₀ (High priority)

Q₁

Q₂ (Low priority)

Great for any response time (pseudo-SEPT), starvation issues!

This is a preemptive scheduler

- If a thread completes, it's done
- If a thread blocks before it is preempted, move it up a queue
- If a thread is preempted (its time slice is used up), move it down a queue

Stanford University
Priority Scheduling Take 2: 4.4 BSD Scheduler

- This scheduling algorithm tracks each thread’s CPU usage and runs the thread with the least recently used CPU usage.
Priority Scheduling Take 2: 4.4 BSD Scheduler

- This scheduling algorithm tracks each thread’s CPU usage and runs the thread with the least recently used CPU usage.
  - Is this fair? Does this work for:
    - Threads that are “I/O” bound (low CPU usage but high priority).
    - Threads that are CPU bound (high CPU usage, not necessarily high priority).

This is quite different from MLFQ.
Priority Scheduling Take 2: 4.4 BSD Scheduler

- This scheduling algorithm tracks each thread’s CPU usage and runs the thread with the least recently used CPU usage
  - Is this fair? Does this work for:
    - Threads that are “I/O” bound (low CPU usage but high priority).
    - Threads that are CPU bound (high CPU usage, not necessarily high priority)
  - In the worst case (no threads get enough CPU time to finish), this system devolves to Round Robin.
    - Fair, but could have high context switching overhead
Scheduling Algorithms (Multicore)

- Share scheduling data structures with all cores
  - What info do you need?
Scheduling Algorithms (Multicore)

- Share scheduling data structures with all cores
  - What info do you need?
    - Ready queue
    - Info about the currently running threads (why?)
Scheduling Algorithms (Multicore)

- Share scheduling data structures with all cores
  - What info do you need?
    - Ready queue
    - Info about the currently running threads (why?)

- Multicore scheduling issues:
  - Contention: The single ready queue is the bottleneck for scheduling. (As # cores scales, this gets really bad)
    - We could have ready queues per core, but this might not balance overall system workload that well.
  - Core Affinity: Moving a thread from one core to another is expensive. It is also ideal to schedule threads from the same process on a *single core*.
    - This will make more sense in a few weeks!
Scheduling Algorithms (Midterm Question)

Consider the following single-threaded processes, and their arrival times, CPU bursts and their priorities (a process with a higher priority number has priority over a process with lower priority number).

<table>
<thead>
<tr>
<th>Thread</th>
<th>Duration (ticks)</th>
<th>Arrival Time (ticks)</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Scheduling Algorithms (Midterm Question)

You will consider four scheduling algorithms: first-in-first-out (FIFO), round robin (RR), shortest remaining processing time (SRPT), and priority scheduling.

The following apply:

- All schedulers other than FIFO are preemptive.
- The time slice of the RR scheduler is 1.
- If a process arrives at time x, it can be scheduled immediately.
- If two processes arrive at the same time, they are inserted in the ready queue in the lexicographical order. For example, if B and C arrive at the same time, B is inserted first, and C second in the ready queue.
- In the case of the RR scheduler, a new arriving process is inserted at the end of the ready queue. When the RR slice expires, the currently running thread is added at the end of the ready list before any newly arriving threads.
- The RR scheduler ignores the process priorities.
- The SRPT scheduler uses priorities to break ties, i.e., if two processes have the same remaining time we schedule the one with the highest priority.
<table>
<thead>
<tr>
<th>Time</th>
<th>FIFO</th>
<th>Round Robin</th>
<th>SRPT</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Completion Time</td>
</tr>
</tbody>
</table>
Breather Slide
Announcements

- A3 is due this Thursday. The handout is a little bit hard to parse, so I’d *strongly suggest* starting soon and asking clarifying questions!
  - This may be one of the harder assignments of the quarter!
- The Section Leading application closes this Thursday at midnight (so many Thursday deadlines!!). Let me know if you have any questions about it :)
What does **static** mean in the context of classes?

**Static methods** - Yes, they're part of the class, but they aren't attached to an object (instance).

- # of vectors ever created is a **static** fact. It's not relative to any instance.

**Static variable** - A variable whose is **shared** among all instances of the class.
A3 Advice

- What is an **initialization list**?
  - Say I had a **Car** class, and I wanted to make a new instance of a **Car**:

```cpp
Car::Car(std::string color, int mpg) {
    // color_ and mpg_ are the private
    // member of the CAR class.
    color_ = color;
    mpg_ = mpg;
}
```

Note that `color_` and `mpg_` are **private** variables defined in the **Car.hh** file.
A3 Advice

- What is an initialization list?
  - An equivalent constructor using an initialization list is the following:

```cpp
Car::Car(std::string color, int mpg) : color_(color), mpg_(mpg) {}  
```

Note that color_ and mpg_ are private variables defined in the Car.hh file.
Game Plan

- Scheduling
- Deadlock
Deadlock

- Deadlock occurs when you have multiple threads and multiple synchronization resources.
Deadlock

- Deadlock occurs when **you have multiple threads and multiple synchronization resources**.
  - Deadlock is **not** itself a race condition (as it does not involve reads and writes of shared data), but it is a bug that results from scheduling and locking order.
Deadlock

- Deadlock occurs when **you have multiple threads and multiple synchronization resources**.
  - Deadlock is **not** itself a race condition (as it does not involve reads and writes of shared data), but it is a bug that results from scheduling and locking order.
- In reality, deadlock occurs when you have these 4 conditions:
  - Mutual Exclusion
    - Resources cannot be shared (i.e. a one-way road, or a **mutex** that can only be locked by a **single** thread)
  - Circularity
    - There’s some kind of cycle in how our threads access resources (like dining philosophers in a circle)
  - No Preemption
    - No one (hint: the OS) can force a thread to give up the resource to fix the deadlock.
  - Hold + Wait
    - Threads ***hold onto their resources*** while waiting to acquire others
Deadlock

- As an aside, your multithreaded program can freeze (forever) for reasons that are not deadlock.
  - As we may have seen in our code example, threads must release their locks before completing.
Assume you have two threads (Thread A & Thread B), that both share 2 locks (L1 & L2)

- Can you give an example locking order between threads A and B that would cause deadlock?

**Bonus** -> What feature(s) of the 4 causes of deadlock is/are present here? How could you fix them?
Deadlock Prevention

- Although there are a few ways for an OS to try and prevent deadlock, they are somewhat impractical.
- The best way to prevent deadlock is as a user, to always have a consistent locking order.
  - If you have Locks A, B, and C, you could always lock in alphabetical order!
  - Does the order in which you unlock matter as well?

\[ \text{NO} \]
Deadlock - Midterm Question

Assume that every bank account has a single mutex associated with it (so that reads/writes are atomic). The transfer() function takes in mutexes for both accounts. Can deadlock occur if multiple accounts call transfer()? If so, how?

```c
void transfer(mutex &from, mutex &to, double amount){
    from.lock();
    to.lock();
    from.withdraw(amount);
    to.deposit(amount);
    to.unlock();
    from.unlock();
}
```
Takeaways

- **Scheduling** is a tradeoff between fairness, average completion time, and starvation
  - Although it’s impossible to *always* service the shortest remaining processing time (and it’s not a fair system), we can use past behavior to inform a thread’s future behavior (as seen in MLFQ)
- **Deadlock** occurs when threads acquire mutually exclusive, non-preemptible resources in a circular ordering.
  - Additionally, the threads must hold onto their resources while they wait to claim the other resources!
  - We can defeat deadlock as programmers by ensuring that our programs adhere to a *consistent* locking order.