Concurrency -
The scheduler, multiple cores, and locks

CS230 Tutorial 11
The Scheduler

- Decides who gets to use the CPU and when
  - Critical component of the operating system
  - Every operating system has its own scheduler

- Responsible for controlling the execution of threads
  - how to divide up resources among the threads to ensure “fair” execution of all the processes?
The Scheduler - Assumptions

In order to simplify talking about the scheduler, we’ll make the following assumptions:

- Every process runs for the same amount of time
- All processes are created simultaneously
- Processes only use the CPU (i.e. we’re ignoring IO)
- The runtime for each process is known in advance

As soon as we stop assuming these things, we need complicated Scheduling Algorithms to figure out optimal process running order.

**Reminder:** these assumptions are not actually true, but we pretend they are true for the purpose of discussing scheduling algorithms only.
Scheduling Algorithms - there are many

- **Shortest Job First**
  - schedule the jobs to run in the order of fastest to slowest
  - this makes sense if we assume that we know how long all the jobs are going to take, and receive them all at the same time

- **Shortest Time to Completion First (aka Preemptive Shortest Job First)**
  - now, relax the assumption that all jobs arrive at the same time
  - if a shorter job comes along, pause the current thread and run that one

- **Round Robin**
  - prevents thread starvation!
  - run each thread for the same amount of time, before pausing and moving to the next one
Scheduling Algorithms - Performance Metrics

There are various performance metrics:

- **Throughput**: number of threads finished execution per unit time

- **Latency**:
  - **Turnaround Time**: $T_{\text{done}} - T_{\text{arrived}}$ (arrived in queue)
  - **Response Time**: $T_{\text{firstrun}} - T_{\text{arrived}}$ (arrived in queue)

- **Fairness**: how fair is the CPU time allocation?

- **Wait time**: time spent ready but not running
Example: Threads are going to the “gym”, and they all want to ride a bike. For some reason the gym only has one bike, so it’s gonna be an awkward workout. If 4 threads want to bike for 10 mins, and they all arrive at the same time, how long would each thread spend at the gym if they use STCF? What about with RR? Compute avg response and turnaround times.
Scheduling Algorithms - Running Example

**Example:** Threads are going to the “gym”, and they all want to ride a bike. For some reason the gym only has one bike, so it’s gonna be an awkward workout. If 4 threads want to bike for 10 mins, and they all arrive at the same time, how long would each thread spend at the gym if they use STCF? What about with RR? Compute avg response and turnaround times.

**STCF:** Since all jobs take the same amount of time, the threads can run (bike :P ) in an arbitrary order. Label the threads A, B, C, D; assume they run in this order (WLOG).

<table>
<thead>
<tr>
<th>Job</th>
<th>Tcompletion</th>
<th>Tarrival</th>
<th>Tturnaround</th>
<th>Tfirstresponse</th>
<th>Tresponse</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>20</td>
<td>0</td>
<td>20</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>30</td>
<td>0</td>
<td>30</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>D</td>
<td>40</td>
<td>0</td>
<td>40</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>

Avg turnaround time = (10 + 20 + 30 + 40)/4 = 25 mins
Avg response time = (0 + 10 + 20 + 30)/4 = 15 mins
Scheduling Algorithms - Running Example

**Same example:** One gym, one bike, 4 threads (see previous slide).

**RR:** Assume 1 minute intervals. Now, the threads are going to alternate every minute so everyone gets a turn with the bike.

Bike:         A B C D A B C D A B C D A B C D A B C D A B C D A B C D A B C D A B C D
Minute:          1  5  9 13 17 21 25 29 33 37

<table>
<thead>
<tr>
<th>Job</th>
<th>Tcompletion</th>
<th>Tarival</th>
<th>Tturnaround</th>
<th>Tfirstresponse</th>
<th>Tresponse</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>37</td>
<td>0</td>
<td>37</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>38</td>
<td>0</td>
<td>38</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>39</td>
<td>0</td>
<td>39</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>40</td>
<td>0</td>
<td>40</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Avg turnaround time = (37 + 38 + 39 + 40)/4 = 38.5 mins
Avg response time = (0 + 1 + 2 + 3)/4 = 1.5 mins
Scheduling Algorithms - Summary

In summary:

- STCF and RR each have their place

- In the example we saw:
  - STCF is better for avg turnaround time
  - RR is better for response time

- There are tradeoffs between these approaches
Scheduling Algorithms - Multicore

How does this change if there are multiple cores?

Imagine in the previous example that we suddenly had 3 bikes. How would things change?

Now, we can run processes in parallel.
**STCF example: Multicore**

**STCF:** Since all jobs take the same amount of time, the threads can run (bike :P ) in an arbitrary order. Label the threads A, B, C, D; assume they run in this order (WLOG).

Now we have 3 bikes, 3 threads can start at the same time. So, A, B, and C start at the same time.

<table>
<thead>
<tr>
<th>Job</th>
<th>Tcompletion</th>
<th>Tarrival</th>
<th>Tturnaround</th>
<th>Tfirstresponse</th>
<th>Tresponse</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>10</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>20</td>
<td>0</td>
<td>20</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Avg turnaround time = \((10 + 10 + 10 + 20)/4 = 12.5\) mins
Avg response time = \((0 + 0 + 0 + 10)/4 = 2.5\) mins
RR example: Multicore

**RR:** Assume 1 minute intervals. Now, the threads are going to alternate every minute so as everyone gets a turn with the bike. Since there are 3 bikes, each thread gets 3 minutes on a bike, one minute off.

Bike 1: A D C B A D C B A D C B A D
Bike 2: B A D C B A D C B A D C B
Bike 3: C B A D C B A D C B A D C

Minute: 1 5 9 13

<table>
<thead>
<tr>
<th>Job</th>
<th>Tcompletion</th>
<th>Tarrival</th>
<th>Tturnaround</th>
<th>Tfirstresponse</th>
<th>Tresponse</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>13</td>
<td>0</td>
<td>13</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>13</td>
<td>0</td>
<td>13</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>13</td>
<td>0</td>
<td>13</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>14</td>
<td>0</td>
<td>14</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Avg turnaround time = (13 + 13 + 13 + 14)/4 = 13.25 mins
Avg response time = (0 + 0 + 0 + 1)/4 = 0.25 mins
Multicore - Summary

In summary:

- Clearly multicore is more efficient in terms of both response time and turnaround time.
- The downside: the hardware is more complicated. Now the scheduler has to work over multiple processors.
Locks

- Last week we covered two basic thread operations: create and join
- Locks are just another set of thread operations
  - If multiple threads access the same thing at the same time bad things happen
  - Need a way to control who can access shared variables
  - A lock is an object that only one thread can own at a time
  - Only access shared variable when you have the lock
- Two new thread operations: lock and unlock
  - lock: the current thread tries to get the lock
    - If it can’t get the lock, it blocks until it can get the lock
  - unlock: the current thread gives up the lock
  - The code between lock and unlock is called a critical section
Locks - Examples

Here we have an example of two threads that use a lock

- This will always print 3
- The threads might add 0+1+2=3 or 0+2+1=3
Deadlock

- Deadlock occurs when two or more threads are blocked waiting for each other in a loop
  - Ex: thread A is waiting for thread B
    thread B is waiting for thread C
    thread C is waiting for thread A
  - Ex: thread D is waiting for thread E
    thread E is waiting for thread D
  - The processes are stuck! They didn't explicitly crash, but they will never make progress.
Deadlock Examples

- Can the following code deadlock? If so, why?

```c
#define DEADLOCK EXAM PLE

#include <stdlib.h>
#include <stdio.h>

int main() {
    t1 = create(funcA, 0);
    t2 = create(funcB, 0);
    join(t1);
    join(t2);
}

int funcA(unused) {
    lock(q_lock);
    lock(r_lock);
    print("hello");
    unlock(r_lock);
    unlock(q_lock);
    return 0;
}

int funcB(unused) {
    lock(r_lock);
    lock(q_lock);
    print("pizza");
    unlock(q_lock);
    unlock(r_lock);
    return 0;
}
```
Deadlock Examples

- Can the following code deadlock? If so, why?

```c
global lock q_lock, r_lock;
global thread t1, t2;
main() {
    t1 = create(funcA, 0);
    t2 = create(funcB, 0);
    join(t1);
    join(t2);
}
```

```c
funcA(unused) {
    lock(q_lock);
    lock(r_lock);
    print("hello");
    unlock(r_lock);
    unlock(q_lock);
}
```

```c
funcB(unused) {
    lock(r_lock);
    lock(q_lock);
    print("pizza");
    unlock(q_lock);
    unlock(r_lock);
}
```

- Yes it can deadlock
  - If t1 goes first and locks q_lock, then gets preempted and t2 locks r_lock, then t2 can’t lock q_lock because t1 has it, and t1 can’t lock r_lock because t2 has it!
Deadlock Examples

- Can the following code deadlock? If so, why?

```c
int main() {
    thread t1, t2;
    t1 = create(funcA, 0);
    t2 = create(funcB, 0);
    join(t1);
    join(t2);
}

int funcA(int unused) {
    lock(main_lock);
    print("hello");
    join(t2);
    unlock(main_lock);
    return 0;
}

int funcB(int unused) {
    lock(main_lock);
    print("pizza");
    join(t2);
    unlock(main_lock);
    return 0;
}
```
Deadlock Examples

- Can the following code deadlock? If so, why?

```c
#include <pthread.h>

struct thread { int tid; }

int main() {
    pthread_t t1, t2;
    pthread_mutex_t main_lock;

    pthread_mutex_init(&main_lock, NULL);

    t1 = pthread_create(NULL, NULL, funcA, NULL);
    pthread_mutex_lock(&main_lock);
    print("hello");
    pthread_mutex_unlock(&main_lock);
    join(t1);
    t2 = pthread_create(NULL, NULL, funcB, NULL);
    join(t2);
    pthread_mutex_destroy(&main_lock);
}

void* funcA(void* unused) {
    pthread_mutex_lock(&main_lock);
    print("pizza");
    pthread_mutex_unlock(&main_lock);
    return NULL;
}

void* funcB(void* unused) {
    return NULL;
}
```

- Yes it can deadlock
  - If t1 goes first and locks `main_lock` and then waits for t2 to end, but t2 cannot end until it locks `main_lock` and t1 has `main_lock`, so t2 cannot get it!
Deadlock Examples

● Can the following code deadlock? If so, why?

```c
global lock q_lock, r_lock;
global thread t1, t2;
main() {
    t1 = create(funcA, 0);
    t2 = create(funcA, 0);
    join(t1);
    join(t2);
}

funcA(unused) {
    lock(q_lock);
    lock(r_lock);
    print("hello");
    unlock(q_lock);
    unlock(r_lock);
}
```
Deadlock Examples

- Can the following code deadlock? If so, why?

```c
global lock q_lock, r_lock;
global thread t1, t2;
main() {
    t1 = create(funcA, 0);
    t2 = create(funcA, 0);
    join(t1);
    join(t2);
}
funcA(unused) {
    lock(q_lock);
    lock(r_lock);
    print("hello");
    unlock(q_lock);
    unlock(r_lock);
}
```

- No it cannot deadlock
  - So long as all the threads lock in the same order, the order they unlock doesn’t matter
    - Unlocking having to be in reverse order of locking is a common misconception, in reality you can unlock in any order
Assignment reminders

● Submit a .txt XOR a .pdf for each question
  ○ Do not submit both for the same question!
  ○ You may submit a .pdf for one question and a .txt for a different question

● Make sure your diagrams and tables are clear and easy to read
  ○ Make sure to leave enough space