Software Systems

Day 21 - Concurrency, Semaphores, and Mutexes

(Software) Engineering is for Everyone

- Over the next few years, Olin is focusing on the mission of "Engineering for Everyone".
- We'd like to see this in the software engineering work we do as well.
- Even something as simple as internationalizing your code, however, may turn out to be quite difficult.
- https://www.youtube.com/watch?v=0j74jcxSunY

Processes vs. Threads

Process 1

- Kernel Memory
- Stack
- Unused
- Heap
- Globals
- Static
- Text

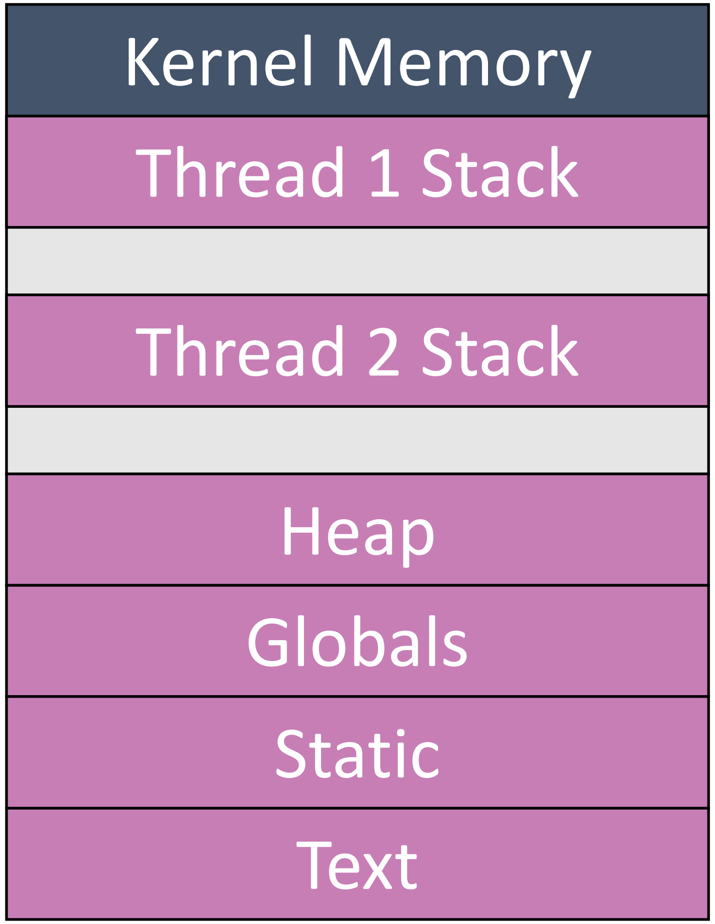
Each has its own address space

Each has at least one thread

Process 2

- Kernel Memory
- Stack
- Unused
- Heap
- Globals
- Static
- Text

Processes vs. Threads



Threads have their own stack

Threads keep their own registers

Threads share address space

Concurrency

- Concurrency does NOT mean that:
  - Multiple events are happening at the same time.
  - Events happen according to a synchronized schedule.
- It means that:
  - Execution times of those events might overlap.
  - You can't tell by looking at the program what order they will happen in.
- Threads can be concurrent in the same process or across processes.

Execution Paths

```
x = 5; /* a1 */
printf("%d\n", x); /* a2 */
```

```
x = 7; /* b1 */
```

- Consider each step of each thread.
- Within each thread, steps are sequential, but you can switch between threads at each step.
- An execution path is an order of steps among threads.
- So a1, b1, a2 represents switching to b1 between steps a1 and a2.

Execution Paths

```
x = 5; /* a1 */
printf("%d\n", x); /* a2 */
```

```
x = 7; /* b1 */
```

- Exercise: describe the execution paths that result in the following:
  - Prints 5 and ends with x = 5
  - Prints 7 and ends with x = 7
  - Prints 5 and ends with x = 7
  - Prints 7 and ends with x = 5

Concurrent Updates

- "Steps" aren't always single lines of a C program.
- Atomic operations always finish before switching threads.
- `x = x + 1` (or `x++`) are atomic on some machines, not on others.
- Think of it like this:

```
int copy = x;
x = copy + 1;
```

Concurrent Updates

Thread 1

```
int copy_a = x; /* a1 */
x = copy_a + 1; /* a2 */
```

Thread 2

```
int copy_b = x; /* b1 */
x = copy_b + 1; /* b2 */
```

- When you do updates like this, each thread gets its own local variables.

Concurrent Updates

Thread 1

```
int copy_a = x; /* a1 */
x = copy_a + 1; /* a2 */
```

Thread 2

```
int copy_b = x; /* b1 */
x = copy_b + 1; /* b2 */
```

- Exercise: consider the threads above.
  - How many possible execution paths are there?
  - What are the possible ending values of x?

Semaphores

- Basic building block in concurrency (there are a few others, too).
- Why semaphores? They're:
  - Simple
  - Versatile
  - Error-prone
- So, they're good to learn with.

Semaphores

- A semaphore is *basically* an integer.
- Three basic operations:
  - Initialize: set to a starting value.
  - Wait: decrement the value, and if it's negative, wait until it isn't.
  - Signal: increment the value, and wake up a waiting thread (if any).
- That's it - but is that really enough?

Semaphores

Thread 1
`puts("Never gonna");`

Thread 2
`puts("give you up");`

- How do we get Thread A's code to execute first?
- Use a semaphore:
  - Initialize it to 0.
  - After Thread 1's line, signal the semaphore.
  - Before Thread 2's line, wait for the semaphore.

Semaphores

Thread 1

```
puts("Never gonna");
signal(semaphore);
```

Thread 2

```
wait(semaphore);
puts("give you up");
```

- If the signaling (Thread 1) happens first, Thread 2 can just go ahead and print the second line.
- If the waiting (Thread 2) happens first, the thread waits until Thread 1 signals.
- Appropriately, this is called *signaling*.

Semaphores

Thread 1

```
puts("Peel the avocado");
puts("Eat the banana");
```

Thread 2

```
puts("Peel the banana");
puts("Eat the avocado");
```

- Exercise:
  - The above threads walk you through how to eat an avocado and banana.
  - Unfortunately, the directions are split between threads.
  - Use semaphores appropriately to make sure that each "peel" direction is printed before its respective "eat" action.

Beware Deadlock

- If you wait in a thread before signaling, you can end up where neither thread can proceed because it's waiting for the other.
- It's surprisingly easy to make this mistake, so be careful.

Mutexes

- Semaphores can be used to implement *mutexes*.
- Mutexes enable mutual exclusion - hence the name.
- They can protect against concurrent access - two threads accessing the same piece of data.
  - If threads are reading the same piece, that's fine - no mutex needed.
  - If one is reading and one is writing, those need to be ordered correctly.
  - If both are writing, they can't do that at once.

Mutexes

- First identify the *critical section* of code.
  - This is the part that actually does the access.
- Then, set up semaphore operations around that critical section.
  - If one thread is in the critical section, the other should not be able to enter it.
  - When a thread is done with the critical section, it should release its "lock" on that section.

Mutexes

Thread 1	Thread 2
`puts("Isolating node");` `puts("Firewall enhanced");`	`puts("Severing database");` `puts("SQL is now PreQL");`

- https://www.youtube.com/watch?v=u8qgehH3kEQ
- Exercise:
  - Set up and use one or more semaphores so that each thread runs both lines at a time.
  - Neither thread should interrupt the other while its lines are printing.