CS 220: Introduction to Parallel Computing

Dining Philosophers, Semaphores

Lecture 27

Today's Schedule

- Dining Philosophers
- Semaphores
- Barriers
- Thread Safety

Today's Schedule

- **Dining Philosophers**
- Semaphores
- Barriers
- Thread Safety

Dining Philosophers Problem

- Five silent philosophers sit around a table
- Each philosopher has two functions:
  - Think
  - Eat
- Five bowls of rice and five chopsticks are placed around the table
  - A philosopher must have two chopsticks to begin eating

Dining Philosophers: Algorithm

- Think until left chopstick is available
  - Pick it up
- Think until right chopstick is available
  - Pick it up
- Eat until full
- Put the forks down
- Repeat

Pitfalls: Deadlock

- What will happen if all the philosophers pick up the chopstick on the left at the same time?
  - Everyone will have one chopstick
  - Everyone will wait
- **Deadlock**

How Likely is Deadlock?

- In this situation, deadlock might not happen right away
- The philosophers will eat and think for eternity
  - It's their job!!
- **Eventually**, deadlock will happen

Problem 2: Starvation

- Let's assume the system won't deadlock. We still have another problem!
- Two (or three) of the philosophers might be a bit quicker than the others
  - Always get the chopsticks first
- The other philosophers wait, wait, and wait
  - Never get a chance to eat
- This demonstrates **resource starvation**

Livelock

- Let's assume that we only let a philosopher hold onto a single chopstick for 1 minute
  - After the minute elapses, they have to put it back down
- This will solve the problem, right?
- Not necessarily: it is possible that all the philosophers put down the chopstick at the same time, and then pick them back up the same time
- **Livelock**: the system keeps moving but makes no progress

One Solution: A Waiter

- If we can introduce a third party arbitrator (Waiter), then we can make sure the philosophers stay alive and get their thinking done
- How is this implemented in code?
  - Mutex
- To pick up a chopstick, you have to ask the waiter for permission
- You can put down a chopstick at any time

Today's Schedule

- Dining Philosophers
- **Semaphores**
- Barriers
- Thread Safety

Semaphores (1/2)

- We discussed using condition variables to protect a shared, limited resource
  - Such as whiteboards
- In our setup, we needed to maintain a mutex, a condition variable, and a counter (number of students at the board)
- There is a higher-level abstraction for handling this situation: **semaphores**

Semaphores (2/2)

- **Counting semaphores** include the counter logic
- Two functions:
  - P – *proberen – "to test"*
  - V – *vrijgave – "release"*
- Invented by Edsger Dijkstra, a Dutch computer scientist

pthread semaphores

- In pthreads, we have these functions:
  - sem_wait
  - sem_post
- Initialize with sem_init:
  - ```
    int sem_init(
        sem_t *sem, int pshared, unsigned int value);
    ```
    ```

Breaking it Down: Functions

- P(s):
  - s = s – 1
  - if (s < 0) , wait

- V(s):
  - s = s + 1
  - Notify waiting threads

Today's Schedule

- Dining Philosophers
- Semaphores
- **Barriers**
- Thread Safety

Barriers

- MPI lets us ensure all ranks call a particular function before moving on: **barrier**
- As you'd expect, pthreads also have barriers

```
pthread_barrier_init(
     pthread_barrier_t *bar_p, N unsigned count);
pthread_barrier_wait(pthread_barrier_t *bar_p);
pthread_barrier_destroy(pthread_barrier_t *bar_p);
```

However...

- Not all implementations of pthreads support barriers
- In particular, macOS does not include them
- We can simulate a barrier using a condition variable
- Each thread increments a shared counter, then waits
- Once the counter reaches the number of threads, we can tell them all, "go!"
  - `pthread_cond_broadcast`
    - No need to maintain a list of threads: just broadcast!

Today's Schedule

- Dining Philosophers
- Semaphores
- Barriers
- **Thread Safety**

Thread Safety

- You may remember earlier in the semster when I mentioned strktok is not **thread safe**
- Let's think back to how strtok works:

```
char *token = strtok(line, ", \n");
while (token != NULL) {
	/* do something with token */
	/* then grab the next token: */
	token = strtok(NULL, ", \n");
}
```

Threads and Strtok

- The second time we call strtok, we pass in NULL
- The function knows to reuse the string we passed in earlier
- How does it remember?
  - A global variable within the C library
- And we all know what happens when a global variable gets accessed by multiple threads…
  - Pandemonium! Well, sort of…

Strtok Race Condition

- If multiple threads are calling strtok at the same time, they may:
  - Erase the string being tokenized and replace it with another
  - Grab the next token before the other thread can
- This leads to unpredictable behavior and lost data
- Therefore, strtok is said to be **not thread safe**

Thread Safety

- Why not just make everything thread safe?
- As we've seen, there is some overhead associated with using **synchronization primitives**
  - Mutexes
  - Condition Variables
- Sometimes adding this overhead to serial applications is unacceptable
  - Let's see what this overhead looks like…

Checking For Thread Safety

- You'll see functions in C, Java, and many other languages that are not thread safe
- This is generally noted in the documentation
  - Java does a great job of highlighting which classes are thread safe
- We can also enforce thread safety ourselves:
  - Create a mutex
  - Make any thread that wishes to call strtok acquire the mutex first