CS 1217

Lecture 22 – Synchronization Wrap

The Bank Example w/ Test and Set

```
+int payMoolah = 0; // Shared variable for our test and set.

void UpdateTheMoolah (account_t account, int largeAmount) {
+ while (TestAndSet(& payMoolah, 1) == 1) {
+ ; // Test it again!
+ }
  int currBal = get_balance (account);
  currBal = currBal + largeAmount;
  put_balance (account, currBal);
+ TestAndSet(& payMoolah, 0); // Clear the test and set.
  return; }
```

- What are the **problems** with this approach?
  - **Busy waiting**: threads wait for the critical section by ”knocking on the door", executing the TAS repeatedly.
  - Bad on a multicore system; Worse on a single core system! **Busy waiting prevents the thread in the critical section from making progress!**

Busy Waiting

Person A

Person B

Balance

₹ 10000

```
while (TestAndSet(&payMoolah, 1));
int currBal = get_balance(account);
```

```
while (TestAndSet(&payMoolah, 1));
```

```
while (TestAndSet(&payMoolah, 1));
```

```
while (TestAndSet(&payMoolah, 1));
```

```
while (TestAndSet(&payMoolah, 1));
```

```
while (TestAndSet(&payMoolah, 1));
```

Locks

- **Locks** are a synchronization primitive used to implement critical sections.

- Threads **acquire** a lock when entering a critical section
- Threads **release** a lock when leaving a critical section

Spinlocks

- What we just saw was a spinlock

- **Lock**: guards a critical section
- **Spin**: The process of acquiring the lock

- Spinlocks are **commonly used** to build more useful synchronization primitives

Bank Example with Locks

```
lock WalletLock; // Need to initialize somewhere

void UpdateTheMoolah (account_t account, int largeAmount) {
+ lock_acquire(& WalletLock);
  int currBal = get_balance (account);
  currBal = currBal + largeAmount;
  put_balance (account, currBal);
+ lock_release(& WalletLock);
  return;
}
```

- What happens if we call lock_acquire() while another thread is in the critical section?
- The thread acquiring the lock must wait until the thread holding the lock calls lock_release().

Ways to Wait

- **Active** (or busy) waiting: repeat some action until the lock is released.

- **Passive** waiting: tell the kernel what we are waiting for, go to sleep, and rely on lock_release() to awaken us.

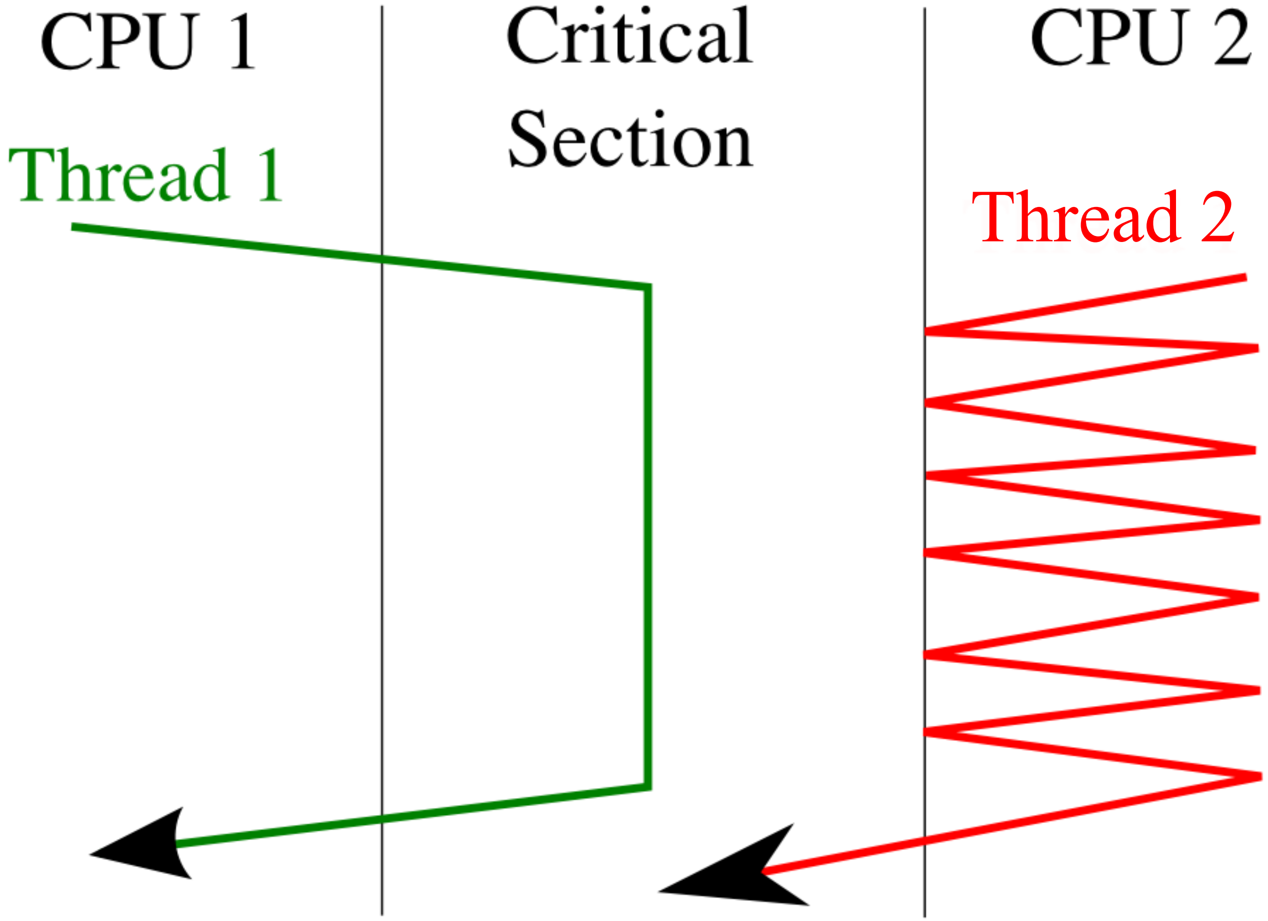
Spinning vs. Sleeping

- There are cases where spinning might be the right thing to do. **When?**

- Only on multicore systems. Why?
  - On single core systems **nothing can change** unless we allow another thread to run!
- If the critical section is **short**
  - Balance the length of the **critical section** against the overhead of a **context switch**.

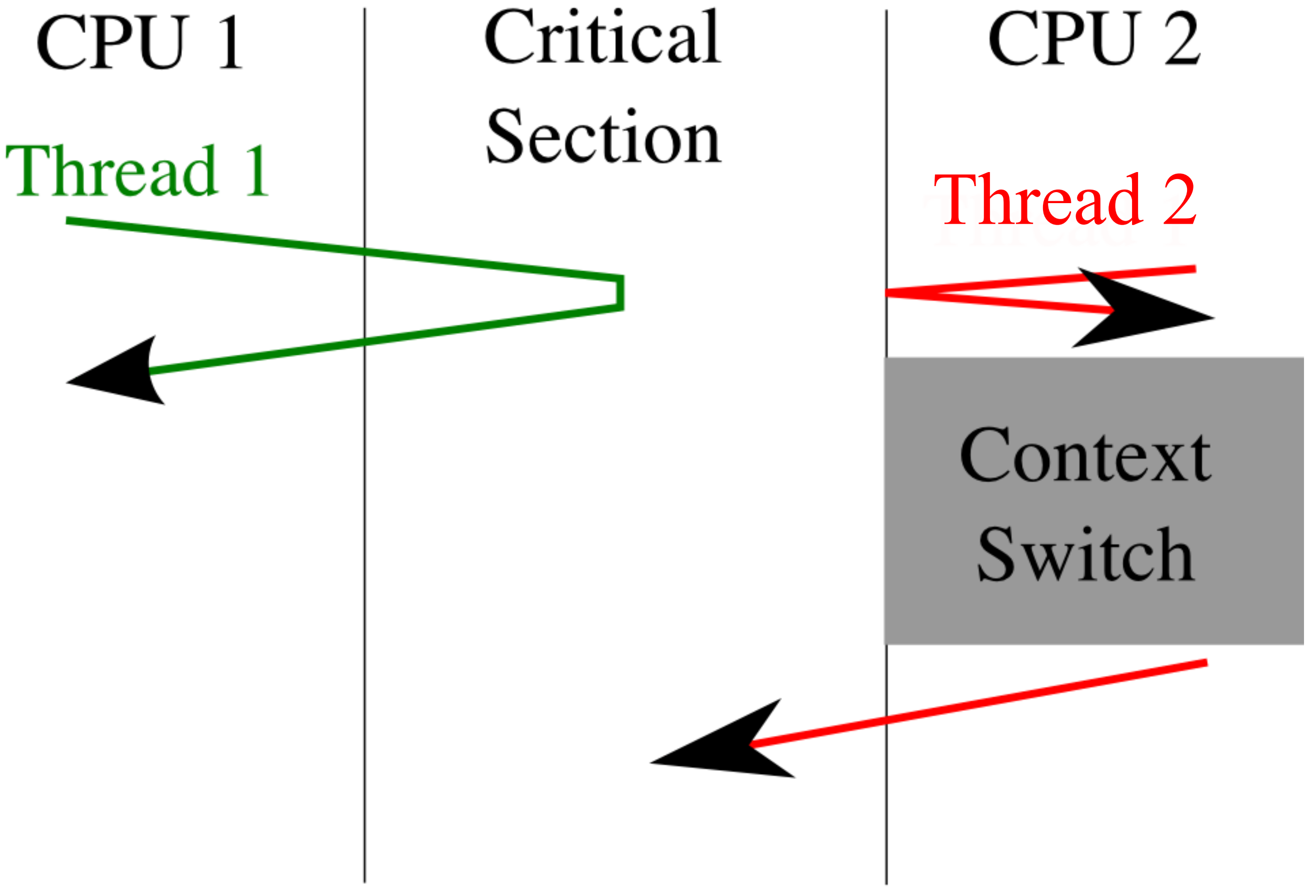
When to Sleep

- When the critical section is long:



When to Spin

- When the critical section is short:



How to Sleep?

- The kernel provides functionality allowing kernel threads to sleep and wake on a **key**:
- `thread_sleep(key)`: "Hey kernel, I’m going to sleep, but please wake me up when **key** happens.”
- `thread_wake(key)`: "Hey kernel, please wake up all (or one of) the threads who were waiting for **key**."
- Similar functionality can be implemented in user space.

Communication Between Threads

- Locks are designed to protect **critical sections**.
- `lock_release()` can be considered a **signal** from a thread
- Which thread?
  - The thread that is inside the critical section
- What is the signal?
  - Indication to other other threads that they can proceed.
- Are there different kind of “signals” that can be delivered?
  - Producer – Consumer relationships
    - A buffer has data in it.
  - The child process has exited

Condition Variables

- A **condition variable** is a signaling mechanism allowing threads to:
  - `cv_wait` until a **condition** is true, and
  - `cv_notify` / `cv_signal` other threads when the condition becomes true.

- The **condition** is usually represented as some change to shared state
  - The buffer has data in it: **bufsize > 0**.
  - `cv_wait`: notify me when the buffer has data in it.
  - `cv_signal`: Data has been put in the buffer, so notify threads that are waiting for the buffer.

Condition Variables

- **Condition variable** can convey **more information** than locks about some change to the state.
- E.g. a buffer can be **full**, **empty**, or **neither**.
  - If the buffer is **full**, let threads **withdraw** but **not add** items.
  - If the buffer is **empty**, let threads **add** but **not withdraw** items.
  - If the buffer is neither full nor empty, let threads **add and withdraw** items.
- We have **three** different buffer states (**full**, **empty**, or **neither**) and **two** different threads (**producer**, **consumer**).

Condition Variables

- Why are condition variables a synchronization mechanism?
- Need to ensure that the condition **does not change** between checking it and deciding to wait!

Thread 1	Thread 2
if (buffer_is_empty)	
	put (buffer) notify (buffer)
Wait_for_buffer()	

Back to Locks

- **Locks** protect access to shared resources.
- Threads may need **multiple** shared resources to perform some operation(s)

Locking Multiple Resources

- Consider two threads A and B that both need **simultaneous** access to resources 1 and 2:
- **Thread A** runs, grabs the lock for **Resource 1**.
- → CONTEXT SWITCH ←
- **Thread B** runs, grabs the lock for **Resource 2**.
- → CONTEXT SWITCH ←
- **Thread A** runs, tries to acquire the lock for **Resource 2**
- → THREAD A SLEEPS ←
- **Thread B** runs, tries to acquire the lock for **Resource 1**.
- → THREAD B SLEEPS ←

- Now what?

Deadlock

- **Deadlock** occurs when a thread or set of threads are waiting for each other to finish and thus nobody ever does.

- ***A*** thread?

Self Deadlock

- Can a single thread deadlock? How?

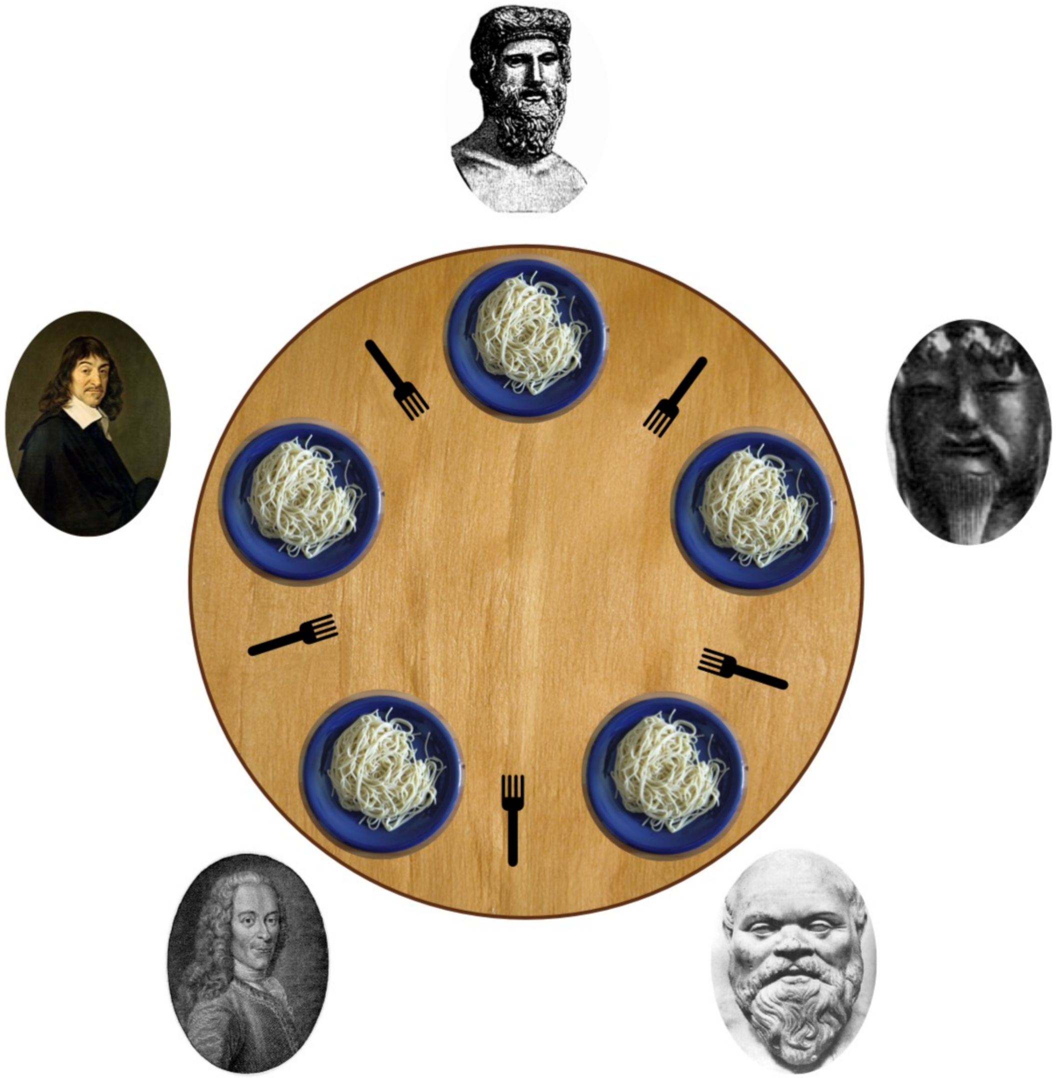
- Thread A acquires Resource 1. Thread A tries to reacquire Resource 1

- Why would this happen?
- foo() needs Resource 1. bar() needs Resource 1. While locking Resource 1 foo() calls bar()

Conditions for Deadlock

- A deadlock **cannot occur** unless all of the following conditions are met:
  - **Protected access** to shared resources, which implies waiting.
  - **No resource preemption**, meaning that the system cannot forcibly take a resource from a thread holding it.
  - **Multiple independent requests**, meaning a thread can hold some resources while requesting others.
  - **Circular dependency graph**, meaning that Thread A is waiting for Thread B which is waiting for Thread C which is waiting for Thread D which is waiting for Thread A.

Dining Philosophers "Problem"



Making sure the Philosophers Eat

- Breaking deadlock conditions usually requires eliminating one of the **requirements** for deadlock.
- **Don’t wait**: don’t sleep if you can’t grab the second fork and put down the first.
- **Break cycles**: usually by acquiring resources in a **well-defined order**. Number forks 0–4, always grab the higher-numbered fork first.
- **Break out**: detect the deadlock cycle and forcibly take away a resource from a thread to break it. (Requires a new mechanism.)
- **Don’t make multiple independent requests**: grab **both** fork at once. (Requires a new mechanism.)

Deadlock vs. Starvation

- **Starvation** : condition in which one or more threads do not make progress.
- Starvation differs from deadlock in that **some** threads make progress and it is, in fact, those threads that are preventing the "starving" threads from proceeding.

Deadlock vs. Starvation

- What is better: a **deadlock** (perhaps from overly careful synchronization) or a **race condition** (perhaps from a lack of correct synchronization)?

Choice of Tool Matters

- Most problems can be solved with a **variety** of synchronization primitives.
- However, there is usually **one primitive** that is more appropriate than the others.

General Approach to Synchronization Problems

- Identify the constraints.
- Identify shared state.
- Choose a primitive.
- Pair waking and sleeping.
- Look out for multiple resource allocations: can lead to deadlock.
- Walk through simple examples and corner cases **before** beginning to code.