Deadlock — Last Time

Deadlock can exist if and only if the following conditions hold.

1. **Mutual Exclusion** - at least one thread must hold a resource in non-sharable mode, i.e., the resource may only be used by one thread at a time.
2. **Hold and Wait** - at least one thread holds a resource and is waiting for other resource(s) to become available. A different thread holds the resource(s).
3. **No Preemption** - A thread can only release a resource voluntarily; another thread or the OS cannot force the thread to release the resource.
4. **Circular wait** - A set of waiting threads $\{t_1, \ldots, t_n\}$ where t_i is waiting on t_{i+1} $(i = 1\ to\ n)$ and t_n is waiting on t_1.

- How can we detect deadlock?
- How can we prevent deadlock?

Today:
Big picture & one more classical synchronization problem.

Synchronization Wrap up — Today

- Low-level synchronization
- High-level synchronization
- Dining Philosophers

Low-Level Synchronization Primitives

Hardware support for synchronization:

	Advantages	**Disadvantages**
Load/Store		
Interrupt Disable		
Test&Set		

What can the OS do with these low-level primitives? the user?

High-Level Synchronization Primitives

Locks :

Value = Free	Initially lock is always free.
Acquire:	Guarantees only one thread has lock. if another thread holds the lock, Acquiring thread waits. Otherwise, the Acquiring thread continues
Release:	Enables another thread to get lock. If threads are waiting, one gets the lock, else, the lock becomes free.

Semaphores :

Value	Initialization depends on problem.
Wait	Decrements value Thread continues if value ≥ 0 (semaphore is available), otherwise, it waits on semaphore.
Signal	Unblocks a process on the wait queue, otherwise, increments value

A **counted semaphore** enables simultaneous access to a fixed number of resources.

What is the relationship between semaphores and locks?

High-Level Synchronization Primitives

Monitors :

Monitor Locks provide mutual exclusion to shared data.

Lock::Acquire – wait until lock is free, then grab it.

Lock::Release – unlock, and wake up any thread waiting in Acquire.

Rules for using a lock:

- Always acquire lock before accessing shared data structure.
- Always release lock when finished with shared data.
- Lock is initially free.

A Condition Variable is a queue of threads waiting for something inside a critical section. Condition variables support three operations:

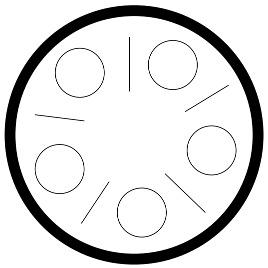
1. Wait() - atomically release lock, go to sleep
2. Signal() - wake up waiting thread (if one exists) and give it the lock
3. Broadcast() - wake up all waiting threads

Rule: thread must hold the lock when doing condition variable operations.

What is the relationship between monitors and locks? and semaphores?

Dining Philosophers — Classical Problem

N philosophers are sitting at a table with N chopsticks, each needs two chopsticks to eat, and each philosopher alternates between thinking, getting hungry, and eating. (This problem is considered a classic since it represents many threads competing for the same resources, and the solution must prevent deadlock and starvation.)



A simple solution using locks.

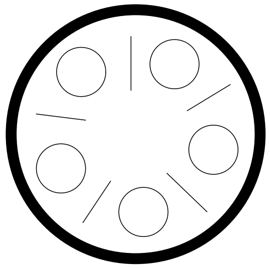
```
do forever
  think
  get hungry
  wait(chopstick[i]);
  wait(chopstick[i+1 mod N]);

  eat

  signal(chopstick[i]);
  sig-
nal(chopstick[i+1 mod N]);
```

Does this solution work?

Dining Philosophers — Classical Problem



Possible solutions:

- Prevent circular wait by having sufficient resources: Only allow N-1 philosophers to come to the table.
- Prevent circular wait by ordering resources: Use an asymmetric solution, where the odd philosophers pick up right then left, and the even philosophers pick up left then right.
- Prevent hold-and-wait: Only let a philosopher pick up chopsticks if both are available. The pick up must be put in a critical section.
- Preempt resources: Designate one philosopher as the head philosopher. Allow that philosopher to take a chopstick from a neighbor if the neighbor is not currently eating.
- Don't require mutual exclusion: Not practical since a chopstick cannot be shared.

```
class PhilosophersMonitor {
   public: Pickup(int i); Putdown(int i);
   private: Test(int k);
     state[N]: array of (thinking, hungry, eating);
     Lock lock;
     CVar self[N]; }
PhilosophersMonitor::PhilosophersMonitor {
   state[0..N-1] = thinking;
   lock = FREE;
}
PhilosophersMonitor::Pickup (int i) {
   lock->Acquire();
   state[i] = hungry;    // indicate you want to eat
   Test(i)  // test if it is ok to eat
   if (state[i] != eating)  // state[i] == eating, you can eat
      self[i]->Wait();
   lock->Release();
}
PhilosophersMonitor::Putdown (int i) {
   lock->Acquire();
   state[i] = thinking;  // stop eating
   Test(i+1 mod N);      // do neighbors want to eat?
   Test(i+N-1 mod N);
   lock->Release();
}
PhilosophersMonitor::Test (int k) {
   if (state[k+1 mod N] != eating and   // k's right neigh-
bor is not eating
       state[k] = hungry and            // k wants to eat
       state[k+N-1 mod N] != eating) {  // k's left neigh-
bor is not eating
      state[k]  = eating;               // k can eat
      self[k]->Signal();                // wake k up if sleepin
}  }
```


What's an interesting calling sequence?

Does this solution prevent starvation?

How would we write the previous solution using semaphores?

Dining Philosophers with Semaphores

```
class Philosophers {
   public: Pickup(int i); Putdown(int i);
   private: Test(int k);
     state[N]: array of (thinking, hungry, eating);
     Lock lock;
     semaphore self[N]; }   // one semaphore per philosopher
Philosophers::Philosophers {
   state[0..N-1] = thinking;
   self[0..N-1] = 0;        // semaphores not available
   lock = FREE;
}
Philosophers::Pickup (int i) {
   lock->Acquire();
   state[i] = hungry;    // indicate you want to eat
   Test(i)  // test if it is ok to eat
   lock->Release();
   self[i]->Wait();
}
Philosophers::Putdown (int i) {
   lock->Acquire();
   state[i] = thinking;  // stop eating
   Test(i+1 mod N);      // do neighbors want to eat?
   Test(i+N-1 mod N);
   lock->Release();
}
Philosophers::Test (int k) {
   if (state[k+1 mod N] != eating and
       state[k] == hungry and state[k+N-1 mod N] != eat-
ing) {
      state[k]  = eating;
      self[k]->Signal();
}  }
```


Dining Philosophers with Resource Ordering

```
do forever
  think
  get hungry
  if ( i is even ) {
    wait(chopstick[i]);
    wait(chopstick[i+1 mod N]);
  }
  else {
    wait(chopstick[i+1 mod N]);
    wait(chopstick[i]);
  }

  eat

  signal(chopstick[i]);
  signal(chopstick[i+1 mod N]);
```

Summary

Deadlock detection and deadlock avoidance (Banker's Algorithm) are both expensive. It is often more practical to prevent deadlock by ensuring that one of the necessary conditions does not hold:

- Mutual exclusion
- Hold-and-wait
- No resource preemption
- No circular wait

Next Time

Exam Review