Rendezvous

A needs to be sure B has advanced to point L, B needs to be sure A has advanced to L

```
semaphore A_madeit(0);

A_rendezvous_with_B()
{
  sema_up(A_madeit);
  sema_down(B_madeit);
}
```

```
semaphore B_madeit(0);

B_rendezvous_with_A()
{
  sema_up(B_madeit);
  sema_down(A_madeit);
}
```


Waiting for an activity to finish


```
semaphore done_with_task(0);
thread_create(
      do_task,
      (void*)&done_with_task);

sema_down(done_with_task);
// safely access task's results
```

```
void
do_task(void *arg)
{
   semaphore *s = arg;
   /* do the task */
   sema_up(*s);
}
```

Works no matter which thread is scheduled first after thread_create (parent or child)

Elegant solution that avoids the need to share a "have done task" flag between parent & child

Two applications of this technique in Pintos Project 2
- signal successful process startup ("exec") to parent
- signal process completion ("exit") to parent

Dining Philosophers (Dijkstra)

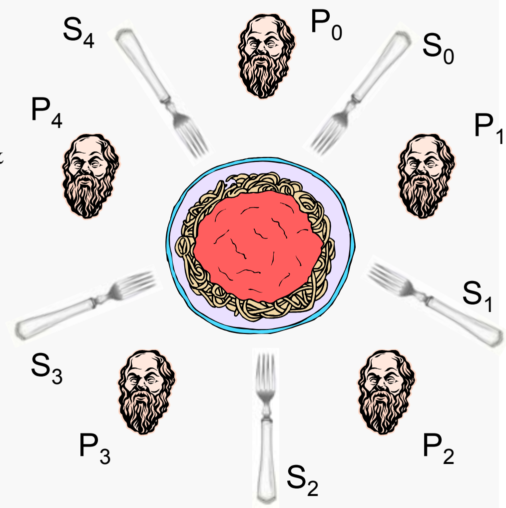
A classic

5 Philosophers, 1 bowl of spaghetti

Philosophers (threads) think & eat ad infinitum

- Need left & right fork to eat (!?)

Want solution that prevents starvation & does not delay hungry philosophers unnecessarily



Dining Philosophers (1)


```
semaphore fork[0..4](1);
philosopher(int i)                  // i is 0..4
{
  while (true) {
    /* think … finally */
    sema_down(fork[i]);             // get left fork
    sema_down(fork[(i+1)%5]);       // get right fork
     /* eat */
     sema_up(fork[i]);              // put down left fork
     sema_up(fork[(i+1)%5]);        // put down right fork
  }
}
```

What is the problem with this solution?
Deadlock if all pick up left fork

Dining Philosophers (2)


```
semaphore fork[0..4](1);
semaphore at_table(4);   // allow at most 4 to fight for
forks
philosopher(int i)                          // i is 0..4
{
  while (true) {
    /* think … finally */
    sema_down(at_table);            // sit down at table
    sema_down(fork[i]);             // get left fork
    sema_down(fork[(i+1)%5]);       // get right fork
     /* eat … finally */
     sema_up(fork[i]);              // put down left fork
     sema_up(fork[(i+1)%5]);        // put down right fork
     sema_up(at_table);             // get up
   }
}
```


Monitors

A monitor combines a set of shared variables & operations to access them
- Think of an enhanced C++ class with no public fields

A monitor provides implicit synchronization (only one thread can access private variables simultaneously)
- Single lock is used to ensure all code associated with monitor is within critical section

A monitor provides a general signaling facility
- Wait/Signal pattern (similar to, but different from semaphores)
- May declare & maintain multiple signaling queues

Monitors (cont'd)

Classic monitors are embedded in programming language
- Invented by Hoare & Brinch-Hansen 1972/73
- First used in Mesa/Cedar System @ Xerox PARC 1978
- Limited version available in Java/C#

(Classic) Monitors are safer than semaphores
- can't forget to lock data – compiler checks this

In contemporary C, monitors are a *synchronization pattern* that is achieved using locks & condition variables
- Must understand monitor abstraction to use it

Infinite Buffer w/ Monitor


```
monitor buffer {
      /* implied: struct lock
mlock;*/
private:
      char buffer[];
      int head, tail;
public:
      produce(item);
      item consume();
}
```

```
buffer::produce(item i)
{  /* try { lock_acquire(&mlock); */
   buffer[head++] = i;
   /* } finally {lock_release(&mlock);} */
}

buffer::consume()
{  /* try { lock_acquire(&mlock); */
   return buffer[tail++];
    /* } finally {lock_release(&mlock);}
*/
}
```

Monitors provide implicit protection for their internal variables
- Still need to add the signaling part

Condition Variables

Variables used by a monitor for signaling a condition

- a general (programmer-defined) condition, not just integer increment as with semaphores
- The actual condition is typically some boolean predicate of monitor variables, e.g. "buffer.size > 0"

Monitor can have more than one condition variable

Three operations:

- Wait(): leave monitor, wait for condition to be signaled, reenter monitor
- Signal(): signal one thread waiting on condition
- Broadcast(): signal all threads waiting on condition