Recap: Synchronization

- Low-level synchronization primitives:
  - Disabling preemption
  - Locks
  - Spinlocks
- Now: higher-level constructs
  - Semaphores
  - Monitors

Semaphores

- Invented by Edsger Dijkstra in 1965s
- Counter S, initialized to some value, with two operations:
  - P(S) or “down” or “wait” – if counter greater than zero, decrement. Else wait until greater than zero, then decrement
  - V(S) or “up” or “signal” – increment counter, wake up any threads stuck in P
- Semaphores don’t go negative:
  - #V + InitialValue - #P >= 0
- Note: direct access to counter value after initialization is not allowed
- Counting vs Binary Semaphores
  - Binary: counter can only be 0 or 1
- Simple to implement, yet powerful
  - Can be used for many synchronization problems

Infinite Buffer w/ Semaphores (1)

```
#include <semaphore.h>

semaphore items_avail(0);

producer() {
  lock_acquire(buffer);
  buffer[head++] = item;
  lock_release(buffer);
  sema_up(items_avail);
}

consumer() {
  sema_down(items_avail);
  lock_acquire(buffer);
  item = buffer[tail++];
  lock_release(buffer);
  return item;
}
```

- Semaphore “remembers” items put into queue (no updates are lost)
Implementing Semaphores

- Implementation is analogous to simple locks on uniprocessor
  - requires counter variable
  - requires disabling preemption
  - requires appropriate blocking/unblocking
- See Pintos synch.cc for details

Semaphores as Locks

- Semaphores can be used to build locks
  - Pintos does just that
  - Must initialize semaphore with 1 to allow one thread to enter critical section

```
semaphore S(1); // allows initial down
lock_acquire()
{    // try to decrement, wait if 0
  sema_down(S);
}
lock_release()
{    // increment (wake up waiters if any)
  sema_up(S);
}
```

Infinite Buffer w/ Semaphores (2)

```
semaphore items_avail(0);
semaphore buffer_access(1);
producer()
{
  sema_down(buffer_access);
  buffer[head++] = item;
  sema_up(buffer_access);
  sema_up(items_avail);
}
consumer()
{
  sema_down(items_avail);
  sema_down(buffer_access);
  item = buffer[tail++];
  sema_up(buffer_access);
  return item;
}
```

Bounded Buffer w/ Semaphores

```
semaphore items_avail(0);
semaphore buffer_access(1);
semaphore slots_avail(CAPACITY);
producer()
{
  sema_down(slots_avail);
  sema_down(buffer_access);
  buffer[head++] = item;
  sema_up(buffer_access);
  sema_up(items_avail);
}
consumer()
{
  sema_down(items_avail);
  sema_down(buffer_access);
  item = buffer[tail++];
  sema_up(buffer_access);
  sema_up(slots_avail);
  return item;
}
```

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(buffer_access);
  buffer[head++] = item;
  sema_up(buffer_access);
  return item;
}
```

Waiting for activity to finish

```
semaphore done_with_task(0);
thread_create;
do_task,
(void*)&done_with_task;
sema_down(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
- Pintos Project 2: signal successful process startup to parent
Dining Philosophers (1)

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

```c
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
  }
}
```

Dining Philosophers (2)

```c
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
  }
}
```