What is a semaphore?

An ADT with

\[ \text{sem\_wait} \quad \text{Down} \quad \text{P} \]

\[ \text{sem\_post} \quad \text{Up} \quad \text{V} \]

The C version looks like this

```c
#include <semaphore.h>
sem_t s;
sem_init(&s, shared, init_value); // shared==0 for single process

int sem_wait(sem_t *s)
{
    decrement the value of semaphore s by one
    wait if value of semaphore s is negative
}

int sem_post(sem_t *s)
{
    increment the value of semaphore s by one
    if there are one or more threads waiting, wake one
}
```

The `sem_wait` and `sem_post` functions are **ATOMIC**!

Using semaphores as locks (binary semaphores)

```c
sem_t s;
sem_init(&s, 0, 1);

sem_wait(&s);
// critical section is here
sem_post(&s);
```

What value should the semaphore be initialized to? 1
Producer/Consumer with Semaphores
Recall from Condition Variables:
- have full and empty condition variables
- producer waits on empty and signals full
- consumer waits on full and signals empty

With semaphores
- Put N buffers on free list.
- Initialize semaphores: empty = N, full = 0, mutex = 1;

<table>
<thead>
<tr>
<th>Process A (Producer)</th>
<th>Process B (Consumer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sem_wait(empty);</td>
<td>sem_wait(full);</td>
</tr>
<tr>
<td>sem_wait(mutex);</td>
<td>sem_wait(mutex);</td>
</tr>
<tr>
<td>get empty buffer from mutex of empties;</td>
<td>get full buffer from mutex of fulls;</td>
</tr>
<tr>
<td>sem_post(mutex);</td>
<td>sem_post(mutex);</td>
</tr>
<tr>
<td>produce data in buffer;</td>
<td>consume data in buffer;</td>
</tr>
<tr>
<td>sem_wait(mutex);</td>
<td>sem_wait(mutex);</td>
</tr>
<tr>
<td>add full buffer to mutex of fulls;</td>
<td>add empty buffer to mutex of empties;</td>
</tr>
<tr>
<td>sem_post(mutex);</td>
<td>sem_post(mutex);</td>
</tr>
<tr>
<td>sem_post(full);</td>
<td>sem_post(empty);</td>
</tr>
</tbody>
</table>

Important questions:
- Why does A sem_wait(empty) but sem_post(full)?
  Deadlock is Bad
- Why is order of sem_wait's important?
- Is order of sem_post's important?
- Could we have two separate mutex semaphores?
- How would this be extended to have two consumers?
The Dining Philosophers – Uniform resource allocation example

Rules:
- Philosophers don’t talk to each other
- They are always either:
  - Thinking
  - Eating
  - Waiting/Hungry
- Process:
  - Think
  - Pick up left fork
  - Pick up right
  - Eat
  - put the right fork down
  - put the left fork down
  - repeat from the beginning

Goals:
- Want every philosopher to be able to eat the same amount, on average
- We want to avoid deadlock

A non-solution: How about everyone gets their forks left then right.

A "solution": ok same thing except one philosopher who gets them right then left.

A Fair Solution

```c
/* dining philosophers */
#define N 5
#define LEFT(i) ((i-1)&N)
#define RIGHT(i) ((i+1)&N)

semaphore mutex = 1;
semaphore s[N]; /* one of */
int state[N]; /* one of */
#define THINKING 0
#define HUNGRY 1
#define EATING 2

philosopher(int i) {

  take_forks(i) {
      while (TRUE) {
          sem_wait(mutex);
          if (state[i] == HUNGRY) {
              think();
              state[i] = THINKING;
              sem_wait(mutex);
              test(LF(i));
          }
      }
      sem_wait(mutex);
      take_forks(i);
      sem_post(mutex);
      eat();
      sem_wait(s[i]);
      sem_post(mutex);
  }
}

put_forks(int i) {

  if (state[i] == HUNGRY && state[LEFT(i)] != EATING && state[RIGHT(i)] != EATING) {
      state[i] = EATING;
      sem_post(s[i]); /* may be a "preemptive" post */
  }
}
```

CS466 Notes 14 Page 3
Readers and Writers – model non-uniform resource allocation

General idea:

Ever booked an airplane flight?
   1. Can multiple seat assignments happen at the same time?
   2. Can multiple seat inspections happen at the same time?
   3. Can (1) and (2) happen at the same time?

Raises need for asymmetric lock types.

typedef struct // Figure 31.13: A Simple Reader-Writer Lock
{
    sem_t lock; // binary semaphore (basic lock)
    sem_t writelock; // used to allow ONE writer or MANY readers
    int readers; // count of readers reading in critical section
} rwlock_t;

void rwlock_init(rwlock_t *rw)
{
    rw->readers = 0;
    sem_init(&rw->lock, 0, 1);
    sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw)
{
    sem_wait(&rw->lock);
    rw->readers++;
    if (rw->readers == 1)
        sem_wait(&rw->writelock); // first reader acquires writelock
    sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw)
{
    sem_wait(&rw->lock);
    rw->readers--;
    if (rw->readers == 0)
        sem_post(&rw->writelock); // last reader releases writelock
    sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw)
{
    sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw)
{
    sem_post(&rw->writelock);
}