HW2 Solution

To be available at

`~cs5460/hw2/hw2_soln.c`

on the CADE filesystem

Last Time

Concurrency pitfalls

- Atomic operations depend on the processor
- Multiprocessors don’t even offer true globals automatically

Solutions

- Processor-supplied operations, e.g., compare-and-swap
- OS-supplied locks, e.g., mutexes

Globals

When is a C global variable actually *global*?

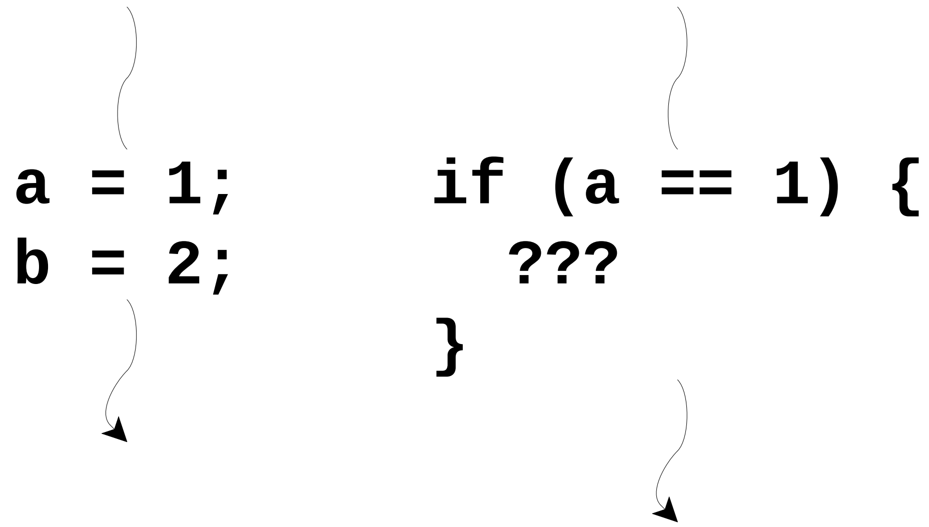
When it’s consistently protected by a lock.

```
static int counter;
...

   lock();
   counter++;
   unlock();
```

Globals

```
static int a = 0;
static int b = 0;
```



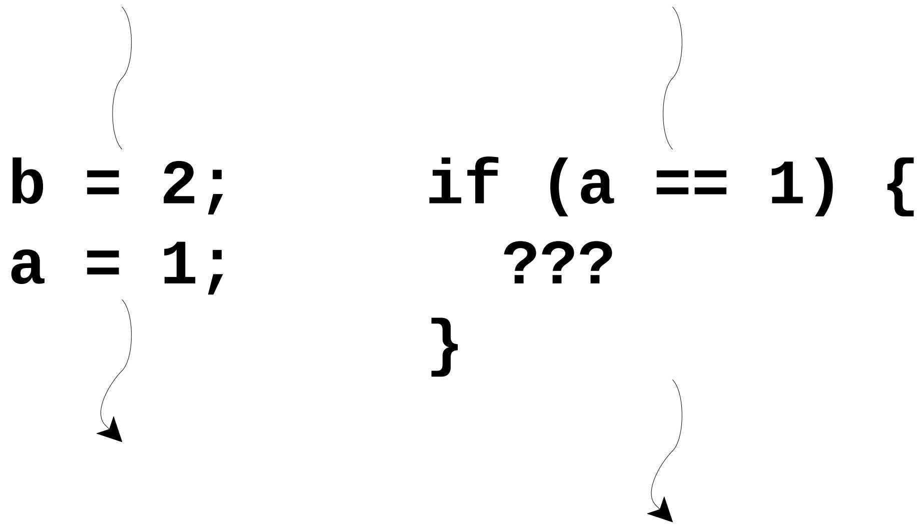
```
a = 1;      if (a == 1) {
b = 2;        ???
            }
```

If the left thread reaches **???**, is **b** necessarily **2**?

No.

Globals

```
static int a = 0;
static int b = 0;
```



```
b = 2;        if (a == 1) {
a = 1;          ???
              }
```

If the left thread reaches **???**, is **b** necessarily **2**?

No.

⇒ use a lock around accesses of **a** and **b**

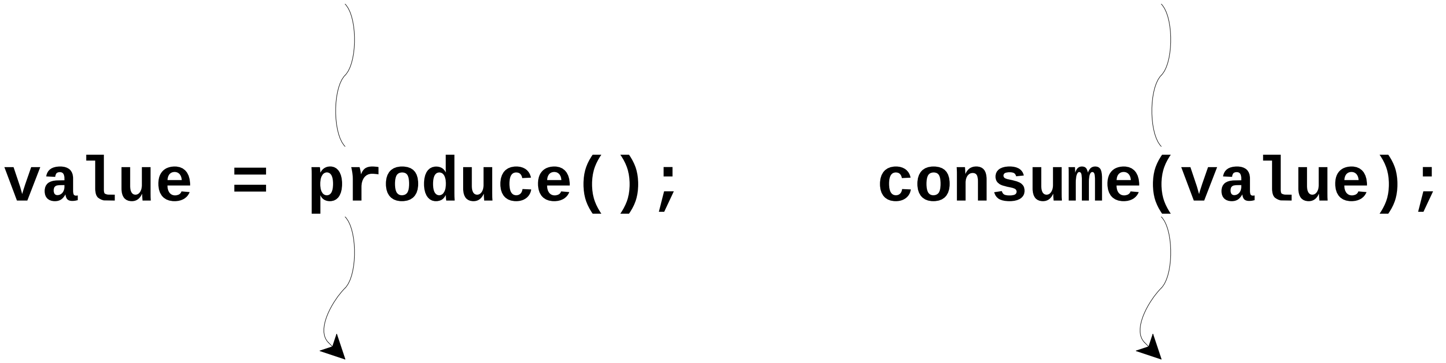
General Points about Shared Data and Concurency

1. Protect shared globals with a lock.

2. No, really, use a lock!

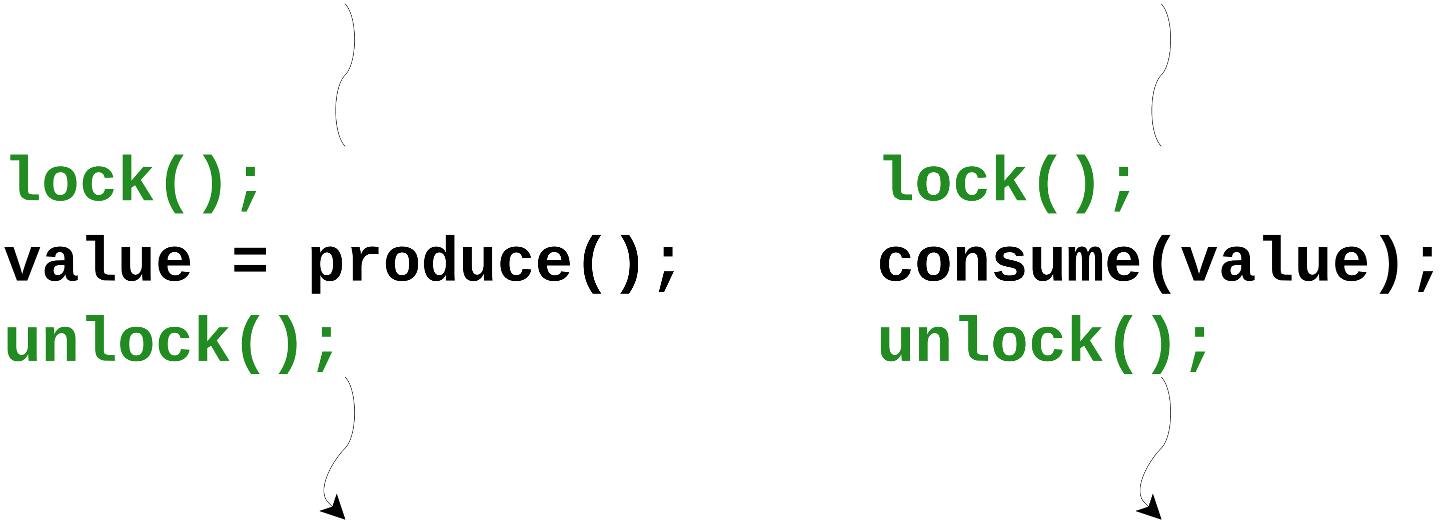
3. I’m not kidding about using locks.

Producer & Consumer



```
value = produce();      consume(value);
```

Producer & Consumer



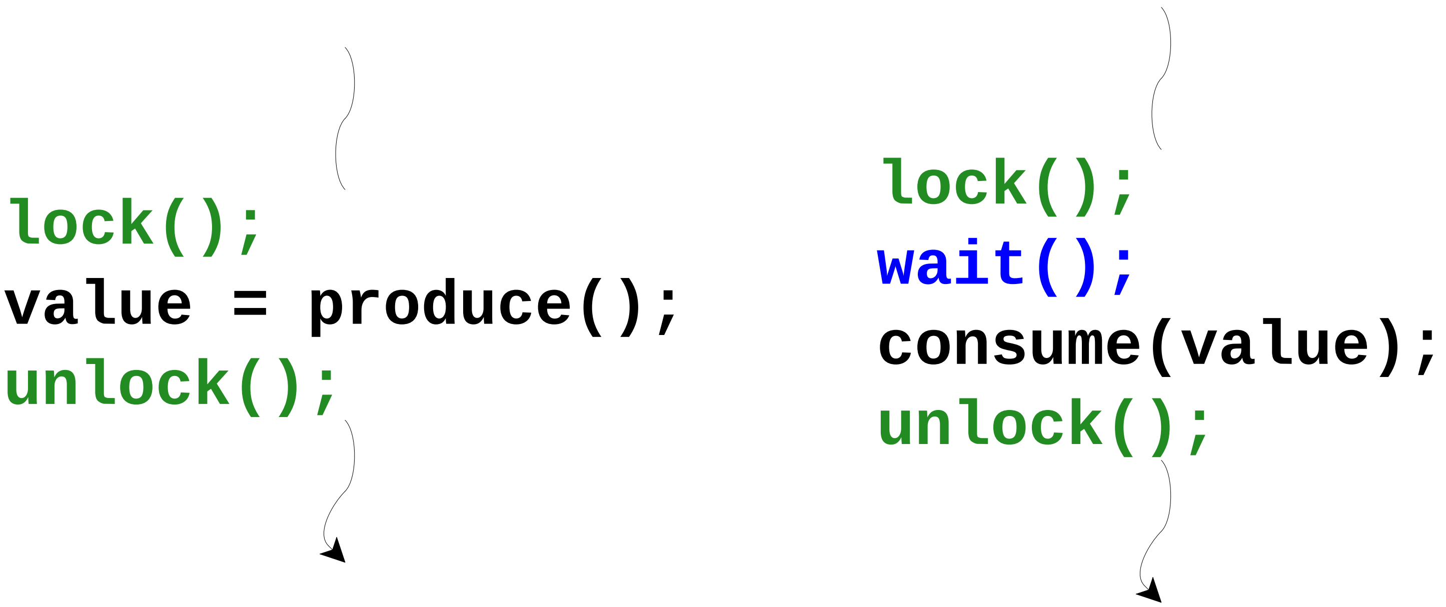
```
lock();                      lock();
value = produce();           consume(value);
unlock();                    unlock();
```

Producer & Consumer

```
lock();
value = produce();
unlock();
```

```
lock();
while (!value) {
  unlock(); lock();
}
consume(value);
unlock();
```

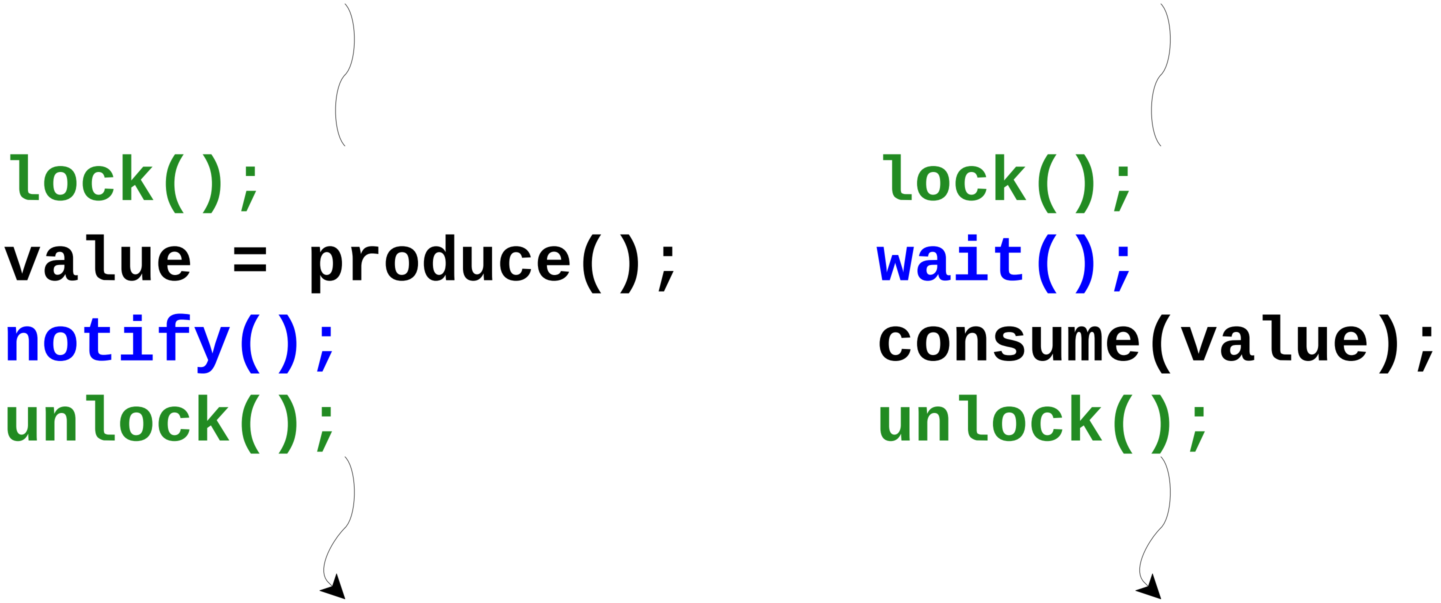
Producer & Consumer



```
lock();
value = produce();
unlock();
```

```
lock();
wait();
consume(value);
unlock();
```

Producer & Consumer



```
lock();
value = produce();
notify();
unlock();
```

```
lock();
wait();
consume(value);
unlock();
```

Producer & Consumer

```
lock();
value = produce();
notify();
unlock();
```

```
lock();
wait();⬅
consume(value);
unlock();
```

waiting temporarily releases the lock

Mutexes + Conditions

See **`prod_cons.c`** and **`prod_cons_2.c`**

The **`while`** plus **`pthread_cond_wait`** pattern avoids a race on starting wait versus delivering signal

Semaphores

A ***semaphore*** encapsulates the mutex + condition pattern

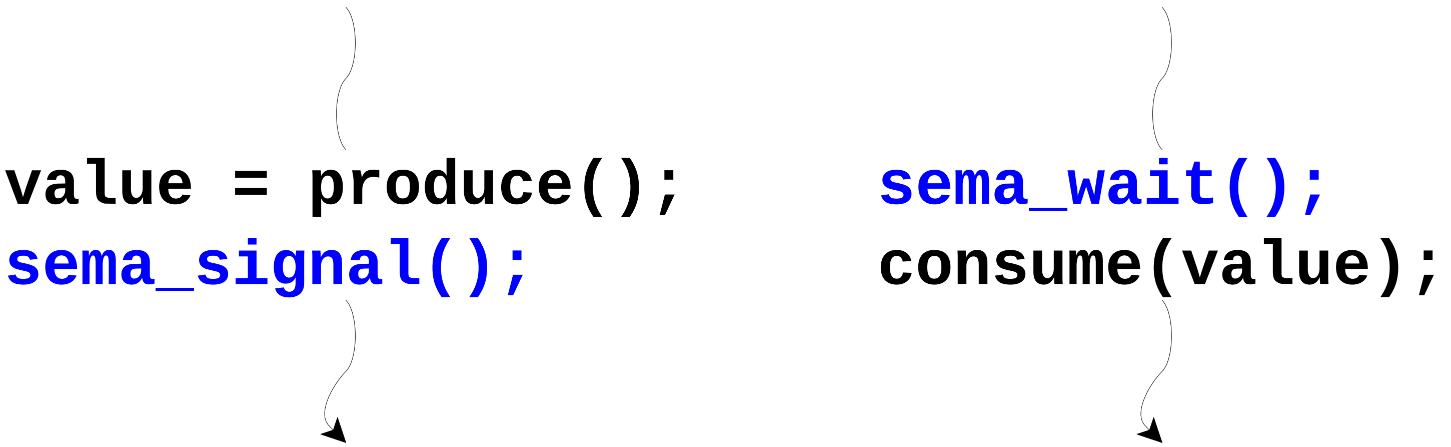
- **`sema_wait()`**

  a.k.a. **`P()`**

- **`sema_signal()`**

  a.k.a. **`V()`**, **`sema_post()`**

Producer & Consumer with a Semaphore



```
value = produce();       sema_wait();
sema_signal();           consume(value);
```

Unlike conditions, a semaphore signal is retained until waited on

Semaphores

See **`sema_prod_cons.c`** and **`sema_prod_cons_2.c`**

Binary vs. Counting Semaphores

A ***binary semaphore*** holds a single signal

A ***counting semaphore*** holds multiple signals to be consumed by multiple waits

Semaphores as Plain Locks

```
mutex_lock(m);          semap_wait(s);
critical region         critical region
mutex_unlock(m);        sema_signal(s);
```

Monitors

What happens if you get it backward?

```
sema_signal(s);
critical region
sema_wait(s);
```

A ***monitor*** is a language construct that helps avoid such mistakes

```
synchronized {
    critical region
}
```

(see book for more details)

Multiple Data

Two different objects:

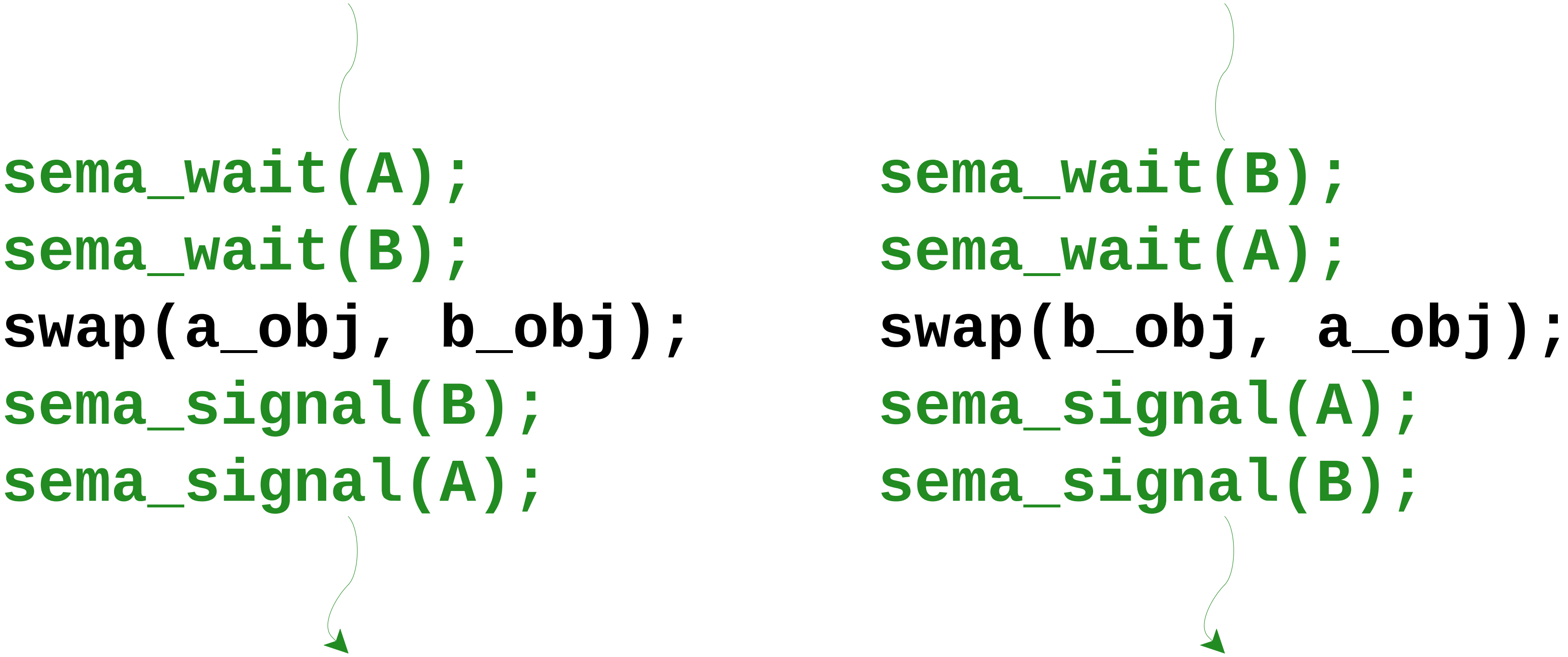
```
static thing_t a_obj;
static thing_t b_obj;
```

One lock or two?

- If **`a_obj`** and **`b_obj`** are always used together, one lock is probably best.
- If **`a_obj`** and **`b_obj`** are often used independently, then give them separate locks.

In the second case, you sometimes need both locks...

Multiple Locks



```
sema_wait(A);               sema_wait(B);
sema_wait(B);               sema_wait(A);
swap(a_obj, b_obj);         swap(b_obj, a_obj);
sema_signal(B);             sema_signal(A);
sema_signal(A);             sema_signal(B);
```

To avoid deadlock when acquiring multple locks:

- Establish a total order on all locks
- Always acquire the locks in order