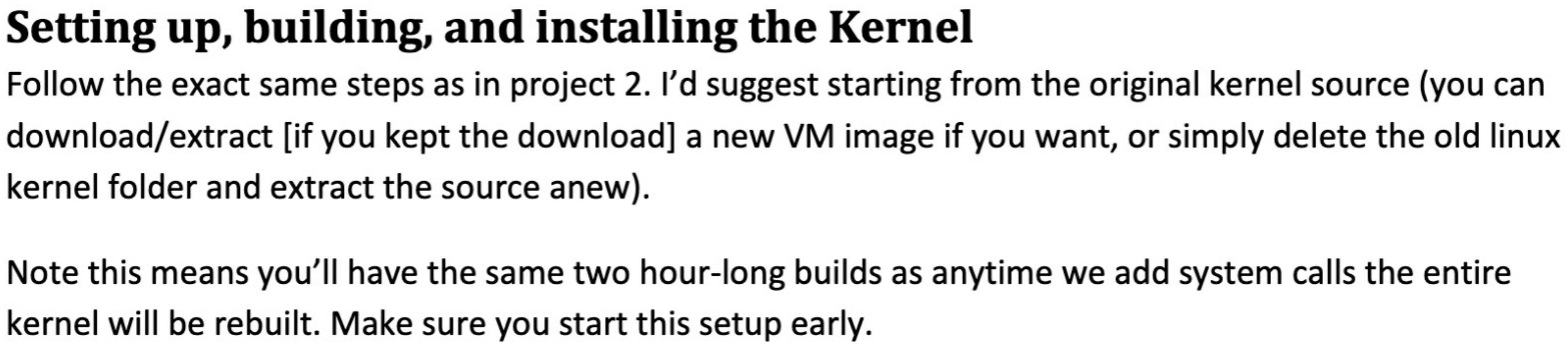
Project 4 Discussion

DIY Semaphores

Setup

- Same as Project 2
- So, start early!!!!



Setting up, building, and installing the Kernel

Follow the exact same steps as in project 2. I’d suggest starting from the original kernel source (you can download/extract [if you kept the download] a new VM image if you want, or simply delete the old linux kernel folder and extract the source anew).

Note this means you’ll have the same two hour-long builds as anytime we add system calls the entire kernel will be rebuilt. Make sure you start this setup early.

Overall

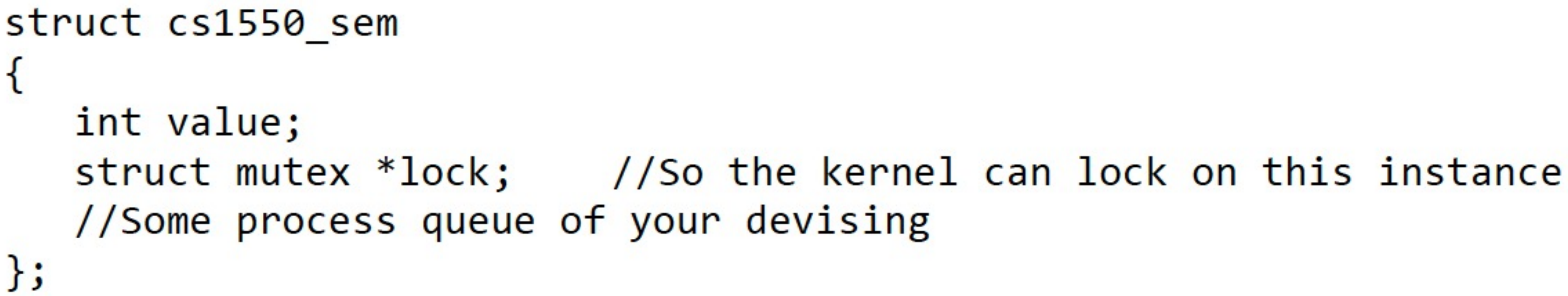
- Simulate producer-consumer problem

- Write the user space code in :
  - prodcons.c

- Add new syscalls in the kernel space by modifying the following:
  - sys.c
  - syscalls.h
  - unistd.h
  - syscall_64.tbl
  - syscall_32.tbl

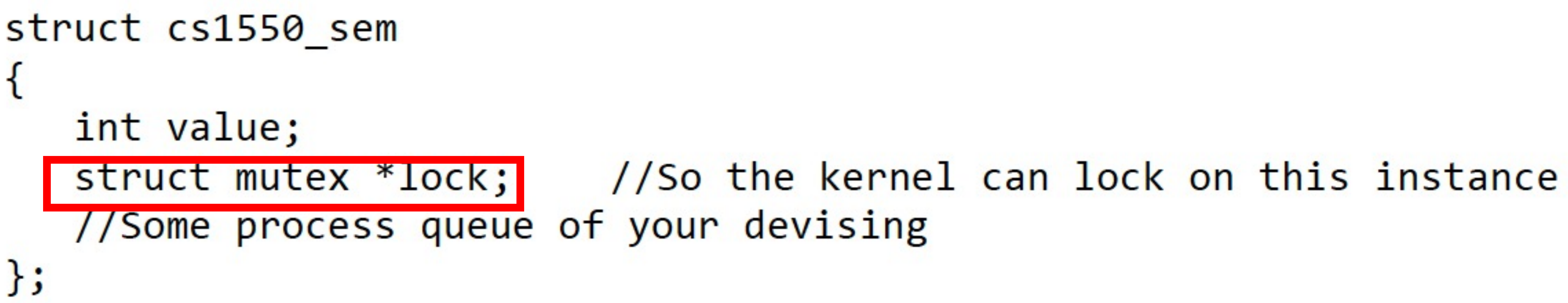
sys.c

Semaphore



```
struct cs1550_sem
{
   int value;
   struct mutex *lock;    //So the kernel can lock on this instance
   //Some process queue of your devising
};
```

What is *struct mutex*?



```
struct cs1550_sem
{
   int value;
   struct mutex *lock;    //So the kernel can lock on this instance
   //Some process queue of your devising
};
```

- Predefined struct defined in *linux/mutex.h*
- *lock* will be used for maintaining atomicity in your up() and down() syscalls

What should be the process queue?

- A queue for storing processes

- What type of data should each node of the queue store?
  - *struct task_struct* is the structure for process metadata
  - The global variable *current* is a **pointer** to the current process
    - Each node should be able to store *current*
    - So, the data structure of each node in the queue should be?

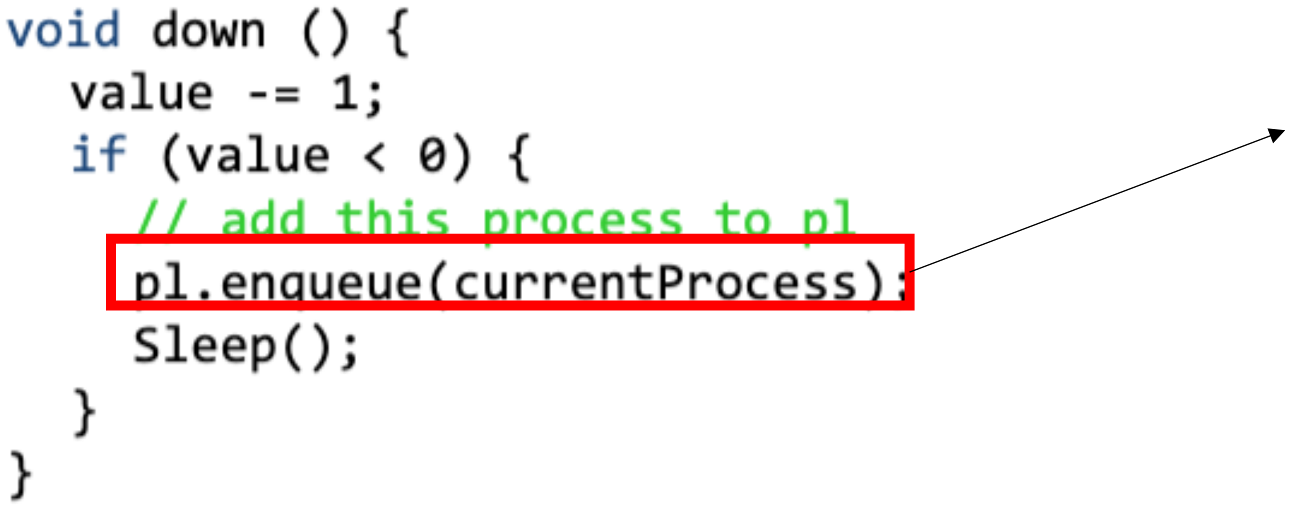
- You can build a queue of your own devising
  - Hint: You can use Linked List with head and tail

Syscalls

asmlinkage long sys_cs1550_seminit(struct cs1550_sem *sem, int value)

- Inititalize the semaphore value
- Initialize sem->lock
  - Allocate memory for *sem->lock*
    - Use kmalloc
  - Initialize using *mutex_init(sem->lock)*

asmlinkage long sys_cs1550_down(struct cs1550_sem *sem)

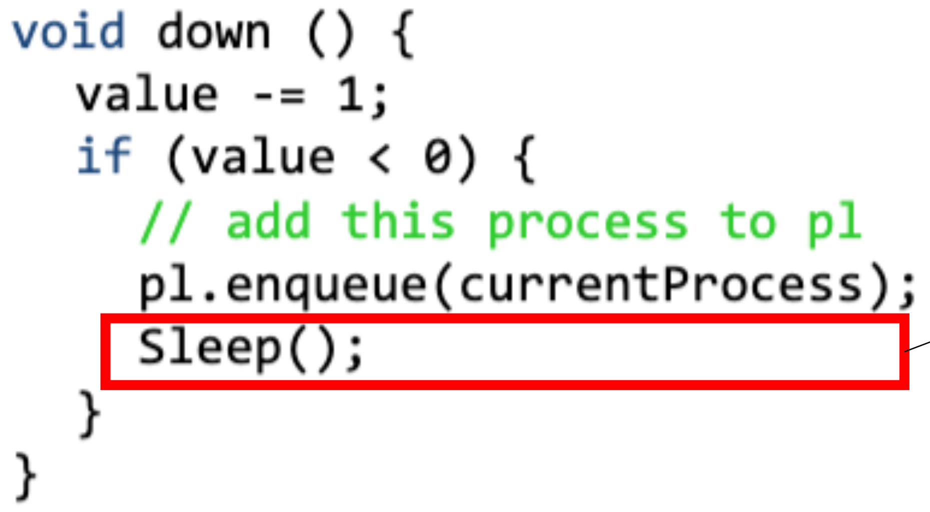


```
void down () {
  value -= 1;
  if (value < 0) {
    // add this process to pl
    pl.enqueue(currentProcess);
    Sleep();
  }
}
```

Enqueue current process (maintained by global pointer variable *current*) to the tail of your queue

Pseudocode from Dr. Misurda's slides

asmlinkage long sys_cs1550_down(struct cs1550_sem *sem)



```
void down () {
   value -= 1;
   if (value < 0) {
      // add this process to pl
      pl.enqueue(currentProcess);
      Sleep();
   }
}
```

Make the current process go to sleep

- set_current_state(TASK_INTERRUPTIBLE);
- schedule();

Pseudocode from Dr. Misurda's slides

asmlinkage long sys_cs1550_down(struct cs1550_sem *sem)

```
sys_cs1550_down(struct cs1550_sem *sem)
{
        mutex_lock(sem->lock)



        //CRITICAL REGION



        mutex_unlock(sem->lock)
}
```

Please note that you may need to call mutex_unlock() more than once if there is some part of your code that is not reachable

Do not return without unlocking (Example)

```
if {
     …
     return x;
}
mutex_unlock(sem->lock);
return y;
```

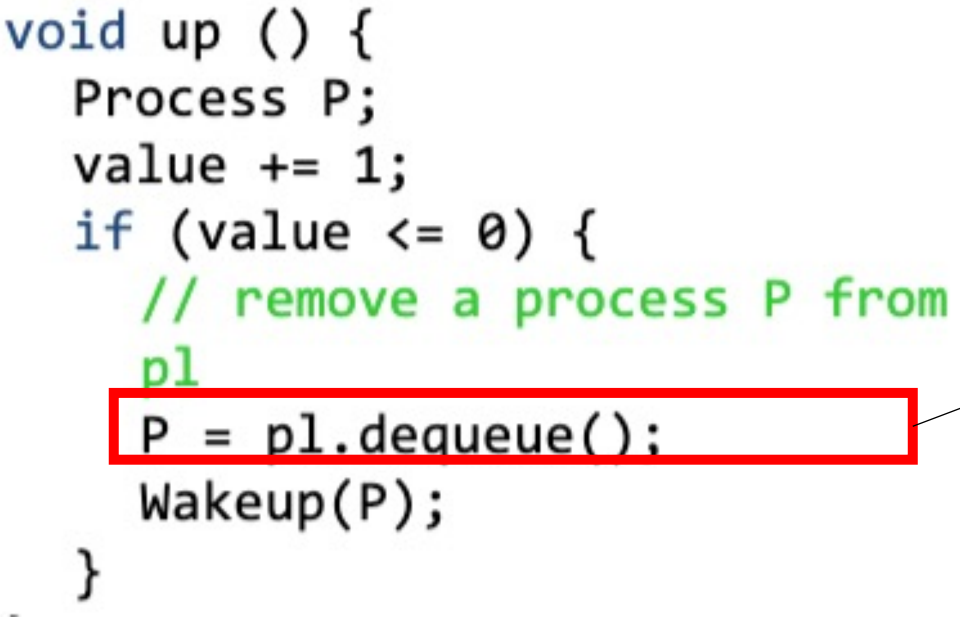
✗

```
result = y;
if {
     …
     result = x;
}
mutex_unlock(sem->lock);

return result;
```

✓

asmlinkage long sys_cs1550_up(struct cs1550_sem *sem)

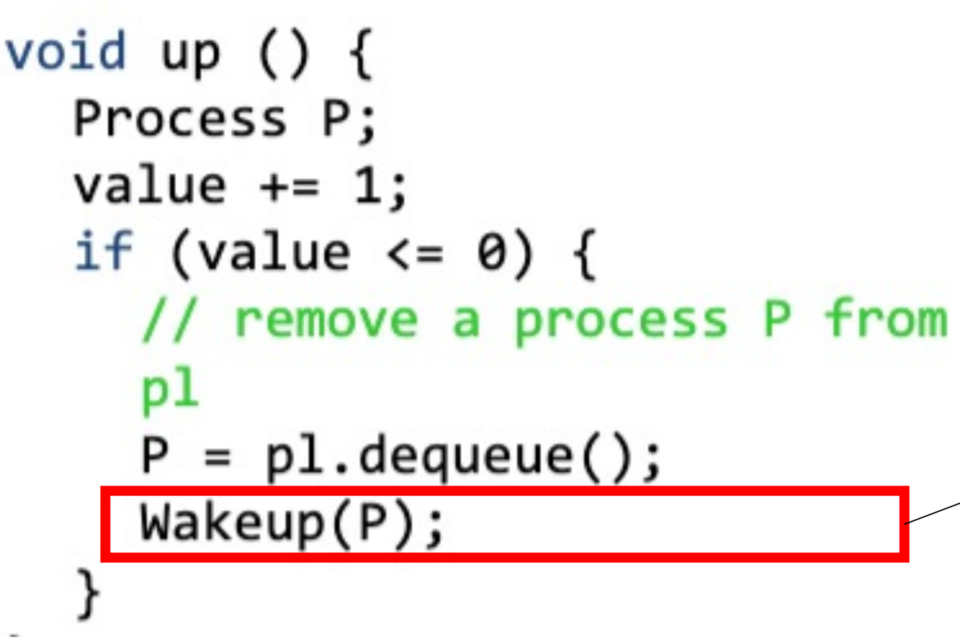


```
void up () {
  Process P;
  value += 1;
  if (value <= 0) {
    // remove a process P from
    pl
    P = pl.dequeue();
    Wakeup(P);
  }
```

Dequeue the process from the head of the queue

Pseudocode from Dr. Misurda's slides

asmlinkage long sys_cs1550_up(struct cs1550_sem *sem)



```
void up () {
  Process P;
  value += 1;
  if (value <= 0) {
    // remove a process P from
    pl
    P = pl.dequeue();
    Wakeup(P);
  }
```

Use

wake_up_process(sleeping_task);

- *sleeping_task* is the process that you just dequeued
- Type *struct task_struct**

Pseudocode from Dr. Misurda's slides

asmlinkage long sys_cs1550_up(struct cs1550_sem *sem)

```
sys_cs1550_up(struct cs1550_sem *sem)
{
        mutex_lock(sem->lock)



        //CRITICAL REGION



        mutex_unlock(sem->lock)
}
```

Please note that you may need to call mutex_unlock() more than once if there is some part of your code that is not reachable

Do not return without unlocking (Example)

```
if {
    …
    return x;
}
mutex_unlock(sem->lock);
return y;
```

✗

```
result = y;
if {
    …
    result = x;
}
mutex_unlock(sem->lock);

return result;
```

✓

prodcons.c

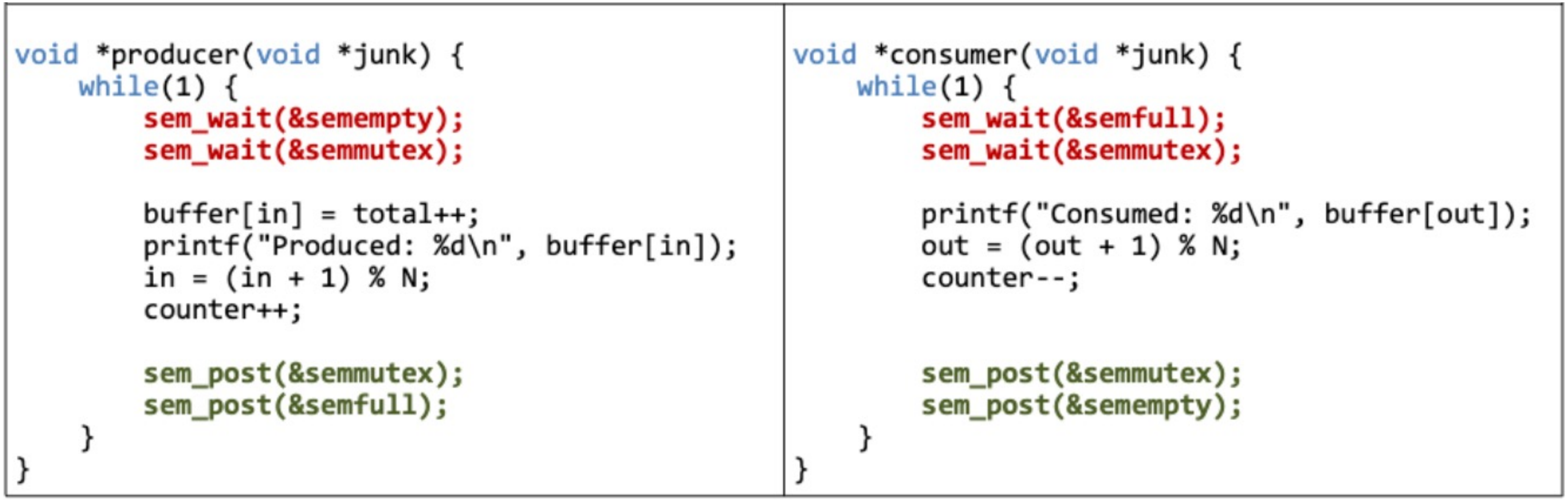
mmap()

- call mmap() once to store
  - Shared semaphore variables
  - Shared buffer
- Use MAP_SHARED|MAP_ANONYMOUS

- Size = Size of three semaphore variables + Size of buffer

- Use pointer arithmetic to access the mmap’d space
  - *struct cs1550_sem* empty = ptr*
  - *struct cs1550_sem* full = empty + 1*
  - and so on …

prodcons.c – producer and consumer



```
void *producer(void *junk) {
    while(1) {
        sem_wait(&sempempty);
        sem_wait(&semmutex);

        buffer[in] = total++;
        printf("Produced: %d\n", buffer[in]);
        in = (in + 1) % N;
        counter++;

        sem_post(&semmutex);
        sem_post(&semfull);
    }
}
```

```
void *consumer(void *junk) {
    while(1) {
        sem_wait(&semfull);
        sem_wait(&semmutex);

        printf("Consumed: %d\n", buffer[out]);
        out = (out + 1) % N;
        counter--;


        sem_post(&semmutex);
        sem_post(&sempempty);
    }
}
```

You can consider this as kind of a Pseudocode for your implementation. Make sure to use the up() and down() syscalls that you added in sys.c

Pseudocode from Dr. Misurda's slides

Important Points

- For producer
  - fork() as many times as the number of producers
  - Each child process should execute the producer part of your code
- For consumer
  - fork() as many times as the number of consumers
  - Each child process should execute the consumer part of your code
- Call the wait() function in the parent process to wait for the child processes to finish execution