LECTURE 16
FEB 8, 2024

Semaphore

- Synchronization primitive like condition variables
- Semaphore is a variable with an underlying counter
- Two functions on a semaphore variable

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);
```

– **Up/post** increments the counter and wakes up one of the processes sleeping/blocked on the semaphore

– **Down/wait** decrements the counter and blocks the calling thread if the resulting value is negative

- A semaphore with init value 1 acts as a simple lock(binary semaphore =mutex)

```
sem_t m;
sem_init(&m, 0, X); // initialize to X; what should X be?

sem_wait(&m);
// critical section here
sem_post(&m);
```

POSIX semaphores

- sem_init
- sem_wait
- sem_post
- sem_getvalue
- sem_destroy

Semaphores for ordering

- Can be used to set order of execution between threads like CV
- Example: parent waiting for child (init = 0)

```
sem_t s;

void *child(void *arg) {
    printf("child\n");
    sem_post(&s); // signal here: child is done
    return NULL;
}

int main(int argc, char *argv[]) {
    sem_init(&s, 0, X); // what should X be?
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL);
    sem_wait(&s); // wait here for child
    printf("parent: end\n");
    return 0;
}
```

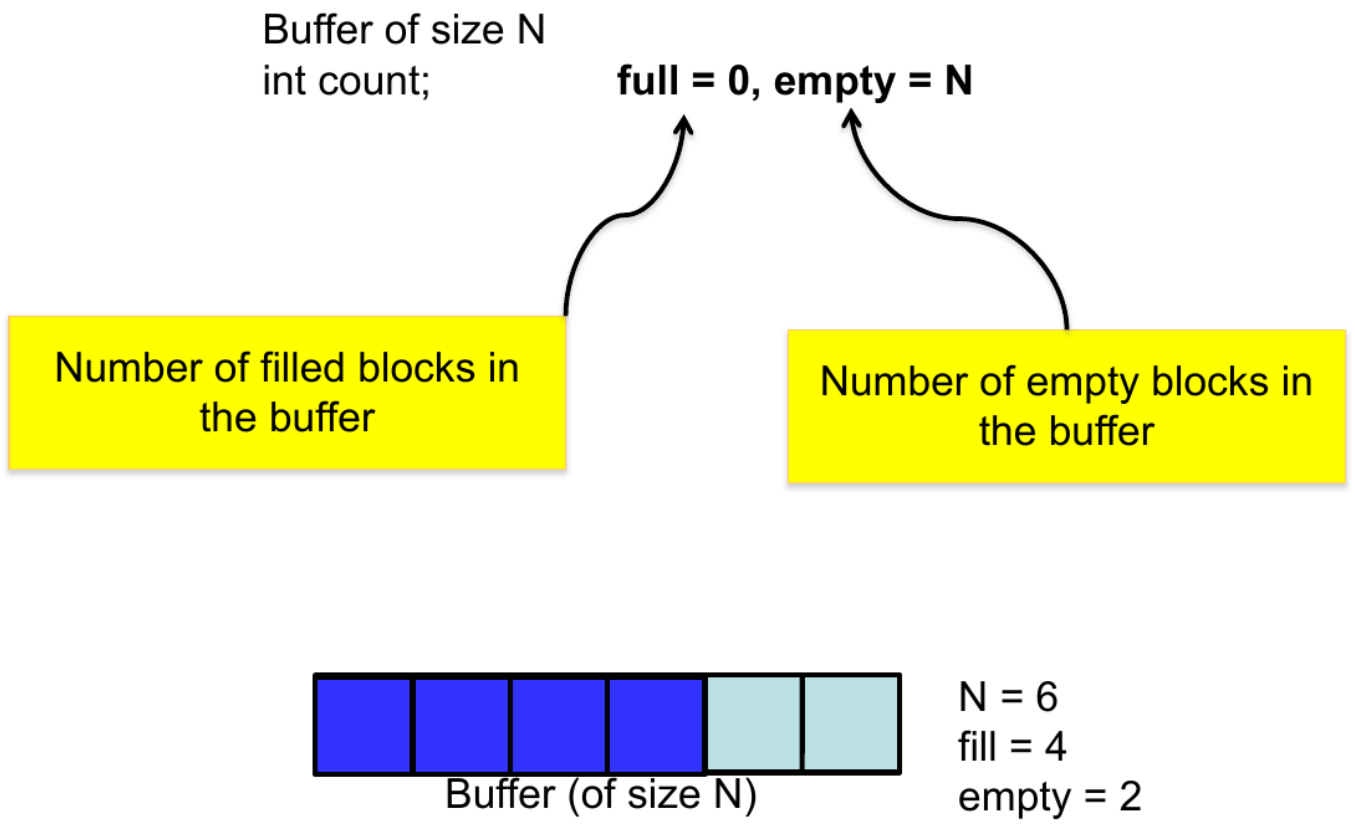
Producer – Consumer Problem

- Need two semaphores for signaling

–One to track empty slots, and make producer wait if no more empty slots (What it should be initialized to ?)

–One to track full slots, and make consumer wait if no more full slots (What it should be initialized to ?)

- One semaphore to act as mutex for buffer

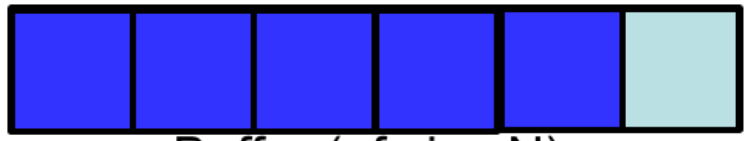

Buffer of size N
int count;
full = 0, empty = N
Number of filled blocks in the buffer
Number of empty blocks in the buffer
Buffer (of size N)
N = 6
fill = 4
empty = 2

full = 0, empty = N

```
void producer(){
    while(TRUE){
        item = produce_item();
        down(empty);

        insert_item(item); // into buffer

        up(full);
    }
}
```

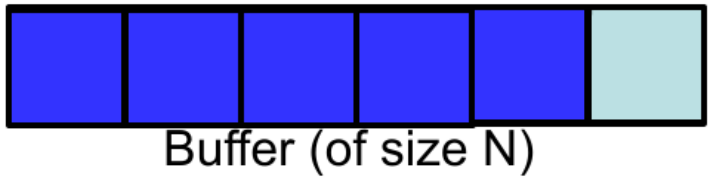


Buffer (of size N)

N = 6

fill = 5 4

empty = 1 2



N = 6

fill = 5

empty = 1

```
void consumer(){
    while(TRUE){
        down(full);

        item = remove_item(); // from buffer

        up(empty);
        consume_item(item);
    }
}
```

The FULL Buffer

```
void producer(){
    while(TRUE){
        item = produce_item();
        down(empty);

        insert_item(item); // into buffer

        up(full);
    }
}
```

```
void consumer(){
    while(TRUE){
        down(full);

        item = remove_item(); // from buffer

        up(empty);
        consume_item(item);
    }
}
```

Buffer (of size N)

N = 6
fill = 6

empty = 0

The Empty Buffer

```
void producer(){
   while(TRUE){
      item = produce_item();
      down(empty);

     insert_item(item); // into buffer

      up(full);
   }
}
```

```
void consumer(){
   while(TRUE){
      down(full);

      item = remove_item(); // from buffer

      up(empty);
      consume_item(item);
   }
}
```

Buffer (of size N)

N = 6
fill = 0

empty = 6

Serializing Access to the Buffer

```
void producer(){
    while(TRUE){
        item = produce_item();
        down(empty);
        lock(mutex)
        insert_item(item); // into buffer
        unlock(mutex)
        up(full);
    }
}
```

```
void consumer(){
    while(TRUE){
        down(full);
        lock(mutex)
        item = remove_item(); // from buffer
        unlock(mutex)
        up(empty);
        consume_item(item);
    }
}
```

Buffer (of size N)

N = 6
fill = 3

empty = 3

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);         // Line P1
        sem_wait(&mutex);         // Line P1.5 (MUTEX HERE)
        put(i);                   // Line P2
        sem_post(&mutex);         // Line P2.5 (AND HERE)
        sem_post(&full);          // Line P3
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);          // Line C1
        sem_wait(&mutex);         // Line C1.5 (MUTEX HERE)
        int tmp = get();          // Line C2
        sem_post(&mutex);         // Line C2.5 (AND HERE)
        sem_post(&empty);         // Line C3
        printf("%d\n", tmp);
    }
}
```

```
int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX are empty
    sem_init(&full, 0, 0);    // 0 are full
    // ...
}
```

Deadlock?

What if lock is acquired before signaling?

- Waiting thread sleeps with mutex and the signaling thread can never wake it up

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);       // Line P0 (NEW LINE)
        sem_wait(&empty);       // Line P1
        put(i);                 // Line P2
        sem_post(&full);        // Line P3
        sem_post(&mutex);       // Line P4 (NEW LINE)
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);       // Line C0 (NEW LINE)
        sem_wait(&full);        // Line C1
        int tmp = get();        // Line C2
        sem_post(&empty);       // Line C3
        sem_post(&mutex);       // Line C4 (NEW LINE)
        printf("%d\n", tmp);
    }
}
```

Multiple people are entering and exiting a room that has a light switch.

You are asked to write a program to model the people in this situation as threads in an application.

You must fill in the functions `onEnter()` and `onExit()` that are invoked by a thread/person when the person enters and exits a room respectively.

We require that the first person entering a room must turn on the light switch by invoking the function `turnOnSwitch()`, while the last person leaving the room must turn off the switch by invoking `turnOffSwitch()`.

You must invoke these functions suitably in your code. You need to use `Mutex` to implement synchronization and achieve the desired goal.

variables: mutex, count

```
onEnter():
lock(mutex)
count++
if(count==1) turnOnSwitch()
unlock(mutex)

onExit():
lock(mutex)
count--
if(count==0) turnOffSwitch()
unlock(mutex)
```

A host of a party has invited N >2 guests to his house. The host does not wish to open the door of his house multiple times to let guests in. Instead, he wishes that all **N** guests, even though they may arrive at different times to his door, wait for each other and enter the house all at once. The host and guests are represented by threads in a multi-threaded program.

Given below is the pseudocode for the host thread, where the host waits for all guests to arrive, then calls `openDoor()`, and signals a condition variable once.

You must write the corresponding code for the guest threads. The guests must wait for all N of them to arrive and for the host to open the door, and must call `enterHouse()` only after that.

You must ensure that all N waiting guests enter the house after the door is opened. You must use only locks and condition variables for synchronization.

The following variables are used in this solution: `lock m`, condition variables `cvhost` and `cvguest`, and integer `guestcount` (initialized to 0). You must not use any other variables in the guest for synchronization.

```
//host
lock(m)
while(guest_count < N)
      wait(cv_host, m)
openDoor()
signal(cv_guest)
unlock(m)
```

//guest thread

---

lock(m)
guest_count++
if(guest_count == N)
signal(cv_host)
wait(cv_guest, m)
signal(cv_guest)
unlock(m)
`enterHouse()`

Consider a scenario where a bus picks up waiting passengers from a bus stop periodically. The bus has a capacity of K.
The bus arrives at the bus stop, allows up to K waiting passengers (fewer if less than K are waiting) to board, and then departs. Passengers have to wait for the bus to arrive and then board it. Passengers who arrive at the bus stop after the bus has arrived should not be allowed to board, and should wait for the next time the bus arrives.
The bus and passengers are represented by threads in a program.
The passenger thread should call the function board() after the passenger has boarded and the bus should invoke depart() when it has boarded the desired number of passengers and is ready to depart.
The threads share the following variables, none of which are implicitly updated by functions like board() or depart().

```
mutex = semaphore initialized to 1.
bus_arrived = semaphore initialized to 0.
passenger_boarded = semaphore initialized to 0.
waiting_count = integer initialized to 0.
```

Below is given synchronized code for the passenger thread. You should not modify this in any way.

```
down(mutex)
waiting_count++
up(mutex)
down(bus_arrived)
board()
up(passenger_boarded)
```

Write down the corresponding synchronized code for the bus thread that achieves the correct behavior specified above.
The bus should board the correct number of passengers, based on its capacity and the number of those waiting. The bus should correctly board these passengers by calling up/down on the semaphores suitably.
The bus code should also update waiting count as required.
Once boarding completes, the bus thread should call depart().
You can use any extra local variables in the code of the bus thread, like integers, loop indices and so on.
However, you must not use any other extra synchronization primitives.

//Passenger thread

```
down(mutex)
N = min(waiting_count, K)
for i= 1 to N
    up(bus_arrived)
    down(passenger_boarded)
waiting_count = waiting_count - N
up(mutex)
depart()
```