CS304 Operating Systems

DR GAYATHRI ANANTHANARAYANAN

gayathri@iitdh.ac.in

Materials in these slides have been borrowed from textbooks and existing operating systems courses

8th Feb 2021, Conditional Variables

- Locks allow one type of synchronization between threads–mutual exclusion
- Another common requirement in multi-threaded applications

–waiting and signaling –E.g., Thread T1 wants to continue only after T2 has finished some task

- Can accomplish this by busy-waiting on some variable, but inefficient
- Need a new synchronization primitive: condition variables

```
void *child(void *arg) {
    printf("child\n");
    // XXX how to indicate we are done?
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    // XXX how to wait for child?
    printf("parent: end\n");
    return 0;
}
```

```
parent: begin
child
parent: end
```

```
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    while (done == 0)
        ; // spin
    printf("parent: end\n");
    return 0;
}
```

Conditon Variable

A condition variable (CV) is a queue that a thread can put itself into when waiting on some condition

- Another thread that makes the condition true can signal the CV to wake up a waiting thread
- Pthreads provides CV for user programs –OS has a similar functionality of wait/signal for kernel threads
- Signal wakes up one thread, signal broadcast wakes up all waiting threads

Wait() and Signal()

A condition variable has two operations associated with it: **`wait()`** and **`signal()`**.

The **`wait()`** call is executed when a thread wishes to put itself to sleep.

The **`signal()`** call is executed when a thread has changed something in the program and thus wants to wake a sleeping thread waiting on this condition.

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

Example

```
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

Why while loop?

In the example code, why do we check condition before calling wait?–In case the child has already run and done is true, then no need to wait

- Why check condition with “while” loop and not “if”?

–To avoid corner cases of thread being woken up even when condition not true (may be an issue with some implementations)

Why use lock when calling wait?

What if no lock is held when calling wait/signal?

- Race condition: missed wakeup

–Parent checks done to be 0, decides to sleep, interrupted

–Child runs, sets done to 1, signals, but no one sleeping yet

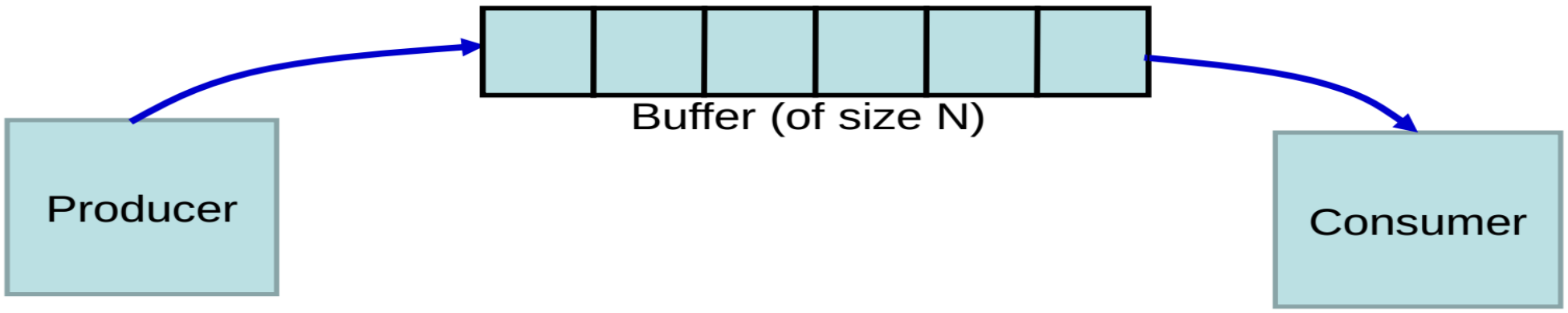
–Parent now resumes and goes to sleep forever

- Lock must be held when calling wait and signal with CV
- The wait function releases the lock before putting thread to sleep, so lock is available for signaling thread

```
void thr_exit() {
    done = 1;
    Pthread_cond_signal(&c);
}

void thr_join() {
    if (done == 0)
        Pthread_cond_wait(&c);
}
```

The Producer/Consumer Problem

- Also known as *Bounded buffer Problem*
- Producer produces and stores in buffer, Consumer consumes from buffer
- Trouble when
  - Producer produces, but buffer is full
  - Consumer consumes, but buffer is empty



Producer/Consumer problem

- A common pattern in multi-threaded programs
- Example: in a multi-threaded web server, one thread accepts requests from the network and puts them in a queue. Worker threads get requests from this queue and process them.
- Setup: one or more producer threads, one or more consumer threads, a shared buffer of bounded size

Attempt-1

```
int loops; // must initialize somewhere...
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // p1
        if (count == 1)                        // p2
            Pthread_cond_wait(&cond, &mutex);  // p3
        put(i);                                // p4
        Pthread_cond_signal(&cond);            // p5
        Pthread_mutex_unlock(&mutex);          // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // c1
        if (count == 0)                        // c2
            Pthread_cond_wait(&cond, &mutex);  // c3
        int tmp = get();                       // c4
        Pthread_cond_signal(&cond);            // c5
        Pthread_mutex_unlock(&mutex);          // c6
        printf("%d\n", tmp);
    }
}
```

```
int buffer;
int count = 0; // initially, empty

void put(int value) {
    assert(count == 0);
    count = 1;
    buffer = value;
}

int get() {
    assert(count == 1);
    count = 0;
    return buffer;
}
```

T_{c1}	State	T_{c2}	State	T_p	State	Count	*Comment*
c1	Run		Ready		Ready	0	
c2	Run		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Run	0	
	Sleep		Ready	p2	Run	0	
	Sleep		Ready	p4	Run	1	Buffer now full
	Ready		Ready	p5	Run	1	T_{c1} awoken
	Ready		Ready	p6	Run	1	
	Ready		Ready	p1	Run	1	
	Ready		Ready	p2	Run	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Run		Sleep	1	T_{c2} sneaks in ...
	Ready	c2	Run		Sleep	1	
	Ready	c4	Run		Sleep	0	... and grabs data
	Ready	c5	Run		Ready	0	T_p awoken
	Ready	c6	Run		Ready	0	
c4	Run		Ready		Ready	0	Oh oh! No data

```
int loops; // must initialize somewhere...
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // p1
        if (count == 1)                        // p2
            Pthread_cond_wait(&cond, &mutex);  // p3
        put(i);                                // p4
        Pthread_cond_signal(&cond);            // p5
        Pthread_mutex_unlock(&mutex);          // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // c1
        if (count == 0)                        // c2
            Pthread_cond_wait(&cond, &mutex);  // c3
        int tmp = get();                       // c4
        Pthread_cond_signal(&cond);            // c5
        Pthread_mutex_unlock(&mutex);          // c6
        printf("%d\n", tmp);
    }
}
```

Attempt-2

```
int loops;
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);             // p1
        while (count == 1)                      // p2
            Pthread_cond_wait(&cond, &mutex);   // p3
        put(i);                                 // p4
        Pthread_cond_signal(&cond);             // p5
        Pthread_mutex_unlock(&mutex);           // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);             // c1
        while (count == 0)                      // c2
            Pthread_cond_wait(&cond, &mutex);   // c3
        int tmp = get();                        // c4
        Pthread_cond_signal(&cond);             // c5
        Pthread_mutex_unlock(&mutex);           // c6
        printf("%d\n", tmp);
    }
}
```

Thread Trace

T_{c1}	State	T_{c2}	State	T_p	State	Count	*Comment*
c1	Run		Ready		Ready	0	
c2	Run		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Run		Ready	0	
	Sleep	c2	Run		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Run	0	
	Sleep		Sleep	p2	Run	0	
	Sleep		Sleep	p4	Run	1	Buffer now full
	Ready		Sleep	p5	Run	1	T_{c1} awoken
	Ready		Sleep	p6	Run	1	
	Ready		Sleep	p1	Run	1	
	Ready		Sleep	p2	Run	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Run		Sleep		Sleep	1	Recheck condition
c4	Run		Sleep		Sleep	0	T_{c1} grabs data
c5	Run		Ready		Sleep	0	Oops! Woke T_{c2}
c6	Run		Ready		Sleep	0	
c1	Run		Ready		Sleep	0	
c2	Run		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Run		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	Everyone asleep...

```
int loops;
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // p1
        while (count == 1)                     // p2
            Pthread_cond_wait(&cond, &mutex);  // p3
        put(i);                                // p4
        Pthread_cond_signal(&cond);            // p5
        Pthread_mutex_unlock(&mutex);          // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // c1
        while (count == 0)                     // c2
            Pthread_cond_wait(&cond, &mutex);  // c3
        int tmp = get();                       // c4
        Pthread_cond_signal(&cond);            // c5
        Pthread_mutex_unlock(&mutex);          // c6
        printf("%d\n", tmp);
    }
}
```

Attempt-3

```
cond_t  empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 1)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&fill, &mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Solution

```
int buffer[MAX];
int fill_ptr = 0;
int use_ptr  = 0;
int count    = 0;

void put(int value) {
    buffer[fill_ptr] = value;
    fill_ptr = (fill_ptr + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use_ptr];
    use_ptr = (use_ptr + 1) % MAX;
    count--;
    return tmp;
}
```

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // p1
        while (count == MAX)                   // p2
            Pthread_cond_wait(&empty, &mutex); // p3
        put(i);                                // p4
        Pthread_cond_signal(&fill);            // p5
        Pthread_mutex_unlock(&mutex);          // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // c1
        while (count == 0)                     // c2
            Pthread_cond_wait(&fill, &mutex);  // c3
        int tmp = get();                       // c4
        Pthread_cond_signal(&empty);           // c5
        Pthread_mutex_unlock(&mutex);          // c6
        printf("%d\n", tmp);
    }
}
```

Semaphore

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

- Proposed by Dijkstra in 1965
- Functions down and up must be atomic
- down also called **P** (Proberen Dutch for try)
- up also called **V** (Verhogen, Dutch form make higher)
- Can have different variants
  - Such as blocking, non-blocking
- If S is initially set to 1,
  - Blocking semaphore similar to a Mutex
  - Non-blocking semaphore similar to a spinlock

```
void down(int *S){
  while( *S <= 0);
  *S--;
}

void up(int *S){
   *S++;
}
```

Atomic

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

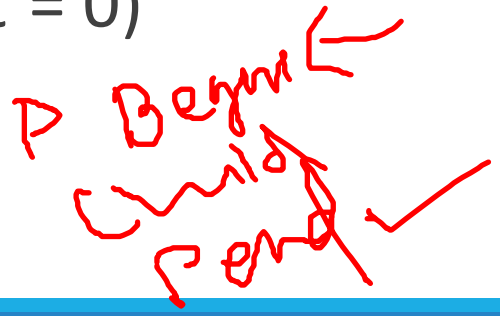
```
sem_t m;
sem_init(&m, 0, X); // initialize to X; what should X be?

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;
}
```

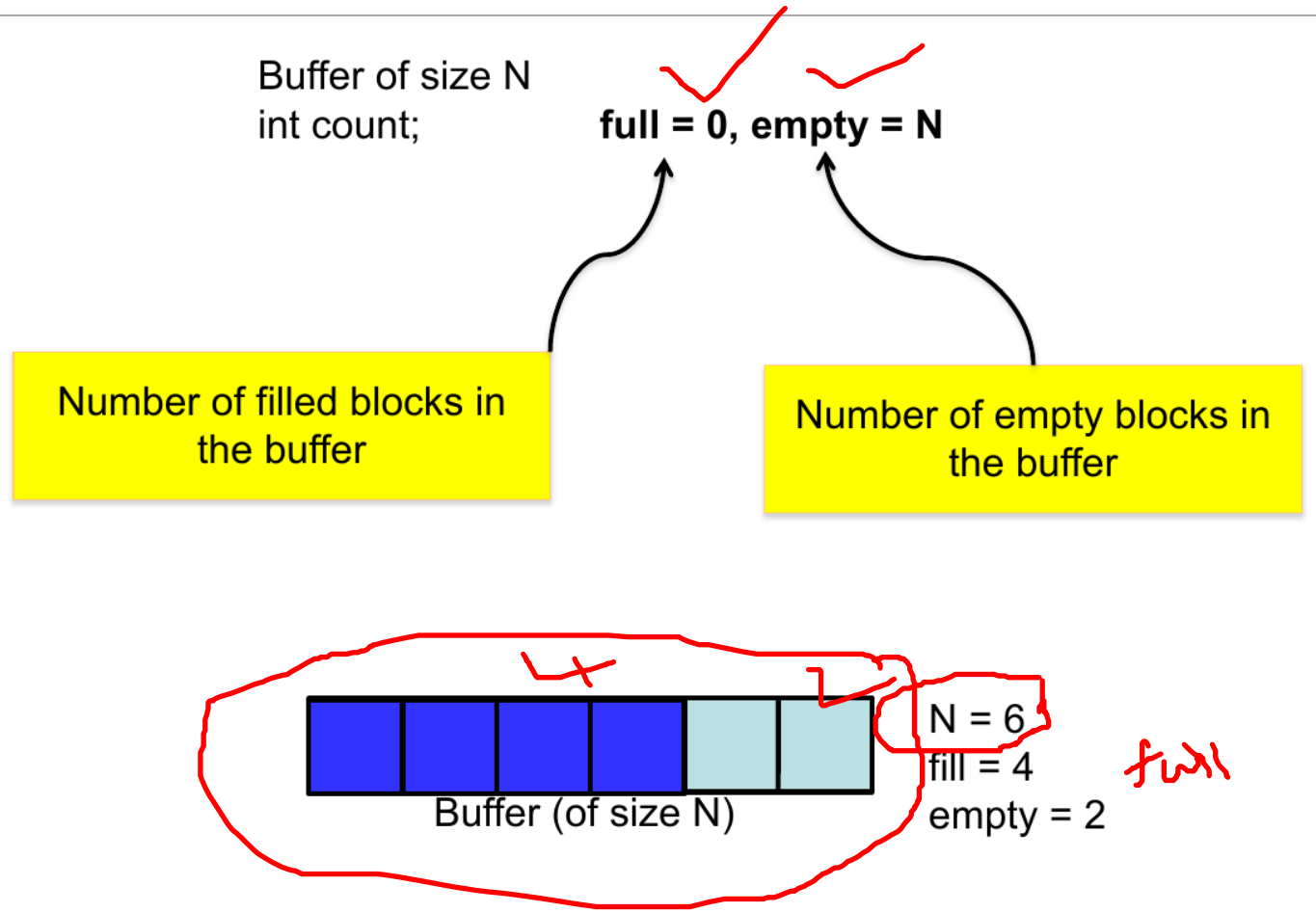
Producer – Consumer Problem

- Need two semaphores for signaling

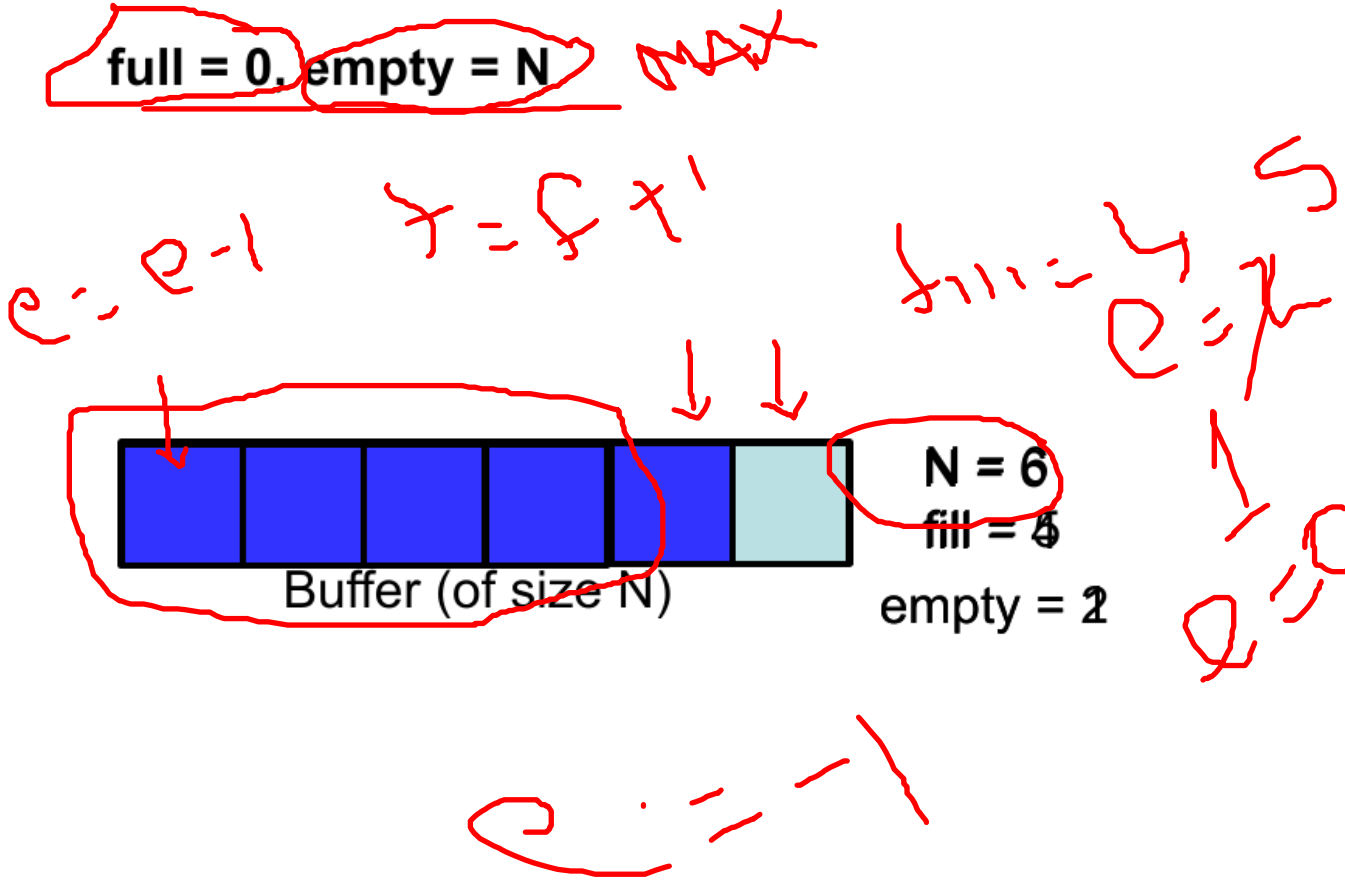
–One to track empty slots, and make producer wait if no more empty slots

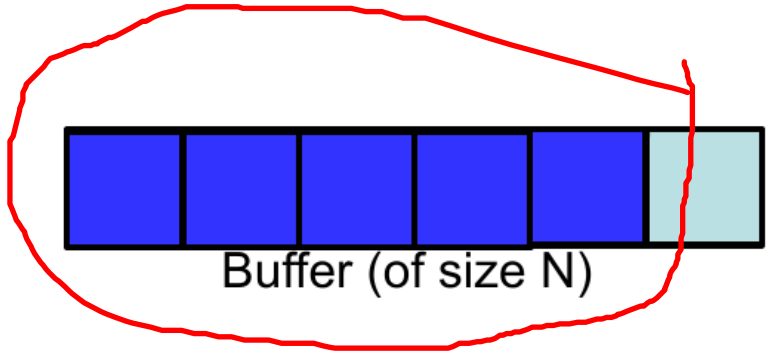
–One to track full slots, and make consumer wait if no more full slots

- 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

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

        insert_item(item); // into buffer

        up(full);
    }
}
```





N = 6

fill = 5

empty = 2

```
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);
    }
}
```

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
    // ...
}
```

Semaphore

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);
    }
}
```