CS 471 Operating Systems

Yue Cheng

George Mason University

Fall 2019

Semaphores

Semaphores

- Introduced by **E. W. Dijkstra**
- Motivation: Avoid busy waiting by blocking a process execution until some condition is satisfied
- Two operations are defined on a semaphore variable s:
  - `sem_wait(s)` (also called `P(s)` or `down(s)`)
  - `sem_post(s)` (also called `V(s)` or `up(s)`)

Semaphores

- Introduced by **E. W. Dijkstra**
- Motivation: Avoid busy waiting by blocking a process execution until some condition is satisfied
- Two operations are defined on a semaphore variable s:
  - `sem_wait(s)` (also called `P(s)` or `down(s)`)
  - `sem_post(s)` (also called `V(s)` or `up(s)`)

OS/161

Semaphore Operations

- Conceptually, a semaphore has an integer value. This value is greater than or equal to 0

- `sem_wait(s):`

```
 s.value-- ;   /*  Executed atomically */
 /* wait/block if s.value < 0 (or negative) */
```

- A process/thread executing the wait operation on a semaphore with value < 0 being blocked until the semaphore's value becomes greater than 0
  - **No busy waiting**

- `sem_post(s):`

```
 s.value++;  /* Executed atomically */
 /* if one or more process/thread waiting, wake one */
```

Semaphore Operations (cont.)

- If multiple processes/threads are blocked on the same semaphore 's', only one of them will be awakened when another process performs post(s) operation

- Who will have higher priority?

Semaphore Operations (cont.)

- If multiple processes/threads are blocked on the same semaphore '`s`', only one of them will be awakened when another process performs post(s) operation

- Who will have higher priority?
  - **A**: FIFO, or whatever queuing strategy

Attacking Critical Section Problem with Semaphores

- Declare and define a semaphore:

```
sem_t s;
sem_init(&s, 0, 1);   /* initially s = 1 */
```

Binary semaphore, which is a lock

- Routine of Thread 0 & 1:

```
do {
    sem_wait(s);
      critical section
    sem_post(s);
      remainder section
} while (1);
```

Attacking Critical Section Problem with Semaphores

- **Single** thread using a **binary** semaphore

Value of Semaphore	Thread 0	Thread 1
1		

Attacking Critical Section Problem with Semaphores

- **Single** thread using a **binary** semaphore

Value of Semaphore	Thread 0	Thread 1
1		
1	call `sem_wait()`	
0	`sem_wait()` returns	

Attacking Critical Section Problem with Semaphores

- **Single** thread using a **binary** semaphore

Value of Semaphore	Thread 0	Thread 1
1		
1	call sem_wait()	
0	sem_wait() returns	
0	(crit sect)	
0	call sem_post()	

Attacking Critical Section Problem with Semaphores

- **Single** thread using a **binary** semaphore

Value of Semaphore	Thread 0	Thread 1
1		
1	call sem_wait()	
0	sem_wait() returns	
0	(crit sect)	
0	call sem_post()	
1	sem_post() returns	

Attacking Critical Section Problem with Semaphores

- **Two** threads using a **binary** semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready

Attacking Critical Section Problem with Semaphores

- **Two** threads using a **binary** semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call sem_wait()	Running		Ready
0	sem_wait() returns	Running		Ready
0	(crit sect: begin)	Running		Ready

Attacking Critical Section Problem with Semaphores

- **Two** threads using a **binary** semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call `sem_wait()`	Running		Ready
0	`sem_wait()` returns	Running		Ready
0	`(crit sect: begin)`	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running

Attacking Critical Section Problem with Semaphores

- **Two** threads using a **binary** semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call `sem_wait()`	Running		Ready
0	`sem_wait()` returns	Running		Ready
0	`(crit sect: begin)`	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	call `sem_wait()`	Running
-1		Ready	`decrement sem`	Running
-1		Ready	`(sem<0)→sleep`	Sleeping

Attacking Critical Section Problem with Semaphores

- **Two** threads using a **binary** semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call `sem_wait()`	Running		Ready
0	`sem_wait()` returns	Running		Ready
0	`(crit sect: begin)`	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	call `sem_wait()`	Running
-1		Ready	`decrement sem`	Running
-1		Ready	`(sem<0)→sleep`	Sleeping
-1		Running	*Switch→T0*	Sleeping

Attacking Critical Section Problem with Semaphores

- **Two** threads using a **binary** semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call `sem_wait()`	Running		Ready
0	`sem_wait()` returns	Running		Ready
0	`(crit sect: begin)`	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	call `sem_wait()`	Running
-1		Ready	`decrement sem`	Running
-1		Ready	`(sem<0)→sleep`	Sleeping
-1		Running	*Switch→T0*	Sleeping
-1	`(crit sect: end)`	Running		Sleeping
-1	call `sem_post()`	Running		Sleeping
0	`increment sem`	Running		Sleeping
0	`wake(T1)`	Running		Ready
0	`sem_post()` returns	Running		Ready

Attacking Critical Section Problem with Semaphores

- **Two** threads using a **binary** semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call `sem_wait()`	Running		Ready
0	`sem_wait()` returns	Running		Ready
0	`(crit sect: begin)`	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	call `sem_wait()`	Running
-1		Ready	`decrement sem`	Running
-1		Ready	`(sem<0)→sleep`	Sleeping
-1		Running	*Switch→T0*	Sleeping
-1	`(crit sect: end)`	Running		Sleeping
-1	call `sem_post()`	Running		Sleeping
0	`increment sem`	Running		Sleeping
0	`wake(T1)`	Running		Ready
0	`sem_post()` returns	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running

Attacking Critical Section Problem with Semaphores

- **Two** threads using a **binary** semaphore

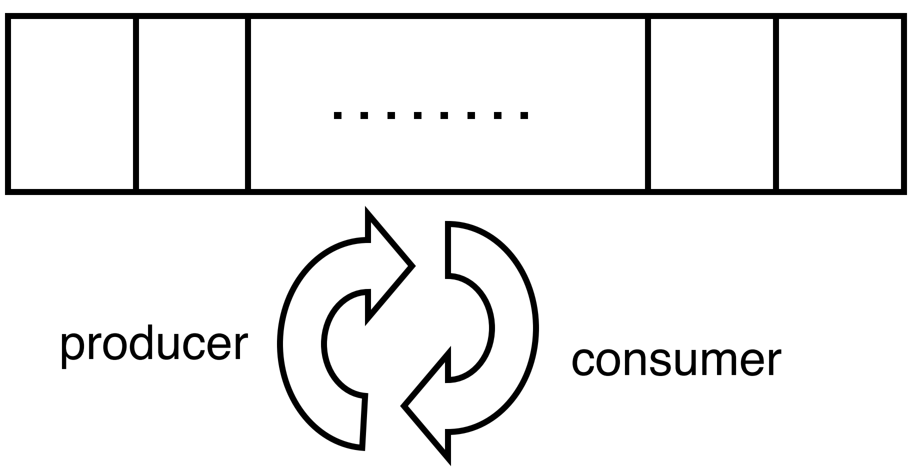
Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call `sem_wait()`	Running		Ready
0	`sem_wait()` returns	Running		Ready
0	`(crit sect: begin)`	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	call `sem_wait()`	Running
-1		Ready	`decrement sem`	Running
-1		Ready	`(sem<0)→sleep`	Sleeping
-1		Running	*Switch→T0*	Sleeping
-1	`(crit sect: end)`	Running		Sleeping
-1	call `sem_post()`	Running		Sleeping
0	`increment sem`	Running		Sleeping
0	`wake(T1)`	Running		Ready
0	`sem_post()` returns	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	`sem_wait()` returns	Running
0		Ready	`(crit sect)`	Running
0		Ready	call `sem_post()`	Running
1		Ready	`sem_post()` returns	Running

Classical Problems of Synchronization

- Producer-Consumer Problem
  - Semaphore version
  - Condition Variable
    - A CV-based version
- Readers-Writers Problem
- Dining-Philosophers Problem

Producer-Consumer Problem

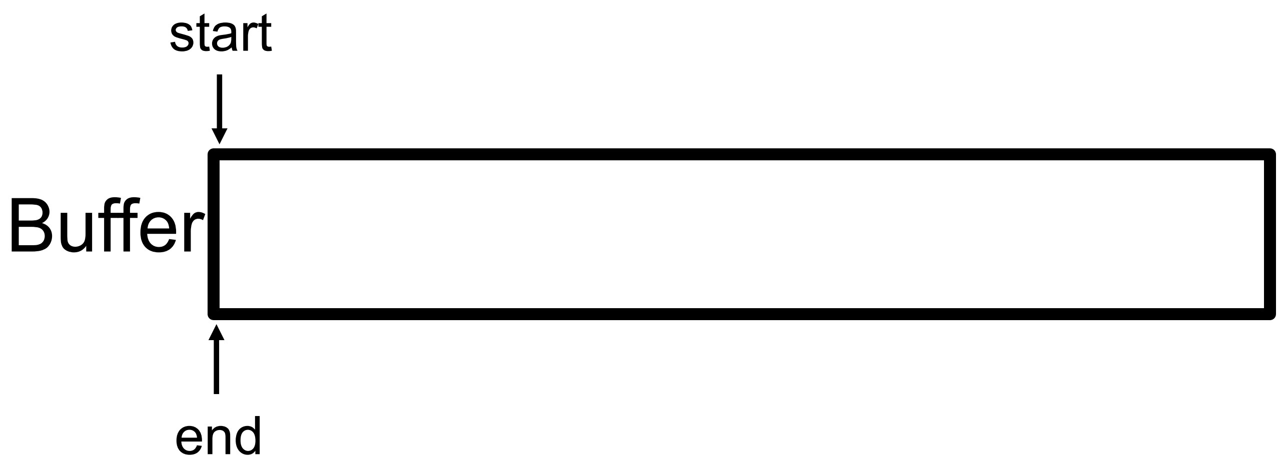
- The bounded-buffer producer-consumer problem assumes that there is a buffer of size *N*
- The producer process puts items to the buffer area
- The consumer process consumes items from the buffer
- The producer and the consumer execute concurrently



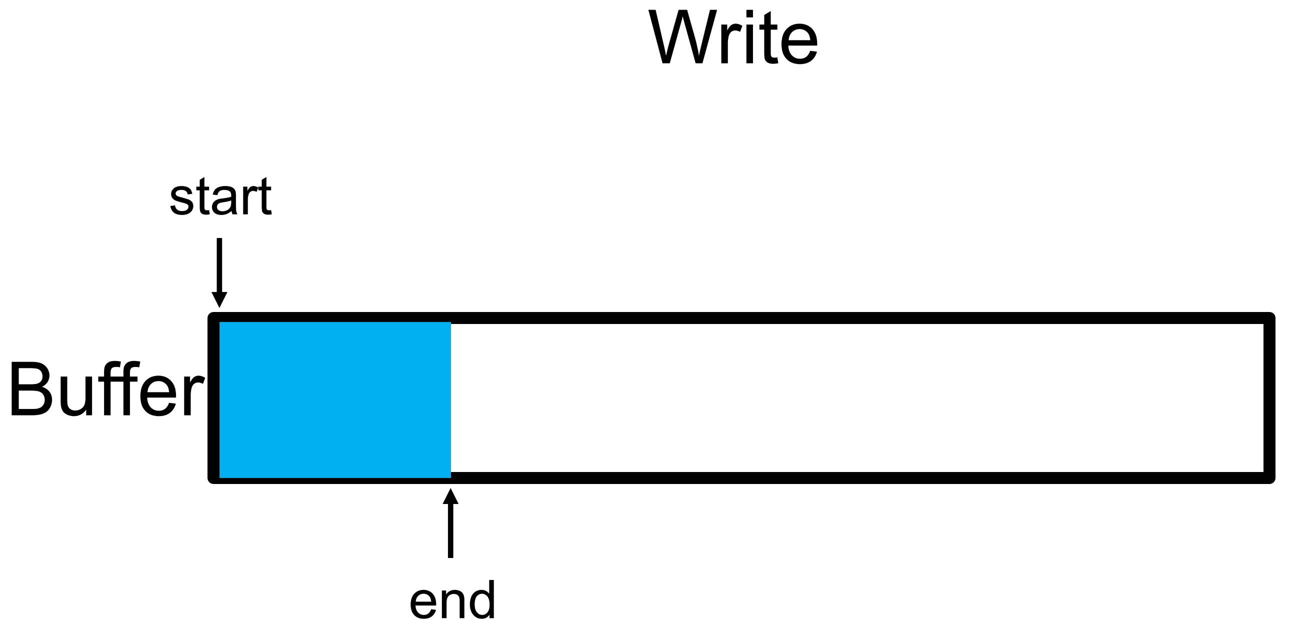
Example: UNIX Pipes

- A pipe may have many writers and readers
- Internally, there is a finite-sized buffer
- Writers add data to the buffer
- Readers remove data from the buffer

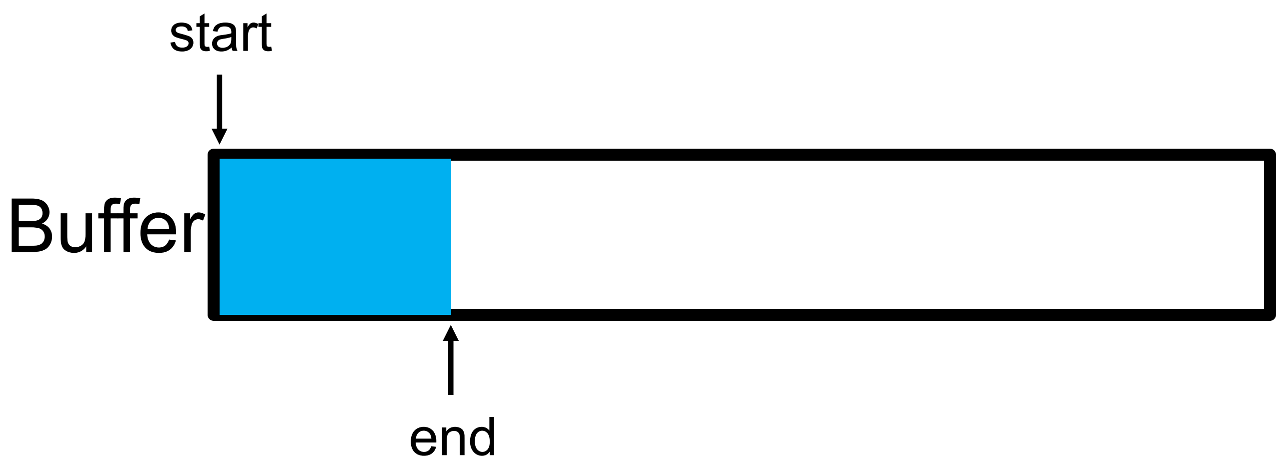
Example: UNIX Pipes



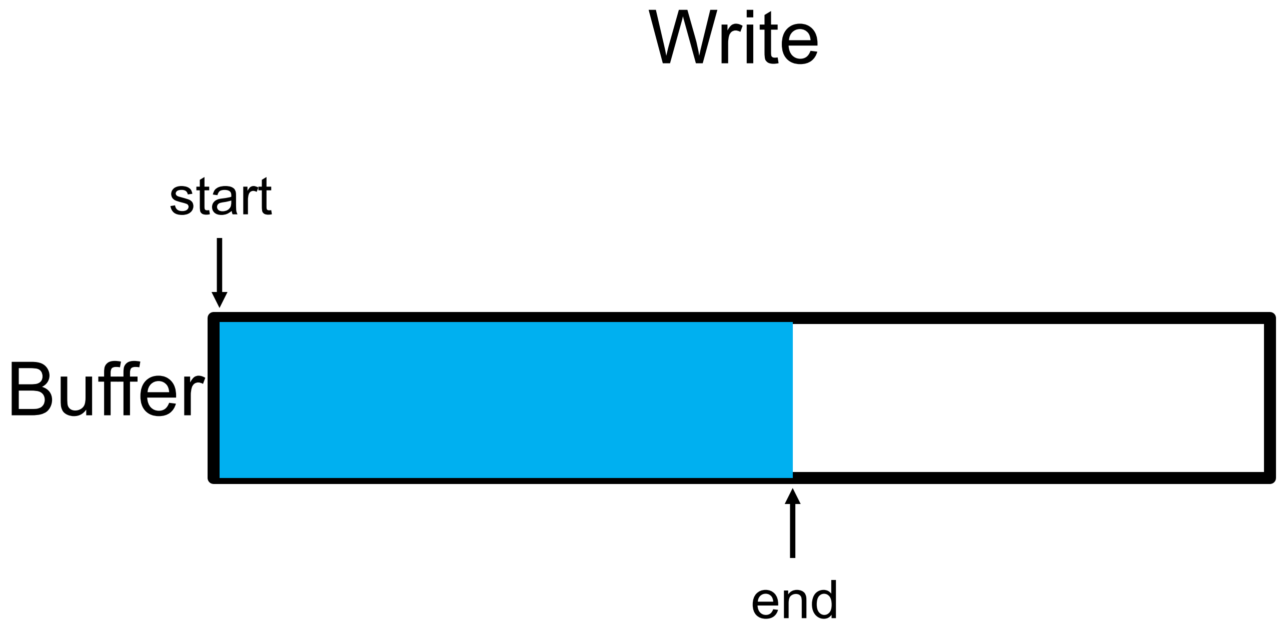
Example: UNIX Pipes



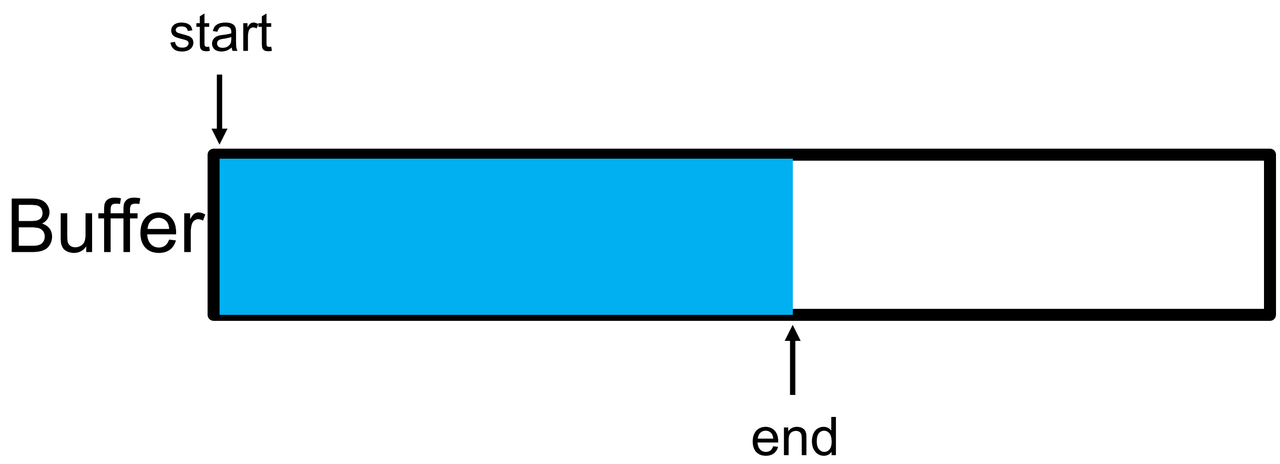
Example: UNIX Pipes



Example: UNIX Pipes

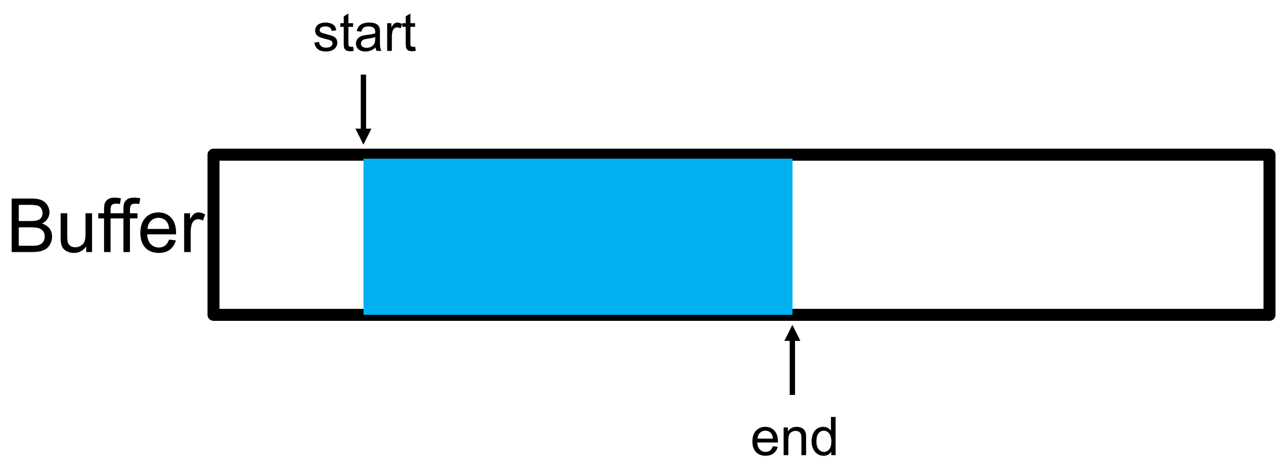


Example: UNIX Pipes

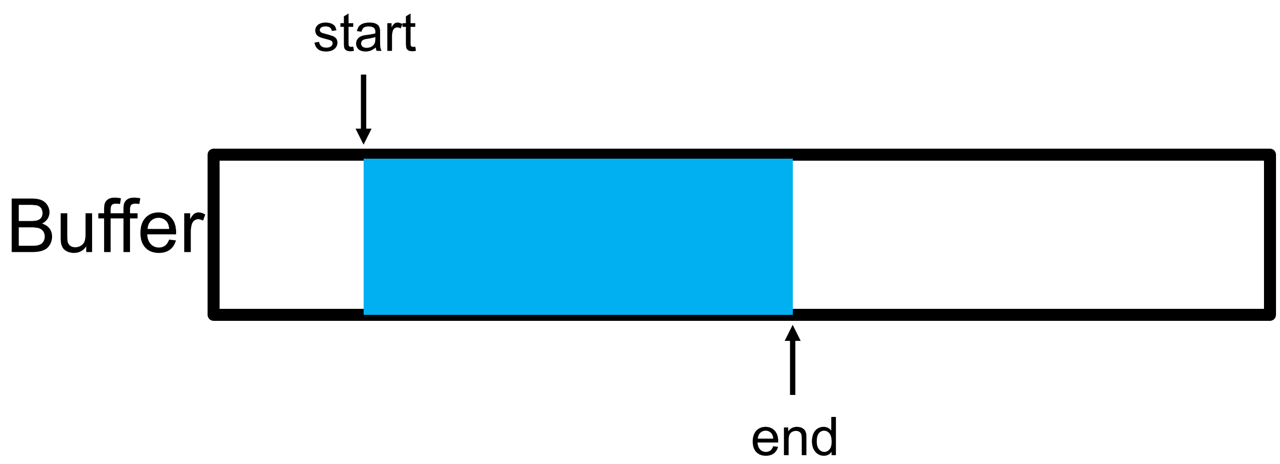


Example: UNIX Pipes

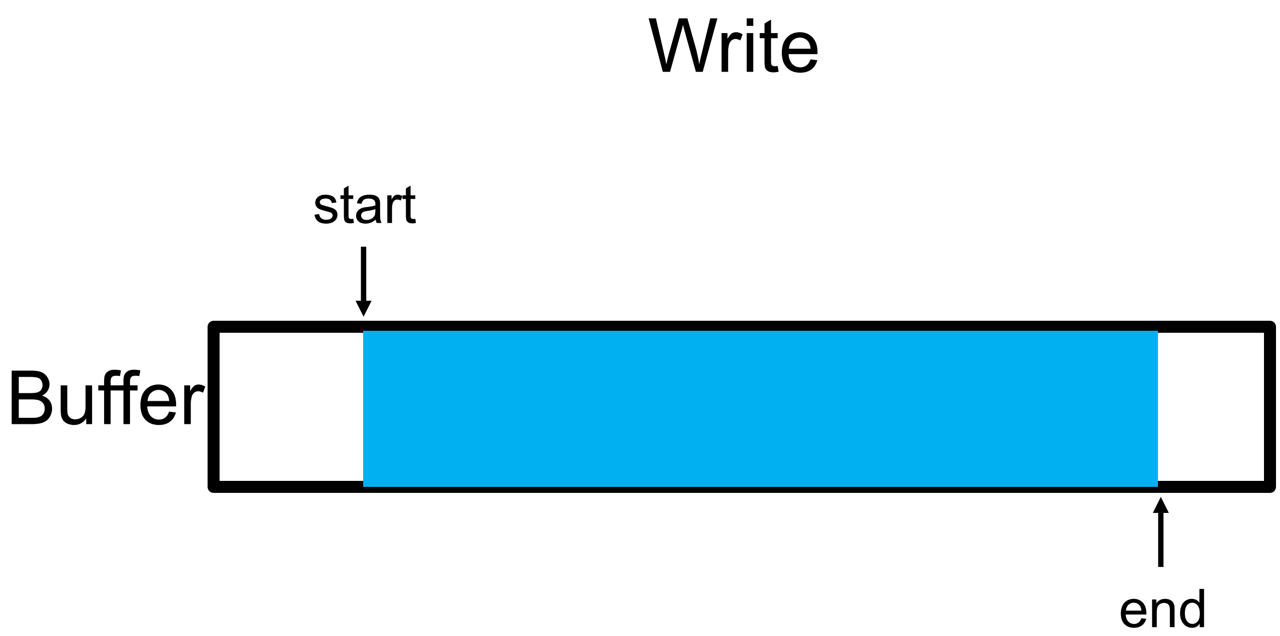
Read



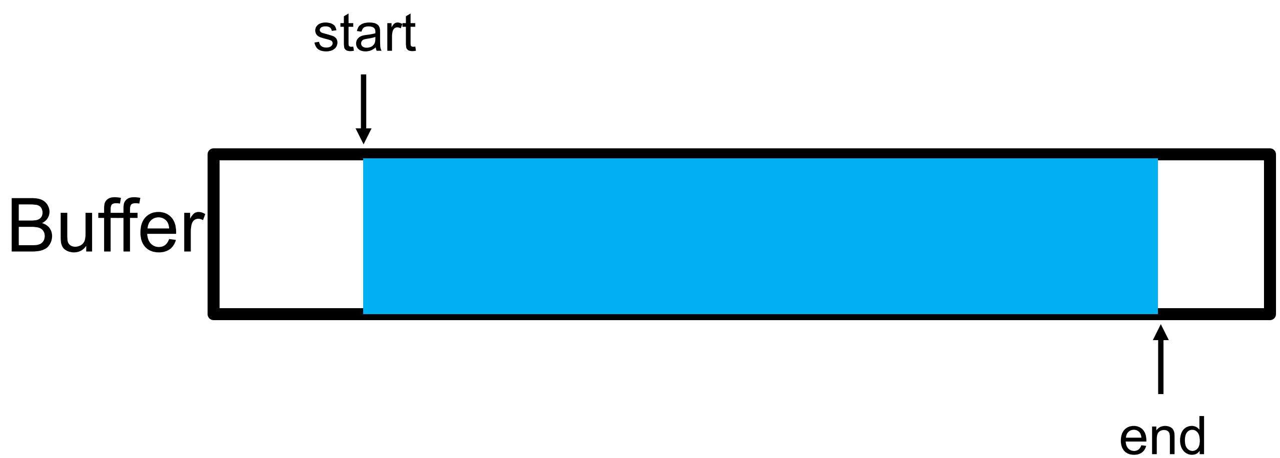
Example: UNIX Pipes



Example: UNIX Pipes

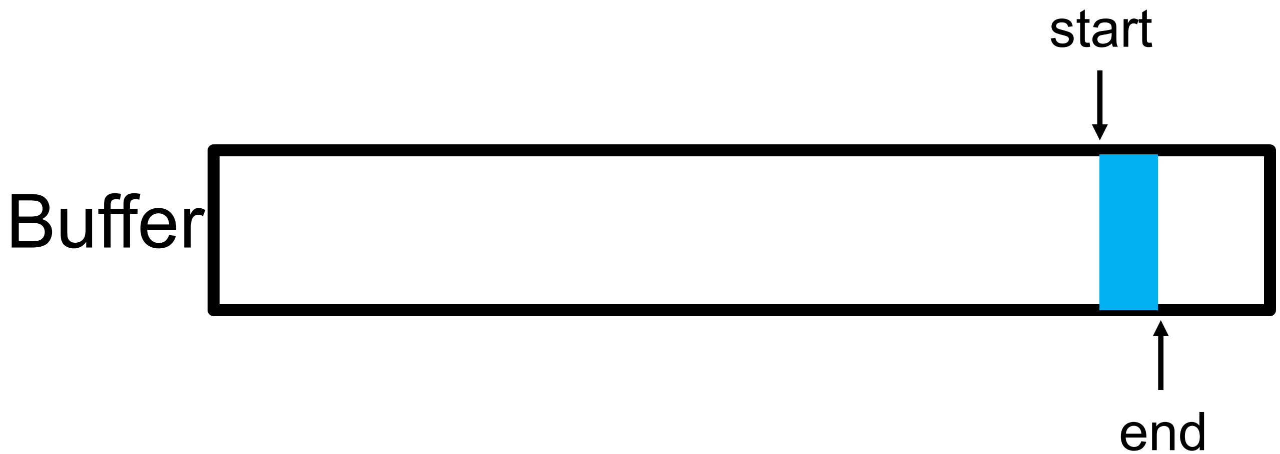


Example: UNIX Pipes



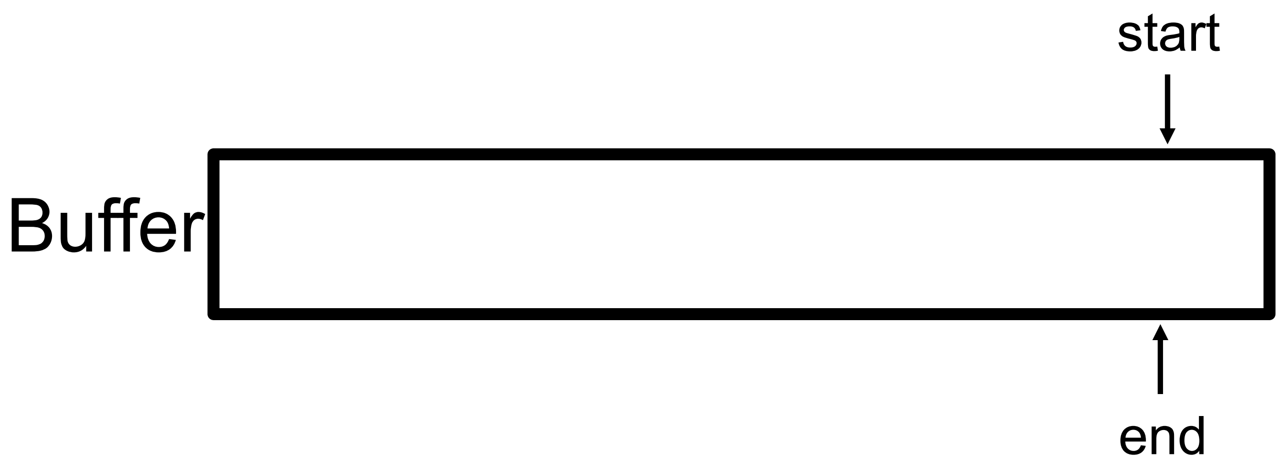
Example: UNIX Pipes

Read



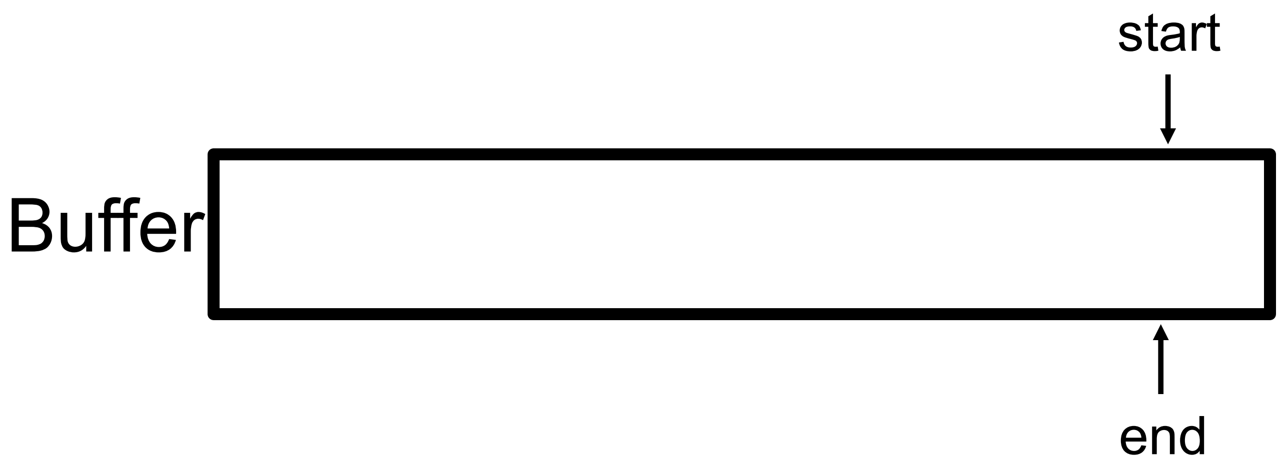
Example: UNIX Pipes

Read



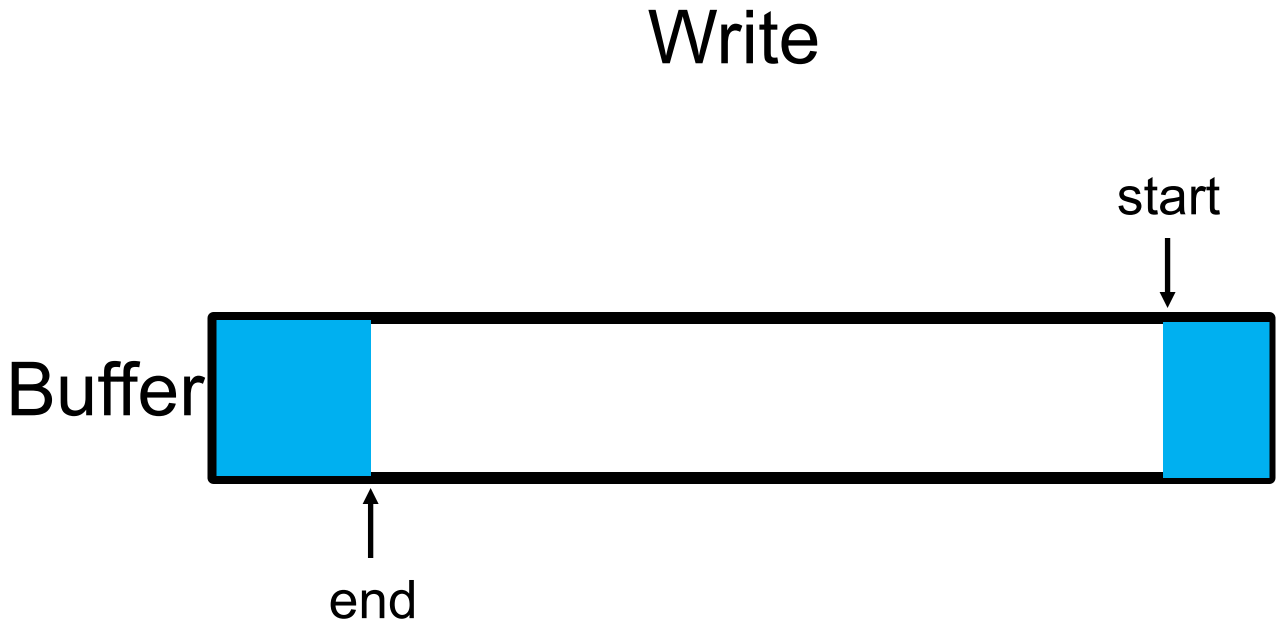
Example: UNIX Pipes

Read



Note: reader must **wait**

Example: UNIX Pipes

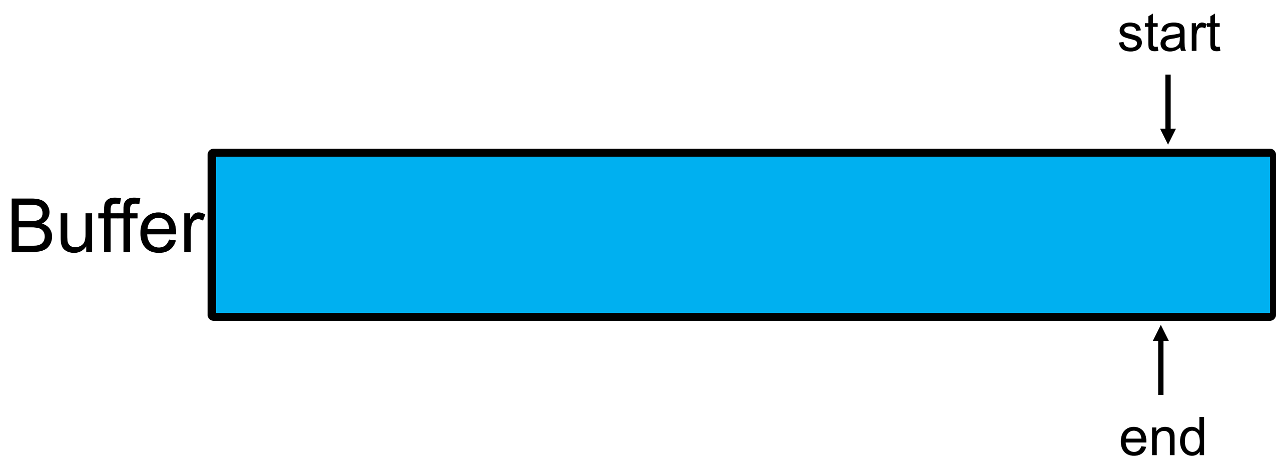


Example: UNIX Pipes

Write

Example: UNIX Pipes

Write



Note: writer must **wait**

Example: UNIX Pipes

- Implementation
  - Reads/writes to buffer require **locking**
  - When buffers are **full**, writers (producers) **must wait**
  - When buffers are **empty**, readers (consumers) **must wait**

Example: UNIX Pipes

Demo

```
% ps aux | less
```

Pipe

```
% cat file | grep <str>
```

Pipe

Producer-Consumer Model: Parameters

- Shared data:

```
sem_t full, empty;
```

- Initially:

```
full = 0        /* The number of full buffers */
empty = MAX     /* The number of empty buffers */
```

First Attempt: MAX = 1

```c
sem_t empty;
sem_t full;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);            // line P1
        put(i);                      // line P2
        sem_post(&full);             // line P3
    }
}

void *consumer(void *arg) {
    int i, tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);             // line C1
        tmp = get();                 // line C2
        sem_post(&empty);            // line C3
        printf("%d\n", tmp);
    }
}

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

```c
int buffer[MAX];
int fill = 0;
int use  = 0;

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

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

Put and Get routines

First Attempt: MAX = 10?

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);            // line P1
        put(i);                      // line P2
        sem_post(&full);             // line P3
    }
}

void *consumer(void *arg) {
    int i, tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);             // line C1
        tmp = get();                 // line C2
        sem_post(&empty);            // line C3
        printf("%d\n", tmp);
    }
}

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

```
int buffer[MAX];
int fill = 0;
int use  = 0;

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

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

Put and Get routines

First Attempt: MAX = 10?

fill = 0

empty = 10

Producer 0: **Running**

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

Producer 1: Runnable

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

First Attempt: MAX = 10?

fill = 0

empty = 9

Producer 0: **Running**

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

Producer 1: Runnable

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

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

First Attempt: MAX = 10?

fill = 0

empty = 9

Producer 0: **Running**

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

Producer 1: Runnable

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

```
void put(int value) {
      buffer[fill] = value;
      Interrupted …
      fill = (fill + 1) % MAX;
}
```

First Attempt: MAX = 10?

fill = 0

empty = 9

Producer 0: **Sleeping**

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

Producer 1: Runnable

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

```
void put(int value) {
        buffer[fill] = value;
        Interrupted …
        fill = (fill + 1) % MAX;
}
```

First Attempt: MAX = 10?

fill = 0

empty = 9

Producer 0: Runnable

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

Producer 1: **Running**

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

```
void put(int value) {
       buffer[fill] = value;
       Interrupted …
       fill = (fill + 1) % MAX;
}
```

First Attempt: MAX = 10?

fill = 0
Overwrite!
empty = 8

Producer 0: Runnable

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
→       put(i);
        sem_post(&full);
    }
}
```

Producer 1: **Running**

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
→       put(i);
        sem_post(&full);
    }
}
```

```
void put(int value) {
        buffer[fill] = value;
        Interrupted …
        fill = (fill + 1) % MAX;
}
```

```
void put(int value) {
→       buffer[fill] = value;
        fill = (fill + 1) % MAX;
}
```

One More Parameter: A `mutex` lock

- Shared data:

```
sem_t full, empty;
```

- Initially:

```
full = 0;     /* The number of full buffers */
empty = MAX; /* The number of empty buffers */
mutex = 1;   /* Semaphore controlling the access
                to the buffer pool */
```

Add “Mutual Exclusion”

```
sem_t empty;
sem_t full;
sem_t mutex;

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);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
    sem_init(&full, 0, 0);    // ... and 0 are full
    sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock (NEW LINE)
    // ...
}
```

Add “Mutual Exclusion”

```
sem_t empty;
sem_t full;
sem_t mutex;

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);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
    sem_init(&full, 0, 0);    // ... and 0 are full
    sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock (NEW LINE)
    // ...
}
```

What if consumer gets to run first??

Adding "Mutual Exclusion"

mutex = 1
full = 0
empty = 10

Producer 0: Runnable

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&empty);
        put(i);
        sem_post(&full);
        sem_post(&mutex);
    }
}
```

Consumer 0: **Running**

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&full);
        int tmp = get();
        sem_post(&empty);
        sem_post(&mutex);
        printf("%d\n", tmp);
    }
}
```

Adding "Mutual Exclusion"

mutex = 0
full = 0
empty = 10

Producer 0: Runnable

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&empty);
        put(i);
        sem_post(&full);
        sem_post(&mutex);
    }
}
```

Consumer 0: **Running**

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&full);
        int tmp = get();
        sem_post(&empty);
        sem_post(&mutex);
        printf("%d\n", tmp);
    }
}
```

Consumer 0 is waiting for full to be greater than or equal to 0

Adding "Mutual Exclusion"

mutex = -1
full = -1
empty = 10

Producer 0: **Running**

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&empty);
        put(i);
        sem_post(&full);
        sem_post(&mutex);
    }
}
```

Consumer 0: Runnable

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&full);
        int tmp = get();
        sem_post(&empty);
        sem_post(&mutex);
        printf("%d\n", tmp);
    }
}
```

Consumer 0 is waiting for full to be greater than or equal to 0

Adding “Mutual Exclusion”

Deadlock!!

mutex = -1
full = -1
empty = 10

Producer 0: **Running**

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&empty);
        put(i);
        sem_post(&full);
        sem_post(&mutex);
    }
}
```

Consumer 0: Runnable

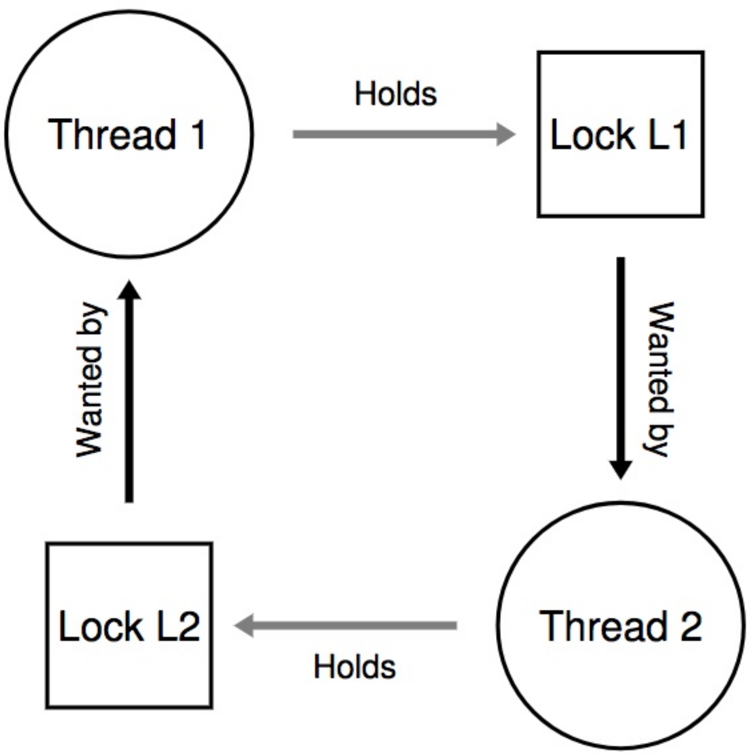
```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&full);
        int tmp = get();
        sem_post(&empty);
        sem_post(&mutex);
        printf("%d\n", tmp);
    }
}
```

Producer 0 gets stuck at acquiring `mutex` which has been locked by Consumer 0!

Consumer 0 is waiting for full to be greater than or equal to 0

Deadlocks

- A set of threads are said to be in a ***deadlock*** state when every thread in the set is waiting for an event that can be caused only by another thread in the set



A typical deadlock dependency graph

Conditions for Deadlock

- **Mutual exclusion**
  - Threads claim exclusive control of resources that require e.g., a thread grabs a lock
- **Hold-and-wait**
  - Threads hold resources allocated to them while waiting for additional resources
- **No preemption**
  - Resources cannot be forcibly removed from threads that are holding them
- **Circular wait**
  - There exists a circular chain of threads such that each holds one or more resources that are being requests by next thread in chain

Correct Mutual Exclusion

```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);            // line p1
        sem_wait(&mutex);            // line p1.5 (MOVED 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 (MOVED 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 buffers are empty to begin with...
    sem_init(&full, 0, 0);    // ... and 0 are full
    sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
    // ...
}
```

Mutex wraps just around critical section! (producer)

Mutex wraps just around critical section! (consumer)

Producer-Consumer Solution

- Make sure that
  1. The producer and the consumer do not access the buffer area and related variables at the same time
  2. No item is made available to the consumer if all the buffer slots are empty
  3. No slot in the buffer is made available to the producer if all the buffer slots are full

Condition Variables

Condition Variables

A parent waiting for its child

```
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;
}
```

Spin-based Approach

Using a shared variable, parent spins until child set it to 1

```
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;
}
```

Spin-based Approach

Using a shared variable, parent spins until child set it to 1

```
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;
}
```

What's the problem of this approach?

Condition Variables (CV)

- Definition:
  - An explicit queue that threads can put themselves when some **condition** is not as desired (by **waiting** on the condition)
  - Other thread can wake one of those waiting threads to allow them to continue (by **signaling** on the condition)
- Pthread CV

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

CV-based Approach

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

??

```
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;
}
```

??

Broken Implementation 1

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

```
void thr_exit() {
    Pthread_mutex_lock(&m);
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void thr_join() {
    Pthread_mutex_lock(&m);
    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;
}
```

Broken Implementation 1

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

```
void thr_exit() {
    Pthread_mutex_lock(&m);
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void thr_join() {
    Pthread_mutex_lock(&m);
    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;
}
```

If parent comes after child, parent sleeps forever

Broken Implementation 2

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

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

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

```
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;
}
```

Broken Implementation 2

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

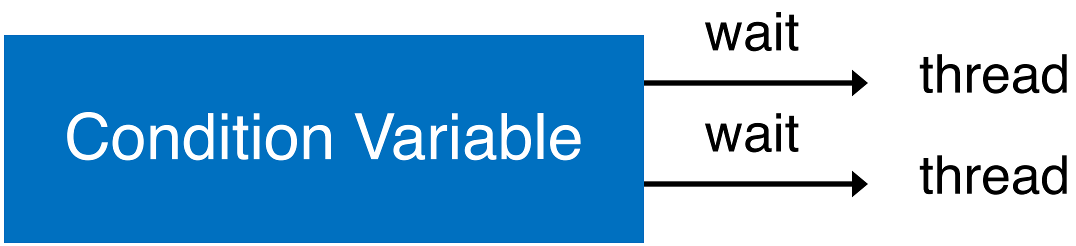
```
void thr_exit() {
    done = 1;
    Pthread_cond_signal(&c);
}

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

```
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;
}
```

No mutual exclusion, hence child may signal before parent calls `cond_wait()`. In this case, parent sleeps forever!

Trap 1 When Using CV



Trap 1 When Using CV

Trap 1 When Using CV

Trap 1 When Using CV

Only one thread gets a signal

Trap 2 When Using CV

Condition Variable

Trap 2 When Using CV

Trap 2 When Using CV

Condition Variable

Trap 2 When Using CV

Trap 2 When Using CV

Trap 2 When Using CV

Signal lost if nobody waiting at that time

Guarantee

Upon signal, there has to be **at least one** thread waiting;
If there are threads waiting, **at least one** thread will wake

CV-based Parent-wait-for-child Approach

```
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;
}
```

```c
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;
}
```

CV-based Parent-wait-for-child Approach

Good Rule of Thumb

Always do **1. wait** and **2. signal** while holding the lock

To prevent lost signal

Worksheet