Locks, Semaphores, and Producer-Consumer Problem

CS 571: Operating Systems (Spring 2020)

Lecture 3

Yue Cheng

Review: Threads

Threads

- Processes vs. threads
  - Parent and child processes do not share address space
  - Inter-process communication w/ message passing or shared memory

  - Threads created by one process share address space, open files, global variables, etc.
  - Much cheaper and more flexible inter-thread communication and cooperation

A Simple Example Using `pthread`

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

Thread Trace 1

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		

join

Time

Thread Trace 1

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	

Thread Trace 1

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		

join

Thread Trace 1

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
		runs
		prints "B"
		returns

Thread Trace 1

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
		runs
		prints "B"
		returns
prints "main: end"		

Thread Trace 2

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		

Thread Trace 2

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
interrupt	runs	
	prints "A"	
	returns	

Thread Trace 2

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		

Thread Trace 2

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns

Thread Trace 2

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Thread Trace 2

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

What would a 3rd thread trace look like?

Synchronization

- Race Conditions
- The Critical Section Problem
- Synchronization Hardware and Locks
- Semaphores

Threaded Counting Example

```c
#include <stdio.h>
#include "common.h"

static volatile int counter = 0;

//
// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char*) arg);
    return NULL;
}

//
// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

```
$ git clone https://github.com/tddg/demo-ostep-code
$ cd demo-ostep-code/threads-intro
$ make
$ ./t1 <loop_count>
```

Try it yourself

Back-to-Back Runs

Run 1…
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 10706438)

Run 2…
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 11852529)

What exactly Happened??

What exactly Happened??

```
% otool -t -v thread_rc          [Mac OS X]
% objdump -d thread_rc           [Linux]
```

Load   mem addr.   CPU register.

```
...
0000000100000d52    movl  0x2f8e %eax
0000000100000d58    addl  $0x1,  %eax
0000000100000d5b    movl  %eax,  0x2f8e
...
```

Store

```
counter = counter + 1;
```

Concurrent Access to the Same Memory Address

OS	Thread 1	Thread 2	Value
	Enter into critical section		
	movl 0x2f8e, %eax		50
	addl $0x1, %eax		51

Time

Concurrent Access to the Same Memory Address

OS	Thread 1	Thread 2	Value

	Enter into critical section
	movl 0x2f8e, %eax
	addl $0x1, %eax
			50
			51

timer **Interrupt** Save T1's state Restore T2's state			

Time

Concurrent Access to the Same Memory Address

	OS	Thread 1	Thread 2	Value
Time		**Enter into critical section**		
		movl 0x2f8e, %eax		50
		addl $0x1, %eax		51
	Interrupt			
	Save T1's state			
	Restore T2's state			
			movl 0x2f8e, %eax	
			addl $0x1, %eax	
			movl %eax, 0x2f8e	

Concurrent Access to the Same Memory Address

Time	OS	Thread 1	Thread 2	Value
		Enter into critical section		
		movl 0x2f8e, %eax		50
		addl $0x1, %eax		51
	Interrupt			
	Save T1's state			
	Restore T2's state			
			movl 0x2f8e, %eax	50
			addl $0x1, %eax	51
			movl %eax, 0x2f8e	51

Concurrent Access to the Same Memory Address

OS	Thread 1	Thread 2	Value
	Enter into critical section		
	movl 0x2f8e, %eax		50
	addl $0x1, %eax		51
Interrupt Save T1's state Restore T2's state			
		movl 0x2f8e, %eax	50
		addl $0x1, %eax	51
		movl %eax, 0x2f8e	51
Interrupt Save T2's state Restore T1's state			

Time

Concurrent Access to the Same Memory Address

Time	OS	Thread 1	Thread 2	Value
		Enter into critical section		
		movl 0x2f8e, %eax		50
		addl $0x1, %eax		51
	Interrupt			
	Save T1's state			
	Restore T2's state			
			movl 0x2f8e, %eax	50
			addl $0x1, %eax	51
			movl %eax, 0x2f8e	51
	Interrupt			
	Save T2's state			
	Restore T1's state			
		movl %eax, 0x2f8e		

Concurrent Access to the Same Memory Address

Time

OS	Thread 1	Thread 2	Value
	Enter into critical section		
	movl 0x2f8e, %eax		50
	addl $0x1, %eax		51
Interrupt Save T1's state Restore T2's state			
		movl 0x2f8e, %eax	50
		addl $0x1, %eax	51
		movl %eax, 0x2f8e	51
Interrupt Save T2's state Restore T1's state			
	movl %eax, 0x2f8e		52 **51**

Concurrent Access to the Same Memory Address

Time

OS	Thread 1	Thread 2	Value
	Enter into critical section		
	movl 0x2f8e, %eax		50
	addl $0x1, %eax		51
Interrupt Save T1’s state Restore T2’s state			
		movl 0x2f8e, %eax	50
		addl $0x1, %eax	51
		movl %eax, 0x2f8e	51
Interrupt Save T2’s state Restore T1’s state			
	movl %eax, 0x2f8e		**51**

Takeaway

- Observe: In a time-shared system, the exact instruction execution order cannot be predicted
  - Deterministic vs. Non-deterministic

- Any possible orders could happen, which result in different output across runs

Race Conditions

- Situations like this, where multiple processes are writing or reading some shared data and the final result depends on who runs precisely when, are called race conditions
  - A serious problem for any concurrent system using shared variables

- Programmers must make sure that some high-level code sections are executed atomically
  - Atomic operation: It completes in its entirety without worrying about interruption by any other potentially conflict-causing process

The Critical-Section Problem

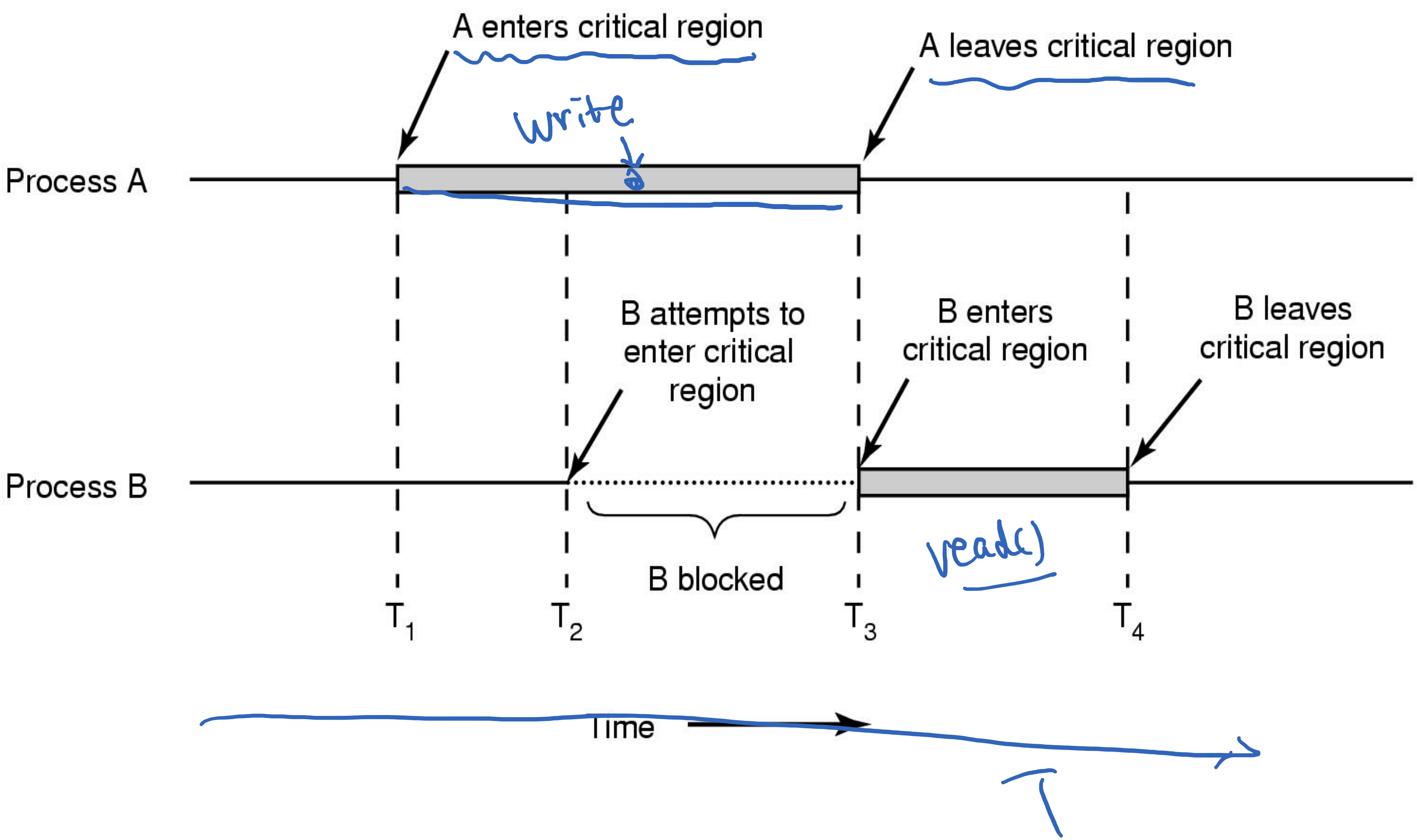
- *N* processes/threads all competing to access the shared data

- Each process/thread has a code segment, called critical section (critical region), in which the shared data is accessed

- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in that critical section

- The execution of the critical sections by the processes must be mutually exclusive in time

Mutual Exclusion



Solving Critical-Section Problem

Any solution to the problem must satisfy **four conditions**!

Mutual Exclusion:

No two processes may be simultaneously inside the same critical section

Bounded Waiting: Fairness.

No process should have to wait forever to enter a critical section

Progress:

No process executing a code segment unrelated to a given critical section can block another process trying to enter the same critical section

Arbitrary Speed:

No assumption can be made about the relative speed of different processes (though all processes have a non-zero speed)

Using Lock to Protect Shared Data

- Suppose that two threads A and B have access to a shared variable "`balance`"

Thread A:

```
balance = balance + 1
```

Thread B:

```
balance = balance + 1
```

```
lock_t mutex; // some globally-allocated lock 'mutex'
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Locks

- A lock is a variable

- Two states
  - Available or free
  - Locked or held

- `lock()`: tries to acquire the lock
- `unlock()`: releases the lock that has been acquired by caller

Building a Lock

- Needs help from hardware + OS
- A number of hardware primitives to support a lock
- Goals of a lock
  - Basic task: Mutual exclusion
  - Fairness
  - Performance

First Attempt: A Simple Flag

- How about just using `loads/stores` instructions?

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)   // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;           // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

First Attempt: A Simple Flag

- How about just using `loads/stores` instructions?

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

→ A spin lock

First Attempt: A Simple Flag

- How about just using `loads/stores` instructions?

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

A spin lock

What's the problem?

First Attempt: A Simple Flag

Flag is 0 initially

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	

timer

First Attempt: A Simple Flag

Flag is 0 initially

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	Checking that Flag is 0, again…
	call `lock()`
	while (flag == 1)

First Attempt: A Simple Flag

Flag is set to 1 by T2

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1

First Attempt: A Simple Flag

Flag is set to 1 again! Two threads both in Critical Section

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

First Attempt: A Simple Flag

Flag is set to 1 again! Two threads both in Critical Section

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Reason:
Lock operation is not atomic!
And therefore, no mutual exclusion!

Getting Help from the Hardware

- One solution supported by hardware may be to use interrupt capability

```
do {
    lock()
      critical section;
    unlock()
      remainder section;
} while (1);
```

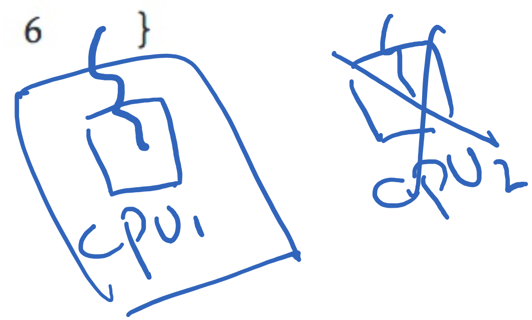
```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Getting Help from the Hardware

- One solution supported by hardware may be to use interrupt capability

```
do {
    lock()
      critical section;
    unlock()
      remainder section;
} while (1);
```

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```



Are we done??

Synchronization Hardware

- Many machines provide special hardware instructions to help achieve mutual exclusion

- The `TestAndSet (TAS)` instruction tests and modifies the content of a memory word atomically

- TAS returns old value pointed to by `old_ptr` and updates said value to `new`

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value
}
```

Operations performed atomically!

Mutual Exclusion with TAS

- Initially, `lock`'s flag set to 0

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0 indicates that lock is available, 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

A correct spin lock

Busy Waiting and Spin Locks

- This approach is based on busy waiting
  - If the critical section is being used, waiting processes loop continuously at the entry point
- A binary "`lock`" variable that uses busy waiting is called a spin lock
  - Processes that find the lock unavailable "spin" at the entry
- It actually works (mutual exclusion)

Busy Waiting and Spin Locks

- This approach is based on busy waiting
  - If the critical section is being used, waiting processes loop continuously at the entry point
- A binary "`lock`" variable that uses busy waiting is called a spin lock
  - Processes that find the lock unavailable "spin" at the entry
- It actually works (mutual exclusion)
- Disadvantages?
  - Fairness?
  - Performance?

Busy Waiting and Spin Locks

- This approach is based on busy waiting
  - If the critical section is being used, waiting processes loop continuously at the entry point
- A binary "`lock`" variable that uses busy waiting is called a spin lock
  - Processes that find the lock unavailable "spin" at the entry
- It actually works (mutual exclusion)
- Disadvantages?
  - Fairness? (A: No. Heavy contention may cause starvation)
  - Performance? (A: Busy waiting wastes CPU cycles)

A Simple Approach: Just Yield ~~(Win)~~!

- When you are going to spin, just give up the CPU to another process/thread

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Lock Worksheet

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)`)

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 */
```

thread

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		

T

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	S/w Interrupt	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

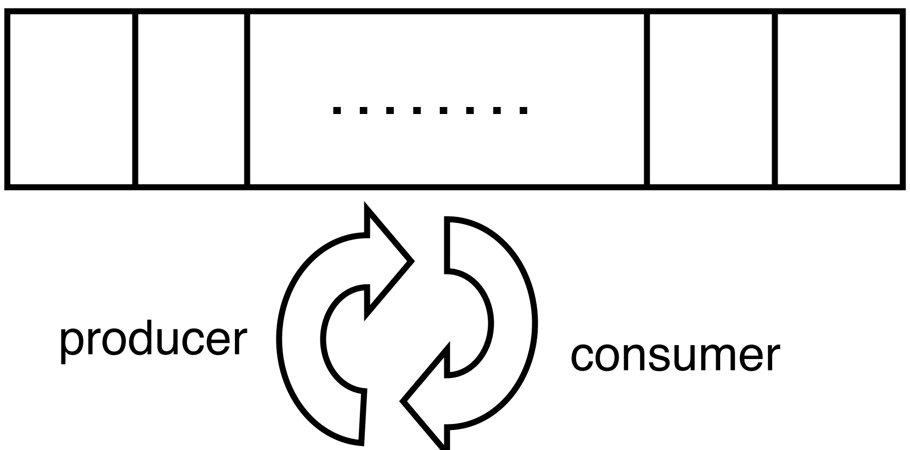
Today

- Readers-Writers Problem

- Dining-Philosophers Problem

Producer-Consumer Problem

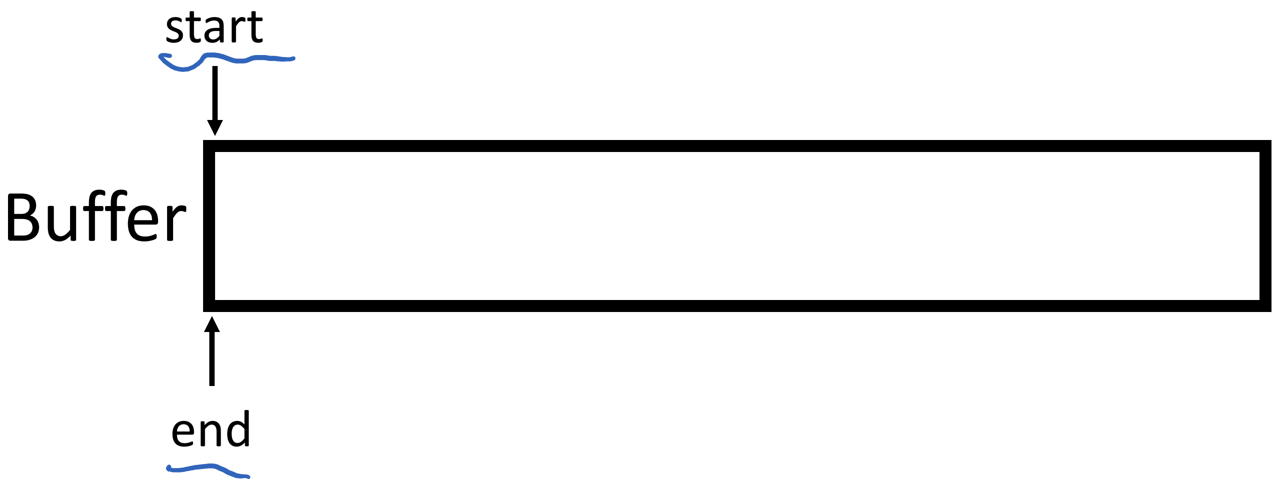
- The bounded-buffer producer-consumer problem assumes that there is a buffer of size N
- The producer process puts items to the buffer area
- The consumer process consumes items from the buffer
- The producer and the consumer execute concurrently



Example: Unix Pipes

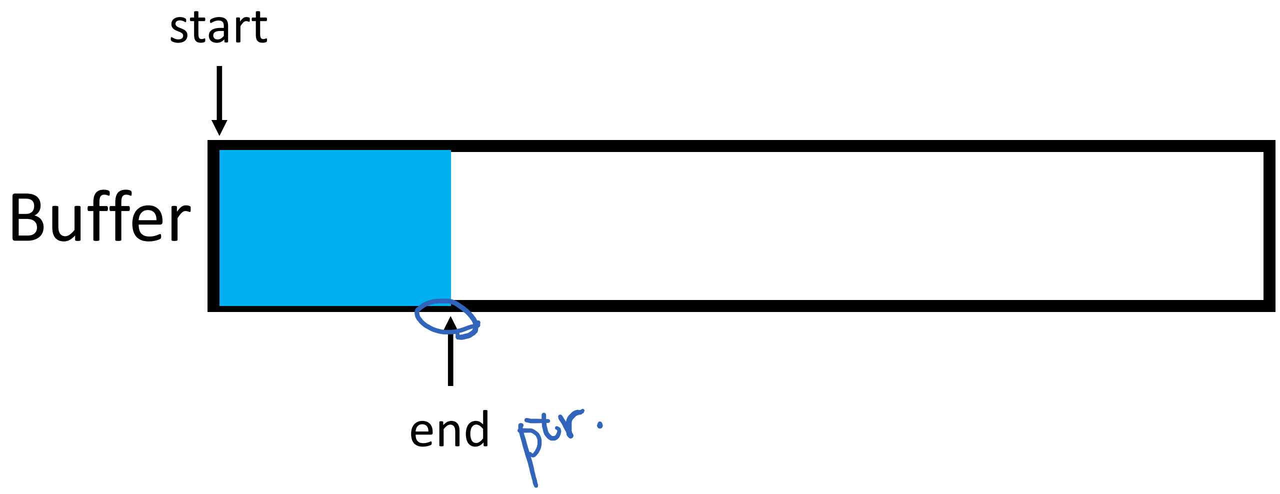
- A pipe may have many writers and readers
- Internally, there is a finite-sized buffer
- Writers add data to the buffer
- Readers remove data from the buffer

Example: Unix Pipes

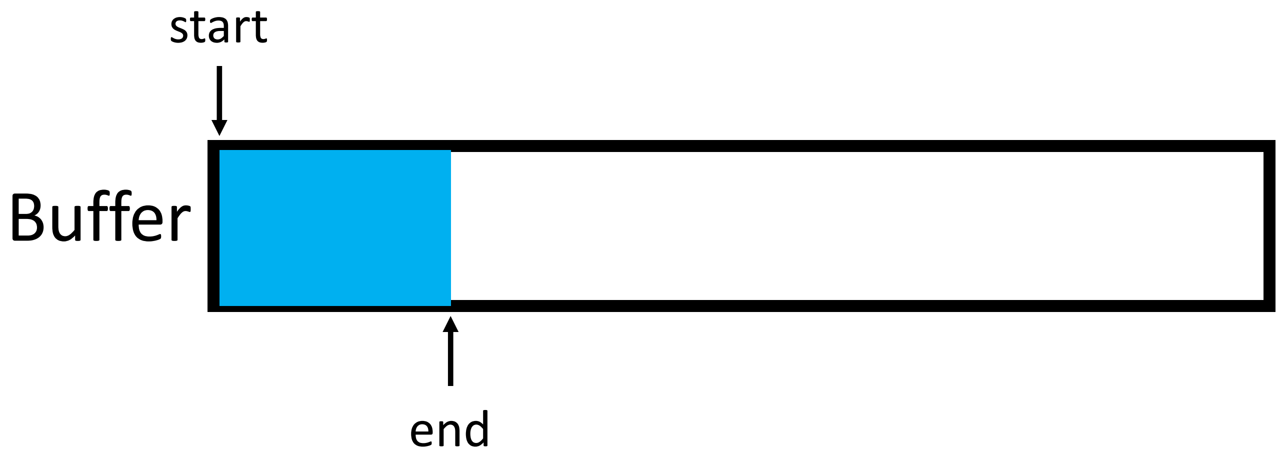


Example: Unix Pipes

Write

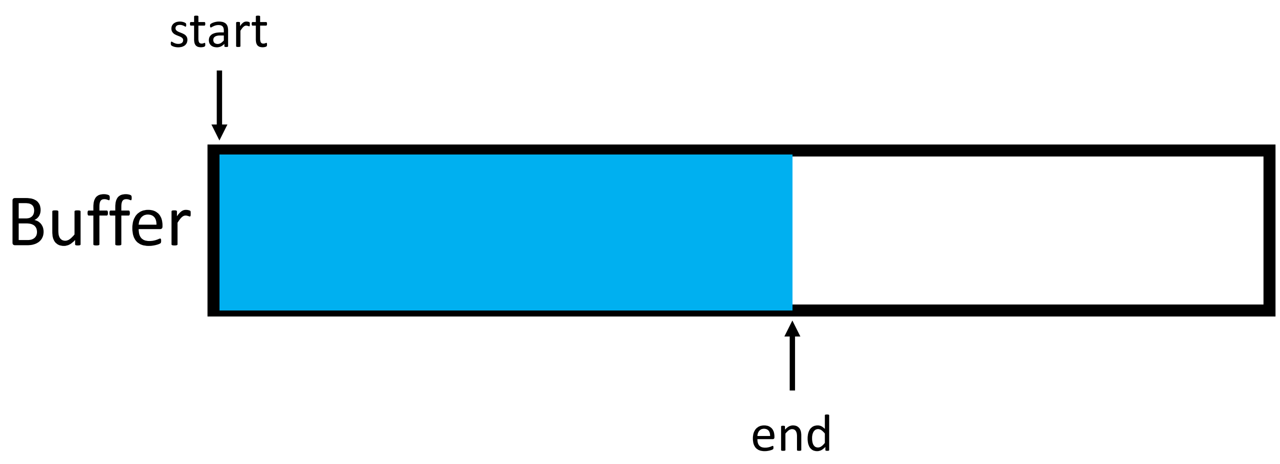


Example: Unix Pipes

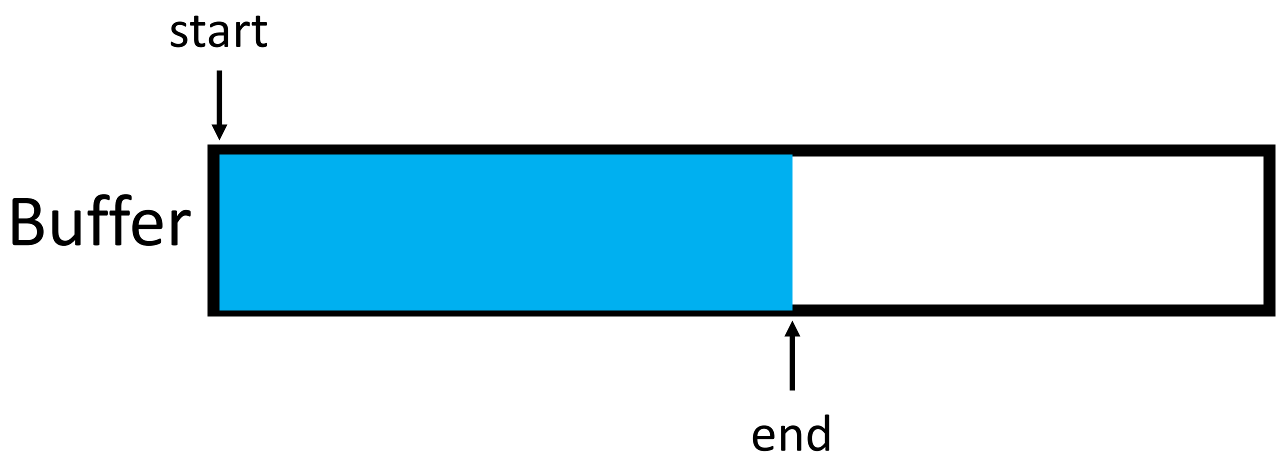


Example: Unix Pipes

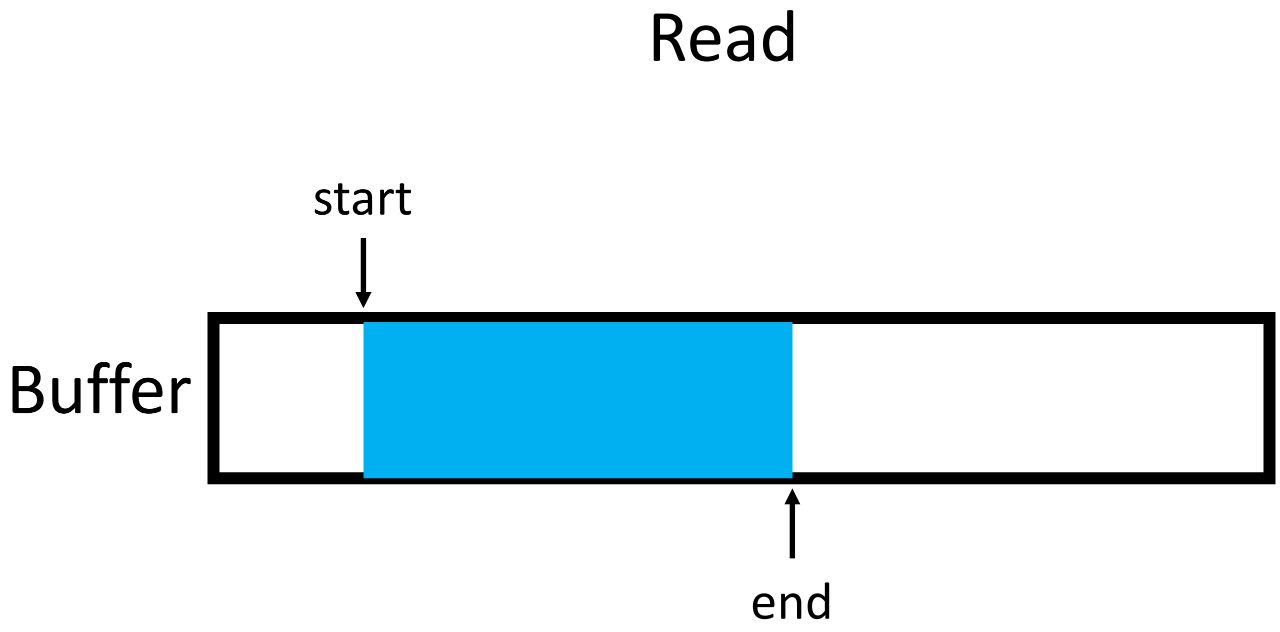
Write



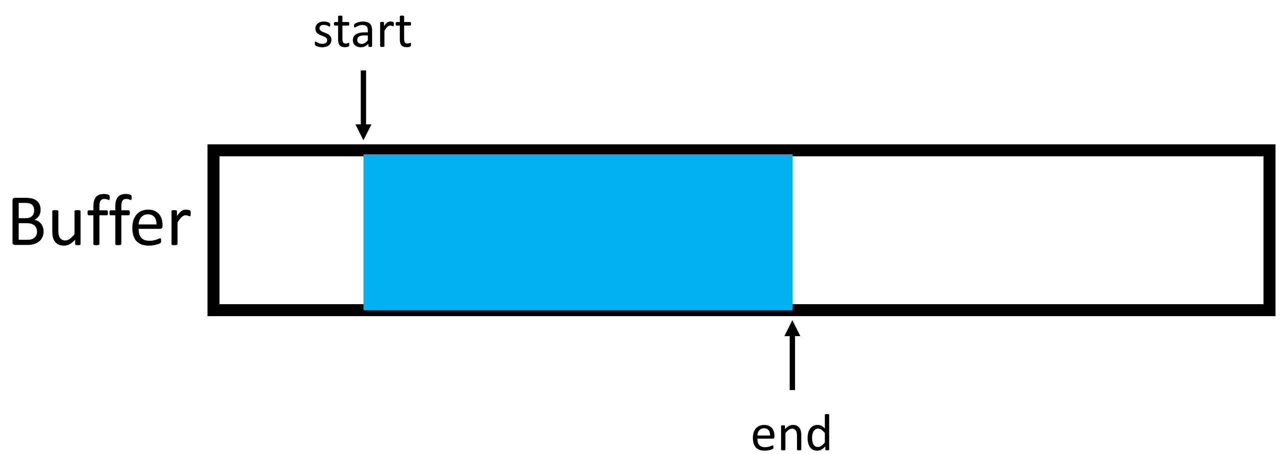
Example: Unix Pipes



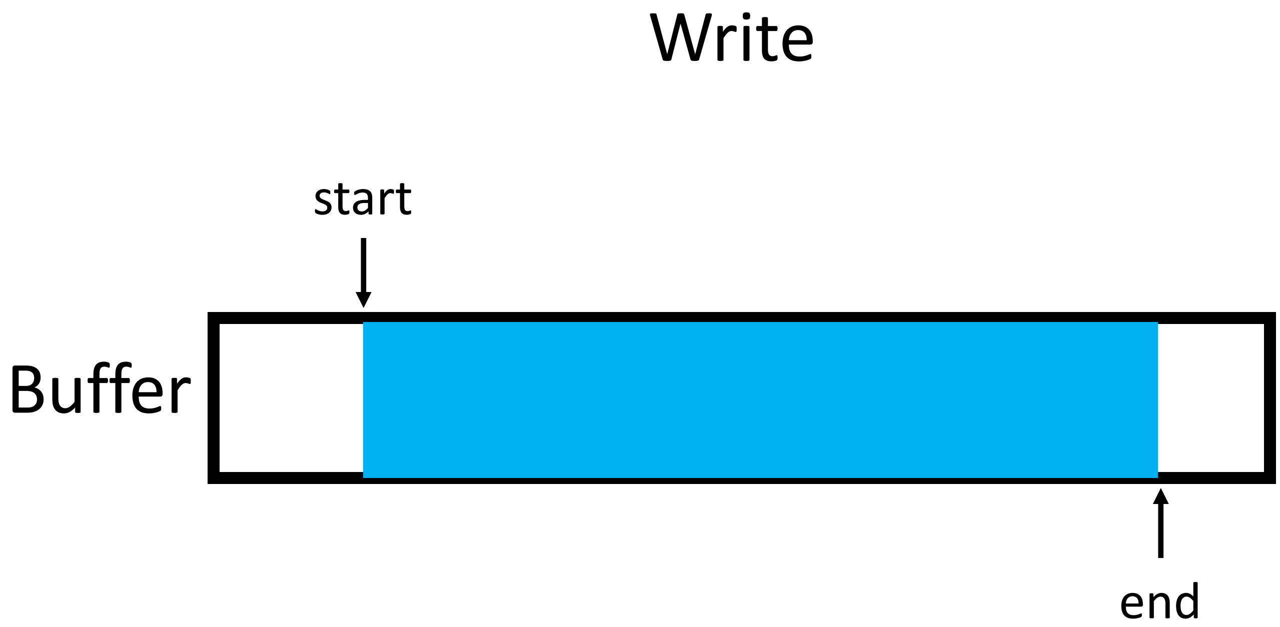
Example: Unix Pipes



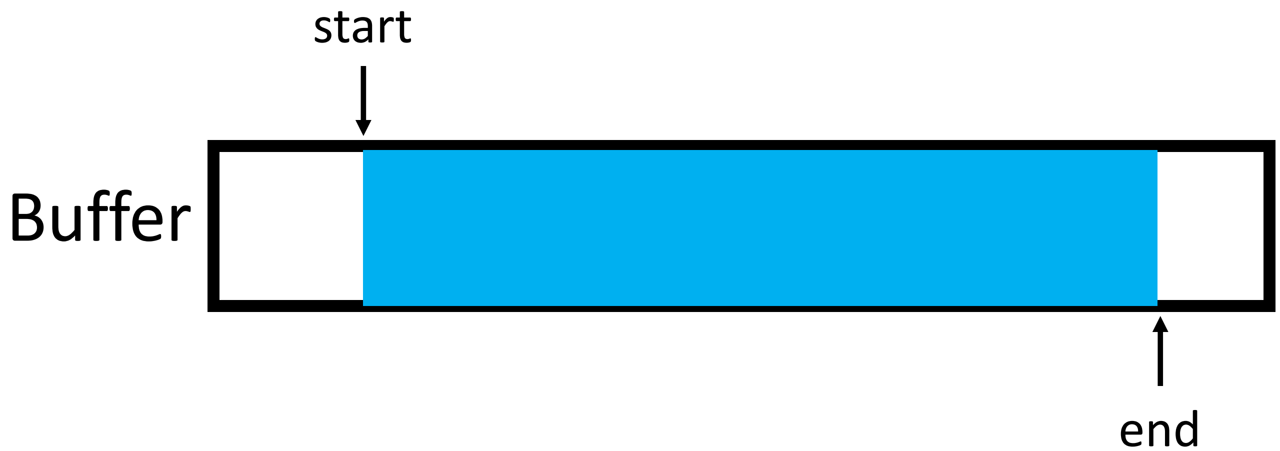
Example: Unix Pipes



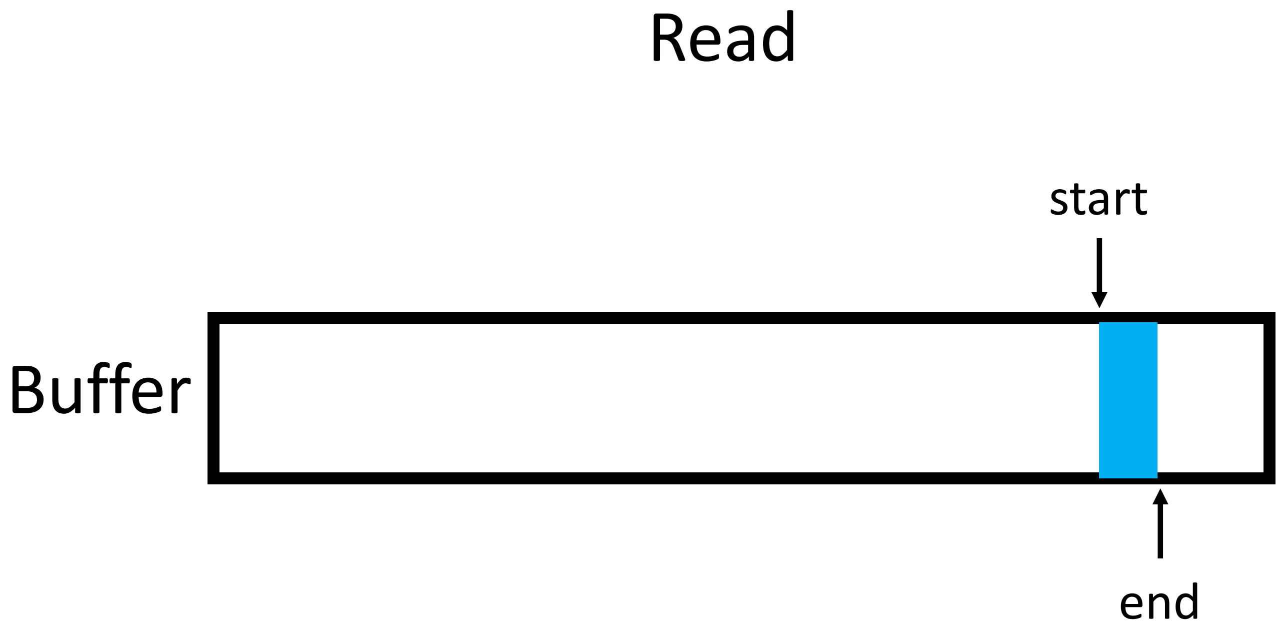
Example: Unix Pipes



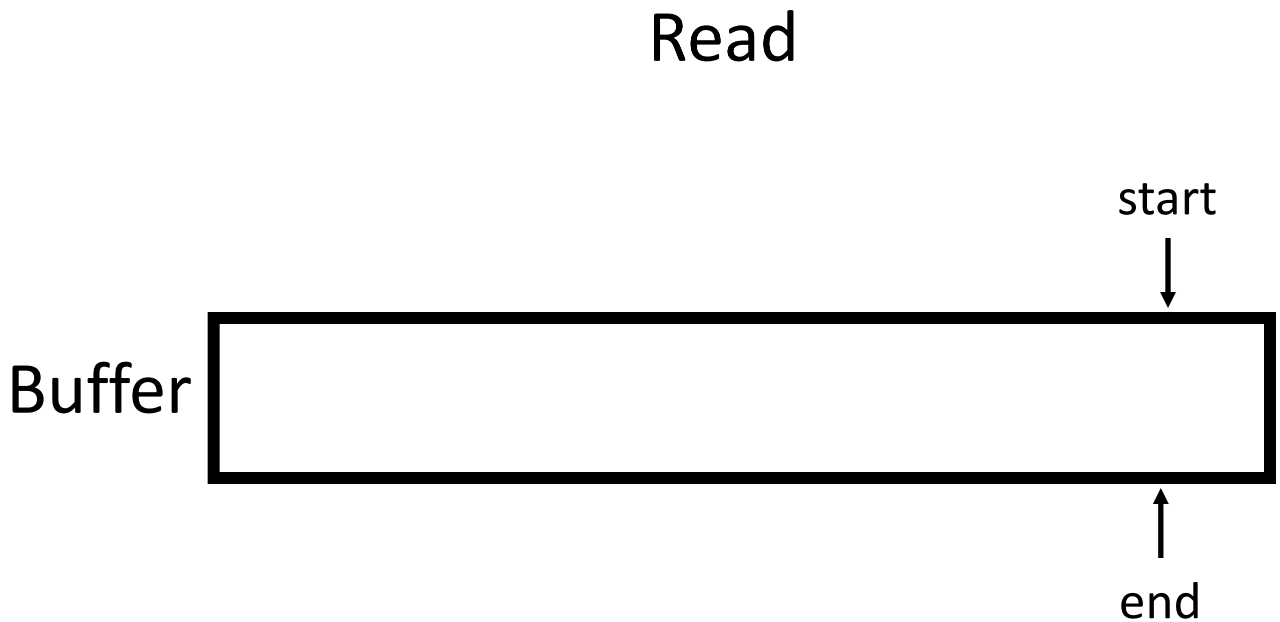
Example: Unix Pipes



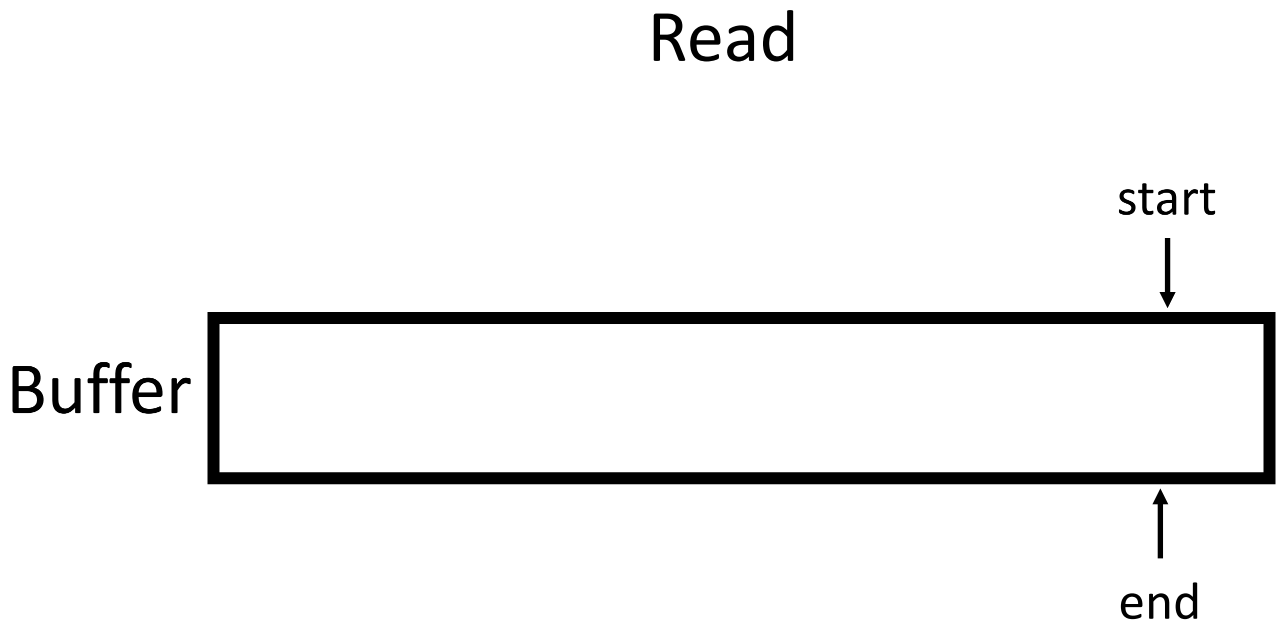
Example: Unix Pipes



Example: Unix Pipes



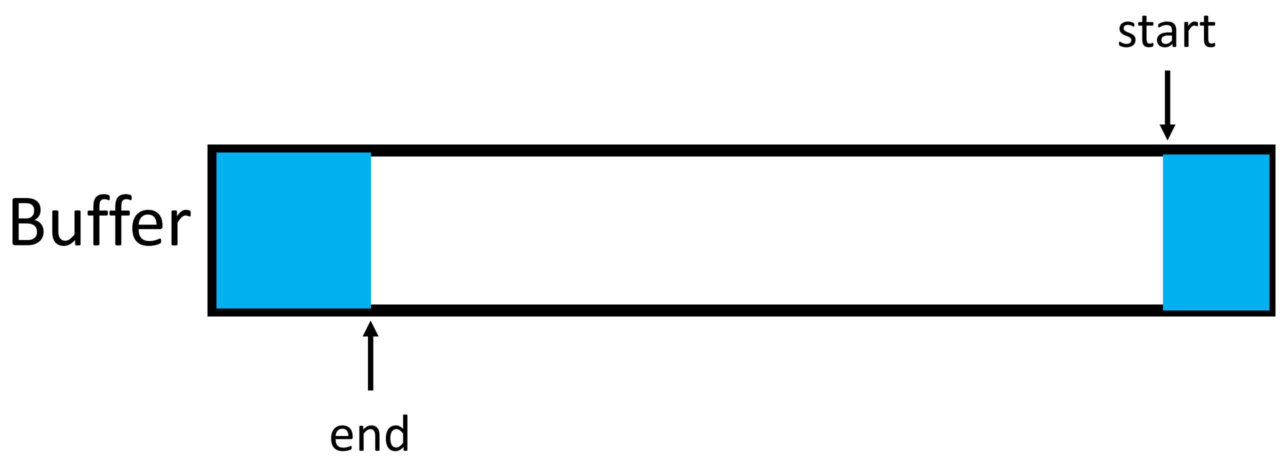
Example: Unix Pipes



Note: reader must **wait**

Example: Unix Pipes

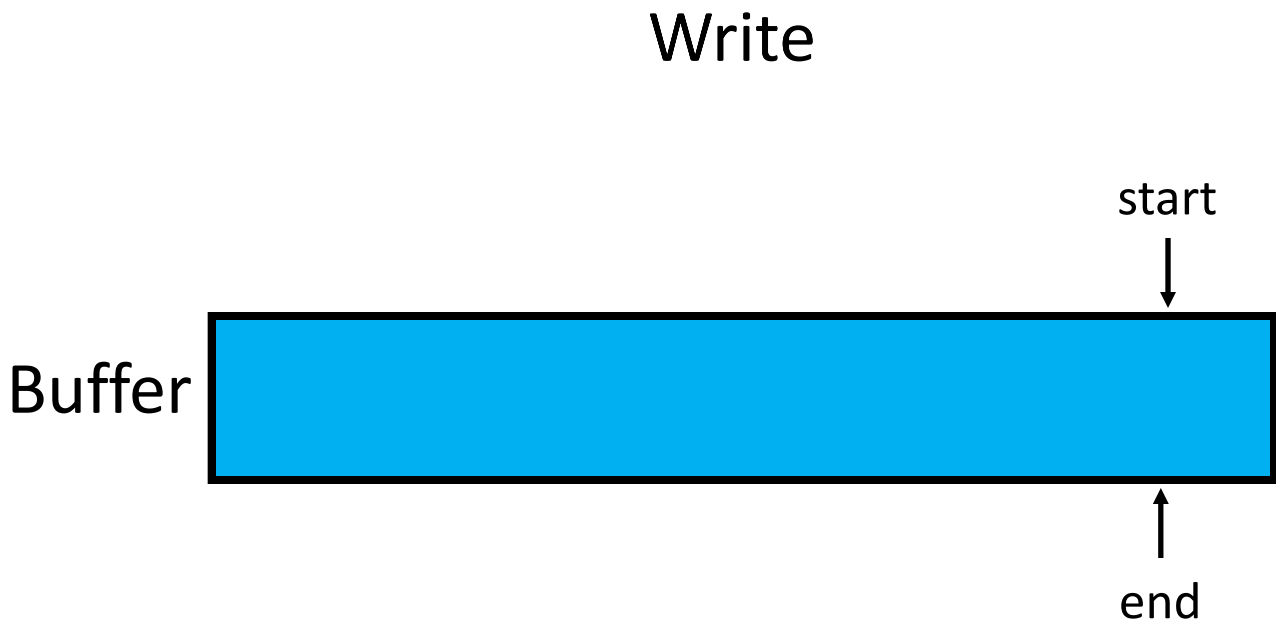
Write



Example: Unix Pipes

Write

Example: Unix Pipes



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

Linux Pipe Commands

producer consumer

```
% 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

```
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?

```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?

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);
    }
}
```

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

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;
    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

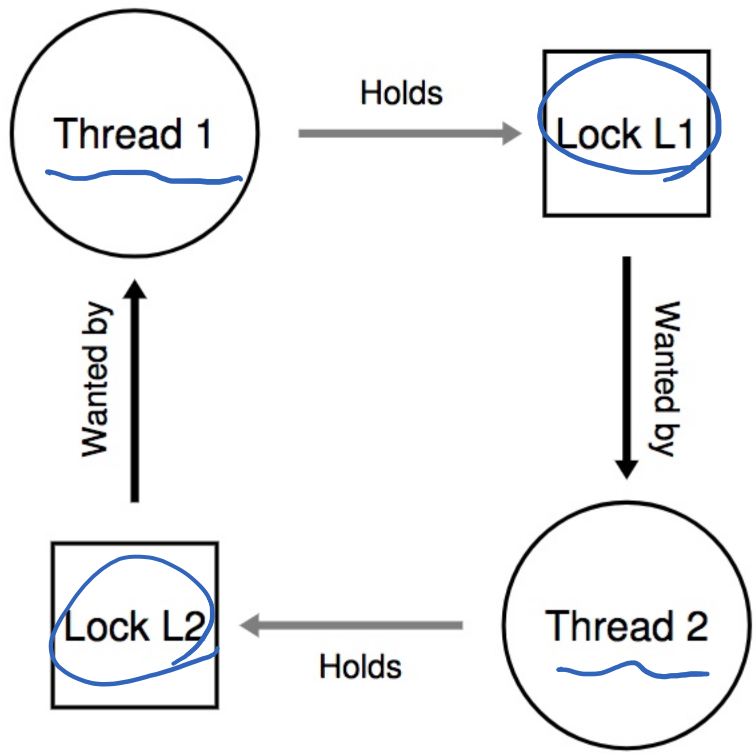
```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&full);
        int tmp = get();
        sem_post(&empty);
        sem_post(&mutex);
        printf("%d\n", tmp);
    }
}
```

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

```c
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

lock

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

Semaphore Worksheet