Concurrency: Threads, Locks, and Semaphores

CS 571: Operating Systems (Spring 2022)

Lecture 6

Yue Cheng

Announcements

- Project checkpoint #1
  - Due in two weeks (03/25)

- This Friday's open studio rescheduled to:
  - 10am -- 10:45am, 03/11

Concurrency + *Sync.*

- Threads

- Race Conditions

- The Critical Section Problem

- Locks

- Semaphores

Threads

# Why Thread Abstraction?
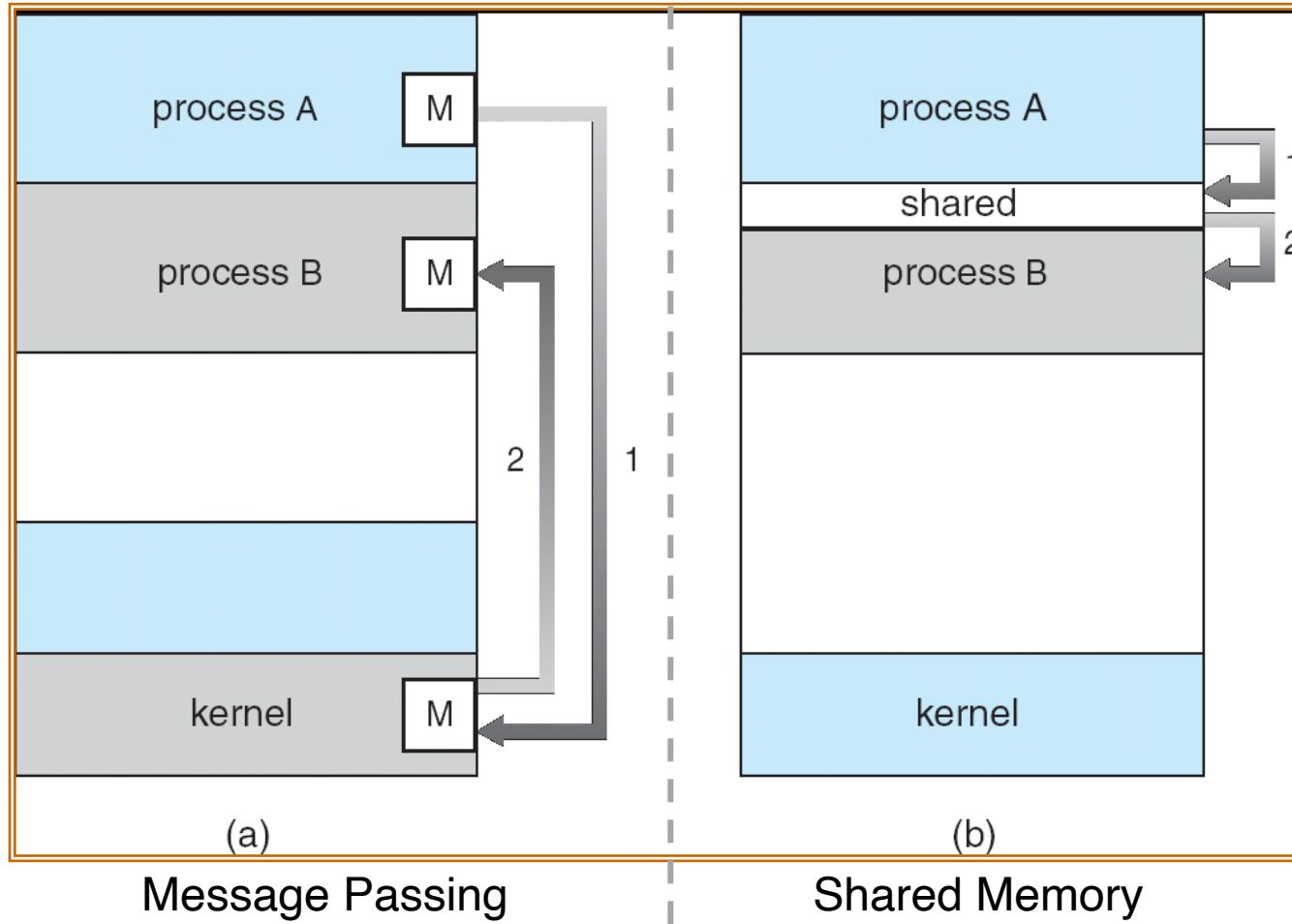
Process Abstraction: Challenge 1

- Inter-process communication (IPC)

Inter-Process Communication

- Mechanism for processes to communicate and to synchronize their actions

- Two models
  - Communication through a shared memory region
  - Communication through message passing

Communication Models

(a) Message Passing   (b) Shared Memory

Communication through Message Passing

- Message system – processes communicate with each other without resorting to shared variables

- A message-passing facility must provide at least two operations:
  - `send(message, recipient)`
  - `receive(message, recipient)`   *System V*

- With **indirect** communication, the messages are sent to and received from mailboxes (or, ports)
  - `send(A, message) /* A is a mailbox */`
  - `receive(A, message)`

Communication through Message Passing

- Message passing can be either **blocking** (synchronous) or **non-blocking** (asynchronous)
  - **Blocking Send:** The sending process is blocked until the message is received by the receiving process or by the mailbox
  - **Non-blocking Send:** The sending process resumes the operation as soon as the message is received by the kernel
  - **Blocking Receive:** The receiver blocks until the message is available
  - **Non-blocking Receive:** "Receive" operation does not block; it either returns a valid message or a default value (null) to indicate a non-existing message

Communication through Shared Memory
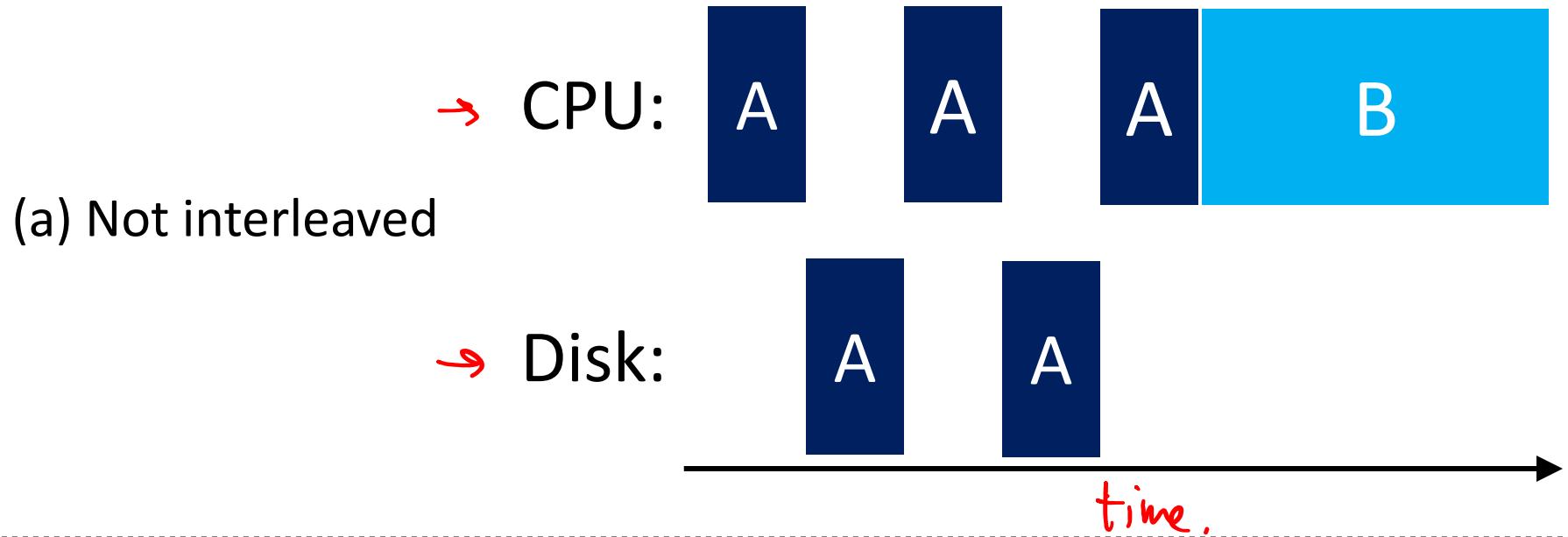
- The memory region to be shared must be explicitly defined
- System calls (Linux):
  - `shmget` or `shm_open` creates a shared memory block
  - `shmat` maps/attaches an existing shared memory block into a process's address space
  - `shmdt` or `shm_unlink` removes ("`unmaps`") a shared memory block from the process's address space
  - `shmctl` is a general-purpose function allowing various operations on the shared block (receive information about the block, set the permissions, lock in memory, …)
- Problems with simultaneous access to the shared variables
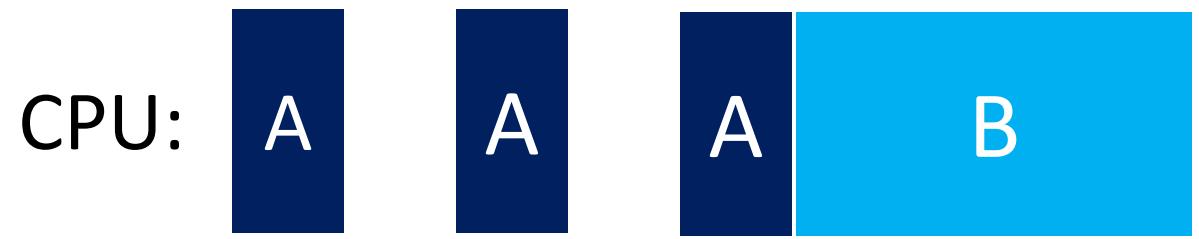
Process Abstraction: Challenge 1

- Inter-process communication (IPC)
  - Cumbersome programming!
  - Copying overheads (inefficient communication)
 - Expensive context switching (why expensive?)
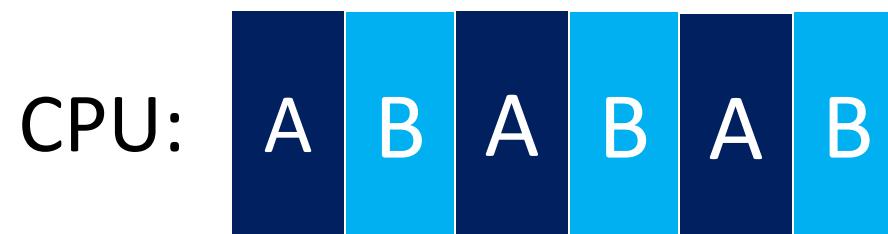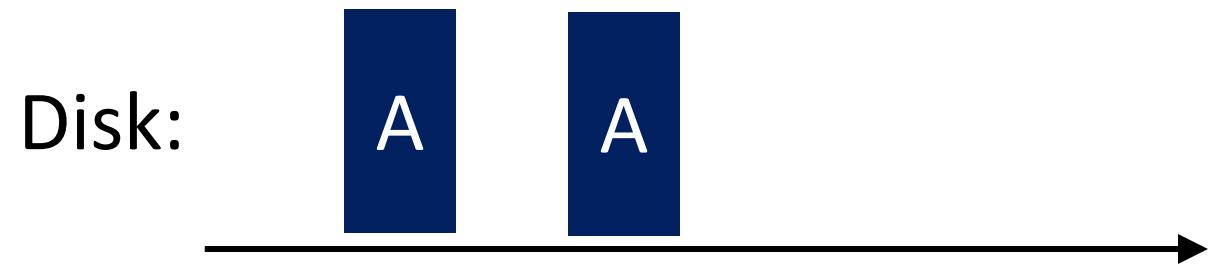
Process Abstraction: Challenge 2

- Inter-process communication (IPC)
  - Cumbersome programming!
  - Copying overheads (inefficient communication)
  - Expensive context switching (why expensive?)

- CPU utilization

CPU: A A A B

(a) Not interleaved

Disk: A A

time.

(a) Not interleaved

CPU: A A A B

Disk: A A

(b) Interleaved

CPU: A B A B A B

Disk: A A

CPU: A A A B

(a) Not interleaved

Disk: A A
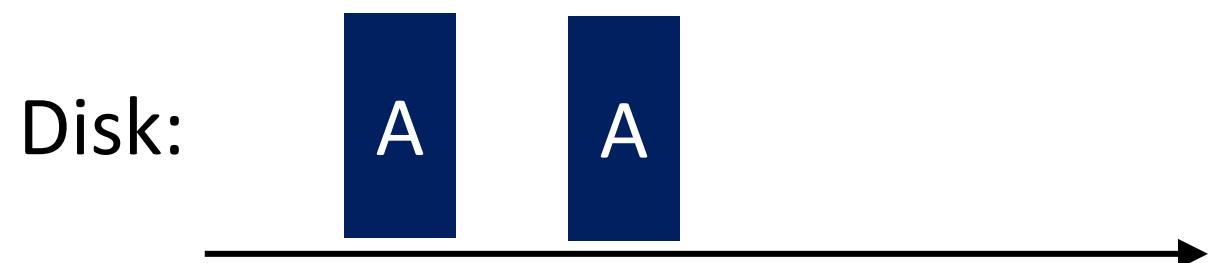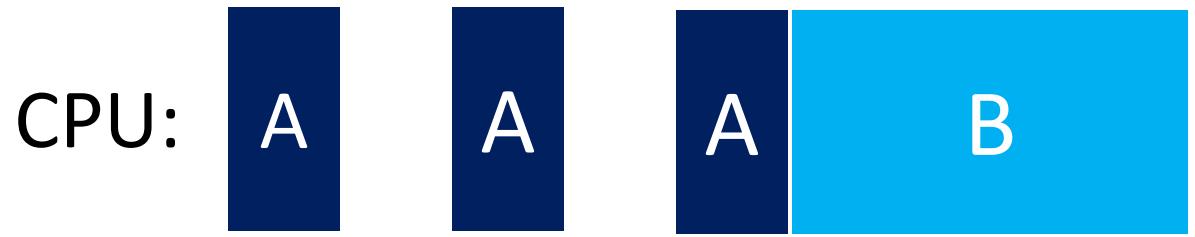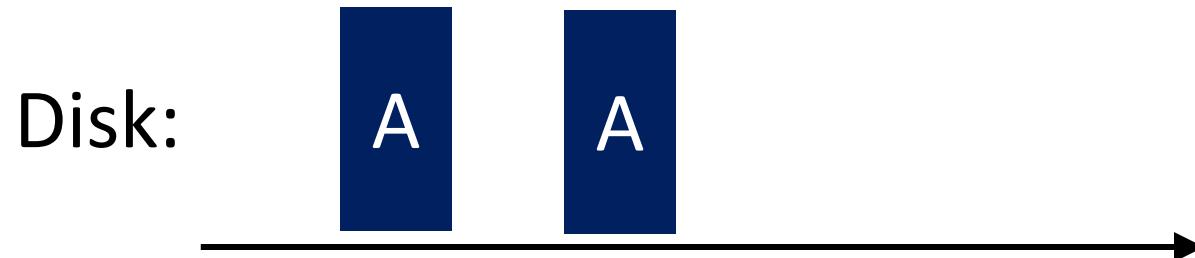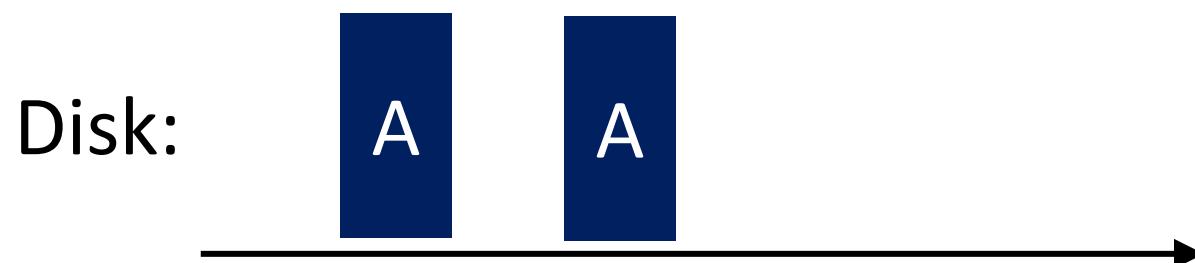
What if there is only one process?
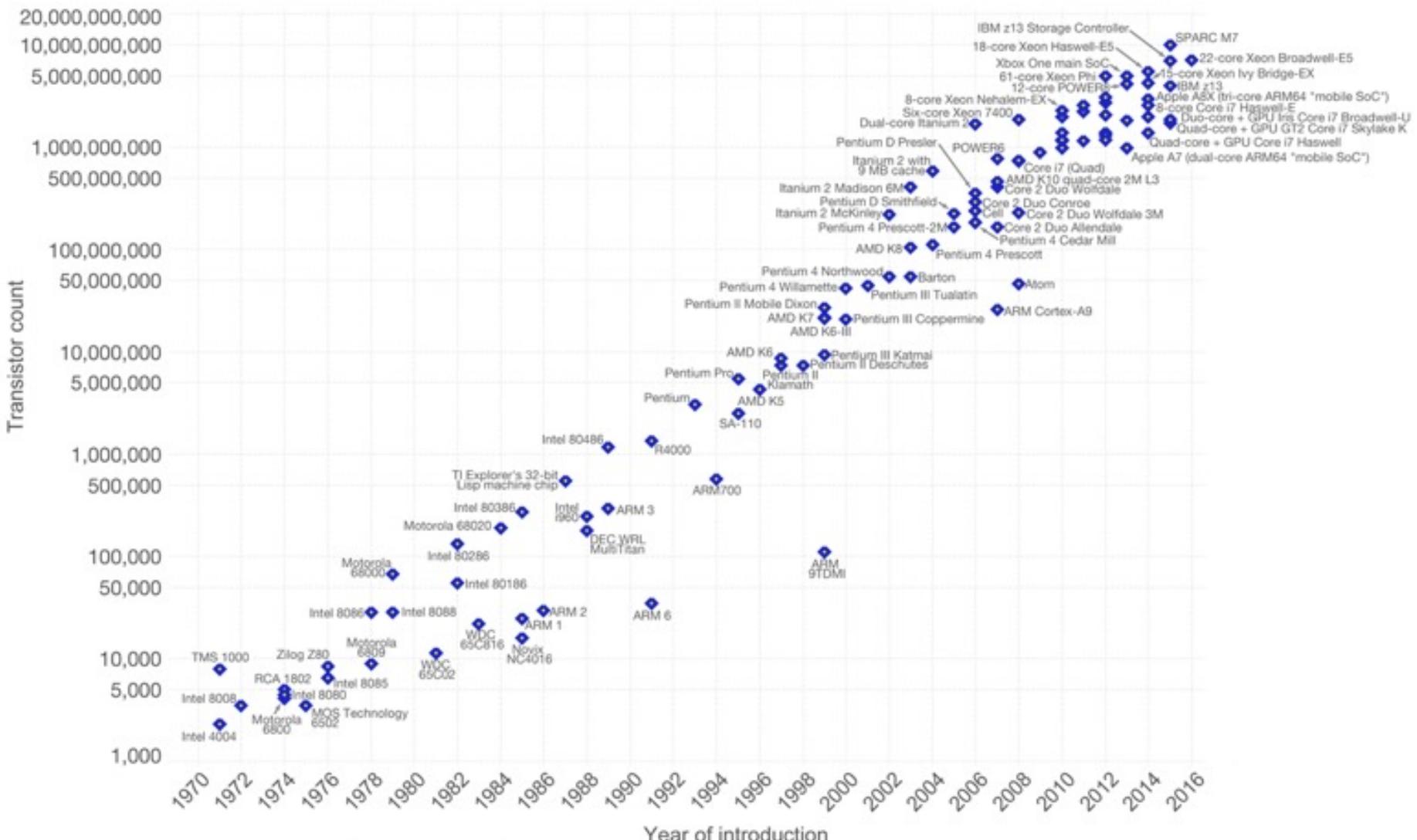
CPU: A B A B A B

(b) Interleaved

Disk: A A

Moore's law: # transistors doubles every ~2 years

Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.

Moore's law: # transistors doubles every ~2 years

Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.

Moore's law is ending!

Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Licensed under CC-BY-SA by the author Max Roser.

CPU Trends – What Moore's Law Implies...

- The future
    - Same CPU speed
    - More cores (to scale-up) *more servers.* → *scale-out*

- Faster programs => concurrent execution

- **Goal**: Write applications that fully utilize many CPU cores…

Goal

- Write applications that fully utilize many CPUs…

Strategy 1

- Build applications from many communication processes
  - Like Chrome (process per tab)
  - Communicate via `pipe()` or similar

- Pros/cons?

Strategy 1

- Build applications from many communication processes
  - Like Chrome (process per tab)
  - Communicate via `pipe()` or similar

- Pros/cons? – That we've talked about in previous slides
  - Pros:
    - Don't need new abstractions!
    - Better (fault) isolation?
  - Cons:
    - Cumbersome programming using IPC
    - Copying overheads
    - Expensive context switching

Strategy 2

- New abstraction: the thread

Introducing Thread Abstraction

- New abstraction: the thread

- Threads are just like processes, but threads share the address space

Thread

- A process, as defined so far, has only one thread of execution

- Idea: Allow multiple threads of concurrently running execution within the same process environment, to a large degree independent of each other
  - Each thread may be executing different code at the same time

Process vs. Thread

- Multiple threads within a process will share
  - The address space
  - Open files (file descriptors)
  - Other resources

- Thread
  - Efficient and fast resource sharing
  - Efficient utilization of many CPU cores with only one process
  - Less context switching overheads

CPU 1

Running
thread 1
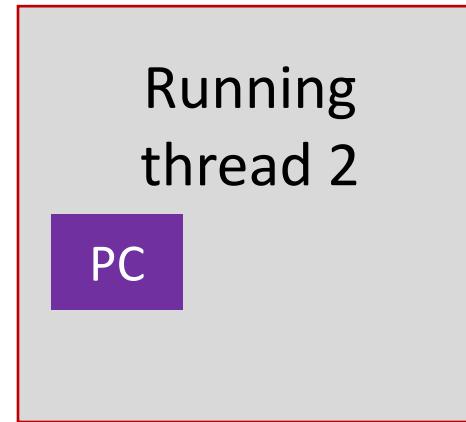
CPU 2

Running
thread 2

CPU 1

Running
thread 1

PC

CPU 2

Running
thread 2

PC

CPU 1

CPU 2

Running thread 1

PC

Running thread 2

PC

Each thread may be executing different code at the same time

CODE   HEAP

Virtual mem

CPU 1

Running thread 1

PC

CPU 2

Running thread 2

PC

CODE **HEAP**

Virtual mem

CPU 1

Running thread 1

PC   SP

CPU 2

Running thread 2

PC   SP

CODE   HEAP

Virtual mem

CPU 1

CPU 2

Running thread 1

Running thread 2

PC

SP

PC

SP
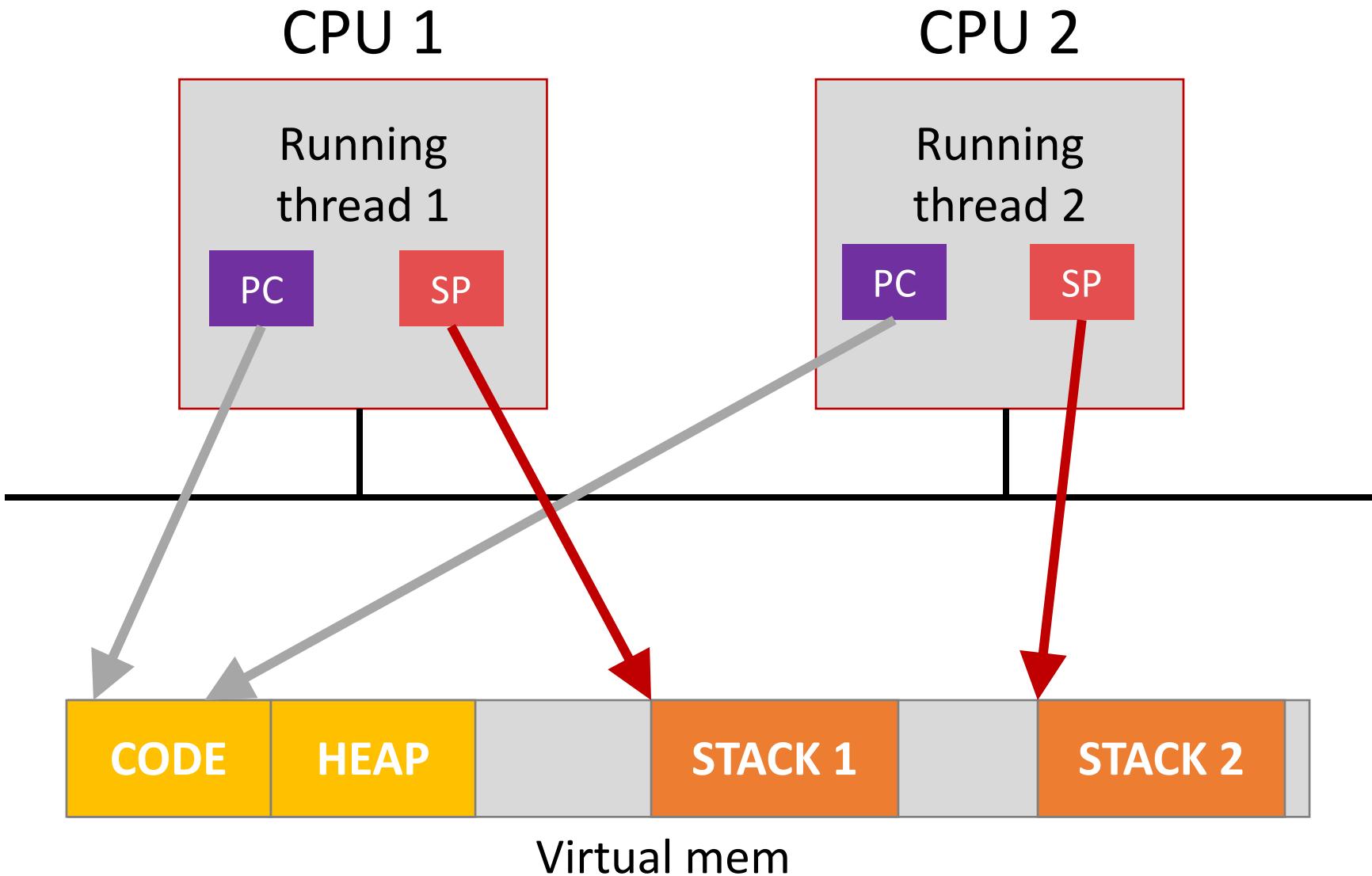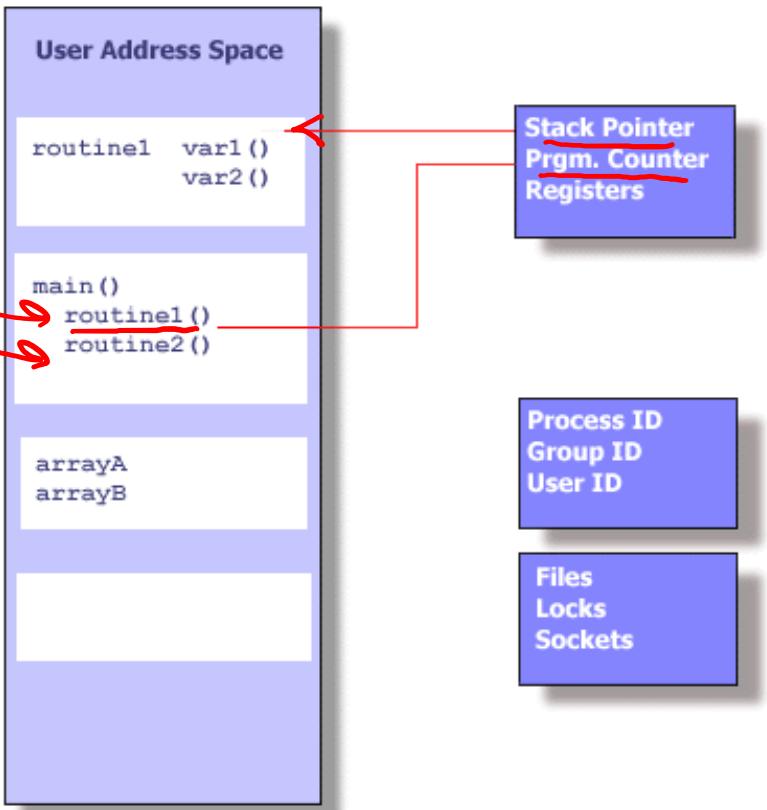
CODE   HEAP   STACK 1   STACK 2
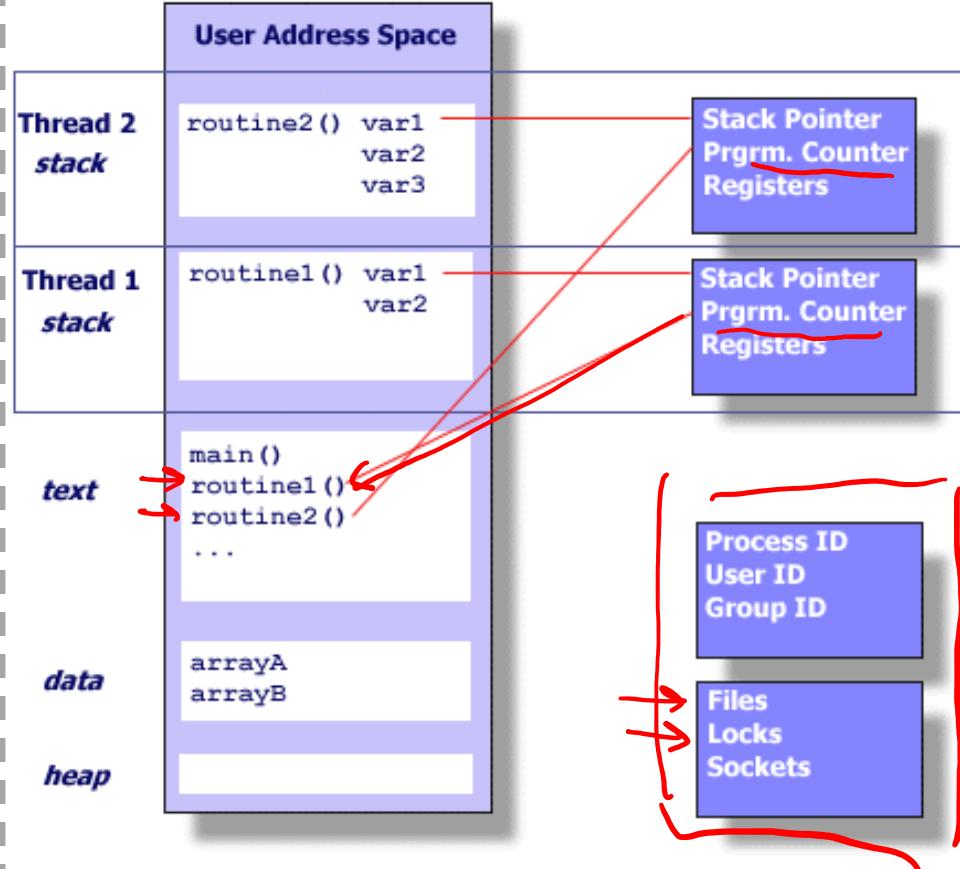
Virtual mem

Different threads need different stacks

Virtual mem

Linux process

Threads within a Linux process

*: https://computing.llnl.gov/tutorials/pthreads/

Using Threads

- Processes usually start with a single thread
- Usually, library procedures are invoked to manage threads
  - thread_create: typically specifies the name of the procedure for the new thread to run
  - thread_exit
  - thread_join: blocks the calling thread until another (specific) thread has exited
  - thread_yield: voluntarily gives up the CPU to let another thread run

Pthread

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization

- API specifies behavior of the thread library, implementation is up to development of the library

- Common in UNIX (e.g., Linux) OSes

Pthread APIs

Thread Call	Description
`pthread_create`	Create a new thread in the caller's address space
`pthread_exit`	Terminate the calling thread
`pthread_join`	Wait for a thread to terminate
`pthread_mutex_init`	Create a new mutex
`pthread_mutex_destroy`	Destroy a mutex
`pthread_mutex_lock`	Lock a mutex
`pthread_mutex_unlock`	Unlock a mutex
`pthread_cond_init`	Create a condition variable
`pthread_cond_destroy`	Destroy a condition variable
`pthread_cond_wait`	Wait on a condition variable
`pthread_cond_signal`	Release one thread waiting on a condition variable

Pthread APIs

Thread Call	Description
`pthread_create`	Create a new thread in the caller's address space
`pthread_exit`	Terminate the calling thread
`pthread_join`	Wait for a thread to terminate
`pthread_mutex_init`	Create a new mutex
`pthread_mutex_destroy`	Destroy a mutex
`pthread_mutex_lock`	Lock a mutex
`pthread_mutex_unlock`	Unlock a mutex
`pthread_cond_init`	Create a condition variable
`pthread_cond_destroy`	Destroy a condition variable
`pthread_cond_wait`	Wait on a condition variable
`pthread_cond_signal`	Release one thread waiting on a condition variable

Thread creation

Thread lock

Thread CV

Example of Using Pthread

```
1   #include <stdio.h>
2   #include <assert.h>
3   #include <pthread.h>
4
5   void *mythread(void *arg) {
6       printf("%s\n", (char *) arg);
7       return NULL;
8   }
9
10  int
11  main(int argc, char *argv[]) {
12      pthread_t p1, p2;
13      int rc;
14      printf("main: begin\n");
15      rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
16      rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
17      // join waits for the threads to finish
18      rc = pthread_join(p1, NULL); assert(rc == 0);
19      rc = pthread_join(p2, NULL); assert(rc == 0);
20      printf("main: end\n");
21      return 0;
22  }
```
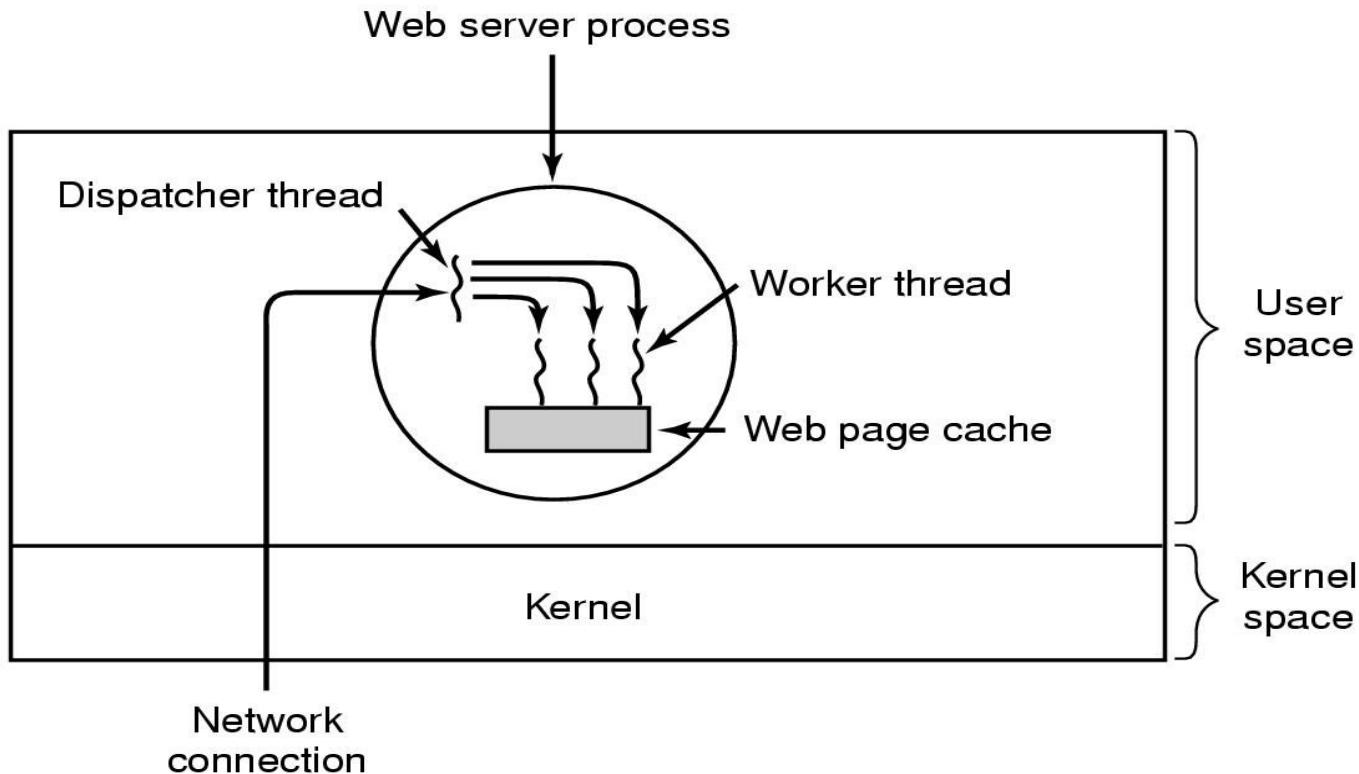
(handwritten annotations)

rc = fork(.)
if (rc == 0) {
p   c
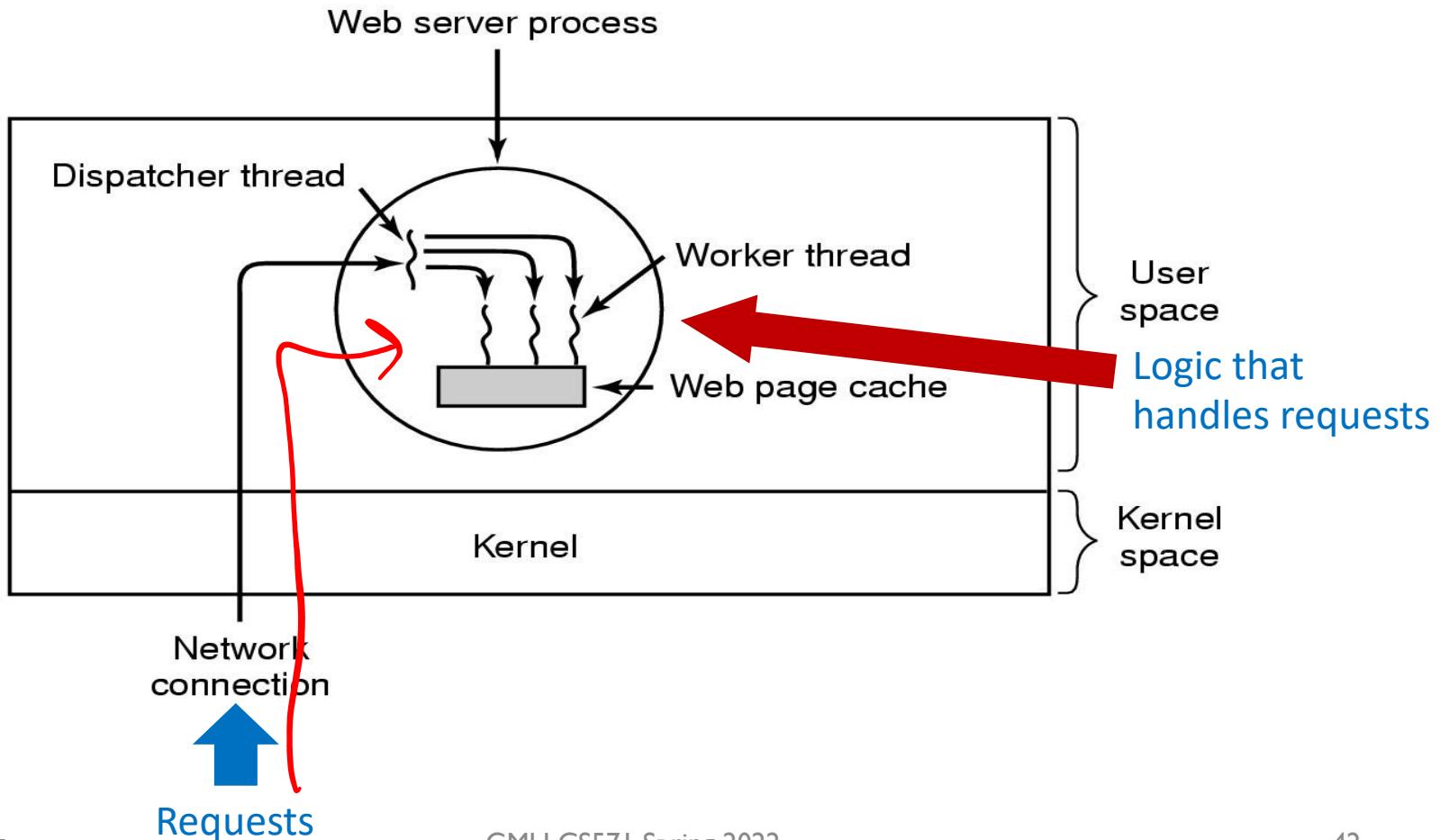
Example Multithreaded Applications

A multithreaded web server

Example Multithreaded Applications

A multithreaded web server

Logic that handles requests

Requests

Code Sketch

P

C

```
while (TRUE) {                        while (TRUE) {

    get_next_request(&buf);              wait_for_work(&buf);
    handoff_work(&buf);                  check_cache(&buf; &page);

}                                        if (not_in_cache)
                                            read_from_disk(&buf,  &page);
                                         return_page(&page);

                                     }
```

(a) Dispatcher thread   (b) Worker thread

Benefits of Multi-threading

- ## Resource sharing
  - Sharing the address space and other resources may result in high degree of cooperation

- ## Economy
  - Creating/managing processes much more time consuming than managing threads: e.g., context switch

- ## Better utilization of multicore architectures
  - Threads are doing job concurrently (or in parallel)
  - Multithreading an interactive application may allow a program to continue running even if part of it is blocked or performing a lengthy operation

Real-world Example: Memcached

- Memcached—A high-performance memory-based caching system
  - Written in C
  - https://memcached.org/

- A typical multithreaded server implementation
  - `Pthread` + `libevent`
  - A dispatcher thread dispatches newly coming connections to the worker threads in a round-robin manner
  - Event-driven: Each worker thread is responsible for serving requests from the established connections
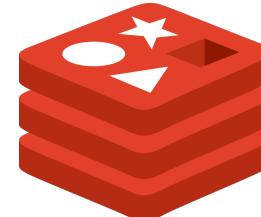
Debate of Multithreading vs. Multi-processes

- Debate of:
  - Multithreading vs. Multi-processes
  - Memcached vs. Redis

- Redis—A single-threaded memory-based data store (written in C)
  - https://redis.io/

Wish List for Redis...

http://goo.gl/N9UTKD
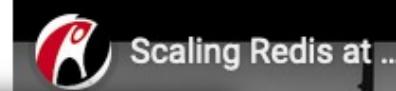
How Twitter Uses Redis To Scale - 105TB RAM, 39MM QPS, 10,000+ Instances

MONDAY, SEPTEMBER 8, 2014 AT 9:05AM

Yao Yue has worked on Twitter's Cache team since 2010.
She recently gave a really great talk: Scaling Redis at

Scaling Redis at ...
at Twitt

Wish List For Redis

- Explicit memory management.

- **Deployable (Lua) Scripts**. Talked about near the start.

- **Multi-threading**. Would make cluster management easier. Twitter has a lot of "tall boxes," where a host has 100+ GB of memory and a lot of CPUs. To use the full capabilities of a server a lot of Redis instances need to be started on a physical machine. With multi-threading fewer instances would need to be started which is much easier to manage.

rience and
orth watching

Concurrency

- Threads

- Race Conditions

- The Critical Section Problem

- 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>
```

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

```
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
addl   $0x1, %eax
```

50

51

Time

Concurrent Access to the Same Memory Address

OS	Thread 1	Thread 2	Value

Thread 1 ↓

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

Interrupt
Save T1's state
Restore T2's state

Thread 2 ↓

movl   0x2f8e, %eax
addl $0x1, %eax
movl   %eax, 0x2f8e

Time

Concurrent Access to the Same Memory Address

OS	Thread 1	Thread 2	Value

Enter into critical section

Thread 1:
```
movl    0x2f8e, %eax        50
addl    $0x1, %eax          51
```

Interrupt
Save T1's state
Restore T2's state

Thread 2:
```
movl    0x2f8e, %eax        50
addl    $0x1, %eax          51
movl    %eax, 0x2f8e        51
```

Time

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

OS	Thread 1	Thread 2	Value

Enter into critical section

Thread 1:
```
movl    0x2f8e, %eax
addl    $0x1, %eax
```
Value: 50, 51

Interrupt
Save T1's state
Restore T2's state

Thread 2:
```
movl    0x2f8e, %eax
addl    $0x1, %eax
movl    %eax, 0x2f8e
```
Value: 50, 51, 51

Interrupt
Save T2's state
Restore T1's state

Thread 1:
```
movl    %eax, 0x2f8e
```

Time

Concurrent Access to the Same Memory Address

| OS	Thread 1	Thread 2	Value

Time

Enter into critical section

movl	0x2f8e, %eax
addl	$0x1, %eax

50

51

Interrupt
Save T1's state
Restore T2's state

movl	0x2f8e, %eax
addl	$0x1, %eax
movl	%eax, 0x2f8e

50

51

51
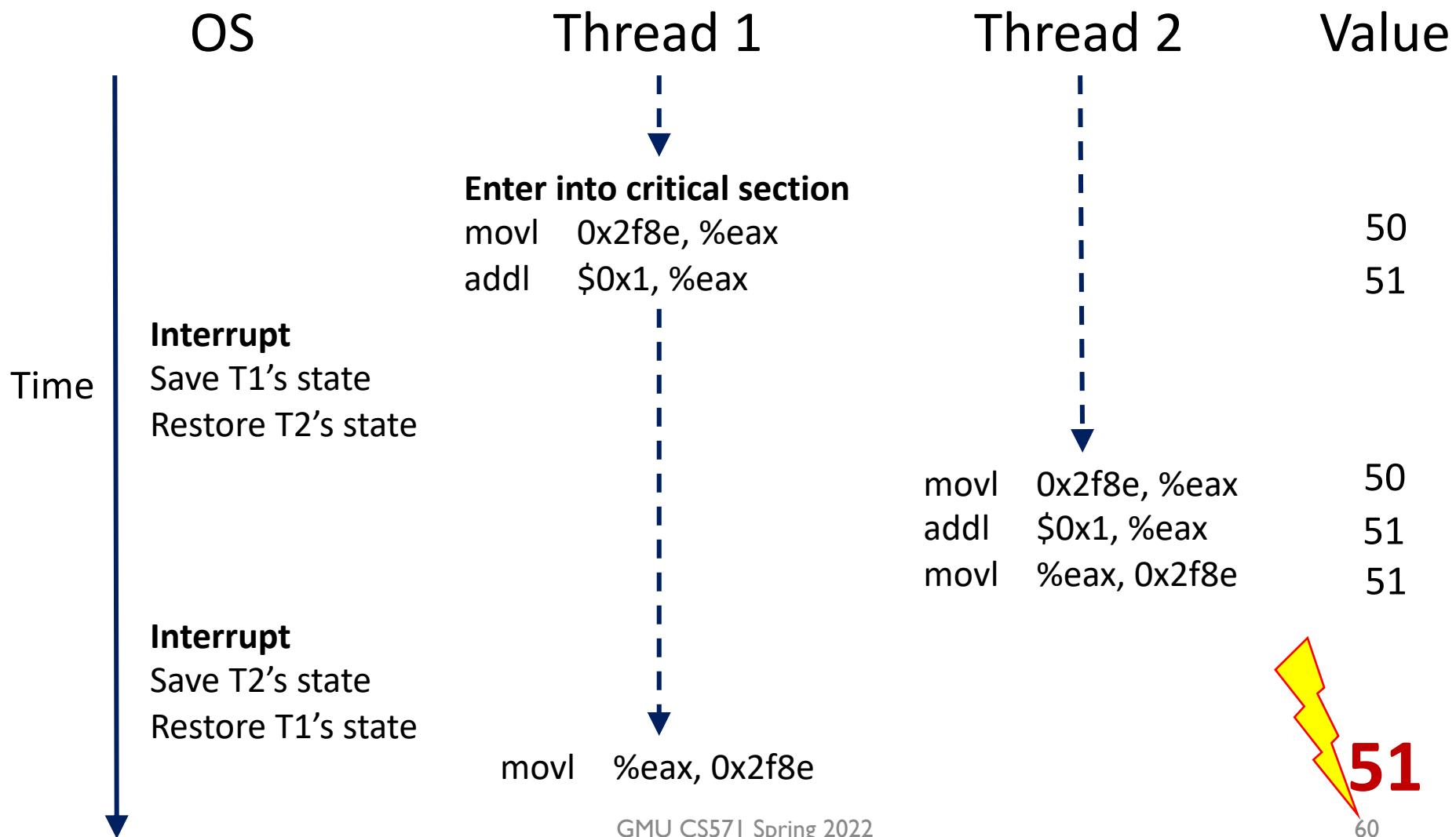
Interrupt
Save T2's state
Restore T1's state

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			
	movl %eax, 0x2f8e		

Time

51

Race Conditions

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

- Any possible orders can happen, which result in different output across runs
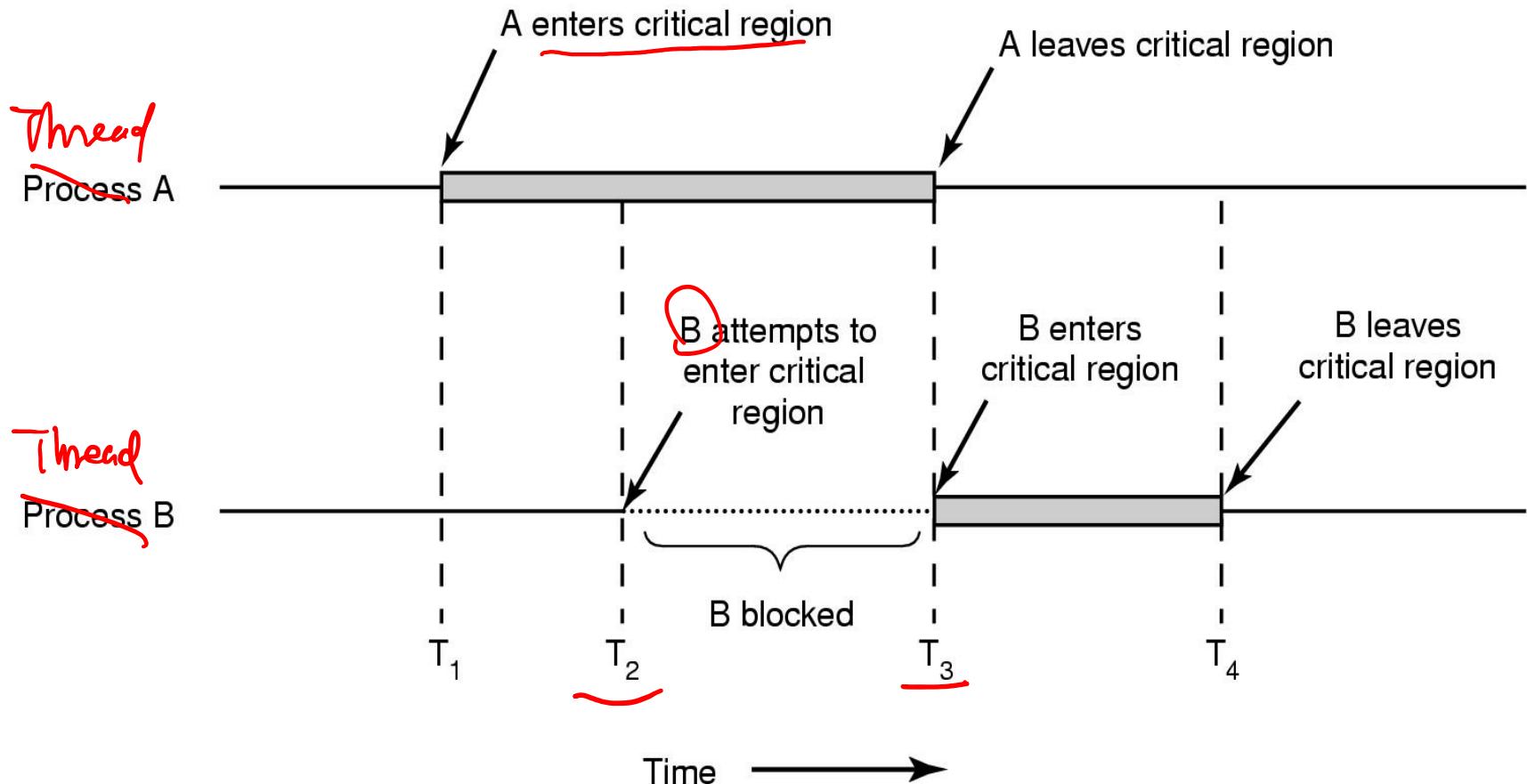
Race Conditions

- Situations like this, where multiple threads 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 thread

The Critical-Section Problem

- *N* 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 thread is executing in its critical section, no other thread is allowed to execute in that critical section

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

Mutual Exclusion

A enters critical region

A leaves critical region

Thread
Process A

B attempts to
enter critical
region

B enters
critical region

B leaves
critical region

Thread
Process B

B blocked

T_1 T_2 T_3 T_4

Time

Using **Lock** to Protect Shared Data

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

Thread A: Thread B:

`balance = balance + 1`   `balance = balance + 1`

```
1   lock_t mutex; // some globally-allocated lock 'mutex'
2   ...
3   lock(&mutex);
4   balance = balance + 1;
5   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?

```
1    typedef struct __lock_t { int flag; } lock_t;
2
3    void init(lock_t *mutex) {
4        // 0 -> lock is available, 1 -> held
5        mutex->flag = 0;
6    }
7
8    void lock(lock_t *mutex) {
9        while (mutex->flag == 1)   // TEST the flag
10           ;  // spin-wait (do nothing)
11       mutex->flag = 1;           // now SET it!
12   }
13
14   void unlock(lock_t *mutex) {
15       mutex->flag = 0;
16   }
```

First Attempt: A Simple Flag

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

```
1   typedef struct __lock_t { int flag; } lock_t;
2
3   void init(lock_t *mutex) {
4       // 0 -> lock is available, 1 -> held
5       mutex->flag = 0;
6   }
7
8   void lock(lock_t *mutex) {
9       while (mutex->flag == 1)   // TEST the flag
10          ; // spin-wait (do nothing)              A spin lock
11      mutex->flag = 1;           // now SET it!
12  }
13
14  void unlock(lock_t *mutex) {
15      mutex->flag = 0;
16  }
```

First Attempt: A Simple Flag

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

```
1    typedef struct __lock_t { int flag; } lock_t;
2
3    void init(lock_t *mutex) {
4        // 0 -> lock is available, 1 -> held
5        mutex->flag = 0;
6    }
7
8    void lock(lock_t *mutex) {
9        while (mutex->flag == 1)   // TEST the flag
10           ; // spin-wait (do nothing)
11       mutex->flag = 1;           // now SET it!
12   }
13
14   void unlock(lock_t *mutex) {
15       mutex->flag = 0;
16   }
```

Interrupt

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	

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!)	

lock. (handwritten annotation spanning Thread 1 lock section)

lock. (handwritten annotation spanning Thread 2 lock section)

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!)	

Culprit:
Lock operation is not atomic!
Therefore, no mutual exclusion!
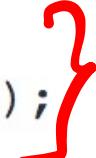
Getting Help from the Hardware

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

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

```
1    void lock() {
2        DisableInterrupts();
3    }
4    void unlock() {
5        EnableInterrupts();
6    }
```

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);
```

```
1   void lock() {
2       DisableInterrupts();
3   }
4   void unlock() {
5       EnableInterrupts();
6   }
```

Are we done??

Synchronization Hardware

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

  read- modify — write.

- 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

  1st   *2nd*

```
1   int TestAndSet(int *old_ptr, int new) {
2       int old = *old_ptr;   // fetch old value at old_ptr
3       *old_ptr = new;        // store 'new' into old_ptr
4       return old;            // return the old value
5   }
```

Operations performed atomically!

Mutual Exclusion with TAS

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

```
1    typedef struct __lock_t {
2        int flag;
3    } lock_t;
4
5    void init(lock_t *lock) {
6        // 0 indicates that lock is available, 1 that it is held
7        lock->flag = 0;
8    }
9
10   void lock(lock_t *lock) {
11       while (TestAndSet(&lock->flag, 1) == 1)
12           ; // spin-wait (do nothing)
13   }
14
15   void unlock(lock_t *lock) {
16       lock->flag = 0;
17   }
```

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)

- Disadvantages?
  - Fairness?
  - Performance?

A Simple Approach: Yield!

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

```
1    void init() {
2        flag = 0;
3    }
4
5    void lock() {
6        while (TestAndSet(&flag, 1) == 1)
7            yield(); // give up the CPU
8    }
9
10   void unlock() {
11       flag = 0;
12   }
```

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))`

  In Linux kernel

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
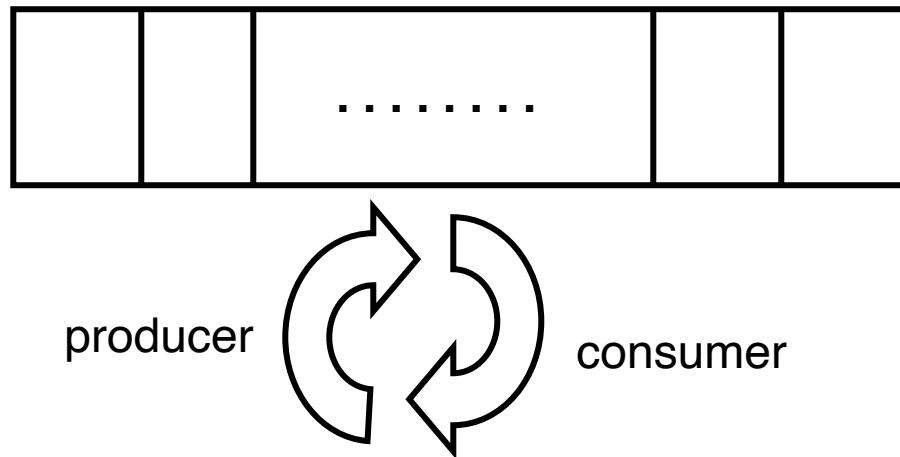
Classical Synchronization Problems

- **Producer-Consumer Problem**
  - **Semaphore version**   Today
  - Condition Variable
    - A CV-based version

- Readers-Writers Problem

- Dining-Philosophers Problem

Producer-Consumer Problem
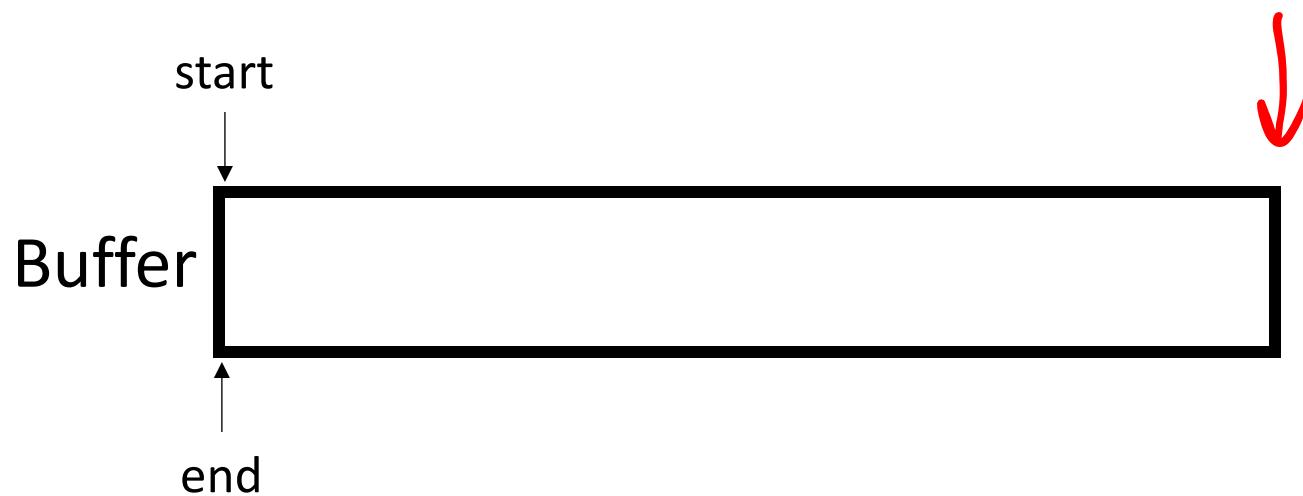
- The bounded-buffer producer-consumer problem assumes that there is a buffer of size N

- The producer process puts items to the buffer area

- The consumer process consumes items from the buffer

- The producer and the consumer execute concurrently

producer   consumer
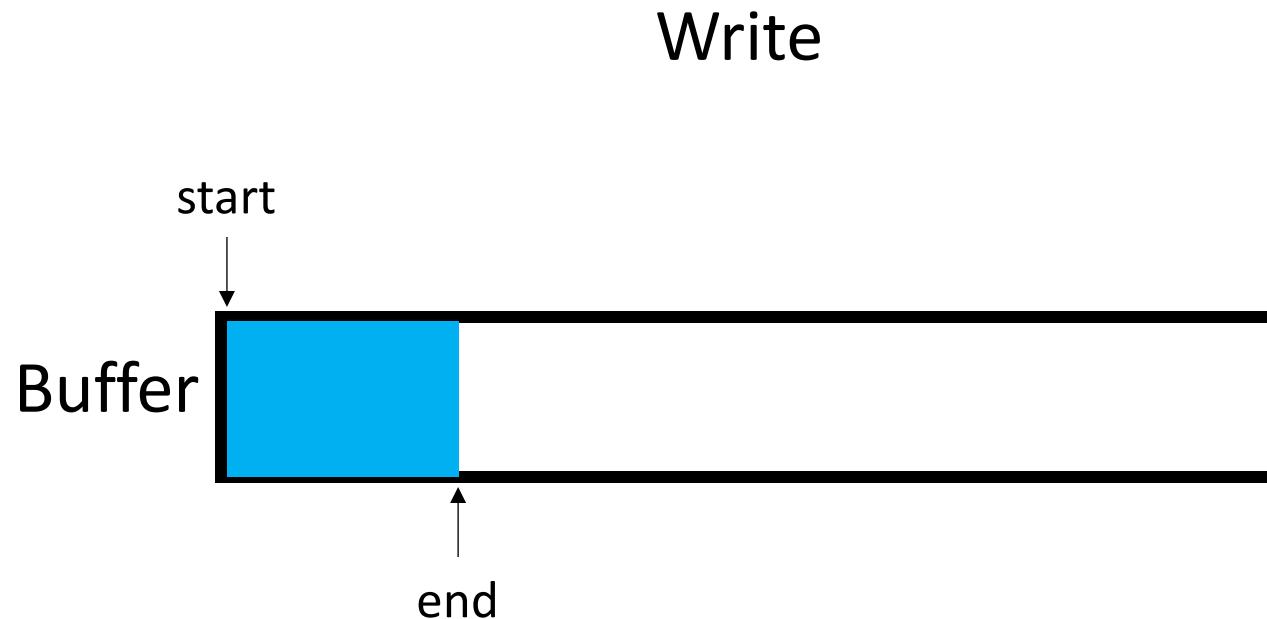
Example: Unix Pipes

- 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
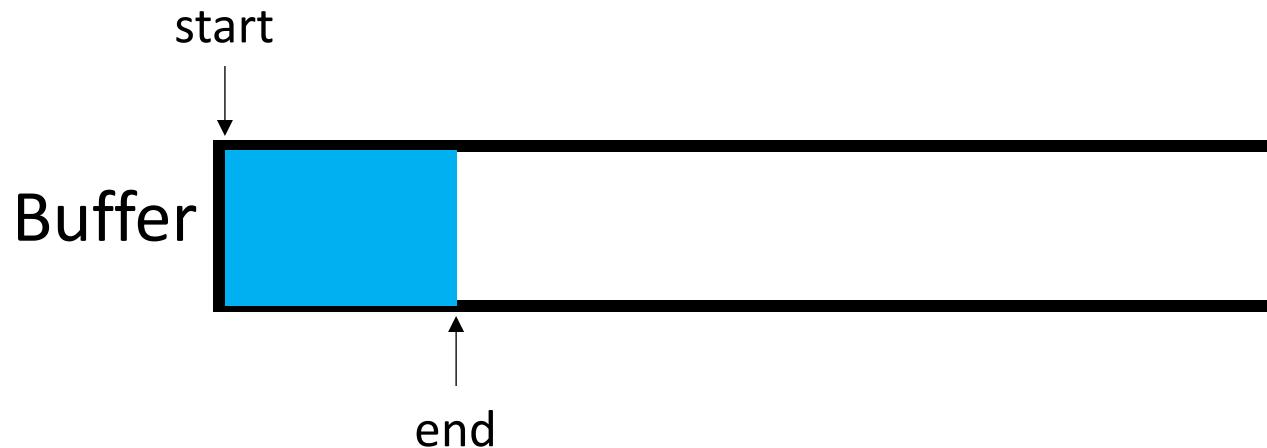
Example: Unix Pipes
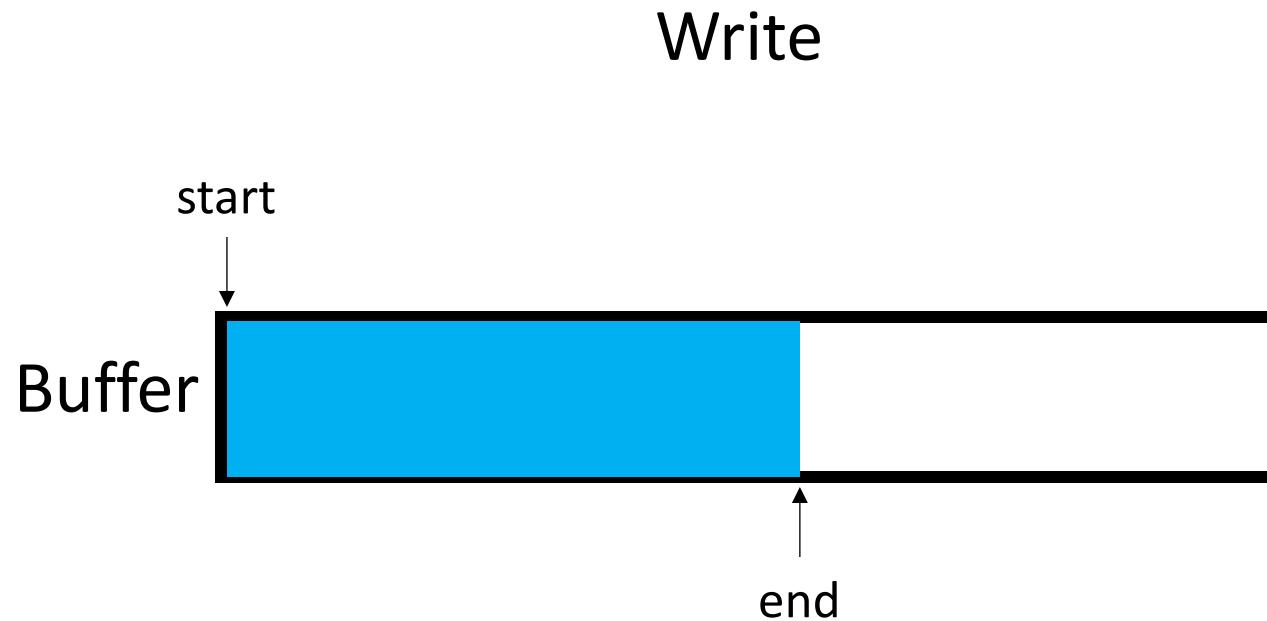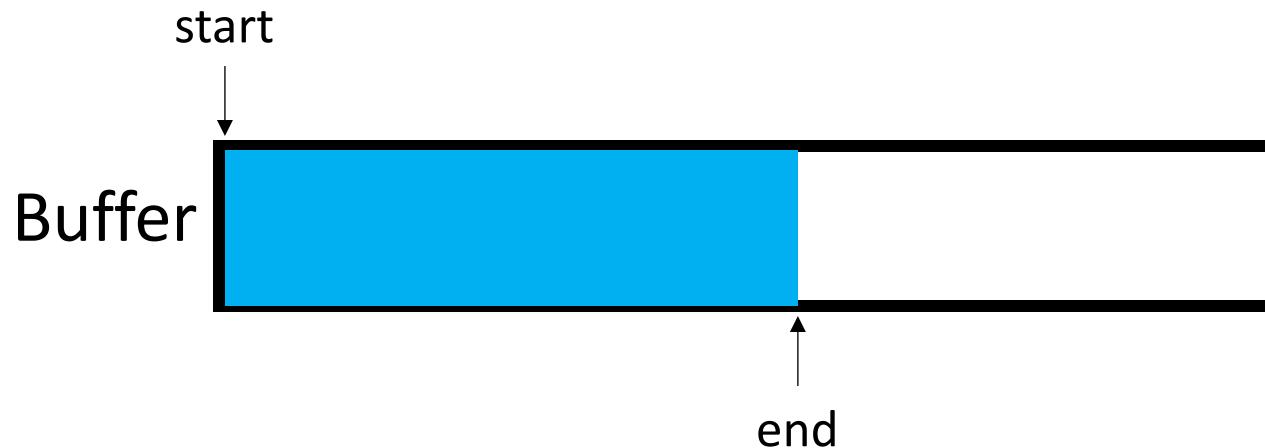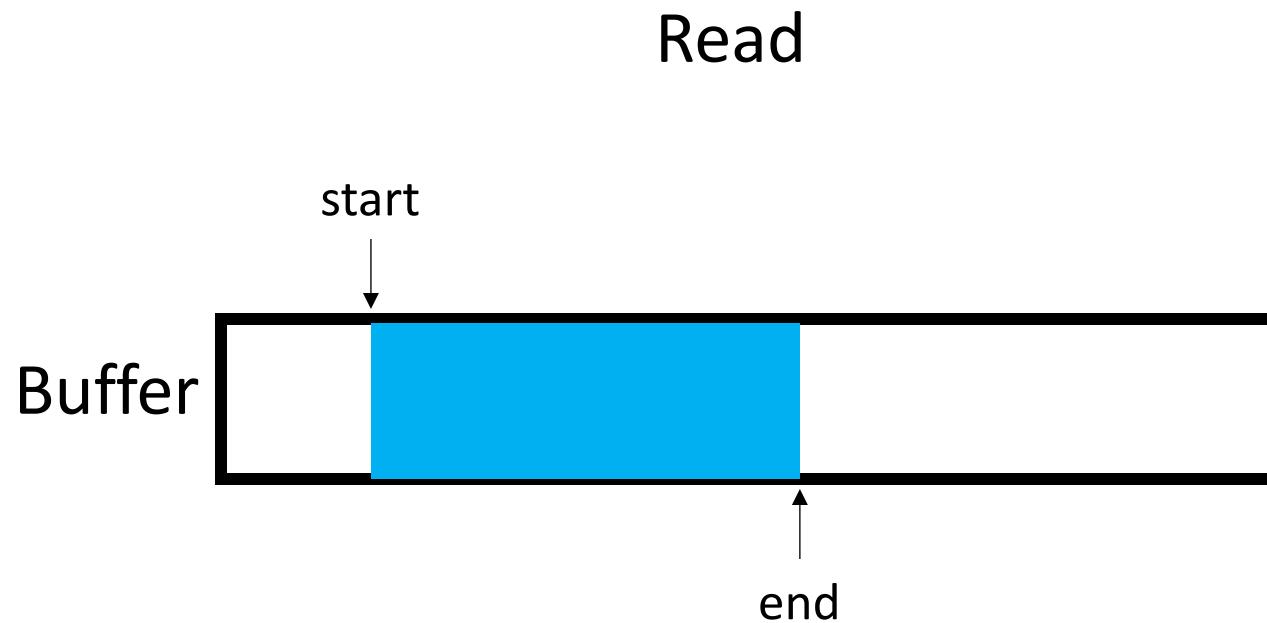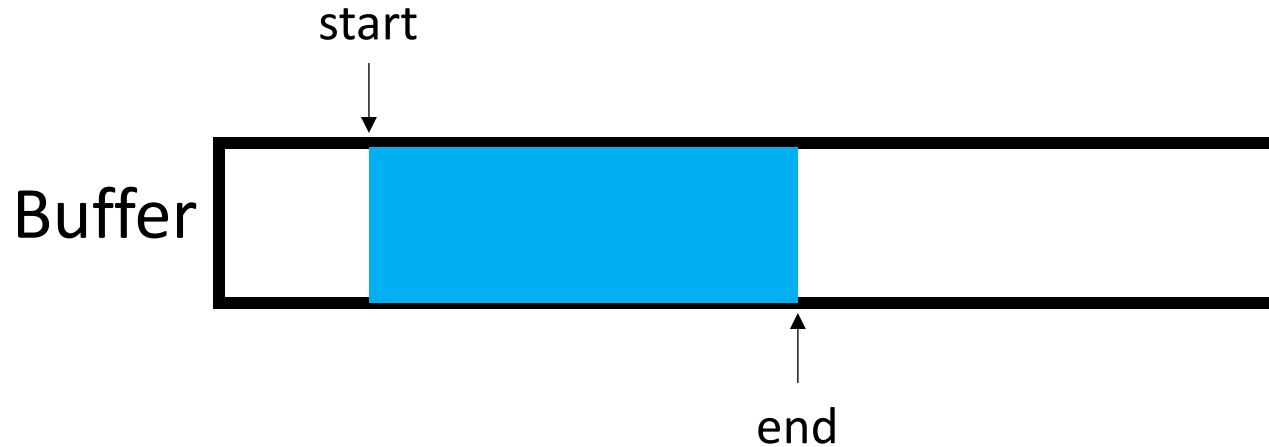
start

Buffer

end

Example: Unix Pipes

Write

start

Buffer

end

Example: Unix Pipes

start

Buffer

end

Example: Unix Pipes

Write

start

Buffer

end

Example: Unix Pipes

start

Buffer

end

Example: Unix Pipes

Read

start

Buffer

end

Example: Unix Pipes

start

Buffer

end

Example: Unix Pipes

Write

start

Buffer

end

Example: Unix Pipes

start

Buffer

end

Example: Unix Pipes

Read

start

Buffer

end

Example: Unix Pipes

Read

start

Buffer

end

Example: Unix Pipes

Read

start

Buffer

end
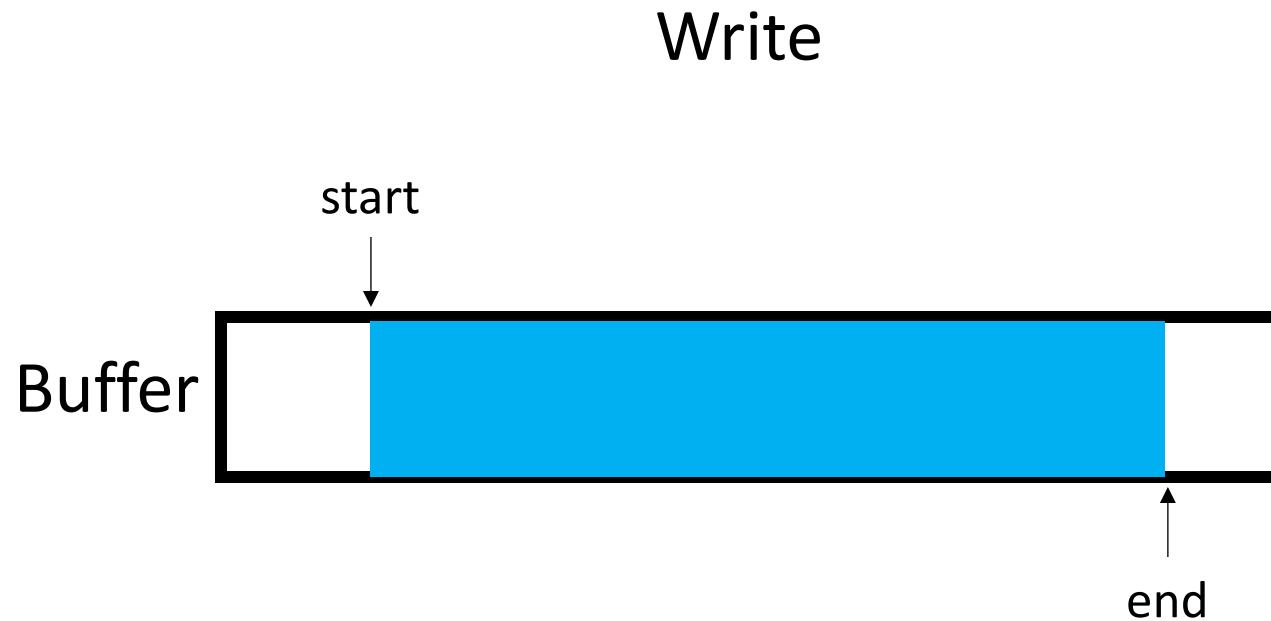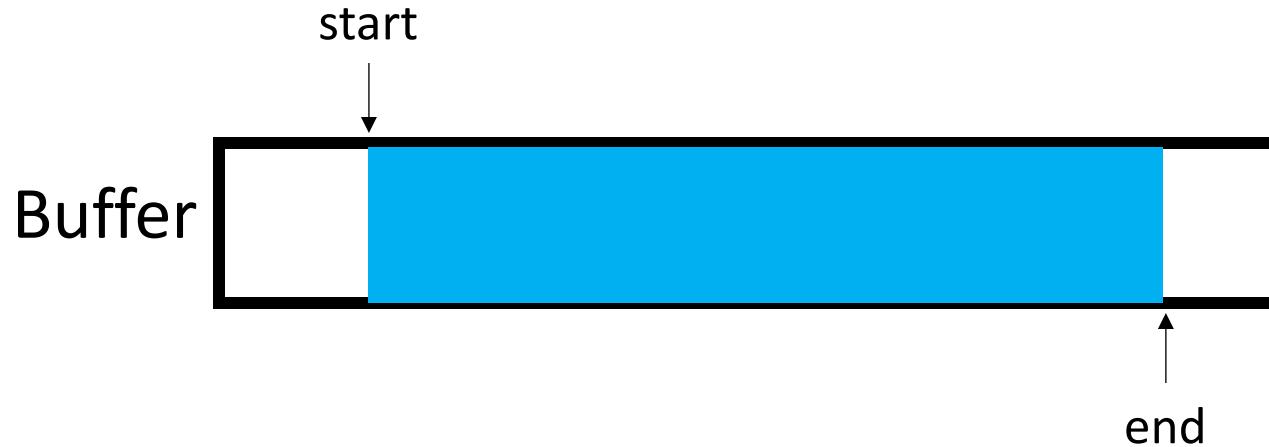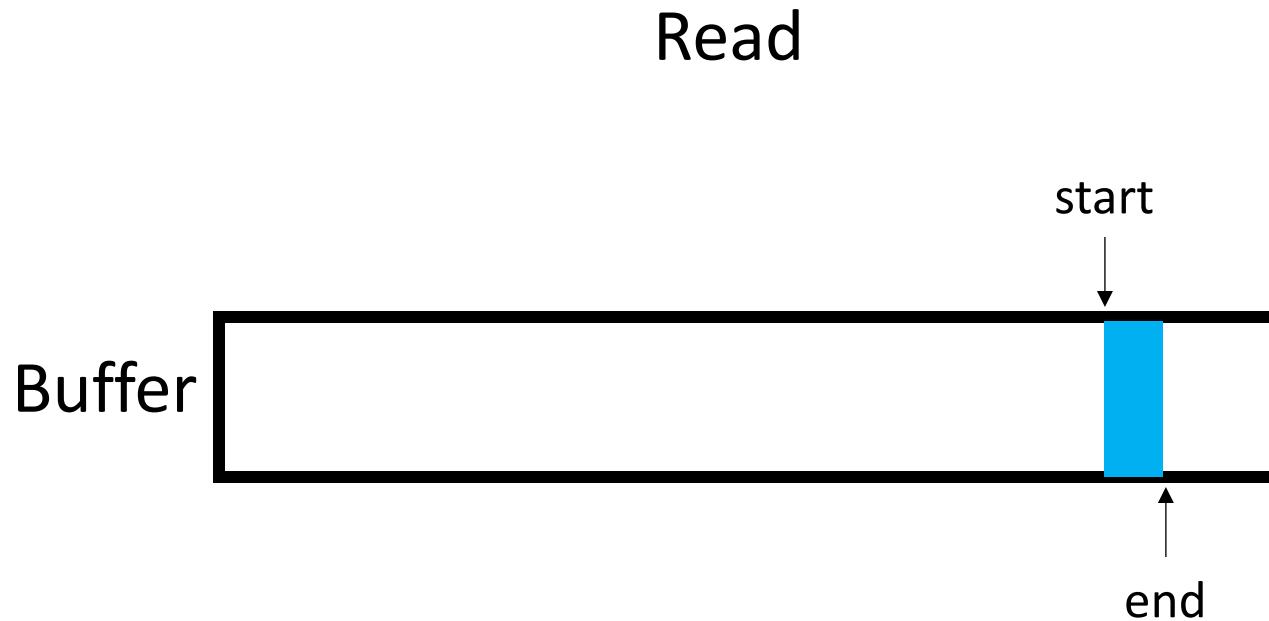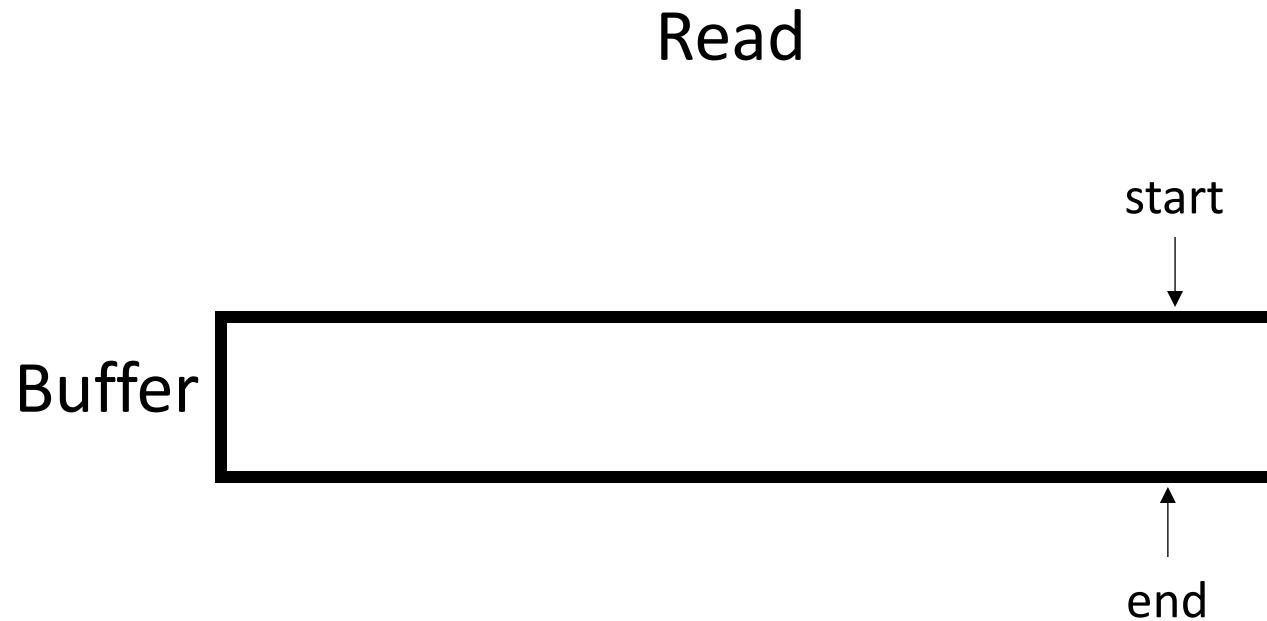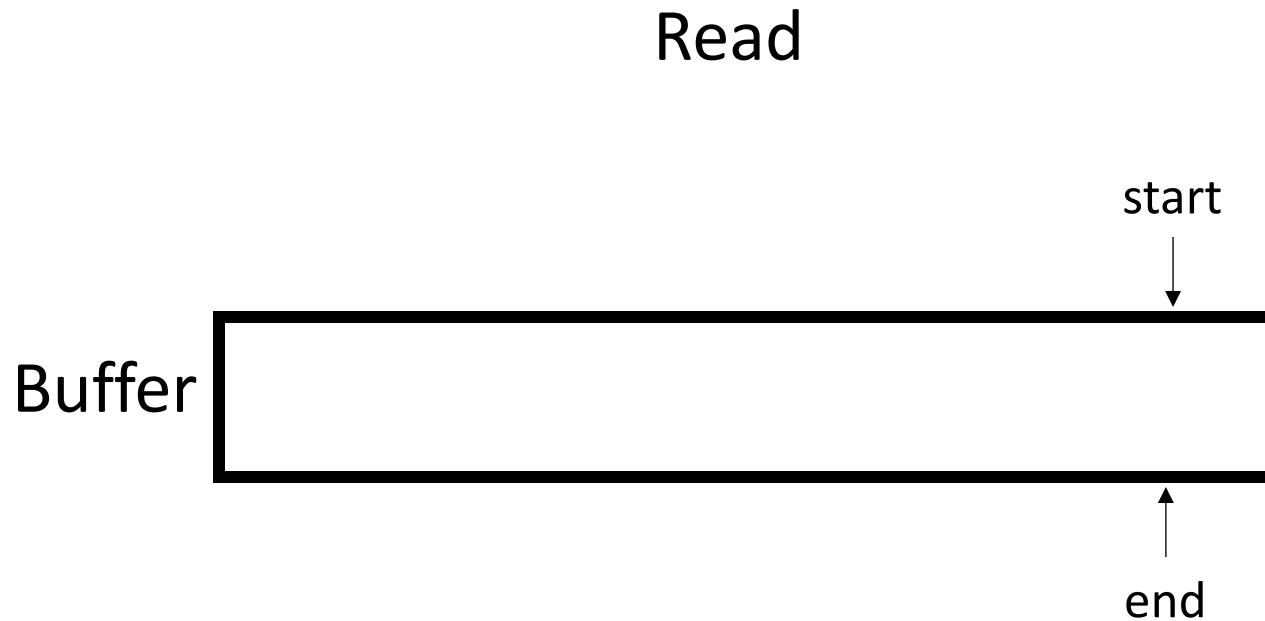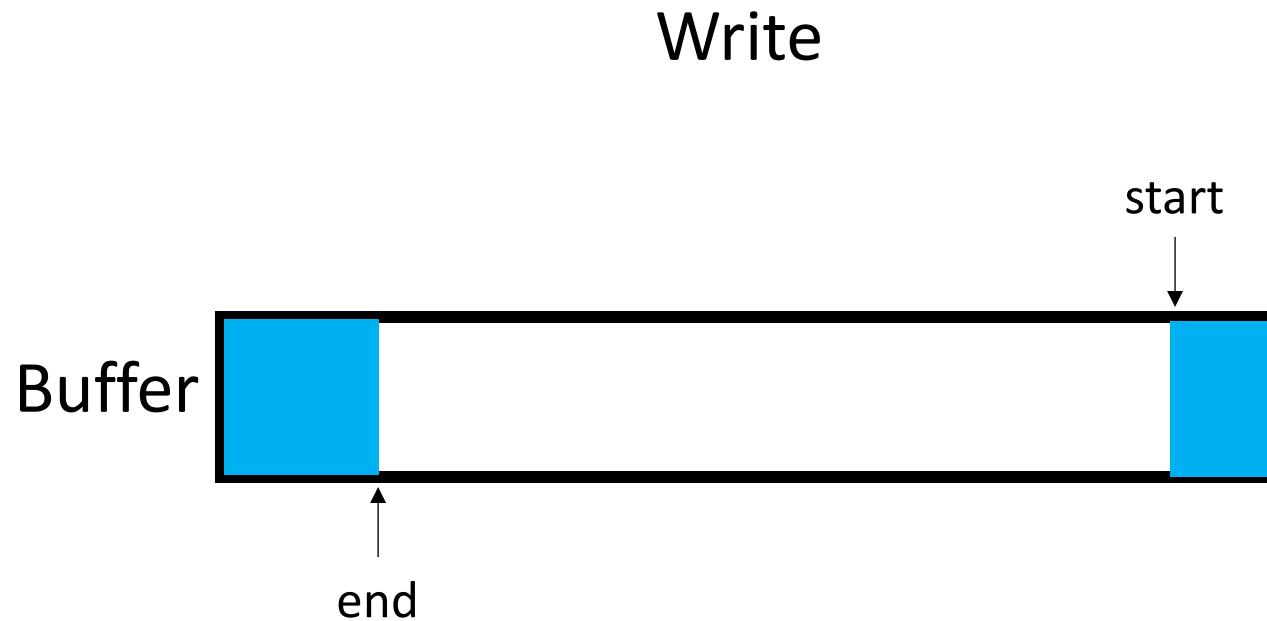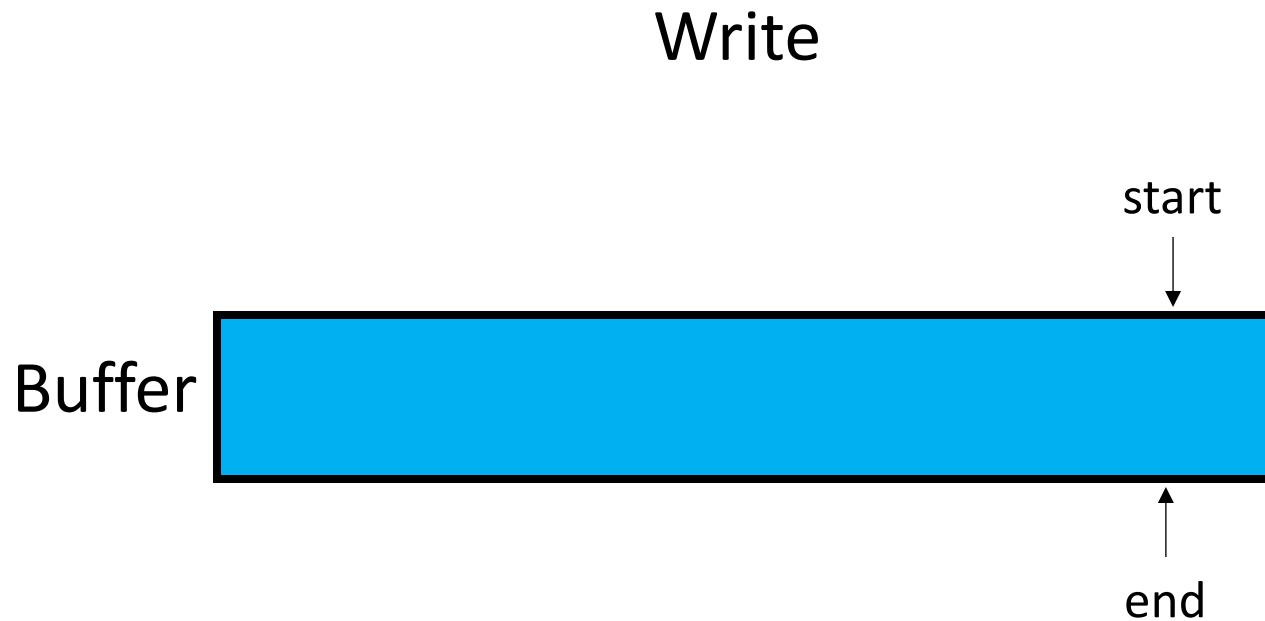
Note: reader must **wait**

Example: Unix Pipes

Write

Buffer

start

end

Example: Unix Pipes

Write

start

Buffer

end

Example: Unix Pipes

Write

start

Buffer

end
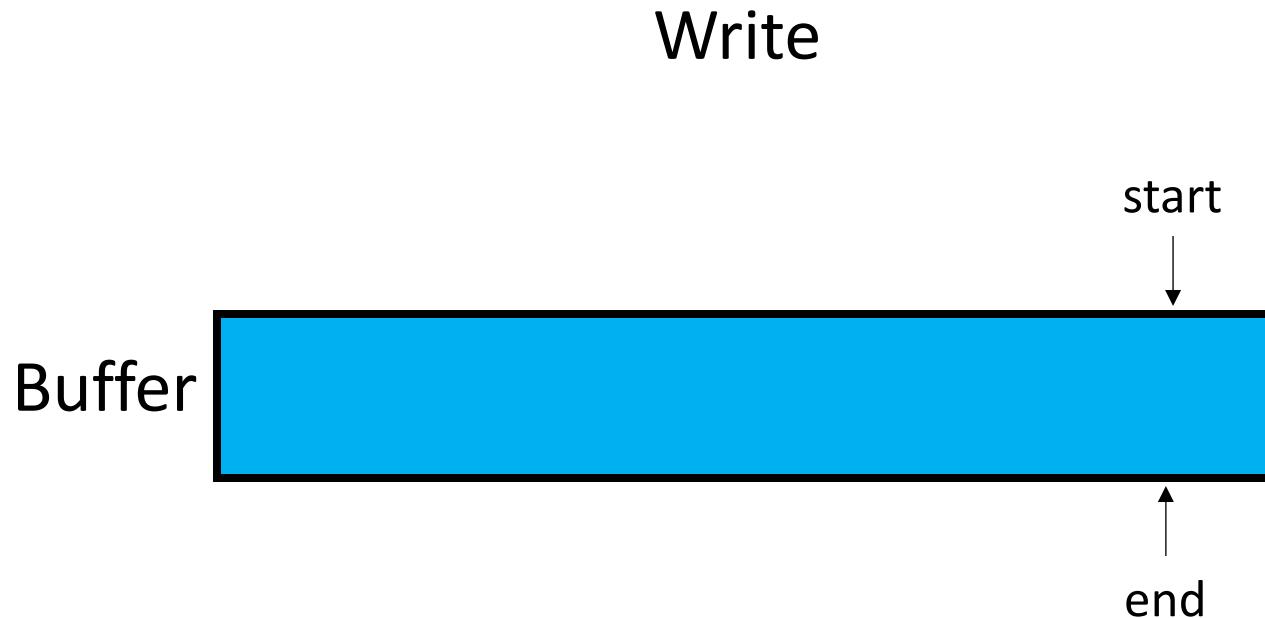
Note: writer must **wait**

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

```
% 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

```
1    sem_t empty;
2    sem_t full;
3
4    void *producer(void *arg) {
5        int i;
6        for (i = 0; i < loops; i++) {
7            sem_wait(&empty);          // line P1
8            put(i);                    // line P2
9            sem_post(&full);           // line P3
10       }
11   }
12
13   void *consumer(void *arg) {
14       int i, tmp = 0;
15       while (tmp != -1) {
16           sem_wait(&full);           // line C1
17           tmp = get();               // line C2
18           sem_post(&empty);          // line C3
19           printf("%d\n", tmp);
20       }
21   }
22
23   int main(int argc, char *argv[]) {
24       // ...
25       sem_init(&empty, 0, MAX);  // MAX buffers are empty to begin with...
26       sem_init(&full, 0, 0);     // ... and 0 are full
27       // ...
28   }
```

```
1    int buffer[MAX];
2    int fill = 0;
3    int use  = 0;
4
5    void put(int value) {
6        buffer[fill] = value;
7        fill = (fill + 1) % MAX;
8    }
9
10   int get() {
11       int tmp = buffer[use];
12       use = (use + 1) % MAX;
13       return tmp;
14   }
```

Put and Get routines

First Attempt: MAX = 10?

```
1    sem_t empty;
2    sem_t full;
3
4    void *producer(void *arg) {
5        int i;
6        for (i = 0; i < loops; i++) {
7            sem_wait(&empty);          // line P1
8            put(i);                    // line P2
9            sem_post(&full);           // line P3
10       }
11   }
12
13   void *consumer(void *arg) {
14       int i, tmp = 0;
15       while (tmp != -1) {
16           sem_wait(&full);           // line C1
17           tmp = get();               // line C2
18           sem_post(&empty);          // line C3
19           printf("%d\n", tmp);
20       }
21   }
22
23   int main(int argc, char *argv[]) {
24       // ...
25       sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
26       sem_init(&full, 0, 0);    // ... and 0 are full
27       // ...
28   }
```

```
1    int buffer[MAX];
2    int fill = 0;
3    int use  = 0;
4
5    void put(int value) {
6        buffer[fill] = value;
7        fill = (fill + 1) % MAX;
8    }
9
10   int get() {
11       int tmp = buffer[use];
12       use = (use + 1) % MAX;
13       return tmp;
14   }
```

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**

Producer 1: Runnable

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

```
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

Producer 1: **Running**

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

```
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;   mutex;
  ```

- 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"

```
1   sem_t empty;
2   sem_t full;
3   sem_t mutex;
4
5   void *producer(void *arg) {
6       int i;
7       for (i = 0; i < loops; i++) {
8           sem_wait(&mutex);              // line p0 (NEW LINE)
9           sem_wait(&empty);             // line p1
10          put(i);                       // line p2
11          sem_post(&full);              // line p3
12          sem_post(&mutex);             // line p4 (NEW LINE)
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          sem_wait(&mutex);             // line c0 (NEW LINE)
20          sem_wait(&full);              // line c1
21          int tmp = get();              // line c2
22          sem_post(&empty);             // line c3
23          sem_post(&mutex);             // line c4 (NEW LINE)
24          printf("%d\n", tmp);
25      }
26  }
27
28  int main(int argc, char *argv[]) {
29      // ...
30      sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31      sem_init(&full, 0, 0);    // ... and 0 are full
32      sem_init(&mutex, 0, 1)    // mutex=1 because it is a lock (NEW LINE)
33      // ...
34  }
```

129

Add "Mutual Exclusion"

```
1   sem_t empty;
2   sem_t full;
3   sem_t mutex;
4
5   void *producer(void *arg) {
6       int i;
7       for (i = 0; i < loops; i++) {
8           sem_wait(&mutex);           // line p0 (NEW LINE)
9           sem_wait(&empty);           // line p1
10          put(i);                     // line p2
11          sem_post(&full);            // line p3
12          sem_post(&mutex);           // line p4 (NEW LINE)
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          sem_wait(&mutex);           // line c0 (NEW LINE)
20          sem_wait(&full);            // line c1
21          int tmp = get();            // line c2
22          sem_post(&empty);           // line c3
23          sem_post(&mutex);           // line c4 (NEW LINE)
24          printf("%d\n", tmp);
25      }
26  }
27
28  int main(int argc, char *argv[]) {
29      // ...
30      sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31      sem_init(&full, 0, 0);    // ... and 0 are full
32      sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock (NEW LINE)
33      // ...
34  }
```

What if consumer gets to run first??

130

Adding "Mutual Exclusion"

mutex = 1
full = 0
empty = 10

Producer 0: Runnable   Consumer 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);
    }
}
```

```
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   Consumer 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);
    }
}
```

```
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**

Consumer 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);
    }
}
```

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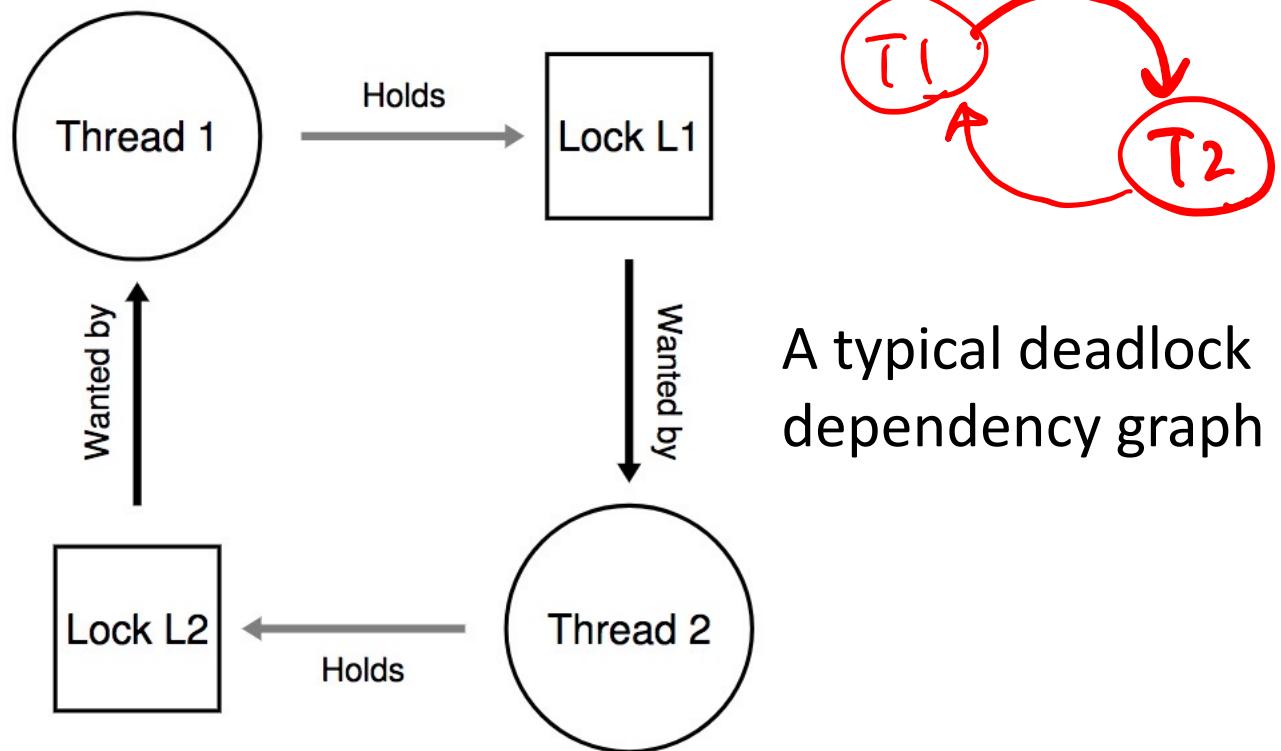
```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&full);
        int tmp = get();
        sem_post(&empty);
        sem_post(&mutex);
        printf("%d\n", tmp);
    }
}
```

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

```
1   sem_t empty;
2   sem_t full;
3   sem_t mutex;
4
5   void *producer(void *arg) {
6       int i;
7       for (i = 0; i < loops; i++) {
8           sem_wait(&empty);              // line p1
9           sem_wait(&mutex);              // line p1.5 (MOVED MUTEX HERE...)
10          put(i);                        // line p2
11          sem_post(&mutex);              // line p2.5 (... AND HERE)
12          sem_post(&full);               // line p3
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          sem_wait(&full);               // line c1
20          sem_wait(&mutex);              // line c1.5 (MOVED MUTEX HERE...)
21          int tmp = get();               // line c2
22          sem_post(&mutex);              // line c2.5 (... AND HERE)
23          sem_post(&empty);              // line c3
24          printf("%d\n", tmp);
25      }
26  }
27
28  int main(int argc, char *argv[]) {
29      // ...
30      sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31      sem_init(&full, 0, 0);    // ... and 0 are full
32      sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
33      // ...
34  }
```

Mutex wraps just around critical section!

Mutex wraps just around critical section!

137

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