Concurrency: Threads, Locks, and Semaphores

CS 571: Operating Systems (Spring 2021)

Lecture 7

Yue Cheng

Some material taken/derived from:

- Wisconsin CS-537 materials created by Remzi Arpaci-Dusseau.

Announcements

- Project 2's deadline is extended by one week
  - Due at 11:59pm, 03/26

- Project 3-5 will be team projects
  - Please fill out the Google form about your team composition: https://forms.gle/DwNN1pZPn5J6jFAS9
  - Feel free to post on Piazza to search for teammates!

Concurrency

- Threads
- Race Conditions
- The Critical Section Problem
- Locks
- Semaphores

Threads

Why Thread Abstraction?

Process Abstraction: Challenge 1

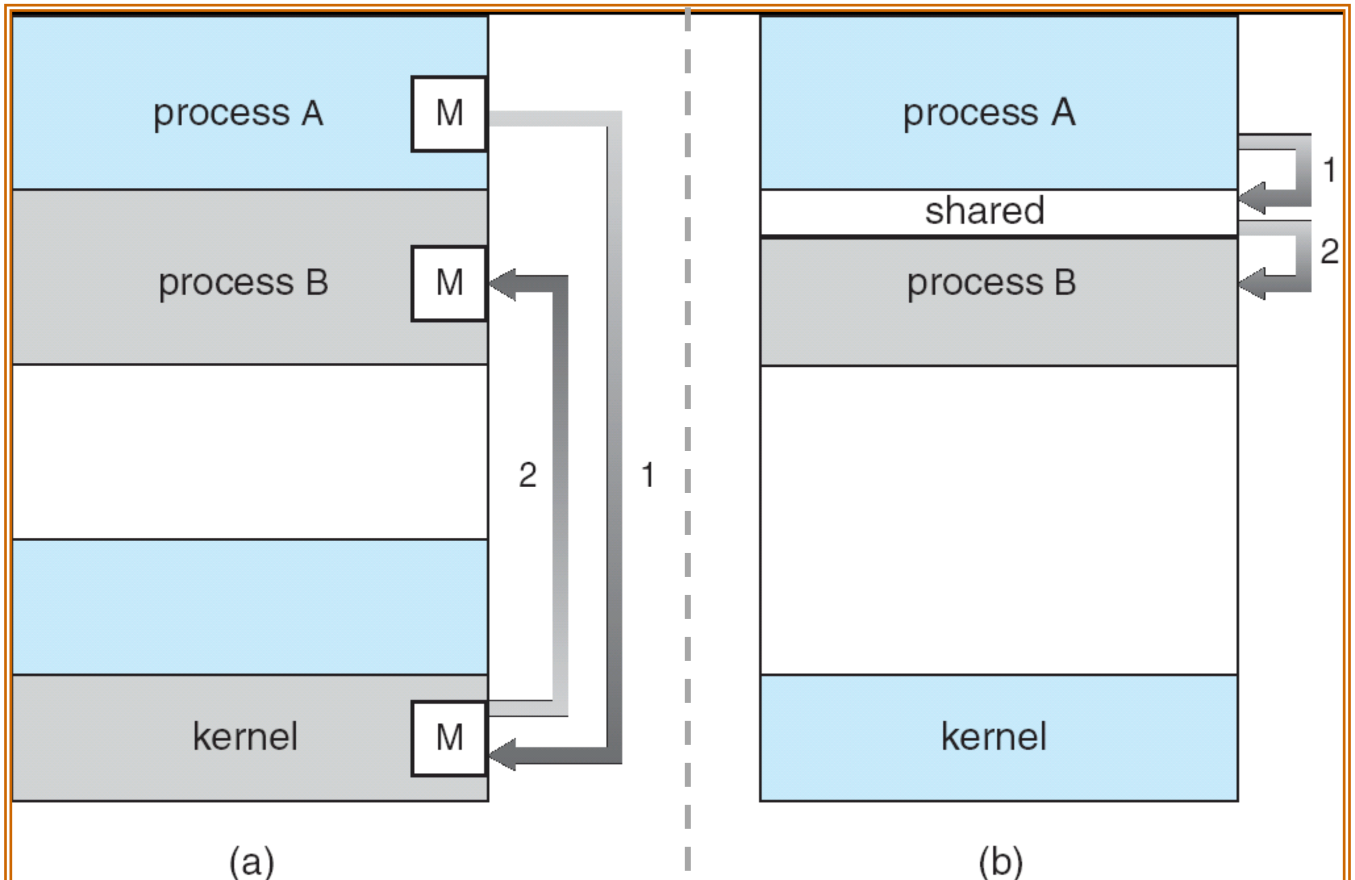
- Inter-process communication (IPC)

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



Message Passing | 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)`

- 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` creates a shared memory block
  - `shmat` maps/attaches an existing shared memory block into a process's address space
  - `shmdt` 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

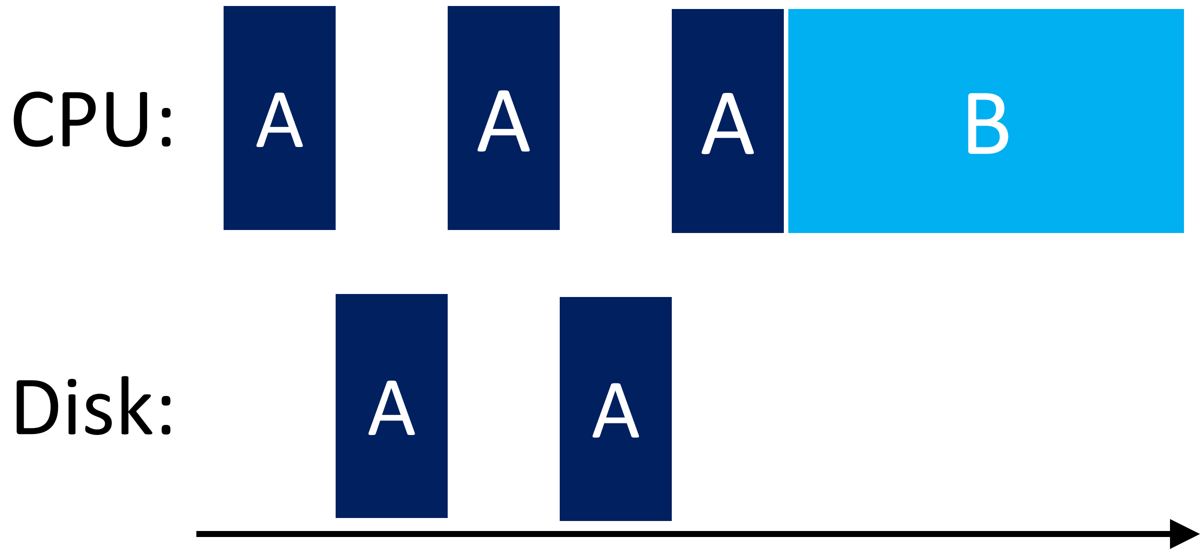
- Inter-process communication (IPC)
  - Cumbersome programming!
  - Copying overheads (inefficient communication)
  - Expensive context switching (why expensive?)

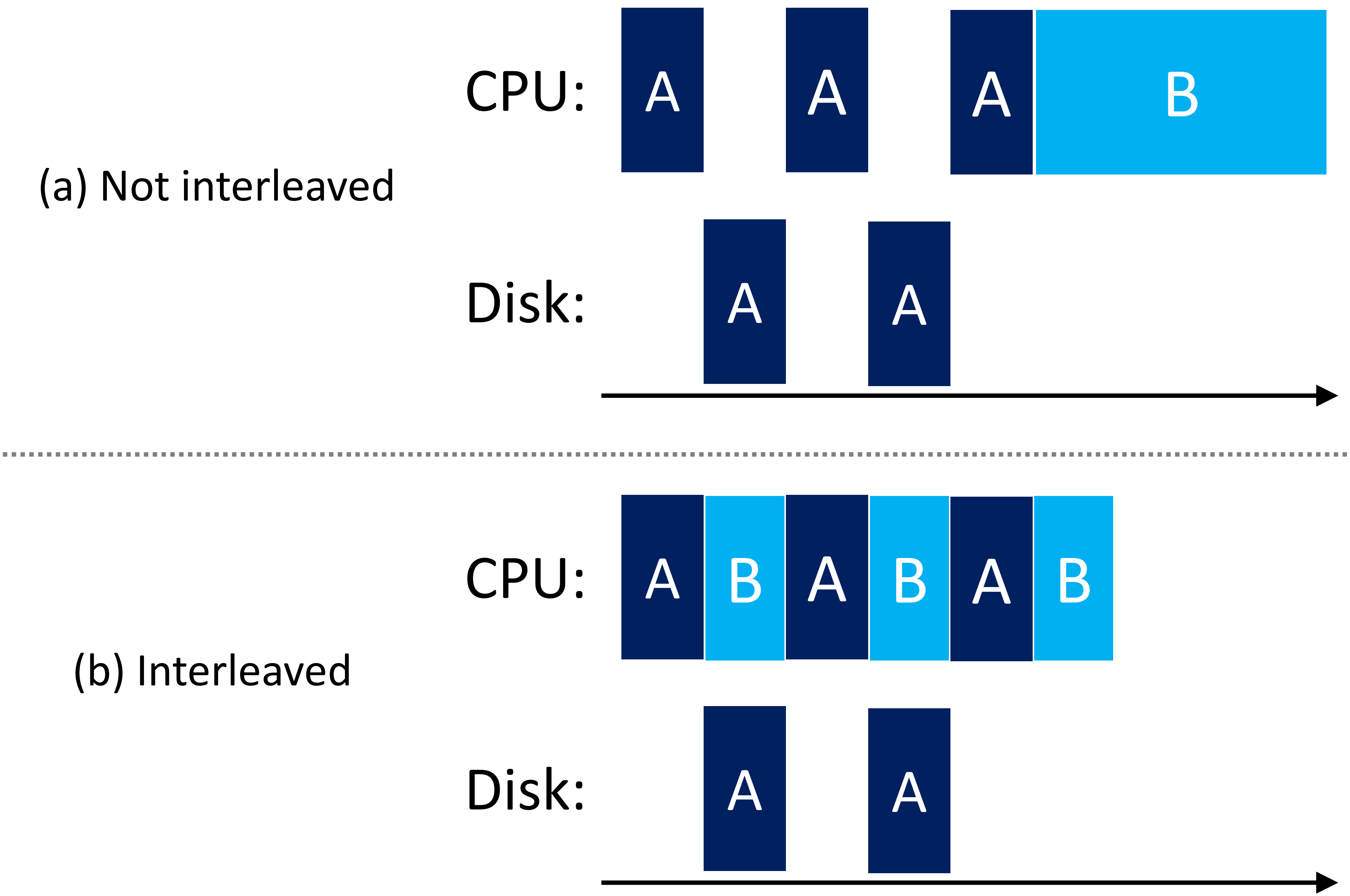
Process Abstraction: Challenge 2

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

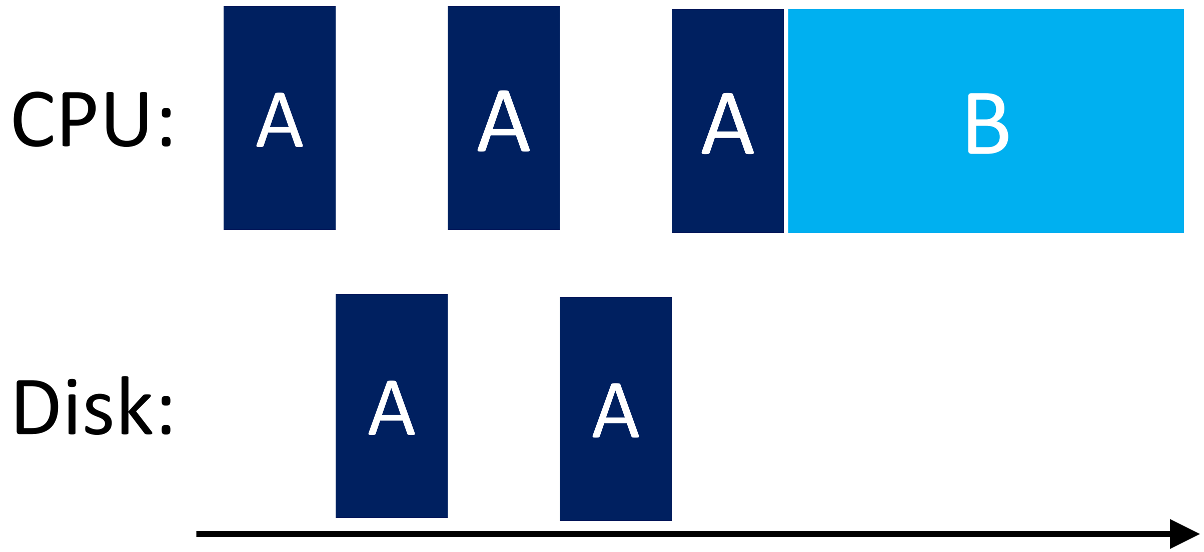
- CPU utilization

(a) Not interleaved



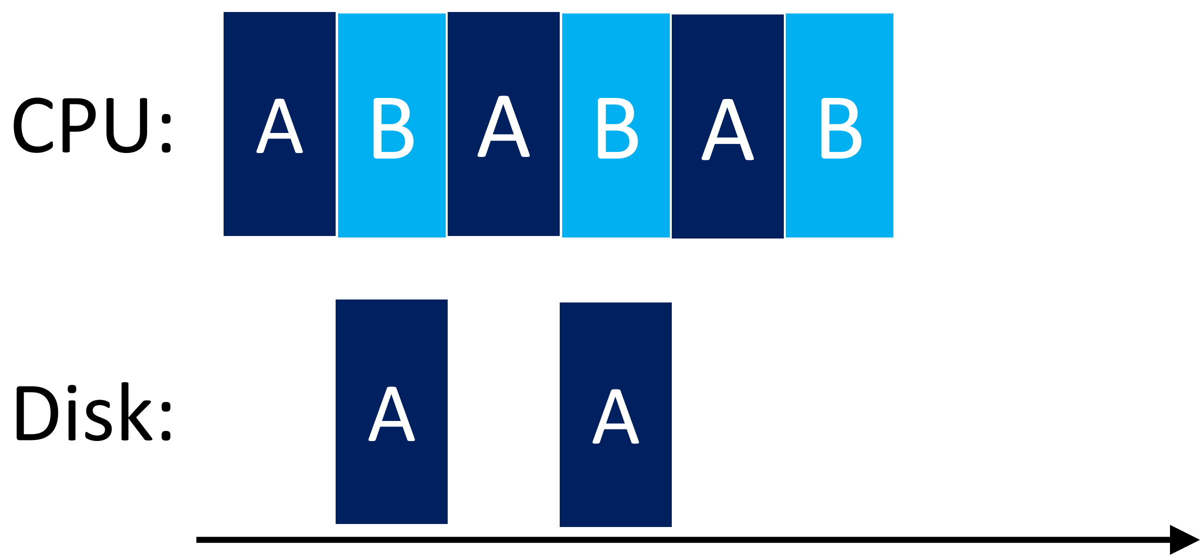


(a) Not interleaved
CPU:
A
A
A
B
Disk:
A
A
(b) Interleaved
CPU:
A
B
A
B
A
B
Disk:
A
A



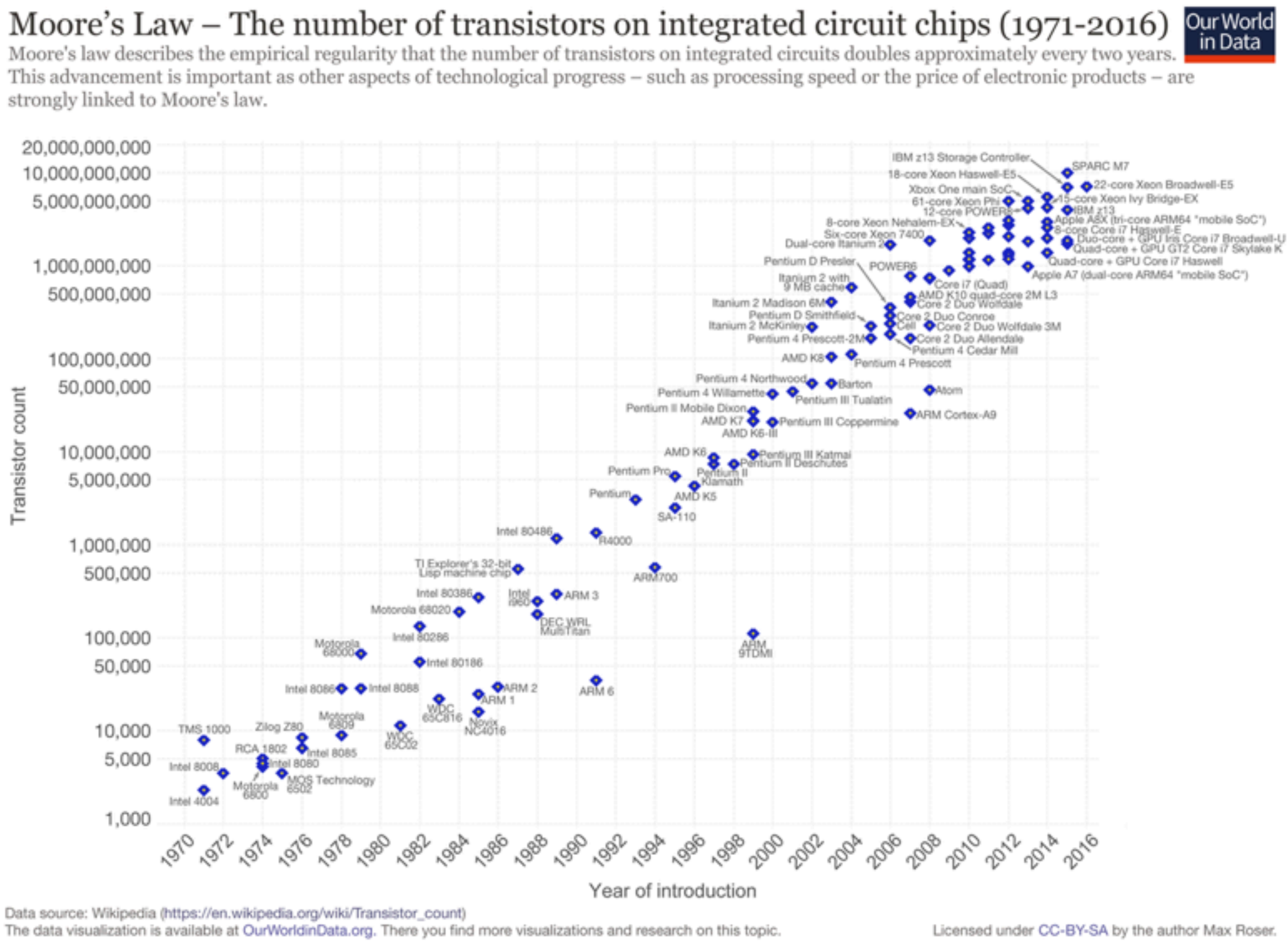
(a) Not interleaved

What if there is only one process?

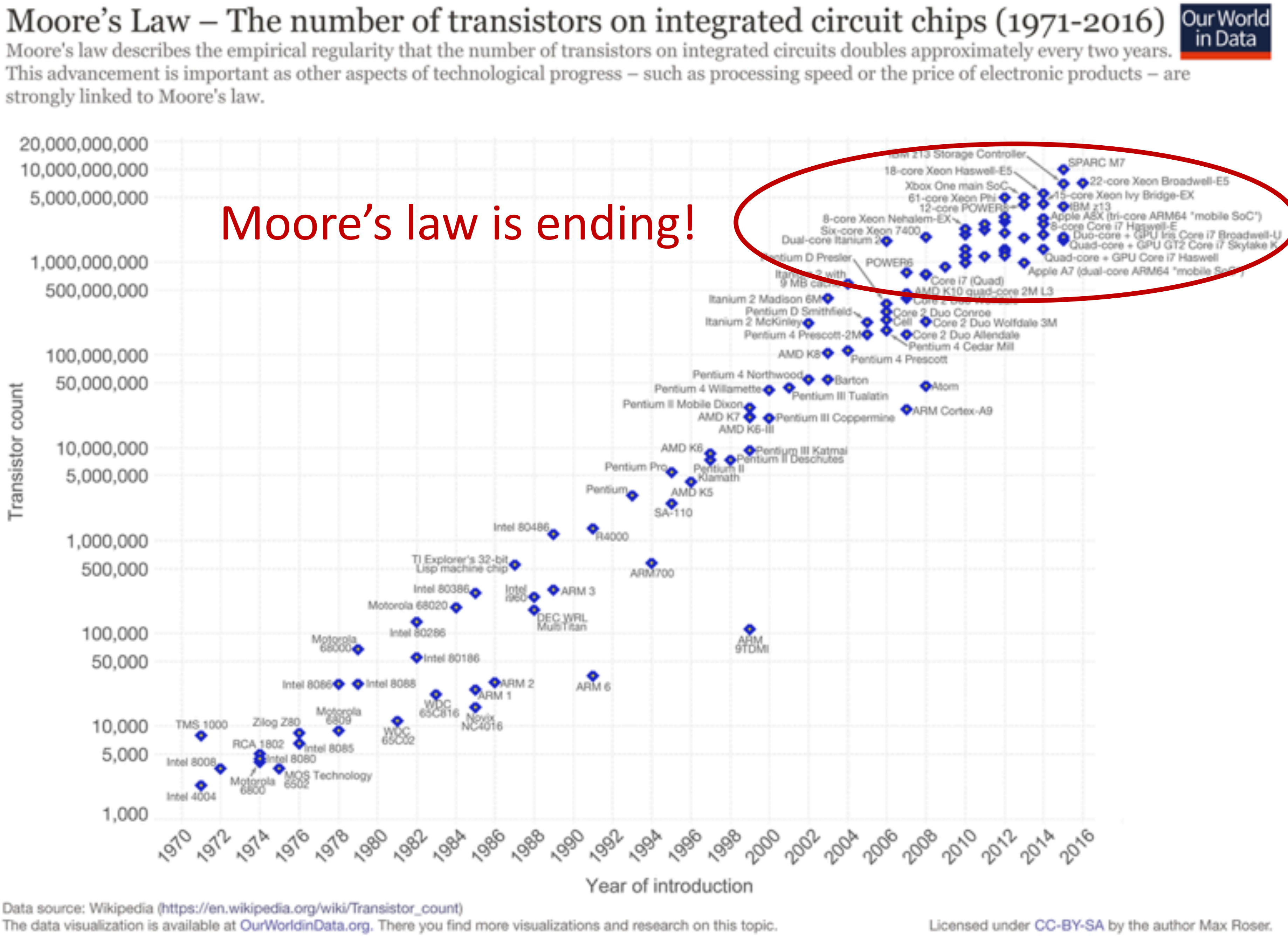


(b) Interleaved

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



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



CPU Trends – What Moore's Law Implies…

- The future
  - Same CPU speed
  - More cores (to scale-up)

- 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

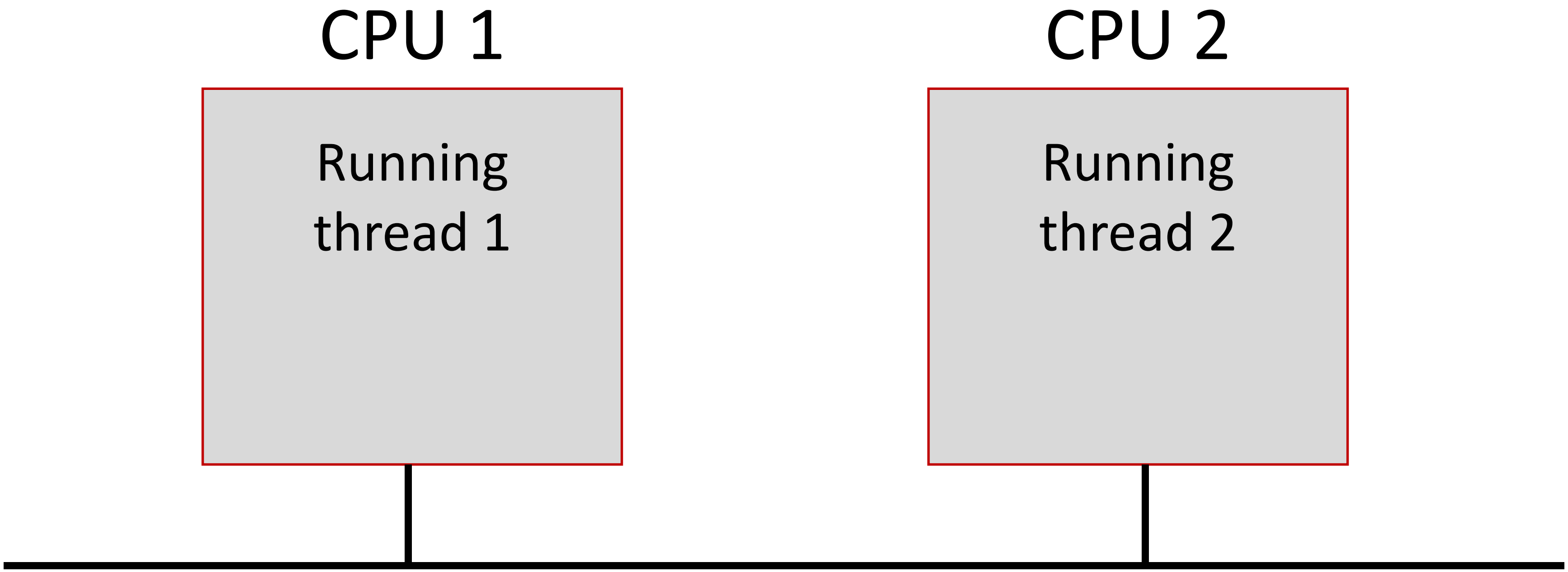
- A process, as defined so far, has only one thread of execution

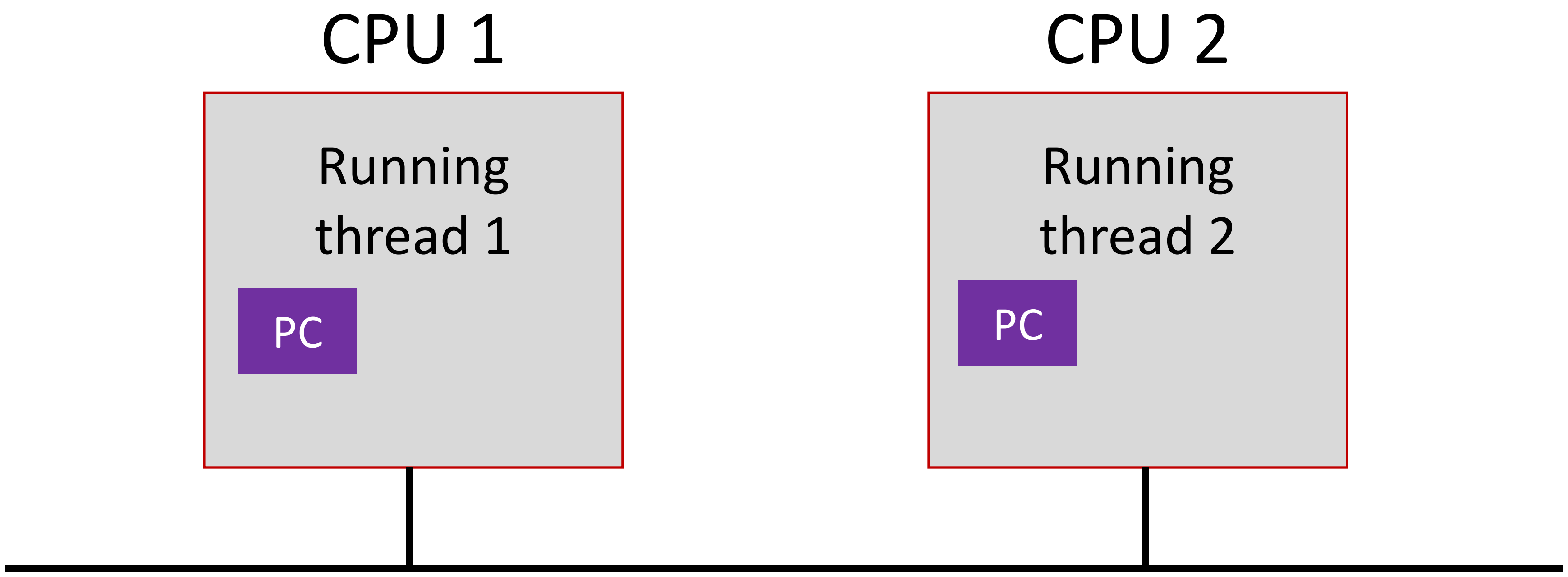
- Idea: Allow multiple threads of concurrently running execution within the same process environment, to a large degree independent of each other
  - Each thread may be executing different code at the same time

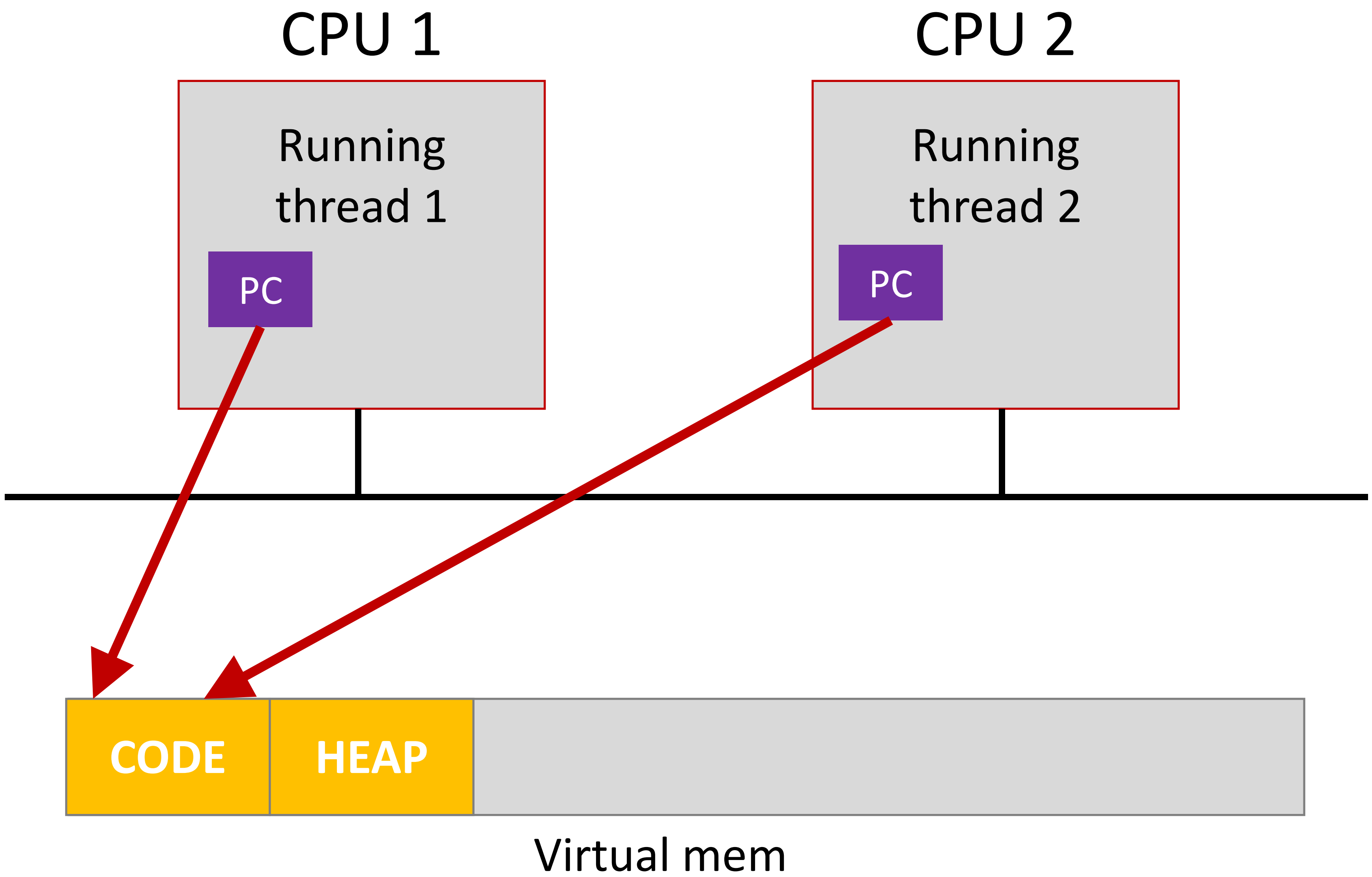
Process vs. Thread

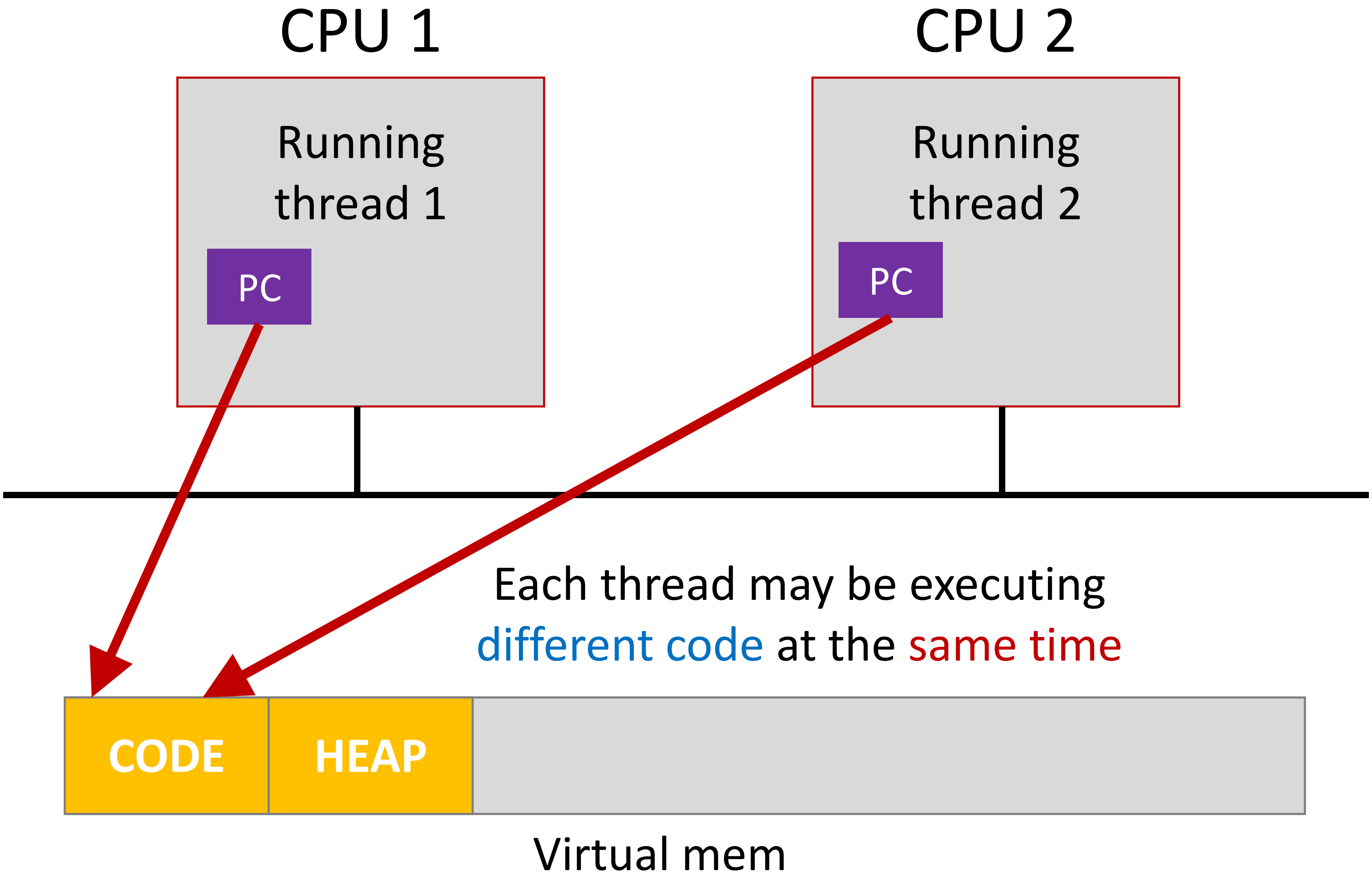
- Multiple threads within a process will share
  - The address space
  - Open files (file descriptors)
  - Other resources

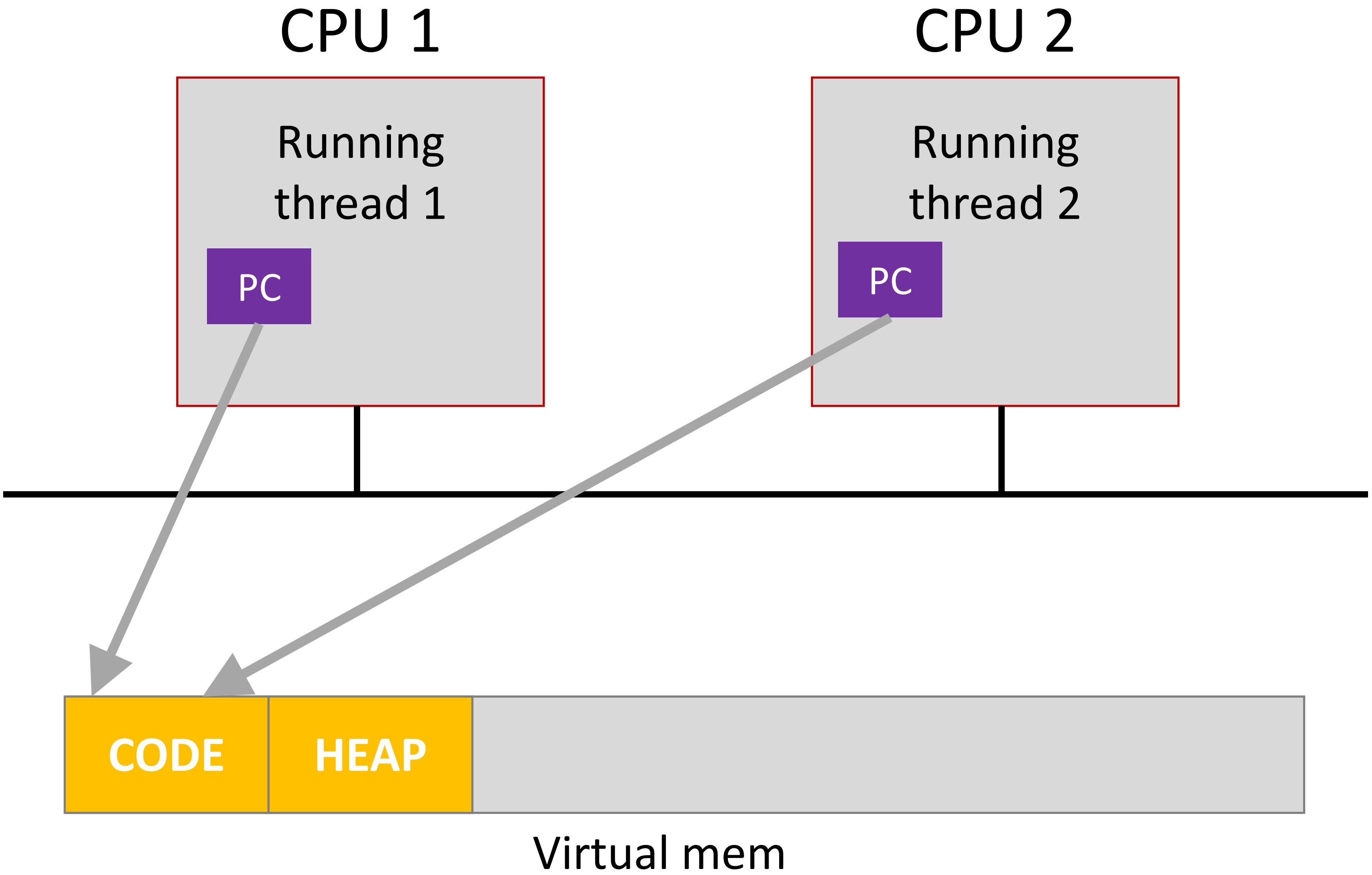
- Thread
  - Efficient and fast resource sharing
  - Efficient utilization of many CPU cores with only one process
  - Less context switching overheads

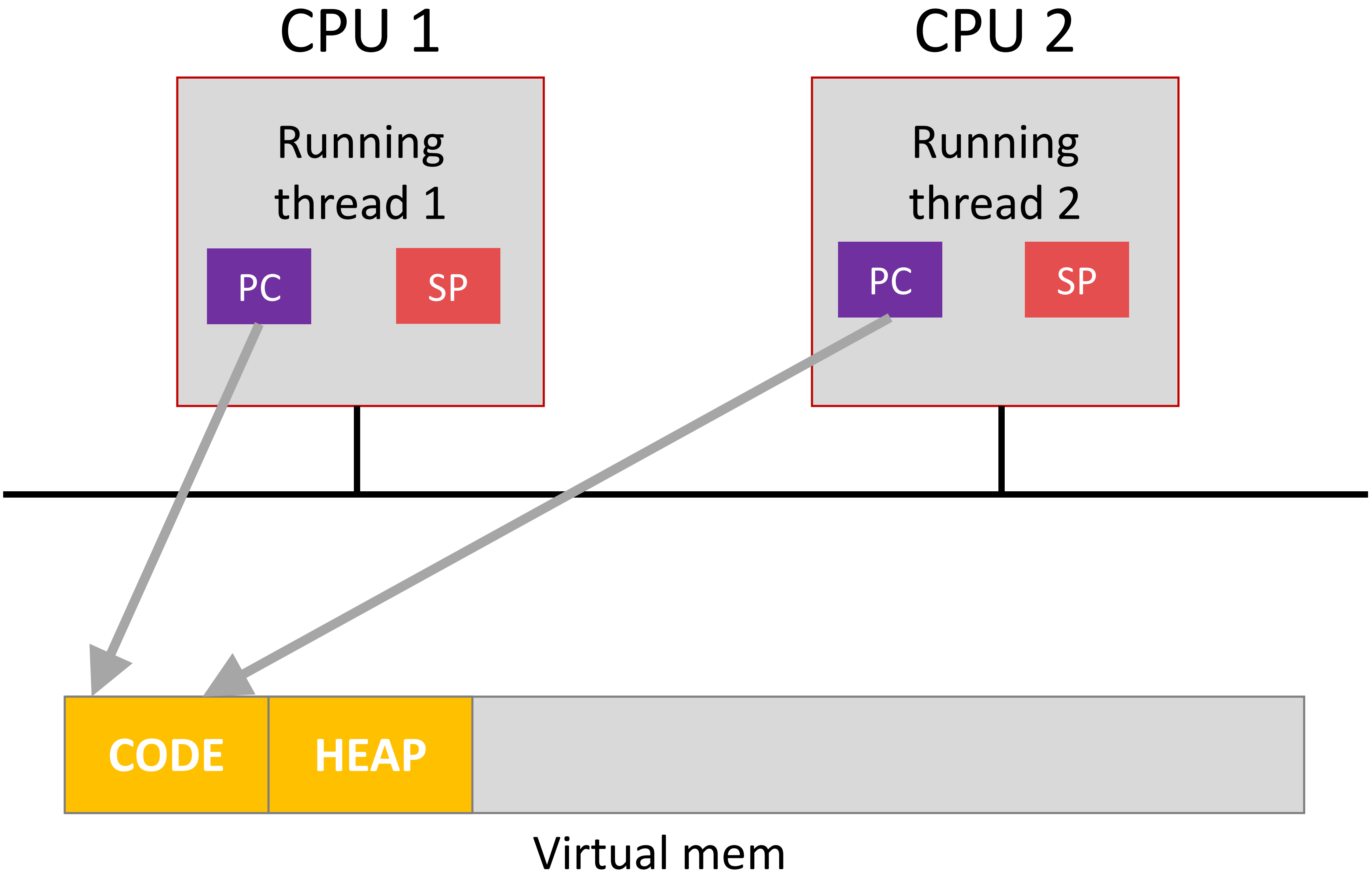

CPU 1
Running thread 1
CPU 2
Running thread 2

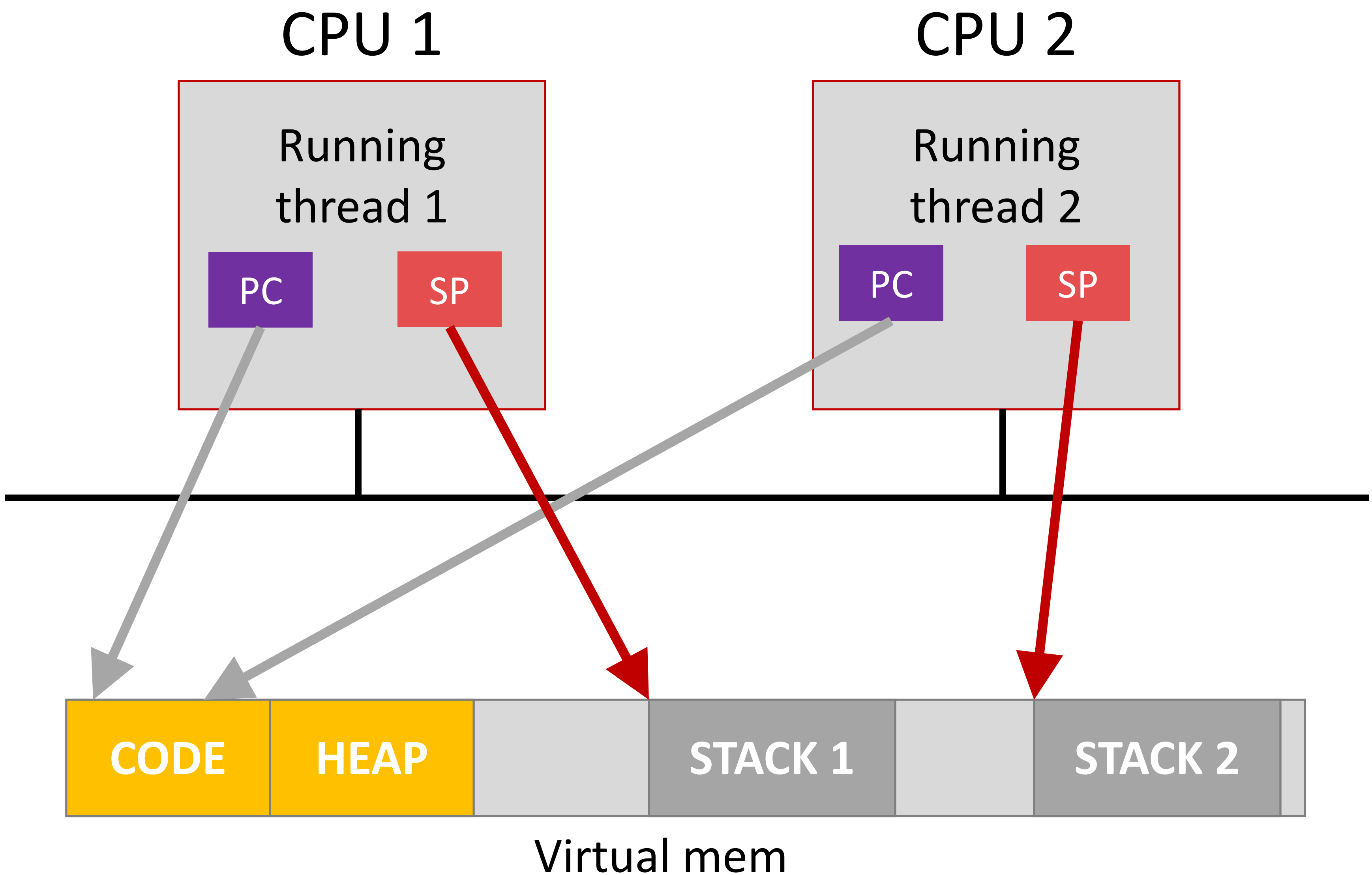

CPU 1
CPU 2
Running thread 1
PC
Running thread 2
PC


CPU 1
CPU 2
Running
thread 1
Running
thread 2
PC
PC
CODE
HEAP
Virtual mem

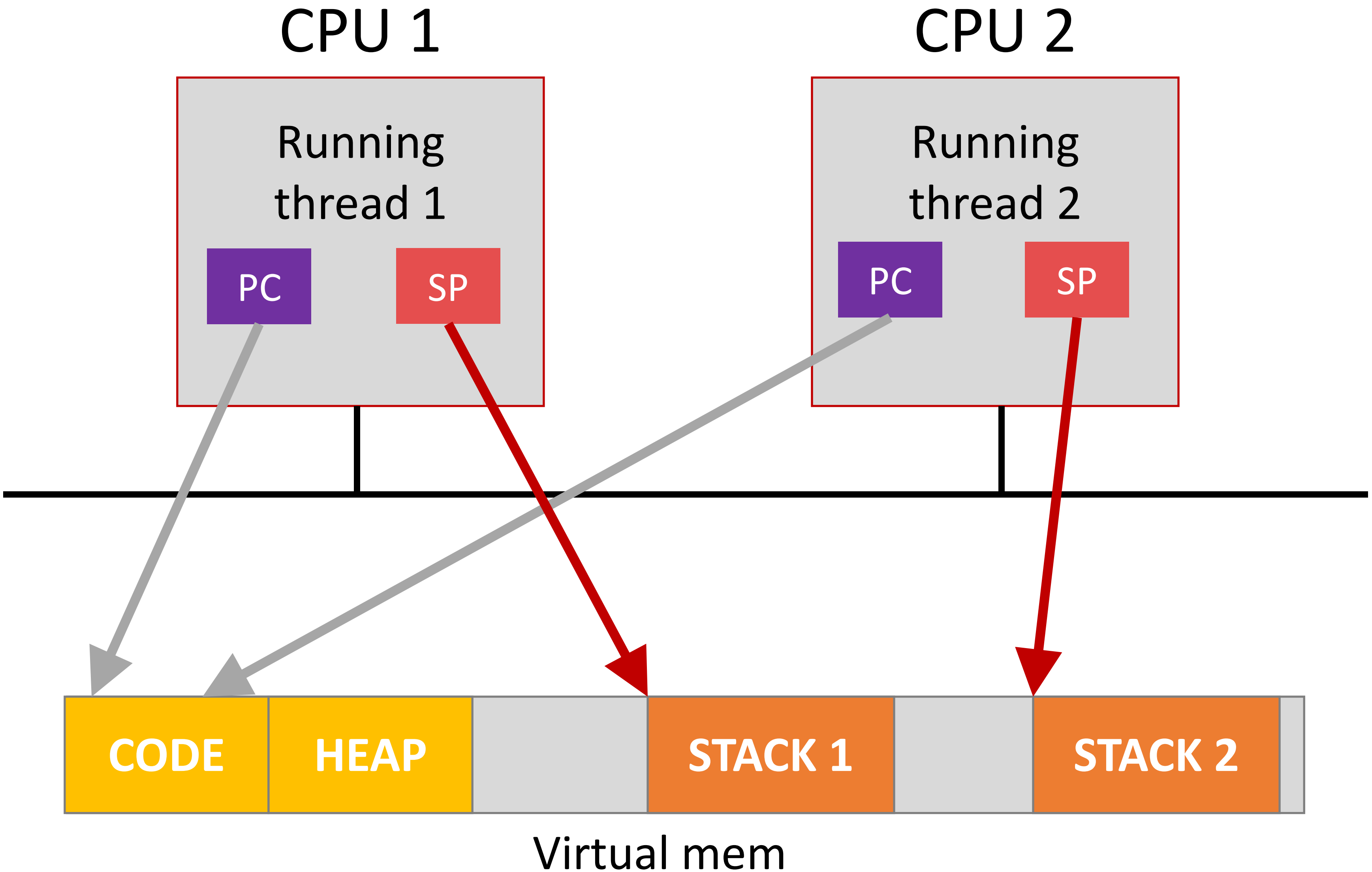

CPU 1
CPU 2
Running thread 1
Running thread 2
PC
PC
Each thread may be executing different code at the same time
CODE
HEAP
Virtual mem

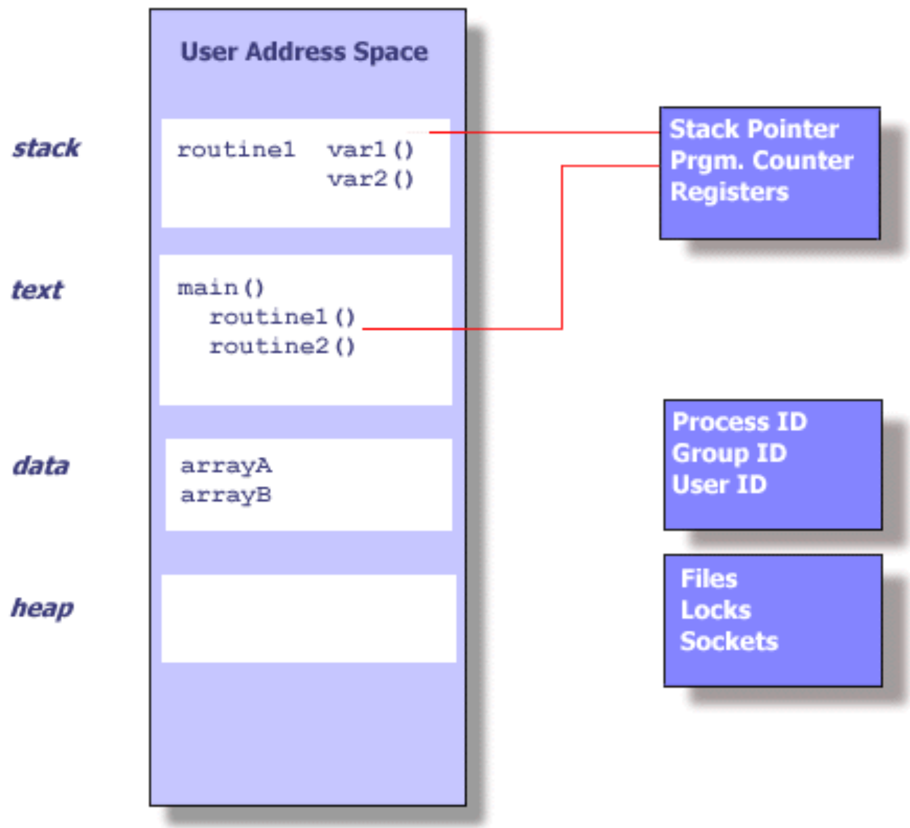

CPU 1
CPU 2
Running thread 1
Running thread 2
PC
PC
CODE
HEAP
Virtual mem


CPU 1
Running thread 1
PC
SP
CPU 2
Running thread 2
PC
SP
CODE
HEAP
Virtual mem

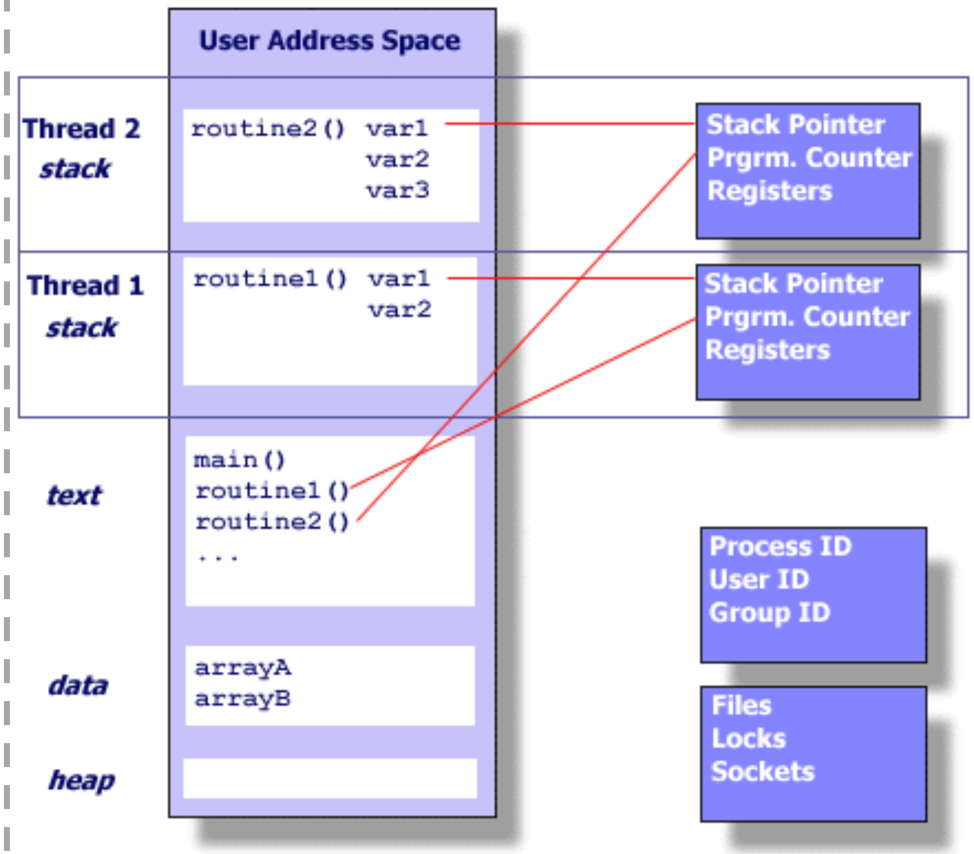

CPU 1
CPU 2
Running thread 1
Running thread 2
PC
SP
PC
SP
CODE
HEAP
STACK 1
STACK 2
Virtual mem

Thread executing different functions need different stacks





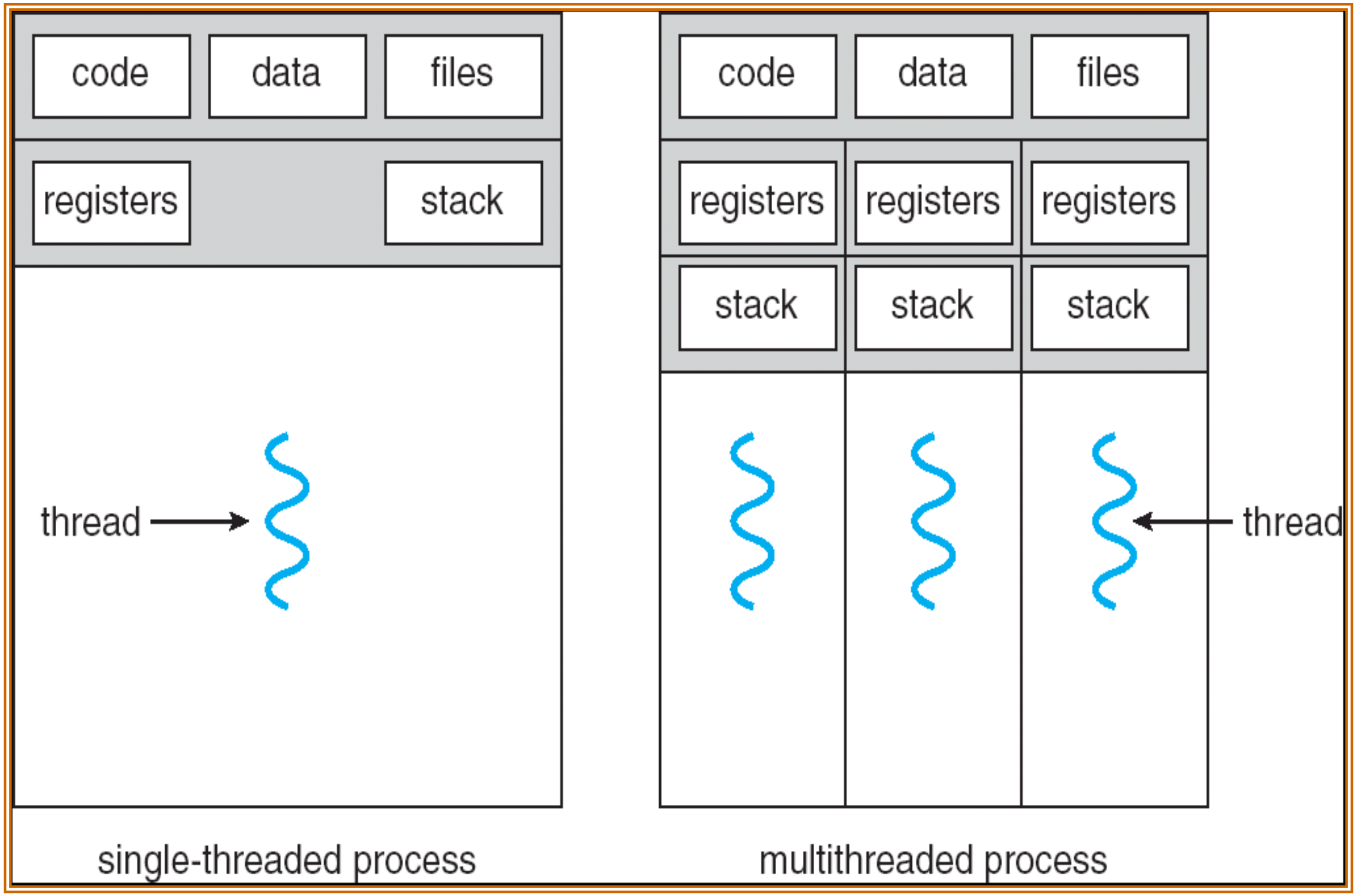
Linux process



Threads within a Linux process

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

Single- vs. Multi-threaded Process



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	Thread creation
`pthread_exit`	Terminate the calling thread	
`pthread_join`	Wait for a thread to terminate	
`pthread_mutex_init`	Create a new mutex	Thread lock
`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	Thread CV
`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	

Example of Using Pthread

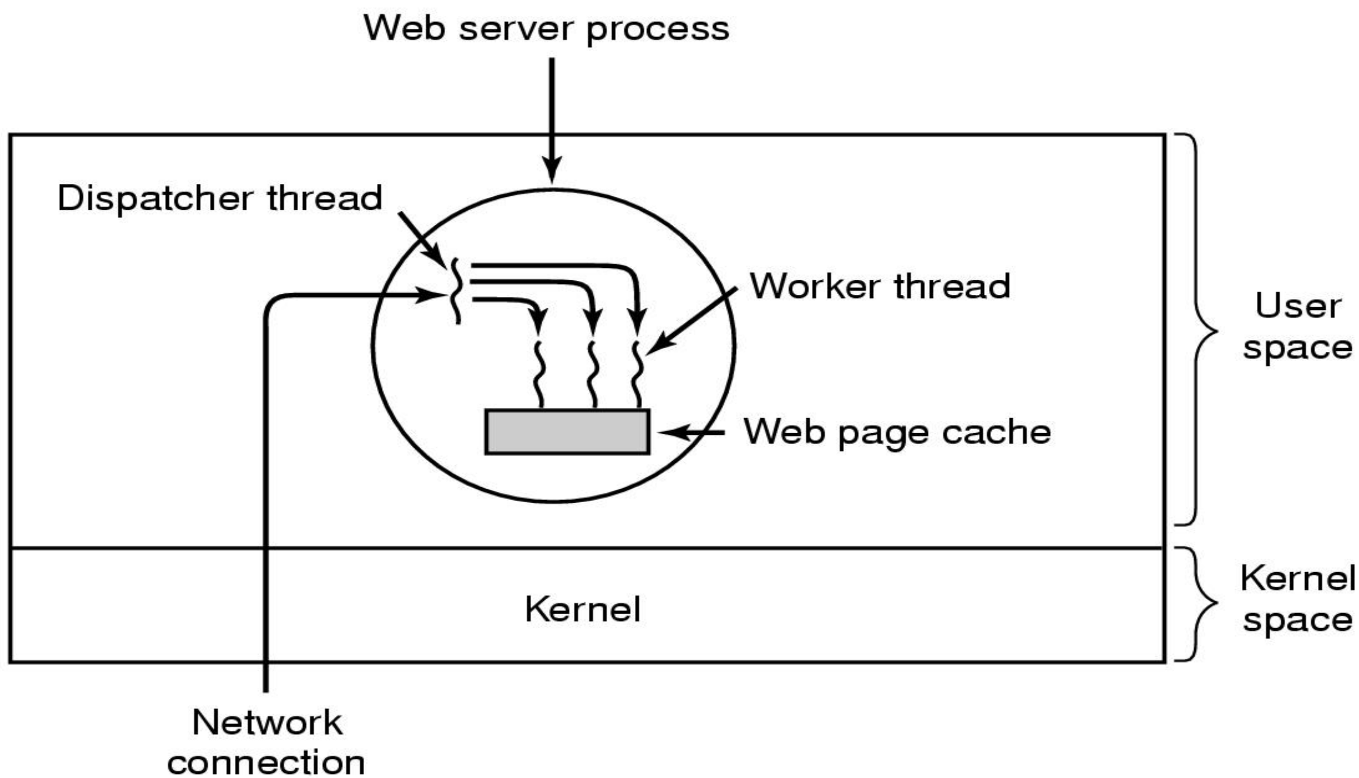
```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

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;
}
```

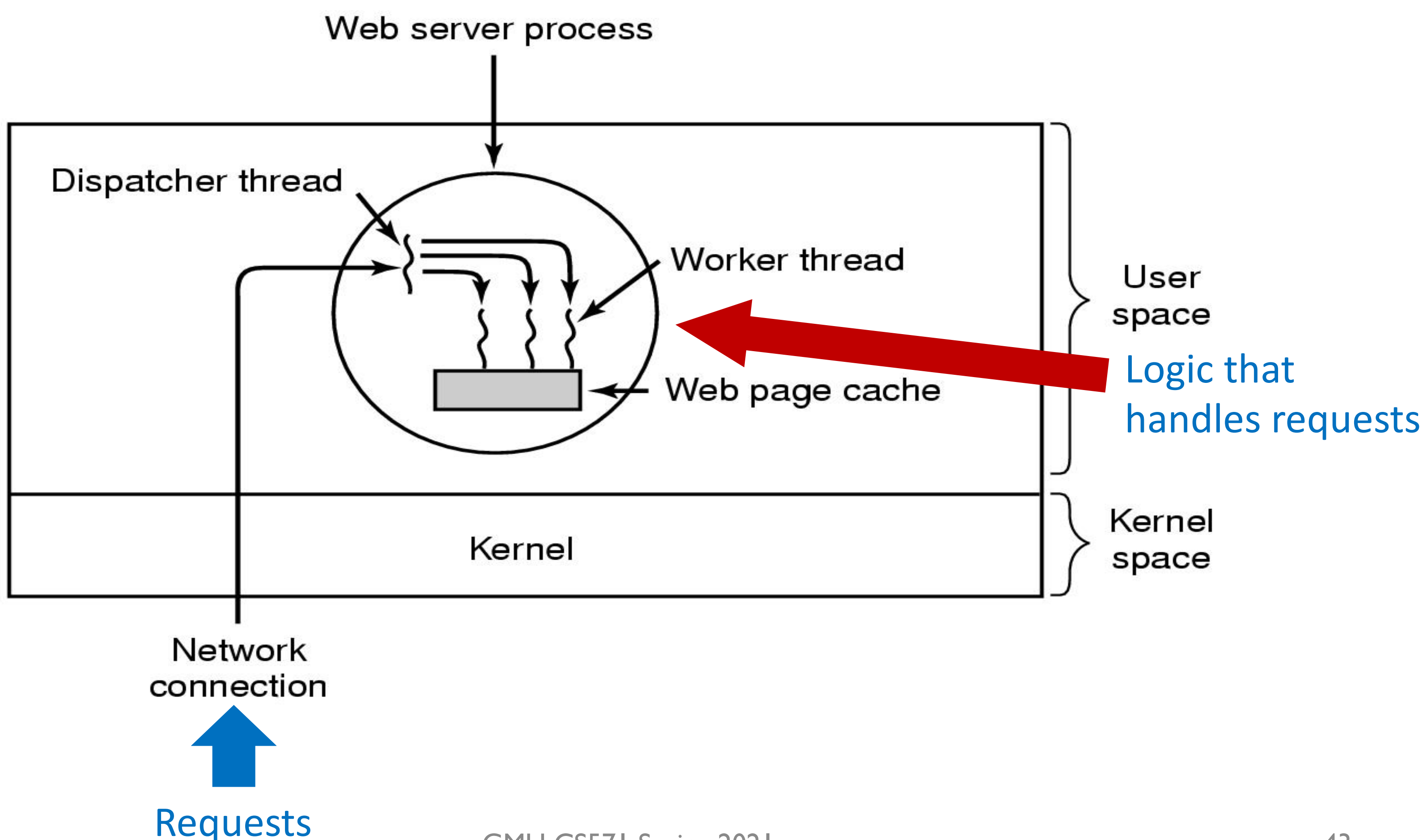
Example Multithreaded Applications

A multithreaded web server



Example Multithreaded Applications

A multithreaded web server



Code Sketch

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a) Dispatcher thread

```
while (TRUE) {
    wait_for_work(&buf);
    check_cache(&buf; &page);
    if (not_in_cache)
        read_from_disk(&buf,  &page);
    return_page(&page);
}
```

(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/

Memcached

- 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

Multithreading vs. Multi-processes

- Real-world debate
  - Multithreading vs. Multi-processes
  - Memcached vs. Redis
- Redis—A single-threaded memory-based data store (written in C)
  - https://redis.io/

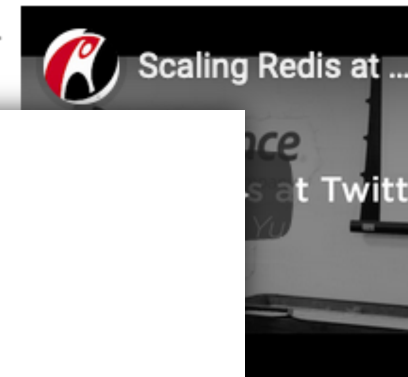
Wish List for Redis…

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



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.

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>
```

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
% objdump -d thread_rc
```

[Mac OS X]
[Linux]

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

```
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		50
	addl $0x1, %eax		51
Interrupt			
Save T1's state			
Restore T2's state			
		movl 0x2f8e, %eax	
		addl $0x1, %eax	
		movl %eax, 0x2f8e	

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

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			

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		**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		**51**

Time

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

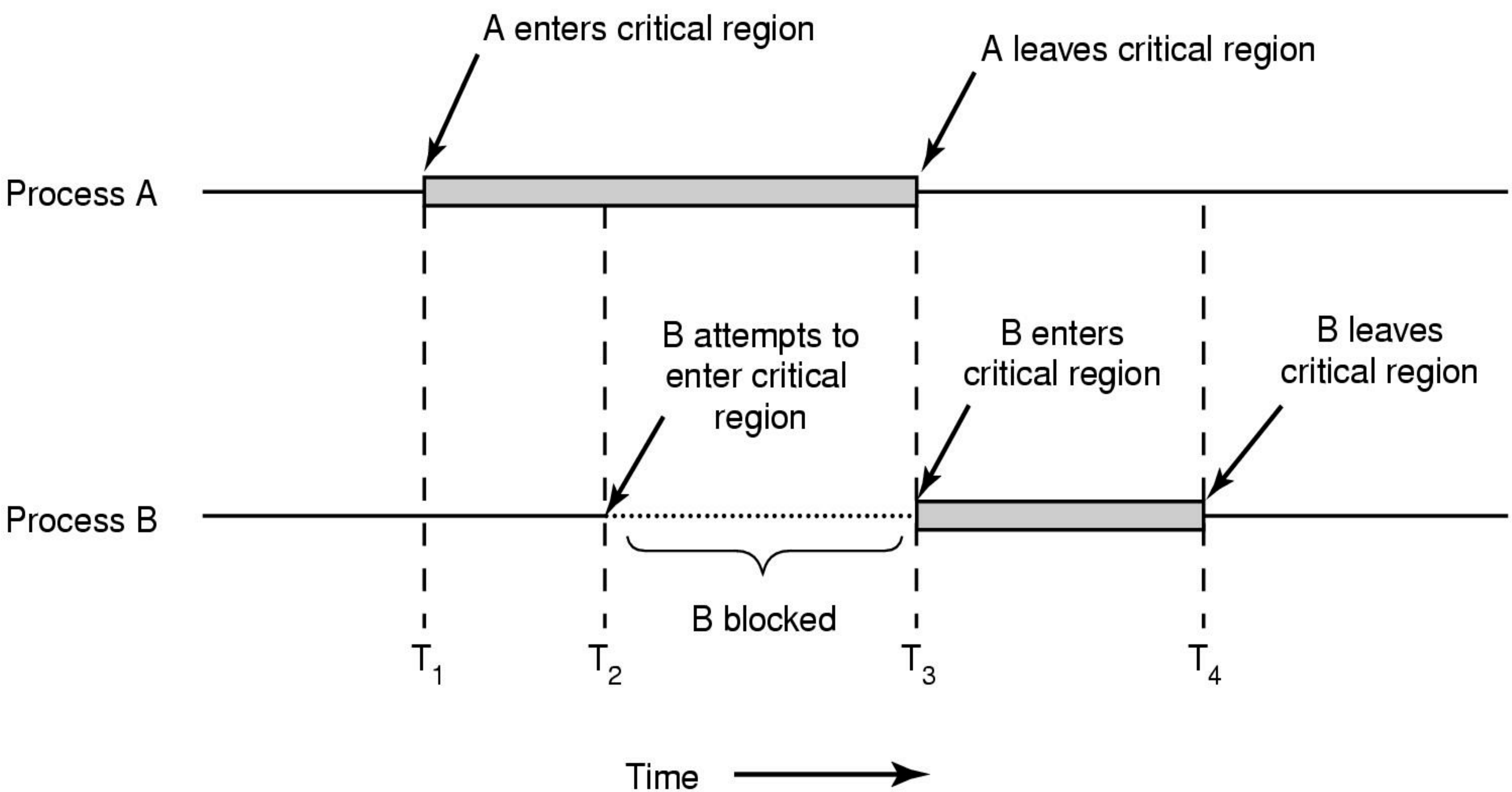
- Situations like this, where multiple threads are writing or reading some shared data and the final result depends on who runs precisely when, are called race conditions
  - A serious problem for any concurrent system using shared variables

- 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



Solving Critical-Section Problem

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

Mutual Exclusion:

No two threads may be simultaneously inside the same critical section

Bounded Waiting:

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

Progress:

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

Arbitrary Speed:

No assumption can be made about the relative speed of different threads (though all threads 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?

```c
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	

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!)	

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);
```

```
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)
- 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: Yield!

- 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;
}
```

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 */
```

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
  - Condition Variable
    - A CV-based version

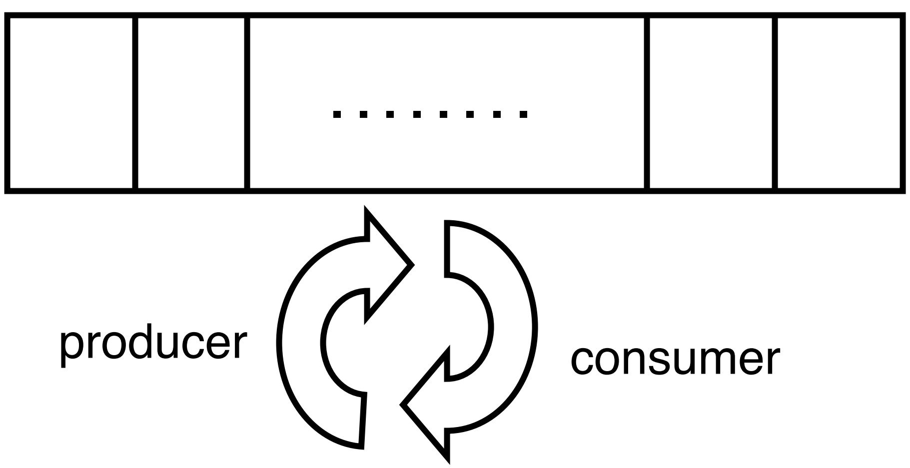
Today

- Readers-Writers Problem

- Dining-Philosophers Problem

Producer-Consumer Problem

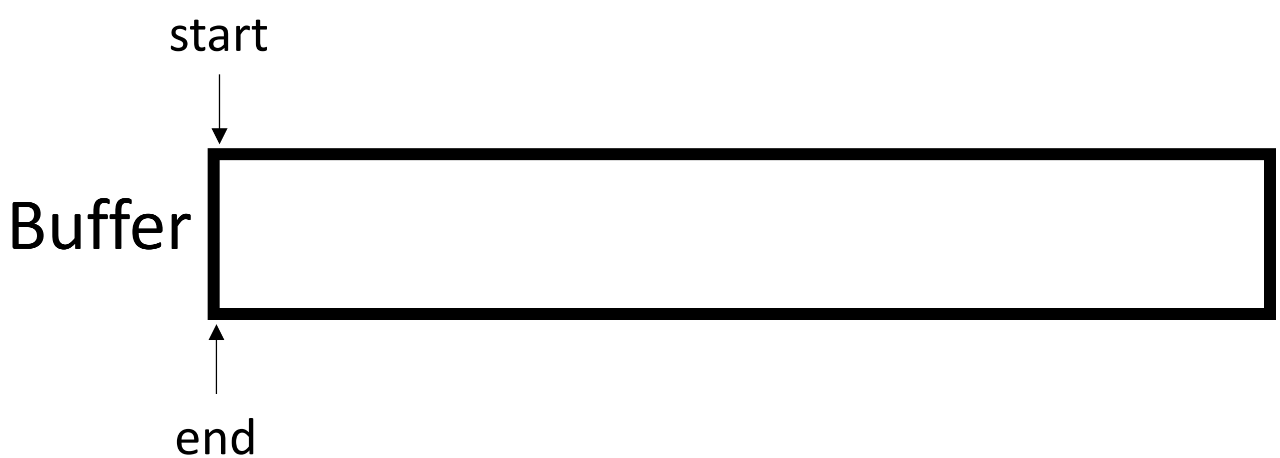
- The bounded-buffer producer-consumer problem assumes that there is a buffer of size N
- The producer process puts items to the buffer area
- The consumer process consumes items from the buffer
- The producer and the consumer execute concurrently



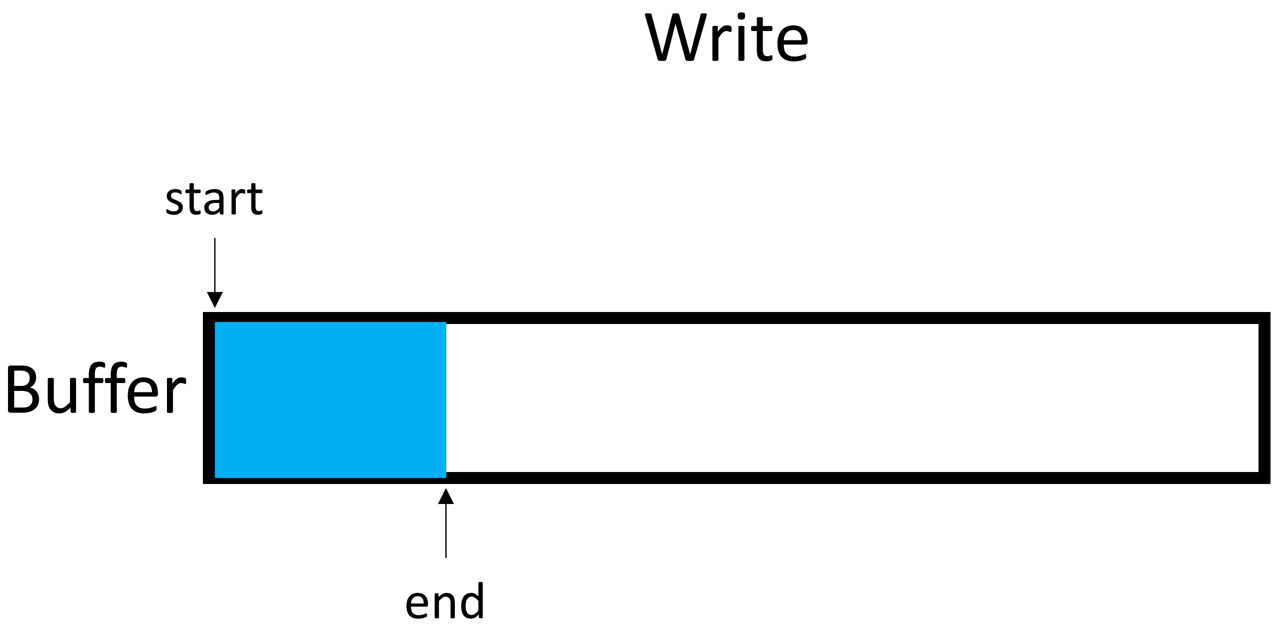
Example: Unix Pipes

- 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

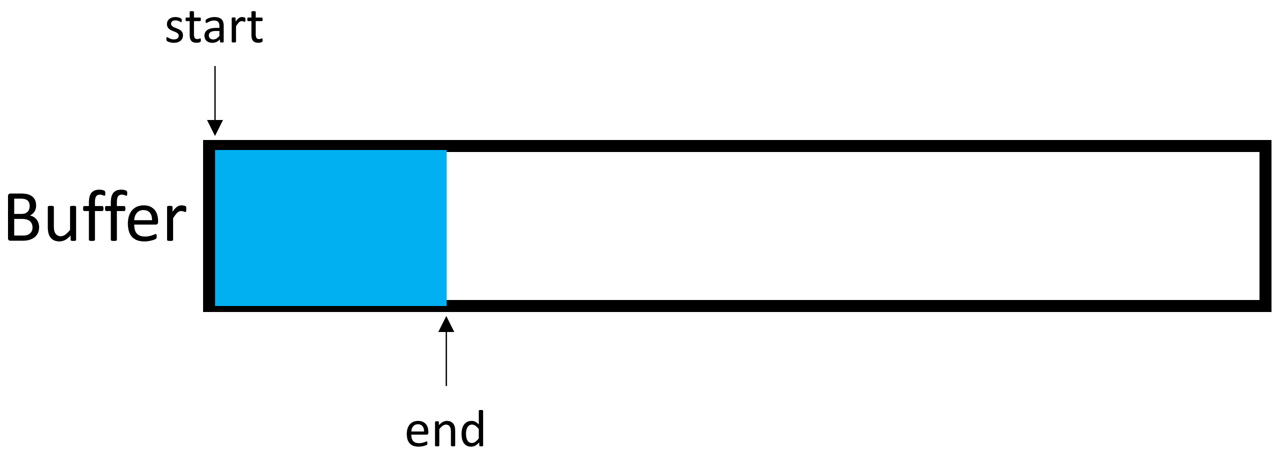
Example: Unix Pipes



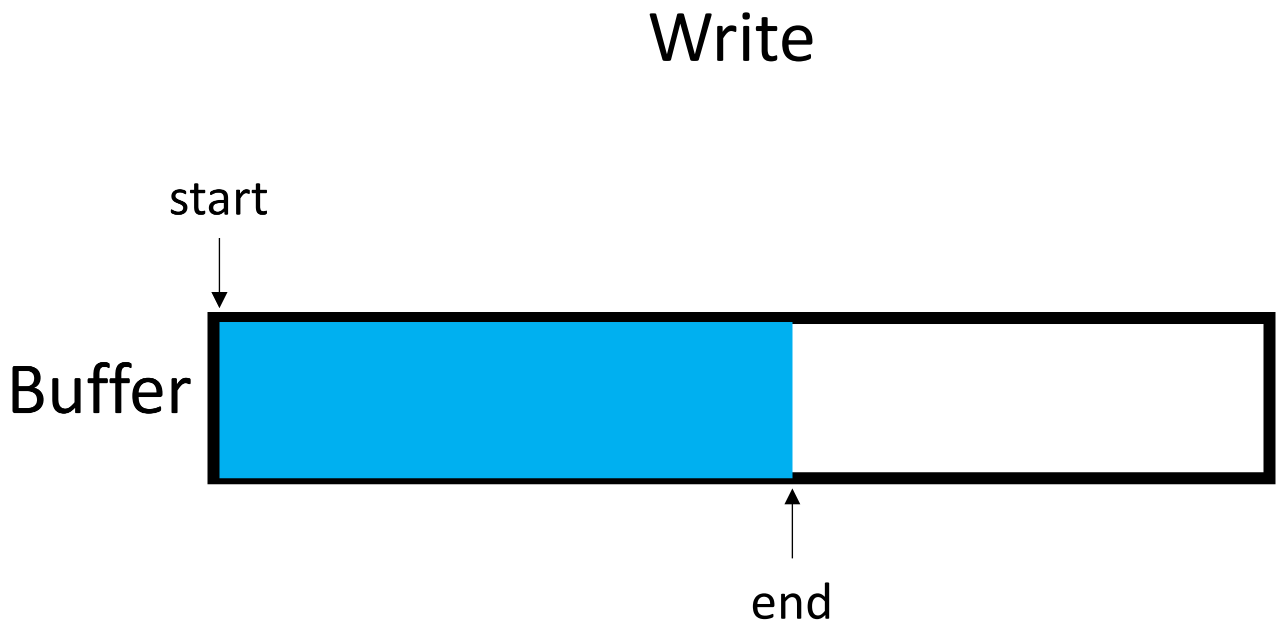
Example: Unix Pipes



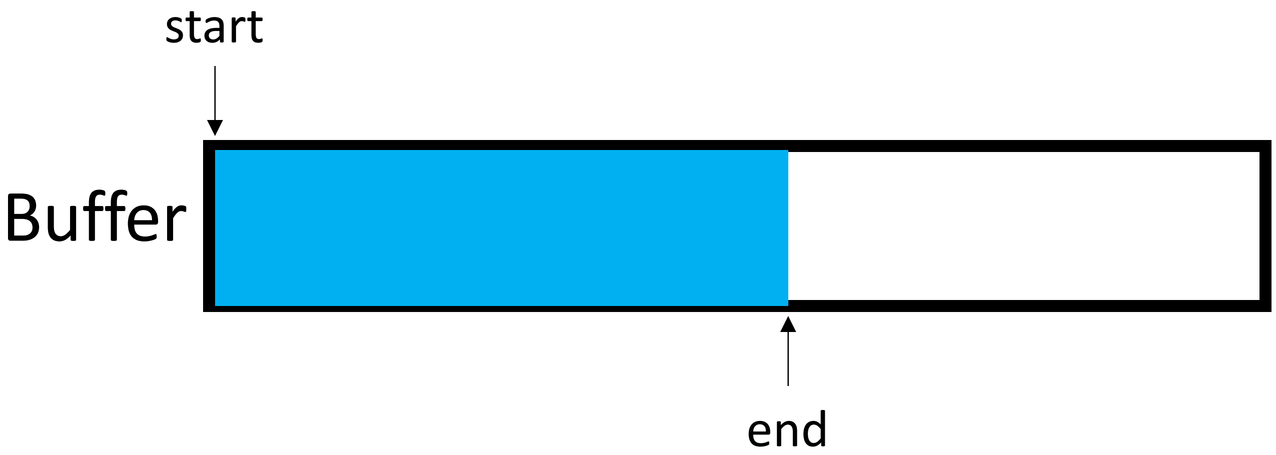
Example: Unix Pipes



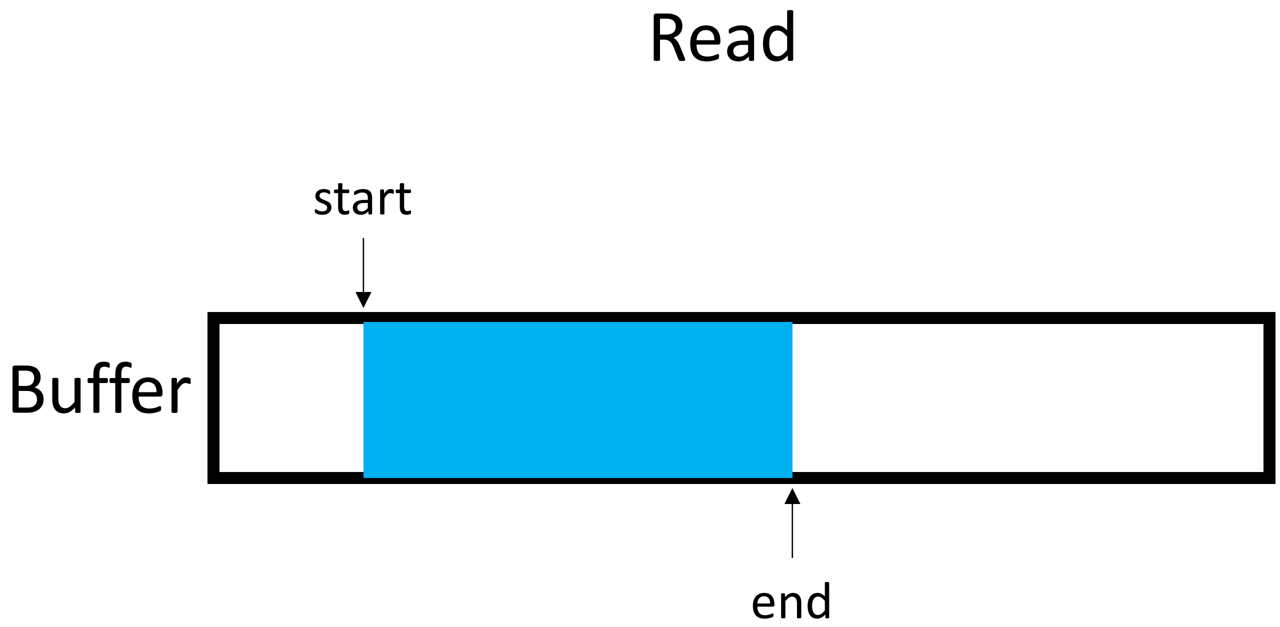
Example: Unix Pipes



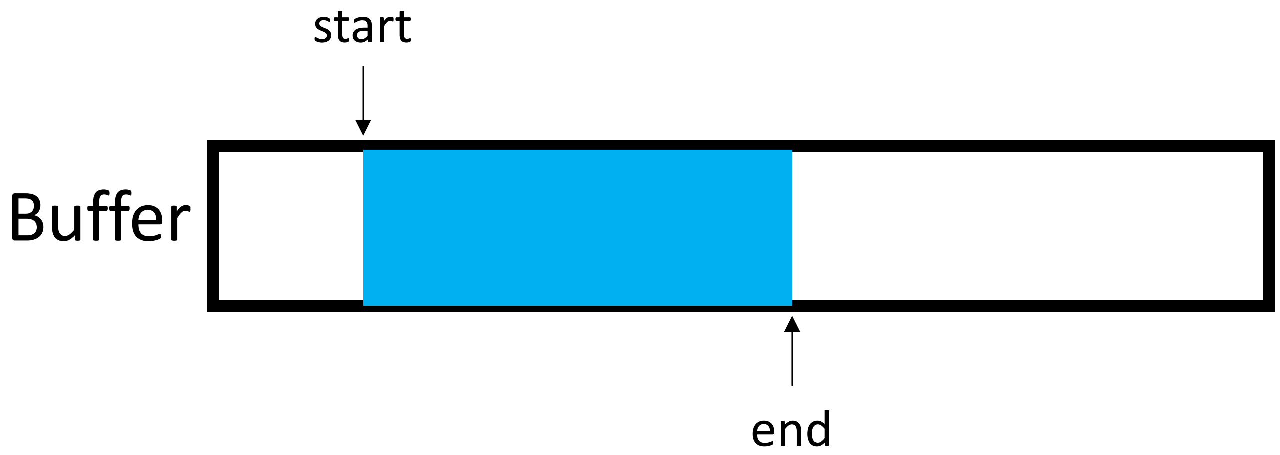
Example: Unix Pipes



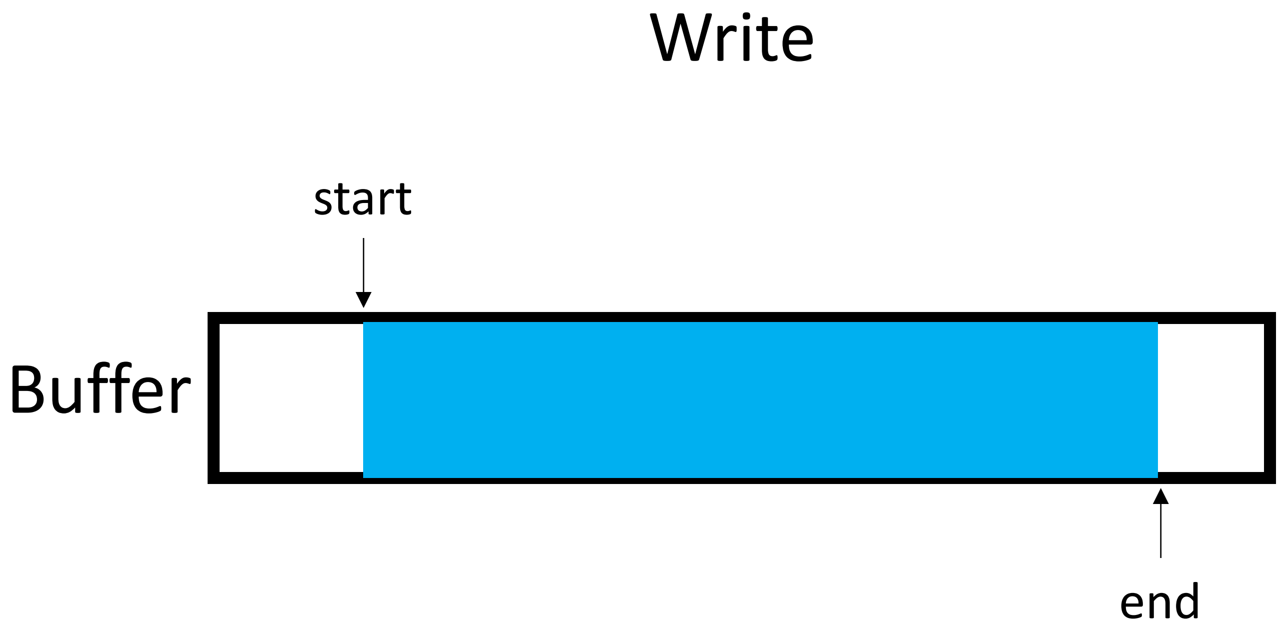
Example: Unix Pipes



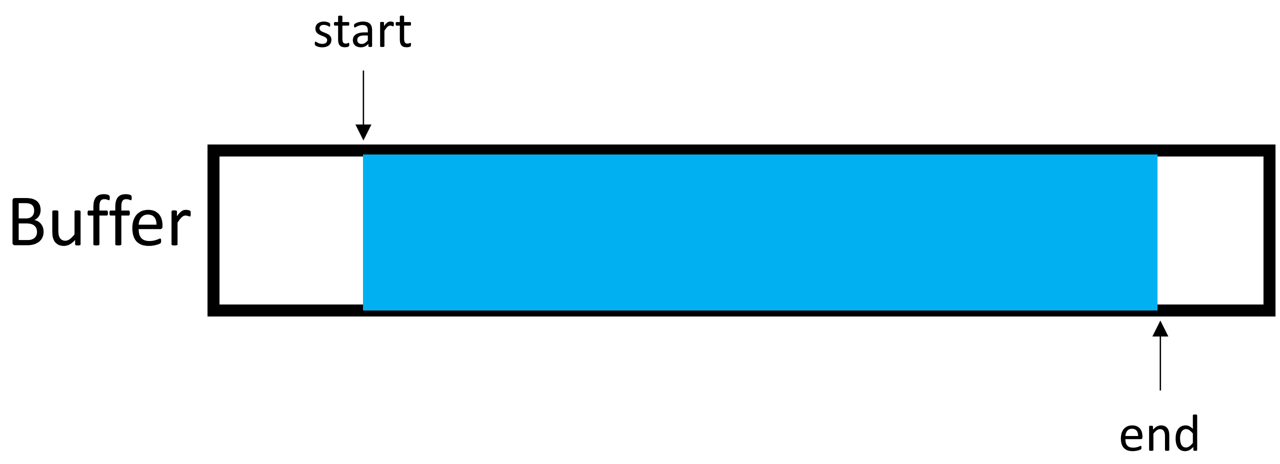
Example: Unix Pipes



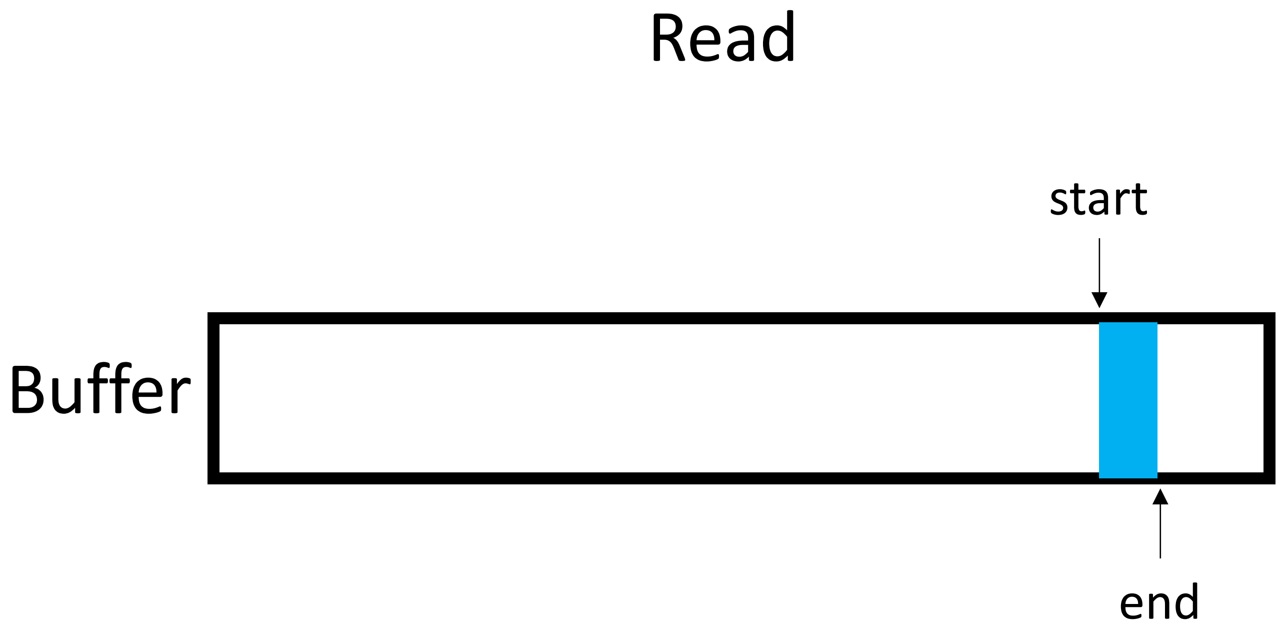
Example: Unix Pipes



Example: Unix Pipes

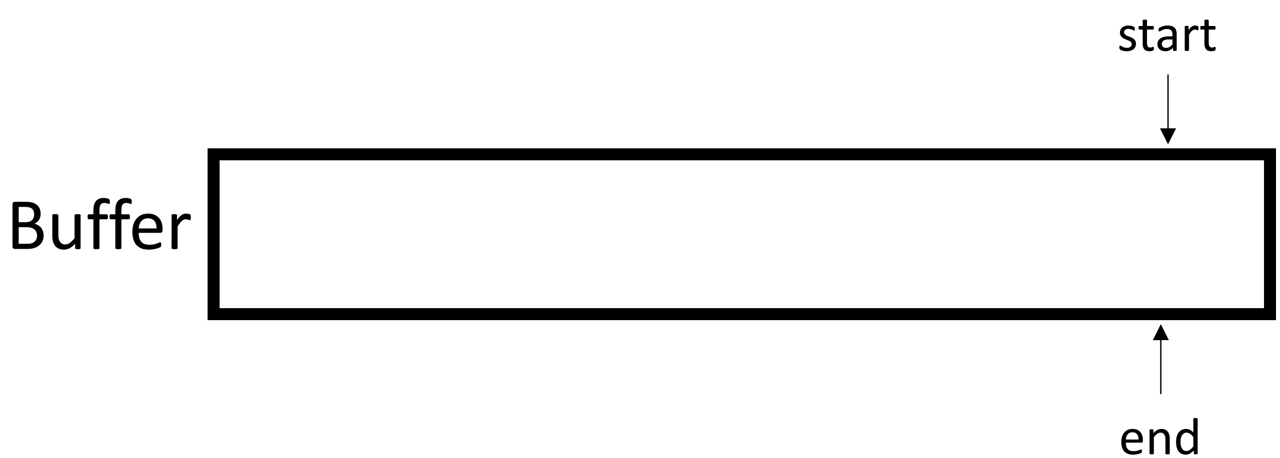


Example: Unix Pipes

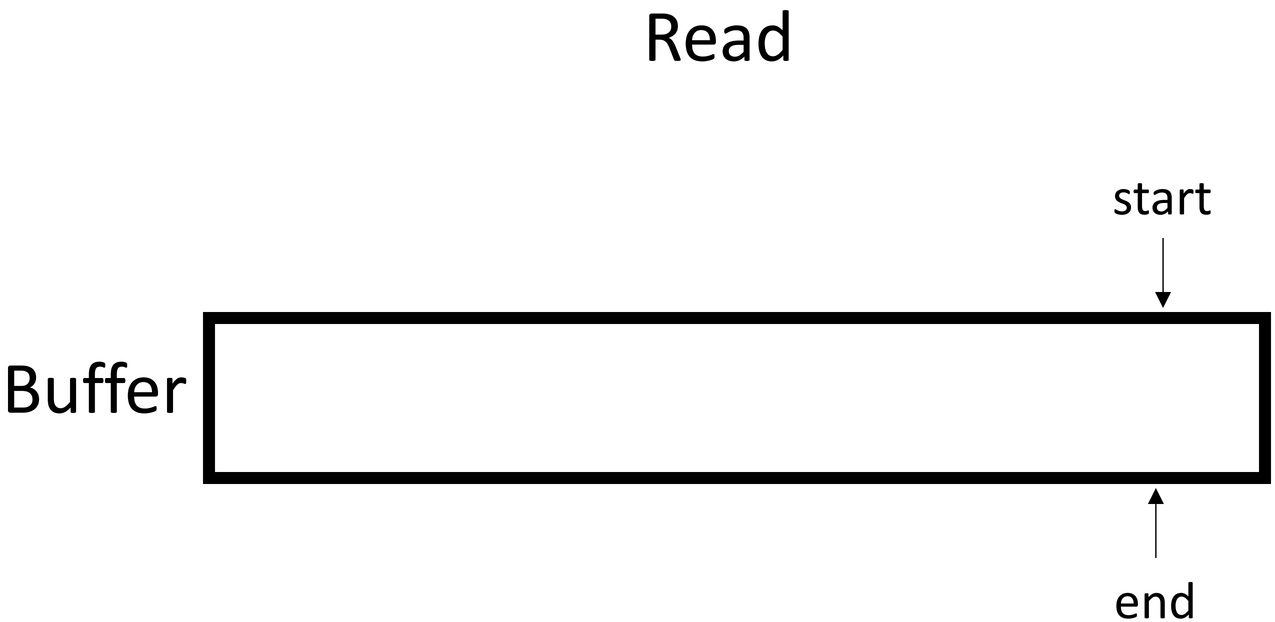


Example: Unix Pipes

Read

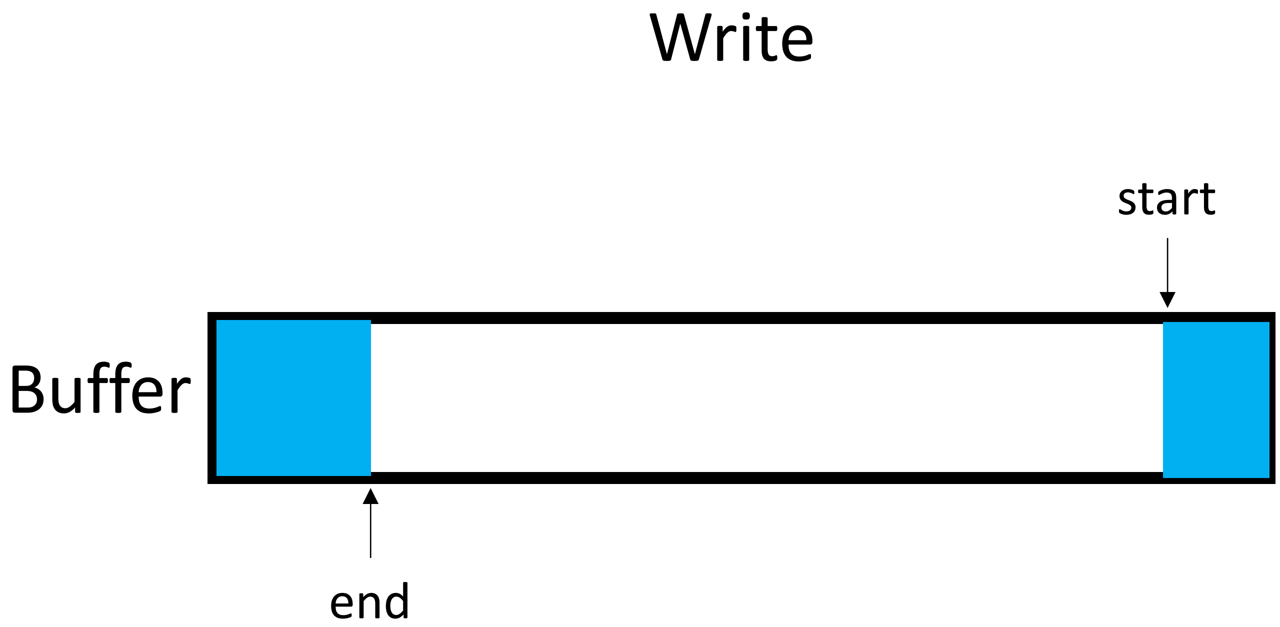


Example: Unix Pipes

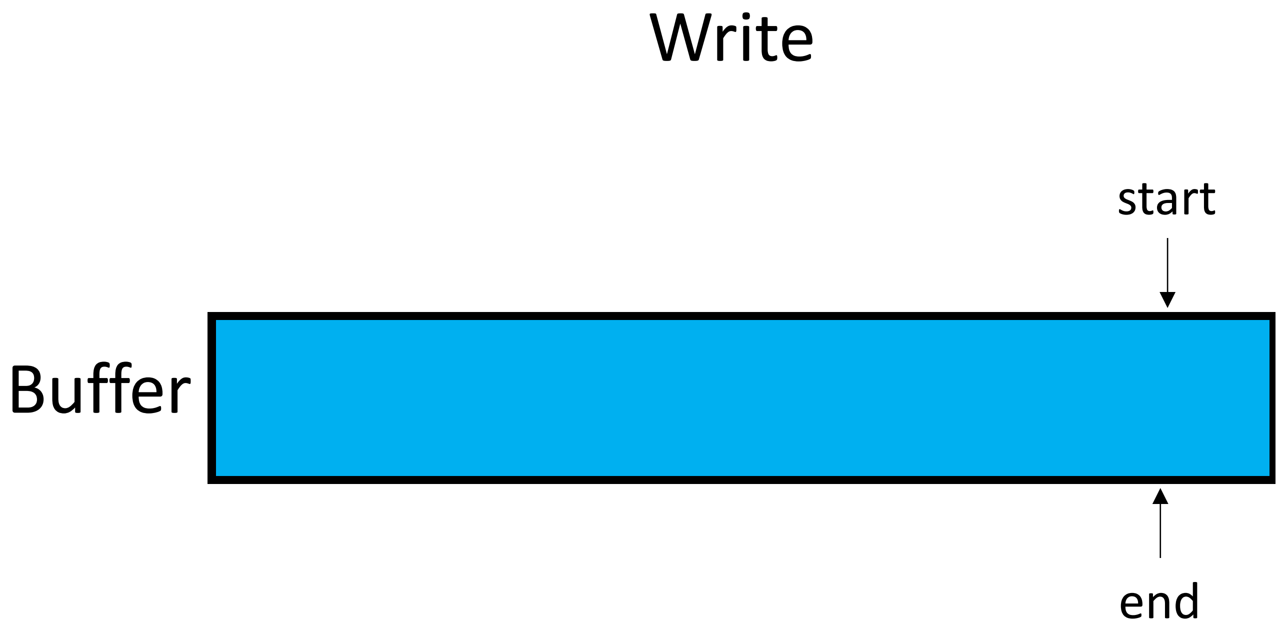


Note: reader must **wait**

Example: Unix Pipes

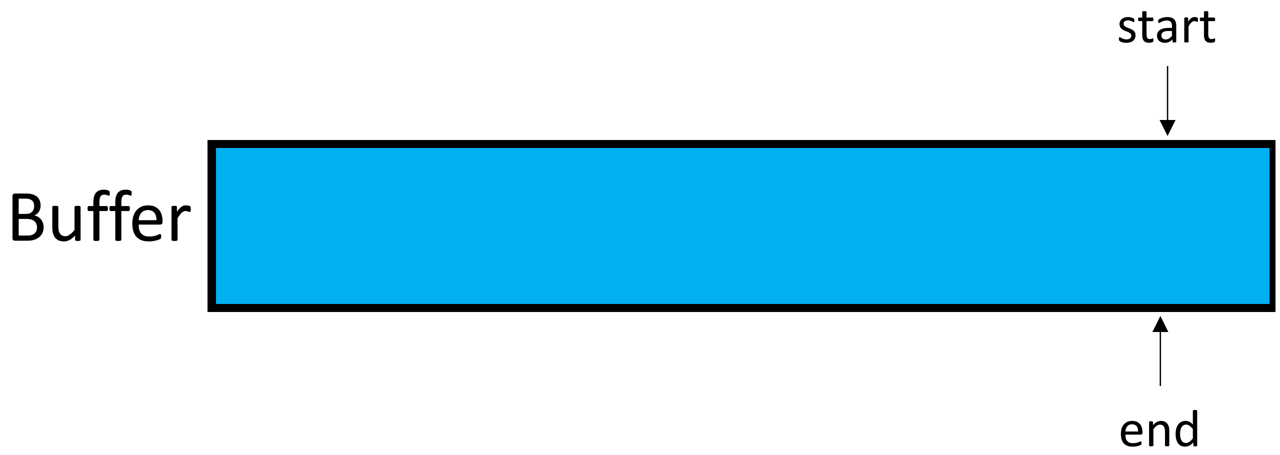


Example: Unix Pipes



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

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

```
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?

```
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

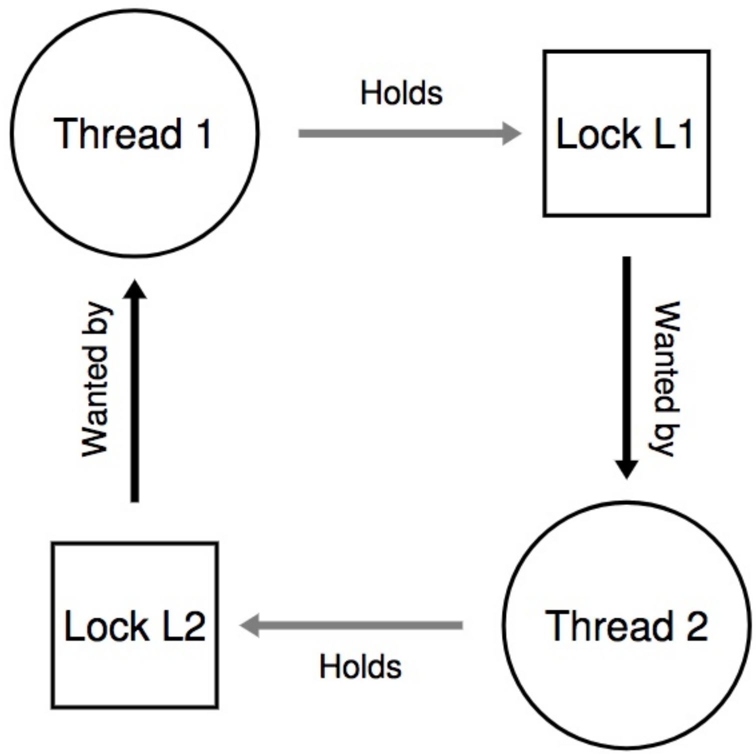
```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&full);
        int tmp = get();
        sem_post(&empty);
        sem_post(&mutex);
        printf("%d\n", tmp);
    }
}
```

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, lines p1.5–p2.5)

Mutex wraps just around critical section! (consumer, lines c1.5–c2.5)

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