slides kindly provided by:

Tam Vu, Ph.D
Professor of Computer Science
Director, Mobile & Networked Systems Lab
Department of Computer Science
University of Colorado Boulder

CSCI 3753 Operating Systems Summer 2020

Christopher Godley

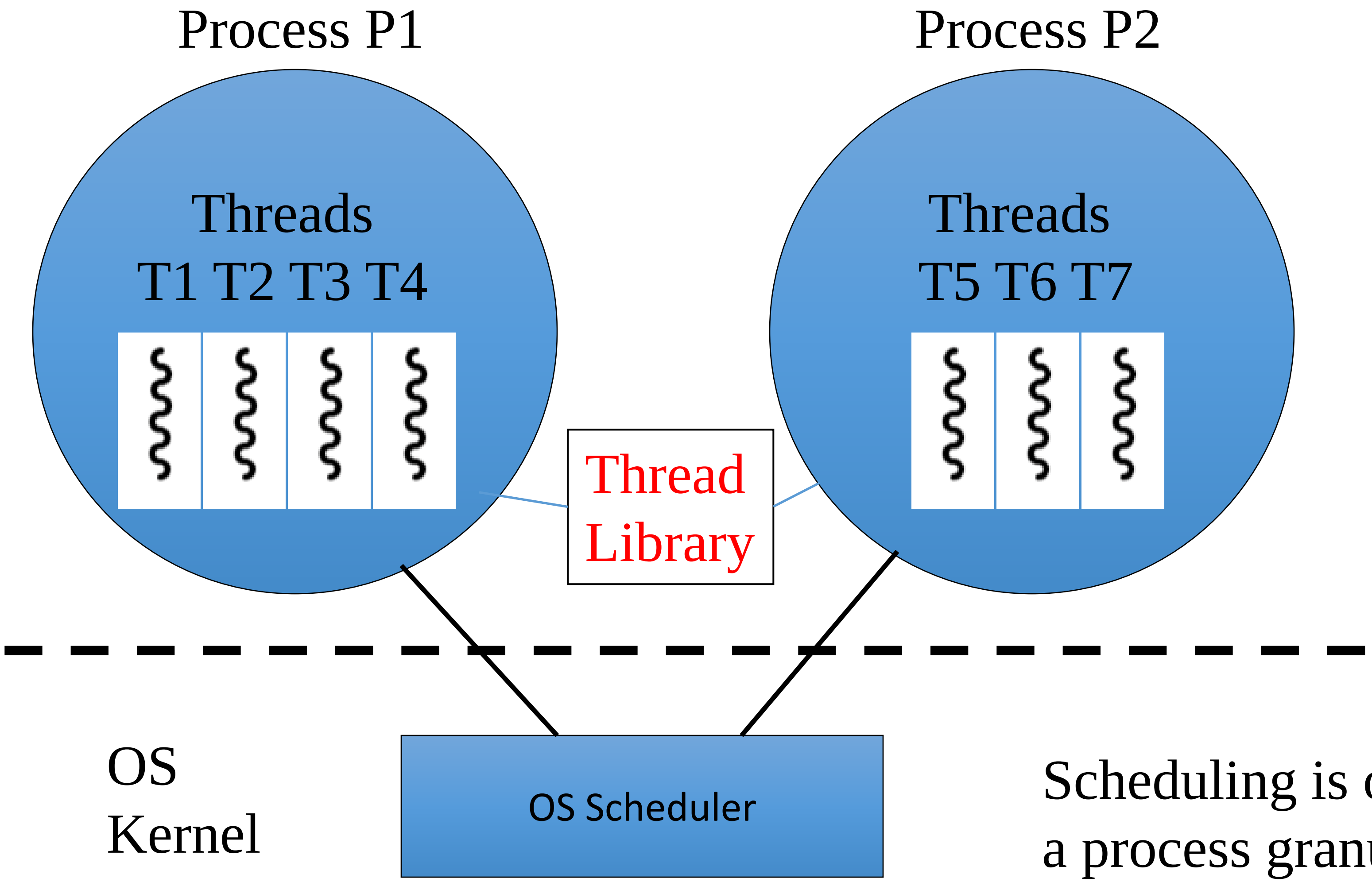
PhD Student

Department of Computer Science

University of Colorado Boulder

Threads

User-Space Threads



Scheduling is on
a process granularity

User-Space Threads

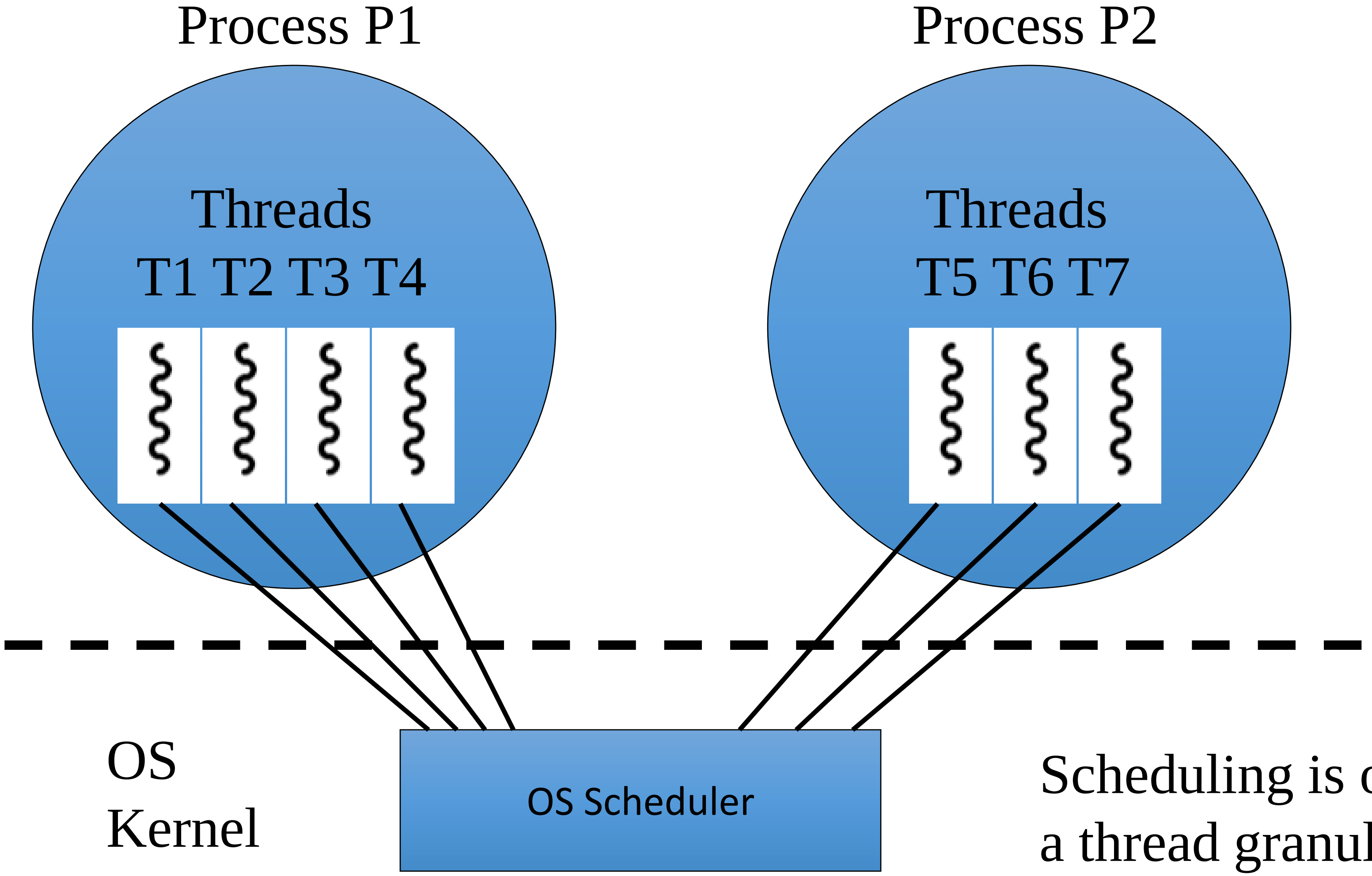
- *User space threads* are usually cooperatively multitasked, i.e. user threads within a process voluntarily give up the CPU to each other
  - threads will synchronize with each other via the user space threading package or library
  - Thread library: provides interface to create, delete threads in the same process

- OS is unaware of user-space threads – only sees user-space processes
  - If one user space thread blocks, the entire process blocks in a many-to-one scenario (see text)

- *pthreads* is a POSIX threading API
  - implementations of pthreads API differ underneath the API; could be user space threads; there is also pthreads support for kernel threads as well

- User space thread also called a *fiber* (cooperatively multitasked)
- Kernel space thread also called a *lightweight process* (preemptively multitasked)

Kernel Threads



Process P1

Threads
T1 T2 T3 T4

Process P2

Threads
T5 T6 T7

OS
Kernel

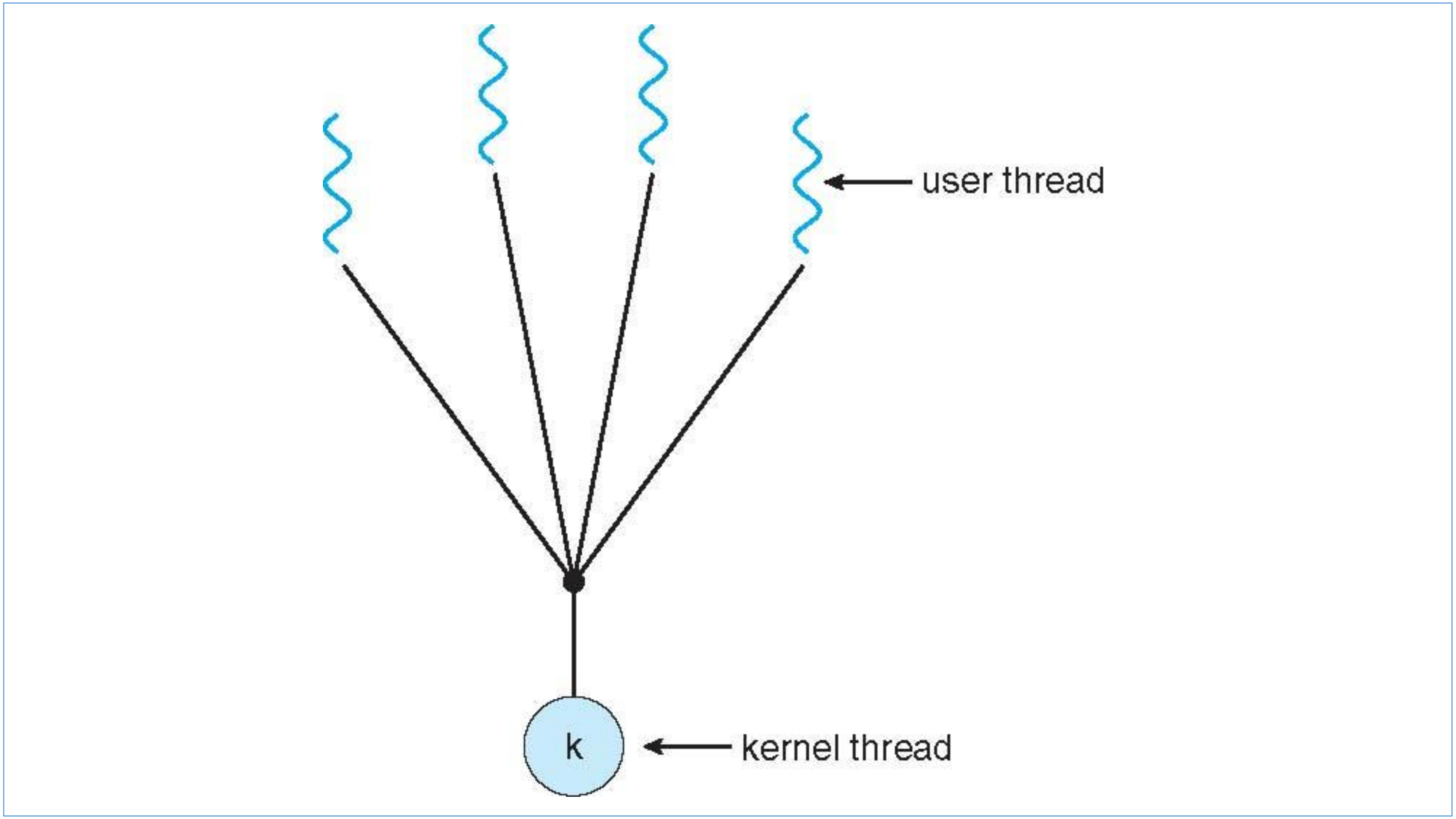
OS Scheduler

Scheduling is on
a thread granularity

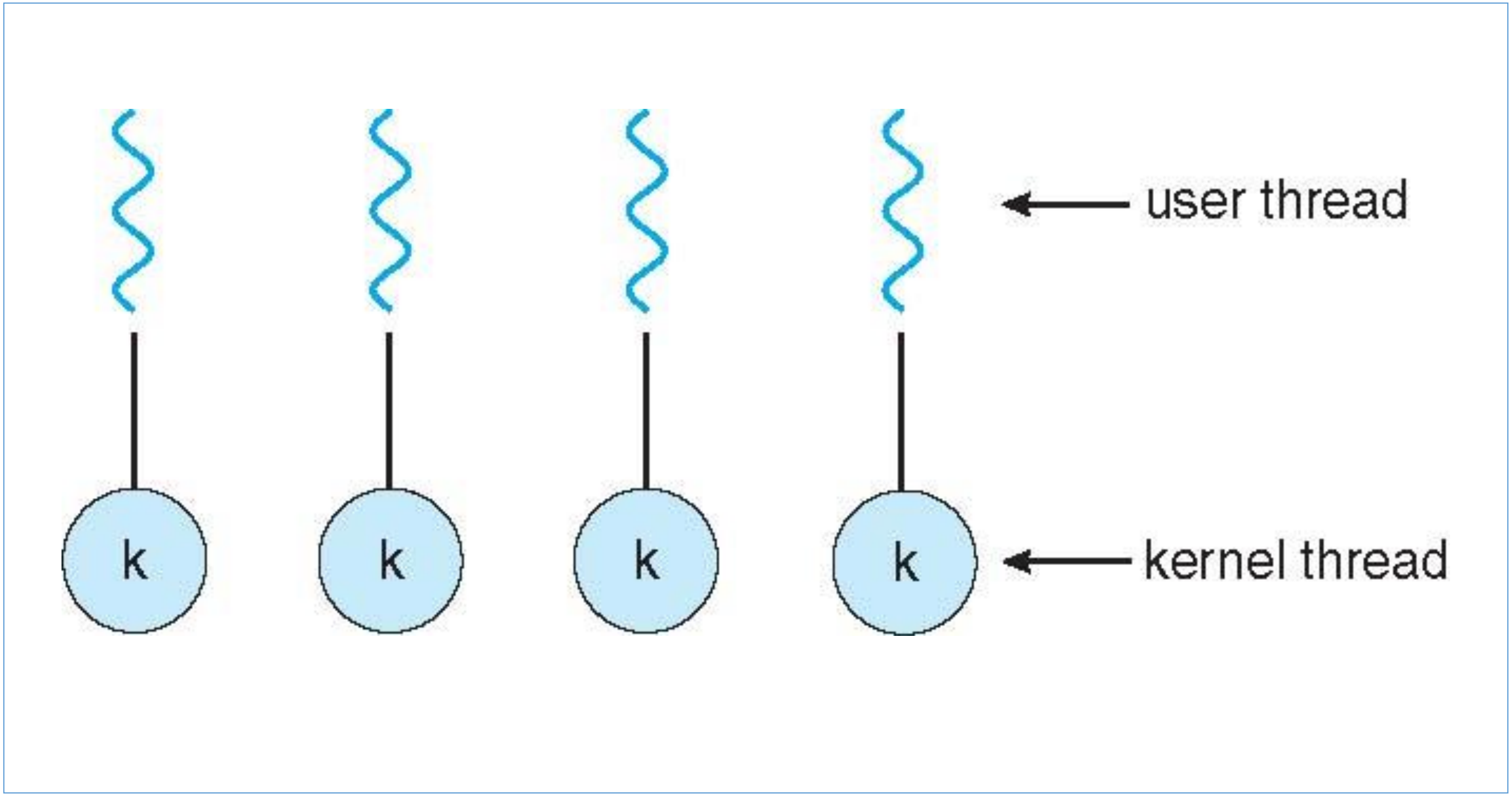
Kernel Threads

- *Kernel threads* are supported by the OS
  - kernel sees threads and schedules at the granularity of threads
  - Most modern OSs like Linux, Mac OS X, Win XP support kernel threads
  - Mapping of user-level threads to kernel threads is usually one-to-one, e.g. Linux and Windows, but could be many-to-one, or many-to-many
  - Win32 thread library is a kernel-level thread library

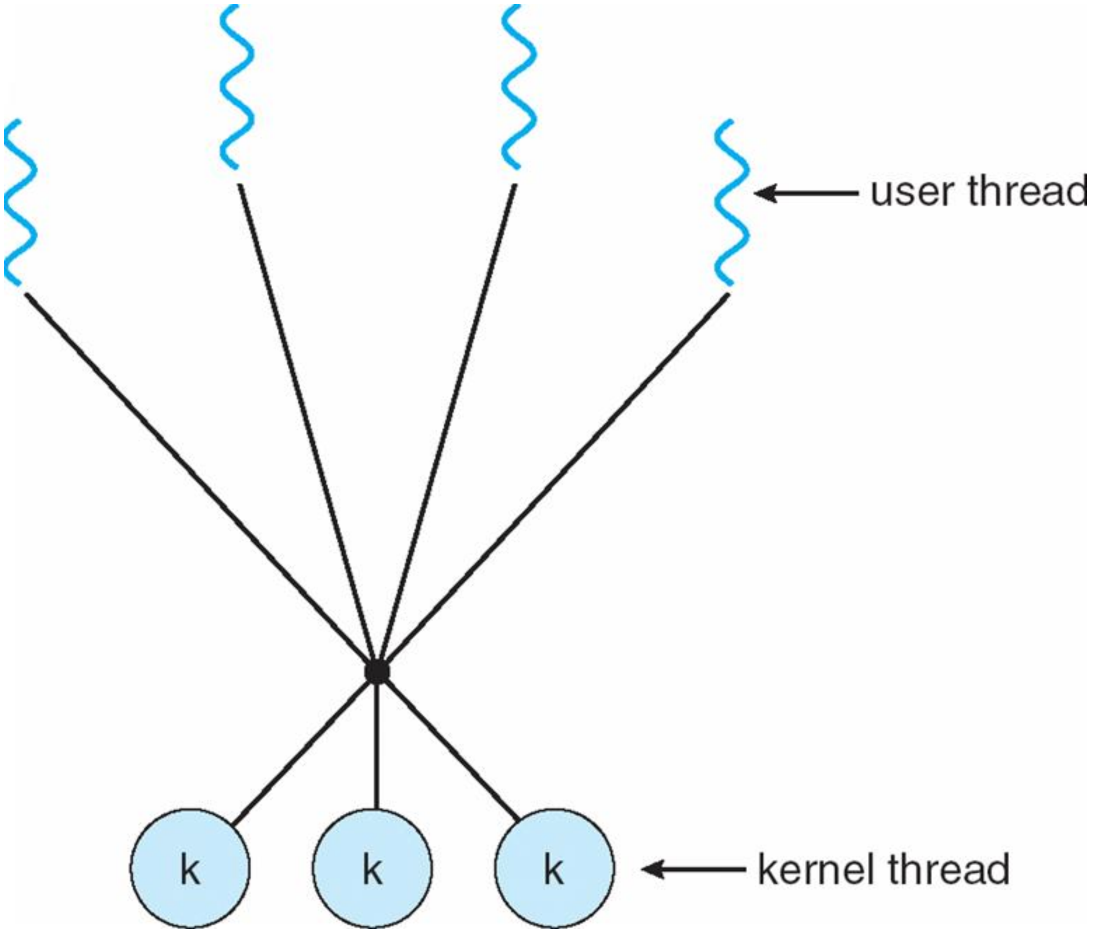
Many-to-One Model



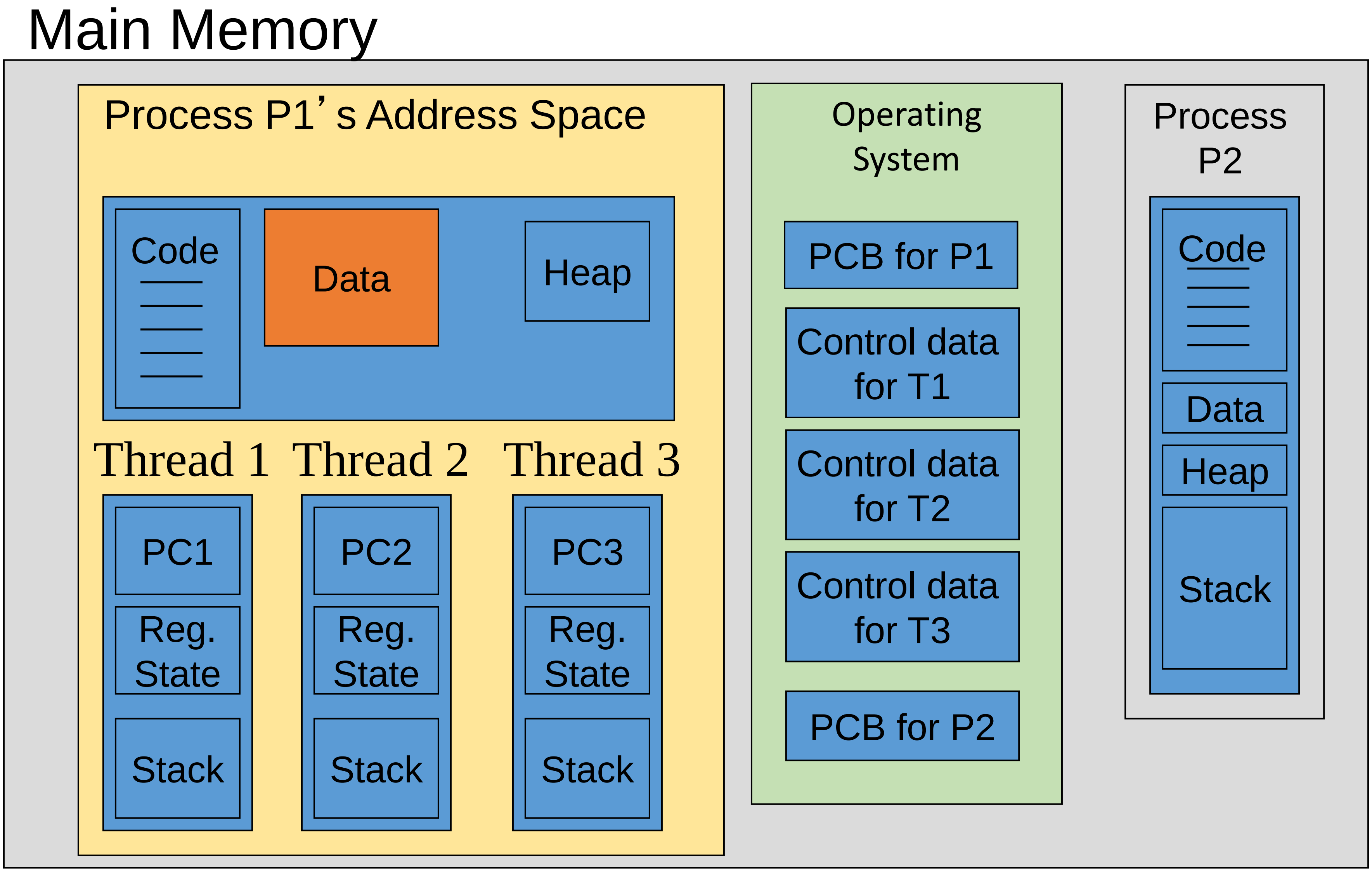
One-to-one Model



Many-to-Many Model



Multiple Threads

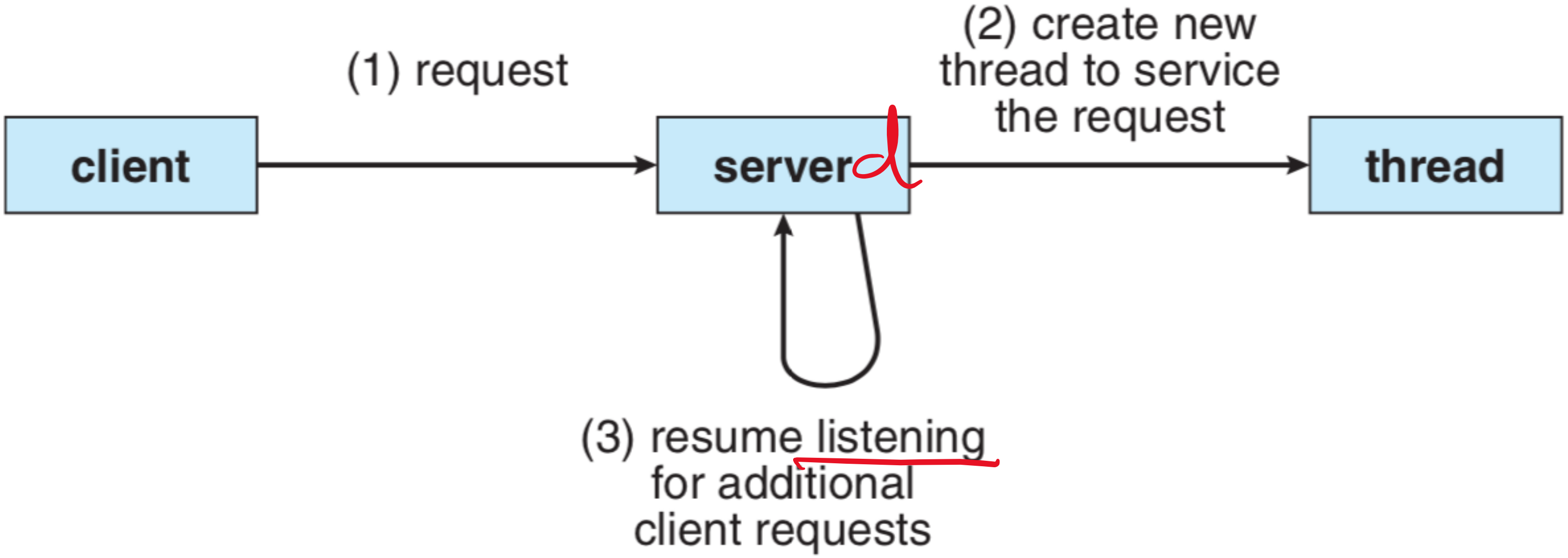


Benefits of multithreaded architecture

- Responsiveness.
  - Useful for interactive applications
- Resource sharing
  - Memory and resources are shared within process
- Context-switching overhead
  - Low creating and managing overhead
- Scalability
  - Threads can run in **parallel** with multicore system

daemon

Example.



client

(1) request

serverd

(2) create new thread to service the request

thread

(3) resume listening for additional client requests

The benefits often come with cost

- Who pay?
  Programmer/system designer

- Identifying tasks
  - Dividing tasks into concurrent subtasks
- Balance
  - Balance the load on different cores
- Splitting data
  - Divide data to be access by different cores
- Identifying data dependency
  - Identifying synchronization mechanisms

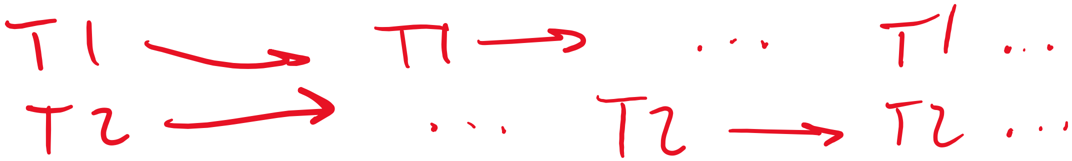
- Testing and debugging
  - Different path of executions need to be considered

Thread Safety

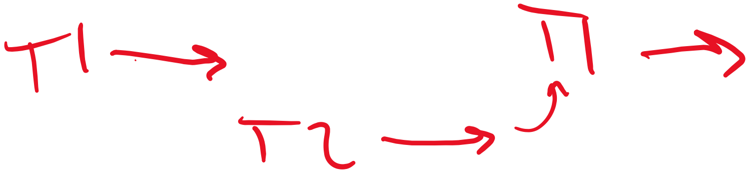
Thread Safety

- Thread Safe

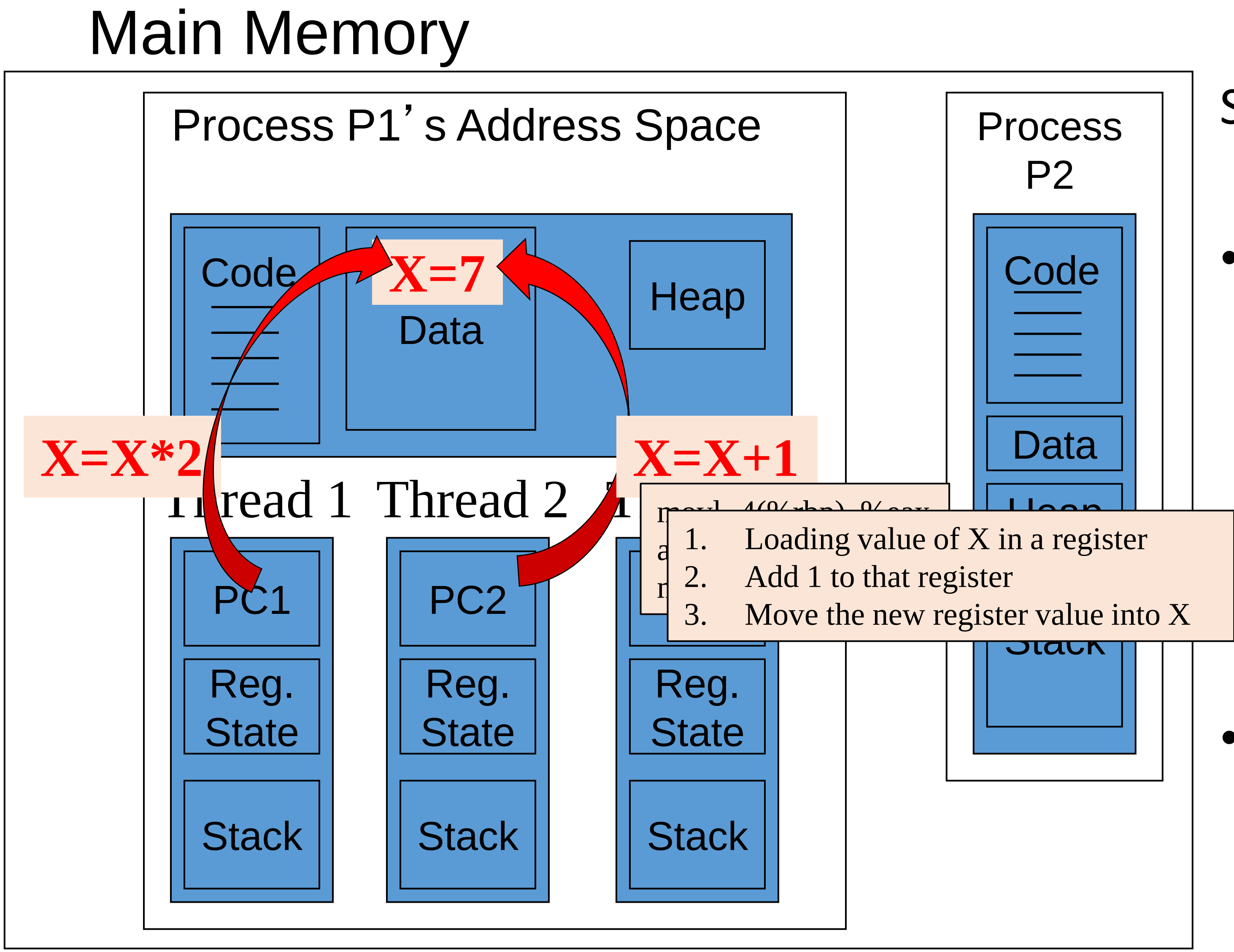
  If the code behaves correctly during **simultaneous** or ***concurrent*** execution by multiple threads



- Reentrant
  - If the code behaves correctly when a ***single*** thread is interrupted in the middle of executing the code reenters the same code



Thread Safety



1. Loading value of X in a register
2. Add 1 to that register
3. Move the new register value into X

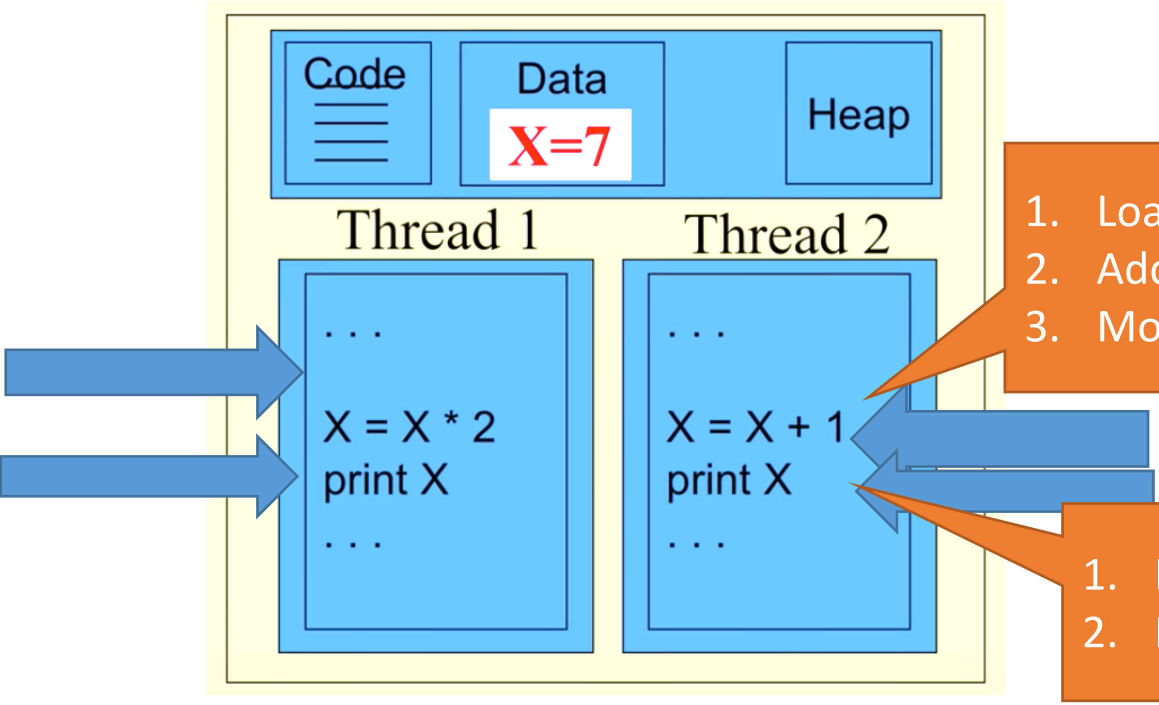
Suppose:

- Thread1 wants to multiply X by 2
- Thread2 wants to increment X
- *Could have a* ***race condition***

University of Colorado Boulder

NOT Thread Safe

- Given the following code, what are the possible values that are printed



```
. . .
X = X * 2
print X
. . .
```

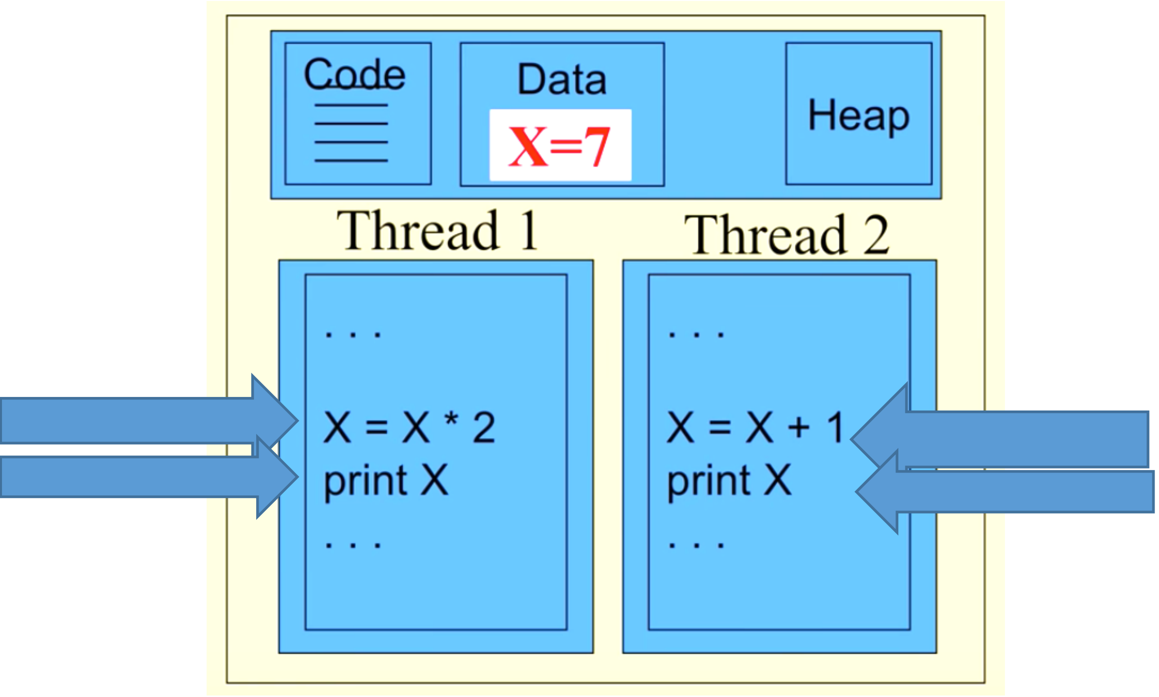
```
. . .
X = X + 1
print X
. . .
```

1. Load value X in a register
2. Add 1 to that of register value
3. Move the new register value into X

1. Load value of X in a register
2. Pass that value to system call

NOT Thread Safe

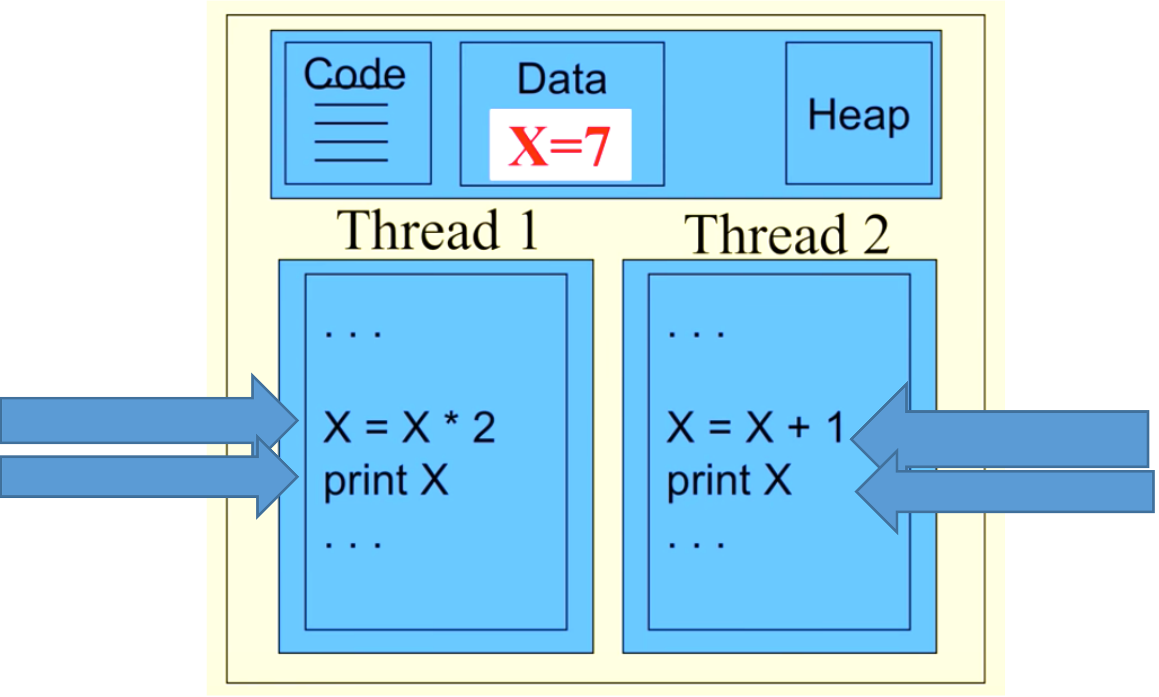
- Given the following code, what are the possible values that are printed



Thread 1	Thread 2
14	
	15

NOT Thread Safe

- Given the following code, what are the possible values that are printed



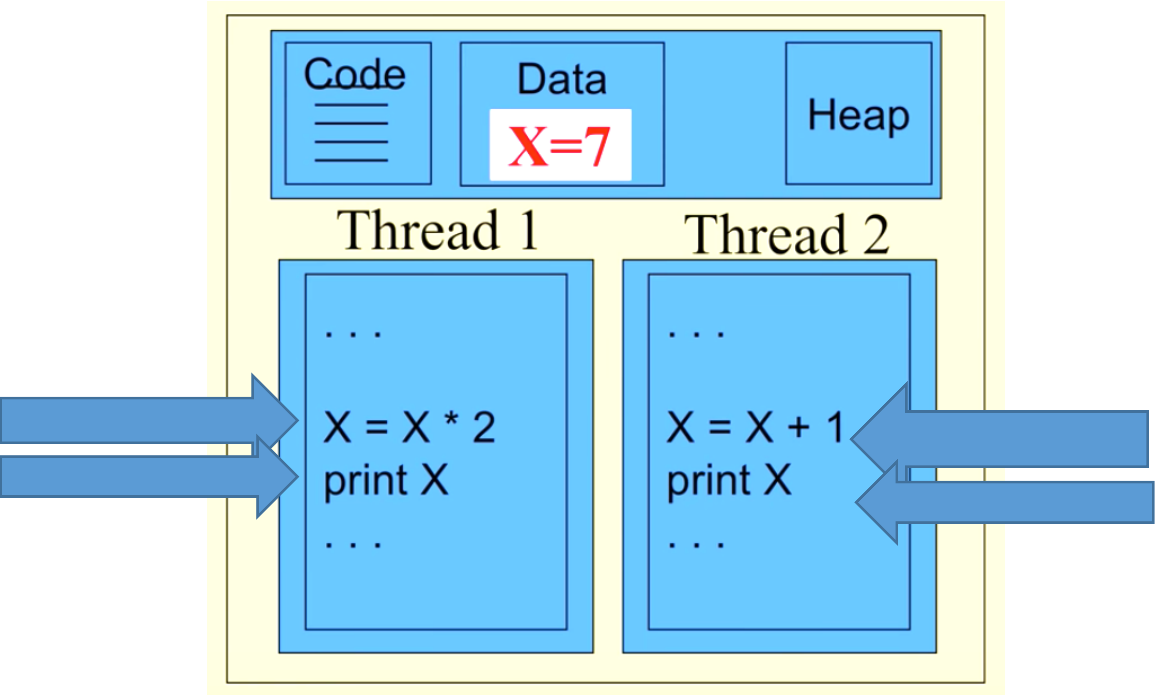
Thread 1 Thread 2

8

16

NOT Thread Safe

- Given the following code, what are the possible values that are printed



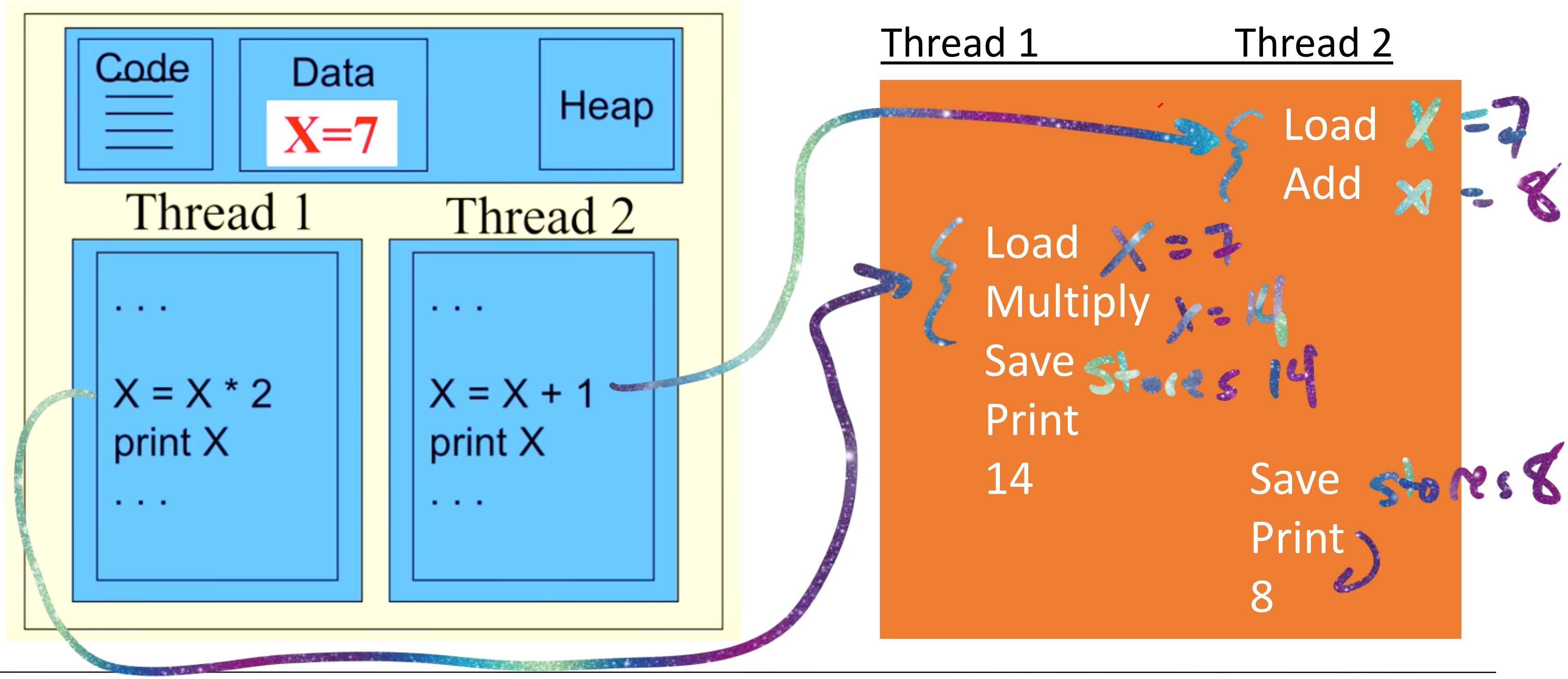
```
. . .
X = X * 2
print X
. . .
```

```
. . .
X = X + 1
print X
. . .
```

Thread 1	Thread 2
14	
	8

NOT Thread Safe

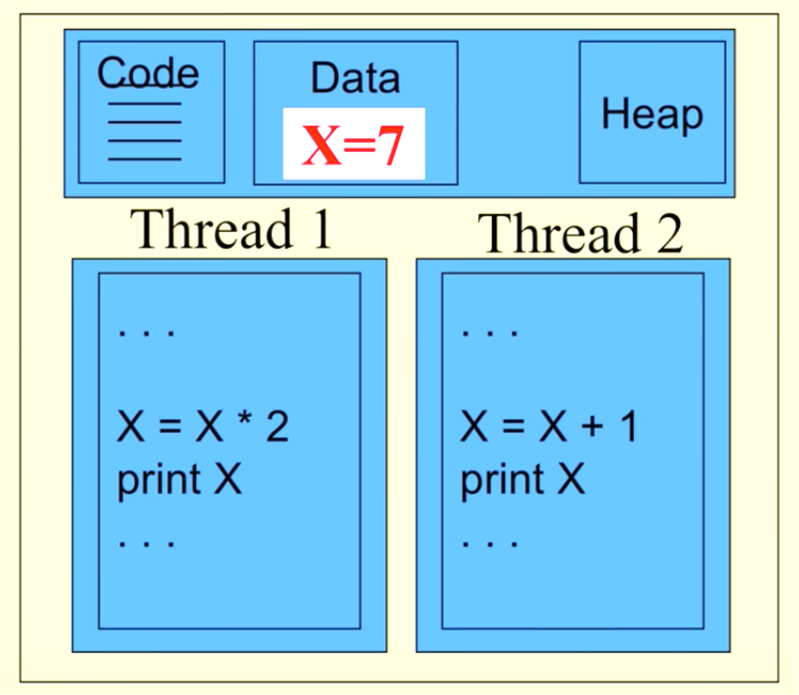
- Given the following code, what are the possible values that are printed



Thread 1	Thread 2
	Load X =7
	Add X = 8
Load X = 7	
Multiply x = 14	
Save stores 14	
Print	
14	
	Save stores 8
	Print
	8

NOT Thread Safe

- Given the following code, what are the possible values that are printed



Code

Data

X=7

Heap

Thread 1

```
. . .
X = X * 2
print X
. . .
```

Thread 2

```
. . .
X = X + 1
print X
. . .
```

Thread 1	Thread 2
14	15
16	8
14	8
8	8
14	14
….	

Thread-Safe Code

- Code is **thread-safe**

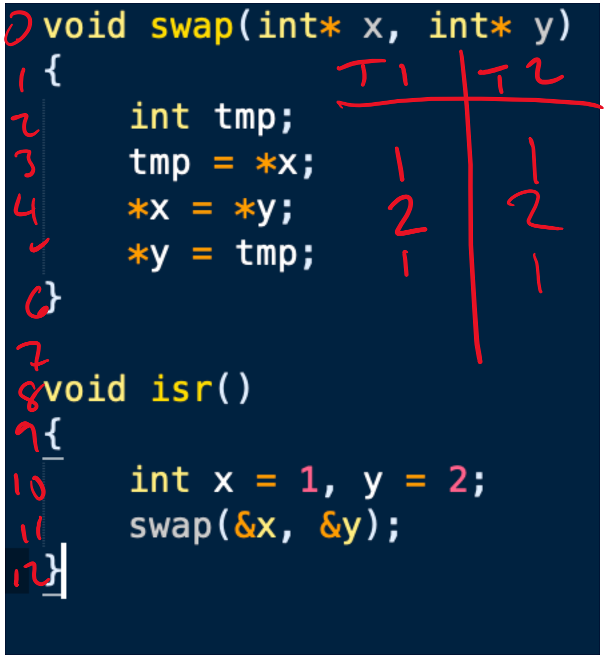
  it behaves correctly during simultaneous
  or *concurrent* execution by multiple threads.

  - multiple threads accessing the same shared data can cause different values to be used in different threads (**race condition**)
  - can be thread safe if each piece of **shared data is only be accessed by only one thread at any given time**

- If two threads share and execute the same code, then it is not safe to use the following unprotected **shared** data
  - use of global variables is not thread safe
  - use of static variables is not thread safe
  - use of heap variables is not thread safe

Reentrant Code

- Reentrancy:
  - Code is reentrant if a **single thread** (really, a sequence of execution) **can be interrupted in the middle of executing the code** and **then reenter the same code later in safe manner** (before the first entry has been completed).



```
0  void swap(int* x, int* y)
1  {
2      int tmp;
3      tmp = *x;
4      *x = *y;
5      *y = tmp;
6  }
7
8  void isr()
9  {
10     int x = 1, y = 2;
11     swap(&x, &y);
12 }
```

T1	T2
1	1
2	2
1	1

Interrupted!

T1 : → l4

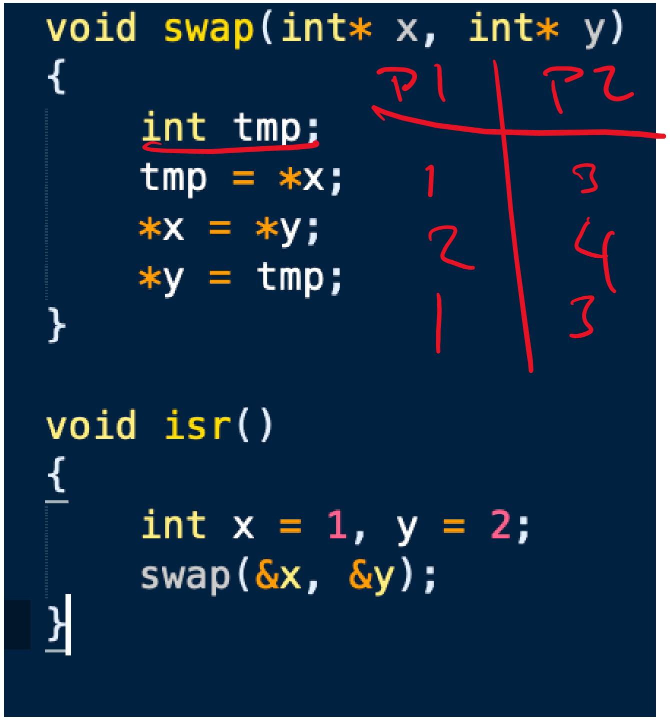
T2 → l4 → return

2

P1: swap(1,2)

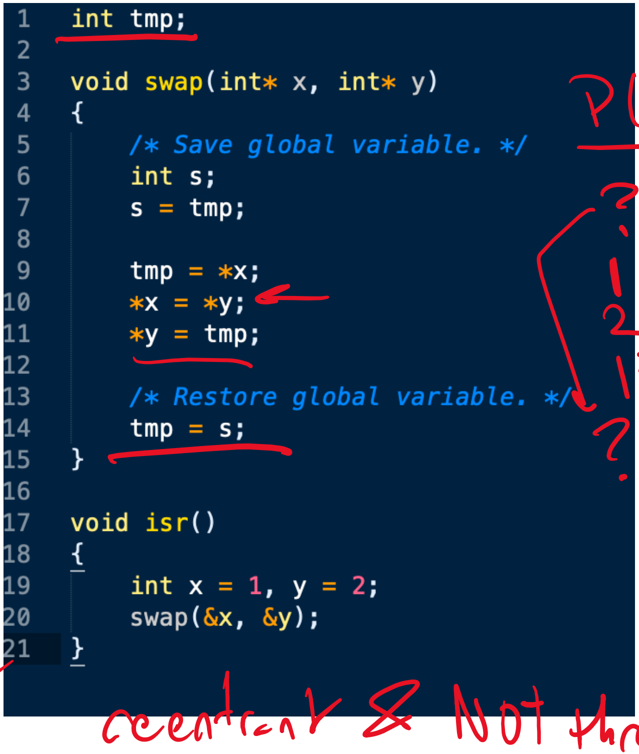
Reentrant Code

P2: swap(3,4)



```c
void swap(int* x, int* y)
{
    int tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;
}

void isr()
{
    int x = 1, y = 2;
    swap(&x, &y);
}
```

P1	P2
1	3
2	4
1	3



```c
int tmp;

void swap(int* x, int* y)
{
    /* Save global variable. */
    int s;
    s = tmp;

    tmp = *x;
    *x = *y;
    *y = tmp;

    /* Restore global variable. */
    tmp = s;
}

void isr()
{
    int x = 1, y = 2;
    swap(&x, &y);
}
```

P1	P2
?	1
1	3
2	4
1	3
?	1

reentrant & Not threadsafe

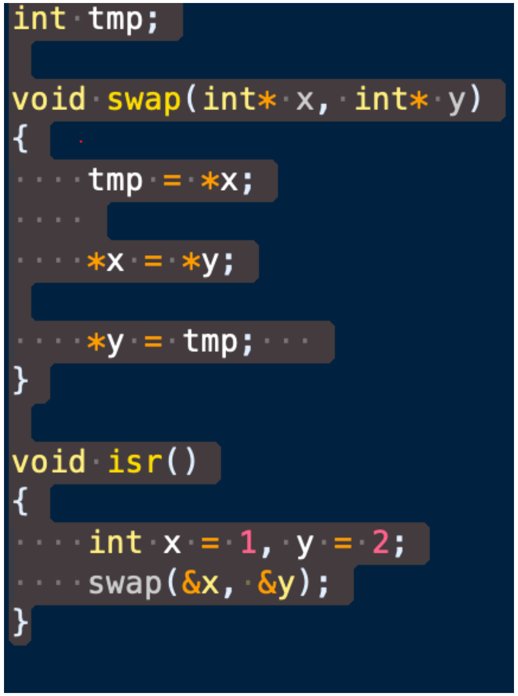
- Reentrancy:
  - Code is reentrant if a **single thread** (really, a sequence of execution) **can be interrupted in the middle of executing the code** and **then reenter the same code later in safe manner** (before the first entry has been completed).

Reentrant Code

- Reentrancy was developed for interrupt service routines (ISRs)
  - In the middle of an OS processing an interrupt, its ISR can be interrupted to process a second interrupt
  - The same OS ISR code may reentered a 2nd time before the 1st interrupt has been fully processed.
  - If the ISR code is not well-written, i.e., reentrant, then the system could hang or crash.

Reentrant Code Example

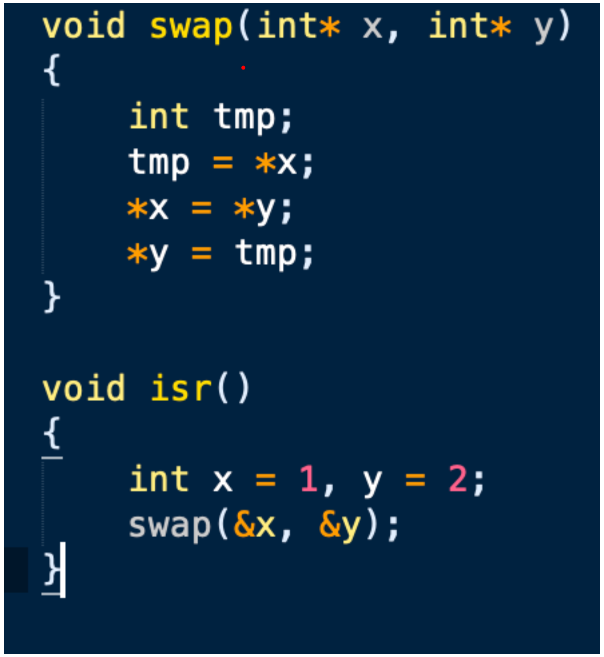
Neither Reentrant Nor Thread-safe



```
int tmp;

void swap(int* x, int* y)
{
    tmp = *x;
    
    *x = *y;

    *y = tmp;   
}

void isr()
{
    int x = 1, y = 2;
    swap(&x, &y);
}
```

P1	P2
1	3
2	4
3 →	3

Reentrant and Thread-safe



```
void swap(int* x, int* y)
{
    int tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;
}

void isr()
{
    int x = 1, y = 2;
    swap(&x, &y);
}
```

P1: swap(1, 2)

P2: swap(3, 4)

→ P1 → P2 → P1

Side note

- Show an example of reentrancy (wiki with swap)
- Show an example of using global variable but still be reentrance
    - How: Store the global variable before and after each call of that function.

Thread-safe but not Reentrant

- Must limit access to one thread at a time

```
function f() {

     lock(); // gives this thread exclusive use
                // keeps all other threads waiting
     change global variable G;
→
     unlock (); // Give use to another thread
}
```

This code is NOT reentrant because if f() is interrupted just before the unlock(), and f() is called a 2nd time, the system will hang, because the 2nd call will try to lock, then be unable to lock, because the 1st call had not yet unlocked the system.

Thread Safe and Reentrant Code

- Code can be

Thread-Safe And Reentrant	Thread-Safe and Not Reentrant
Not Thread-Safe and Reentrant	Neither Thread-Safe Nor Reentrant

How to write Safe Code

- What is safe?
  - The output would be something that you actually mean to get

- **How to write reentrance code:**
  - Do not access mutable global or function-static variables.
  - Do not self-modify code.
  - Do not invoke another function that is itself non-reentrant.

- Tips
  - Make sure all functions have NO state
  - Make sure your objects are "recursive-safe"
  - Make sure your objects are correctly encapsulated
  - Make sure you thread-safe code is recursive-safe
  - Make sure users know your object is not thread-safe.

Synchronization

Concurrency

- Multiple processes/threads executing at the same time accessing a shared resource
  - Reading the same file/resource
  - Accessing shared memory

- Benefits of Concurrency
  - Speed
  - Economics

Concurrency

- **Concurrency** is the interleaving of processes in time to give the appearance of simultaneous execution.
  - Differs from parallelism, which offers genuine simultaneous execution.

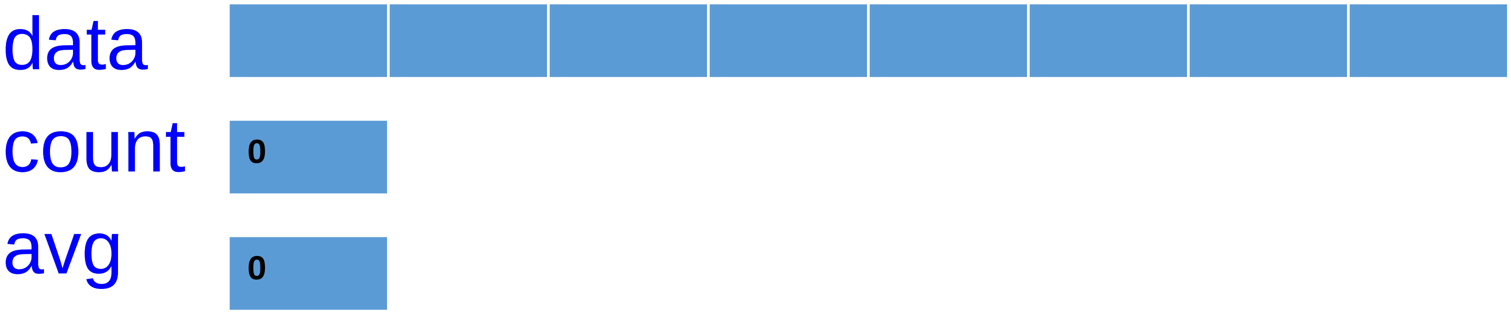
- **Parallelism** introduces the issue that different processors may run at different speeds
  - This problem is mirrored in concurrency as different processes progress at different rates

Condition for Concurrency

- Must give the same results as serial execution
- Using shared data requires synchronization

Example

Shared data/variables



data

count 0

avg 0

Serial Execution

```
// Waiting for data
while (true)
{
  v = get_value()
  add_new_value (v)
}
```

```
// Process data
add_new_value (v) {
    int sum = avg * count + v;
    data[count] = v;
    count++;
    avg = (sum + v) / count;
}
```

Concurrent Execution

Process 1

```
while (true)
{
     v = get_value() = 2
     add_new_value (v)
}
add_new_value (v) {
     int sum = avg * count + v;
     data[count] = v;
     count++;
     avg = (sum  + v)/ count;
}
```

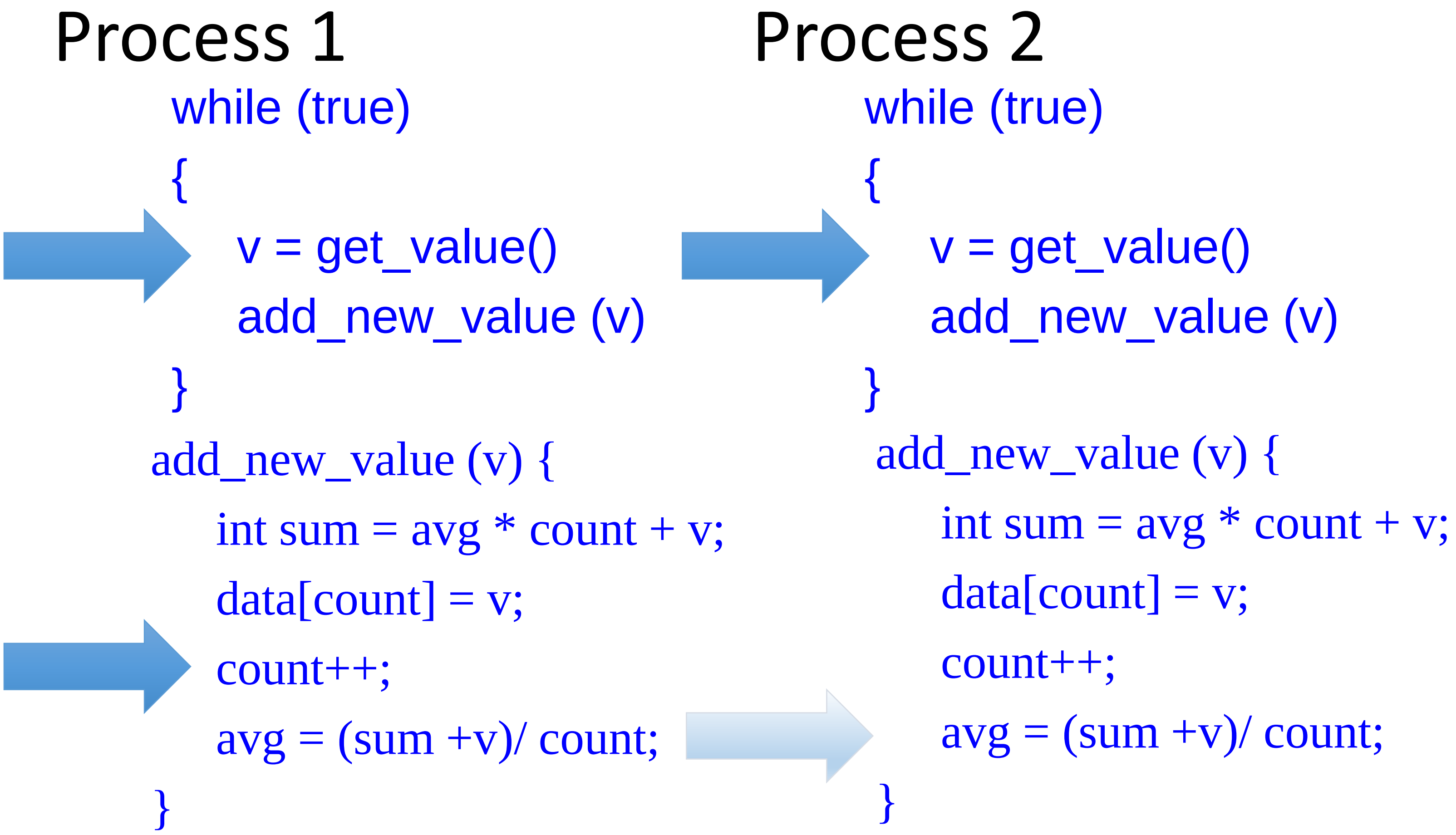
Process 2

```
while (true)
{
     v = get_value() = 4
     add_new_value (v)
}
add_new_value (v) {
     int sum = avg * count + v;
     data[count] = v;
     count++;
     avg = (sum  + v)/ count;
}
```

Are we still going to have data consistency?

Correct values at all times in all conditions?

Concurrency



Process 1

```
while (true)
{
   v = get_value()
   add_new_value (v)
}
add_new_value (v) {
   int sum = avg * count + v;
   data[count] = v;
   count++;
   avg = (sum +v)/ count;
}
```

Process 2

```
while (true)
{
   v = get_value()
   add_new_value (v)
}
add_new_value (v) {
   int sum = avg * count + v;
   data[count] = v;
   count++;
   avg = (sum +v)/ count;
}
```

Concurrency

More tricky issue:

C statements can compile into several machine language instructions

e.g. count++

```
mov R2, count
inc R2
mov count, R2
```

P1	P2
2	4
3	→ 4
4	4

If these low-level instructions are *interleaved*, e.g. one process is preempted, and the other process is scheduled to run, then the results of the ***count*** value can be unpredictable

Concurrency

- Suppose we have the following sequence of interleaving.
- Let count = 5 initially.

```
// P1 - counter++
(1)  reg1 = count;
(3)  reg1 = reg1 + 1;
(5)  counter = reg1;
```

```
// P2 - counter++
(2) reg2 = count;
(4) reg2 = reg2 + 1;
(6) count = reg2;
```

		1	2	3	4	5	6
count	5	5	5	5	5	6	6
reg1	?	5	5	6	6	6	6
reg2	?	?	5	5	6	6	6

RACE CONDITION

Concurrency

- Must give the same results as serial execution

- Using shared data requires synchronization

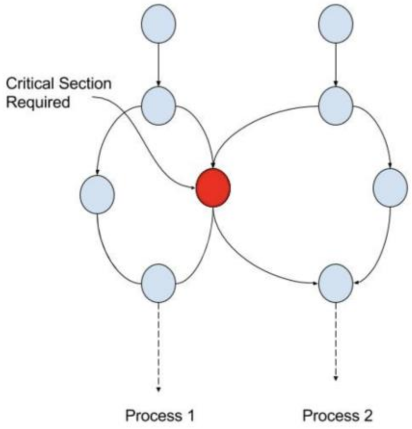
How can various mechanisms be used to ensure the **orderly execution of cooperating processes** that share address space so that data consistency is maintained?

Synchronization techniques:

- Mutex
- Semaphore
- Condition Variable
- Monitor

Race Condition

- Occurs in situations where:
  - Two or more processes (or threads) are accessing a shared resource
  - and the final result **depends on the order of instructions** are executed
- ***The part of the program where a shared resource*** is accessed is called *critical section*
- We need a mechanism to ***prohibit multiple processes from accessing a shared resource at the same time***



Critical Section

Process 1

```
while (true)
{
    v = get_value()
    add_new_value (v)
}
add_new_value (v) {
    int sum = avg * count + v;
    data[count] = v;
    count++;
    avg = sum / count;
}
```

Process 2

```
while (true)
{
    v = get_value()
    add_new_value (v)
}
add_new_value (v) {
    int sum = avg * count + v;
    data[count] = v;
    count++;
    avg = sum / count;
}
```

Critical Section

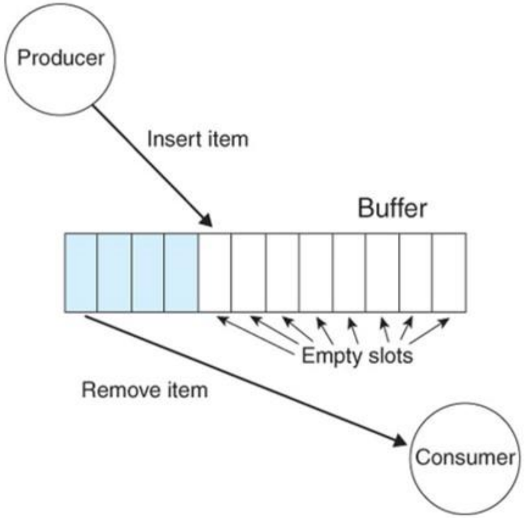
Where is the critical section in the ***add_new_value ()*** code?

Race Condition: Solution

- Solution must satisfy the following conditions:
  - **mutual exclusion**
    - if process P_i is executing in its critical section, then no other processes can be executing in their critical sections
  - **progress**
    - if no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that wish to enter their critical sections can participate in the decision on which will enter its critical section next
    - this selection **cannot** be postponed indefinitely (OS must run a process, hence "progress")
  - **bounded waiting**
    - there exists a bound, or limit, on the number of times other processes can enter their critical sections after a process X has made a request to enter its critical section and before that request is granted **(no starvation)**

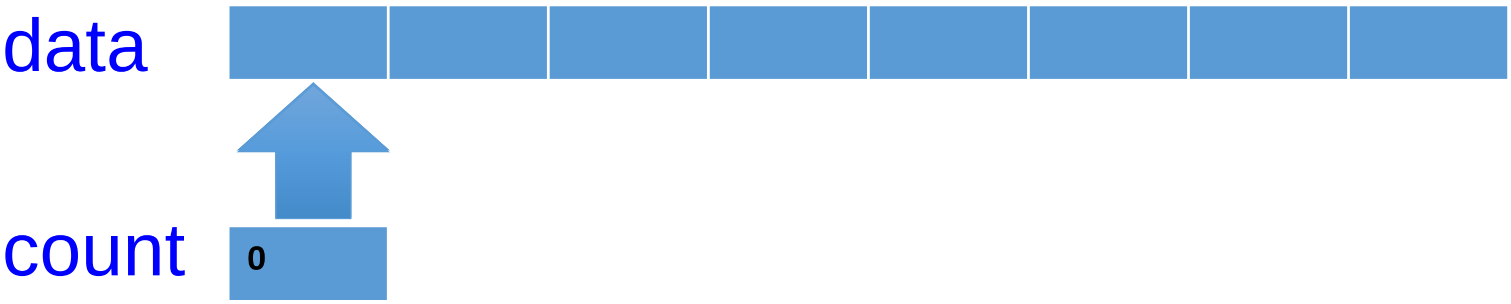
Producer-Consumer Problem (*Bounded Buffer Problem)*

- Two processes (producer and consumer) share a fixed size buffer
- Producer puts new information in the buffer
- Consumer takes out information from the buffer

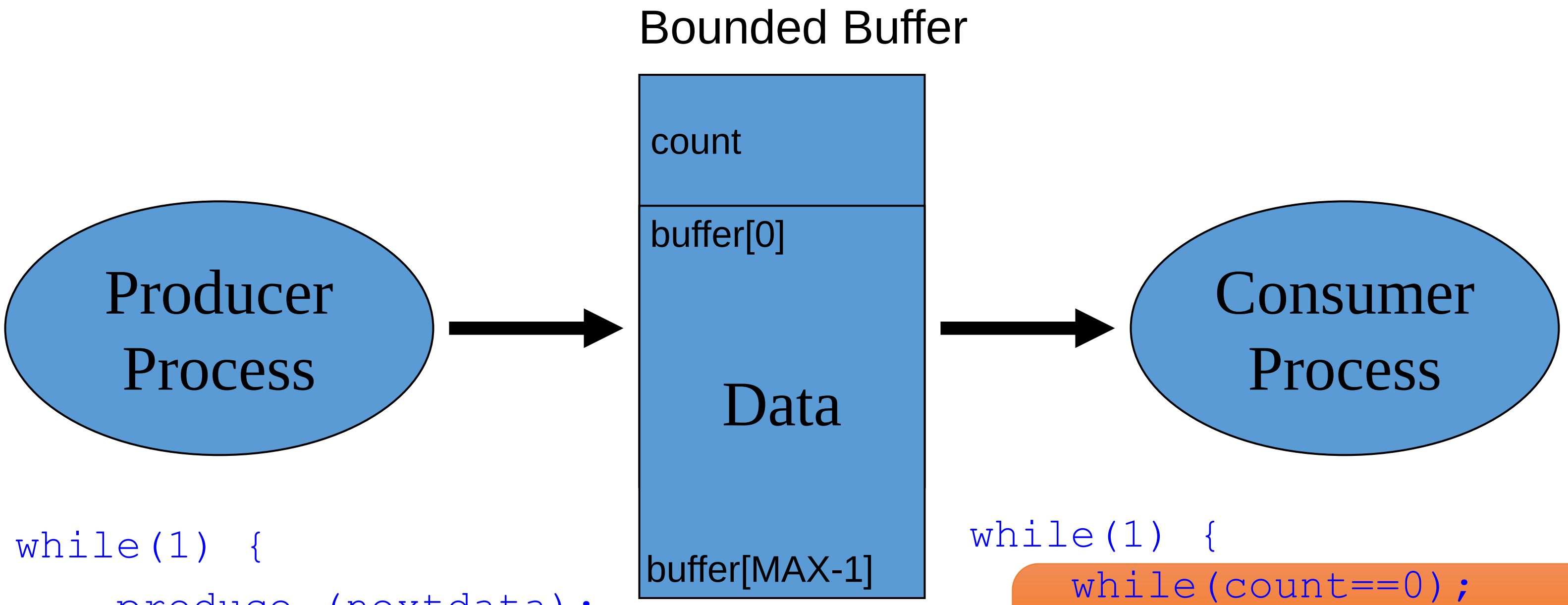


Producer-Consumer Problem

Shared data



Producer-Consumer Problem



```
while(1) {
    produce (nextdata);
    while(count==MAX);
    buffer[count] = nextdata;
    count++;
}
```

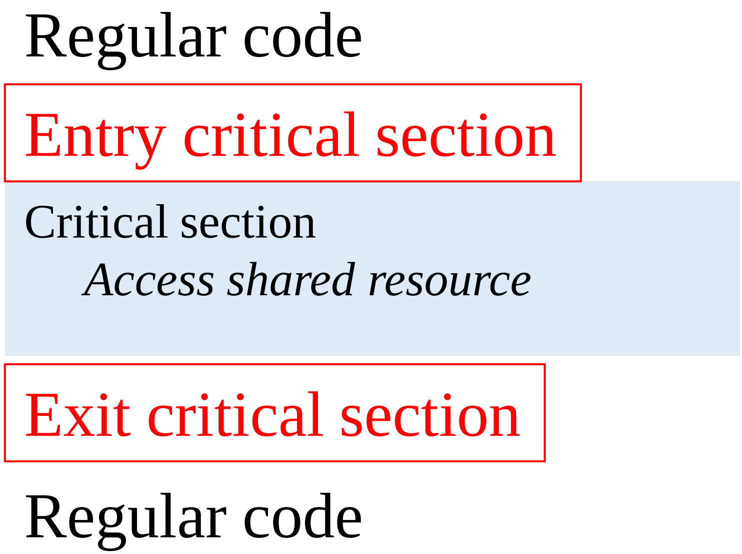
Producer writes new data into buffer and increments counter

```
while(1) {
    while(count==0);
    getdata = buffer[count-1];
    count--;
    consume (getdata)
}
```

Consumer reads new data from buffer and decrements counter

Mutual Exclusion

- Mechanism to make sure no more than one process can execute in a critical section at any time
- How can we implement mutual exclusion?



Regular code

Entry critical section

Critical section

Access shared resource

Exit critical section

Regular code

Critical Section

```
// Producer
while(1) {
    produce (nextdata);
    while (count==MAX);
    Entry critical section
    buffer[count] = nextdata;
    count++;
    Exit critical section
}
```

```
// Consumer
while(1) {
    while (count==0);
    Entry critical section
    getdata = buffer[count-1];
    count--;
    Exit critical section
    consume (getdata)
}
```

Critical Section

Solution 1: Disabling interrupts

- Ensure that when a process is executing in its critical section, it cannot be preempted

- Disable all interrupts before entering a CS

- Enable all interrupts upon exiting the CS

```
shared int counter;

      producer code                    consumer code
disableInterrupts();             disableInterrupts();
counter++;                       counter--;
enableInterrupts();              enableInterrupts();
. . . remaining producer code    . . . remaining consumer code
```

Solution 1: Disabling Interrupts

Problems:

- If a user forgets to enable interrupts???
- Interrupts could be disabled arbitrarily long
- Really only want to prevent p_1 and p_2 from interfering with one another; disabling interrupts blocks all processes
- Two or more CPUs???

Solution 2: **Software Only Solution**

```
shared boolean lock = FALSE;
shared int counter;
```

Code for producer

```
/* Acquire the lock */
  while(lock){ no_op;}(1)
  lock = TRUE; (3)

/* Execute critical
    section */
  counter++;

/* Release lock */
  lock = FALSE;
```

Code for consumer

```
/* Acquire the lock */
while(lock){ no_op;} (2)
lock = TRUE;  (4)

/* Execute critical
        section */
   counter--;

/* Release lock */
lock = FALSE;
```

A flawed lock implementation:

Both processes may enter their critical section if there is a context switch just before the <lock = TRUE> statement

Solution 2: Software Only Solution

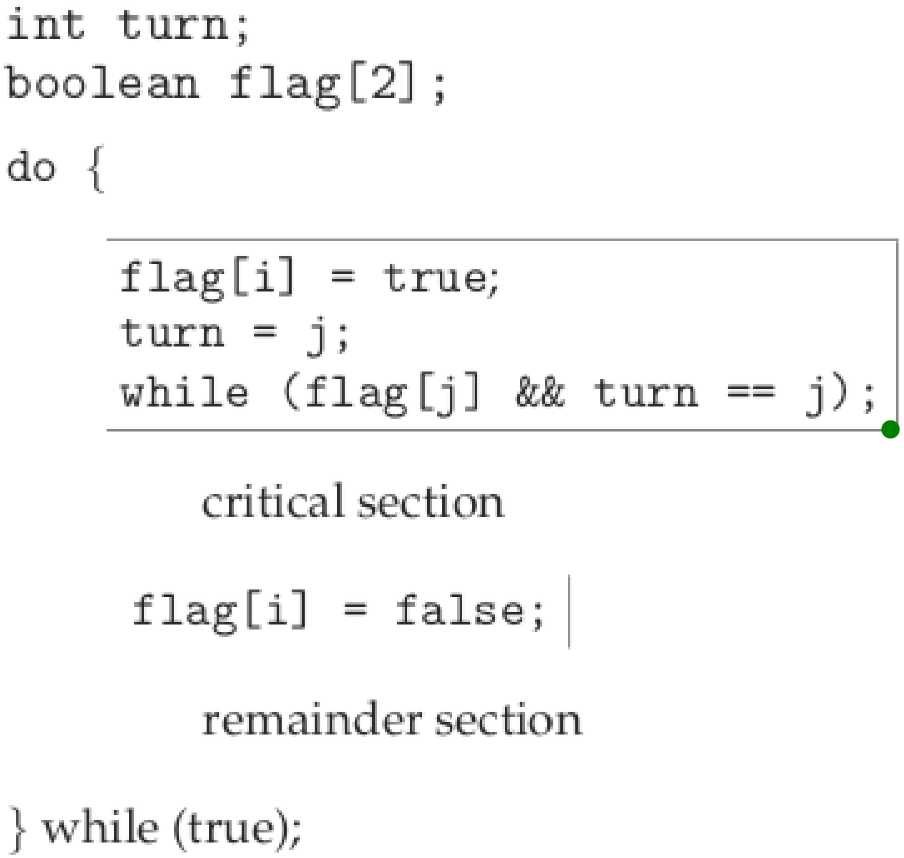
- Implementing mutual exclusion in software is not always work

- Need help from hardware

- Modern processors provide such support
    - Atomic test and set instruction
    - Atomic compare and swap instruction

Solution 2: Software Only Solution

- Peterson's solution:
  Restricted to only 2 processes.



```
int turn;
boolean flag[2];

do {

    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

- Turn indicates who will be run next
- Flag indicates who is ready to run next

Need to prove:
(1) MutEx is preserved
(2) Progress is made
(3) Bounded-waiting is met

Atomic Test-and-Set

- Need to be able to look at a variable and set it up to some value without being interrupted

  y = read (x); //Check the value;
  x = value;   //Set the value

- Modern computing systems provide such an instruction called *test-and-set (TS);*

```
boolean TS(boolean *target){
      boolean rv = *target;
      *target = TRUE;
      return rv; // returns original value of the target
}
```

- The entire sequence is a single instruction (atomic), implemented in hardware

Solution 3: Mutual exclusion using TS

```
shared boolean lock = FALSE;
shared int counter;
```

Code for p_1

```
/* Acquire the lock */
  while(TS(&lock)) ;

/* Execute critical section */
  counter++;

/* Release lock */
  lock = FALSE;
```

Code for p_2

```
    /* Acquire the lock */
    while(TS(&lock)) ;

    /* Execute critical section */
    counter--;

    /* Release lock */
    lock = FALSE;
```

Atomic Test-and-Set

- The boolean TS () instruction is essentially a swap of values
- Mutual exclusion is achieved - no race conditions
  - If one process X tries to obtain the lock while another process Y already has it, X will wait in the loop
  - If a process is testing and/or setting the lock, no other process can interrupt it
- The system is exclusively occupied for only a short time - the time to test and set the lock, and not for entire critical section
- Don’t have to disable and reenable interrupts
- Do you see any problems?
  - busy waiting.

```
while(TS(&lock)) ;
```

Mutual exclusion using TS

shared boolean lock = FALSE;

shared int count;

Shared data_type buffer [MAX];

Code for p_1

```
while(1) {
    produce (nextdata)
    while(count==MAX);
    Acquire(lock);
    buffer[count] = nextdata;
    count++;
    Release(lock);
}
```

Code for p_2

```
while (1) {
    while(count==0);
    Acquire(lock);
    data = buffer[count-1];
    count--;
    Release(lock);
    consume(data);
}
```

Mutual exclusion using TS

shared boolean lock = FALSE;
shared int count;
Shared data_type buffer [MAX];

Code for p_1

```
while(1) {
    produce (nextdata)
    while(count==MAX);
    Acquire(lock);
    buffer[count] = nextdata;
    count++;
    Release(lock);
}
```

Code for p_2

```
while (1) {
    while(count==0);
    Acquire(lock);
    data = buffer[count-1];
    count--;
    Release(lock);
    consume(data);
}
```

CPU is spinning & waiting

Mutual exclusion using TS

shared boolean lock = FALSE;
shared int count;
Shared data_type buffer [MAX];

Code for p_1

```
while(1) {
    produce (nextdata)
    while(count==MAX);
    Acquire(lock);
    buffer[count] = nextdata;
    count++;
    Release(lock);
}
```

Code for p_2

```
while (1) {
    while(count==0);
    Acquire(lock);
    data = buffer[count-1];
    count--;
    Release(lock);
    consume(data);
}
```

CPU is spinning & waiting

Mutual exclusion using TS

shared boolean lock = FALSE;
shared int count;
Shared data_type buffer [MAX];

Code for p_1

```
while(1) {
    produce (nextdata)
    while(count==MAX);
    Acquire(lock);
    buffer[count] = nextdata;
    count++;
    Release(lock);
}
```

Code for p_2

```
while (1) {
    while(count==0);
    Acquire(lock);
    data = buffer[count-1];
    count--;
    Release(lock);
    consume(data);
}
```

CPU is spinning & waiting

Mutual exclusion using TS

shared boolean lock = FALSE;
shared int count;
Shared data_type buffer [MAX];

Code for p_1

```
while(1) {
    produce (nextdata)
    while(count==MAX);
    Acquire(lock);
    buffer[count] = nextdata;
    count++;
    Release(lock);
}
```

Code for p_2

```
while (1) {
    while(count==0);
    Acquire(lock);
    data = buffer[count-1];
    count--;
    Release(lock);
    consume(data);
}
```

How can we eliminate the busy waiting?

Mutual exclusion using TS

shared boolean lock = FALSE;
shared int count;
Shared data_type buffer [MAX];

Code for p_1

```
while(1) {
    produce (nextdata)
    while(count==MAX);
    Acquire(lock);
    buffer[count] = nextdata;
    count++;
    Release(lock);
}
```

Code for p_2

```
while (1) {
    while(count==0);
    Acquire(lock);
    data = buffer[count-1];
    count--;
    Release(lock);
    consume(data);
}
```

Mutual exclusion using TS

shared boolean lock = FALSE;
shared int count;
Shared data_type buffer [MAX];

Code for p_1

```
while(1) {
    produce (nextdata)
    while(count==MAX);
    Acquire(lock);
    buffer[count] = nextdata;
    count++;
    Release(lock);
}
```

Code for p_2

```
while (1) {
    while(count==0);
    Acquire(lock);
    data = buffer[count-1];
    count--;
    Release(lock);
    consume(data);
}
```

- Look first at the busy wait when there is no **room for data (p1)** or **no data (p2)**
  - Need a way to pause a process/thread until the space is available or data is available.

Mutual Exclusion

- How can we eliminate the busy waiting?

- Need a way to pause a process/thread until the lock is available

sleep( ) and **wakeup( )** primitives

- *sleep()*: causes a running process to block

- *wakeup(pid)*: causes the process whose id is *pid* to move to ready state
  - No effect if process *pid* is not blocked

```
// producer – place data into buffer
while(1) {
   if (counter==MAX) sleep();
   buffer[in] = nextdata;
   in = (in+1) % MAX;
   counter++;
   if (counter == 1) wakeup (p2);
}
```

```
// consumer – take data out of buffer
while(1) {
   if (counter==0) sleep();
   getdata = buffer[out];
   out = (out+1) % MAX;
   counter--;
   if (counter == MAX – 1) wakeup (p1);
}
```

sleep( ) and wakeup( ) primitives

- Problem with counter++ and counter-- still exist
  - Can be solved using TS but it has busy waiting
- Possible problem with order of execution:
  - Consumer reads counter and counter == 0
  - Scheduler schedules the producer
  - Producer puts an item in the buffer and signals the consumer to wake up
    - Since consumer has not yet invoked sleep(), the wakeup() invocation by the producer has no effect
  - Consumer is scheduled, and it blocks
  - Eventually, producer fills up the buffer and blocks
  - How can we solve this problem?
    - Need a mechanism to count the number of sleep() and wakeup() invocations

Semaphores

- Dijkstra proposed more general solution to mutual exclusion

- Semaphore **S is an abstract data type** that is accessed only through two standard atomic operations
  - wait( ) (also called P(), from Dutch word *proberen* “to test”)
    - somewhat equivalent to a test-and-set
    - also involves *decrementing* the value of S
  - signal( ) (V(), from Dutch word *verhogen* “to increment”)
    - *increments* the value of S

- OS provides ways to create and manipulate semaphores atomically

Semaphores

```
typedef struct {
    int value;
    PID *list[ ];
} semaphore;
```

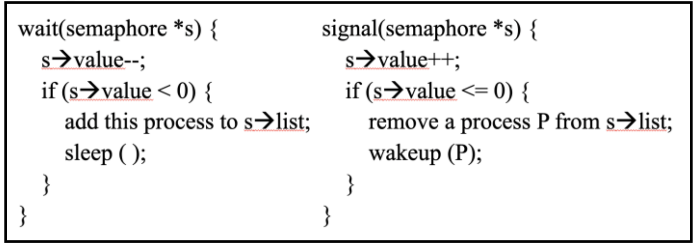
Both wait() and signal() operations are atomic

```
wait(semaphore *s) {
    s→value--;
    if (s→value < 0) {
        add this process to s→list;
        sleep ( );
    }
}
```

```
signal(semaphore *s) {
    s→value++;
    if (s→value <= 0) {
        remove a process P from s→list;
        wakeup (P);
    }
}
```

Mutual Exclusion using Semaphores

Binary Semaphore



```
wait(semaphore *s) {
    s→value--;
    if (s→value < 0) {
        add this process to s→list;
        sleep ( );
    }
}
```

```
signal(semaphore *s) {
    s→value++;
    if (s→value <= 0) {
        remove a process P from s→list;
        wakeup (P);
    }
}
```

```
semaphore S = 1;  // initial value of semaphore is 1
int counter;      // assume counter is set correctly somewhere in code
```

Process P1:	**Process P2:**
wait(S);	wait(S);
// execute critical section	// execute critical section
counter++;	counter--;
signal(S);	signal(S);

- Both processes atomically wait() and signal() the semaphore S, which enables mutual exclusion on critical section code, in this case protecting access to the shared variable ***counter***
- ***This solve mutual exclusion but will also eliminate busy wait***

University of Colorado
Boulder

Problems with semaphores

```
shared R1, R2;
semaphore Q = 1;    // binary semaphore as a mutex lock for R1
semaphore S = 1;    // binary semaphore as a mutex lock for R2
```

Process P1:

```
wait(S);        (1)
wait(Q);        (3)

modify R1 and R2;

signal(S);
signal(Q);
```

Process P2:

```
wait(Q);    (2)
wait(S);    (4)

modify R1 and R2;

signal(Q);
signal(S);
```

Potential for deadlock

Deadlock

- In the previous example,
  - Each process will block on a semaphore
  - The signal() statements will never get executed, so there is no way to wake up the two processes
  - There is no rule about the order in which wait( ) and signal( ) operations may be invoked
  - In general, with more processes sharing more semaphores, the potential for deadlock grows

Other problematic scenarios

- A programmer mistakenly follows a wait() with a second wait() instead of a signal()

- A programmer forgets and omits the wait(mutex) or signal(mutex)

- A programmer reverses the order of wait() and signal()

Another problem with synchronization

shared R1, R2;

semaphore S=1; // binary semaphore as a mutex lock for R1 & R2

Process P1:	*Process P2:*	*Process P3:*
wait(S)	wait(S)	wait(S);
modify R1 and R2;	modify R1 and R2;	modify R1 and R2;
Signal (S);	signal(S);	signal(S);

Potential for starvation

Starvation

- The possibility that a process would never get to run
- For example, in a multi-tasking system the resources could switch between two individual processes
- E.g. Depending on how the processes are scheduled, a third process may never get to run
- E.g. The third task is being starved of accessing the resource

Semaphore Solution for Mutual Exclusion

```
shared lock = 1; // initial value of semaphore 1
shared int count;
Shared data_type buffer [MAX];
```

Code for p_1

```
while(1) {
    produce (nextdata)
    while(count==MAX);
    wait(lock);
    buffer[count] = nextdata;
    count++;
    signal(lock);
}
```

Code for p_2

```
while (1) {
    while(count==0);
    wait(lock);
    data = buffer[count-1];
    count--;
    signal(lock);
    consume(data);
}
```

Semaphore Solution for Mutual Exclusion

```
shared lock = 1; // initial value of semaphore 1
shared int count;
Shared data_type buffer [MAX];
```

Code for p_1

```
while(1) {
    produce (nextdata)
    while(count==MAX);
    wait(lock);
    buffer[count] = nextdata;
    count++;
    signal(lock);
}
```

Code for p_2

```
while (1) {
    while(count==0);
    wait(lock);
    data = buffer[count-1];
    count--;
    signal(lock);
    consume(data);
}
```

- Busy waiting removed from the mutual exclusion when waiting on lock
- Does it solve all busy waiting issues?

pthread Synchronization

- Mutex locks in Linux

- Some implementations provide semaphores through POSIX SEM extension
  - Not part of pthread standard

```
#include <pthread.h>
pthread_mutex_t m; //declare a mutex object
pthread_mutex_init (&m, NULL); // initialize mutex object
```

```
//thread 1
pthread_mutex_lock (&m);
   //critical section code for th1
pthread_mutex_unlock (&m);
```

```
//thread 2
pthread_mutex_lock (&m);
   //critical section code for th2
pthread_mutex_unlock (&m);
```

th3

pthread mutex

- pthread mutexes can have only one of two states:
  lock or unlock

- Important restriction
  - Mutex ownership:
    Only the thread that locks a mutex can unlock that mutex
  - Mutexes are strictly used for mutual exclusion while binary semaphores can also be used for synchronization between two threads or processes

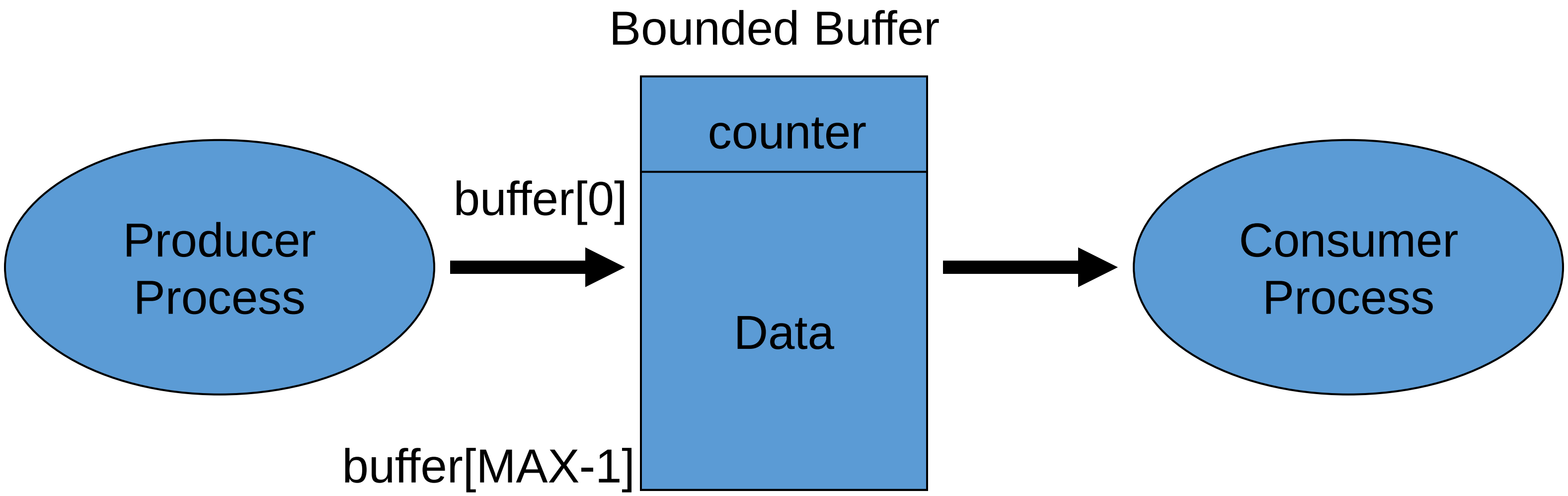
POSIX semaphores

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
//  pshared: 0 (among threads); 1 (among processes)

int sem_wait(sem_t *sem); //same as wait( )

int sem_post(sem_t *sem); //same as signal( )

sem_getvalue( ); // check the current value of the semaphore
sem_close( ); // done with the semaphore then close
```

Producer-Consumer Problem
also known as
Bounded Buffer Problem

- We have already seen this problem with one producer and one consumer
- General problem: **multiple** producers and multiple consumers
- **Producers** puts new information in the buffer
- **Consumers** takes out information from the buffer

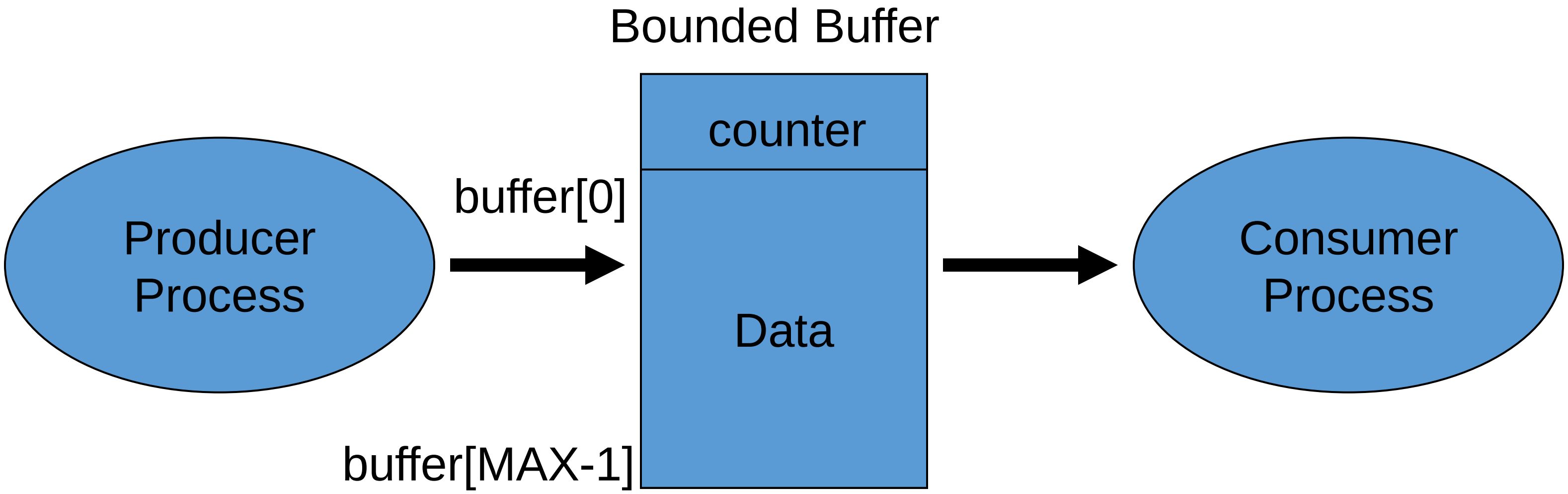
Prior Bounded-Buffer P/C Approach



- Producer places data into a buffer at the next available position

- Consumer takes information from the earliest item

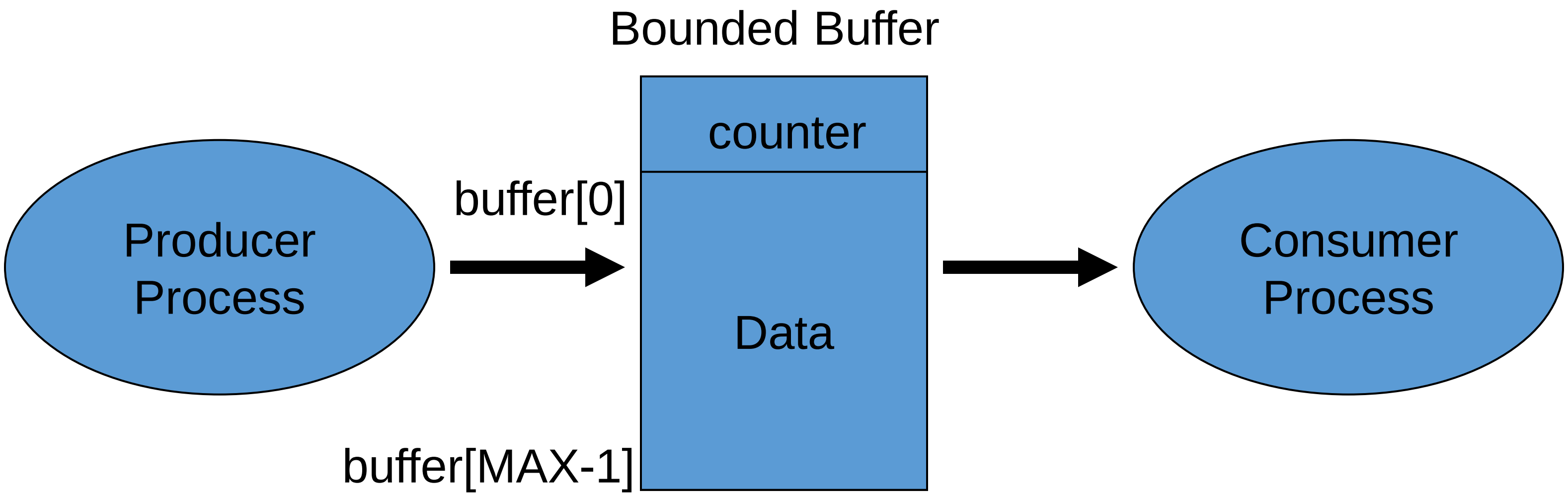
Prior Bounded-Buffer P/C Approach



```
while(1) {
    produce (nextdata)
    while(count==MAX);
    Acquire(lock);
    buffer[count] = nextdata;
    count++;
    Release(lock);
}
```

```
while (1) {
    while(count==0);
    Acquire(lock);
    data = buffer[count-1];
    count--;
    Release(lock);
    consume(data);
}
```

Prior Bounded-Buffer P/C Approach

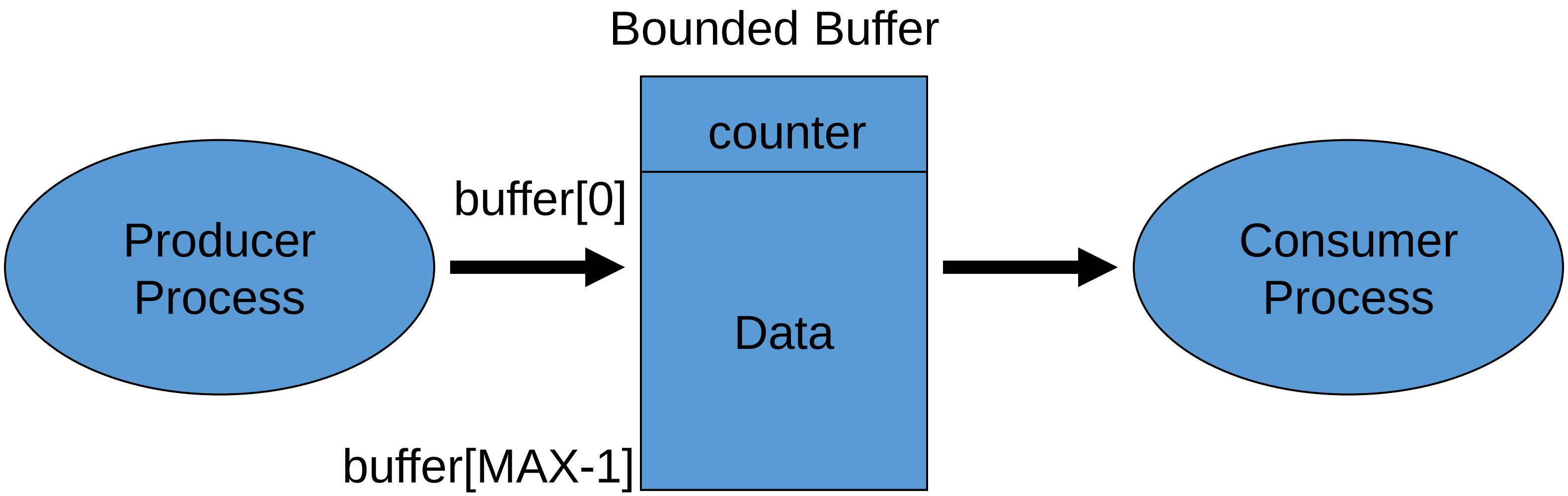


```
while(1) {
    produce (nextdata)
    while(count==MAX);
    Acquire(lock);
    buffer[count] = nextdata;
    count++;
    Release(lock);
}
```

```
while (1) {
    while(count==0);
    Acquire(lock);
    data = buffer[count-1];
    count--;
    Release(lock);
    consume(data);
}
```

Busy-wait!

Prior Bounded-Buffer P/C Approach



```
while(1) {
    produce (nextdata)
    while(count==MAX);
    Acquire(lock);
    buffer[count] = nextdata;
    count++;
    Release(lock);
}
```

```
while (1) {
    while(count==0);
    Acquire(lock);
    data = buffer[count-1];
    count--;
    Release(lock);
    consume(data);
}
```

while (TS(&lock));

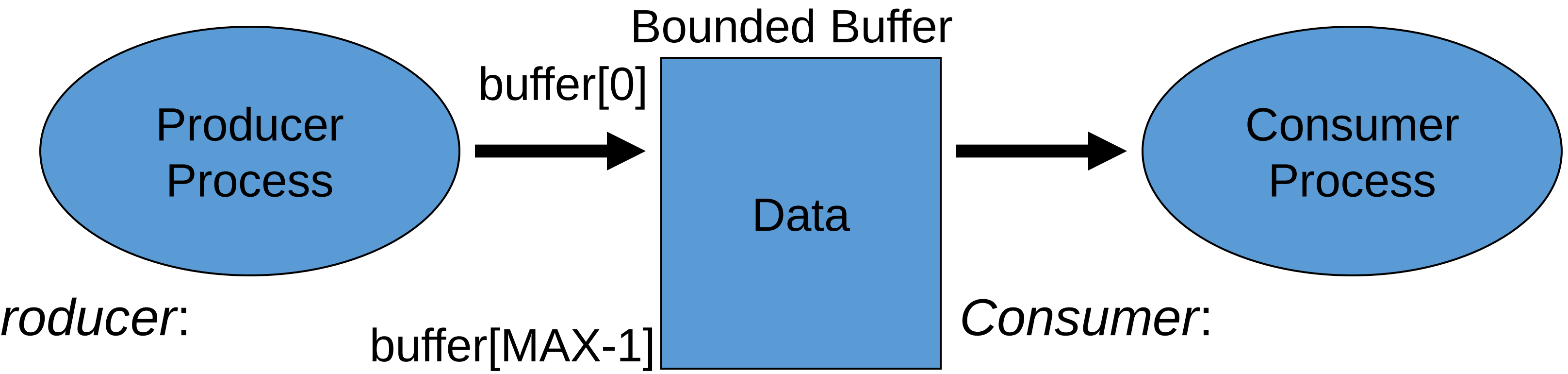
Busy-wait!

Bounded-Buffer Goals

- In the prior approach, both the producer and consumer are ***busy-waiting*** using locks

- Instead, want both to sleep as necessary
  - Goal #1: Producer should block when buffer is full
  - Goal #2: Consumer should block when the buffer is empty
  - Goal #3: mutual exclusion when buffer is partially full
    - Producer and consumer should access the buffer in a synchronized mutually exclusive way

Bounded Buffer Solution



Producer:

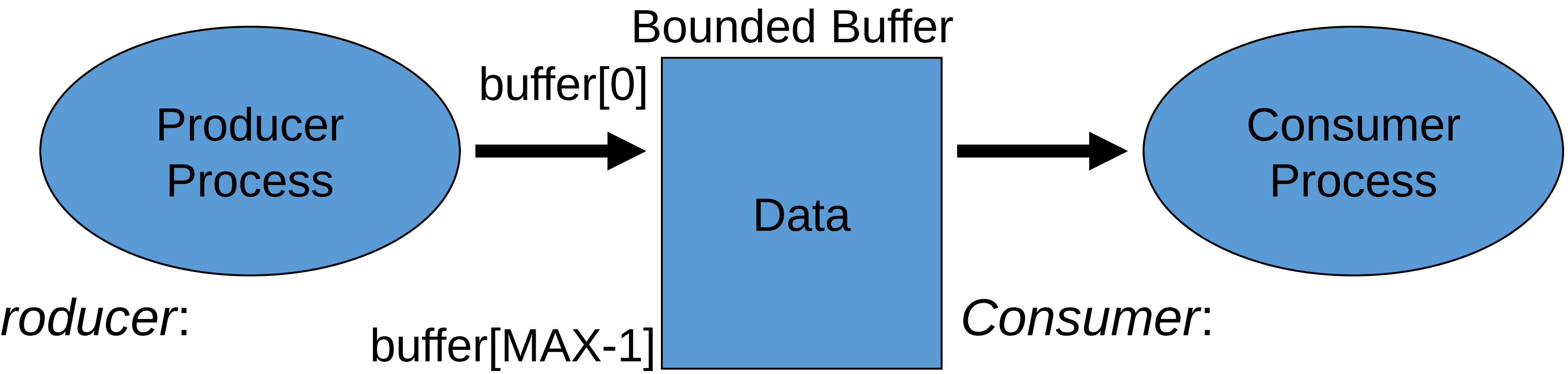
```
while(1) {
    …
    P(mutex)   // Goal 3: mutual
                  exclusion
       add item to buffer
    V(mutex)
    …
}
```

Consumer:

```
while(1) {
    ...
    P(mutex)   // Goal 3: mutual
                  exclusion
       remove item from buffer
    V(mutex)
    ...
}
```

Assume $mutex_{init} = 1$

Bounded Buffer Solution (2)



Goal #1: Producer should block when buffer is full

Producer:

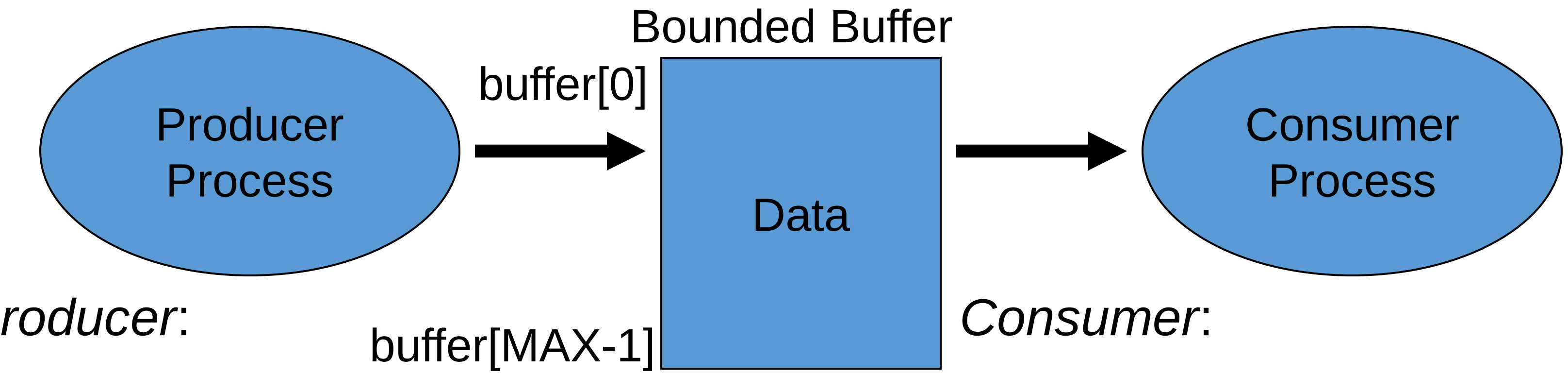
```
while(1) {
  P(empty) // Goal 1: full buffer
              blocks producer
    P(mutex)
       add item to buffer
    V(mutex)
    …
}
```

Assume $mutex_{init}$=1, $empty_{init}$=MAX

Consumer:

```
while(1) {
      ...
      P(mutex)
         remove item from buffer
      V(mutex)
   V(empty) // Goal 1: full
    buff
              blocks producer
}
```

Bounded Buffer Solution (3)



Goal #2: Consumer should block when the buffer is empty

Producer:

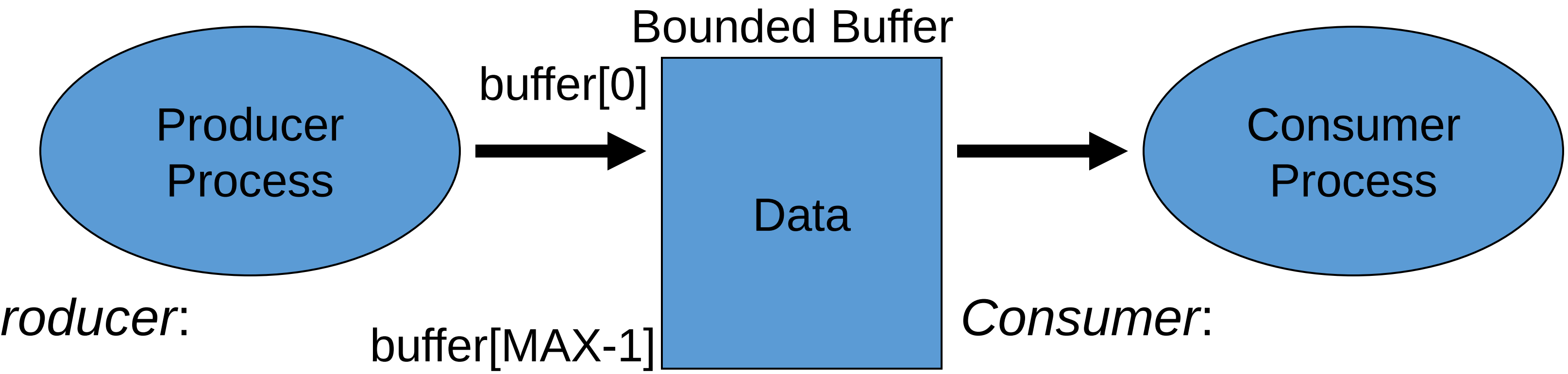
```
while(1) {
  P(empty)
    P(mutex)
      add item to buffer
    V(mutex)
  V(full) // Goal 2: empty buffer
             blocks consumer
}
```

Consumer:

```
while(1) {
  P(full) // Goal 2: empty
             buff blocks consumer
    P(mutex)
      remove item from buffer
    V(mutex)
  V(empty)
}
```

Assume *mutex*$_{init}$=1, *empty*$_{init}$=MAX, *full*$_{init}$=0

Bounded Buffer Solution (4)



Producer:

Consumer:

```
while(1) {
  P(empty)
    P(mutex)
      add ite
    V(mutex)
  V(full)
}
```

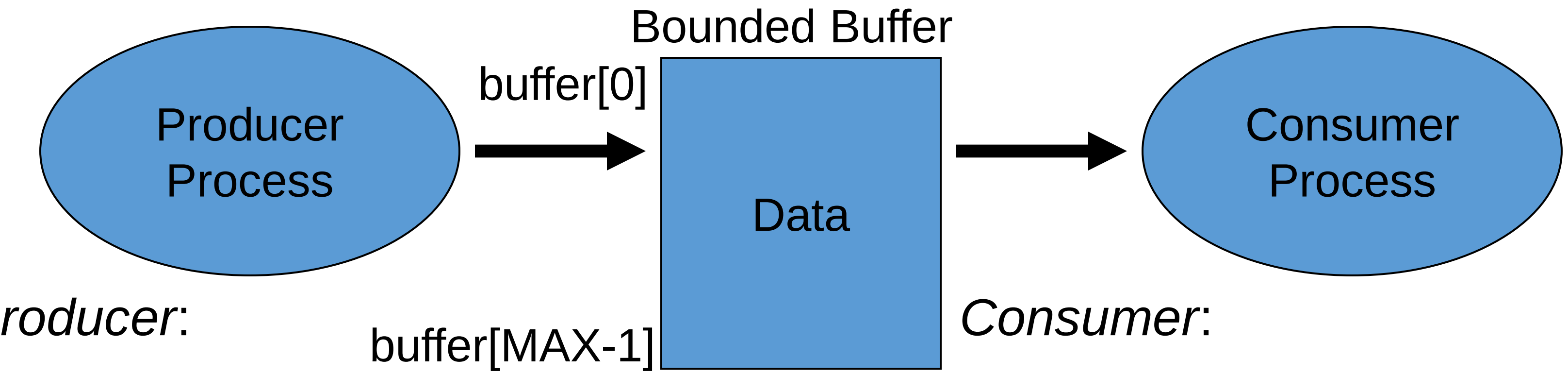
```
while(1) {

      tex)
      emove item from buffer
      tex)
      )
}
```

Achieves:

1) mutual exclusion
2) blocked producer if buffer full
3) blocked consumer if buffer empty
4) no deadlock
5) is efficient (no busy wait)

Assume $mutex_{init}$=1, $empty_{init}$=MAX, $full_{init}$=0

Bounded Buffer Solution (5)



Producer:

```
while(1) {
  P(space_avail)
    P(mutex)
       add item to buffer
    V(mutex)
  V(items_avail)
}
```

Consumer:

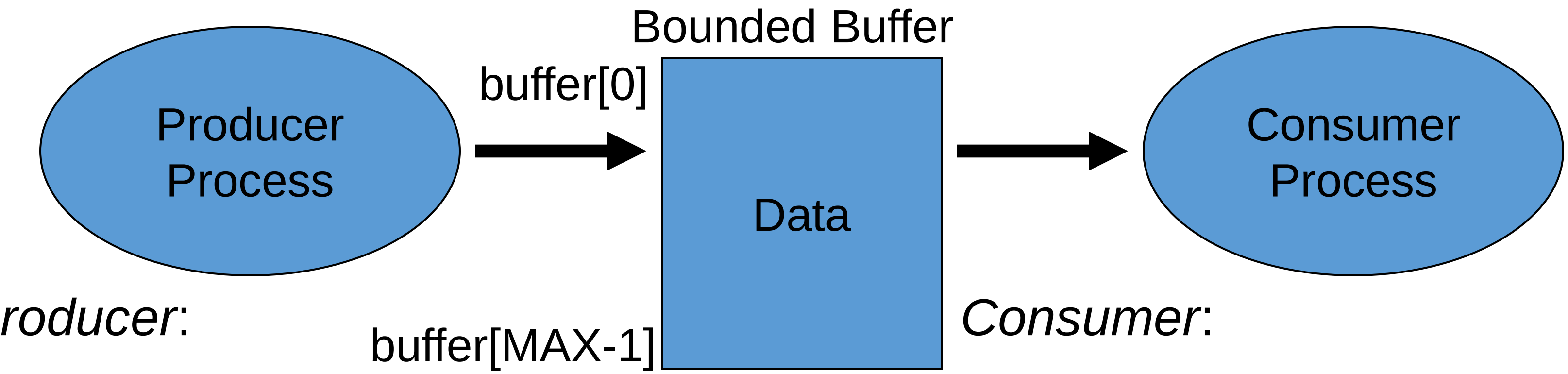
```
while(1) {
  P(items_avail)
    P(mutex)
       remove item from buffer
    V(mutex)
  V(space_avail)

}
```

Assume $mutex_{init}$=1,
$space\_avail_{init}$=MAX,
$items\_avail_{init}$=0

Bounded-Buffer Design

- **Goal #1: Producer should block when buffer is full**
  - Use a counting semaphore called *empty* that is initialized to empty_{init} = MAX
  - Each time the producer adds an object to the buffer, this decrements the # of empty slots, until it hits 0 and the producer blocks
- **Goal #2: Consumer should block when the buffer is empty**
  - Define a counting semaphore *items_avail* that is initialized to $\mathit{items\_avail}_{init}$ = 0
  - *items_avail* tracks the # of full slots and is incremented by the producer
  - Each time the consumer removes a full slot, this decrements *items_avail*, until it hits 0, then the consumer blocks
- **Goal #3: Mutual exclusion when buffer is partially full**
  - Use a mutex semaphore to protect access to buffer manipulation, mutex_{int} = 1

Bounded Buffer Solution (6)



Producer:

```
while(1) {
  P(space_avail)
    P(mutex)
      add item
    V(mutex)
  V(items_avail)
}
```

Will this solution work for multiple Producers and multiple consumers?

Consumer:

```
                1)

        item from buffer

  V(space_avail)

}
```

Assume $mutex_{init}$=1,
$space\_avail_{init}$=MAX,
$items\_avail_{init}$=0

University of Colorado
Boulder

Monitors and Condition Variables

Deadlock when using Semaphores

- It can easily occur
  - Two tasks, each desires a resource locked by the other process
  - Circular dependency
  - Programming errors
    - Switching order of P() and V()
    - Calling wait multiple times
    - Forgetting a signal

Semaphores provide mutual exclusion, but can introduce deadlock

Monitors

- Semaphores can result in deadlock due to programming errors
  - forget to add a P() or V(), or misordered them, or duplicated them

- To reduce these errors, introduce high-level synchronization primitives, e.g. *monitors with condition variables*
  - essentially automates insertion of P() and V() for you
  - As high-level synchronization constructs, monitors are found in high-level programming languages (e.g. Java and C#)
  - underneath, the OS may implement monitors using semaphores and mutex locks

Monitors

- Declare a monitor as follows (looks somewhat like a C++ class):

```
monitor monitor_name {
      // shared local variables

      function f1(...) {
      ...
      }
      ...
      function fN(...) {
      ...
      }
      init_code(...) {
      ...
      }
   }
```

- A monitor ensures that only 1 process/thread at a time can be active within a monitor
  - simplifies programming, no need to explicitly synchronize
- Implicitly, the monitor defines a mutex lock

  semaphore mutex = 1;

- Implicitly, the monitor also defines mutual exclusion around each function
  - Each function's critical code is effectively:

```
function fj(...) {
   P(mutex)
   // critical code
   V(mutex)
}
```

Monitors

- Declare a monitor as follows (looks somewhat like a C++ class):

```
monitor monitor_name {
     // shared local variables
      int count;
      data_type data[MAX_COUNT];
     function f1(...) {
     ...
     }
     ...
     function fN(...) {
     ...
     }
     init_code(...) {
     ...
     }
  }
```

- The monitor's local variables can only be accessed by local monitor functions

- Each function in the monitor can only access variables declared locally within the monitor and its parameters

Monitors and Condition Variables

- Previous definition of a monitor achieves
  - mutual exclusion
  - hiding of wait() and signal() from user
  - loses the ability that semaphores had to enforce order
    - wait() and signal() are used to provide mutual exclusion
    - but have lost the unique ability for one process to signal another blocked process using signal()
    - there is no way to have a process sleep waiting on the signal

- In general, there may be times when one process wishes to signal another process based on a condition, much like semaphores.
  - Thus, augment monitors with *condition variables*.

Condition Variables

- Augment the mutual exclusion of a monitor with an ordering mechanism
  - Recall: Semaphore P() and V() provide both mutual exclusion and ordering
  - Monitors alone only provide mutual exclusion
- A condition variable provides ordering through Signaling
  - **Used when one (1st) task wishes to wait until a condition is true before proceeding**
    - Such as a queue being full to evict a data entry or data being ready to process
  - A **2nd task will signal the waiting task**, thereby waking up the waiting task to proceed

Monitors and Condition Variables

condition y;

A condition variable ***y*** allows three operations

- y.wait()
  - blocks the calling process
  - can have multiple processes suspended on a condition variable, typically released in FIFO order
  - another variation specifying a priority p,   i.e. call x.wait(p)
- y.signal()
  - resumes exactly 1 suspended process.
  - If no process is waiting, then function has *no effect*.
- y.queue()
  - Returns true if there is at least one process blocked on y

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
 boolean busy; // local variable
 condition x;   //condition variable
 void acquire(int time) {
           if (busy)
              x.wait(time); // queue a process here
           busy = TRUE;
 }
 void release() {
           busy = FALSE;
           x.signal();
 }
  initialization code() {
           busy = FALSE;
 }
}
```

Condition Variables Example

Block Task 1 until a condition holds true, e.g. queue is empty

```
condition wait_until_empty;
```

Task 1:

```
wait_until_empty.wait();
…// proceed after
            queue empty
```

Task 2:

```
…
// queue is empty so signal
wait_until_empty.signal();
```

Problem: If Task 2 signals before Task 1 waits, then Task 1 waits *forever* because CV *has no state*!

Condition Variables vs Semaphores

- Both have wait() and signal(), but semaphore’s signal()/V() preserves states in its integer value

```
Semaphore wait_until_empty=0;
```

Task 1:

```
wait(wait_until_empty);
…// proceed after
           queue empty
```

Task 2:

```
…
// queue is empty so signal
signal(wait_until_empty);
```

If Task 2 signals before Task 1 waits, then Task 1 does not wait forever because semaphore has state and remembers earlier signal()

Semaphores vs. Condition Variables

Semaphores

```
typedef struct {
    int value;
    PID *list[ ];
} semaphore;
```

Both wait() and signal() operations are atomic

```
wait(semaphore *s) {
    s→value--;
    if (s→value < 0) {
        add this process to s→list;
        sleep ( );
    }
}
```

```
signal(semaphore *s) {
    s→value++;
    if (s→value <= 0) {
        remove a process P from s→list;
        wakeup (P);
    }
}
```

Condition Variables

condition y;

A condition variable *y* in a monitor allows three operations

- y.wait()
  - blocks the calling process
  - can have multiple processes suspended on a condition variable, typically released in FIFO order
  - textbook describes another variation specifying a priority p, i.e. call x.wait(p)
- y.signal()
  - resumes exactly 1 suspended process.
  - If no process is waiting, then function has *no effect*.
- y.queue()
  - Returns true if there is at least one process blocked on y

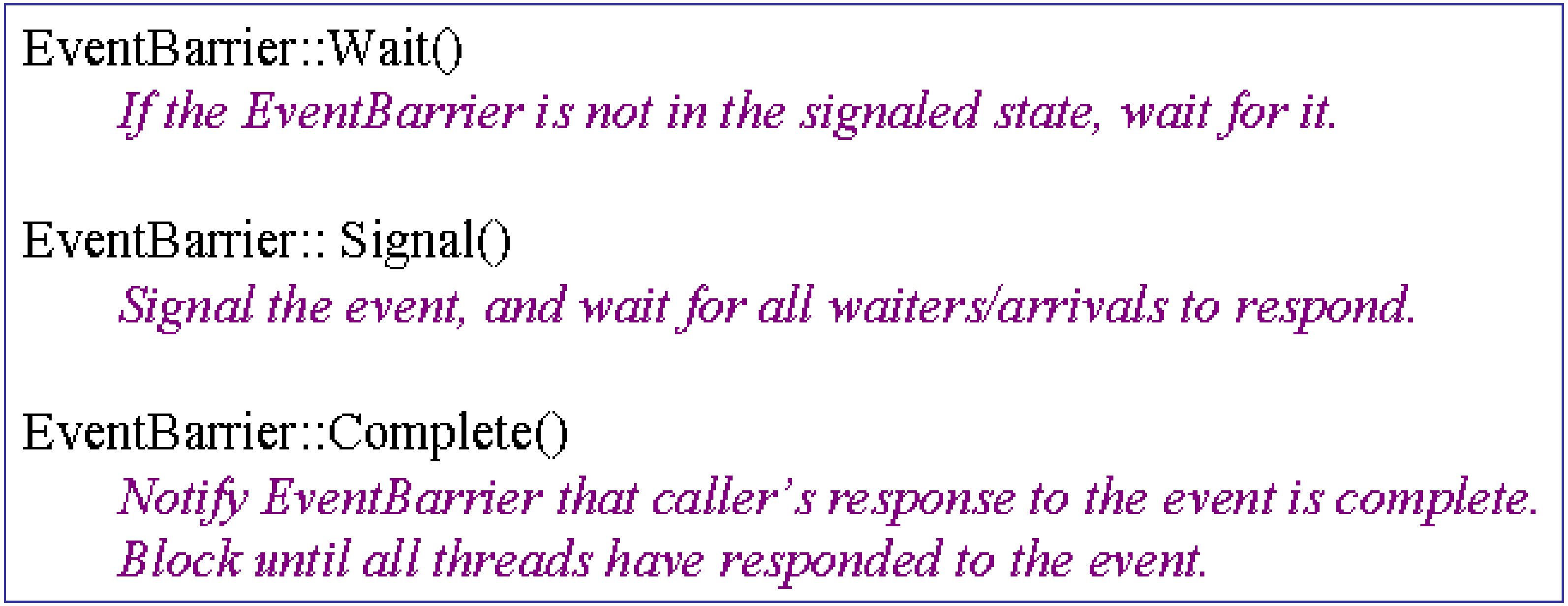
Semaphores vs. Condition Variables

- S.signal vs. C.signal
  - C.signal has no effect if no thread is waiting on the condition
    - Condition Variables are not variables (ha!) They have no value
  - S.signal has the same effect whether or not a thread is waiting
    - Semaphore retains a "memory"

- S.wait vs. C.wait
  - S.signal check the condition and block only if necessary
    - Programmer does not need to check the condition after returning from S.wait
    - Wait condition is defined internally but limited to a counter.
  - C.wait is explicit: It does not check the condition, ever
    - Condition is defined externally and protected by integrated mutex

Other Mechanisms

- EventBarrier
  - Combine semaphores and condition variables
  - Has binary memory and no associated mutex
  - Broadcast to notify all waiting threads
  - Wait until an event is handled



EventBarrier::Wait()

If the EventBarrier is not in the signaled state, wait for it.

EventBarrier:: Signal()

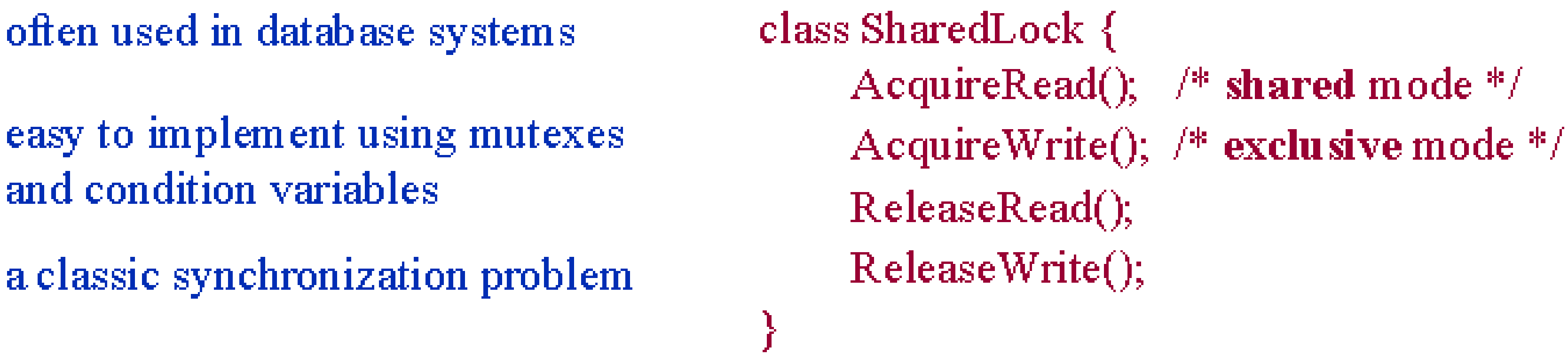
Signal the event, and wait for all waiters/arrivals to respond.

EventBarrier::Complete()

Notify EventBarrier that caller's response to the event is complete.
Block until all threads have responded to the event.

Other Mechanisms

- SharedLock: Reader/Writer Lock
  - Support Acquire and release primitives
  - Guarantee mutual exclusion when writer is present
  - Provides better concurrency for readers when no writer is present



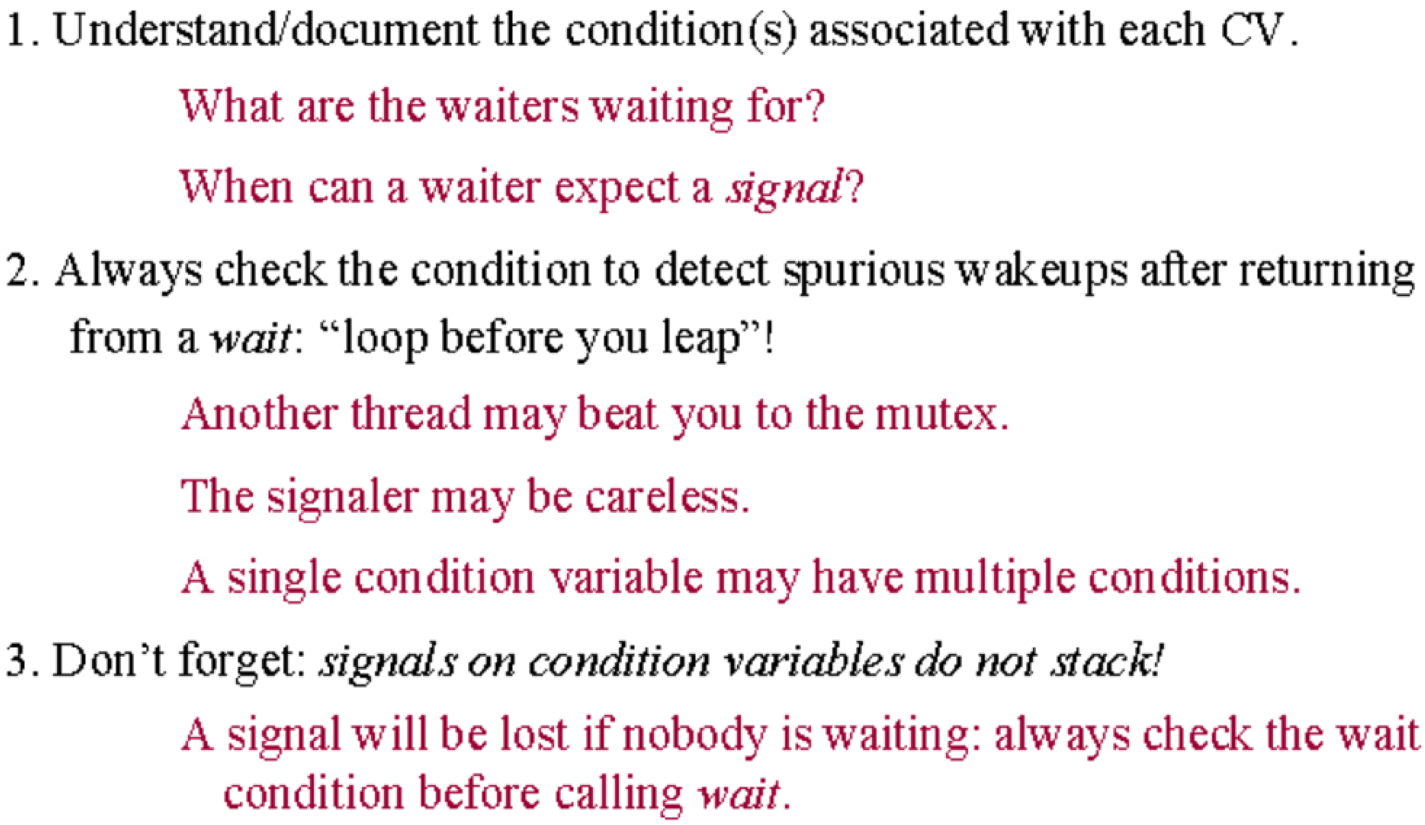
often used in database systems

easy to implement using mutexes and condition variables

a classic synchronization problem

```
class SharedLock {
        AcquireRead();   /* shared mode */
        AcquireWrite();  /* exclusive mode */
        ReleaseRead();
        ReleaseWrite();
}
```

Guidelines for Condition Variables



1. Understand/document the condition(s) associated with each CV.

   What are the waiters waiting for?

   When can a waiter expect a *signal*?

2. Always check the condition to detect spurious wakeups after returning from a *wait*: "loop before you leap"!

   Another thread may beat you to the mutex.

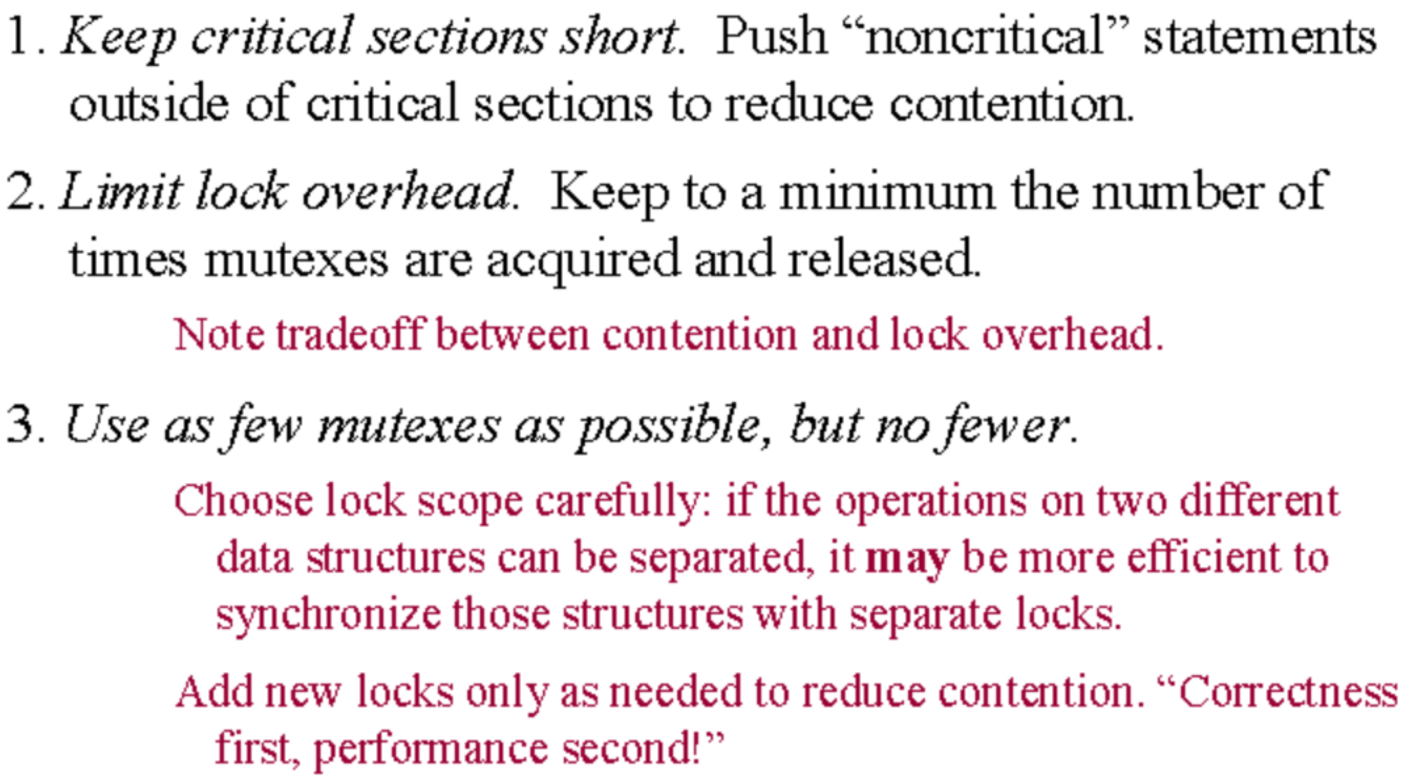
   The signaler may be careless.

   A single condition variable may have multiple conditions.

3. Don't forget: *signals on condition variables do not stack!*

   A signal will be lost if nobody is waiting: always check the wait condition before calling *wait*.

Guidelines for Choosing Lock Granularity



1. *Keep critical sections short.* Push "noncritical" statements outside of critical sections to reduce contention.

2. *Limit lock overhead.* Keep to a minimum the number of times mutexes are acquired and released.

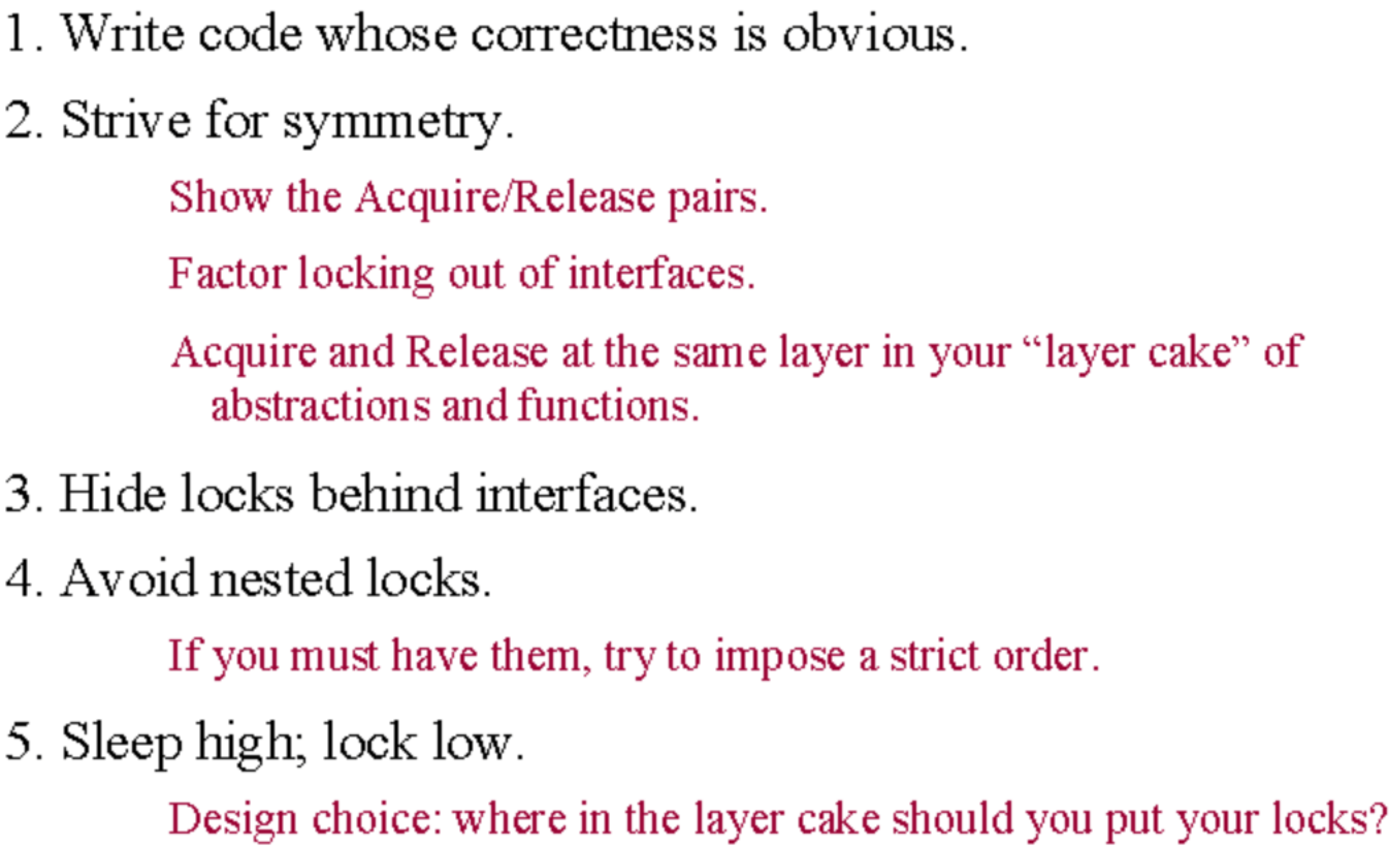
   Note tradeoff between contention and lock overhead.

3. *Use as few mutexes as possible, but no fewer.*

   Choose lock scope carefully: if the operations on two different data structures can be separated, it **may** be more efficient to synchronize those structures with separate locks.

   Add new locks only as needed to reduce contention. "Correctness first, performance second!"

More Guidelines for Locking



1. Write code whose correctness is obvious.
2. Strive for symmetry.

   Show the Acquire/Release pairs.

   Factor locking out of interfaces.

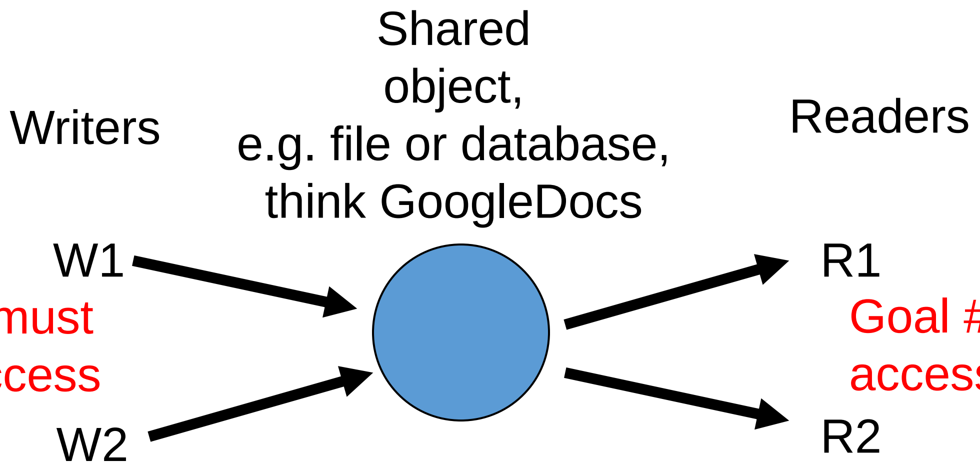
   Acquire and Release at the same layer in your "layer cake" of abstractions and functions.
3. Hide locks behind interfaces.
4. Avoid nested locks.

   If you must have them, try to impose a strict order.
5. Sleep high; lock low.

   Design choice: where in the layer cake should you put your locks?

The Readers/Writers Problem

- N tasks want to write to a shared file
- M other tasks want to read from same shared file
- Must synchronize access



Goal #1: a writer must have exclusive access

Goal #2: Readers can share access with other readers

- Goal #2: should support multiple concurrent readers
  - – Hence, a single mutex lock won’t support that

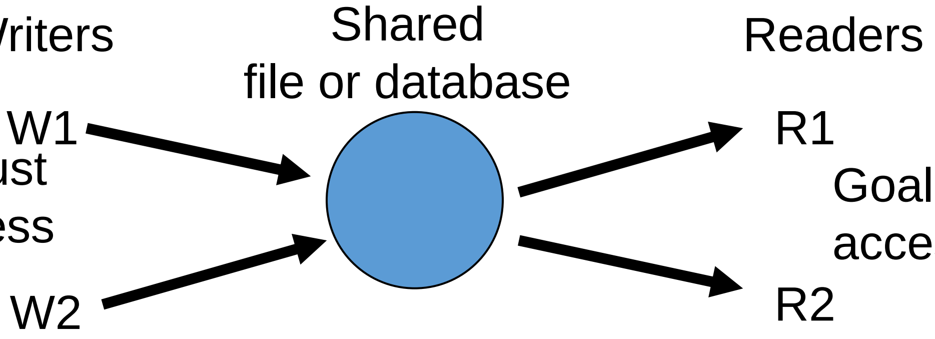
Readers/Writers Problems

- Additional requirement #1:
  - no reader is kept waiting unless a writer already has seized the shared object

- Additional requirement #2:
  - a pending writer should not be kept waiting indefinitely by readers that arrived after the writer
  - i.e. a pending writer cannot starve

1st Readers/Writers Solution

Assume *wrt_{init}*=1 is a mutex lock/binary semaphore



Goal #1: a writer must have exclusive access

Goal #2: Readers can share access with other readers

Writer:

```
while(1) {
     wait(wrt);   // Goal 1
       // writing
     signal(wrt);
}
```

Reader:

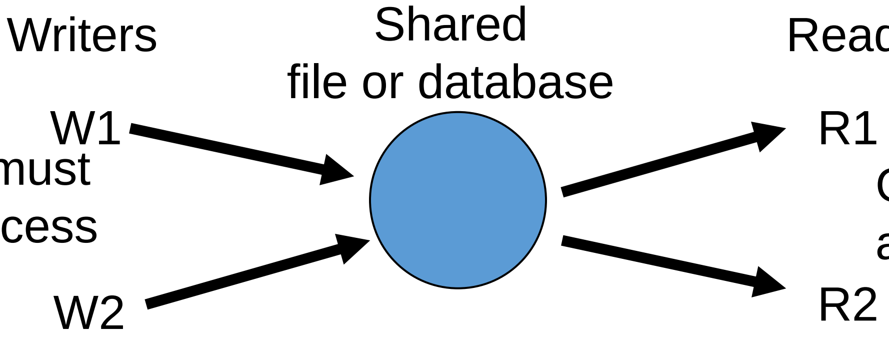
```
while(1) {
     ...
     wait(wrt);   // Goal 1 but not 2
       // reading
     signal(wrt);
     ...
}
```

Problem: first reader grabs lock, preventing other readers (& writers)

Solution: only the first reader needs to grab the lock,
and last reader release the lock.

1st Readers/Writers Solution (2)

Assume *wrt*$_{init}$=1, readcount initialized = 0



Goal #1: a writer must have exclusive access

Goal #2: Readers can share access with other readers

Writer:

```
while(1) {
     wait(wrt);
       // writing
     signal(wrt);
}
```

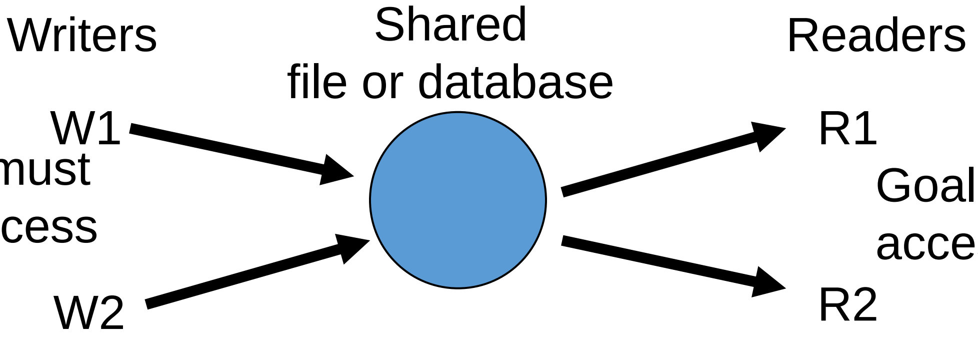
Reader:

```
while(1) {
     readcount++;
     if (readcount==1) wait(wrt);
       // reading
     readcount--;
     if (readcount==0) signal(wrt);
     ...
}
```

Problem: both readcount++ and readcount– lead to race conditions
Solution: surround access to **readcount** with a 2nd mutex

1st Readers/Writers Solution (3)

Assume *wrt*$_{init}$=1, readcount = 0, *mutex*$_{init}$=1



Goal #1: a writer must have exclusive access

Goal #2: Readers can share access with other readers

Writer:

```
while(1) {
     wait(wrt);
       // writing
     signal(wrt);
}
```

Reader:

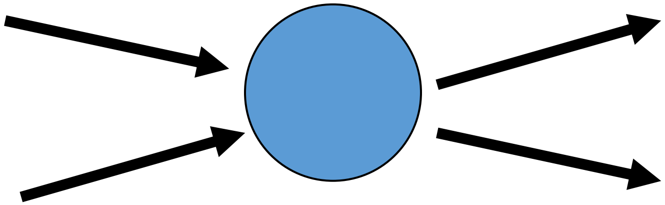
```
while(1) {
     wait(mutex)
     readcount++;
     if (readcount==1) wait(wrt);
     signal(mutex)
        // reading
     wait(mutex)
     readcount--;
     if (readcount==0) signal(wrt);
     signal(mutex)
}
```

So a writer excludes other writers and readers.
Multiple readers are allowed and exclude writers while at least 1 reader

1st Readers/Writers Solution (4)

Assume *wrt_{init}*=1, readcount = 0, *$mutex_{init}$*=1

Goal #1: a writer must have exclusive access



Writers

Shared
file or database

Readers

W1

W2

R1

R2

Goal #2: Readers can share access with other readers

Writer:

```
while(1) {
     wait(wrt);
       // writing
     signal(wrt);
}
```

Reader:

```
while(1) {
     wait(mutex)
     readcount++;
     if (readcount==1) wait(wrt);
     signal(mutex)
       // reading
     wait(mutex)
     readcount--;
     if (readcount==0) signal(wrt);
     signal(mutex)
}
```

Problem: this solution could starve pending writers!

2nd Readers/Writers Solution

A pending writer should not be kept waiting indefinitely by readers that arrived after the writer

- 1st R/W solution gave precedence to readers
    - new readers can keep arriving while any one reader holds the write lock, which can starve writers until the last reader is finished

- Instead, allow a pending writer to block future reads
    - This way, writers don't starve.
    - If there is a writer,
        - New readers should be blocked
        - Existing readers should finish then signal the waiting writer

Original Solution to the 2nd Readers/Writers Problem

```
int readCount = 0, writeCount = 0;
semaphore mutex1= 1, mutex2 = 1;
semaphore readBlock = 1, writeBlock = 1,
          writePending = 1;

writer() {
  while(TRUE) {

    P(mutex2);
      writeCount++;
      if(writeCount == 1)
        P(readBlock);
    V(mutex2);

    P(writeBlock);
      write(resource);
    V(writeBlock);

    P(mutex2)
      writeCount--;
      if(writeCount == 0)
        V(readBlock);
    V(mutex2);
  }
}
```

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

    P(writePending);
      P(readBlock);
        P(mutex1);
          readCount++;
          if(readCount == 1)
            P(writeBlock);
        V(mutex1);
      V(readBlock);
    V(writePending);

    read(resource);

    P(mutex1);
      readCount--;
      if(readCount == 0)
        V(writeBlock);
    V(mutex1);
  }
}
```

Red = changed from1st R/W problem solution

2nd Readers/Writers Starvation

- Once 1st writer grabs readBlock,
    - any number of writers can come through while the 1st reader is blocked on readBlock
    - and subsequent readers are blocked on writePending
    - So, behavior is that a writer can block not just new readers, but also some earlier readers
    - Note now that readers can be starved!

- Instead, want a solution that is starvation-free for both readers and writers

Starvation-free Solution to 2^{nd} R/W

Semaphore *wrt_{init}*=1, *$mutex_{init}$*=1, readBlock=1
int readcount = 0

Writer:

```
while(1) {
   wait(readBlock)
      wait(wrt);  // Goal 1
         // writing
      signal(wrt);
   signal(readBlock)
}
```

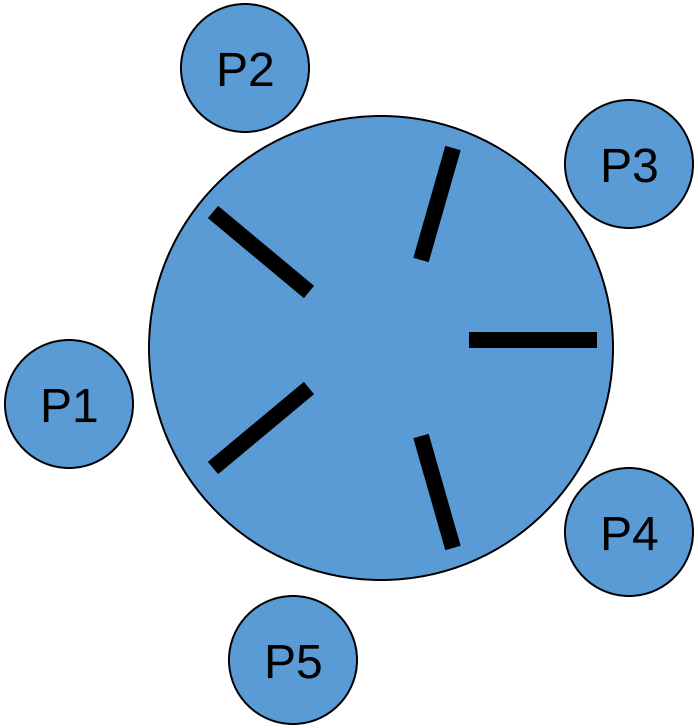
Reader:

```
while(1) {
      wait(readBlock)
         wait(mutex)
         readcount++;
         if (readcount==1) wait(wrt);
         signal(mutex)
      signal(readBlock)

          // reading

      wait(mutex)
      readcount--;
      if (readcount==0) signal(wrt);
      signal(mutex)
}
```

This is starvation-free solution
Note how it is a minor variant of the 1^{st} R/W solution.

Problem 3 - Dining Philosophers Problem

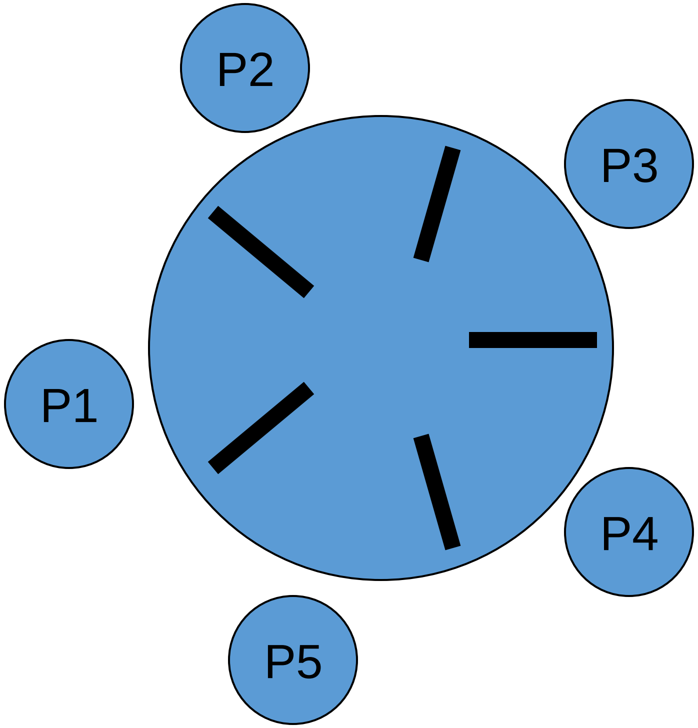
- Simple algorithm for protecting access to chopsticks:
  - `Access to each chopstick is protected by a mutual exclusion semaphore`
  - `Prevent any other phylosophers from pickup the choptick when it is already in use by a philosopher`
- **`Semaphore chopstick[5];`**
  - `Each philosohper grabs a chipstick I by` **`Wait(chopstick[i])`**
  - `Each philosopher release a chopstick I by` **`Signal(chopstick[i])`**



Dining Philosophers Problem

- Pseudo code for Philosopher **i**:

```
while(1) {
    // obtain 2 chopsticks to my
       immediate right and left
    P(chopstick[i]);
    P(chopstick[(i+1)%N];

    // eat

    // release both chopsticks
    V(chopstick[(i+1)%N];
    V(chopstick[i]);
}
```



Problem?

- Guarantees that no two neighbors eat simultaneously, i.e. a chopstick can only be used by one its two neighboring philosophers

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously:

- **Mutual exclusion**: only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait**: there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
  - Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

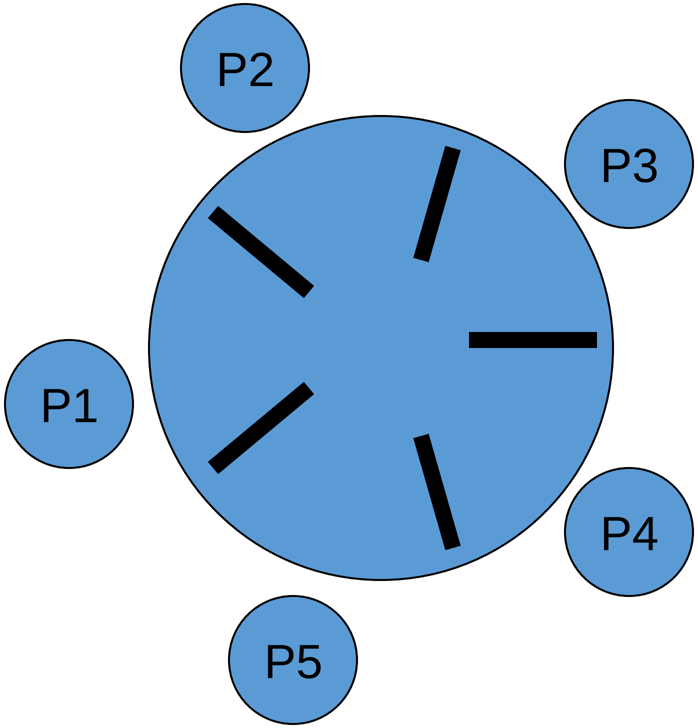
- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released - **All or nothing**
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Dining Philosophers Problem

- Possible Deadlock-free solutions are:
  - allow at most 4 philosophers at the same table when there are 5 resources
  - odd philosophers pick first left then right, while even philosophers pick first right then left
  - allow a philosopher to pick up chopsticks *only if both are free.*
    - This requires protection of critical sections to test if both chopsticks are free before grabbing them.
    - We'll see this solution next using monitors
- A deadlock-free solution is not necessarily starvation-free
  - for now, we'll focus on breaking deadlock

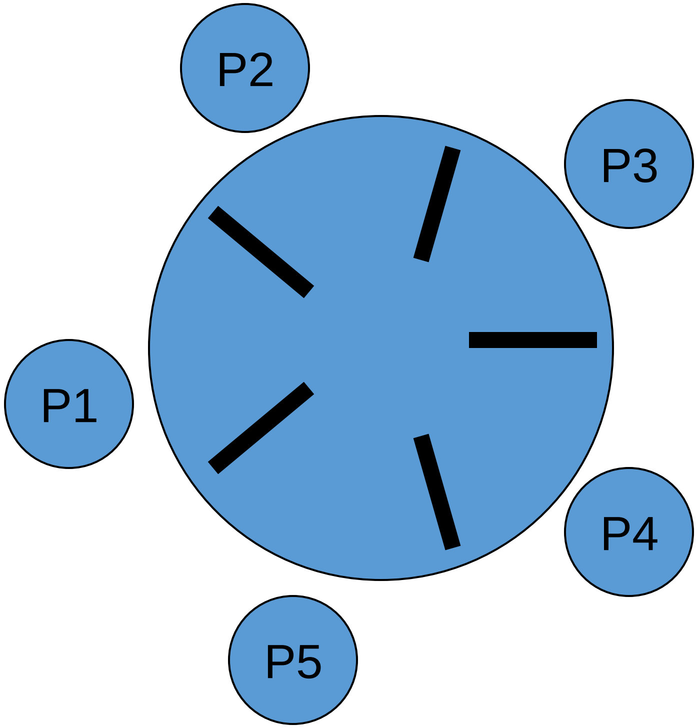
Monitor-based Solution to Dining Philosophers

- Key insight: Pick up 2 chopsticks only if both are free
  - this avoids deadlock
  - **reword insight**: a philosopher moves to his/her eating state only if both neighbors are not in their eating states
  - thus, need to define a state for each philosopher – What are the states?



Monitor-based Solution to Dining Philosophers (2)

- 2nd insight: if one of my neighbors is eating, and I'm hungry, ask them to signal() me when they're done
  - thus, states of each philosopher are: thinking, hungry, eating
  - thus, need condition variables to signal() waiting hungry philosopher(s)
- Also need to Pickup() and Putdown() chopsticks



Dining Philosophers: Monitor-based Solution

```
philosopher (int i)
{
  while (1) {
    //Think
    DiningPhilosophers.pickup(i);
    // pick up chopsticks and eat
    DiningPhilosophers.putdown(i);
  }
}
```

Dining Philosophers: Monitor-based Solution

```
monitor DP
{
            enum {thinking, hungry, eating} state[5]; // maintain philosopher's state
            condition self[5]; //to block a philosopher when hungry

            void pickup(int i) {
                - Set state[i] to hungry
                - If at least one neighbor is eating,  (test the neighbors)
                          block on self[i]
                - Otherwise return
            }

            void putdown(int i) {
                - Change state[i] to thinking
                 - Signal neighbors who are waiting to eat
            }
        …
        void test(int p) {
          +Check if  both neighbors of p are not eating and p is hungry
          +If so,
            - set state[p] to eating
             - and signal philosopher p
        }

        init( ) {
           for (int i = 0; i < 5; i++)
              state[i] = thinking;
        }
}
```

Monitor-based Solution to Dining Philosophers (3)

```
monitor DP {
      status state[5];
      condition self[5];
      Pickup(int i);
      Putdown(int i);
      test();
     init();
   }
```

- Each philosopher *i* runs pseudo-code:

```
DP.Pickup(i);
 ... // eat - grab both
        chopsticks
DP.Putdown(i);
```

Monitor-based Solution to Dining Philosophers (4)

```
monitor DP {
   status state[5];
   condition self[5];

   Pickup(int i) {
```

atomic

```
      state[i] = hungry;
      test(i);
      if(state[i]!=eating)
         self[i].wait;
   }
```

atomic

```
   test(int i) {
      if (state[(i+1)%5] != eating &&
         state[(i-1)%5] != eating &&
         state[i] == hungry) {

         state[i] = eating;
         self[i].signal();
      }
   }
... monitor code continued next slide ...
```

- Pickup chopsticks (atomic)
  - – indicate that I'm hungry
  - – Atomically test if both my left and right neighbors are not eating. If so, then atomically set my state to eating.
  - – if unable to eat, wait to be signaled

- signal() has no effect during Pickup(), but is important to wake up waiting hungry philosophers during Putdown()

Monitor-based Solution to Dining Philosophers (5)

… monitor code continued from previous slide…

atomic

```
...

Putdown(int i) {
        state[i] = thinking;
        test((i+1)%5);
        test((i-1)%5);
    }


    init() {
        for i = 0 to 4
            state[i] = thinking;
    }

} // end of monitor
```

- Put down chopsticks (atomic)
  - – if left neighbor L=(i+1)%5 is hungry and both of L's neighbors are not eating, set L's state to eating and wake it up by signaling L's CV
- Thus, **eating philosophers are the ones who (eventually) turn waiting hungry neighbors into active eating philosophers**
  - not all eating philosophers trigger the transformation
  - At least one eating philosophers will be the trigger

Complete Monitor-based Solution to Dining Philosophers

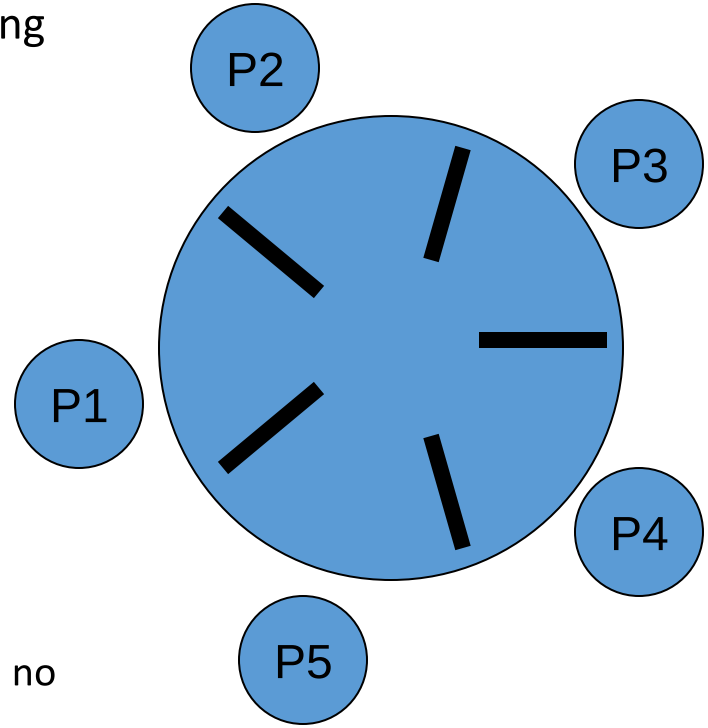
```
monitor DP {
   status state[5];
   condition self[5];

   Pickup(int i) {
      state[i] = hungry;
      test(i);
      if(state[i]!=eating)
         self[i].wait;
   }

   test(int i) {
      if (state[(i+1)%5] != eating &&
         state[(i-1)%5] != eating &&
         state[i] == hungry)
        {
         state[i] = eating;
         self[i].signal();
      }
   }
```

```
Putdown(int i) {
      state[i] = thinking;
      test((i+1)%5);
      test((i-1)%5);
   }

   init() {
      for i = 0 to 4
         state[i] = thinking;
   }
}  // end of monitor
```

- Pickup(), Putdown() and test() are all mutually exclusive, i.e. only one at a time can be executing
- Verify that this monitor-based solution is
  - deadlock-free
  - mutually exclusive in that no 2 neighbors can eat simultaneously

DP Monitor Deadlock Analysis

- Try various scenarios to verify for yourself that deadlock does not occur in them
- Start with one philosopher P1
- Now suppose P2 arrives to the left of P1 while P1 is eating
  - What is the perspective from P1?
  - What is the perspective from P2?
- Now supposes P5 arrives to the right of P1 while P1 is eating and P2 is waiting
  - Perspective from P1?
  - Perspective from P5?
  - Perspective from P2?
- Suppose P2 arrives while both P1 and P3 are eating
  - If P1 finishes first, it can't wake up P2
  - But when P3 finishes, its call to test(P2) will wake up P2, so no deadlock
- Suppose there are 6 philosophers and the evens are eating. How do the odds get to eat?



DP Monitor Solution

- Note that starvation is still possible in the DP monitor solution
  - Suppose P1 and P3 arrive first, and start eating, then P2 arrives and sets its state to hungry and blocks on its CV
  - When P1 stops eating, it will call test(P2), but nothing will happen, i.e. P2 won't be signaled because the signal only occurs inside the if statement of test, and the if condition is not satisfied
  - Next, P1 can eat again, repeatedly, starving P2

Complete Monitor-based Solution to Dining Philosophers

```
monitor DP {
   status state[5];
   condition self[5];

   Pickup(int i) {
      state[i] = hungry;
      test(i);
      if(state[i]!=eating)
         self[i].wait;
   }

   test(int i) {
      if (state[(i+1)%5] != eating &&
         state[(i-1)%5] != eating &&
         state[i] == hungry) {

         state[i] = eating;
         self[i].signal();
      }
   }
```

```
Putdown(int i) {
      state[i] = thinking;
      test((i+1)%5);
      test((i-1)%5);
   }

   init() {
      for i = 0 to 4
         state[i] = thinking;
   }
}  // end of monitor
```

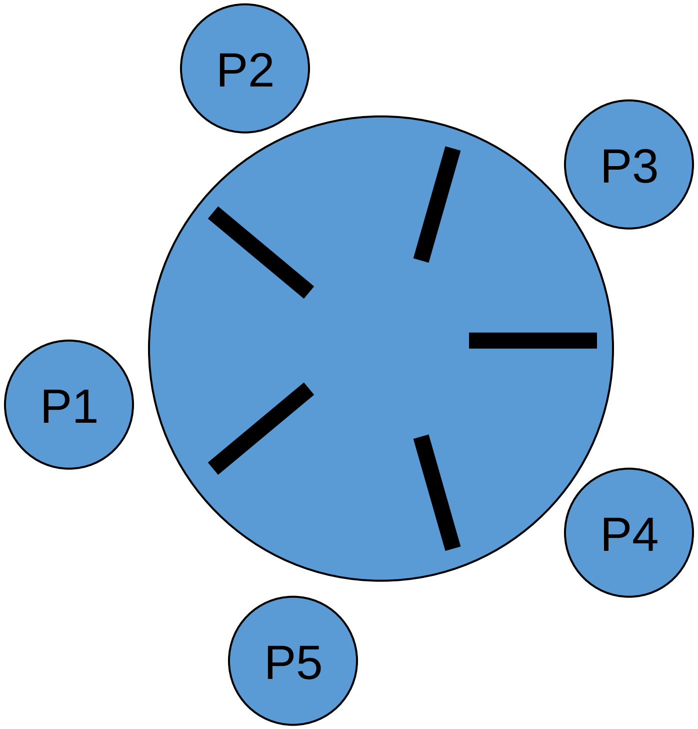
DP Solution Analysis

- Signal() happening before the wait():
  - Signal() in Pickup() has no effect the 1st time
  - Signal() called in Putdown() is the actual wakeup

Deadlock Free Solution

- Monitors for implicit mutual exclusion
- Condition variables for ordering
  - cv.wait()
  - cv.signal() differs from semaphore's signal()

- Monitor-based solution to Dining Philosophers


P2
P3
P1
P4
P5