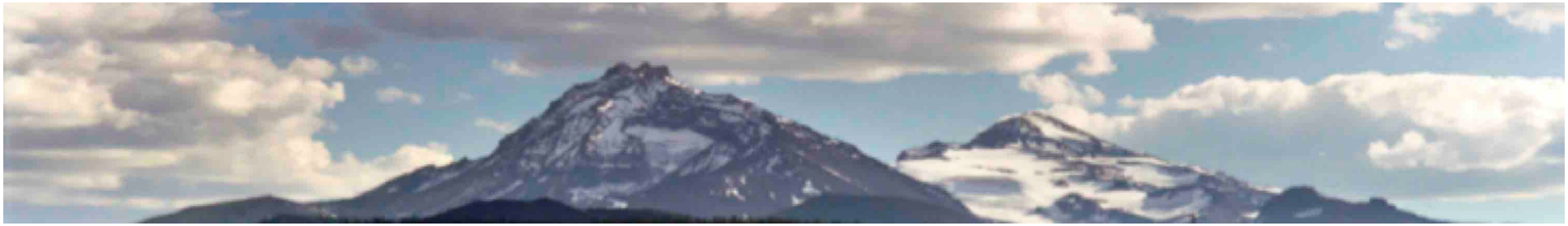


# CS 415
# Operating Systems
# Concurrency and Synchronization

Prof. Jieyang Chen
Department of Computer and Information Science
Fall 2025

UNIVERSITY OF OREGON

# *Logistics*

- Project 2 due extended to Wednesday
- Project 3 will be released on Friday (threads+sync.)
- Read Chapters 6 & 7 (this week) and 8 (next week)
- Next week's lectures switched to Zoom-only
  - Yuhang and I will be traveling
  - Both Tue.(11/18) and Thru.(11/20) will be Zoom-only
  - Lecture time remains unchanged (4-5:20 pm)
  - Labs remain in person (Nate will lead all sessions)
  - No OH for me and Yuhang
  - We will respond to emails

# *Midterm*

- Midterm grades are out
- Most of you did well!
- If your grade is lower than expected
  - Please don't get discouraged
  - Talk to me and can make a plan
  - Other components are important too
    - Lab 5% + project 45% + final 28%
- If you get ≥ 84, you can expect a A- or better
- If you get ≥ 74, you can expect a B- or better
- If you get ≥ 40, you can expect a pass

# *Midterm Question Review*

- Review common mistakes in midterm
- Lecture schedule is tight
- Have to be outside lecture time, sorry
- **Midterm review: Thursday (11/20)**
  - **5:20pm - 6:20pm on Zoom (Immediately after lecture)**
- Attendance is optional
- Recording will be provided

# *Outline*

- Background
- Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutual Exclusion (Mutex) Locks
- Semaphores
- Monitors
- Classic Problems of Synchronization

# *Objectives*

- To present the concept of process synchronization
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

# *Roadmap: Concurrency and Synchronization*

## *Concurrent Applications*

## *Shared Objects*

Bounded buffer  Barrier

## *Synchronization Variables*

Semaphores  Locks  Condition Variables

## *Atomic Instructions*

Interrupt disable  Test-and-Set  Compare-and-Swap

## *Hardware*

Multiple processors  Hardware interrupts

# *Roadmap: Concurrency and Synchronization*

## *Concurrent Applications*

## *Shared Objects*

Bounded buffer  Barrier

## *Synchronization Variables*

Semaphores  Locks  Condition Variables

## *Atomic Instructions*

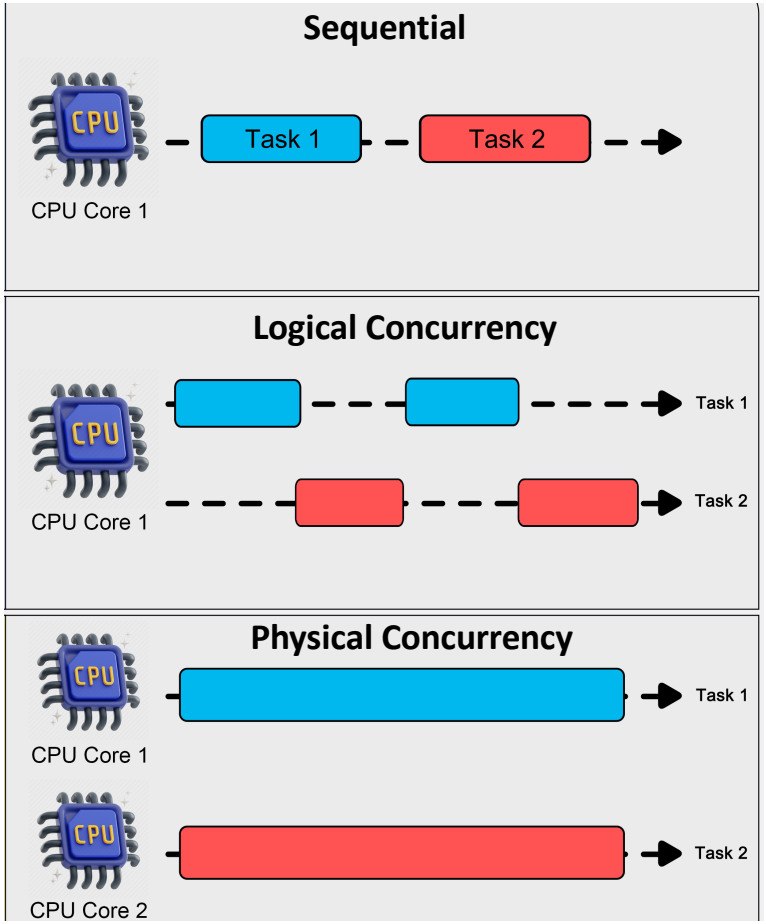
Interrupt disable  Test-and-Set  Compare-and-Swap

## *Hardware*

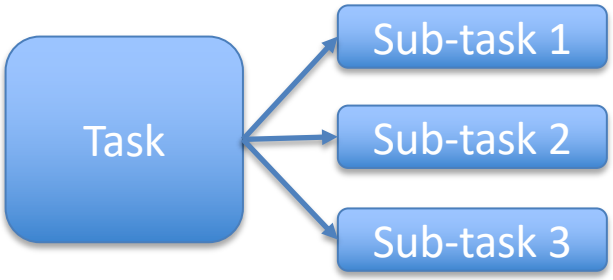
Multiple processors  Hardware interrupts

# *Review: Concurrency*

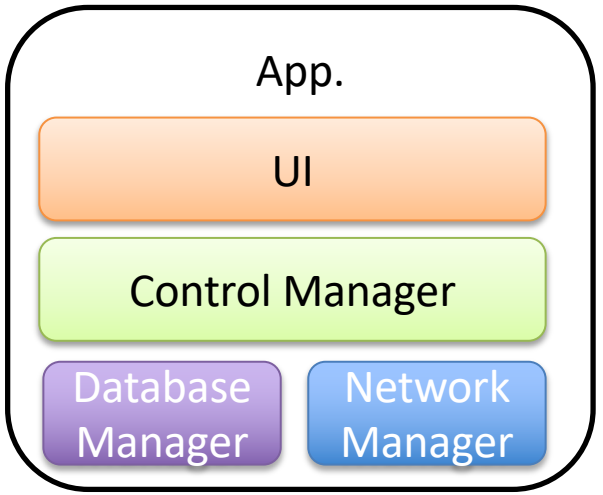
- *Concurrency* -- multiple things happening (being able to happen) "*at the same time*"
- *logical* vs. *physical*
  - *Logical concurrency:* multiple tasks that are capable of executing
    - Does not say anything about the computing resources
    - Not all tasks are guaranteed to be executed simultaneously.
  - *Physical concurrency:* multiple tasks are actually executing
    - Specifically, multiple tasks running simultaneously on CPU cores
    - Default physical concurrency is when 1 task is running on a CPU



# *Why do we care about concurrency?*




Parallelization: accelerate computation




Separation of concerns (SOC): Different processes can do things at the same "logical" time

OS must function as a concurrent system … Why?

How to achieve concurrency?

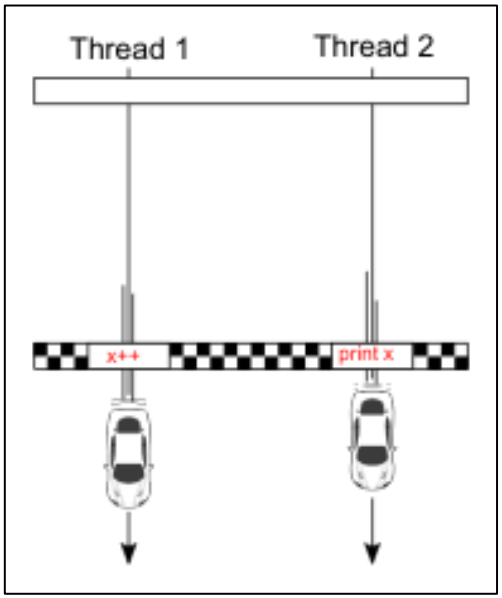
# *Background: Problems of Concurrency*

- Early operating systems research was where fundamental problems of *concurrency* first arose
- E.W. Dijkstra, "*Cooperating Sequential Processes*," Technical Report, TU Eindhoven, the Netherlands,1965 (see link to paper on schedule)
  - Introduced the *critical section problem*
  - Dijkstra was the 1972 Turing Award winner!
- Concurrent systems must address the problems of how concurrent processes work consistently together

https://en.wikipedia.org/wiki/Edsger_W._Dijkstra

# *Problems of Concurrency*

- What might happen to concurrent processes/threads?
  - Some might cooperate during execution
  - Anyone might be interrupted at any time
  - Some might share resources:
    - Physical (memory, terminal, disk, network, …)
    - Logical (files, sockets, data, …)
- Access to shared resources must be coordinated
  - Otherwise, unexpected errors could arise
- How to coordinate processes/threads?
  - Synchronization

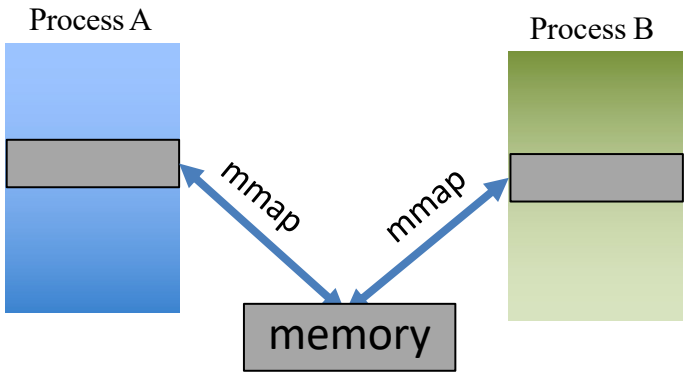


Data race problem

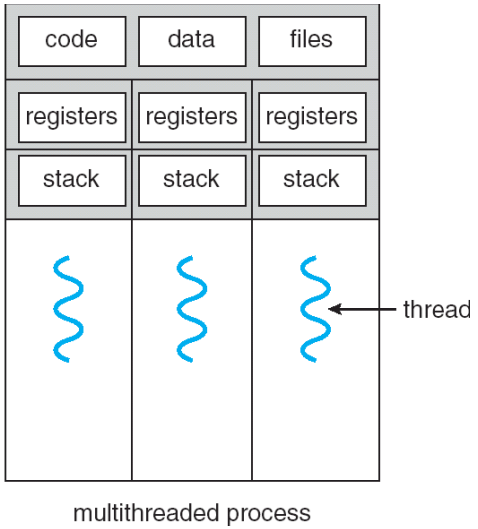
# *Processes (Threads) Sharing Memory*

- Let us focus on "*memory*" as the shared resource

How to share memory? (Think about what we learned)




IPC: shared memory




Memory is shared by default between threads

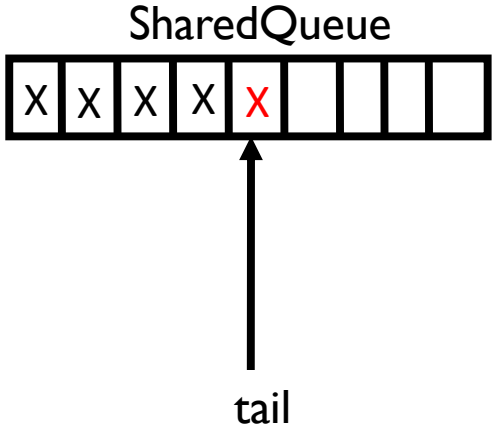
Concurrency problems exist regardless of how resources are shared

# *Example: Shared Queue*

- Consider a shared printer queue
  - *SharedQueue[N]*
  - *tail* is the index designating the current end of queue
  - Both *SharedQueue* and *tail* are shared between processes
- Consider 2 processes (can also be threads)
  - Each wants to enqueue an element to this queue
- Each process needs to do

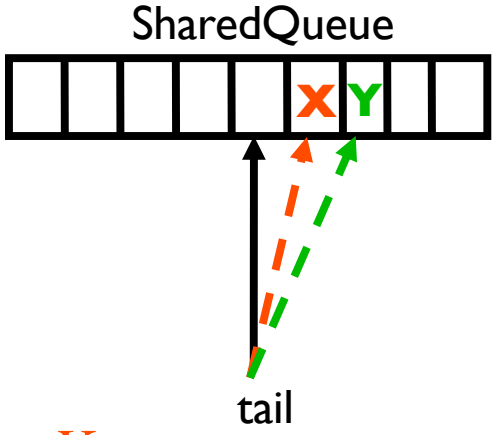
  *tail = tail + 1;*

  *SharedQueue[tail] = "element";*



# *What are we trying to do?*

- Want to have the "correct" execution




**Process 1 (P1)**

*tail = tail + 1;*
*SharedQueue[tail] = X*

**Process 2 (P2)**

*tail = tail + 1;*
*SharedQueue[tail] = Y*

If we don't care about the order of X and Y, what could go wrong?

- Remember, Process 1 and 2 are "concurrent"
  - They are executing their instructions at the same time
  - One instruction ≠ One statement

# *What is the problem?*

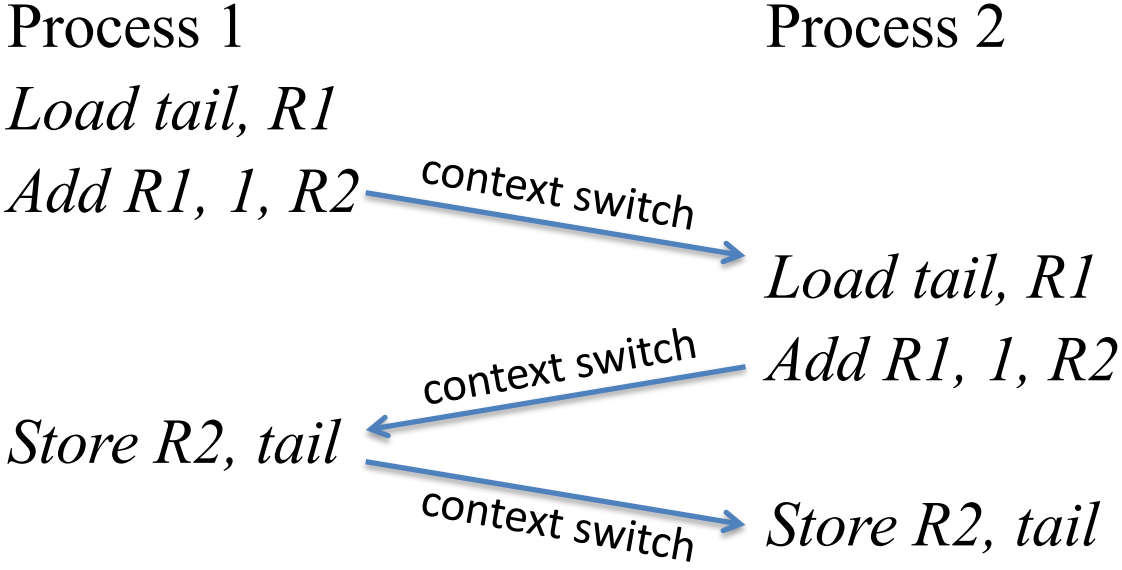
- *tail = tail + 1* is NOT a single machine instruction
  - So, what? Why do we care?
- What assembly code does the compiler produce?

```
Load tail, R1       % read the value of tail from memory to R1
Add R1, 1, R2       % add 1 to R1 and put in R2
Store R2, tail      % write R2 to tail in memory
```

What happens if P1 and P2 execute these instructions concurrently?

# *Instruction Interleaving*

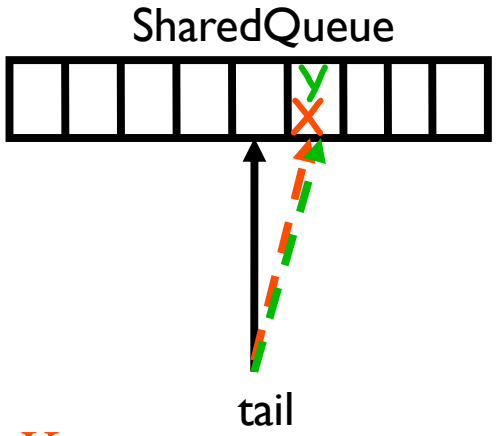
- Each process executes the set of 3 instructions
- Interrupts might happen at any time!
  - Thus, a context switch can happen at any time
  - Uh, ok, so what?
- Suppose we have the following scenario:




See any problems?

# *Leading to …*

- Incorrect execution




**Process 1**

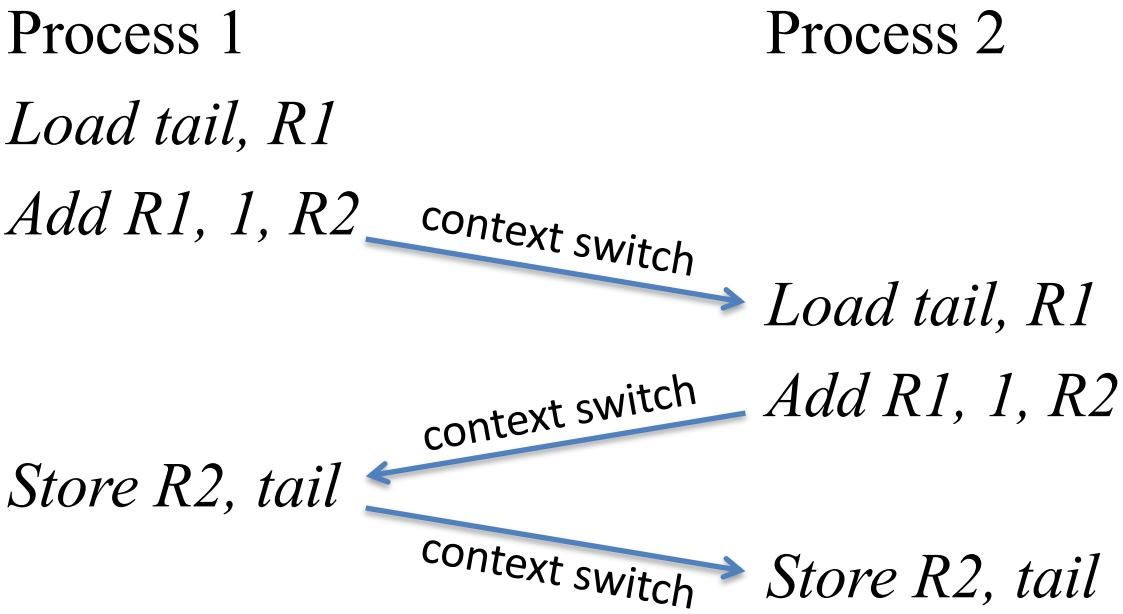
*tail = tail + 1;*
*SharedQueue[tail] = X*

**Process 2**

*tail = tail + 1;*
*SharedQueue[tail] = Y*

- This is not what we want
- Hmm, how can we fix it?

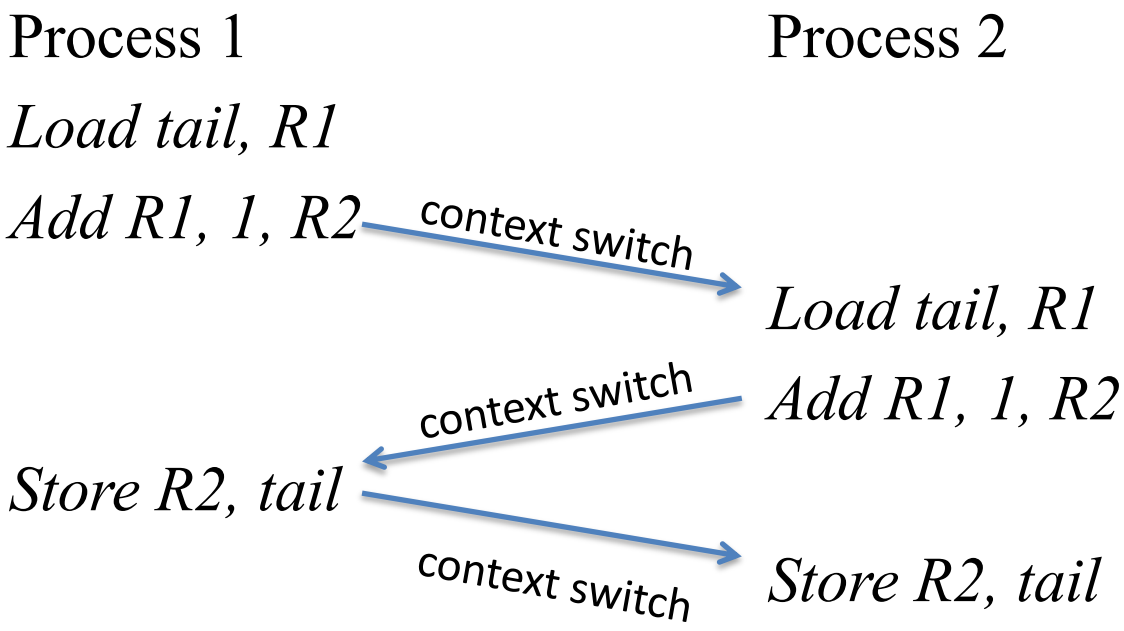
# *Instruction Interleaving*




Will the problem be resolved if processes 1 and 2 use different sets of registers?

# *Race Conditions*

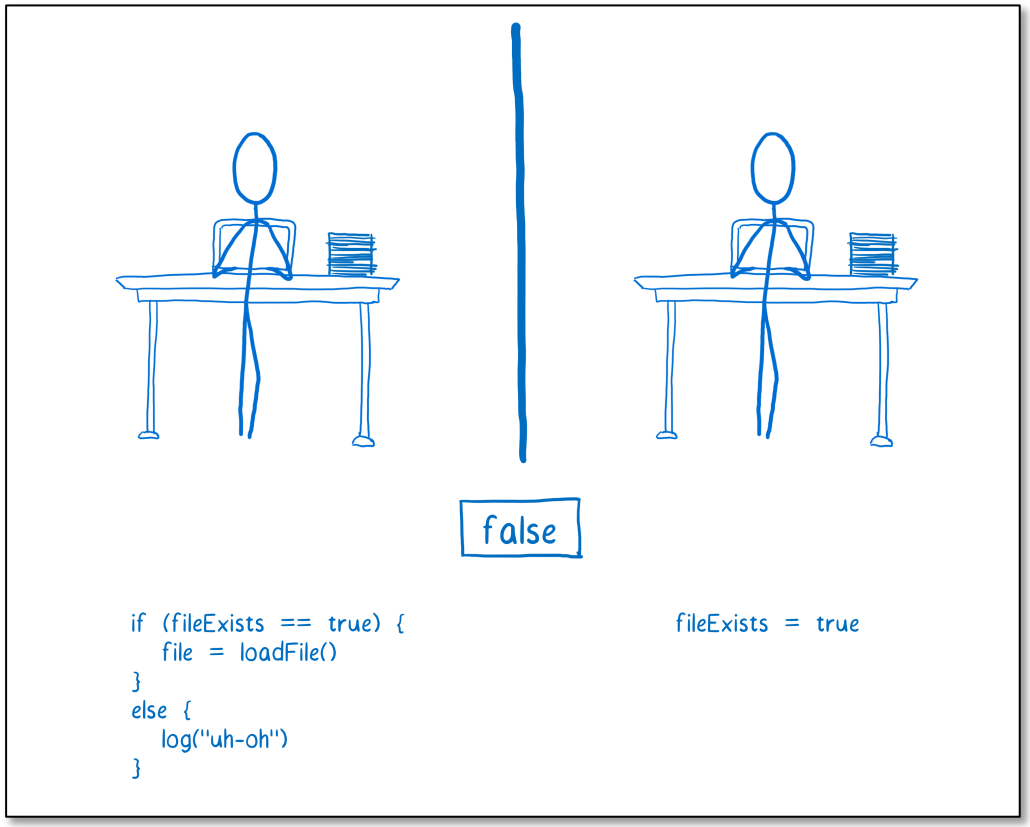
**Definition:** a *race condition* in concurrent execution is when the outcome of the execution depends on the particular interleaving of concurrent instructions



# *Race Conditions*

Debugging can be challenging

- Race conditions are timing-dependent
- Errors can be non-repeatable



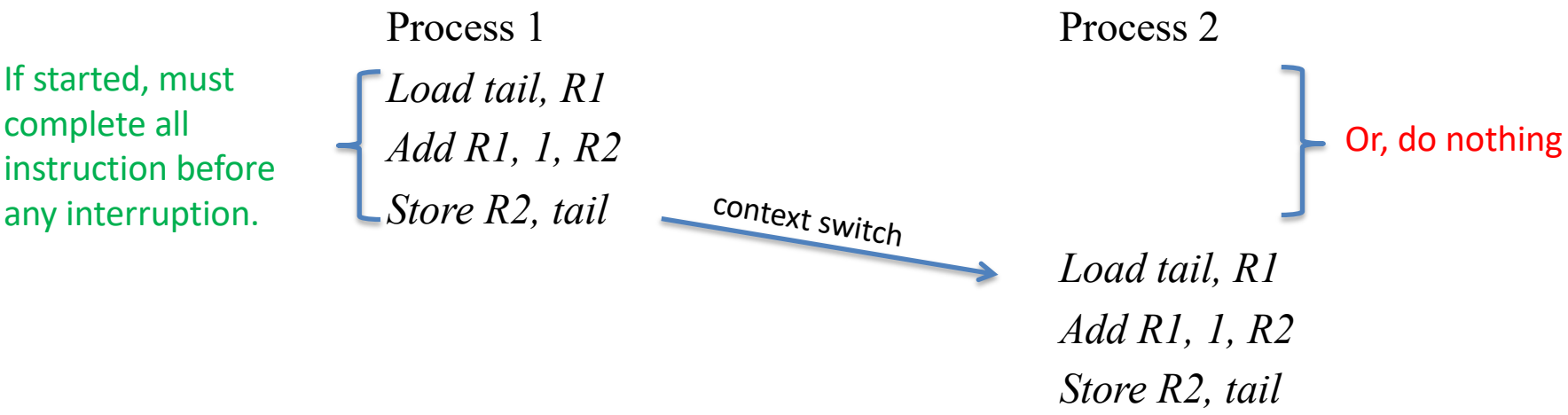

Does it mean that, because of a race condition, the output WILL be incorrect?

# *Ok, how do we avoid race conditions?*

**Definition** of Atomic

A set of instructions is *atomic* if it is executed *as if* it were a single instruction.

- What does this mean exactly?
  - Only two possible outcomes: do nothing OR do all
  - No possible to be interrupted when partially done



# *Does atomic avoid race conditions?*

- Depends on the scope
  - Yes, for a single word/variable
  - No, for multiple statements or function calls
- Why? Hardware restriction: atomic is typically achieved at the CPU instruction level, which applies only for operating on a variable

Each statement can be atomic if we use special HW instructions

```
void Dequeue(int N, int * ret) {
    for (int i = 0; i < N; ++i) {
        ret[i] = SharedQueue[tail];
        tail = tail - 1;
    }
}
```

/* Assume we want to dequeue N consecutive elements*/

But the whole function is NOT atomic (no HW instruction)

***Why are we learning atomic?***
Atomic is the building block of higher-level synchronization tools!

# *Critical Section Problem (Dijkstra, 1965)*

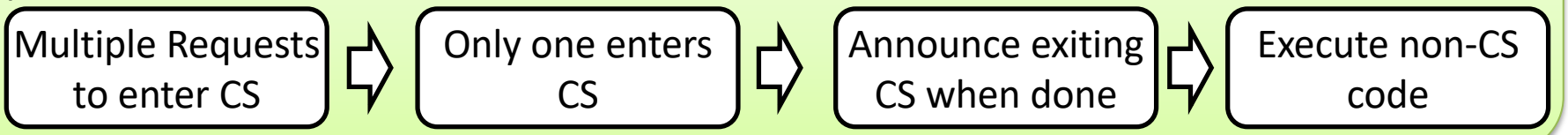
**Definition** of Critical Section (CS)

Given a system of *n* processes {$P_0$, $P_1$, … $P_{n-1}$}. A Critical Section is a segment of code in each process that:

- Access shared variables/data structures/files…
- At least one process is updating the shared entity in the critical section
- Has mutual exclusion: When one process is in its critical section, no other may be in its critical section

**Definition** of Critical Section Problem

*Critical section (CS) problem* is to design a *protocol* (rules) between the processes to enable:



# *Critical Section (CS) Pseudo Code*

- General structure of process $P_i$

```
do {
    code before entry (program does other things)
    code to enter the critical section
    CRITICAL SECTION
    code to exit from critical section
    code after exit (program does other things)
} while (true);
```

entry section

Only 1 process can be in critical section at a time

exit section

It is the entry and exit code that defines the critical section protocol.

The infinite "do" loop is just to suggest that a process will possibly want to enter its critical section multiple #s of times, including never (0 times) or just once (1 time).

# *Requirements for Solution to CS Problem*

- *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 there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
- *Bounded waiting*
  - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
  - Assume that each process executes at a nonzero speed
  - No assumption concerning relative speed of the *N* processes

# *Translate into a more understandable language…*

- *Mutual exclusion*

  Only one process may enter the critical section at a time

- *Progress*

  If no one is inside and someone wants in, the decision of who gets in should be made quickly (not delayed by irrelevant processes).

- *Bounded waiting*

  When a process asks to enter, it won't be stuck waiting forever — there's a maximum number of turns others can go before it gets its chance.

# *Roadmap: Concurrency and Synchronization*

## *Concurrent Applications*

## *Shared Objects*

Bounded buffer  Barrier

## *Synchronization Variables*

Semaphores  Locks  Condition Variables

## *Atomic Instructions*

Interrupt disable  Test-and-Set  Compare-and-Swap

## *Hardware*

Multiple processors  Hardware interrupts

# *How to Implement Critical Sections*

- Implementing critical section solutions follows three fundamental approaches

1. Disable Interrupts
   - Effectively stops the scheduling of other processes
   - Does not allow another process to get the CPU
   - Very disruptive and should not allow a process to do this
2. Busy-wait (spinlock) solutions
   - Pure software solutions
   - Integrated hardware-software solutions
3. Blocking solutions

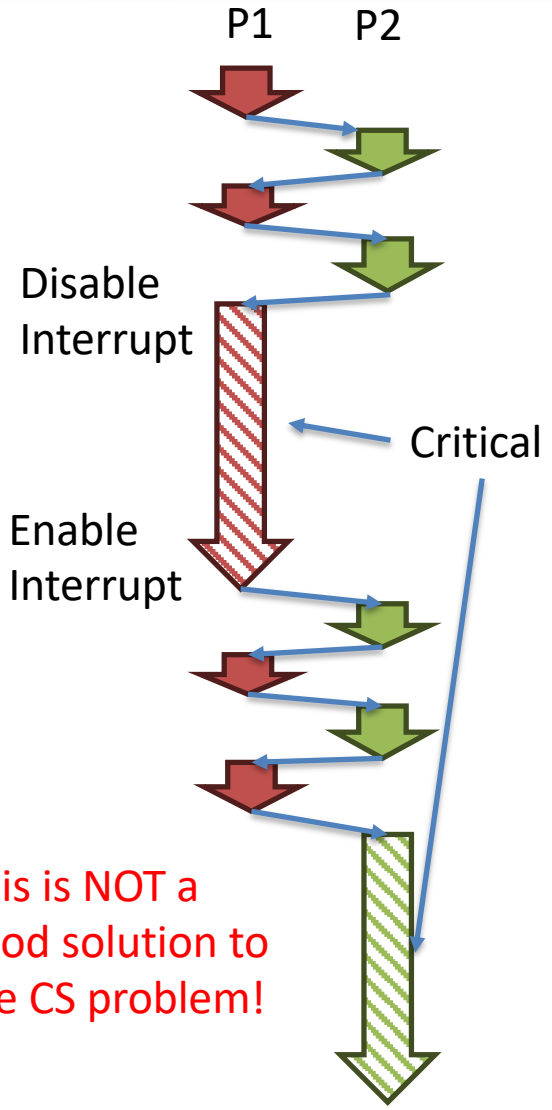
- In each case, think about whether the critical section solution requirements are being met

# *Approach 1: Disabling Interrupts*

```
do {
    code before entry (program does other things)
    Disable Interrupt

      CRITICAL SECTION

    Enable Interrupt
    code after exit (program does other things)
} while (true);
```




Does not work on a multiprocessor … Why?

Disables multiprogramming even if another process is NOT interested in critical section … Why

This is NOT a good solution to the CS problem!

# *Note: WHILE Loops*

- In the following slides, WHILE loops with empty statements are used to mean infinite loops until the WHILE condition is met
- The following are equivalent:

```
while (lock == TRUE) {
    ;
}
while (lock == TRUE)
    ;
while (lock == TRUE) ;
```

; represents a NULL instruction

# *Caveat: Can only use software for a solution*

- In 1965, computer hardware was very primitive compared to today
- There was no hardware to help with developing solutions to the critical section problem
  - We will discuss this later
- All solutions had to be implemented with CPU instructions that did read or write operations
- For sake of discussion, let's assume that all variables and data structures are in memory only
  - That is, do not worry about being in registers or cache

# *Approach 2: Busy-Waiting (aka Spinning)*

- Overall philosophy:
  - Keep checking some shared state (variables) until they indicate other process(es) are not in critical section

```
locked = FALSE; // initial value (shared)
```

other code before/after entry/exit

```
P1 do {
   ...
   while (locked == TRUE)
      ;
   locked = TRUE;

   /************
   (critical section code)
   /************

   locked = FALSE;
   ...
} while (true);
```

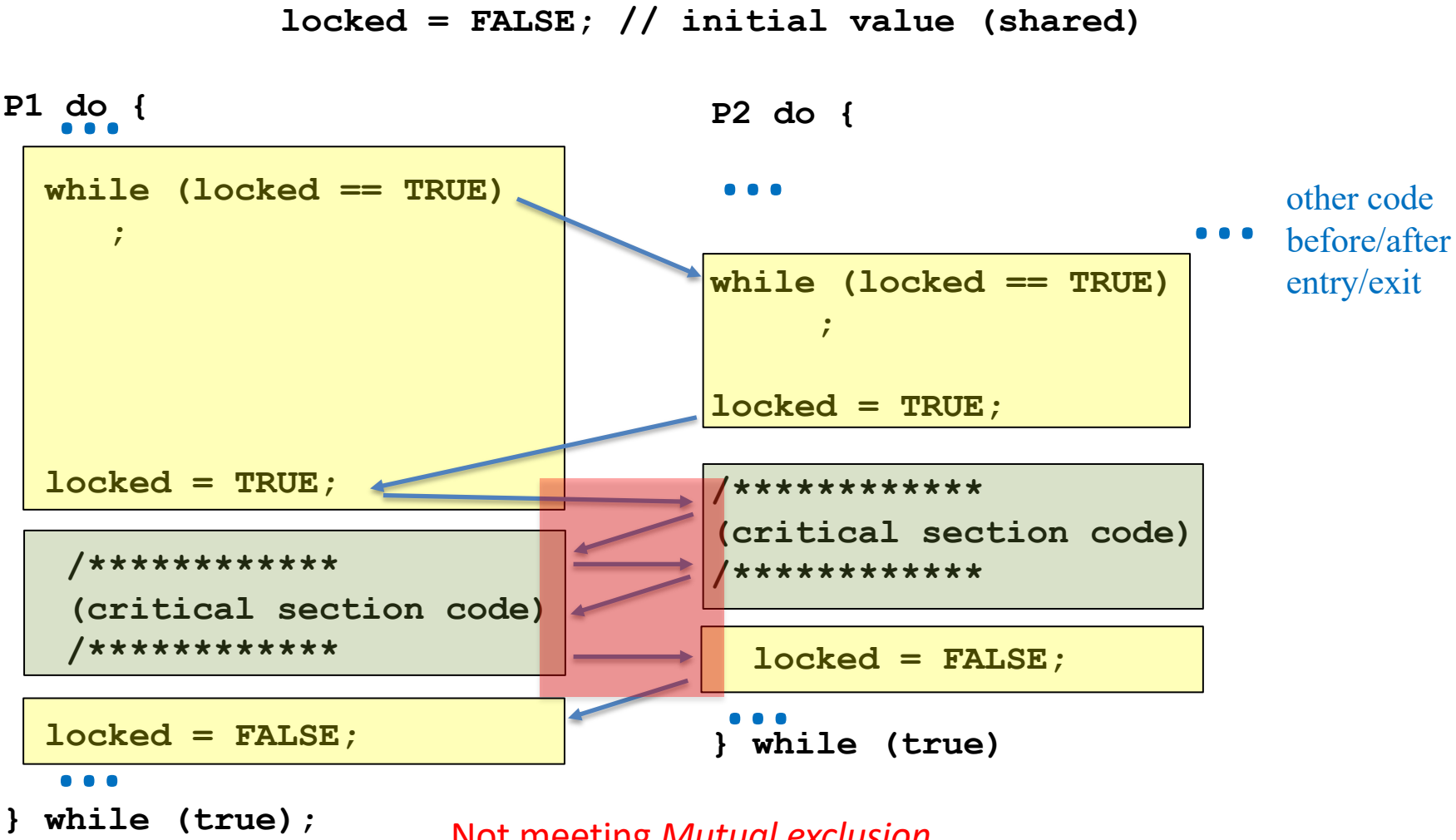
```
P2 do {
   ...
   while (locked == TRUE)
      ;
   locked = TRUE;

   /************
   (critical section code)
   /************

   locked = FALSE;
   ...
} while (true)
```

- Is there an interleaving where this fails?

# *Approach 2: Busy-Waiting (aka Spinning)*




Not meeting *Mutual exclusion* requirement

# *Reading, Writing, and Testing Locks*

- Is the instruction below atomic?

```
while (locked == TRUE);
```

A: load locked, R1
cmp R1, 1
beq A

- How about these instructions?

```
locked = TRUE;
```

store 1, locked

```
locked = FALSE;
```

store 0, locked

- Generally, if the high-level statement compiles to a single machine instruction, it is atomic
- Need all three reading, writing, testing of locks to be atomic
  - Hard to achieve even if each instruction is atomic

# *Try Strict Alternation*

... other code before/after entry/exit

- Idea: take turns using the critical section (CS)
  - Shared variable *turn* is used to identify which process should enter the CS next
- Does it work?
- What problems do you see?
  - Is there mutual exclusion?
  - Is there progress?
  - Is there bounded waiting?

Remember, P1 and P2 might want to enter the critical section multiple #s of times, including 0.

```
turn = 0; // initial value
P0 do { // proces P0
  ...
  while (turn != 0);

  /*********/
  critical section
  /*********/

  turn = 1;
  ...
} while (true)
P1 do{ // proces P1
  ...
  while (turn != 1);

  /*********/
  critical section
  /*********/

  turn = 0;
  ...
} while (true)
```

# *Try Strict Alternation*

... other code before/after entry/exit

```
turn = 0; // initial value
```

Not meeting the *Progress* requirement

```
P0 do { // proces P0
  ...
  while (turn != 0);

  /*********/
  critical section
  /*********/

  turn = 1;
  ...
} while (true)
```

```
P1 do{ // proces P1
  ...
  while (turn != 1);

  /*********/
  critical section
  /*********/

  turn = 0;
  ...
} while (true)
```

What if a process never wants to enter?

What if a process wants to enter, but it's not its turn, and the other is not ready?

Remember, P1 and P2 might want to enter the critical section multiple #s of times, including 0.

# *Fixing the "progress" requirement*

- What about this code?
- Each process has a flag to say that they want to enter the critical section
- Problems?
- Deadlocked!
  - They got mutual exclusion, for sure!
- For this reason, it does NOT meet the progress or bounded waiting requirements either

Remember, P1 and P2 might want to enter the critical section multiple #s of times, including 0.

```
bool flag[2]; // initialize to FALSE
P0 do {
    • • •
    flag[0] = TRUE;
    while (flag[1] == TRUE);

      /* critical section */

    flag[0] = FALSE;
    • • •
} while (true)
P1 do {
    • • •
    flag[1] = TRUE;
    while (flag[0] == TRUE);

      /* critical section */

    flag[1] = FALSE;
    • • •
} while (true)
```

• • • other code before/after entry/exit

# *Peterson's Solution*

- Consider 2 processes
- Assume that the LOAD and STORE instructions are atomic and cannot be interrupted
- The two processes share two variables:
  - *int turn;*
  - *boolean flag[2]*
- Variable *turn* indicates whose turn it is to enter the critical section
- The *flag* array is used to indicate if a process is ready to enter the critical section
  - *flag[i] = true* implies that process $P_i$ is ready!

# *Algorithm for Process $P_i$ and Process $P_j$*

```
flag[2]; // initialize to FALSE
```

Process $P_i$

```
while (TRUE) {
  ...
  flag[i] = TRUE;
  turn = j;
  while (flag[j]&&turn == j);

    critical section

  flag[i] = FALSE;

  ...
}
```

Process $P_j$

```
while (TRUE) {
  ...
  flag[j] = TRUE;
  turn = i;
  while (flag[i]&&turn == i);

    critical section

  flag[j] = FALSE;

  ...
}
```

... other code before/after entry/exit

The infinite do loop is just to suggest that a process will possibly want to enter its critical section multiple #s of times, including 0 times and 1 time.

# *Does Peterson's Solution work?*

- Prove that the 3 CS requirements are met:
  - Mutual exclusion is preserved
    $P_i$ enters CS only if: either *flag[j] = false* or *turn = i*
    - If $P_i$ is interested in entering, and $P_j$ is not. $P_i$ can enter regardless of turn.
    - If both processes are interested in entering, then *turn* determines who gets in
  - Progress requirement is satisfied
    - a process wanting to enter will be able to do so at some point
    - Why?
  - Bounded-waiting requirement is met
    - eventually it will be $P_i$'s turn if $P_i$ wants to enter

# *Algorithm for Process $P_i$ and Process $P_j$*

```
flag[2]; // initialize to FALSE
```

Process $P_i$

```
while (TRUE) {
  ...
  flag[i] = TRUE;
  turn = j;
  while (flag[j]&&turn == j);

    critical section

  flag[i] = FALSE;

  ...
}
```

Process $P_j$

```
while (TRUE) {
  ...
  flag[j] = TRUE;
  turn = i;
  while (flag[i]&&turn == i);

    critical section

  flag[j] = FALSE;

  ...
}
```

... other code before/after entry/exit

Think deeper:
What happens when an interrupt occurs when a process is in CS?

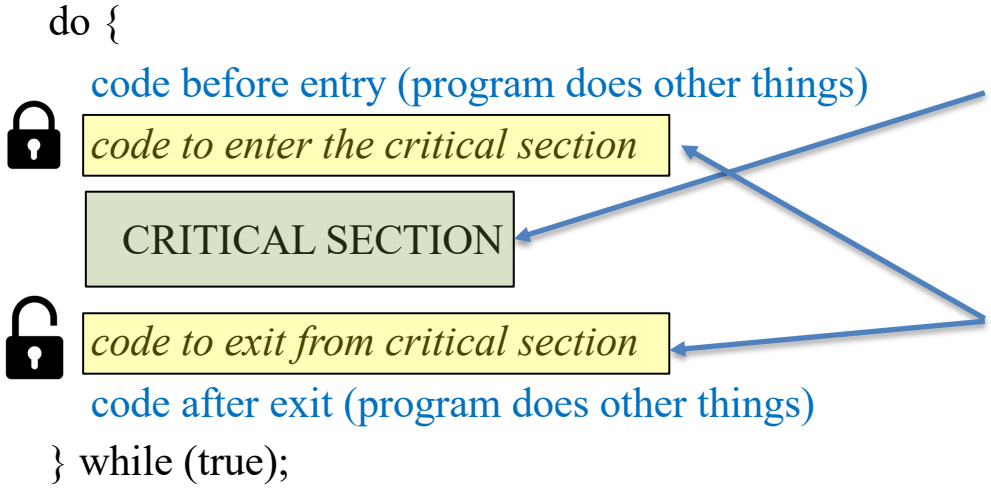
# *Limitation of Peterson's Solution*

- ONLY works for two processes
- Busy waiting is not efficient
- Requires LOAD and STORE to be atomic

# *Can hardware help?*

- Think deeper: we had atomicity only at the granularity of a machine instruction

Can we build specialized hardware instructions to provide additional functionality?




Q: Can HW provide instructions to execute the critical section atomically?

A: Perfect solution, but impractical. Why?

What if HW provides a shared lock?
Unlocked: no one is in CS
Locked: someone is in, needs to wait

# *Synchronization Hardware*

- Modern CPUs provide atomic hardware instructions (2 general types)

***Test-and-Set()***

- Test (read) memory word and set value

***Compare-and-Swap()***

- Swap the contents of two memory words

- How does this help?

https://en.wikipedia.org/wiki/X86_instruction_listings

# *Critical-section Solution Using Locks*

The infinite do loop is just to suggest that a process will possibly want to enter its critical section multiple #s of times, including 0 times and 1 time.

The problem before was that these could not be done in a single instruction

```
while (TRUE){
   code before entry to CS
   acquire lock
   critical section
   release lock
   code after exit from CS
}
```

Suppose they are done with single atomic instructions

Process repeatedly tries to get the lock and only when successful, enters the critical section

- Refer to this as a "spinlock"

# *test_and_set* Instruction

*boolean* types have values of 0 or 1

**Definition (*functionally what happens, not implementation*)**

```
boolean test_and_set (boolean *target) {
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```

- Returns (test) the original value of the (boolean) value at the "target" address (**target*)
- Set the new value of **target* to TRUE
- CPU instruction set architecture implements the *test_and_set* instruction
  - It executes as a single atomic instruction! (not as a function)

Note: where does "**test**" come from?
The original value of the target is returned and typically used in a test to determine if we made a change (so the lock is acquired).

*Think deeper:*
*What does it mean if it returns TRUE? Or FALSE?*

# *Solution using test_and_set()*

- Shared boolean variable *lock*, initialized to FALSE
- Solution:

... other code before/after entry/exit

```
while (TRUE){
  ...
    while (test_and_set(&lock) == TRUE);

     /* critical section */

    lock = FALSE;
  ...
}
```

**Does it work to solve the critical section problem?**
Think about mutually exclusive, progress, and bounded waiting

# *compare_and_swap Instruction*

**Definition (***functionally what happens, not implementation***)**

```
int compare_and_swap(int *target, int expected, int newvalue)
{
        int temp = *target;
        if (*target == expected)
            *target = newvalue;
        return temp;
}
```

- Returns the original value at **target*
- Set **target* to *newvalue* only if **target == expected*
  - That is, the swap takes place only if we guess the current value correctly
- CPU instruction set architecture implements a *compare_and_swap* instruction
  - It executes as a single atomic instruction!

Note: where does "**swap**" come from?
We are swaping between expected(current) and newvalue on target

*Think deeper:*
*What does it mean if we made the right guess?*

# *Solution using compare_and_swap*

... other code before/after entry/exit

- Shared integer *lock* initialized to 0
- Solution:

```
while (TRUE){
  ...
  while(compare_and_swap(&lock, 0, 1) != 0);
     /* critical section */
  lock = 0;
  ...
}
```

**How to understand this?**
We guess it is unlocked(0), we try to swap unlocked(0) with locked(1). If we are right, it should return unlocked(0) as old value.

**Does it work to solve the critical section problem?**
Think about mutually exclusive, progress, and bounded waiting

# Bounded-waiting with test_and_set

... other code before/after entry/exit

N processes: $P_0$ , $P_1$ , …, $P_{N-1}$

```
while (TRUE){
   ...
      waiting[i] = TRUE;
      key = TRUE;
      while (waiting[i] && key)
         key = test_and_set(&lock);
      waiting[i] = FALSE;
       /* critical section */
      j = (i + 1) % n;
      while ((j != i) && !waiting[j])
         j = (j + 1) % n;
      if (j == i)
         lock = FALSE;
      else
         waiting[j] = FALSE;
   ...
}
```

Spinning if I am waiting and do not have the lock

Stop waiting if I get the lock

Find the next waiting process (Round-robin order)

If no one is waiting, release the lock

If someone is waiting, stop its waiting and hand over the lock (no need to release)

# Spinning vs. Blocking

other code before/after entry/exit

```
while (TRUE){
   ...
      waiting[i] = TRUE;
      key = TRUE;
      while (waiting[i] && key)
         key = test_and_set(&lock);
      waiting[i] = FALSE;
       /* critical section */
      j = (i + 1) % n;
      while ((j != i) && !waiting[j])
         j = (j + 1) % n;
      if (j == i)
         lock = FALSE;
      else
         waiting[j] = FALSE;
   ...
}
```

N processes: $P_0$ , $P_1$ , …, $P_{N-1}$

We are spinning (*busy-waiting*) for some condition to change

We are "presuming" that this other process will eventually get scheduled on CPU

**Inefficiency caused by spinning:**

- You are wasting your time spinning … no work is done
- If CPU scheduler is round-robin and uses a quantum, process will use the quantum just to wait
- Then other process is scheduled and (maybe) changes the condition, otherwise spinning process will continue
- OS does not know process is spinning

# *Approach 3: Blocking instead of Spinning*

```
EnterCS(L) {
    Disable Interrupts
    Check if anyone is using L
    If not {
         Set L to being used
    }
    else {
         Move this PCB to Blocked
              queue for L
    }
    Enable Interrupts
}
```

Think of L as a lock

```
ExitCS(L) {
    Disable Interrupts
    Check if blocked queue
       for L is empty
    if so {
       Set L to free
    }
    else {
        Move PCB from head of
             Blocked queue of L to
             Ready queue
    }
    Enable Interrupts
}
```

1. *If someone is in CS, the process enters a blocking queue and releases the CPU*
2. *If a process is exiting from CS, it moves one process from blocking queue to ready queue*
3. *If no one is waiting, release the lock*

# *Blocking instead of Spinning*

```
EnterCS(L) {
    Disable Interrupts
    Check if anyone is using L
    If not {
         Set L to being used
    }
    else {
         Move this PCB to Blocked
              queue for L
    }
    Enable Interrupts
}
```

Think of L as a lock

```
ExitCS(L) {
    Disable Interrupts
    Check if blocked queue
       for L is empty
    if so {
       Set L to free
    }
    else {
        Move PCB from head of
             Blocked queue of L to
             Ready queue
    }
    Enable Interrupts
}
```

Advantage:

- Blocking processors will not unnecessarily occupy CPU cycles
- Flexibility on ordering in the blocked queue

Disadvantages?

- Requires extra synchronization logic to make it work

# Roadmap: Concurrency and Synchronization

## Concurrent Applications

## Shared Objects

Bounded buffer  Barrier

## Synchronization Variables

Semaphores  Locks  Condition Variables

## Atomic Instructions

Interrupt disable  Test-and-Set  Compare-and-Swap

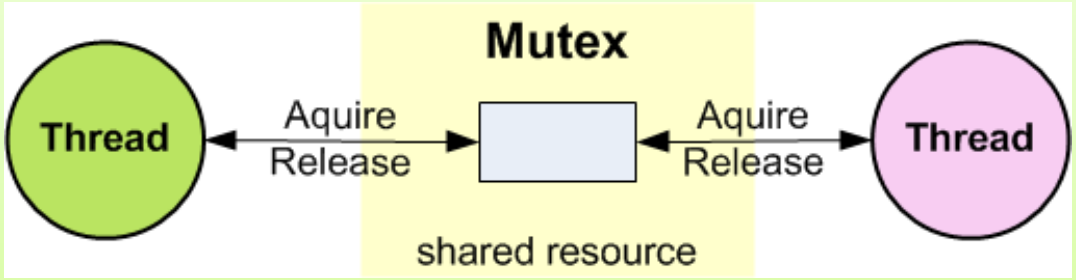
## Hardware

Multiple processors  Hardware interrupts

# *Mutex Locks*

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem

Simplest solution: ***mutex* lock** ("*mut*ual *ex*clusion")
Protect a critical section by first *acquire()* a lock then *release()* the lock
implemented via hardware atomic instructions



# *acquire() and release()*

```
acquire() {
   while (!available)
      ; /* busy wait */
   available = false;
}
release() {
   available = true;
}
while (TRUE){
   ...
   acquire lock

   critical section

   release lock
   ...
}
```

The assumption is that *acquire()* and *release()* will be executed atomically

... other code before/after entry/exit

- Mutex objects are intended to serve as a low-level primitive from which other thread synchronization functions can be built.
- Pthreads supports mutex objects

See example: posix_mutex_lock.c

int ***pthread_mutex_lock***(pthread_mutex_t **mutex*)
int ***pthread_mutex_unlock***(pthread_mutex_t **mutex*)

Check out the Linux manual for other variants of lock: trylock and timedlock

Potentially, how can they be implemented with test_and_set or compare_and_swap?

See example: mutex_implementation.c

# *If there are more resources to share*

- A critical section only allows one process to enter
  - Only 1 unit of resource
- What if we can allow up to N processors to enter
  - Up to N units of resource
    - Up to 3 printers, or 5 database connections
  - Producer–consumer synchronization
    - Consumer must wait if the queue has 0 element
    - Producer must wait if the queue has N element (full)

# *Semaphore (Dijkstra)*

- Synchronization tool that provides more sophisticated ways (than mutex locks) for processes to synchronize their activities.
- A semaphore *S* is an integer variable
- Can only be accessed via two atomic operations
  - *wait()* and *signal()*

Originally called *P()* and *V()* by Dijkstra
*P* = Probeer ('try' in Dutch)
*V* = Verhoog ('increment' in Dutch)

```
wait(S) { // P()
  while (S <= 0)
    ; // busy wait
  S--;
}
```

```
signal(S) { // V()
  S++;
}
```

https://en.wikipedia.org/wiki/Semaphore_(programming)

# *Semaphore Usage*

- *Binary semaphore*
  - An integer value can range only between 0 and 1
  - Functionally the same as a mutex lock
- *Counting semaphore*
  - An integer value can be set based on the problem
- Can solve various synchronization problems with semaphores

Consider $P_1$ and $P_2$ that require $F_1$ to happen before $F_2$

Create a semaphore *S* initialized to 0

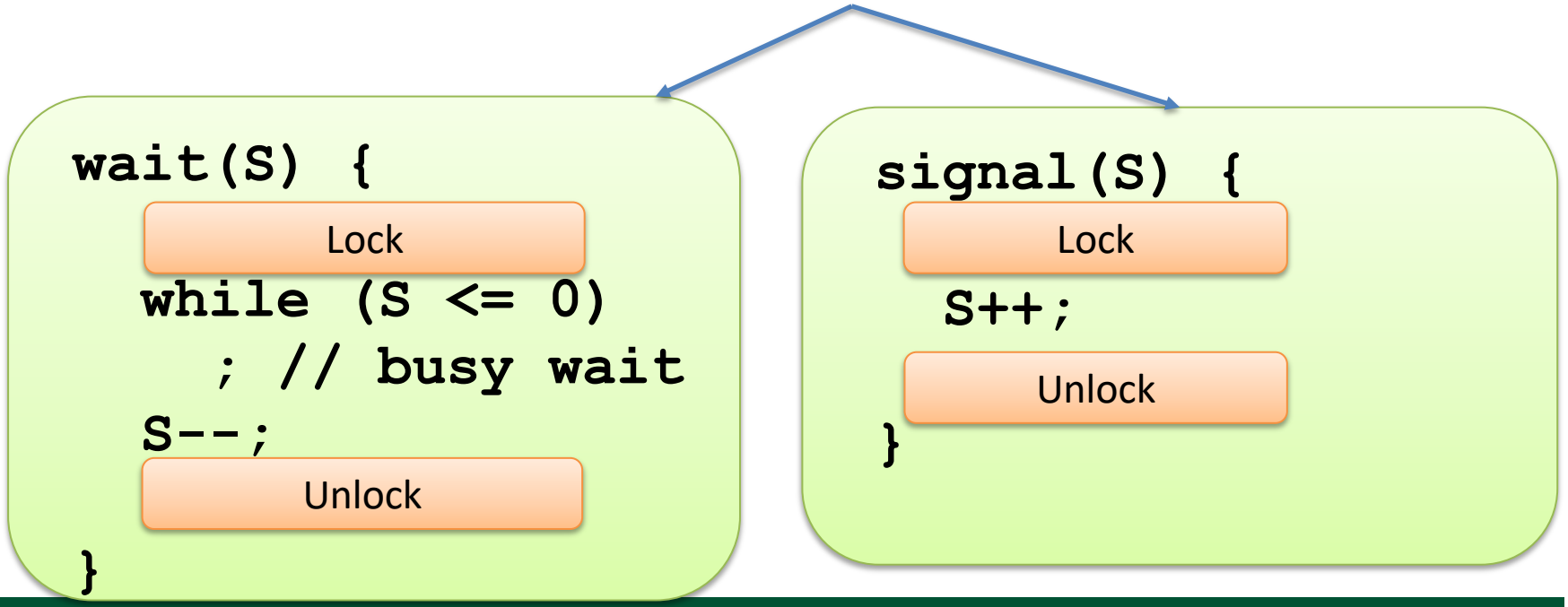
```
P1:                          P2:
   F1();                        wait(S);
   signal(S);                   F2();
```

# *Semaphore Implementation*

- Must guarantee that no two processes can execute the *wait()* and *signal()* …
  - … on the same semaphore …
  - … at the same time

What is the problem?

critical section problem



# *Semaphores with no Busy waiting (Blocking)*

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - *block* – place the process invoking the operation on the appropriate waiting queue
  - *wakeup* – remove one of processes in the waiting queue and place it in the ready queue

```
typedef struct{
  int value;
  struct process *list;
} semaphore;
```

# *Implementation with no Busy waiting*

```
wait(semaphore *S) {
  lock();
  S->value--;
  if (S->value < 0) {
    add this process to S->list;
    unlock();
    block();
  }
  unlock();
}
signal(semaphore *S) {
  lock();
  S->value++;
  if (S->value <= 0) {
    remove a process P from S->list;
    wakeup(P);
  }
  unlock();
}
```

**NOTE:** block() and wakeup() are system calls

# *POSIX semaphores*

- POSIX semaphores allow processes and threads to synchronize their actions
  - A semaphore is an integer whose value is $\geq 0$

***int sem_post(sem_t *sem)***

- Increment the semaphore value by one

***int sem_wait(sem_t *sem)***

- Decrement the semaphore value by one
- If the value of a semaphore is currently zero, then a *sem_wait()* will block until the value becomes $>0$

Check out the Linux manual for other variants of wait: trywait and timedwait

See example:
semaphore_printer.c

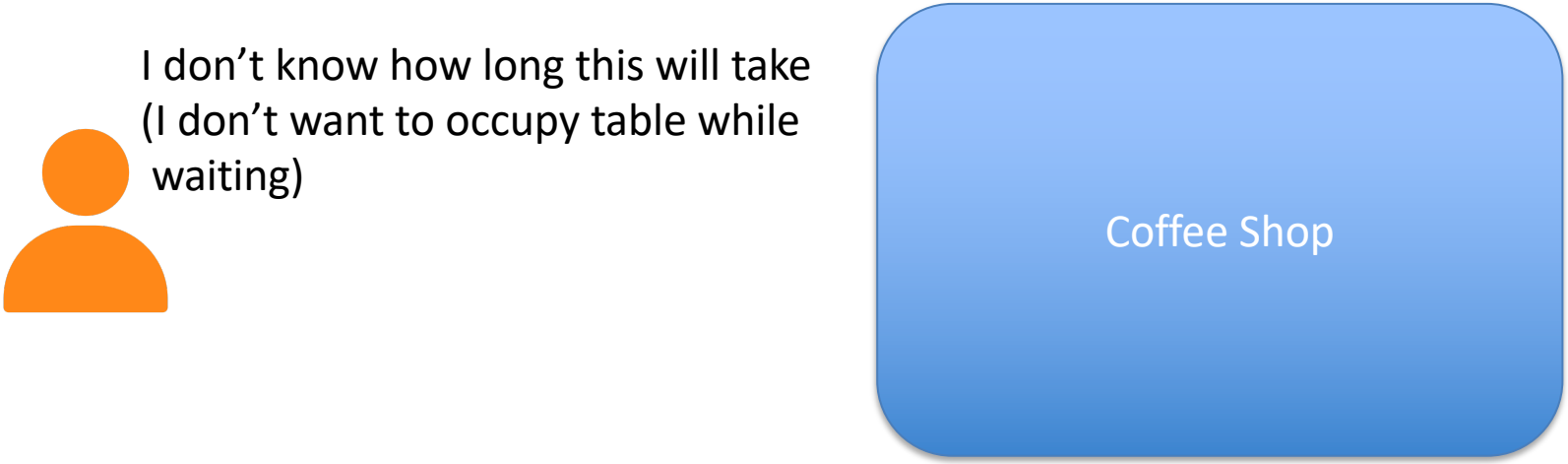
See example:
semaphore_printer_timedwait.c

# *POSIX Condition Variable*

- A condition variable allows a process/thread to wait for a certain condition to become true.
- Always used together with a mutex.

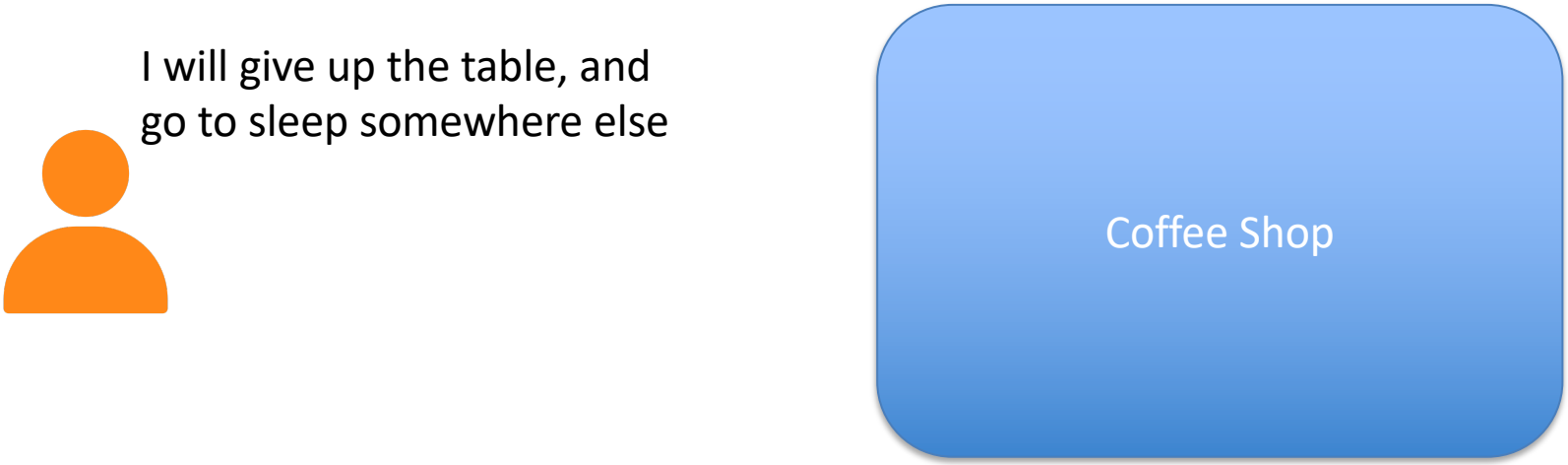
# *POSIX Condition Variable*

- A condition variable allows a process/thread to wait for a certain condition to become true.
- Always used together with a mutex.



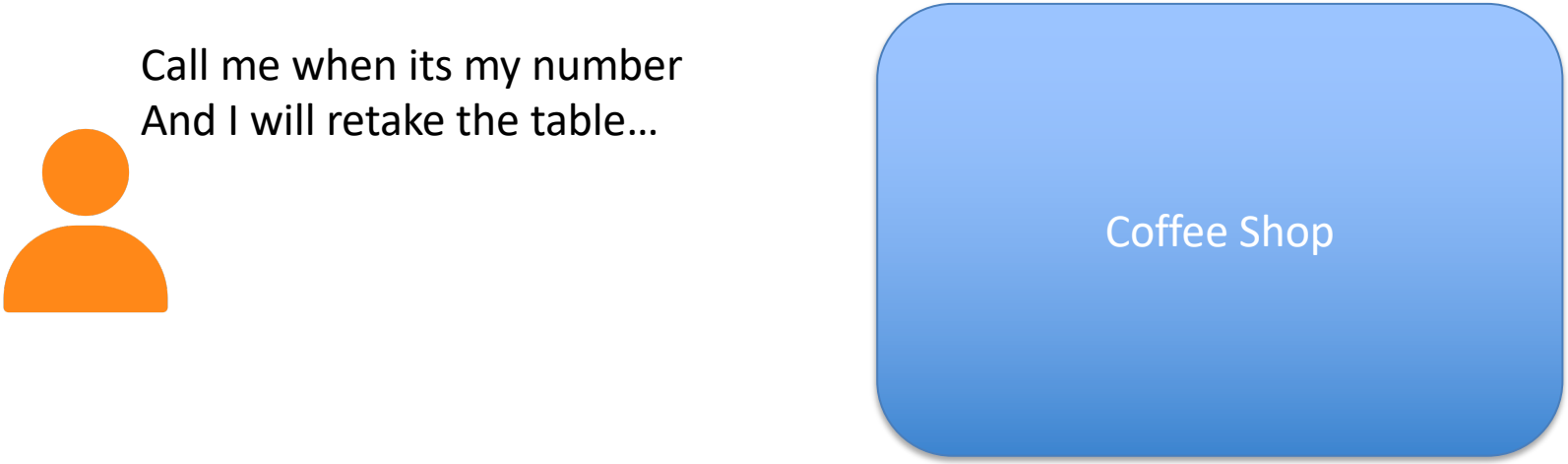
# *POSIX Condition Variable*

- A condition variable allows a process/thread to wait for a certain condition to become true.
- Always used together with a mutex.



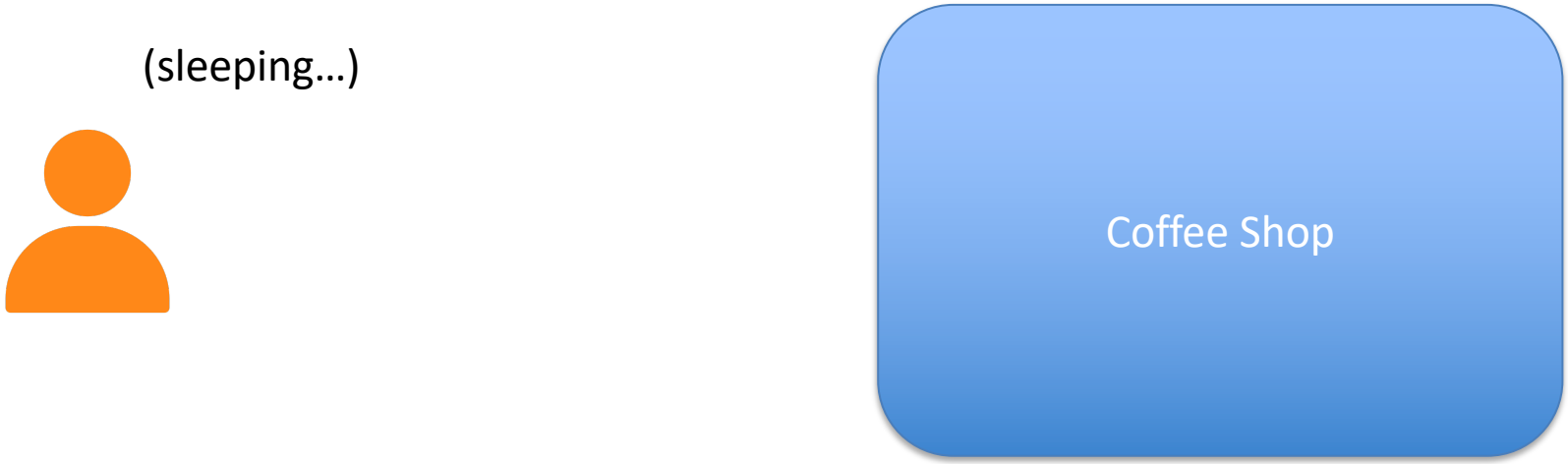
# *POSIX Condition Variable*

- A condition variable allows a process/thread to wait for a certain condition to become true.
- Always used together with a mutex.



# *POSIX Condition Variable*

- A condition variable allows a process/thread to wait for a certain condition to become true.
- Always used together with a mutex.

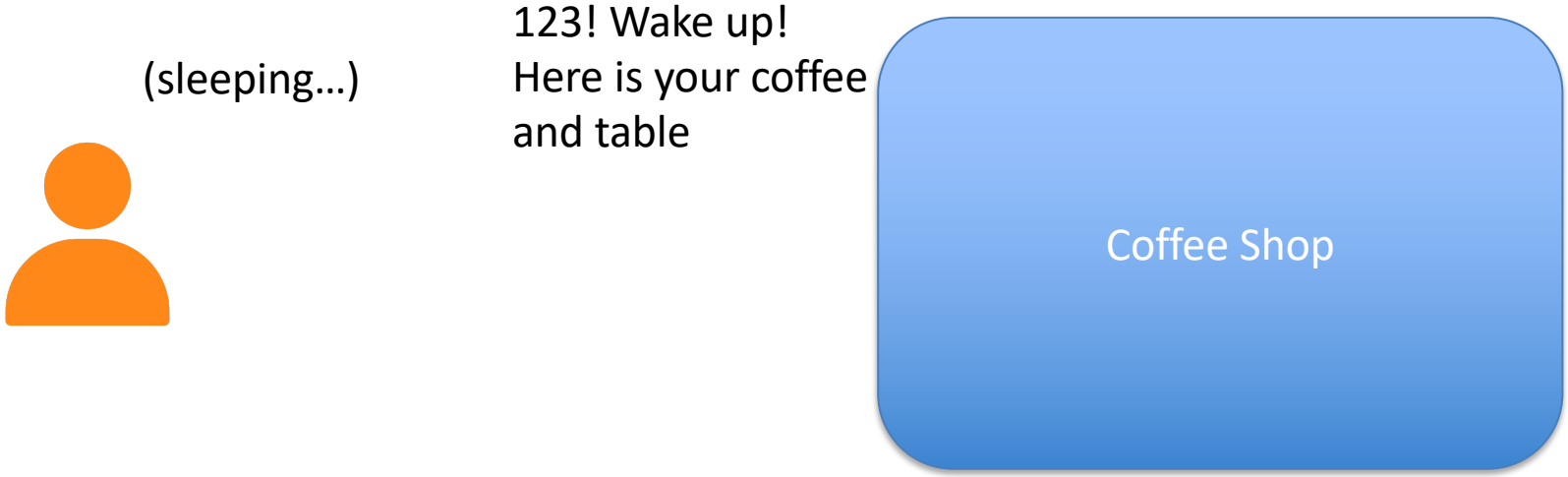


# *POSIX Condition Variable*

- A condition variable allows a process/thread to wait for a certain condition to become true.
- Always used together with a mutex.

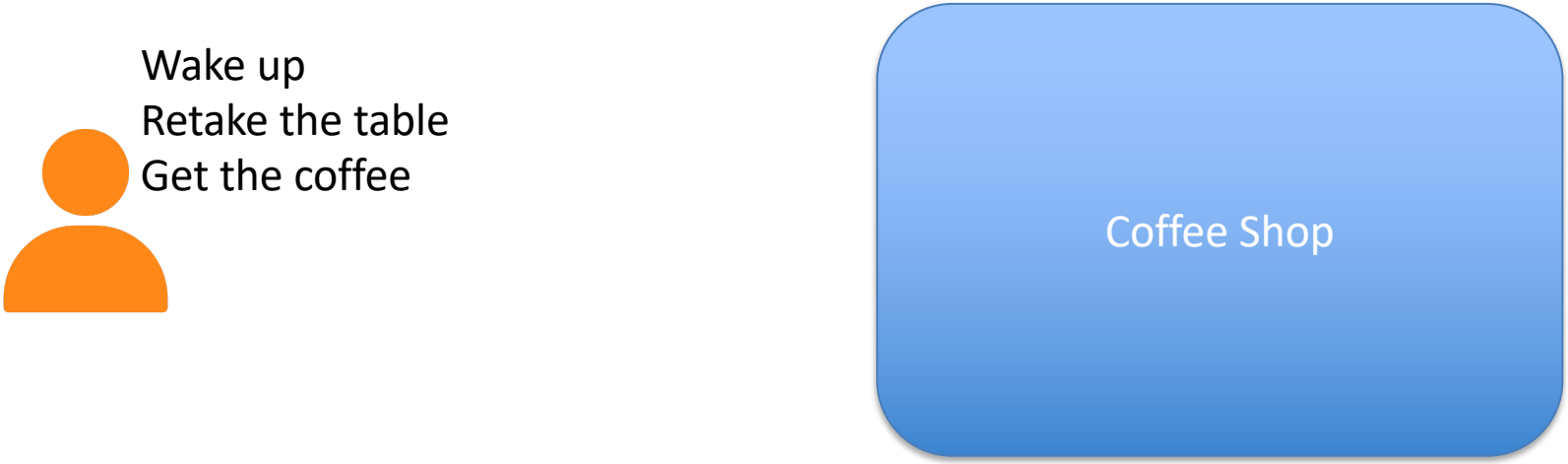
# *POSIX Condition Variable*

- A condition variable allows a process/thread to wait for a certain condition to become true.
- Always used together with a mutex.



# *POSIX Condition Variable*

- A condition variable allows a process/thread to wait for a certain condition to become true.
- Always used together with a mutex.



# *POSIX Condition Variable*

- A condition variable allows a process/thread to wait for a certain condition to become true.
- Always used together with a mutex.

int **pthread_cond_wait**(pthread_cond_t * cont, pthread_mutex_t * *mutex)*
- releases the mutex and waits for a signal.

int **pthread_cond_signal**(pthread_cond_t *cont)
- Wakes **one** waiting thread.

Check out the Linux manual for other ways to wake up: broadcast
Check out the Linux manual for other ways to wait: timedwait


See example:
condition_variable_simple.c


See example:
condition_variable_coffeeshop.c

# *Problems with Synchronization*

- Need to be careful about how synchronization tools are used in relation to other processes/threads
- Incorrect use can lead to…
  - Deadlock and starvation are possible
  - More to come in next lecture

# *Roadmap: Concurrency and Synchronization*

## *Concurrent Applications*

## *Shared Objects*

Bounded buffer  Barrier

## *Synchronization Variables*

Semaphores  Locks  Condition Variables

## *Atomic Instructions*

Interrupt disable  Test-and-Set  Compare-and-Swap

## *Hardware*

Multiple processors  Hardware interrupts

# *Classical Problems of Synchronization*

- Classical problems used to test newly-proposed synchronization schemes
  - Master-Worker Problem
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

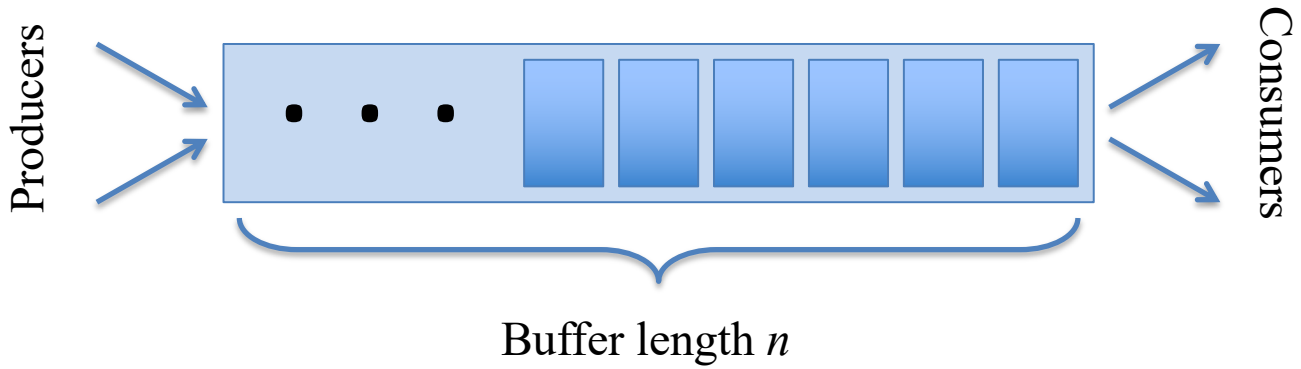
# *Classical Problems of Synchronization*

- Classical problems used to test newly-proposed synchronization schemes
  - Master-Worker Problem
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

# *Bounded-Buffer Problem*

- Fixed-size buffer holds *n* items
- Consumers remove items in order (wait when empty)
- Producers insert items in order (wait when full)
- Must protect any modifications to the buffer

- Semaphore ***mutex***: initialized to 1
- Semaphore ***full***: initialized to 0
- Semaphore ***empty***: initialized *n*



# Bounded Buffer Problem – Producer

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Request an empty spot
(**empty = empty - 1**)
If none (full), then wait

Request a filled spot.
(**full = full - 1**)
If none (empty), then wait

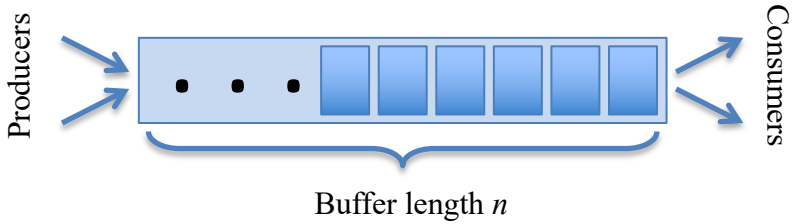
Provide a filled spot
(**full = full + 1**)

Provide an empty spot
(**empty = empty + 1**)

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```

Semaphore ***mutex*** =1 ← Only one access buffer
Semaphore ***full*** = 0 ← # of filled spots
Semaphore ***empty*** = *n* ← # of empty spots



# *Bounded Buffer Problem – Producer*

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Semaphore *mutex* =1
Semaphore *full* = 0
Semaphore *empty* = *9*

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```


Buffer length *9*

# Bounded Buffer Problem – Producer

- Producer:

```
while (TRUE){
   ...
   /* produce item*/
   ...
   wait(empty);
   wait(mutex);
   ...
   /* add to buffer */
   ...
   signal(mutex);
   signal(full);
}
```

Semaphore *mutex* =1
Semaphore *full* = 0
Semaphore *empty* = *9*

- Consumer:

```
while (TRUE){
   wait(full);
   wait(mutex);
   ...
   /* take from buffer*/
   ...
   signal(mutex);
   signal(empty);
   ...
   /* consume item */
   ...
}
```

wait


Buffer length *9*

# Bounded Buffer Problem – Producer

- Producer:

```
while (TRUE){
   ...
   /* produce item*/
   ...
   wait(empty);
   wait(mutex);
   ...
   /* add to buffer */
   ...
   signal(mutex);
   signal(full);
}
```

Semaphore *mutex* =1
Semaphore *full* = 0
Semaphore *empty* = 8

- Consumer:

```
while (TRUE){
   wait(full);
   wait(mutex);
   ...
   /* take from buffer*/
   ...
   signal(mutex);
   signal(empty);
   ...
   /* consume item */
   ...
}
```

wait


Buffer length 9

# *Bounded Buffer Problem – Producer*

- Producer:

```
while (TRUE){
   ...
   /* produce item*/
   ...
   wait(empty);
   wait(mutex);
   ...
   /* add to buffer */
   ...
   signal(mutex);
   signal(full);
}
```

Semaphore *mutex* = 0
Semaphore *full* = 0
Semaphore *empty* = 8

- Consumer:

```
while (TRUE){
   wait(full);
   wait(mutex);
   ...
   /* take from buffer*/
   ...
   signal(mutex);
   signal(empty);
   ...
   /* consume item */
   ...
}
```

wait


Buffer length 9

# Bounded Buffer Problem – Producer

- Producer:

```
while (TRUE){
   ...
   /* produce item*/
   ...
   wait(empty);
   wait(mutex);
   ...
   /* add to buffer */
   ...
   signal(mutex);
   signal(full);
}
```

Semaphore *mutex* = 0
Semaphore *full* = 0
Semaphore *empty* = 8

- Consumer:

```
while (TRUE){
   wait(full);
   wait(mutex);
   ...
   /* take from buffer*/
   ...
   signal(mutex);
   signal(empty);
   ...
   /* consume item */
   ...
}
```

wait


Buffer length 9

# Bounded Buffer Problem – Producer

- Producer:

```
while (TRUE){
   ...
   /* produce item*/
   ...
   wait(empty);
   wait(mutex);
   ...
   /* add to buffer */
   ...
   signal(mutex);
   signal(full);
}
```

Semaphore *mutex* = 1
Semaphore *full* = 0
Semaphore *empty* = 8

- Consumer:

```
while (TRUE){
   wait(full);
   wait(mutex);
   ...
   /* take from buffer*/
   ...
   signal(mutex);
   signal(empty);
   ...
   /* consume item */
   ...
}
```

wait


Buffer length 9

# Bounded Buffer Problem – Producer

- Producer:

```
while (TRUE){
   ...
   /* produce item*/
   ...
   wait(empty);
   wait(mutex);
   ...
   /* add to buffer */
   ...
   signal(mutex);
   signal(full);
}
```

Semaphore *mutex* = 1
Semaphore *full* = 1
Semaphore *empty* = 8

- Consumer:

```
while (TRUE){
   wait(full);
   wait(mutex);
   ...
   /* take from buffer*/
   ...
   signal(mutex);
   signal(empty);
   ...
   /* consume item */
   ...
}
```

wait


Buffer length 9

# Bounded Buffer Problem – Producer

- Producer:

```
while (TRUE){
   ...
   /* produce item*/
   ...
   wait(empty);
   wait(mutex);
   ...
   /* add to buffer */
   ...
   signal(mutex);
   signal(full);
}
```

Semaphore *mutex* = 1
Semaphore *full* = 0
Semaphore *empty* = 8

- Consumer:

```
while (TRUE){
   wait(full);
   wait(mutex);
   ...
   /* take from buffer*/
   ...
   signal(mutex);
   signal(empty);
   ...
   /* consume item */
   ...
}
```

resume


Buffer length 9

# Bounded Buffer Problem – Producer

- Producer:

```
while (TRUE){
   ...
   /* produce item*/
   ...
   wait(empty);
   wait(mutex);
   ...
   /* add to buffer */
   ...
   signal(mutex);
   signal(full);
}
```

Semaphore *mutex* = 0
Semaphore *full* = 0
Semaphore *empty* = 8

- Consumer:

```
while (TRUE){
   wait(full);
   wait(mutex);
   ...
   /* take from buffer*/
   ...
   signal(mutex);
   signal(empty);
   ...
   /* consume item */
   ...
}
```


Buffer length 9

# Bounded Buffer Problem – Producer

- Producer:

```
while (TRUE){
   ...
   /* produce item*/
   ...
   wait(empty);
   wait(mutex);
   ...
   /* add to buffer */
   ...
   signal(mutex);
   signal(full);
}
```

Semaphore *mutex* = 0
Semaphore *full* = 0
Semaphore *empty* = 8

- Consumer:

```
while (TRUE){
   wait(full);
   wait(mutex);
   ...
   /* take from buffer*/
   ...
   signal(mutex);
   signal(empty);
   ...
   /* consume item */
   ...
}
```


Buffer length 9

# Bounded Buffer Problem – Producer

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Semaphore *mutex* = 1
Semaphore *full* = 0
Semaphore *empty* = 8

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```


Buffer length 9

# Bounded Buffer Problem – Producer

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Semaphore *mutex* = 1
Semaphore *full* = 0
Semaphore *empty* = 9

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```


Buffer length 9

# Bounded Buffer Problem – Producer

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Semaphore *mutex* = 1
Semaphore *full* = 0
Semaphore *empty* = 9

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```


Buffer length 9

# Bounded Buffer Problem – Producer

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Semaphore *mutex* = 1
Semaphore *full* = 0
Semaphore *empty* = 8

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```


Buffer length 9

# Bounded Buffer Problem – Producer

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Semaphore *mutex* = 0
Semaphore *full* = 0
Semaphore *empty* = 8

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```


Buffer length 9

# Bounded Buffer Problem – Producer

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Semaphore *mutex* = 0
Semaphore *full* = 0
Semaphore *empty* = 8

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```


Buffer length 9

# Bounded Buffer Problem – Producer

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Semaphore *mutex* = 1
Semaphore *full* = 0
Semaphore *empty* = 8

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```


Buffer length 9

# Bounded Buffer Problem – Producer

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Semaphore *mutex* = 1
Semaphore *full* = 1
Semaphore *empty* = 8

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```


Buffer length 9

# Bounded Buffer Problem – Producer

- Producer:

```
while (TRUE){
   ...
   /* produce item*/
   ...
   wait(empty);
   wait(mutex);
   ...
   /* add to buffer */
   ...
   signal(mutex);
   signal(full);
}
```

Semaphore *mutex* = 1
Semaphore *full* = 1
Semaphore *empty* = 8

- Consumer:

```
while (TRUE){
   wait(full);
   wait(mutex);
   ...
   /* take from buffer*/
   ...
   signal(mutex);
   signal(empty);
   ...
   /* consume item */
   ...
}
```


Buffer length 9

# Bounded Buffer Problem – Producer

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Semaphore *mutex* = 1
Semaphore *full* = 2
Semaphore *empty* = 7

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```


Buffer length 9

# Bounded Buffer Problem – Producer

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Semaphore *mutex* = 1
Semaphore *full* = 3
Semaphore *empty* = 6

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```


Buffer length 9

# Bounded Buffer Problem – Producer

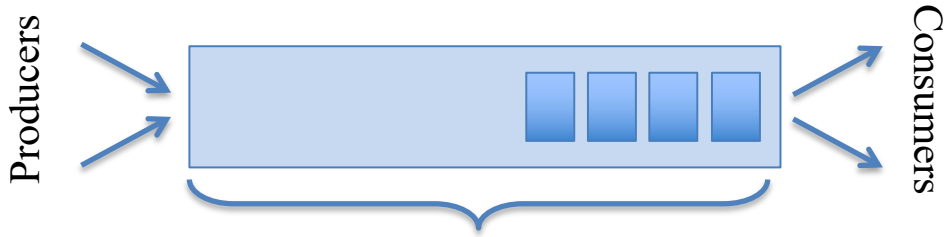
- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Semaphore *mutex* = 1
Semaphore *full* = 4
Semaphore *empty* = 5

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```




Buffer length 9

# Bounded Buffer Problem – Producer

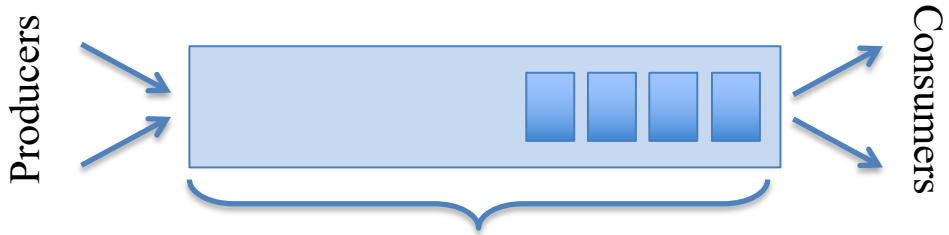
- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Semaphore *mutex* = 1
Semaphore *full* = 5
Semaphore *empty* = 4

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```




Buffer length 9

# Bounded Buffer Problem – Producer

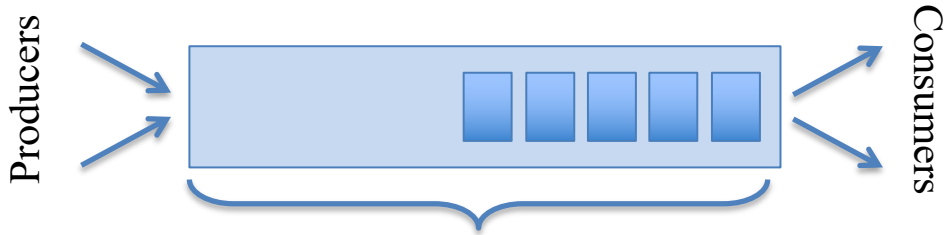
- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Semaphore *mutex* = 1
Semaphore *full* = 5
Semaphore *empty* = 4

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```




Buffer length 9

# Bounded Buffer Problem – Producer

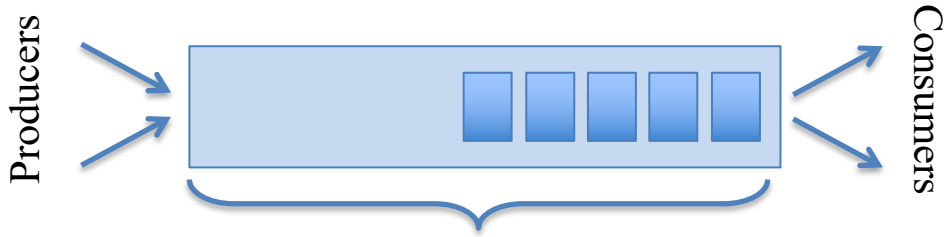
- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Semaphore *mutex* = 1
Semaphore *full* = 5
Semaphore *empty* = 4

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```




Buffer length 9

# Bounded Buffer Problem – Producer

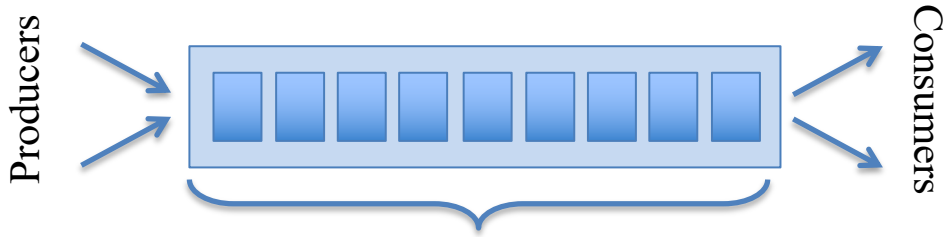
- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Semaphore *mutex* = 1
Semaphore *full* = 9
Semaphore *empty* = 0

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```




Buffer length 9

# Bounded Buffer Problem – Producer

- Producer:

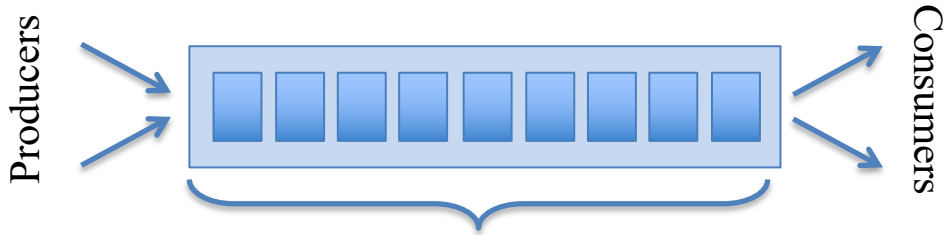
```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

wait

Semaphore *mutex* = 1
Semaphore *full* = 9
Semaphore *empty* = 0

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```




Buffer length 9

# Bounded Buffer Problem – Producer

- Producer:

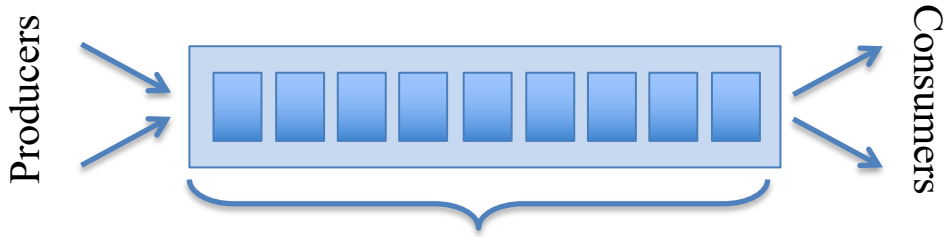
```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

wait

Semaphore *mutex* = 1
Semaphore *full* = 8
Semaphore *empty* = 0

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```




Buffer length 9

# Bounded Buffer Problem – Producer

- Producer:

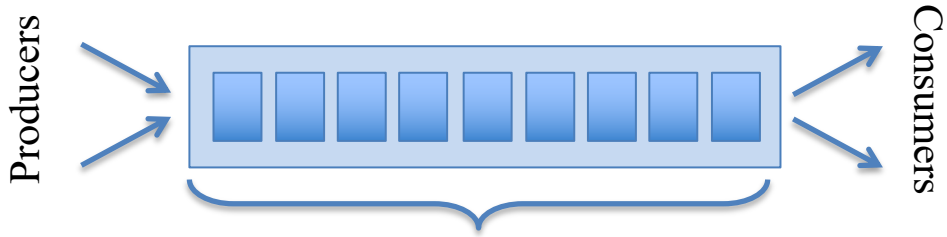
```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

wait

Semaphore *mutex* = 0
Semaphore *full* = 8
Semaphore *empty* = 0

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```




Buffer length 9

# Bounded Buffer Problem – Producer

- Producer:

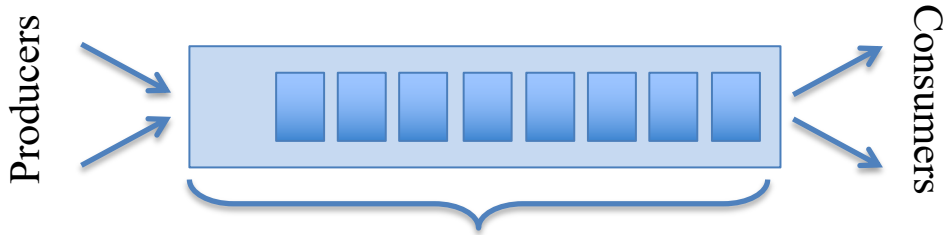
```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

wait

Semaphore *mutex* = 0
Semaphore *full* = 8
Semaphore *empty* = 0

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```




Buffer length 9

*Assume this is circular buffer

# Bounded Buffer Problem – Producer

- Producer:

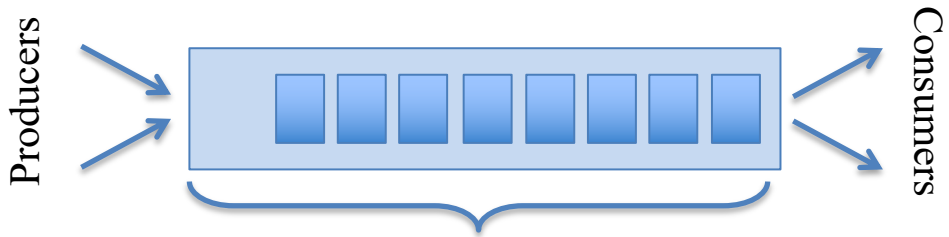
```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

wait

Semaphore *mutex* = 1
Semaphore *full* = 8
Semaphore *empty* = 0

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```




Buffer length 9

# Bounded Buffer Problem – Producer

- Producer:

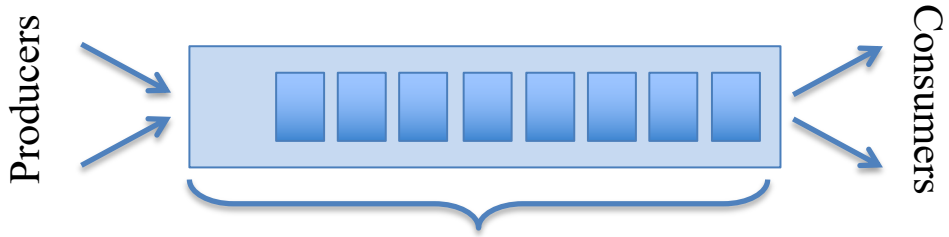
```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

resume

Semaphore *mutex* = 1
Semaphore *full* = 8
Semaphore *empty* = 1

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```




Buffer length 9

# Bounded Buffer Problem – Producer

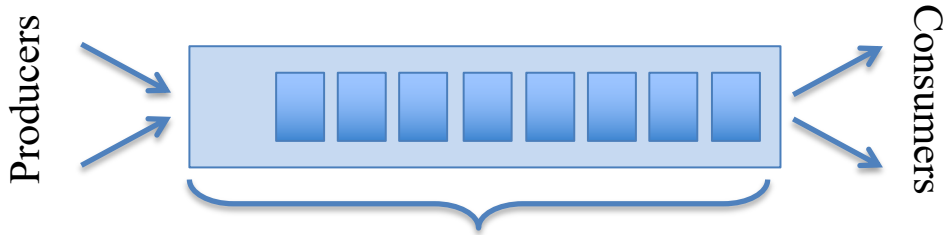
- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Semaphore *mutex* = 1
Semaphore *full* = 8
Semaphore *empty* = 0

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```




Buffer length 9

# *Bounded Buffer Problem – Producer*

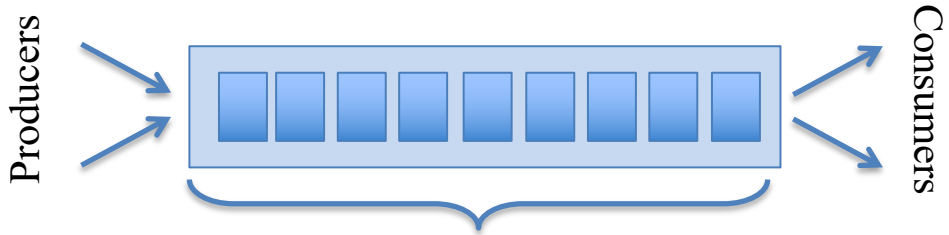
- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Semaphore *mutex* = 0
Semaphore *full* = 8
Semaphore *empty* = 0

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```




Buffer length 9

# Bounded Buffer Problem – Producer

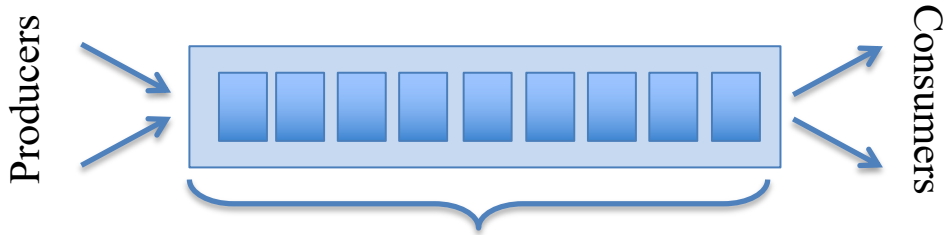
- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Semaphore *mutex* = 1
Semaphore *full* = 8
Semaphore *empty* = 0

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```




Buffer length 9

# Bounded Buffer Problem – Producer

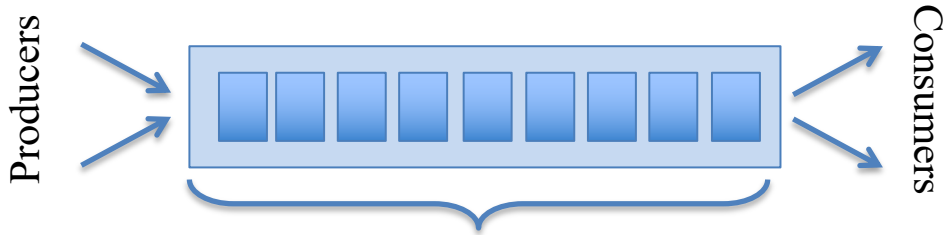
- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Semaphore *mutex* = 1
Semaphore *full* = 9
Semaphore *empty* = 0

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```




Buffer length 9

# Bounded Buffer Problem – Producer

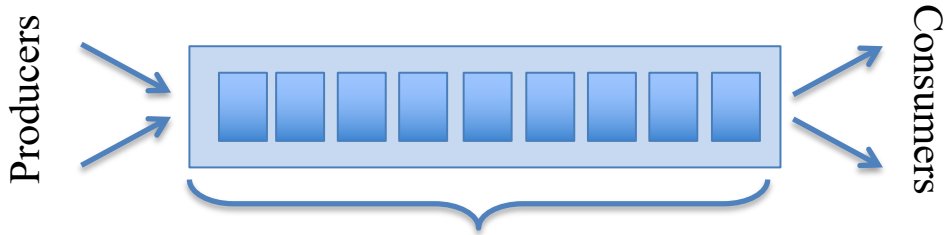
- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Semaphore *mutex* = 1
Semaphore *full* = 9
Semaphore *empty* = 0

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```




Buffer length 9

# *Next Class*

- Multi-object synchronization (deadlock)