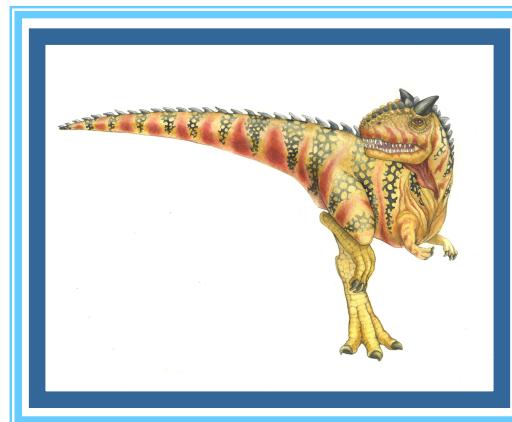


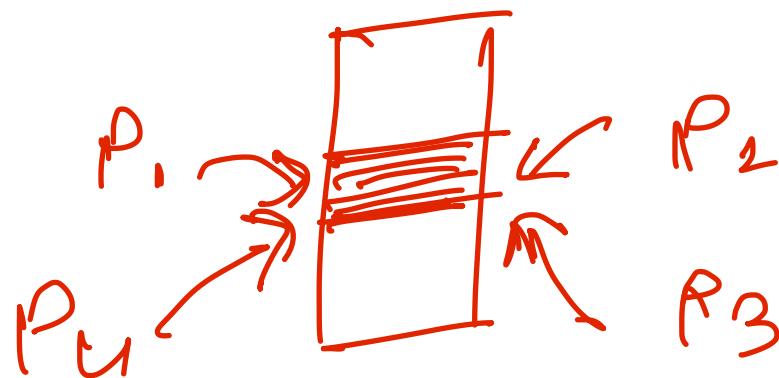
# Chapter 6: Synchronization Tools





# Chapter 6: Synchronization Tools

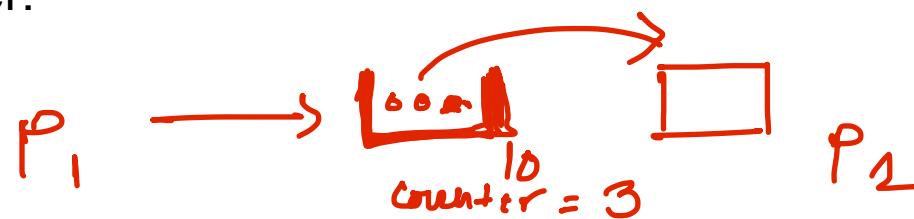
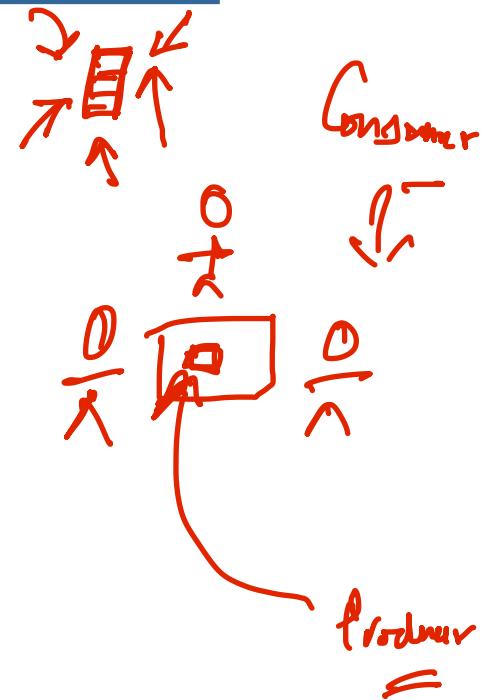
- Background
- The Critical-Section Problem ✓
- Peterson's Solution ✓
- Hardware Support for Synchronization ✓
- Mutex Locks ✓
- Semaphores ✓
- Monitors ✓
- Liveness ✓
- Evaluation





# Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:  
Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer counter that keeps track of the number of full buffers. Initially, counter is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.



requires balance  
Synchronization





# Producer

```
while (true) {  
    /* produce an item in next_produced */  
  
    while (counter == BUFFER_SIZE)  
        ; /* do nothing */  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```





# Consumer

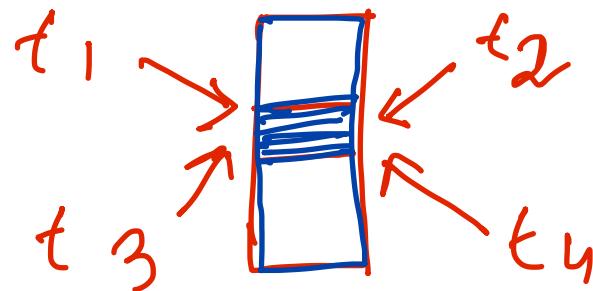
```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
    /* consume the item in next consumed */  
}
```





# Race Condition

- A **race condition** is an undesirable situation that occurs when a device or system attempts to perform **two or more operations at the same time**, but because of the nature of the device or system, the **operations must be done in the proper sequence** to be done correctly.
- A **race condition** occurs when **two or more threads** can access **shared data** and they **try to change it at the same time**.



Shared Memory





# Race Condition

- `counter++` could be implemented as

```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

- `counter--` could be implemented as

```
register2 = counter  
register2 = register2 - 1  
counter = register2
```

- Consider this execution interleaving with “count = 5” initially:

- $P_1$  S0: producer execute `register1 = counter`  
S1: producer execute `register1 = register1 + 1`
- $P_2$  S2: consumer execute `register2 = counter`  
S3: consumer execute `register2 = register2 - 1`
- $P_1$   
 $P_2$  S4: producer execute `counter = register1`  
S5: consumer execute `counter = register2`

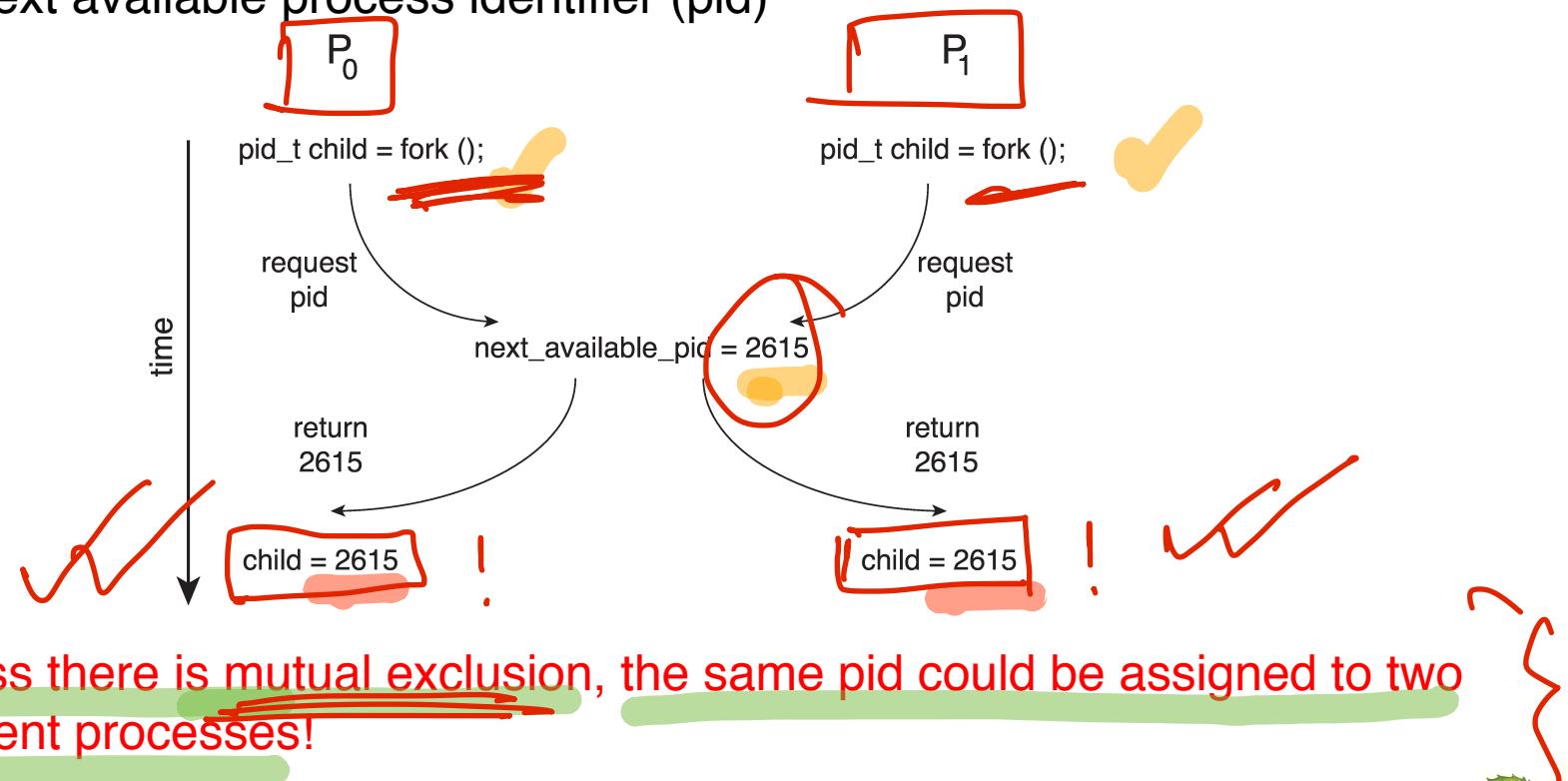
{register1 = 5}  
{register1 = 6} ✓  
{register2 = 5} 5  
{register2 = 4} ✓  
{counter = 6} ✓  $5+1$  ✓  
{counter = 4} ✓  $5-1$





# Race Condition

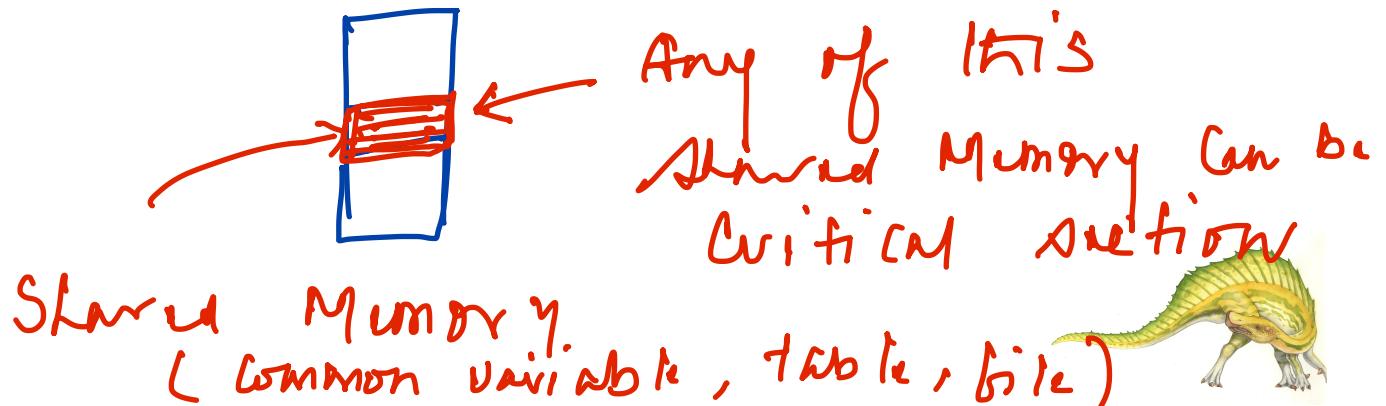
- Processes  $P_0$  and  $P_1$  are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)





# Critical Section Problem

- Consider system of  $n$  processes  $\{p_0, p_1, \dots p_{n-1}\}$  ✓
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section \*
- **Critical section problem** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**





# Critical Section

- General structure of process  $P_i$

```
do {  
    entry section  
    ||  
    critical section  
    ||  
    exit section  
    remainder section  
} while (true);
```





# Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections
2. **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 *allow*
3. **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

$P_1$  ~~executing~~  
 $P_2$  ~~executing~~  
 $P_3$  ~~executing~~  
⋮  
 $P_n$  ✓

time between  
request to  
granted





# Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - ▶ Essentially free of race conditions in kernel mode





# Peterson's Solution

- Not guaranteed to work on modern architectures! (But good algorithmic description of solving the problem) *- may not work in modern architecture but good to know*
- Two process solution !!!
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - `int turn;`
  - `boolean flag[2]`
- The 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 $P_i$

```
while (true) {
    flag[i] = true; ✓
    turn = j; ✓
    while (flag[j] && turn == j)
        ;
    /* critical section */
    flag[i] = false; ✓ // after completion of critical
    /* remainder section */
}
}
```



completion of critical  
section execution





## Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:

- ✓ 1. Mutual exclusion is preserved

$P_i$  enters CS only if:

either `flag[j] = false` or `turn = i`

- ✓ 2. Progress requirement is satisfied

- ✓ 3. Bounded-waiting requirement is met





# Peterson's Solution

- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures.
- Understanding why it will not work is also useful for better understanding race conditions.
- To improve performance, processors and/or compilers may reorder operations that have no dependencies. //
- For single-threaded this is ok as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!

Mutigra / Multiqroussw !!!





# Peterson's Solution

- Two threads share the data:

```
boolean flag = false;  
int x = 0;
```

- Thread 1 performs

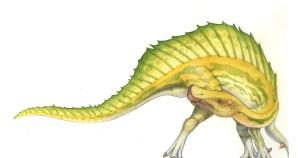
```
while (!flag)    bly = false  
    ;  
print x    ✓
```

- Thread 2 performs

```
x = 100;    ✓  
flag = true    ✓
```

- What is the expected output?

Problem! inconsistent data!



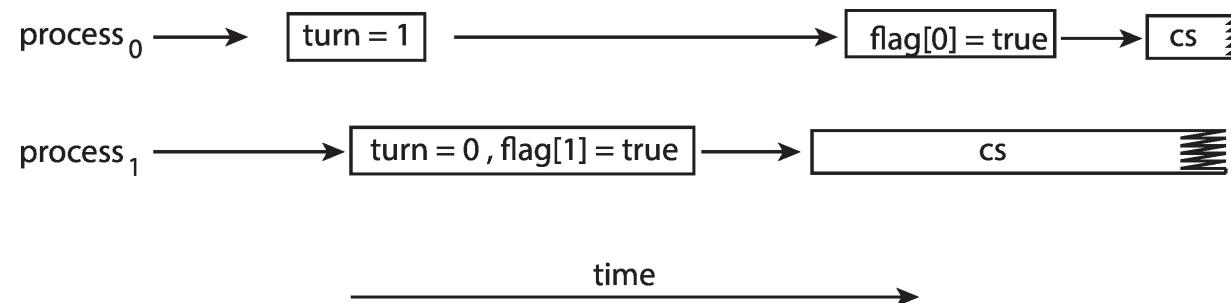


# Peterson's Solution

- 100 is the expected output.
- However, the operations for Thread 2 may be reordered:

```
flag = true;  
x = 100;
```

- If this occurs, the output may be 0!
- The effects of instruction reordering in Peterson's Solution



- This allows both processes to be in their critical section at the same time!





# Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - ▶ Operating systems using this not broadly scalable
- We will look at three forms of hardware support:
  1. Memory barriers
  2. Hardware instructions
  3. Atomic variables



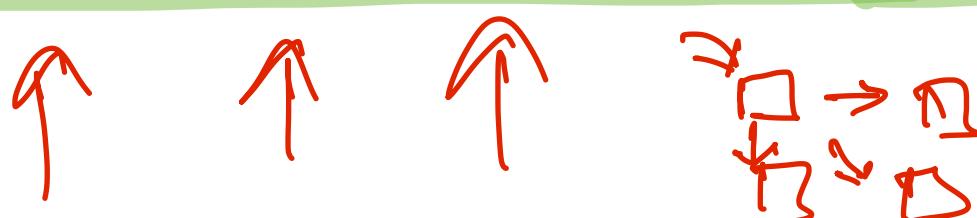


# Memory Barriers

- **Memory model** are the memory guarantees a computer architecture makes to application programs.
- Memory models may be either:

- **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors.
- **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors.

- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors.





# Memory Barrier

- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
    memory_barrier(); ✓ ← will ensure that same
print x
```

Shared action is not  
accessible by other  
processors.

- Thread 2 now performs

```
x = 100;
memory_barrier(); ✓
flag = true
```





# Hardware Instructions

- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or two *swap* the contents of two words atomically (uninterruptibly.)
- ✓ ■ **Test-and-Set** instruction
- ✓ ■ **Compare-and-Swap** instruction





# test\_and\_set Instruction

Definition:

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

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to true





# Solution using test\_and\_set()

- Shared boolean variable `lock`, initialized to `false`
- Solution:

```
do {  
    while (test_and_set(&lock))  
        ; /* do nothing */  
  
    /* critical section */ ✓  
    lock = false;  
    /* remainder section */  
}  
} while (true);
```





# compare\_and\_swap Instruction

Definition:

```
int compare_and_swap(int *value, int expected, int new_value) {  
    int temp = *value;  
  
    if (*value == expected)  
        *value = new_value;  
    return temp;  
}
```

1. Executed atomically ✓
2. Returns the original value of passed parameter **value** ✓
3. Set the variable **value** the value of the passed parameter **new\_value** but only if **\*value == expected** is true. That is, the swap takes place only under this condition.





# Solution using compare\_and\_swap

- Shared integer `lock` initialized to 0;
- Solution:

```
while (true) {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

    /* critical section */
    lock = 0; ??
    /* remainder section */
}
```





# Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.
- One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and booleans.
- For example, the **increment()** operation on the atomic variable **sequence** ensures **sequence** is incremented without interruption:

```
increment(&sequence);
```





# Atomic Variables

- The `increment()` function can be implemented as follows:

```
void increment	atomic_int *v)
{
    int temp;

    do {
        temp = *v;
    }
    while (temp != (compare_and_swap(v, temp, temp+1)));
}
```

4

Atomic variable  
use





# Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock ✓
- Protect a critical section by first **acquire()** a lock then **release()** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **acquire()** and **release()** must be atomic
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**





# Solution to Critical-section Problem Using Locks

```
while (true) {  
    acquire lock  
    critical section  
    release lock  
    -  
    remainder section  
}
```





# Mutex Lock Definitions

- `acquire() {  
 while (!available)  
 ; /* busy wait */  
 available = false;  
}  
  
■ release() {  
 available = true;  
}`

These two functions must be implemented atomically.  
Both test-and-set and compare-and-swap can be  
used to implement these functions.





# Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- **Semaphore  $S$  – integer variable**
- Can only be accessed via two indivisible (atomic) operations
  - **wait()** and **signal()**
    - ▶ (Originally called **P()** and **V()**)
- Definition of the **wait()** operation

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

- Definition of the **signal()** operation

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





# Semaphore Usage

✓ **Counting semaphore** – integer value can range over an unrestricted domain

✓ ■ **Binary semaphore** – integer value can range only between 0 and 1

- Same as a **mutex lock** ✓
- Can solve various synchronization problems
- Consider  $P_1$  and  $P_2$  that require  $S_1$  to happen before  $S_2$   
Create a semaphore “**synch**” initialized to 0

**P1 :**

```
S1;  
signal(synch);
```

**P2 :**

```
wait(synch);  
S2;
```

■ Can implement a counting semaphore **S** as a binary semaphore





# Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - ▶ But implementation code is short
    - ▶ Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

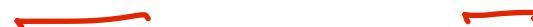




# Problems with Semaphores

- Incorrect use of semaphore operations:

- `signal (mutex)` .... `wait (mutex)`



- `wait (mutex)` ... `wait (mutex)`



- Omitting of `wait (mutex)` and/or `signal (mutex)`



- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.

*Problem examples*





# Monitors

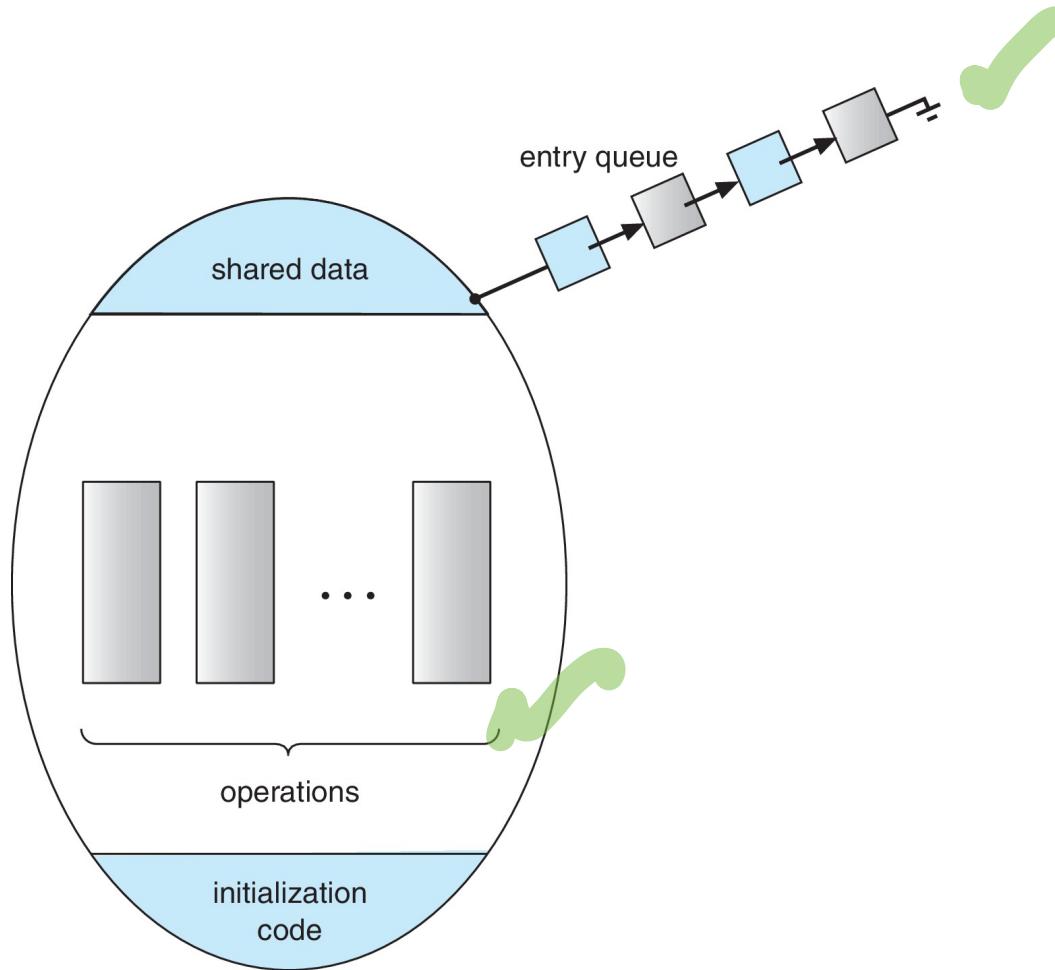
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Abstract data type, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- Pseudocode syntax of a monitor:

```
monitor monitor-name
{
    // shared variable declarations
    function P1 (...) { .... } ✓
    ✓
    function P2 (...) { .... } ✓
    ✓
    function Pn (...) {.....} ✓
    initialization code (...) { ... }
}
```





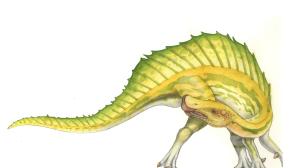
# Schematic view of a Monitor





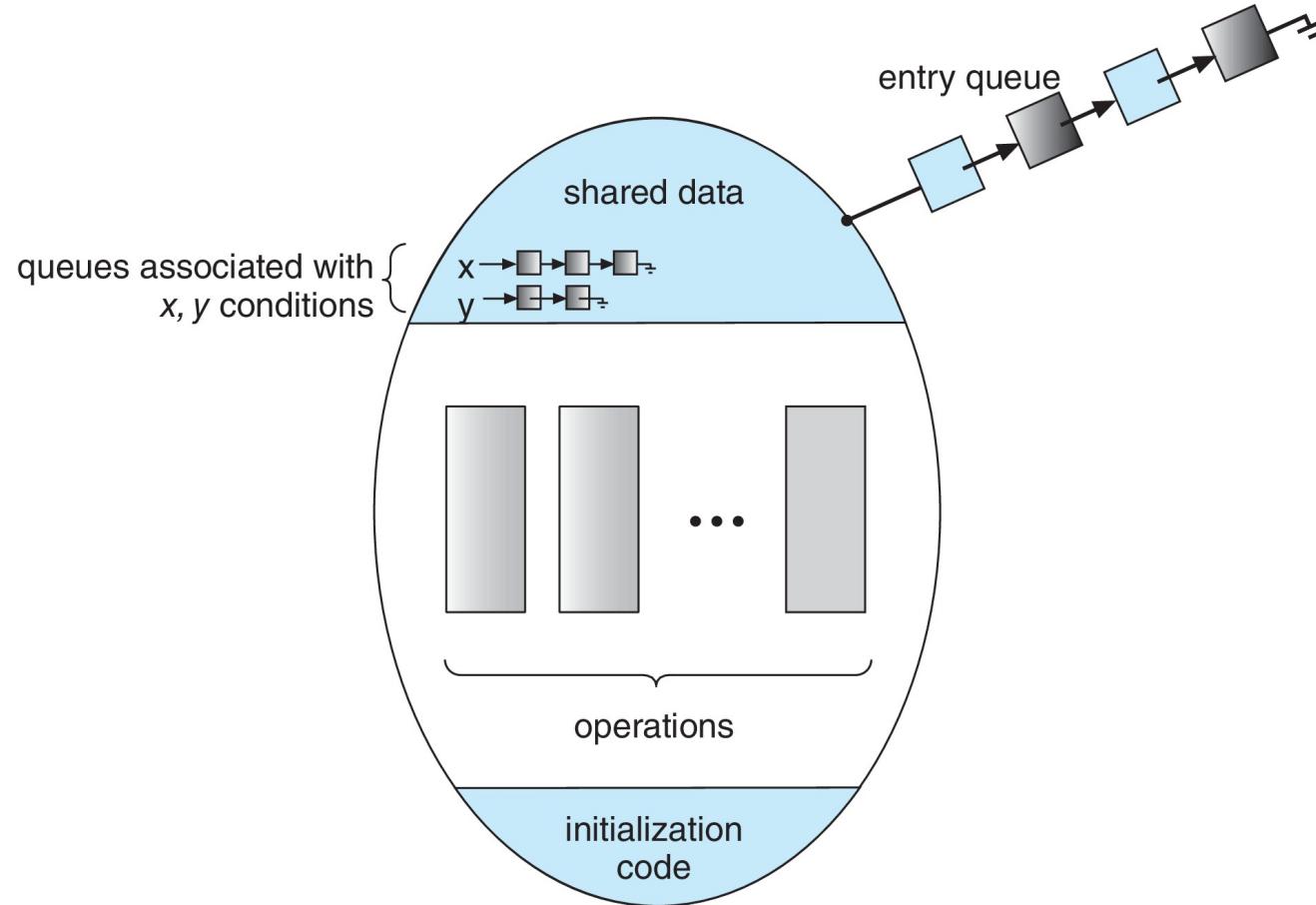
# Condition Variables

- **condition x, y;** ✓✓
- Two operations are allowed on a condition variable:
  - **x.wait()** – a process that invokes the operation is suspended until **x.signal()**
  - **x.signal()** – resumes one of processes (if any) that invoked **x.wait()**
    - ▶ If no **x.wait()** on the variable, then it has no effect on the variable





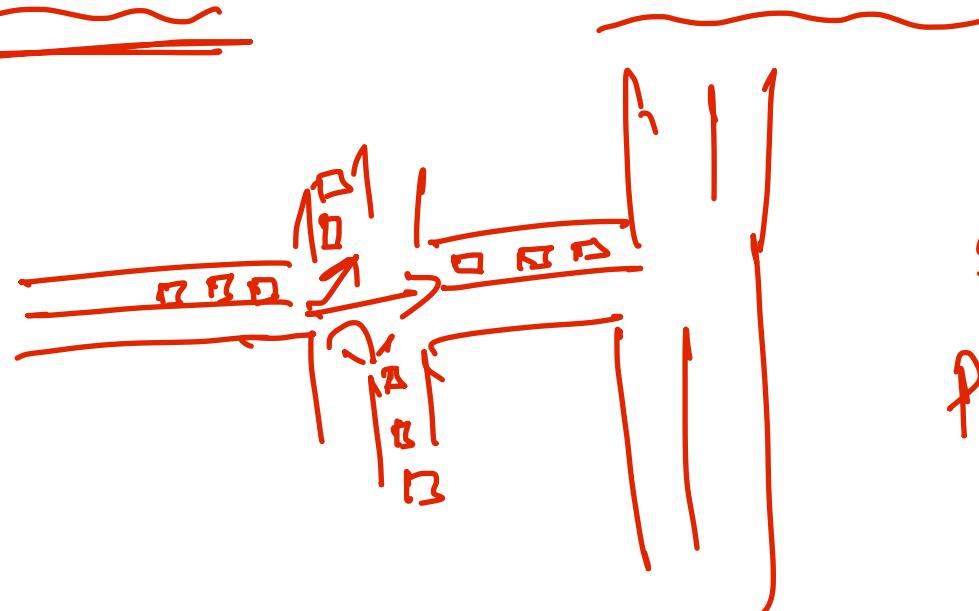
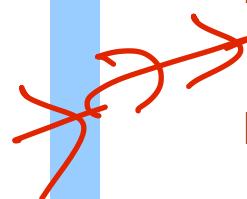
# Monitor with Condition Variables





# Liveness

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.
- Waiting indefinitely violates the progress and bounded-waiting criteria discussed at the beginning of this chapter.
- Liveness refers to a set of properties that a system must satisfy to ensure processes make progress.
- Indefinite waiting is an example of a liveness failure.



Main Objectives  
So fast wavy  
process make  
progress





# Liveness

*↙ the opposite of liveness*

- ✓ ■ **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let  $S$  and  $Q$  be two semaphores initialized to 1

$P_0$

```
wait(S) ;  
wait(Q) ;  
...  
signal(S) ;  
signal(Q) ;
```

$P_1$

```
wait(Q) ;  
wait(S) ;  
...  
signal(Q) ;  
signal(S) ;
```

- Consider if  $P_0$  executes  $\text{wait}(S)$  and  $P_1$   $\text{wait}(Q)$ . When  $P_0$  executes  $\text{wait}(Q)$ , it must wait until  $P_1$  executes  $\text{signal}(Q)$
- However,  $P_1$  is waiting until  $P_0$  execute  $\text{signal}(S)$ .
- Since these  $\text{signal}()$  operations will never be executed,  $P_0$  and  $P_1$  are **deadlocked**.





# Liveness

- Other forms of deadlock:
- **Starvation** – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
- Solved via **priority-inheritance protocol**



# End of Chapter 6

