



# Chapter 6

## Synchronization Tools

**Da-Wei Chang**

**CSIE.NCKU**

# Outline

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

# Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
  - cooperating processes: processes that access the shared data
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers.
  - use an integer count
    - keeps track of the number of full buffers
    - Initially set to 0
    - Incremented by the producer after it produces a new buffer
    - Decrement by the consumer after it consumes a buffer

# Producer

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

# Consumer

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

# Race Condition

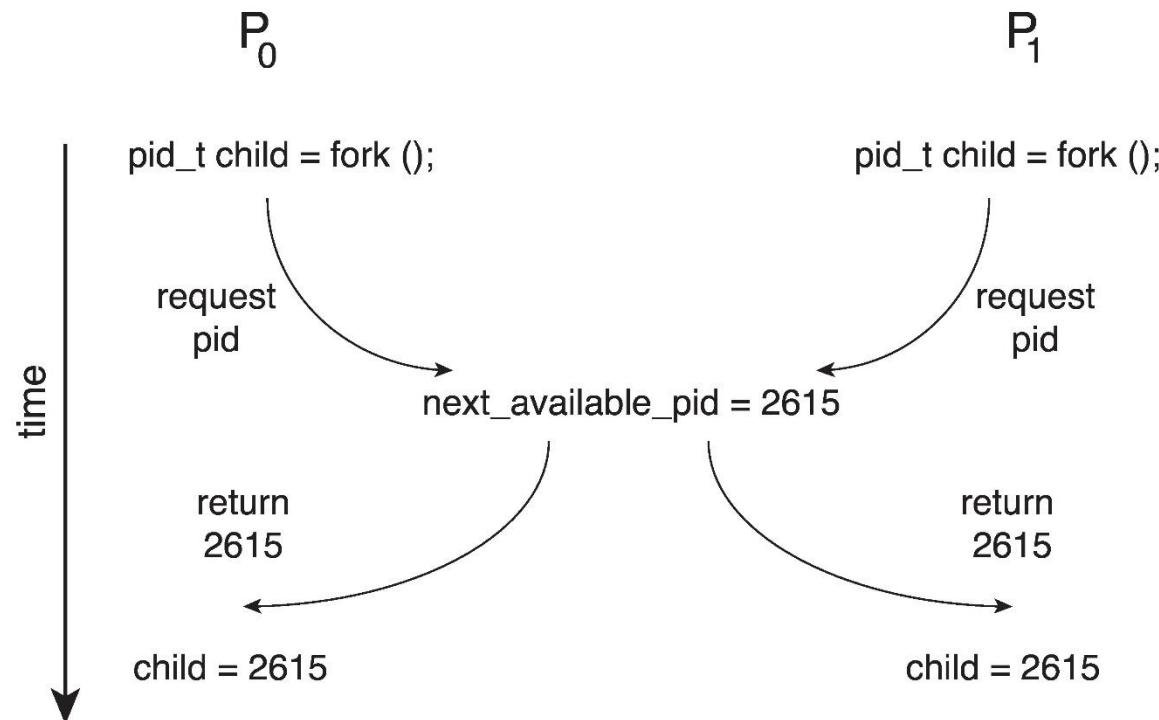
- `count++` could be implemented as  
`register1 = count`  
`register1 = register1 + 1`  
`count = register1`
- `count--` could be implemented as  
`register2 = count`  
`register2 = register2 - 1`  
`count = register2`
- Consider this execution interleaving with “`count = 5`” initially:

S0: producer execute `register1 = count` {`register1 = 5`}  
S1: producer execute `register1 = register1 + 1` {`register1 = 6`}  
S2: consumer execute `register2 = count` {`register2 = 5`}  
S3: consumer execute `register2 = register2 - 1` {`register2 = 4`}  
S4: producer execute `count = register1` {`count = 6`}  
S5: consumer execute `count = register2` {`count = 4`}

→ ***Race condition...***

# Race Condition – Another Example

- 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

- Consider system of  $n$  processes  $\{p_0, p_1, \dots p_{n-1}\}$
- Each process has a **critical section**
  - A segment of code, in which the process may update shared data
- When a process is in the critical section, others cannot enter that section
  - no two processes in the same critical section at the same time
- Critical section problem
  - Design a protocol that processes can use to coordinate
    - Entry section: code to try/request to enter the critical section
    - Exit section: code that follows the critical section

# Critical Section

```
do  
{  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (TRUE)
```

# Solution to Critical-Section Problem

**A solution should satisfy the following requirements**

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
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 – guarantee a given process will finally be selected
  - Assume that each process executes at a **nonzero speed**
  - No assumption concerning relative speed of the N processes

# Peterson's Solution

- Two-process solution; a software based solution
- Assume that the LOAD and STORE 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 whether or not a process is **ready** to enter the critical section
  - **flag[i]** = TRUE implies that process **P<sub>i</sub>** is ready!

# Peterson's Solution

## Algorithm for Process $P_i$

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while ( flag[j] && turn == j); // you first  
  
    CRITICAL SECTION  
  
    flag[i] = FALSE; // exit  
  
    REMAINDER SECTION  
  
} while (TRUE);
```

## Algorithm for Process $P_j$

```
do {  
    flag[j] = TRUE;  
    turn = i;  
    while ( flag[i] && turn == i); // you first  
  
    CRITICAL SECTION  
  
    flag[j] = FALSE; // exit  
  
    REMAINDER SECTION  
  
} while (TRUE);
```

# Peterson's Solution

- Mutual exclusion is met
  - **turn** can either be **i** or **j**
- Progress and bounded-waiting are met
  - Once  $P_j$  exits, it sets  $\text{flag}[j]$  to false, allows  $P_i$  to enter
  - If  $P_j$  resets  $\text{flag}[j]$  to true, it must also set **turn** to **i**
    - Allow  $P_i$  to enter
  - $P_i$  will enter critical section after at most one entry by  $P_j$  (bounded-waiting)
  - Selection will not be postponed forever (progress)

# 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!

# Peterson's Solution

- Two threads share the data

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

- Thread 1 performs

```
while (!flag);  
print x;
```

- Thread 2 performs

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

- What is the expected output?

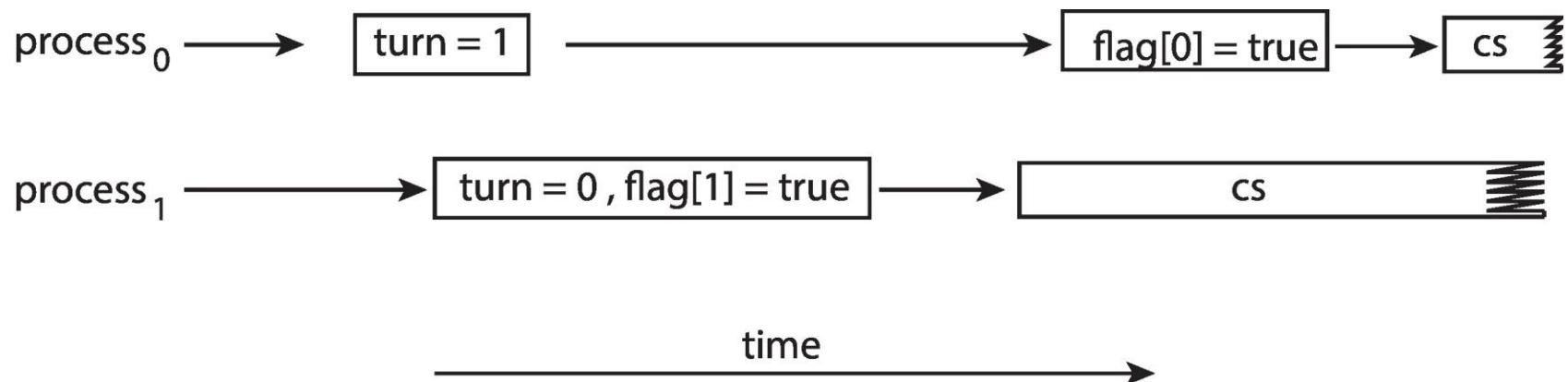
- 100? right?

# Peterson's Solution

- 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!**

# Hardware Support for Synchronization

- Many systems provide **hardware support** for synchronization
- Uniprocessor – could **disable interrupts**
  - Currently running code would execute without preemption
  - The approach used by **non-preemptive kernels** to protect shared data in the kernels
  - Generally too **inefficient** on multiprocessor systems
    - Need to ask other processors to disable interrupts
    - Operating systems using this not broadly scalable
  - Clocks may stop in the critical section

# Hardware Support for Synchronization

- We will look at three forms of hardware support
  1. Memory barriers
  2. Hardware instructions
  3. Atomic variables

# Memory Barriers

- Memory models
  - **Strongly ordered:** a memory modification of one processor is immediately visible to all other processors.
  - **Weakly ordered:** a memory modification of one processor may not be immediately visible to all other processors.
- A ***memory barrier* (or *memory fence*)**
  - an instruction (e.g., mfence for x86) that forces any change in memory to be propagated (made visible) to all other processors.
  - ensure that all loads/stores are **finished** before any subsequent load/store operations are performed

# 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(); //flag is loaded before x
    print x;
```

- Thread 2 now performs

```
x = 100;
memory_barrier(); // x is assigned before flag
flag = true;
```

# Using Memory Barriers for Peterson's Solution

- Place a memory barrier between the first two assignment statements

```
flag[i] = TRUE;  
memory_barrier();  
turn = j;
```

# Critical Sections and Locks

do

{

Acquire lock

Critical section...

Release lock

Remainder section...

} while (TRUE)

# Synchronization Hardware

- Modern machines provide special **atomic hardware instructions** for locks
  - Atomic = non-interruptable
- Test-and-Set instruction
- Compare-and-Swap instruction
- Swap instruction

# test\_and\_set Instruction

## Definition:

```
boolean test_and_set (boolean *target)
```

```
{  
    boolean rv = *target;  
    *target = true;  
    return rv;  
}
```



**Done by a single instruction**

- Executed atomically
- Returns the original value of passed parameter
- 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) )          0→ 1  
        ; // do nothing                      1→ 1  
  
        ..... critical section here  
  
    lock = FALSE;  
  
        ..... remainder section here  
  
} 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;
}
```

## Done by a single instruction

- Executed atomically
- Returns the original value of **\*value**
- Set the **\*value** to the **new\_value** if **\*value == expected**. 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 here...

```
    lock = 0;
```

... remainder section here ...

```
}
```

# swap Instruction

- Definition

```
void swap (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

**Done by a single instruction**

# Solution using swap

- Shared Boolean variable **lock** initialized to **FALSE**; Each process has a **local** Boolean variable **key**.
- Solution:

```
do {  
    key = TRUE;  
    while (key == TRUE) Swap ( &lock, &key );  
    ..... critical section here  
    lock = FALSE;  
    ..... remainder section here  
} while (TRUE);
```

**Used for swapping with lock**

**Lock: TRUE, key: FALSE**

# Synchronization Hardware

- The above examples
  - do not satisfy bounded waiting...
- The following code satisfy all the three requirements

```
Boolean waiting[n], lock=FALSE;  
do {  
    waiting[i] = TRUE;           // wish to acquire the lock  
    key = TRUE;  
    while (waiting[i] && key)  key = TestandSet(&lock); // try to acquire the lock  
    waiting[i] = FALSE;  
    // .....critical section here.... //  
    j = (i+1)%n;  
    while ( (j!= i) && !waiting[j] ) j = (j+1)%n; // keep locked and allow the  
                                                    // next to come in  
    if ( j == i ) lock = FALSE;  
    else waiting[j] = FALSE;  
    // .....remainder section here.... //  
} while (TRUE);
```

# 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 updates** on **basic data types** such as integers and booleans.
  - Useful when the critical section contains the update of a single shared variable
- For example, the **increment()** operation on the atomic variable  $k$  ensures  $k$  is incremented without interruption:

**increment(& $k$ );**

# 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)));
}
```

**\*v = 5**

Thread 1	Thread 2
<b>temp = *v = 5</b>	---
---	<b>temp = *v = 5</b>
---	compare_and_swap(v, temp, temp+1); // *v = temp+1 = 6; return 5
compare_and_swap(v, temp, temp+1); // *v != temp, v = 6, return 6;	
<b>temp != 6; while(TRUE), next round</b>	
temp = *v = 6	
	temp == 5; while (FALSE)
	increment() finishes Return to calling procedure
compare_and_swap(v, temp, temp+1); // *v == temp+1=7, return 6;	
<b>temp == 6; while (FALSE)</b>	
increment() finishes Return to calling procedure	

# Mutex Locks

- Previous solutions not target for application programmers
- A less complicated syn. interface to programmers - **mutex lock**
- Protect a critical section by first **acquire()** a lock then **release()** the lock
  - ensure mutual exclusion
- **acquire()/release()** must be atomic
  - Usually implemented via hardware atomic instructions such as compare-and-swap
- This solution requires **busy waiting**
  - a **spinlock**

# Critical Sections and Locks

do

{

Acquire lock

Critical section...

Release lock

Remainder section...

} while (TRUE)

# Mutex Lock Definitions

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

```
release() {  
    available = true;  
}
```

- These two functions must be implemented atomically.
- Hardware instructions such as test-and-set can be used to implement these functions.

# Semaphore

- More general than mutex locks
- Typical semaphore implementations do **NOT** require busy waiting
- Semaphore  $S$  – **integer** variable
- Two standard operations modify  $S$ : **wait()** and **signal()**
  - Originally called **P()** and **V()**
- Can only be accessed via two indivisible (**atomic**) operations
  - `wait (S) {`  
    `while S <= 0                  // atomic for each iteration`  
    `; // no-op                // iteration: if S>0, S--; else, no-op and test again`  
    `S--;`  
    `}`
  - `signal (S) {`  
    `S++;                        // atomic`  
    `}`

# Semaphore as General Synchronization Tool

- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Same as **mutex locks**
- A counting semaphore **S** can be used to implement a binary semaphore

# Semaphore as General Synchronization Tool

- Provides mutual exclusion
  - Semaphore S; // initialized to 1
  - wait (S);  
    Critical Section  
    signal (S);
- Other usages on semaphores
  - Control the usage of resources with N instances
    - S is initialized to N
  - Synchronization between processes
    - S is initialized to 0

# Semaphore Implementation

- Previous semaphore definition requires busy waiting...
- Note that applications may spend lots of time in critical sections. In this case, busy waiting is not a good approach.
- Solution
  - Sleep/block instead of busy waiting

# Semaphore Implementation WITHOUT Busy Waiting

- In addition to the semaphore **value**, each semaphore has an associated **waiting queue**
- Require blocking/waking-up a process
  - **block**
    - switch process state to “waiting”
    - remove the process from the ready queue
  - **wakeup**
    - switch process state to “ready”
    - place the process in the ready queue
- Both operations may lead to the invocation of the scheduler

# Semaphore Implementation WITHOUT Busy waiting (Cont.)

- Implementation of wait:

```
wait (S){  
    value--;  
    if (value < 0) { // the value can be negative → # of processes waiting  
        add this process to the waiting queue  
        block(); /* suspend the process */ }  
}
```

- Implementation of signal:

```
signal (S){  
    value++;  
    if (value <= 0) {  
        remove a process P from the waiting queue  
        wakeup(P); }  
}
```

# Semaphore Implementation

- The list of waiting processes?
  - A list of PCBs
  - FIFO → ensure bounded waiting
- Must guarantee that no two processes can execute `wait()` and `signal()` on the same semaphore at the same time
  - The implementation in the previous slide should be **atomic**
- Thus, implementation becomes the **critical section problem** where the ***wait and signal code are placed in the critical section.***
  - For uniprocessor → disable interrupt
  - For MP → typically use a **busy waiting** approach., e.g. `test_and_set`
    - Acceptable for busy waiting in critical sections of the `wait()` and `signal()` operations since they are short...

# Problems with Semaphores

- Using semaphores incorrectly can result in bugs that are hard to detect
- Incorrect use of semaphore operations:
  - signal (mutex) .... wait (mutex) → race
  - wait (mutex) ... wait (mutex) → deadlock (*described later*)
  - Omitting of wait (mutex) or signal (mutex) (or both) → race or deadlock
- The bugs are hard to detect
  - Faults don't always take place

# Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization

```
monitor monitor-name
{
    ... shared variable declarations

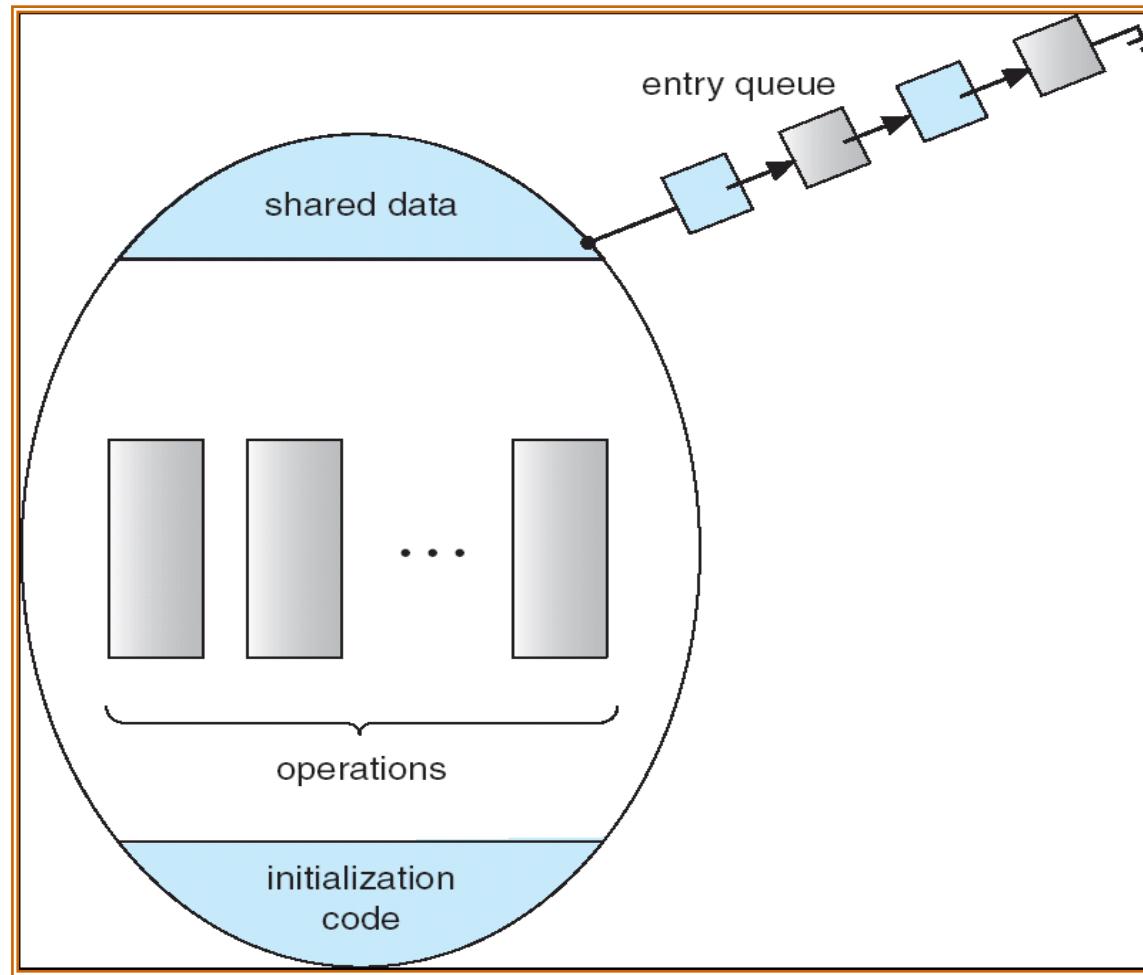
    procedure P1 (...) { .... }

    ...
    procedure Pn (...) {.....}
    Initialization code ( ....) { ... }

    ...
}
```

- Programmers provide the operations
- Monitor makes sure **only one process** may be **active within the monitor at a time**

# Schematic View of a Monitor



A list of processes waiting for entering the monitor

# Monitors

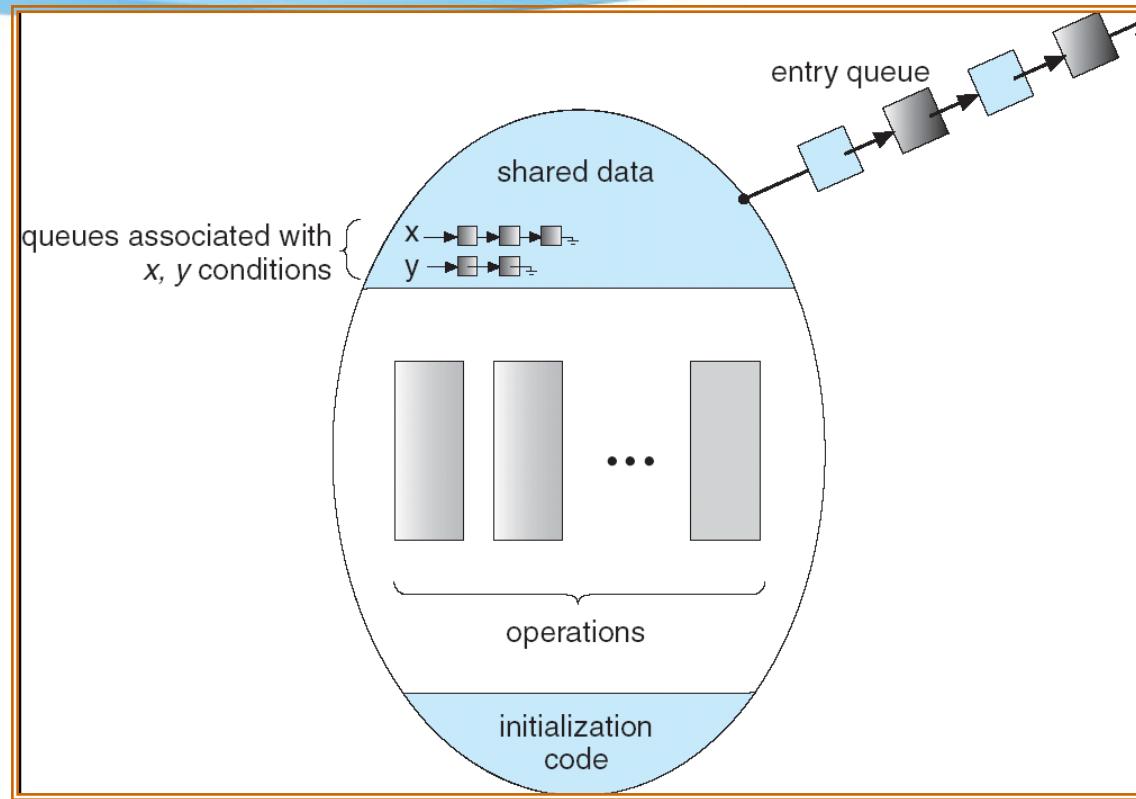
- With monitors, programmers do not need to code the synchronization constraint
- However, the function provided by monitors is limited
  - In many cases, programmers still need to define additional syn. mechanisms
    - Use condition variables

# Condition Variables

- condition x, y;
- Two operations on a condition variable
  - x.wait()
    - a process that invokes the operation is suspended
  - x.signal()
    - resumes **one** of processes (if any) that invoked x.wait ()
    - No effect if no one is waiting
- No values, just **suspend** and **resume**
  - Different from semaphores

***Using them incorrectly are still hard-to-be-detected bugs***

# Monitor with Condition Variables



If Q wait X and then P signal X

- P is now in monitor, but it will make Q become active in the monitor
- Two possibilities
  - A. **signal & wait**: Q becomes active immediately, P leaves
  - B. **signal & continue**: Q becomes active after P leaves

# Implementing a Monitor Using Semaphores

- Use a semaphore **mutex** (init. as 1) for each monitor
  - wait(mutex) before entering the monitor
  - signal(mutex) after leaving the monitor
- A signaling process may wait until the resumed process to leave or wait
  - Use another semaphore **next** (init. as 0)
    - Signaling process can suspend on it
  - **Next\_count**
    - Count the number of processes suspend on **next**

# Implementing a Monitor Using Semaphores

- Each external procedure F is replaced by  
wait (mutex);

F

```
if (next_count >0 )      signal(next);    ← Wakeup a signaling process
else                      signal(mutex);   ← Allow another process to
                                         come in
```

- Each condition variable is implemented by using
  - x\_sem : semaphore corresponds to x (init. as 0)
  - x\_count: number of processes waiting on x\_sem

# Implementing a Monitor Using Semaphores

```
x.wait()
{
    x_count++;
    if(next_count >0 )      signal(next);
    else                      signal(mutex);
    wait(x_sem);             ← wait here....
    x_count--;               ← have been waken
}

x.signal()
{
    if(x_count >0 )
    {
        next_count++;
        signal(x_sem);
        wait(next);           ← wait here....
        next_count--;
    }
}
```

# Process Resuming Order

- Which process (waiting for the monitor) should be resumed first?
  - FCFS
  - Priority
    - Users can specify priority on suspension
      - `x.wait(priority)`
      - Process with the highest priority is scheduled next

# Single Resource Allocation

- Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource

**R.acquire(t);**

...

**access the resource;**

...

**R.release();**

- Where R is an instance of type **ResourceAllocator**
  - *See next slide*

# A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
```

```
{
```

```
    boolean busy; //whether resource is busy or not  
    condition x; // wait on x if resource busy
```

```
    void acquire (int time) {  
        if (busy)      x.wait(time);  
        busy = true;  
    }
```

```
    void release () {  
        busy = FALSE;  
        x.signal();  
    }  
    initialization code() { busy = false; }  
}
```

# Incorrect Use of Monitor

- Access a resource without being granted
- Never release the resource
- Release a resource that the process doesn't have
  - Release the resource twice
- Monitors make sure only one process is in it at a time, but it can not solve all the problems related to resource allocation/deallocation

# Liveness

- Processes may have to wait indefinitely while trying to acquire a lock
- 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.
- Examples of liveness failure
  - Deadlock
  - Starvation
  - Priority inversion

# Liveness Failures

- **Deadlock** – two or more processes are waiting infinitely 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);

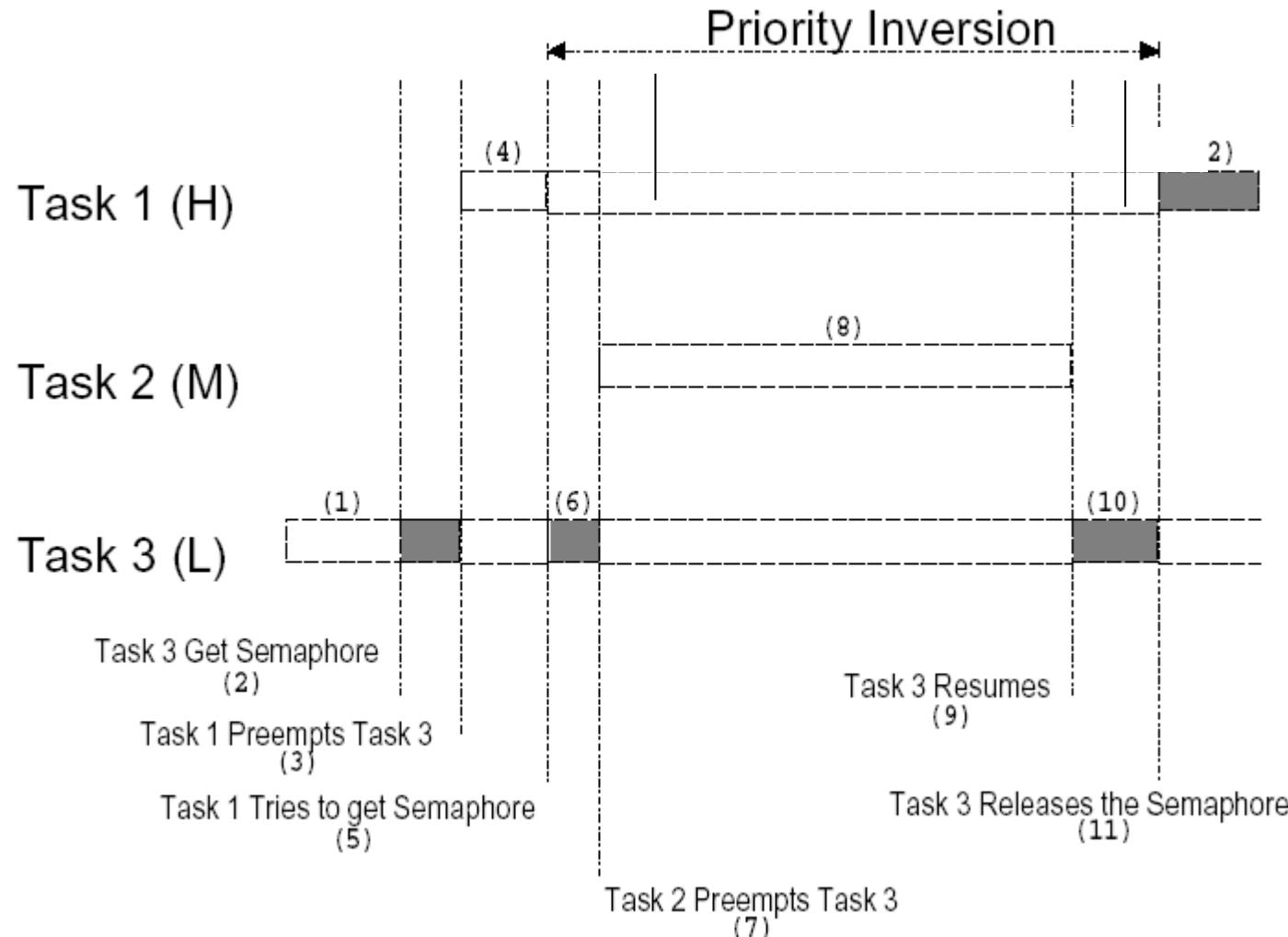
- Consider if  $P_0$  executes wait(S) and  $P_1$  wait(Q). When  $P_0$  executes wait(Q), it must wait for  $P_1$ . However,  $P_1$  also waits for  $P_0$  when it executes wait(S).

# Liveness Failures

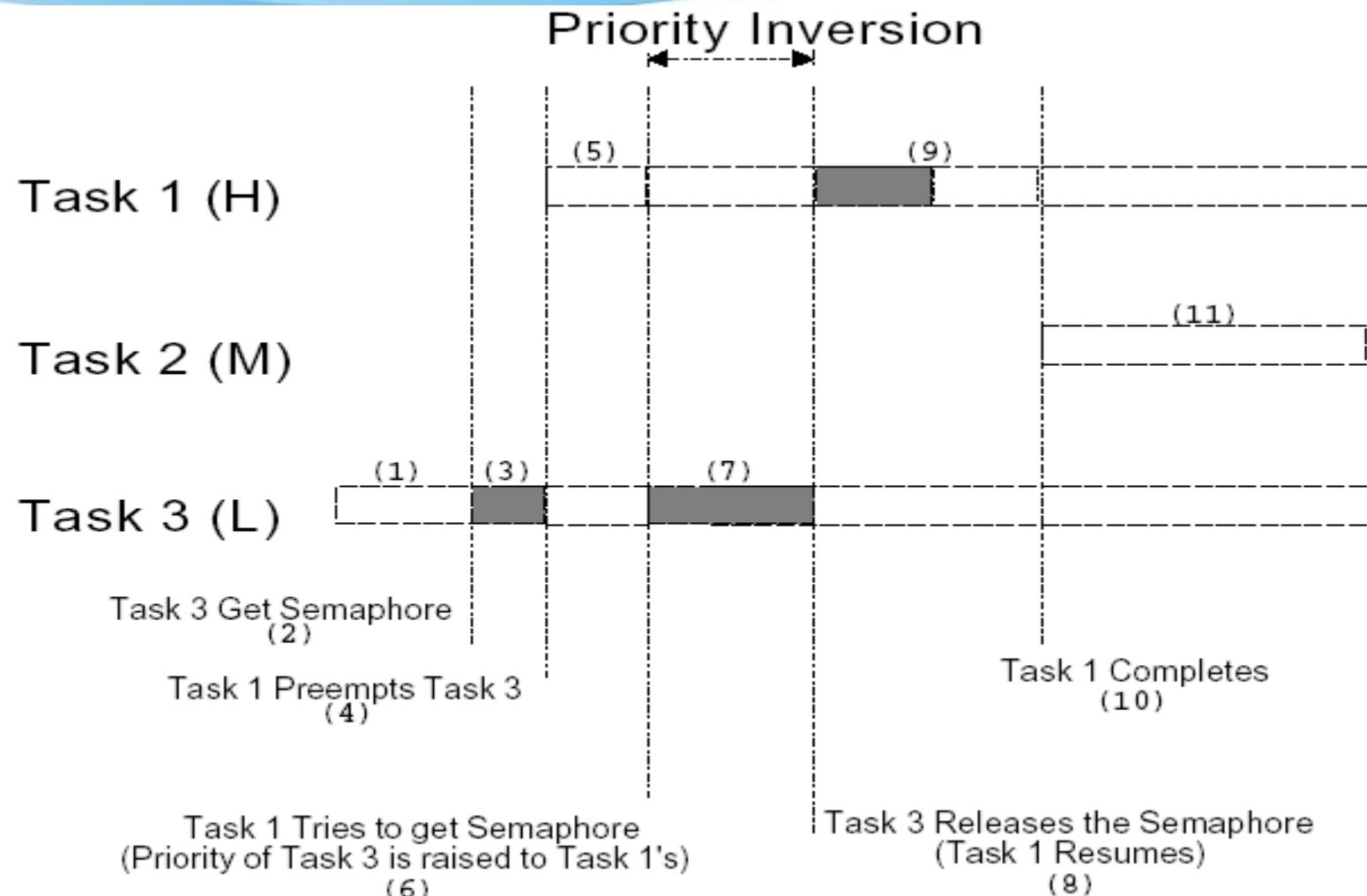
- **Starvation** – infinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
- **Priority Inversion** – scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - can be solved by **priority-inheritance protocol**

# Priority Inversion

- A problem in real-time systems



# Priority Inheritance Protocol (PIP)



*Some level of priority inversion can not be avoided!!*