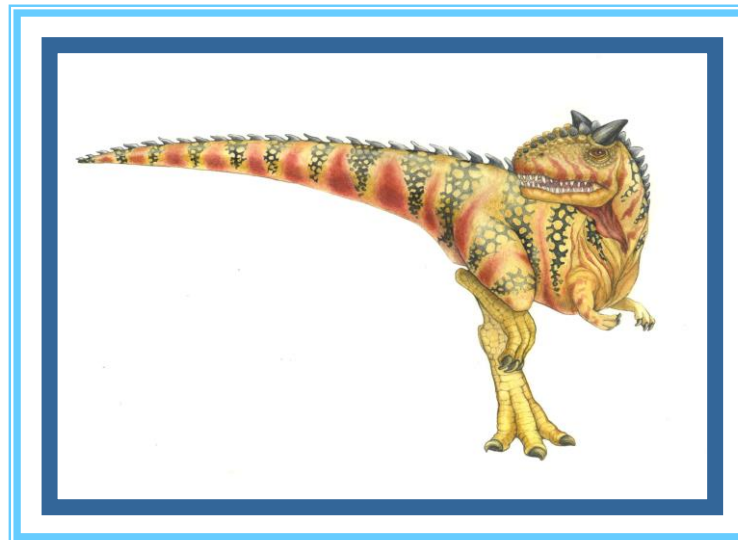


# Chapter 6: Process Synchronization

## Chapter 7: Synchronization Examples

---



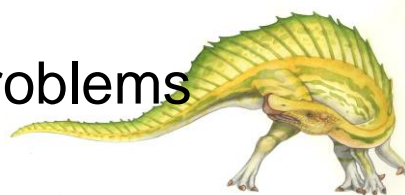


# Chapter 6: Process Synchronization

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

## ■ Objective

- 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





# 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.





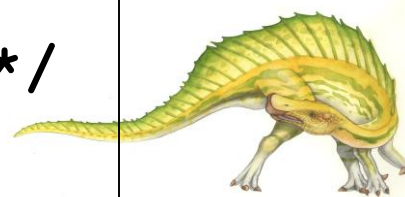
# Producer – Consumer Problem

## 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

- **counter++** could be implemented as

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

- **counter--** could be implemented as

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

**Race condition:** several processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order where the access takes place.

- 4

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

S0: producer execute	<b>register1 = counter</b>	{register1 = 5}
S1: producer execute	<b>register1 = register1+1</b>	{register1 = 6}
S2: consumer execute	<b>register2 = counter</b>	{register2 = 5}
S3: consumer execute	<b>register2 = register2-1</b>	{register2 = 4}
S4: producer execute	<b>counter = register1</b>	{counter = 6}
S5: consumer execute	<b>counter = register2</b>	{counter = 4}





# Critical Section Problem

■ Consider system of  $n$  processes  $\{p_0, p_1, \dots, p_{n-1}\}$

■ Each process has **critical section** 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**

General structure of process  $p_i$  is

```
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
  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
- Critical sections in operating system:
- **Preemptive kernel** – allows preemption of process when running in kernel mode
    - ▶ More responsive, good for real-time applications. → prone to have race condition
  - **Non-preemptive kernel** – runs until exits kernel mode
    - ▶ Essentially free of race conditions in kernel mode





# Peterson's Solution

- Good algorithmic description of solving the problem
- **Two-process** solution
- The processes share two variables:
  - `int turn;`
  - `Boolean flag[2]`
    - ▶ `turn`: indicates whose turn it is to enter the critical section
    - ▶ `flag[2]`: indicates if a process is ready to enter the critical section.
    - ▶ `flag[i] = true` → implies that process  $P_i$  is ready!
- Provable that
  1. Mutual exclusion is preserved
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met







# Peterson's Solution

## Algorithm for Process $P_0$

```
do {  
    flag[ 0 ] = true;  
    turn = 1;  
    while( flag[ 1 ] && turn == 1 );  
    critical section  
    flag[ 0 ] = false;  
  
    remainder section  
} while (true);
```

## Algorithm for Process $P_1$

```
do {  
    flag[ 1 ] = true;  
    turn = 0;  
    while( flag[ 0 ] && turn == 0 );  
    critical section  
    flag[ 1 ] = false;  
  
    remainder section  
} while (true);
```



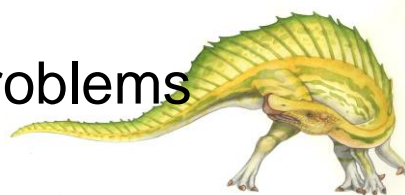


# Chapter 6: Process Synchronization

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

## ■ Objective

- 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





# Synchronization Hardware

- Many systems provide hardware support for critical section code
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
- Modern machines provide special atomic hardware instructions
  - ▶ **Atomic** = non-interruptible
  - *test\_and\_set instruction*
  - *compare\_and\_swap instruction*

**Solution to Critical-section Problem Using Locks**



```
Do {  
    acquire lock  
    critical section  
    release lock  
    ....  
    remainder section  
} while (TRUE);
```





# test\_and\_set Instruction

## ■ Definition:

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

⏪ *atomic  
instruction*

## ■ Solution: Shared boolean variable **lock**, initialized to **FALSE**

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

**Mutual exclusion ?**  
**Progress ?**  
**Bounding waiting ?**

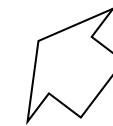




# Solution using compare\_and\_swap

## ■ Definition:

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



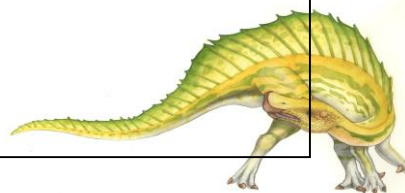
***atomic  
instruction***

## ■ Solution:

**Mutual exclusion ?**  
**Progress ?**  
**Bounding waiting ?**

Shared Boolean variable ***lock*** initialized to **FALSE**;

```
do {  
    while( compare_and_swap( &lock, 0, 1 ) != 0 )  
        ; /*do nothing*/  
    . . . /* critical section */  
    lock = 0;  
    . . . /* remainder section */  
} while (true);
```





# Bounded-waiting Mutual Exclusion with test\_and\_set

```
boolean waiting[n];  
boolean lock;
```

Both are initialized to **false**

Bounded waiting

```
do {  
    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;  
    /* remainder section */  
} while ( true );
```

← No one is waiting  
when i leaves

← Someone is waiting  
when i leaves





# 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**
- Product critical regions with it by first **acquire()** a lock then **release()** it
  - **Boolean variable** indicating if lock is available or not
  - Update to lock must be atomic
    - ▶ Usually by hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

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


critical section

  
    release lock  


remainder section

  
} while (true);
```







# Semaphore

- Semaphore **S** – **integer variable**
- Two standard operations modify **S**: **wait()** and **signal()**
  - Originally called **P()** and **V()**
- Less complicated
- Update to variable must be indivisible (atomic) operations

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

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





# Semaphore Usage

- **Counting semaphore** – integer value not only 0 or 1
  - Control access to a resource consisting of a finite number of instances
  - The semaphore is initialized to the **number of resources available**
  - `wait( )` : decrements the count
  - `signal()` : increases the count
  - `count = 0` : all resources are being used and processes that wish to use a resource will block until the count becomes  $> 0$ .
  
- **Binary semaphore** – integer value can range only between 0 and 1
  - Then a **mutex lock**





# Semaphore Usage

- Not only for resource mutual exclusive, but also for synchronization problems
- Consider  $P_1$  and  $P_2$  that require  $S_1$  to happen before  $S_2$ 
  - The common Semaphore **synch** is initialized to **0**

P1:

$S_1$ ;  
**signal( synch );**

P2:

**wait( synch );**  
 $S_2$ ;

Recall:

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

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

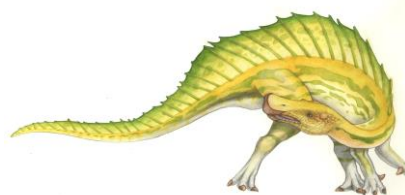
- Because synch is initialized to 0, P2 will execute  $S_2$  only after P1 has invoked signal(synch) which will increase synch to 1.





# Semaphore Implementation with no Busy waiting

- The implementation using busy waiting is not a good solution because applications may spend lots of time in critical sections.
- Solution with no busy waiting
  - 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





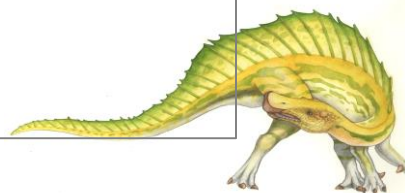
# Semaphore Implementation with no Busy waiting

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

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

- if  $S \rightarrow \text{value}$  is negative  
→ its magnitude is the number of processes waiting on that semaphore
- To ensure *bounded waiting* :  
use **FIFO** for the waiting list.

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





# Deadlock, Starvation, Priority Inversion

- **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

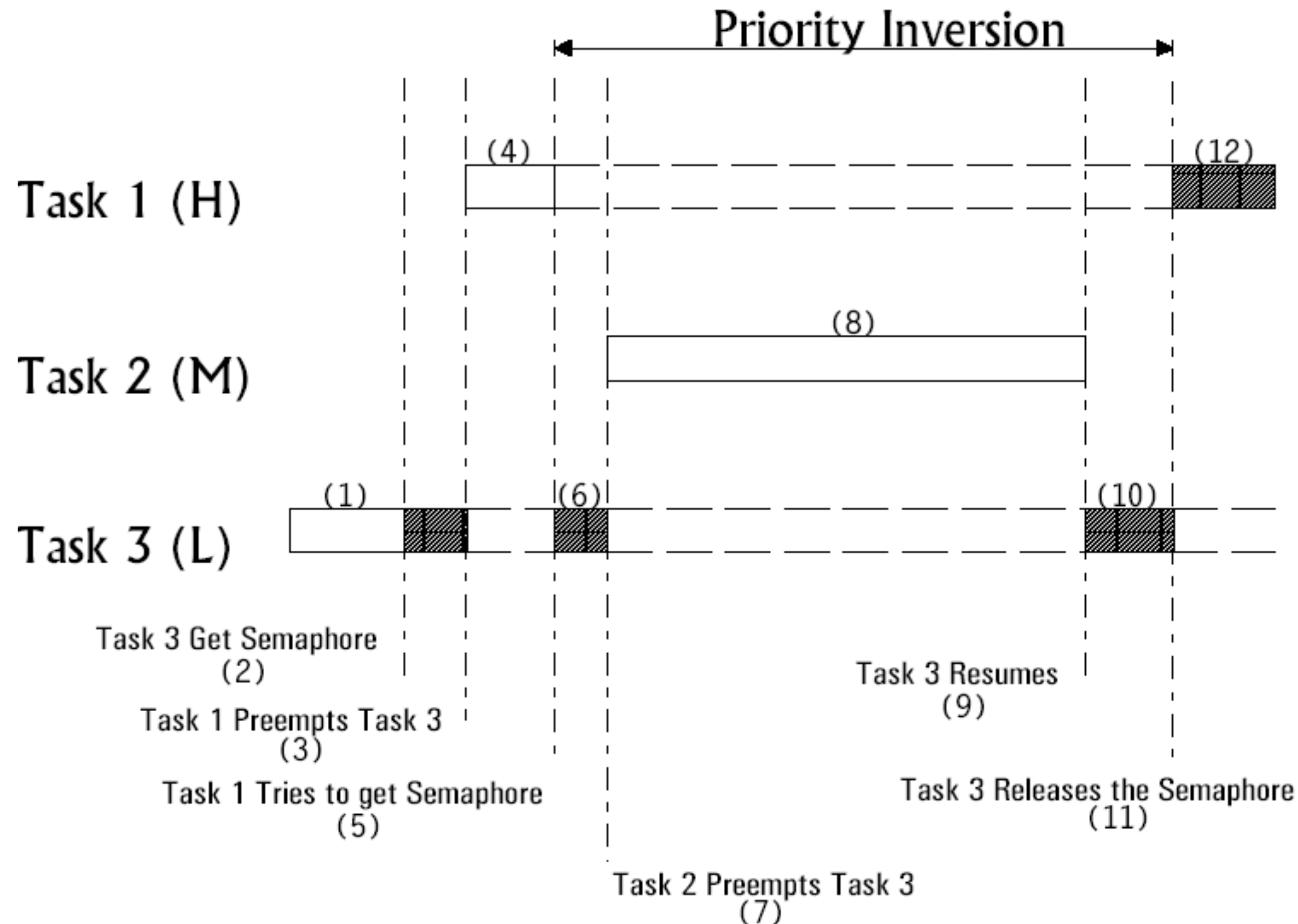
$P_0$	$P_1$
wait(S);	wait(Q);
wait(Q);	wait(S);
▪	▪
signal(S);	signal(Q);
signal(Q);	signal(S);

- **Starvation – indefinite blocking**
  - A process may never be removed from the semaphore queue
  - It may occur if we remove processes from the list in LIFO order.
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance, or priority ceiling protocol**





# Priority Inversion Problem







# Priority Inversion Problem - Solution

## ■ Priority inheritance

- A low-priority task that holds the lock requested by a high-priority task temporarily “*inherits*” the priority of that high-priority task, from the moment the high-priority task does the request.
- So, the L-task won't be preempted by the M-task, and can finish its critical section without holding up H-task any longer than needed.
- When L-task releases the lock, its priority drops to its original level.
- It generates *run-time* overhead, because the scheduler has to check the priorities of all tasks that access a lock.

## ■ Priority ceiling

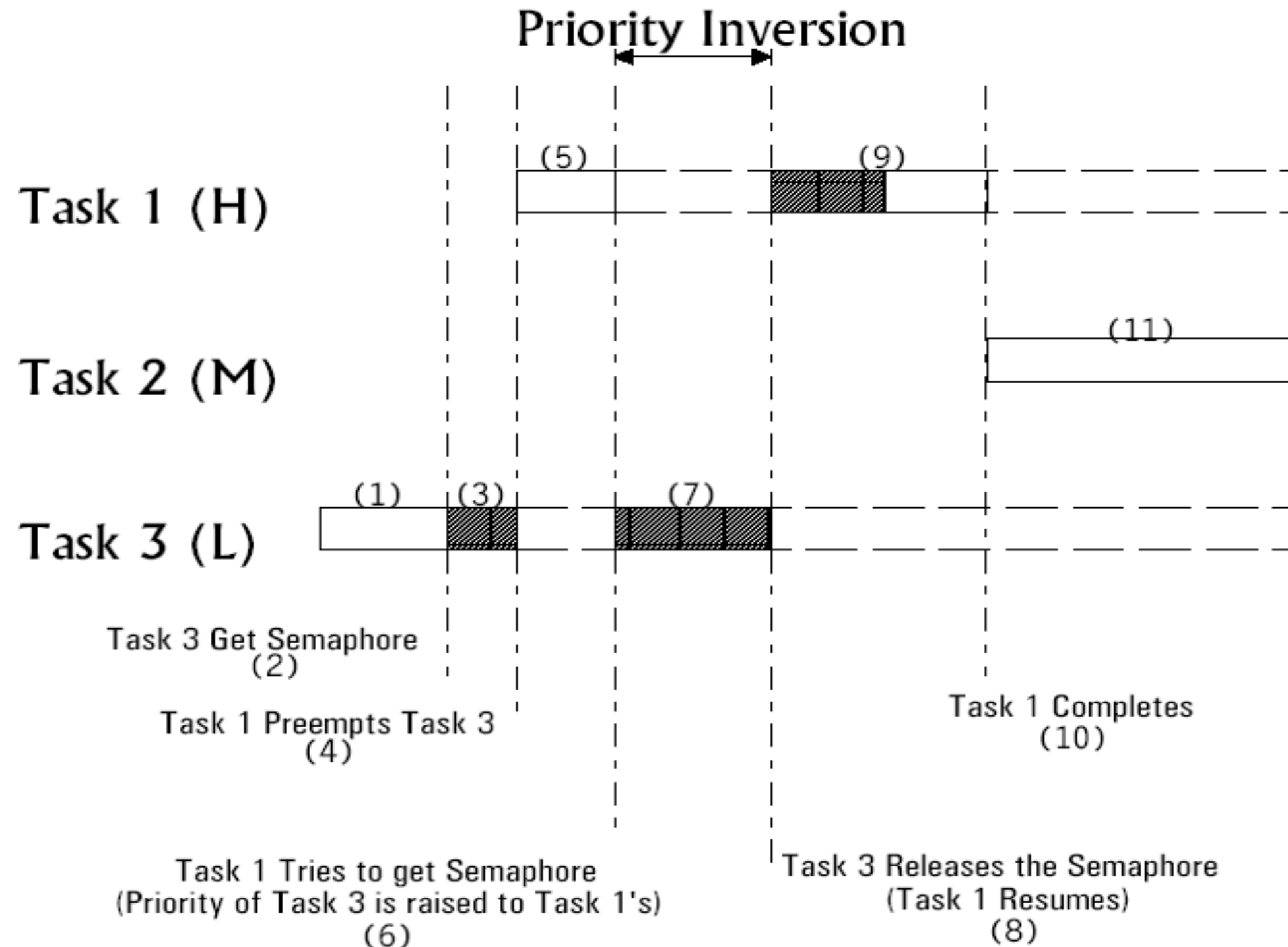
- Every lock gets a priority level equal to the priority of the highest-priority task that *can* use the lock. This level is called *ceiling priority*.
- when L-task enters the critical section, it immediately gets ceiling priority from the lock, so it will not be preempted by any M-task.
- It generates *compile-time* overhead only.
  - ▶ The priority is changed *no matter* another task requests the lock or not.
  - ▶ That makes the L-task run at higher priority for longer time than needed.





# Priority Inversion problem

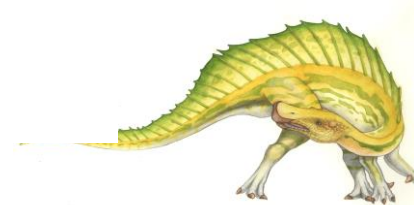
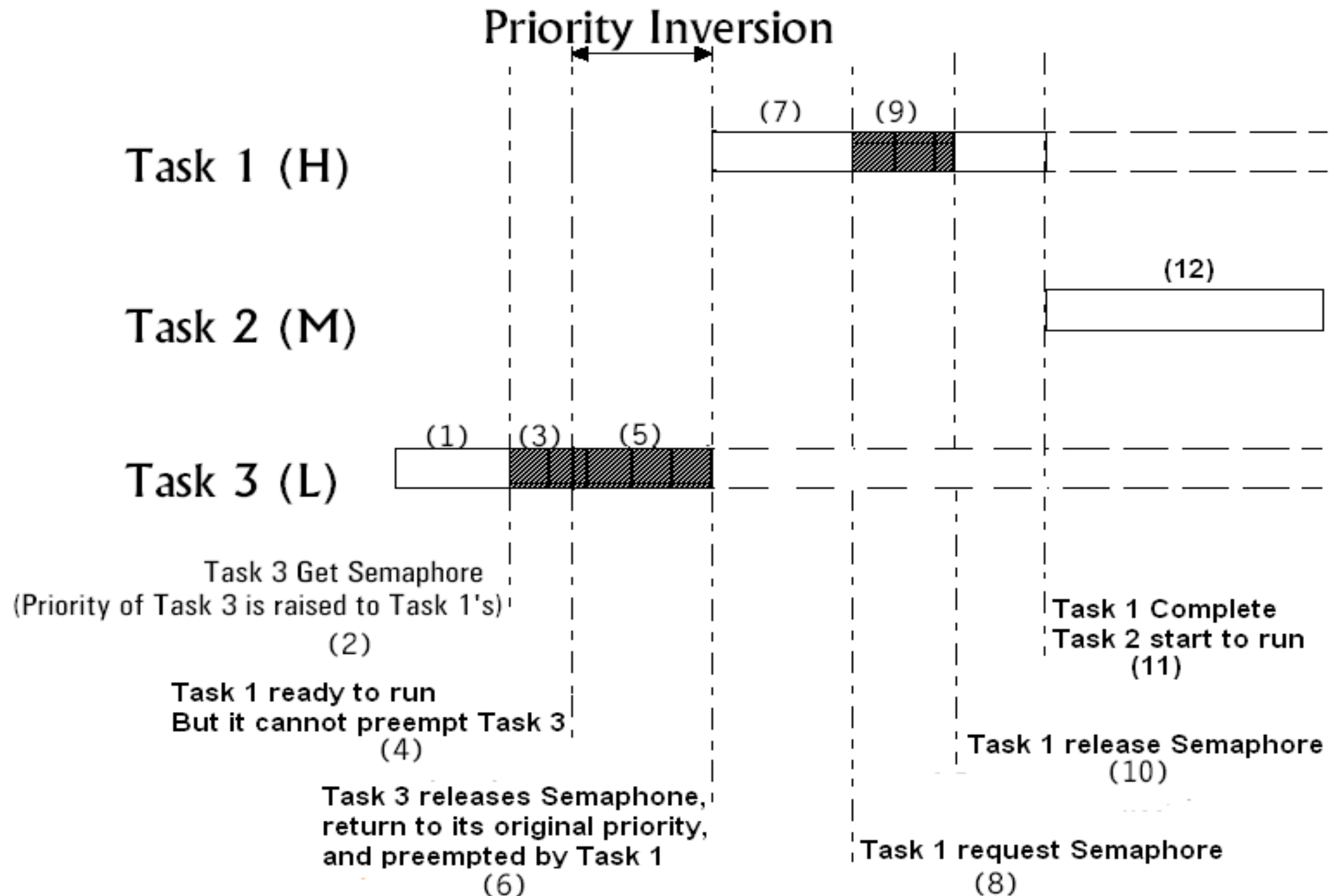
## – Solved by **Priority Inheritance**





# Priority Inversion problem

## – Solved by Priority Ceiling





# Problems with Semaphores

- Incorrect use of semaphore operations:
  - **signal (mutex) .... wait (mutex)**  
→ violate mutual-exclusion
  - **wait (mutex) ... wait (mutex) → deadlock**
  - **Omitting of wait (mutex) or signal (mutex) (or both)**  
→ either violate mutual-exclusion or deadlock
- Deadlock and starvation





# Chapter 6: Process Synchronization

---

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors





# 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  
→ programmer does not need to code mutual exclusion explicitly
- But not powerful enough to model some synchronization schemes

**monitor monitor-name**

**{**

**// shared variable declarations**

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

**procedure Pn (...) {.....}**

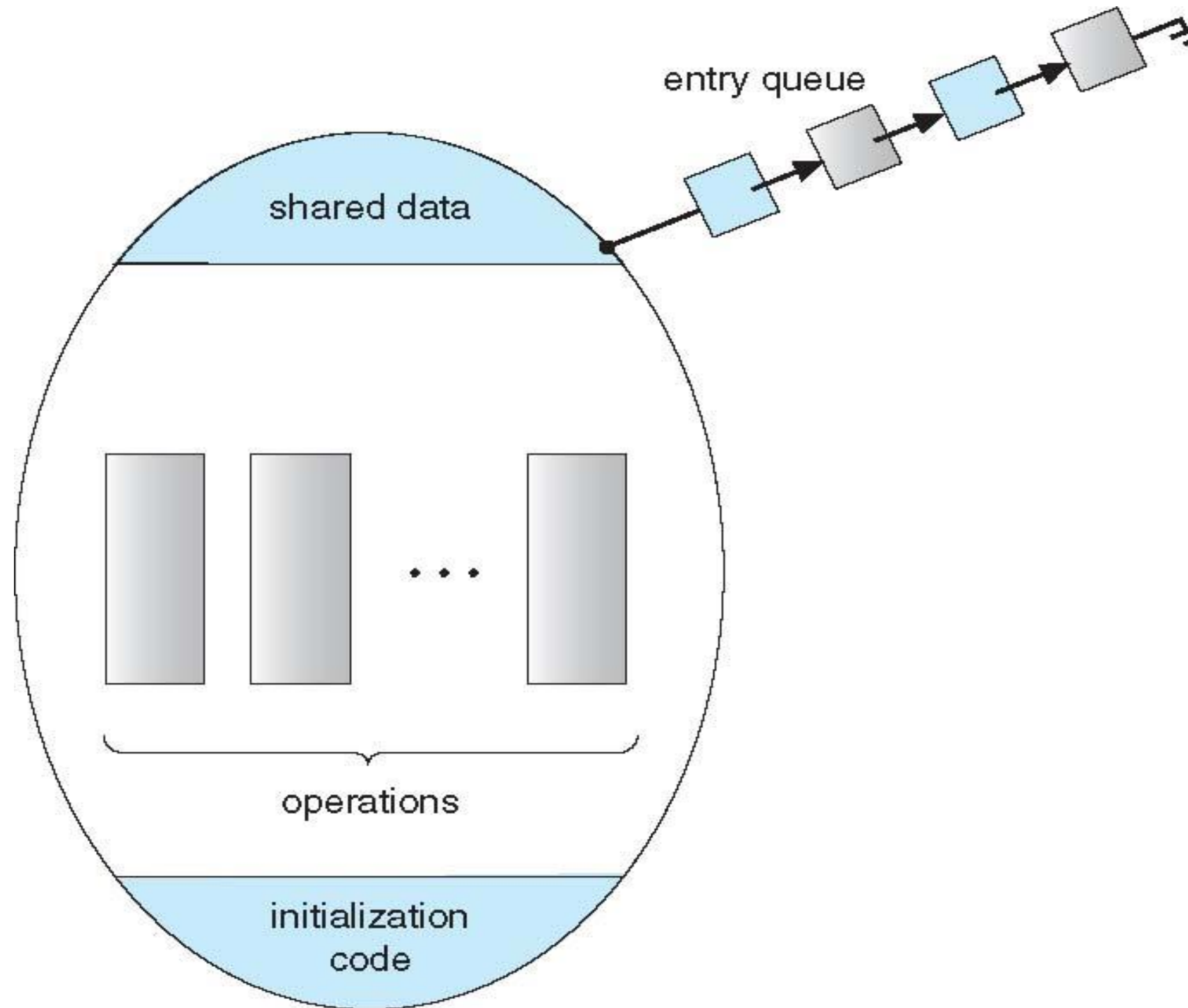
**Initialization code (...) { ... }**

**}**





# Schematic view of a Monitor







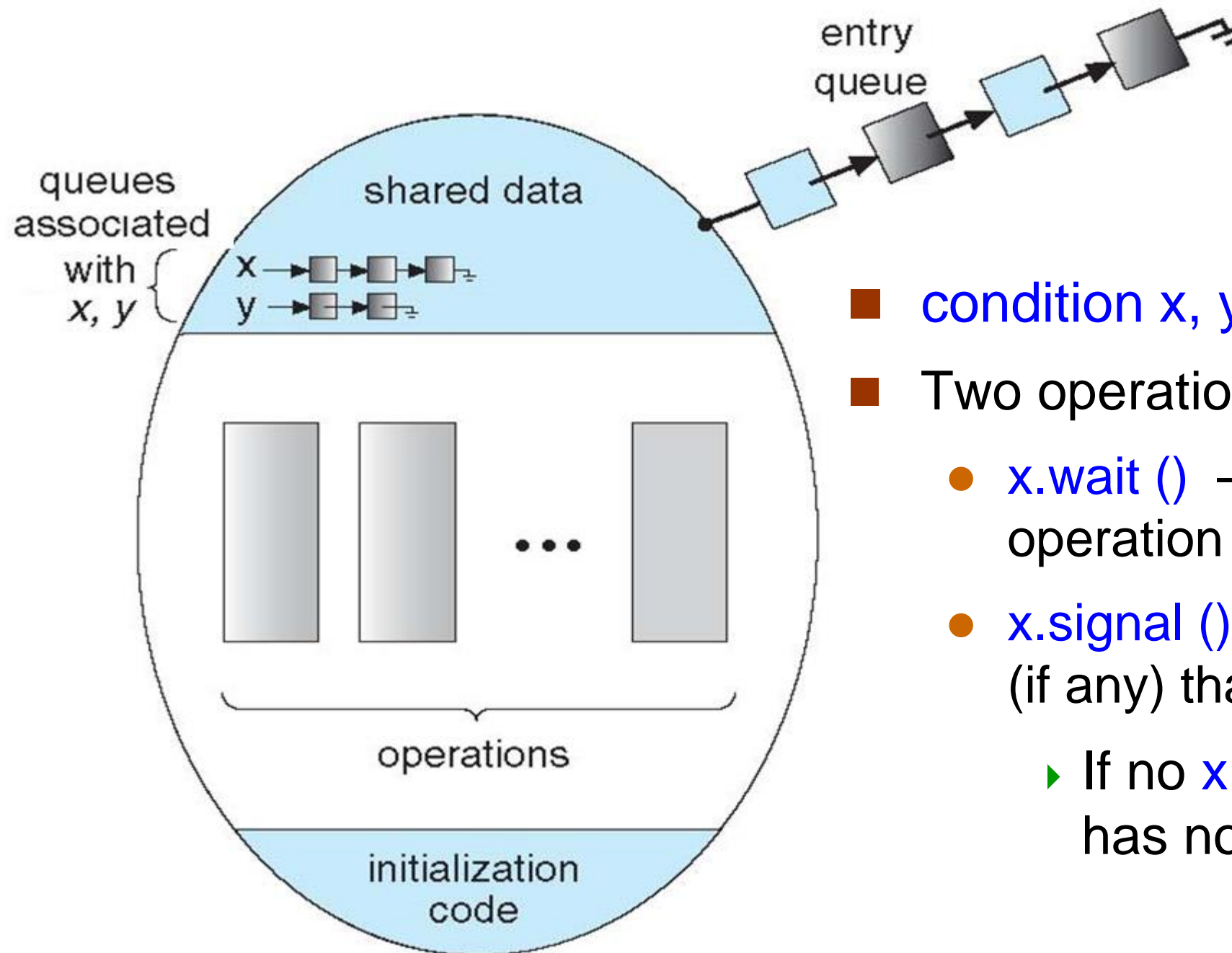
# Condition Variables

- condition `x, y;`
- Two operations on a condition variable:
  - `x.wait ()` – the process that invokes this 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



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





# Condition Variables Choices

- If process P invokes `x.signal ()`, with Q in `x.wait ()` state, what should happen next?
  - If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q leaves monitor or waits for another condition
  - **Signal and continue** – Q waits until P leaves the monitor or waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - ▶ P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java





# Resuming Processes within a Monitor

- If several processes queued on condition x, and x.signal() executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form x.wait(c)
  - Where c is **priority number**
  - Process with lowest number (highest priority) is scheduled next





# Monitor Implementation Using Semaphores

- Variables

semaphore mutex; // (initially = 1)

For each monitor a semaphore is provided (mutex initialized to 1)

- Each procedure  $F$  will be replaced by

wait( mutex );

...

body of  $F$ ;

...

signal(mutex);

A process must execute wait (mutex) before entering the monitor, and must execute signal (mutex) when leaving the monitor.

- Mutual exclusion within a monitor is ensured





# Chapter 7: Synchronization Examples

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Monitors
- **Classic Problems of Synchronization**
  - **Bounded-Buffer Problem**
  - **Readers and Writers Problem**
  - **Dining-Philosophers Problem**
- Synchronization Examples





# Bounded Buffer Problem

## ■ The **producer** process

```
do {  
    ...  
    /* produce an item in  
       next_produced */  
    ...  
    wait(empty);  
    wait(mutex);  
    /* add next produced to  
       the buffer */  
    signal(mutex);  
    signal(full);  
    ...  
} while (true);
```

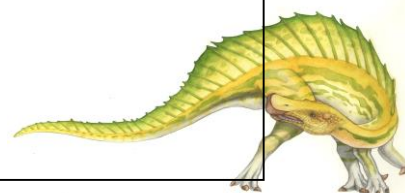
there are  $n$  buffers

```
semaphore mutex = 1;  
semaphore full = 0;  
semaphore empty = n;
```

} Shared data

## ■ The **consumer** process

```
do {  
    ...  
    wait(full);  
    wait(mutex);  
    /* remove an item from  
       buffer to next_consumed */  
    signal(mutex);  
    signal(empty);  
    ...  
    /* consume the item in next  
       consumed */  
} while (true);
```

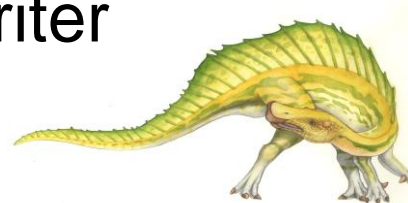






# Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- **Readers-Writers Problem Variations** - Several variations of how readers and writers are treated – all involve priorities
  - *First* variation – no reader kept waiting unless writer has permission to use shared object
  - *Second* variation – once writer is ready, it performs write asap
  - Both may have starvation leading to even more variations
  - Problem is solved on some systems by kernel providing reader-writer locks





# Readers-Writers Problem (Cont.)

Shared Data

data set

semaphore `rw_mutex` = 1;

semaphore `mutex` = 1;

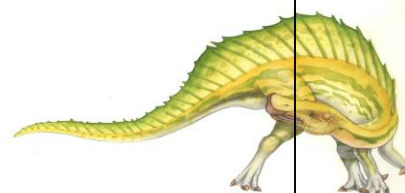
integer `read_count` = 0;

## ■ **Writer** process

```
do {  
    wait( rw_mutex );  
  
    ...  
    /* writing is performed */  
    ...  
    signal( rw_mutex );  
} while (true);
```

## ■ **Reader** process

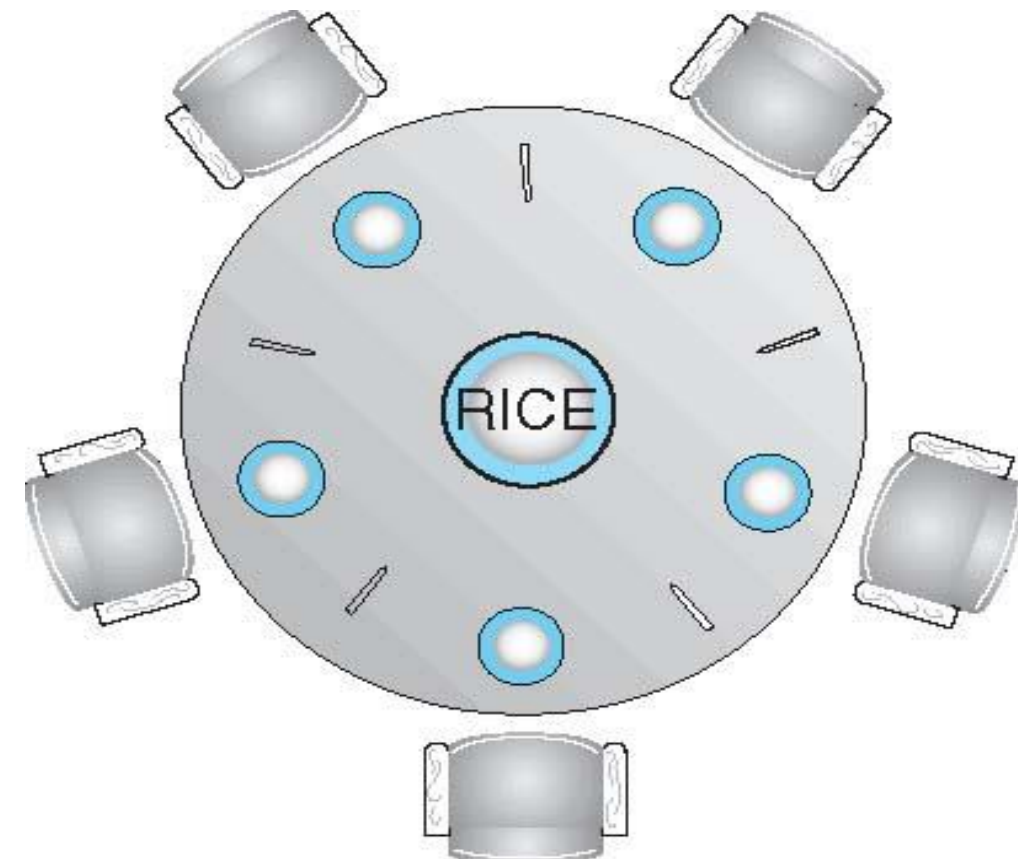
```
do {  
    wait( mutex );  
    read count++;  
    if ( read_count == 1 )  
        wait( rw_mutex );  
    signal( mutex );  
  
    ...  
    /* reading is performed */  
    ...  
    wait( mutex );  
    read count--;  
    if ( read_count == 0 )  
        signal( rw_mutex );  
    signal( mutex );  
} while (true);
```





# Dining-Philosophers Problem

- Philosophers spend their lives thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- It represent the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.
- In the case of 5 philosophers
  - Shared data
    - ▶ Bowl of rice (data set)
    - ▶ Semaphore **chopstick [5]** initialized to 1





# Dining-Philosophers Problem Algorithm

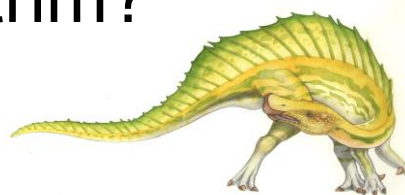
## Shared data:

Bowl of rice (data set)

Semaphore **chopstick** [5];  
// initialized to 1

```
■ The structure of Philosopher i:  
    do {  
        wait ( chopstick[i] );  
        wait ( chopStick[ (i + 1) % 5] );  
        // eat  
        signal ( chopstick[i] );  
        signal ( chopstick[ (i + 1) % 5] );  
        // think  
    } while (TRUE);
```

■ What is the problem with this algorithm?





# Dining-Philosophers Problem Algorithm

---

- What is the problem with the algorithm in previous page ?
  - Suppose all five philosophers become hungry at the same time and each grabs her left chopstick.  
→ deadlock
- Several possible solutions
  1. Allow at most four philosophers to be sitting at the table
  2. Allow philosopher to pick up her chopsticks only if both chopsticks are available (i.e., must pick them up in a critical section)
  3. Use an asymmetric solution –
    - odd-numbered philosophers: pick up left chopstick first and then right one
    - Even-numbered philosophers: pick up right chopstick first and then left one



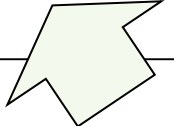


# Monitor Solution to Dining Philosophers

**monitor DiningPhilosophers**

```
{  
    enum { THINKING; HUNGRY, EATING) state [5] ;  
    condition self [5];  
  
    void pickup (int i) {  
        state[i] = HUNGRY;  
        test(i);  
        if (state[i] != EATING) self [i].wait;  
    }  
  
    void putdown (int i) {  
        state[i] = THINKING;  
        // test left and right neighbors  
        test((i + 4) % 5);  
        test((i + 1) % 5);  
    }  
}
```

To make sure two neighbors are not eating



```
void test (int i) {  
    if ( (state[(i + 4) % 5] != EATING) &&  
        (state[i] == HUNGRY) &&  
        (state[(i + 1) % 5] != EATING) ) {  
        state[i] = EATING ;  
        self[i].signal () ;  
    }  
}  
  
    }  
  
    initialization_code() {  
        for (int i = 0; i < 5; i++)  
            state[i] = THINKING;  
    }  
}
```



# Solution to Dining Philosophers (Cont.)

- Each philosopher  $i$  invokes the operations `pickup()` and `putdown()` in the following sequence:

**`DiningPhilosophers.pickup (i);`**

**EAT**

**`DiningPhilosophers.putdown (i);`**

- No deadlock, but starvation is possible
  - It is deadlock free because it imposes the restriction that a philosopher may pick up chopsticks only if both them are available.







# Chapter 7: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Monitors
- Classic Problems of Synchronization
- Synchronization Examples
  - **Solaris**
  - **Windows XP**
  - **Linux**
  - **Pthreads**





# Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutexes** for efficiency when protecting data from short code segments
  - Starts as a standard semaphore spin-lock
  - If lock held, and by a thread running on another CPU, spins
  - If lock held by non-run-state thread, block and sleep waiting for signal of lock being released (i.e., always sleep waiting on a single-processor system)
- Uses **condition variables**: for long sections of code
- Uses **readers-writers**: for long sections of code
  - To protect data that are accessed frequently in a read-only manner.





# Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides dispatcher objects, by which threads synchronize according to mechanisms like mutexes, semaphores, events, and timers
  - **Events**
    - ▶ An event acts much like a condition variable
  - **Timers** notify one or more thread when time expired
  - Dispatcher objects either in signaled-state (object available) or non-signaled state (object not available)
    - ▶ a thread will block when attempting to acquire the non-signaled object.





# Linux Synchronization

## ■ Linux:

- Prior to kernel Version 2.6, where kernel is nonpreemptive, disables interrupts to implement short critical sections
- Version 2.6 and later, kernel is fully preemptive (so a task can be preempted when it is running in the kernel)

## ■ Linux provides:

- semaphores
- spinlocks
- reader-writer versions of both

## ■ On SMP, spinlock is used for short code protection.

## ■ For a longer period, semaphores or mutex locks are appropriate to use.

## ■ On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption





# Pthreads Synchronization

---

- Pthreads API is available at user level (OS-independent)
- It provides:
  - mutex locks
    - ▶ `pthread_mutex_lock()`, `pthread_mutex_unlock()`;
  - condition variables
  - read-write locks
  - spinlocks

