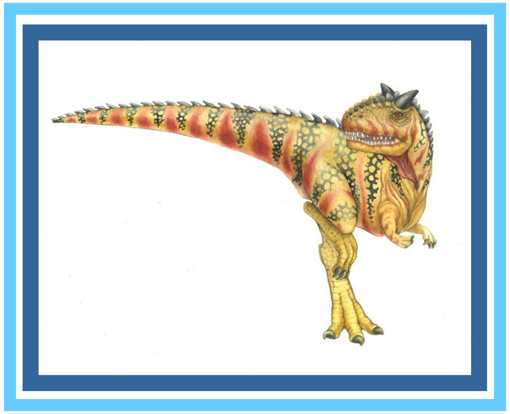
Synchronization Tools



Outline

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

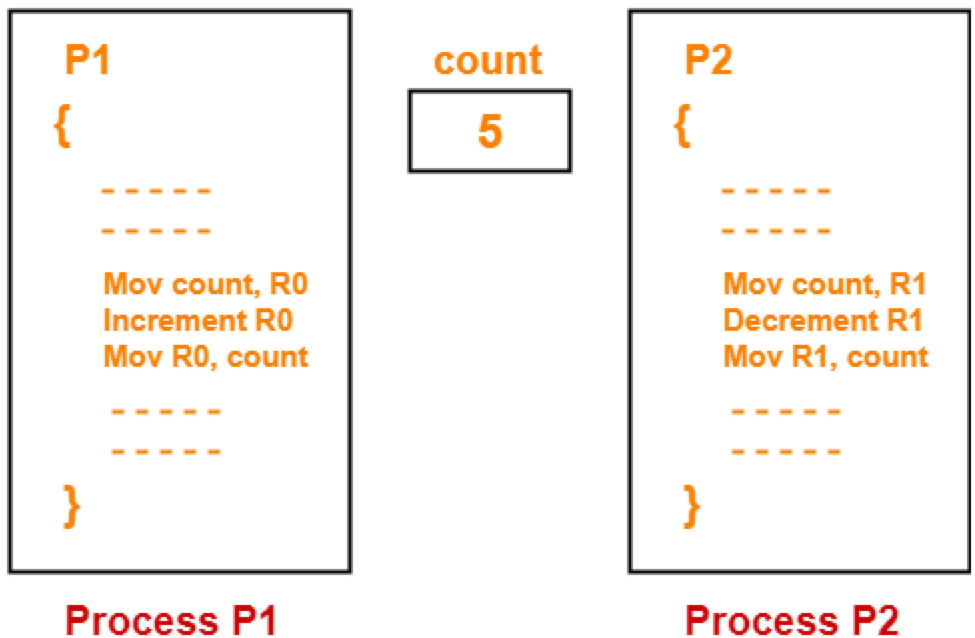
Objectives

- Describe the critical-section problem and illustrate a race condition
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical section problem
- Evaluate tools that solve the critical-section problem in low-, Moderate-, and high-contention scenarios

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
- We illustrated in problem when we considered the Bounded Buffer problem with use of a counter that is updated concurrently by the producer and consumer,. Which lead to race condition.

Race Condition



Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … 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);
```

Critical-Section Problem (Cont.)

Requirements for 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 process 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
   - No assumption concerning **relative speed** of the ***n*** processes

Interrupt-based Solution

- Entry section: disable interrupts
- Exit section: enable interrupts
- Will this solve the problem?
  - What if the critical section is code that runs for an hour?
  - Can some processes starve – never enter their critical section.
  - What if there are two CPUs?

Software Solution 1

- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share one variable:
  - **`int turn;`**
- The variable **`turn`** indicates whose turn it is to enter the critical section

Algorithm for Process P_i

```
while (true){

        turn = i;
        while (turn = = j);
                ;

        /* critical section */

        turn = j;

        /* remainder section */

}
```

Correctness of the Software Solution

- Mutual exclusion is preserved

  P_i enters critical section only if:

  turn = I

  and **turn** cannot be both 0 and 1 at the same time
- What about the Progress requirement?

  FOObar
- What about the Bounded-waiting requirement?

Peterson's Solution

- Software Solution
- 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;

        /* remainder section */

 }
```

Correctness of Peterson's Solution

- 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

Bakery Algorithm

```
lock(integer i) {
      Entering[i] = true;
      Number[i] = 1 + max(Number[1], ...,
Number[NUM_THREADS]);
      Entering[i] = false;
      for (integer j = 1; j <= NUM_THREADS; j++) {
          // Wait until thread j receives its number:
          while (Entering[j]) { /* nothing */ }
          // Wait until all threads with smaller numbers or
with the same
          // number, but with higher priority, finish their
work:
          while ((Number[j] != 0) && ((Number[j], j) <
(Number[i], i))) { /* nothing */ }
      }
  }

  unlock(integer i) {
      Number[i] = 0;
  }
```

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. Hardware instructions

2. Atomic variables

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 (uninterruptedly.)
  - **Test-and-Set** instruction
  - **Compare-and-Swap** instruction

The test_and_set Instruction

- Definition

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

- Properties
  - 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))
        ; /* do nothing */

              /* critical section */

     lock = false;
              /* remainder section */
 } while (true);
```

- Does it solve the critical-section problem?

Swap Instruction

- Definition

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

.

Solution using swap

- Shared integer **lock** initialized to false;
- Solution:

```
do {
        key = true;
        while (key == true)
        {
        swap (&lock, &key);
        )

        /* critical section */

    lock = false;

    /* remainder section */
} while (true)
```

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
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First **`acquire()`** a lock
  - Then **`release()`** the lock
- 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 CS Problem Using Mutex Locks

```
while (true) {
        acquire lock

            critical section

        release lock

remainder section
}
```

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes 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 (Cont.)

- **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 implement a counting semaphore ***S*** as a binary semaphore
- With semaphores we can solve various synchronization problems

Semaphore Usage Example

- Solution to the CS Problem
  - Create a semaphore "**mutex**" initialized to 1

```
wait(mutex);
    CS
signal(mutex);
```

- Consider P_1 and P_2 that with two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
  - Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

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

Semaphore Implementation 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

Implementation with no Busy waiting (Cont.)

- Waiting queue

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

Implementation with no Busy waiting (Cont.)

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

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

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.

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:

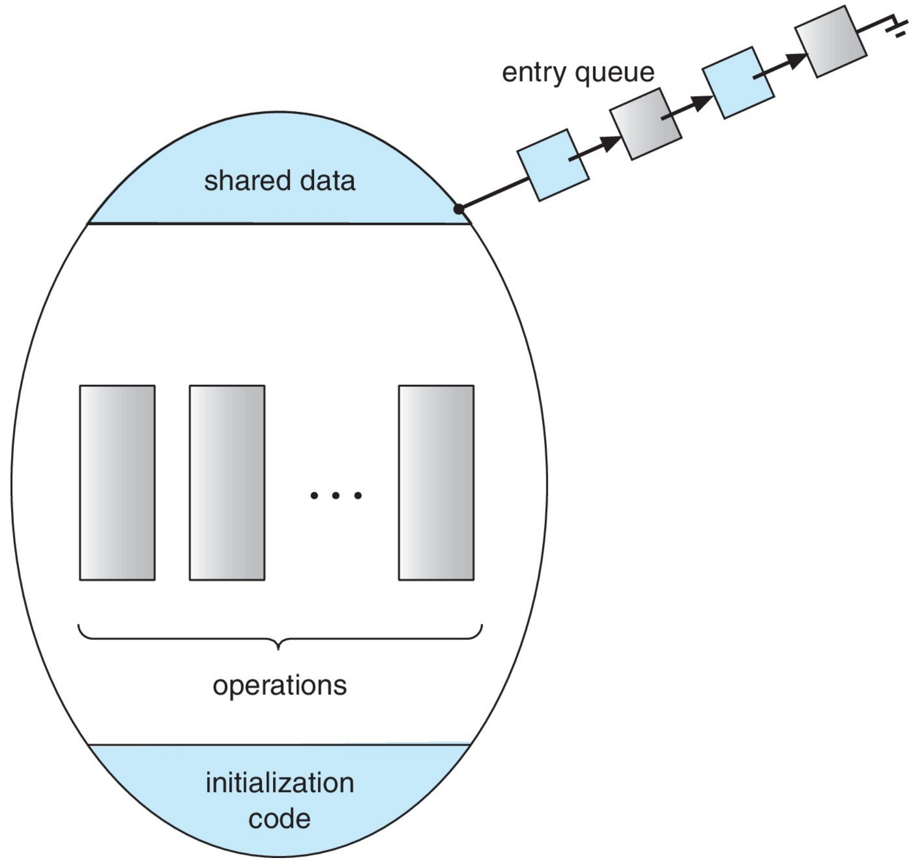
```
monitor monitor-name
{
  // shared variable declarations
  function P1 (…) { …. }

  function P2 (…) { …. }

  function Pn (…) {……}

  initialization code (…) { … }
}
```

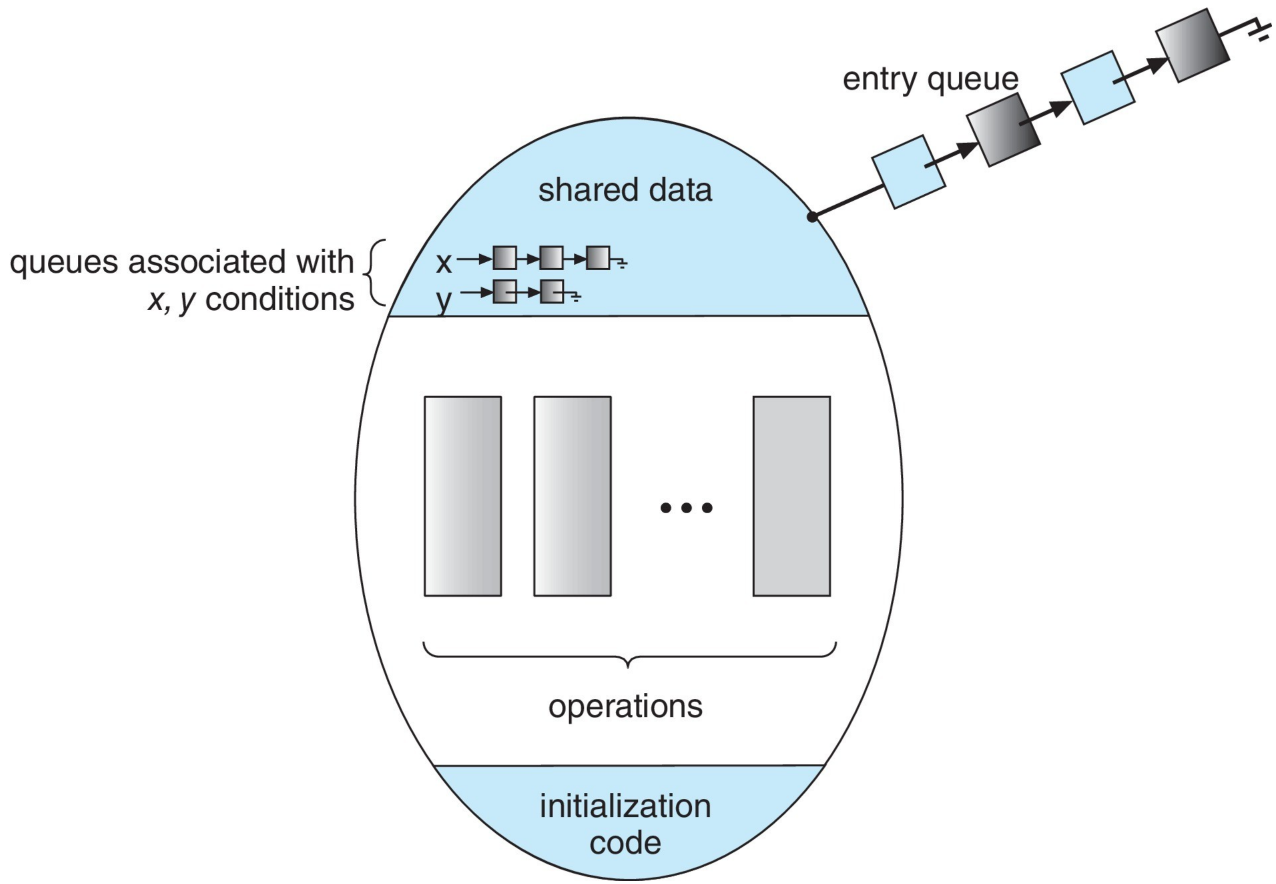
Schematic view of a Monitor



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



Condition Variables Choices

- If process P invokes **`x.signal()`**, and process Q is suspended in **`x.wait()`**, what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it 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

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0; // number of processes waiting

                      inside the monitor
```

- Each function ***F*** will be replaced by

```
wait(mutex);
    …
    body of F;
    …
if (next_count > 0)
      signal(next)
else
      signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Implementation – Condition Variables

- For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

- The operation **`x.wait()`** can be implemented as:

```
x_count++;
if (next_count > 0)
        signal(next);
else
       signal(mutex);
wait(x_sem);
x_count--;
```

Implementation (Cont.)

- The operation **`x.signal()`** can be implemented as:

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

Resuming Processes within a Monitor

- If several processes queued on condition variable `x`, and `x.signal()` is executed, which process 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

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 resurce;
    ...

R.release;
```

- Where R is an instance of type **`ResourceAllocator`**

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;
   void acquire(int time) {
          if (busy)
                 x.wait(time);
          busy = true;
   }
   void release() {
          busy = FALSE;
          x.signal();
   }
   initialization code() {
    busy = false;
   }
}
```

Single Resource Monitor (Cont.)

- Usage:

  ```
  acquire
   ...
  release
  ```

- Incorrect use of monitor operations
  - `release()` … `acquire()`
  - `acquire()` … `acquire())`
  - Omitting of `acquire()` and/or `release()`

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.

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

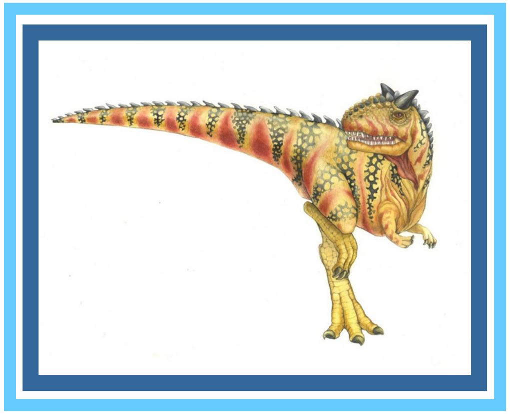
```
        P0                      P1
     wait(S);                wait(Q);
     wait(Q);                wait(S);
   ...                     ...
     signal(S);              signal(Q);
     signal(Q);              signal(S);
```

- Consider if P_0 executes wait(S) and P_1 wait(Q). When P_0 executes wait(Q), it must wait until P_1 executes signal(Q)
- However, P_1 is waiting until P_0 execute signal(S).
- Since these 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



Priority Inheritance Protocol

- Consider the scenario with three processes **P1, P2**, and **P3**. **P1** has the highest priority, **P2** the next highest, and **P3** the lowest.
- Assume a resource **P3** is assigned a resource **R** that **P1** wants.
  - Thus, **P1** must wait for **P3** to finish using the resource. However, **P2** becomes runnable and preempts **P3**.
  - What has happened is that **P2** - a process with a lower priority than **P1** - has indirectly prevented **P3** from gaining access to the resource.
- To prevent this from occurring, a **priority inheritance protocol** is used. This simply allows the priority of the highest thread waiting to access a shared resource to be assigned to the thread currently using the resource. Thus, the current owner of the resource is assigned the priority of the highest priority thread wishing to acquire the resource.

Priority Inheritance Protocol

- Consider the scenario with three processes **P1, P2**, and **P3**. **P1** has the highest priority, **P2** the next highest, and **P3** the lowest.
- Assume a resource **P3** is assigned a resource **R** that **P1** wants. Thus, **P1** must wait for **P3** to finish using the resource. However, **P2** becomes runnable and preempts **P3**. What has happened is that **P2** - a process with a lower priority than **P1** - has indirectly prevented **P3** from gaining access to the resource.
- To prevent this from occurring, a **priority inheritance protocol** is used. This simply allows the priority of the highest thread waiting to access a shared resource to be assigned to the thread currently using the resource. Thus, the current owner of the resource is assigned the priority of the highest priority thread wishing to acquire the resource.