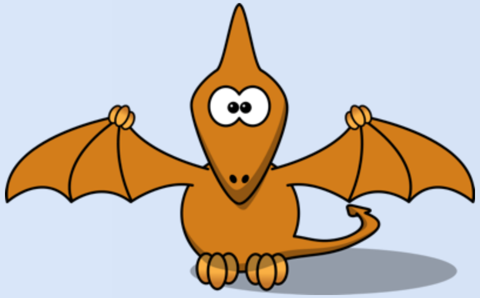
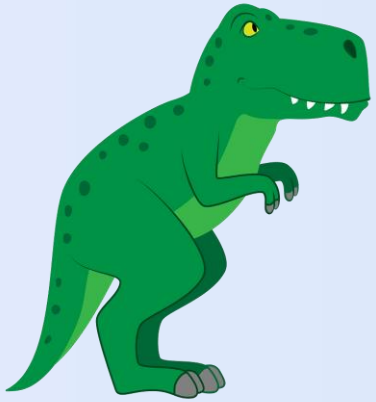


Chapter 6.

Synchronization Tools (2)



Operating System Concepts (10th Ed.)

6.5 Mutex Locks

- Higher-level software tools to solve the CSP:
  - *Mutex Locks*: the simplest tools for synchronization.
  - *Semaphore*: more robust, convenient, and effective tool.
  - *Monitor*: overcomes the demerits of mutex and semaphore.
  - *Liveness*: ensures for processes to make progress.

6.5 Mutex Locks

- Mutex Lock
  - ***mutex***: ***mut***ual ***ex***clusion.
  - to protect critical section and prevent race condition.
  - a process must *acquire* the ***lock*** before *entering* a critical section.
  - *releases* the ***lock*** when it *exits* the critical section.

6.5 Mutex Locks

- Two functions and one variable for the Mutex Locks:
  - `acquire()` and `release()`
  - `available:` a Boolean variable whose value indicates
    - if the lock is available or not.

```
while (true) {

    acquire lock

        critical section

    release lock

        remainder section

}
```

Figure 6.10 *Solution to the critical-section problem using mutex locks.*

- The definition of acquire() and release():

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

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

- Calls to either acquire() and release() must be performed *atomically*.
- can be implemented using the *compare_and_swap* operation.

6.5 Mutex Locks

- **Busy waiting**:
  - Any other process trying to enter its critical section
    - must *loop continuously* in the call to acquire().
  - Busy waiting is clearly a *problem* in a real multiprogramming system,
    - where a single CPU core is shared among many processes.
    - *wastes CPU cycles* for some other processes to use productively.

6.5 Mutex Locks

- **Spinlock**:
  - the type of mutex lock using the method of *busy waiting*.
  - the process *spins* while waiting for the lock to become available.
  - However, spinlocks do have an *advantage*,
    - in that *no context switch* is required waiting on a lock.
    - a context switch may take considerable time.
  - In certain circumstances *on multicore systems*,
    - spinlocks are the *preferable* choice for locking.
    - One thread can *spin on one processing core*
    - while another thread performs its critical section *on another core*

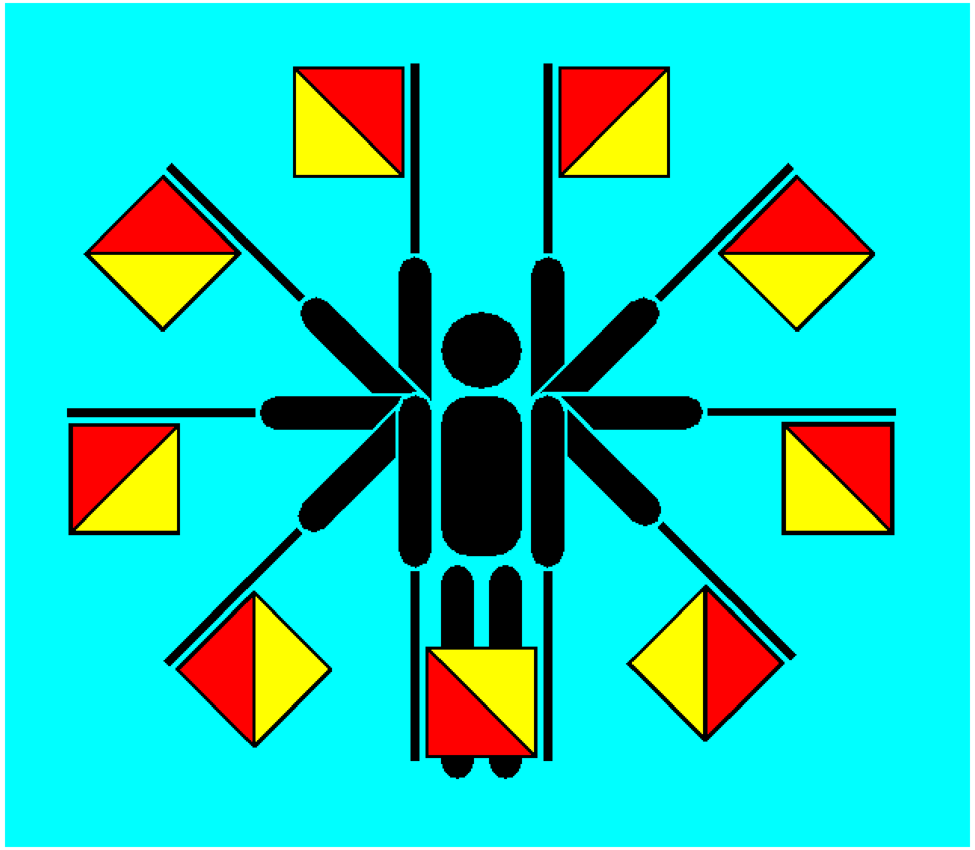
6.5 Mutex Locks

```c
void *counter(void *param)
{
    int k;
    for (k = 0; k < 10000; k++) {
        /* entry section */
        pthread_mutex_lock(&mutex);

        /* critical section */
        sum++;

        /* exit section */
        pthread_mutex_unlock(&mutex);

        /* remainder section */
    }
    pthread_exit(0);
}
```

```c
#include <stdio.h>
#include <pthread.h>

int sum = 0;  // a shared variable

pthread_mutex_t mutex;


int main()
{
    pthread_t tid1, tid2;
    pthread_mutex_init(&mutex, NULL);
    pthread_create(&tid1, NULL, counter, NULL);
    pthread_create(&tid2, NULL, counter, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("sum = %d\n", sum);
}
```

6.6 Semaphores

- Semaphore
  - semaphore: 신호장치. 신호기.



- Defining the Semaphore
  - A ***semaphore*** *S* is
    - an integer variable that, apart from initialization,
    - is accessed only through *two standard atomic operations*:
    - `wait()` and `signal()`, or sometimes `P()` and `V()`.

  - P() and V() are introduced by Edsger Dijkstra
    - Proberen(to test) and Verhogen(to increment)

6.6 Semaphores

- Definition of wait() and signal():

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

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

- All *modifications* to the integer value of the semaphore
  - in the wait() and signal() operations must be executed *atomically*.

6.6 Semaphores

- Binary and Counting Semaphores
  - *Binary* Semaphore
    - range only between 0 and 1: similar to *mutex lock*.
  - *Counting* Semaphore
    - range over an unrestricted domain.
    - can be used to resources with *a finite number of instances*.

6.6 Semaphores

- Using the counting semaphore:
  - Initialize a semaphore to *the number of resources available*.
  - When a process uses a resource
    - wait() on the semaphore: *decrements the count*.
  - When a process release a resource
    - signal() on the semaphore: *increment the count*.
  - When the count goes to 0, all resources are being used.
    - Then, processes that wish to use a resource *will block*
    - *until* the count becomes greater than 0.

6.6 Semaphores

- Using the semaphore to solve synchronization problem:
  - Consider two processes P_1 and P_2 running concurrently.
    - P_1 with a statement S_1, and P_1 with a statement S_2.
  - Suppose that S_2 should be *executed only after* S_1 has completed.
    - Let P_1 and P_2 share a *semaphore* **synch**, initialized to 0.

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

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

6.6 Semaphores

- Semaphore Implementation:
  - Semaphores also suffer from the problem of *busy waiting*.
  - To overcome this problem, modify the definition of P() and V().
  - When a process executes the **wait()** operation
    - and finds that the semaphore *is not positive*, it must *wait*.
    - rather than busy waiting, suspend itself and goes to the *waiting queue*.
  - When other process executes the **signal()** operation
    - waiting processes can be *restarted* and placed into the *ready queue*.

6.6 Semaphores

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

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

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

6.6 Semaphores

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

int sum = 0;  // a shared variable

sem_t sem;


int main()
{
    pthread_t tid1, tid2;
    sem_init(&sem, 0, 1);
    pthread_create(&tid1, NULL, counter, NULL);
    pthread_create(&tid2, NULL, counter, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("sum = %d\n", sum);
}
```

```c
void *counter(void *param)
{
    int k;
    for (k = 0; k < 10000; k++) {
        /* entry section */
        sem_wait(&sem);

        /* critical section */
        sum++;

        /* exit section */
        sem_post(&sem);

        /* remainder section */
    }
    pthread_exit(0);
}
```

6.6 Semaphores

```c
int main()
{
    pthread_t tid[5]; int i;
    sem_init(&sem, 0, 5);
    for (i = 0; i < 5; i++)
        pthread_create(&tid[i], NULL, counter, NULL);
    for (i = 0; i < 5; i++)
        pthread_join(tid[i], NULL);
    printf("sum = %d\n", sum);
}
```

6.7 Monitors

- The *difficulty* of using semaphores:
  - The semaphore is *convenient* and *effective* for synchronization.
  - However, *timing errors* can happen
    - if particular execution sequences take place.
    - these sequences *do not always occur*,
    - and it is *hard to detect*.

6.7 Monitors

- An illustrative example of semaphore's problem
  - All processes share a *binary semaphore* **mutex** initialized to 1.
    - Each process must **wait(mutex)** before entering the CS
    - and **signal(mutex)** afterward.
  - If this sequence is not observed,
    - two processes *may be in* their critical sections *simultaneously*.

6.7 Monitors

- Situation 1:
  - Note that the difficulty arises
    - even if a single process is not well behaved.
  - Suppose that a program *interchanges the order*.
    - in which **wait()** and **signal()** on the semaphore **mutex** are executed.

```
signal(mutex);
    ...
  critical section
    ...
wait(mutex);
```

6.7 Monitors

- Situation 2 & 3:
  - Suppose that a program *replaces* **signal()** *with* **wait()**.

```
wait(mutex);
    ...
  critical section
    ...
wait(mutex);
```

  - Suppose that a process omits the **wait()**, or the **signal()**, or both of them.

6.7 Monitors

- How to deal with these kinds of difficulties?
  - These situations may be caused
    - by an honest programming error or an uncooperating programmer.
  - Various types of errors can be generated easily
    - when programmers use semaphores (or mutex locks) incorrectly.
  - Incorporate simple synchronization tools
    - as high-level language constructs
    - ***monitor***: one fundamental high-level synchronization construct.

6.7 Monitors

- A ***monitor type*** is
  - an **ADT** that includes a set of *programmer-defined operations*
    - that are provided with mutual exclusion within the ***monitor***.
  - declares the ***variables***
    - whose values define the *state of an instance* of that type.
    - along with the bodies of ***function*** that operate on those *variables*.

6.7 Monitors

```
monitor monitor name
{
  /* shared variable declarations */

  function P1 ( . . . ) {
     . . .
  }

  function P2 ( . . . ) {
     . . .
  }

           .
           .
           .

  function Pn ( . . . ) {
     . . .
  }

  initialization_code ( . . . ) {
     . . .
  }
}
```

Figure 6.11 *Pseudocode syntax of a monitor.*

6.7 Monitors

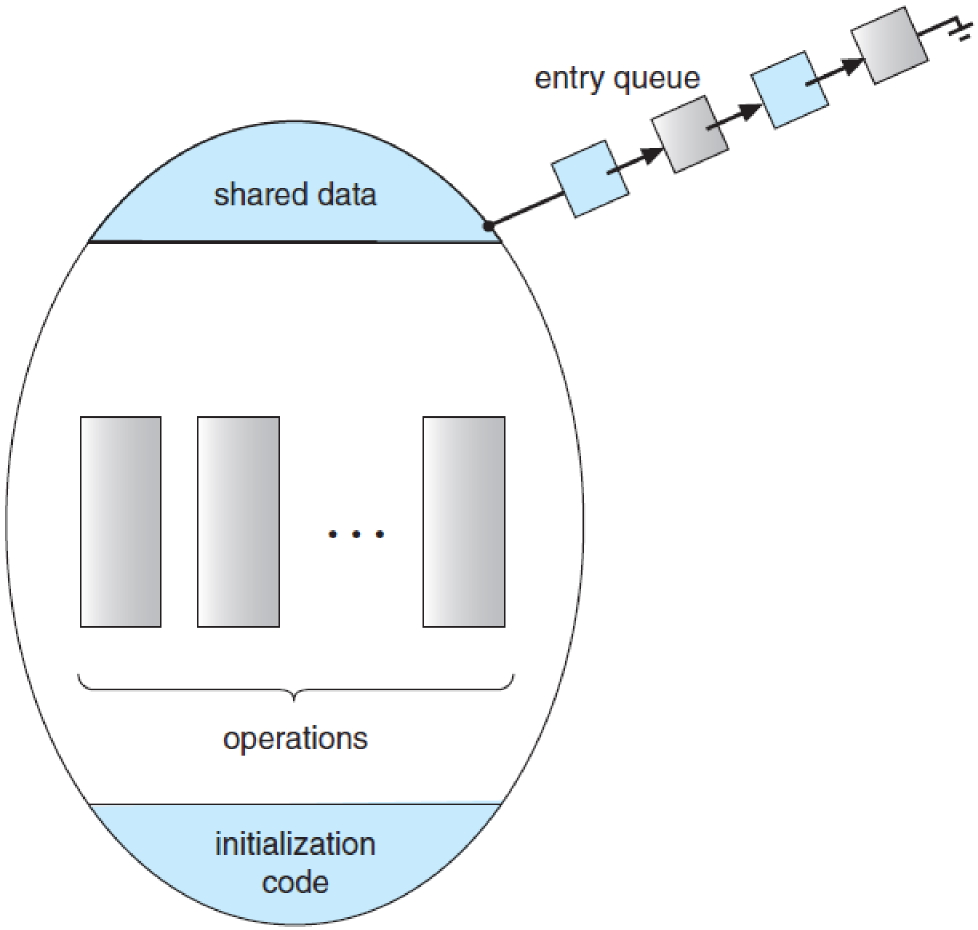


Figure 6.12 *Schematic view of a monitor.*

6.7 Monitors

- **Conditional Variables**:
  - The monitor construct is not sufficiently powerful
    - for modeling some synchronization schemes.
  - We need to define the **condition** construct
    - to provide additional synchronization mechanisms.

6.7 Monitors

- Using conditional variables:
  - One can define one or more variables of type **condition**:
  - The only operations that can be invoked
    - on a ***condition variable*** are **wait()** and **signal()**.

```
condition x, y;

x.wait();

x.signal();
```

6.7 Monitors

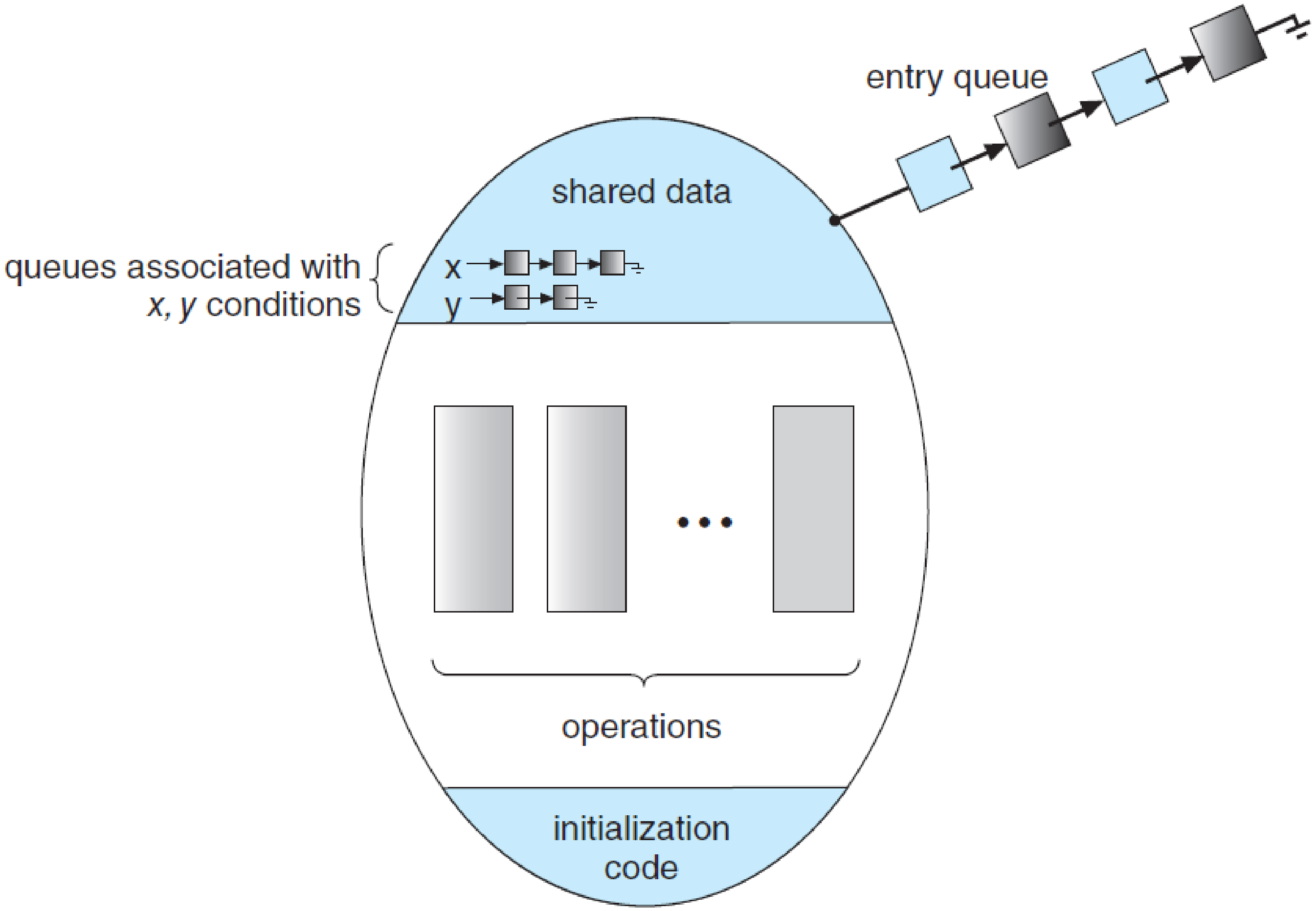


Figure 6.13 *Monitor with condition variables.*

6.7 Monitors

- Java Monitors
  - Java provides a ***monitor-like***
    - *concurrency* mechanism for *thread synchronization*.
    - called as ***monitor-lock*** or ***intrinsic-lock***.
  - Basic language constructs for Java Synchronization
    - **synchronized** keyword.
    - **wait()** and **notify()** method.

6.7 Monitors

- **synchronized** keyword:
  - 임계영역에 해당하는 코드 블록을 선언할 때 사용하는 자바 키워드
  - 해당 코드 블록(임계영역)에는 모니터락을 획득해야 진입 가능
  - 모니터락을 가진 객체 인스턴스를 지정할 수 있음
  - 메소드에 선언하면 메소드 코드 블록 전체가 임계영역으로 지정됨
    - 이 때, 모니터락을 가진 객체 인스턴스는 this 객체 인스턴스임

```
synchronized (object) {
    // critical section
}
```

```
public synchronized void add() {
        // critical section
}
```

6.7 Monitors

- **wait()** and **notify()** methods:
  - `java.lang.Object` 클래스에 선언됨: 모든 자바 객체가 가진 메소드임
  - 쓰레드가 어떤 객체의 wait() 메소드를 호출하면
    - 해당 객체의 모니터락을 획득하기 위해 대기 상태로 진입함.
  - 쓰레드가 어떤 객체의 notify() 메소드를 호출하면
    - 해당 객체 모니터에 대기중인 쓰레드 하나를 깨움.
  - notify() 대신에 notityAll() 메소드를 호출하면
    - 해당 객체 모니터에 대기중인 쓰레드 전부를 깨움.

- Java Synchronization Example 1:

```
public class SynchExample1 {

    static class Counter {
        public static int count = 0;
        public static void increment() {
            count++;
        }
    }

    static class MyRunnable implements Runnable {
        @Override
        public void run() {
            for (int i = 0; i < 10000; i++)
                Counter.increment();
        }
    }
```

6.7 Monitors

```
    public static void main(String[] args) throws Exception {
        Thread[] threads = new Thread[5];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(new MyRunnable());
            threads[i].start();
        }
        for (int i = 0; i < threads.length; i++)
            threads[i].join();
        System.out.println("counter = " + Counter.count);
    }

}
```

- Java Synchronization Example 2:

```
public class SynchExample2 {

    static class Counter {
        public static int count = 0;
        synchronized public static void increment() {
            count++;
        }
    }

    static class MyRunnable implements Runnable {
        @Override
        public void run() {
            for (int i = 0; i < 10000; i++)
                Counter.increment();
        }
    }
```

6.7 Monitors

- Java Synchronization Example 3:

```
public class SynchExample3 {

    static class Counter {
        private static Object object = new Object();
        public static int count = 0;
        public static void increment() {
            synchronized (object) {
                count++;
            }
        }
    }

    static class MyRunnable implements Runnable {
        @Override
        public void run() {
            for (int i = 0; i < 10000; i++)
                Counter.increment();
        }
    }
```

6.7 Monitors

- Java Synchronization Example 4:

```
public class SynchExample4 {

    static class Counter {
        public static int count = 0;
        public void increment() {
            synchronized (this) {
                Counter.count++;
            }
        }
    }

    static class MyRunnable implements Runnable {
        Counter counter;
        public MyRunnable(Counter counter) {
            this.counter = counter;
        }
        @Override
        public void run() {
            for (int i = 0; i < 10000; i++)
                counter.increment();
        }
    }
```

6.7 Monitors

```
    public static void main(String[] args) throws Exception {
        Thread[] threads = new Thread[5];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(new MyRunnable(new Counter()));
            threads[i].start();
        }
        for (int i = 0; i < threads.length; i++)
            threads[i].join();
        System.out.println("counter = " + Counter.count);
    }

}
```

6.7 Monitors

- Java Synchronization Example 5:

```
public static void main(String[] args) throws Exception {
        Thread[] threads = new Thread[5];
        Counter counter = new Counter();
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(new MyRunnable(counter));
            threads[i].start();
        }
        for (int i = 0; i < threads.length; i++)
            threads[i].join();
        System.out.println("counter = " + Counter.count);
    }
```

6.8 Liveness

- **Liveness**
  - Two criteria for the CSP: the progress and bounded-waiting.
    - Semaphores and monitors cannot solve these requirements.
  - ***Liveness*** refers to
    - a set of properties that a system must satisfy
    - to ensure that processes make progress during their execution cycle.
  - Two situations that can lead to liveness failures.
    - *deadlock* and *priority inversion*.

6.8 Liveness

- ***Deadlock***
  - a situation where two or more processes are *waiting indefinitely*
    - for an event that *can be caused only by* one of the *waiting process*.

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
.	.
.	.
.	.
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

6.8 Liveness

- ***Priority Inversion***
  - A situation where a higher-priority processes have to wait
    - for a lower-priority one to finish the resource.
  - It can arise when a *higher*-priority process
    - needs to *read or modify kernel data*
    - that are currently being accessed by a *lower*-priority process.
  - Typically, priority inversion is avoided
    - by implementing a ***priority-inheritance*** protocol.
  - All processes accessing resources needed by a higher-priority process
    - inherit the higher priority
    - until they releases that resources.

Any Questions?