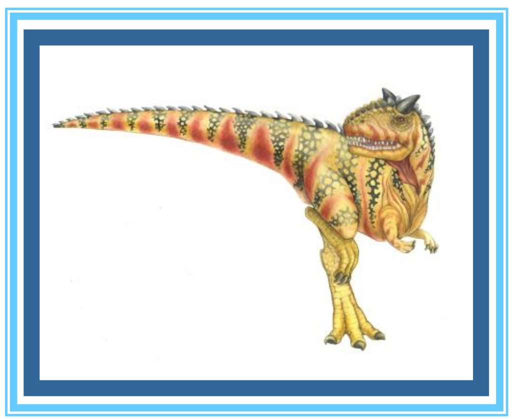
Announcements

- Midterm exam will be on 25 Sep Friday, 6:45pm (night exam)
- Study guide posted
- Exam Review 23-Sep
- Recitation will discuss homework solutions
- Project1 due Sep 20th, list of commands posted

Chapter 6: Synchronization Tools

Section 6.1-6.7



Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution before another process is scheduled
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of **cooperating processes**
  - A **cooperating process** shares data with other processes, can affect or be affected by other processes
- Illustration of the problem:
  - A producer process & a consumer process share a buffer of certain size
  - A shared integer variable **`counter`** is used to keep track of the number of items in buffer
  - **`counter`** incremented by producer after it produces an item, decremented by consumer after it consumes an item

Producer-Consumer Problem

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
}item;
item buffer[BUFFER_SIZE]; //shared variable
int counter = 0; //shared variable
```

```
int in = 0;
while (true) {
    /* produce an item in
       next_produced */
    while (counter == BUFFER_SIZE);
        /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Producer code

```
int out = 0;
while (true) {
    while (counter == 0);
      /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    counter--;
    /* consume the item in
       next_consumed */
}
```

consumer code

Race Condition

- **`counter++`** could be implemented in machine language as

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

- **`counter--`** could be implemented in machine language as

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

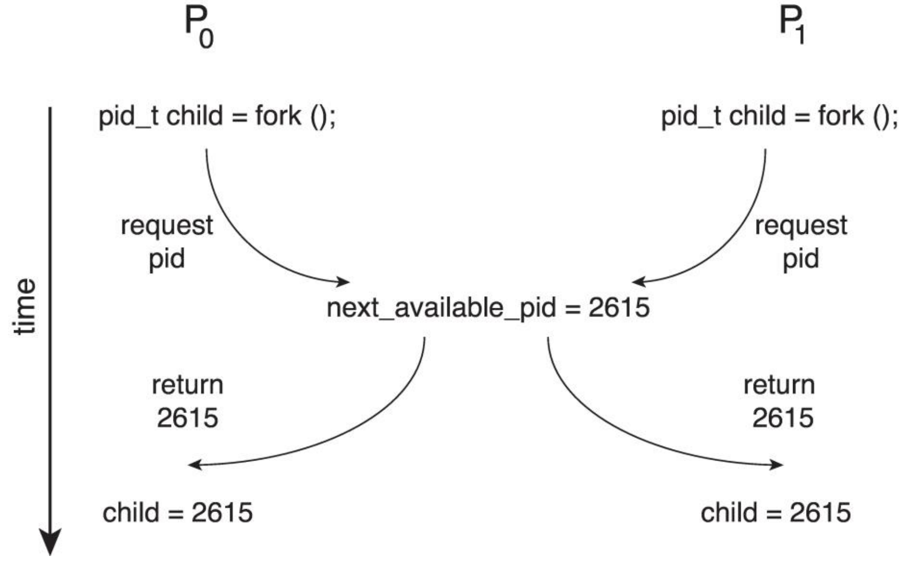
- Consider this execution interleaving with "counter = 5" initially:

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

- A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place is called a **race condition**

Race Condition in Kernel Code

- 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)



- The same pid could be assigned to two different processes!

How to avoid race condition?

- Disable interrupts when shared data is modified?
- Preemptive vs non-preemptive kernel
- Use mode vs Kernel mode
- Single processor vs Multiprocessors

Critical-Section Problem

- Consider a system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has a segment of code, called a **critical section (CS)**, in which the process may be accessing and updating shared data
  - When one process is executing in its CS, no other process is allowed to execute in its CS
- ***Critical-Section Problem*** is to design a protocol to solve this
  - Each process must ask permission to enter critical section in **entry section**
  - The critical section may be followed by an **exit section**, then **remainder section**

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

General structure of a typical process

Solution to Critical-Section Problem

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual Exclusion** - If process P_i is executing in its CS, then no other processes can be executing in their CSs
2. **Progress** - If no process is executing in its CS and one or more processes wish to enter their CSs, then one of these processes must be allowed to enter its CS
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their CSs after a process has made a request to enter its CS and before that request is granted

Peterson's Solution

- A software-based solution to the critical-section problem
- A system of two processes P_i and P_j, which 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 $\mathbf{P_i}$ 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 */

}
```

Peterson's Solution (Cont.)

- Provable that the three CS requirements are met:
  1. Mutual exclusion requirement is satisfied

     P_i enters CS only if:

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

  2. Progress requirement is satisfied
     - If only one process wants to enter CS, it can enter
     - If both processes want to enter CS, the variable **`turn`** determines who can enter next

  3. Bounded waiting requirement is satisfied
     - A process will enter CS after at most one entry by the other process

Peterson's Solution (Cont.)

- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern computer architectures
- To improve performance, processors may reorder instructions that have no dependencies
- For single-threaded application this is OK as the result will always be the same
- For multithreaded application the reordering may produce inconsistent or unexpected results!

Peterson's Solution (Cont.)

- 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?

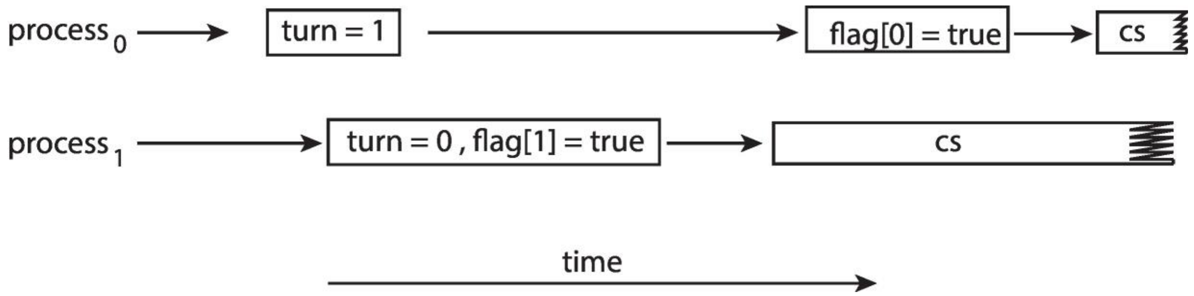
Peterson's Solution (Cont.)

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

Thread 1 performs / Thread 2 performs

```
while (!flag)        flag = true;
     ;               x = 100;
print x;
```

- 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 Instructions

- Many computer systems provide special **atomic** hardware instructions (**atomic** = non-interruptible) that can be used to solve the critical-section problem
  - **test_and_set** instruction: *test-and-modify* the content of a word atomically
  - **compare_and_swap** instruction: *swap* the contents of two words atomically

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`**
- Implementing mutual exclusion:

```
do {
        while (test_and_set(&lock))
         ; /* do nothing */

             /* critical section */

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

Structure of process P_i

The first process that invokes test_and_set will set the lock to true, and then enter CS

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** to **new_value** only if ***value == expected** is true. That is, the swap takes place only under this condition.

Solution using compare_and_swap

- Shared integer variable **lock** initialized to 0;
- Implementing mutual exclusion:

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

    /* critical section */

    lock = 0;

    /* remainder section */
}
```

Structure of process P_i

The first process that invokes compare_and_swap will set the lock to 1, and then enter CS

Mutex Locks

- Hardware-based solutions to critical section problem are complicated and generally inaccessible to application programmers
- OS designers build higher-level software tools to solve critical section problem
- Simplest tool is **mutex lock**
  - Protect a critical section by first **`acquire()`** a lock and then **`release()`** the lock

```
while (true) {
        acquire lock

        critical section

        release lock

        remainder section
}
```

Mutex Lock Definitions

- A mutex lock has a boolean variable **`available`** indicating if lock is available or not
  - **`acquire()`** succeeds if lock is available
  - If lock is not available, **`acquire()`** blocks until lock is released

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

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

- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions such as compare-and-swap
- This solution requires **busy waiting**; this lock therefore called a **spinlock**

Spinlocks

- Spinlocks waste CPU cycles
  - Not appropriate for single-processor systems
  - Can avoid busy waiting by putting the waiting process to sleep and then awakening it once the lock becomes available
- Spinlocks have the advantage of not requiring context switch
  - On a multicore system, one thread can spin on one processing core while another thread performs its critical section on another core
  - On multicore systems, spinlocks are the preferable choice for locking if a lock is to be held for a short duration (i.e., less than 2 context switches)

Semaphore

- A synchronization tool that provides more sophisticated ways (than mutex locks) for process to synchronize their activities
- A semaphore ***S*** is an integer variable that can only be accessed via two atomic (i.e., non-interruptible) operations
  - **`wait()`** and **`signal()`**
- Definition of the **`wait() operation`**

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

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

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

Semaphore Usage

- Two types of semaphores
  - **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**
- Semaphores can be used to solve various synchronization problems
- Example 1: a counting semaphore S can be used to control access to a given resource consisting of N instances
  - S is initialized to N

```
Structure of a process:
     wait(S);
     use resource
     signal(S);
```

Semaphore Usage

- Example 2: Consider ***P_1*** and ***P_2*** that require statement ***S_1*** to happen before statement ***S_2***

Create a semaphore "**`synch`**" initialized to 0

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

Semaphore Usage

- Example 3: use semaphore for mutual exclusion

Create a semaphore **sem_CS** initialized to 1.

```
Structure of a process:
   do {
       wait(sem_CS);
       Critical section
       signal(sem_CS);
       Remainder section
   } while (true);
```

Semaphore Implementation

- The wait operation suffers from busy waiting

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

- To eliminate busy waiting, we associate each semaphore with a waiting queue

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

  - When a process executes **`wait()`** and finds that the semaphore value is not positive, the process is added to the waiting queue
  - A **`signal()`** operation removes one process from the waiting queue and awakens that process

Semaphore Implementation

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

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

Can use a FIFO queue to implement the list of waiting processes to ensure bounded waiting

If semaphore value is negative, its magnitude = number of processes waiting on the semaphore

Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- The implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in critical section
  - Can use test_and_set or compare_and_swap to implement critical section
  - OK to have **busy waiting** in critical section implementation because critical section is short

Problems with Semaphores

- Incorrect use of semaphore operations:
  - **`signal (mutex) Critical Section wait (mutex)`**

    Mutual exclusion requirement is violated!
  - **`wait (mutex) Critical Section wait (mutex)`**

    Process will permanently block on the second call to **`wait()`**
  - Omitting **`wait (mutex)`** or **`signal (mutex)`**

    Either mutual exclusion is violated or other processes can never enter critical section
- These – and others – are examples of what can occur when semaphores are used incorrectly

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- A **monitor type** is an abstract data type that encapsulates data with a set of functions to operate on that data
  - The set of operations are provided with mutual exclusion; only one process may be active within the monitor at a time
  - Provided in C#, Java
- Pseudocode syntax of a monitor:

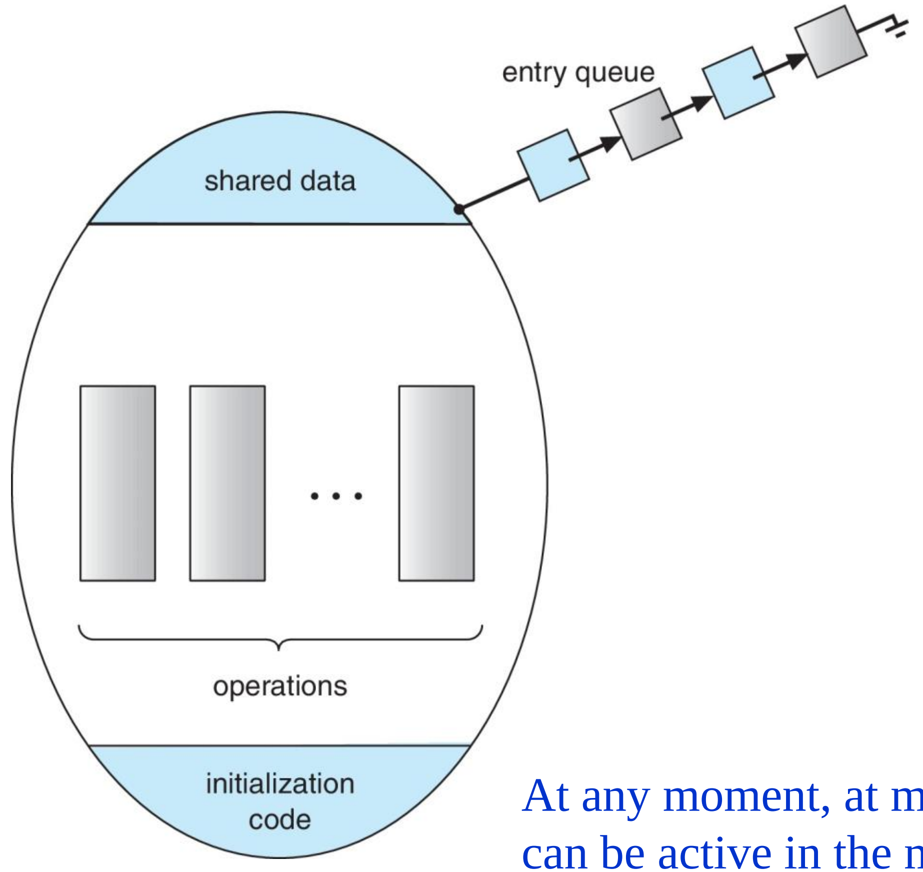
```
monitor monitor-name
{
  // shared variable declarations
  function P1 (…) { …. }

  function P2 (…) { …. }

  function Pn (…) {……}

  initialization code (…) { … }
}
```

Schematic view of a Monitor



At any moment, at most one process can be active in the monitor

A Monitor Example

```
monitor SharedAccount
{
    int balance;

    void credit (int amount) {
        balance = balance + amount
    }

    void debit (int amount) {
        balance = balance - amount;
    }

    initialization_code() {
        balance = 100;
    }
}
```

SharedAccount.credit(100) **P1**

SharedAccount.debit(50) **P2**

Note: Only one process in monitor at a time

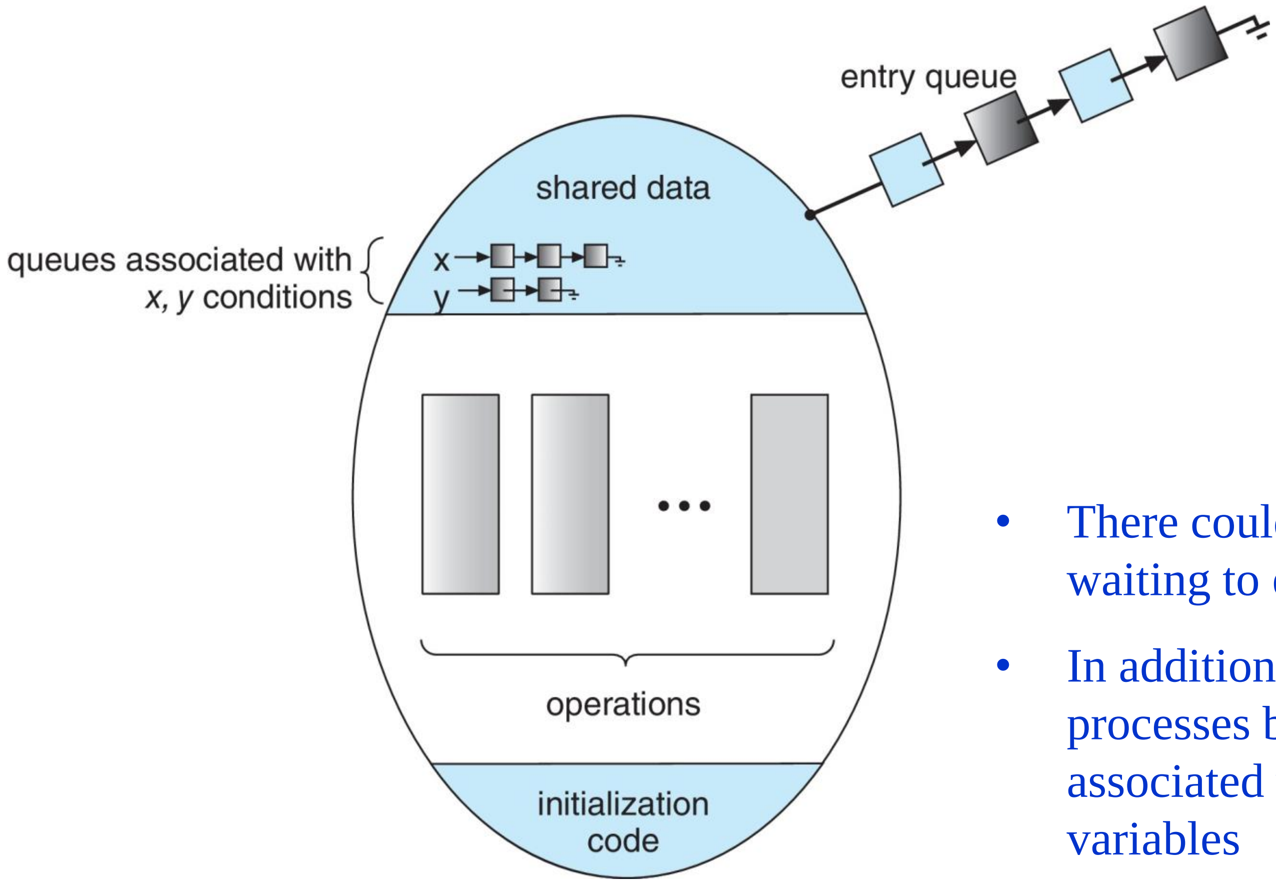
Condition Variables

- We can use condition variables in monitors to implement more sophisticated synchronization
- Define a condition variable:

  `condition x, y;`

- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended
  - **`x.signal()`** – resumes one of the processes (if any) that invoked **`x.wait()`**
    - If no process is suspended on **`x`**, then it has no effect
    - In case of semaphore, the state is changed on signal()

Monitor with Condition Variables



- There could be processes waiting to enter a monitor
- In addition, there could be processes blocked in queues associated with condition variables

Condition Variable Example

```
monitor SharedAccount
{
    int balance;
    condition c;

    void credit (int amount) {
            balance = balance + amount;
            c.signal();
    }

    void debit (int amount) {
            while (balance – amount < 0)
                  c.wait();
            balance = balance - amount;
    }

    initialization_code() {
            balance = 100;
    }
}
```

SharedAccount.debit(250) **P1**

SharedAccount.credit(100) **P2**

SharedAccount.credit(50) **P3**

Condition Variables Choices

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

Deadlocks

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused only by 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);`

- Suppose P_0 executes wait(S) and P_1 executes wait(Q).
- When P_0 executes wait(Q), it must wait until P_1 executes signal(Q).
- Similarly, When P_1 executes wait(S), it must wait until P_0 executes signal(S).
- Since signal(S) and signal(Q) can never be executed, P_0 and P_1 are **deadlocked**.