CS343 Operating Systems

Lecture 9

Process Synchronization

John Jose

Associate Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati

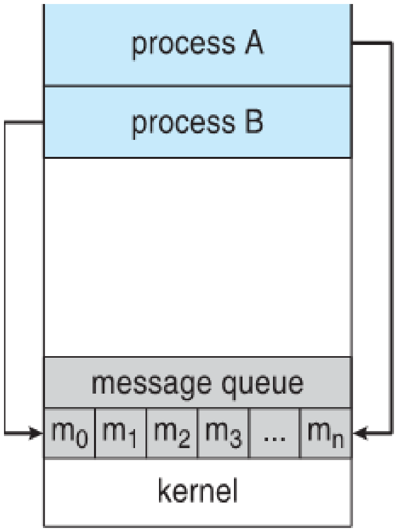
Process Management

- Creating and deleting both user and system processes
- Suspending and resuming processes (context switching, scheduling)
- Providing mechanisms for process communication
- Providing mechanisms for process synchronization
- Providing mechanisms for deadlock handling

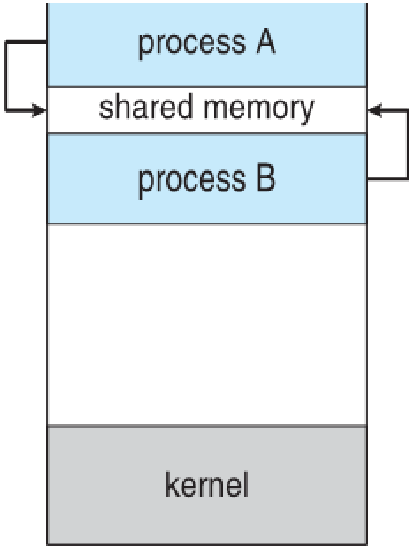
Communications Models

- Cooperating processes need **Interprocess communication** (**IPC**)
- Two models of IPC:

Message passing



Shared memory



Bounded-Buffer – Producer & Consumer

```
item buffer[BUFFER_SIZE]; int in = 0; int out = 0;
```

Producer

```
item next_produced;

while (true)

 {      /* produce an item in next
        produced */

    while(((in + 1)% BUFFER_SIZE)
    == out) ; /* do nothing */

    buffer[in] = next_produced;

    in = (in + 1) % BUFFER_SIZE;

}
```

Consumer

```
item next_consumed;

while (true)

 {
  while (in == out); /*do nothing */


    next_consumed = buffer[out];

    out = (out + 1) % BUFFER_SIZE;
    /* consume the item in next
consumed */

 }
```

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.

Bounded-Buffer – Producer & Consumer

```
item buffer[BUFFER_SIZE]; int in = 0; int out = 0;
```

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

- Consider this execution interleaving with count = 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}

Critical Section Problem

- Consider system of **n** processes {$\mathbf{p_0}$, $\mathbf{p_1}$, **…** $\mathbf{p_{n-1}}$}
- 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

Critical Section

- 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**

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

```
do {
 while (turn == j);
    critical section
 turn = j;
    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

Peterson's Solution

- Applicable for 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 $\mathbf{P_i}$ is ready!

Peterson's Solution

Algorithm for Process P_i

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j]&&turn==j);

    critical section

    flag[i] = false;

    remainder section

 } while (true);
```

Algorithm for Process P_j

```
do {
    flag[j] = true;
    turn = i;
    while (flag[i]&&turn==i);

    critical section

    flag[j] = false;

    remainder section

 } while (true);
```

Peterson's Solution

- All 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

Algorithm for Process P_i

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j]&&turn==j);

    critical section

    flag[i] = false;

    remainder section

  } while (true);
```

Synchronization Hardware - Locks

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
- Modern machines provide special atomic hardware instructions
  - **Atomic** = non-interruptible
  - Test memory word and set value

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

Mutex Locks

- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **acquire()** a lock then **release()** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **acquire()** and **release()** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Synchronization Using acquire() and release()

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

```
do
 {
    acquire lock
       critical section
    release lock
      remainder section
 } while (true);
```

Critical Section

- 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**

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

```
do {
 while (turn == j);
    critical section
 turn = j;
    remainder section
 } while (true);
```

Mutual Exclusion :: Progress :: Bounded Waiting

Semaphore

- Synchronization tool for processes to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations

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

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

Semaphore Usage

- **Binary semaphore** – value can range only between 0 and 1
  - Represents single access to a resource
- **Counting semaphore** – integer value (unrestricted range)
  - Represents a resource with N concurrent access
- Consider P_1 and P_2 that require S_1 to happen before S_2
  - Create a semaphore "**synch**" initialized to 0

```
P1:
   S1;
   signal(synch);
```

```
P2:
      wait(synch);
      S2;
```

Semaphore Implementation

- With each semaphore there is an associated waiting queue
- 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

- Semaphore uses two atomic operations
- Each semaphore has a queue of waiting processes
- When wait() is called by a thread:
  - If semaphore is open, thread continues
  - If semaphore is closed, thread blocks on queue
- When signal() opens the semaphore:
  - If a thread is waiting on the queue, the thread is unblocked
  - If no threads are waiting on the queue, the signal is remembered for the next thread

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

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

Semaphore Implementation

```
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);

    }

}
```

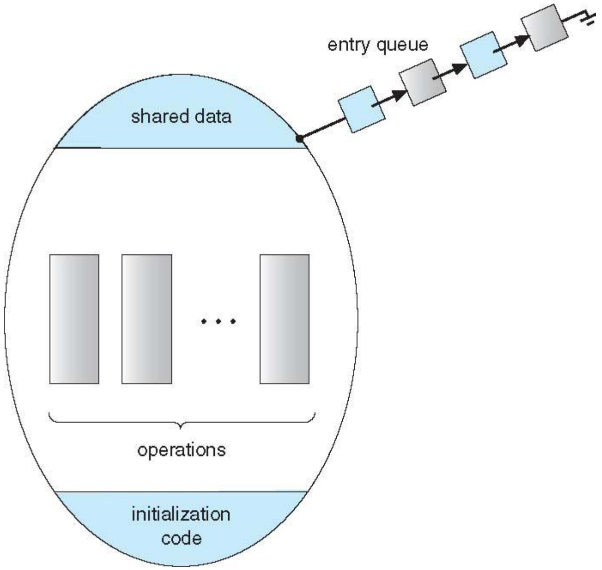
Monitors

- A monitor is a programming language construct that controls access to shared data
- A monitor is a module that encapsulates
  - Shared data structures
  - Procedures that operate on the shared data structures
  - Synchronization between concurrent procedure invocations

Monitors

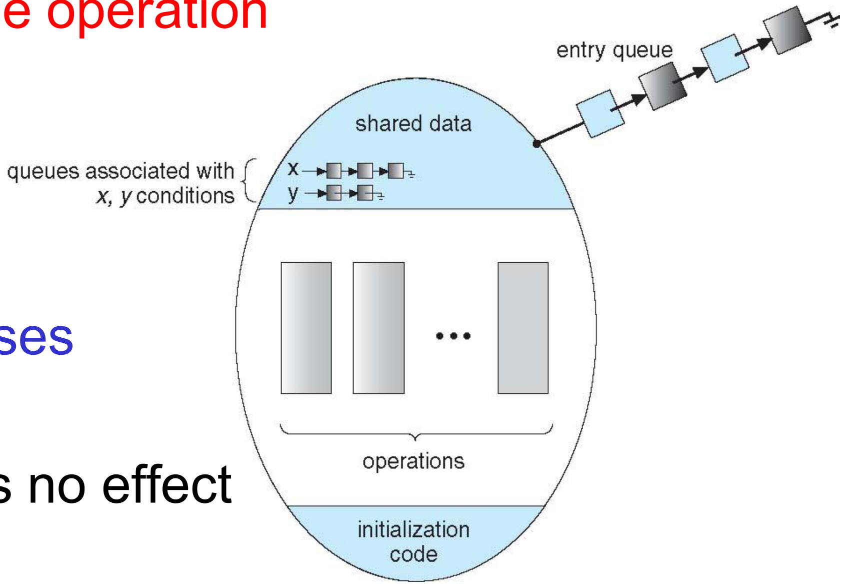
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Abstract data type, internal variables only accessible by code within the procedure
- One process may be active within the monitor at a time

```
monitor monitor-name
{  // shared variable declarations
   procedure P1 (…) { …. }
   procedure Pn (…) {……}
    Initialization code (…) { … }
   }
}
```



Condition Variables

- 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



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

johnjose@iitg.ac.in
http://www.iitg.ac.in/johnjose/