Lecture for IPC (Part-1)

Topics to Be Covered:

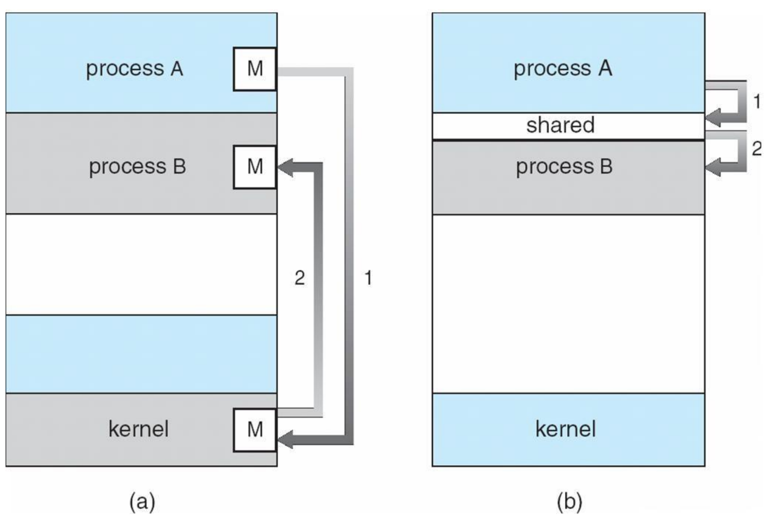
- inter process communication
- Race Condition
- Solution to critical-section Problem
- mutual exclusion

Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data Reasons for cooperating processes:
- Information sharing
- Computation speedup
- Modularity
- Convenience
- Cooperating processes need **interprocess communication** (**IPC**)

Two models of IPC Shared memory and Message passing

Communications Models



Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process

Advantages of process cooperation

- Information sharing
- Computation speed-up
- Modularity
- Convenience

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
- *unbounded-buffer* places no practical limit on the size of the buffer
- *bounded-buffer* assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

Shared data

```
#define BUFFER_SIZE 10
typedef struct {
        . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded-Buffer – Producer

```
        while (true) {
  /* Produce an item */
      while (((in = (in + 1) % BUFFER SIZE count) == out)
            ; /* do nothing -- no free buffers */
           buffer[in] = item;
           in = (in + 1) % BUFFER SIZE;
    }
```

Bounded Buffer – Consumer

```
while (true) {
       while (in == out)
            ; // do nothing -- nothing to consume
            // remove an item from the buffer
            item = buffer[out];
            out = (out + 1) % BUFFER SIZE;
        return item;
    }
```

Interprocess Communication – Message Passing

Mechanism for processes to communicate and to synchronize their actions

- Message system – processes communicate with each other without resorting to shared variables IPC facility provides two operations:
- **send**(*message*) – message size fixed or variable
- **receive**(*message*)

1. If *P* and *Q* wish to communicate, they need to: establish a *communication*

2. *link* between them exchange messages
   via send/receive physical
3. Implementation of communication link
   (e.g., shared memory, hardware bus)
   logical (e.g., logical properties)

Direct Communication

- Processes must name each other explicitly:
- **send** (*P, message*) – send a message to process P
- **receive**(*Q, message*) – receive a message from process
- Q Properties of communication link
-
- Links are established automatically
- A link is associated with exactly one pair of communicating processes
- Between each pair there exists exactly one link
  The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
- Each mailbox has a unique id
- Processes can communicate only if they share a mailbox
- Properties of communication link
- Link established only if processes share a common mailbox
- A link may be associated with many processes
- Each pair of processes may share several communication links
- Link may be unidirectional or bi-directional
- Operations
- create a new mailbox
- send and receive messages through mailbox
- destroy a mailbox
- Primitives are defined as:
- **send**(*A, message*) – send a message to mailbox A
- **receive**(*A, message*) – receive a message from mailbox A
- Mailbox sharing
-
- *P1, P2,* and *P3* share mailbox A
- *P1*, sends; *P2* and *P3* receive
- Who gets the message?
-
- Solutions
-
- Allow a link to be associated with at most two processes
  Allow only one process at a time to execute a receive operation

Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

Message passing may be either blocking or non-blocking

- **Blocking** is considered **synchronous**
- **Blocking send** has the sender block until the message is received
- **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
- **Non-blocking** send has the sender send the message and continue
- **Non-blocking** receive has the receiver receive a valid message or null

CONCURRENCY

Process Synchronization

- 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 introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- 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 count that keeps track of the number of full buffers. Initially, count 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

```
while (true) {

    /* produce an item and put in nextProduced */
        while (count == BUFFER_SIZE)
                ; // do nothing
            buffer [in] = nextProduced;
            in = (in + 1) % BUFFER_SIZE;
            count++;
}
```

Consumer

```
while (true)  {
```

```
        while (count == 0)
                ; // do nothing
                nextConsumed =  buffer[out];
                 out = (out + 1) % BUFFER_SIZE;
              count--;
                   /* consume the item in nextConsumed
    }
```

Race Condition

count++ could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

count-- could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

Consider this execution interleaving with "count = 5" initially:

S0: producer execute register1 = count {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = count {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute count = register1 {count = 6 }
S5: consumer execute count = register2 {count = 4}

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

No assumption concerning relative speed of the N processes

Peterson's Solution

Two process solution

Assume that the LOAD and STORE 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 Pi

```
do {
        flag[i] = TRUE;
        turn = j;
        while (flag[j] && turn == j);
            critical section
        flag[i] = FALSE;
            remainder section
    } while (TRUE);
```

Synchronization Hardware

- Many systems provide hardware support for 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
- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable
- Either test memory word and set value Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

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

TestAndSet Instruction

Definition:

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

Solution using TestAndSet
Shared boolean variable lock., initialized to false.
Solution:

```
            do {
        while ( TestAndSet (&lock ))
                ;   // do nothing
              //    critical section
        lock = FALSE;
              //      remainder section
    } while (TRUE);
```

Swap Instruction
Definition:

```
    void Swap (boolean *a, boolean *b)
    {
        boolean temp = *a;
        *a = *b;
        *b = temp:
    }
```

Solution using Swap
Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key
Solution:

```
    do {
        key = TRUE;
        while ( key == TRUE)
            Swap (&lock, &key );

              //    critical section
        lock = FALSE;


              //      remainder section
    } while (TRUE);
```

Bounded-waiting Mutual Exclusion with TestandSet()

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
            key = TestAndSet(&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);
```