Chapter : Process Synchronization

Process Synchronization

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples

Background

- **Co-Operating Process: that can affect or be affected by other processes executing in system**
- Concurrent access to shared data may result in data inconsistency
- **Process Synchronization: Ensures coordination among processes and maintains Data Consistency**

Process P1	**Process P2**
1. x=5	
2. x=5+2	
	1. read(x);
	2. x=x+5;
3. Printf(x);	

Producer Consumer Problem

There can be two situations:

1. Producer Produces Items at Fastest Rate Than Consumer Consumes

2. Producer Produces Items at Lowest Rate Than Consumer Consumes

Eg. Computer → Producer

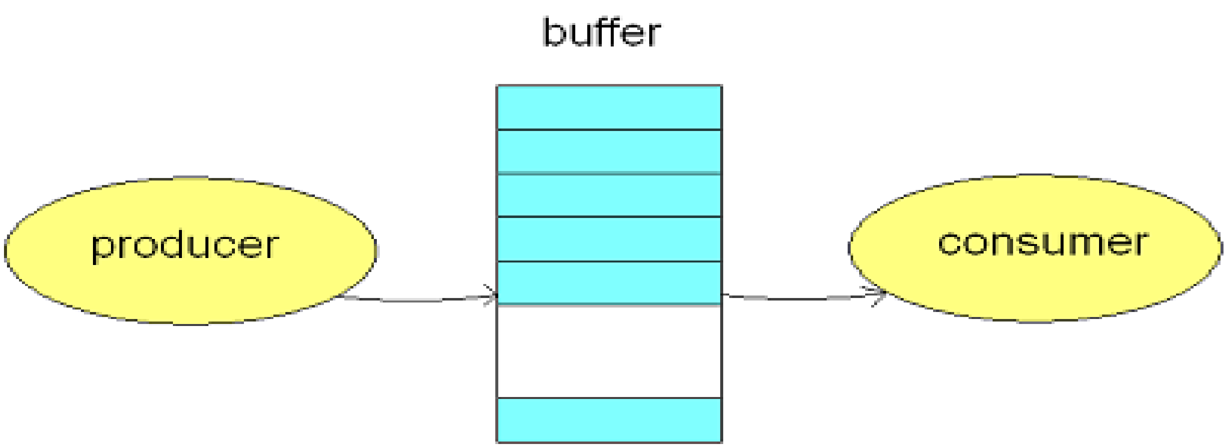
Printer → Consumer

Producer Consumer Problem

Solution:

To avoid mismatch of items Produced or Consumed → Take Buffer

Idea is: Instead of sending items from Producer to Consumer directly→ Store items into buffer



Producer Consumer Problem

Buffer Can be:

1. Unbounderd Buffer:

    1. No buffer size limit
    2. Any no. of items can be stored
    3. Producer can produce on any rate, there will always be space in buffer

2. Bounderd Buffer:

    1. Limited buffer size

Producer Consumer Problem

Bounderd Buffer:

If rate of Production > rate of Consumption:

Some items will be unconsumed in buffer

If rate of Production < rate of Consumption:

At some time buffer will be empty

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

Race Condition

- When multiple processes access and manipulate the same data at the same time, they may enter into a race condition.
- **Race Condition: When output of the process is dependent on the sequence of other processes.**
- Race Condition occurs when processes share same data

Process P1	**Process P2**
1. reads i=10	
2. i=i+1 =11	
	1. P2 reads i=11 from memory 2. i=i+1 = 12
3. Stores i=11 in memory	3. Stores 12 in memory

Critical Section Problem

- Section of code or set of operations, in which process may be changing shared variables, updating common data.
- **A process is in the critical section if it executes code that manipulate shared data and resources.**
- Each process should seek permission to enter its critical section→ **Entry Section**
- **Exit Section**
- **Remainder section:** Contains remaining code

Structure of a process

Locks are set here

Critical Section (a section of code where processes work with shared data)

Locks are released here

Repeat

{

// Entry Section

Critical Section

// Exit Section

Remainder Section

} until false.

Solution to: Critical Section

1. Mutual Exclusion:

It states that if one process is executing in its critical section, then no other process can execute in its critical section.

2. Bounded Wait:

It states that if a process makes a request to enter its critical section and before that request is granted, there is a limit on number of times other processes are allowed to enter that critical section.

Solution to: Critical Section

3. Progress:

It states that process cannot stop other process from entering their critical sections, if it is not executing in its CS.

i.e. Decision of entry into Critical Section is made by processes in **Entry Section** By Setting Lock in Entry Section

Once process leaves C.S, Lock is released (i.e in Exit Section)

Solution to: Critical Section

3. Progress: Cont..

- N processes share same Critical Section

- Decision about which process can enter the Critical Section is based on Processes which are interested to enter the C.S

(i.e. Processes of Entry Section)

- Processes that do not want to execute in CS should not take part in decision

Solutions For C.S

Software Solutions

Consider 2 Process Syatem

- **Algorithm 1.**

Consider a Shared Variable, that can take 2 values: 1 or 2

a) if (shared variable == 1)

P1 can enter to Critical Section

b) if (shared variable == 2)

P2 can enter to Critical Section

2 Process S/w Approaches For C.S

- Let shared variable is **turn** **i.e.** turn : 1 or 2

Pi:

while (turn != i)

{ do skip; }

Pi is in Critical Section;

turn=j;

Thus Mutual Exclusion is Satisfied

2 Process S/w Approaches For C.S

- **Progress**

Now, Turn =2 i.e. process Pj can enter C.S.

But

Suppose **Pj does not want to enter C.S**.

So.. turn =Pi is never set

Progress is not satisfied.

2 Process S/W Approach/Solution

Algorithm 2:

Algorithm 1 does not focus on state of process just consider which process is in critical section.

Sol: Replace variable **“turn”** with an array containing 2 values(True and False)

boolean flag[2].

i.e. use flags

Elements of the array are initialized to false

Algorithm 2

do{

flag[i] = true; //Pi is ready to enter C.S

while (flag[j]); //Pi checks for Pj whether Pj is ready to enter to C.S or not

Critical Section //Pi enters to C.S

flag[i]= false; Pi allows other processes to enter the C.S

Remainder Section

} while (1);

Algorithm 3: Peterson's Solution

- It is a Two process solution
- 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

do {

flag[i] =true; // Pi is ready to enter C.S

turn=j;

while(flag[j] && turn ==j); // (flag and turn of Pj is True, Until this condition is true, Pi can not enter C.S)

CRITICAL SECTION

flag[i]=false;

REMAINDER SECTION

}while(1);

Synchronization Hardware

Hardware Solution to C.S.

- Many systems provide hardware support for critical section code

- Uni-processors – could disable interrupts
  - Currently running code would execute without preemption

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