Process synchronization

Revya Naik V
Assistant Professor
Computer science & engineering
RGUKT Basar

objectives

- 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.

Cooperating processes concurrent access

- 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-consumer

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++;
}
```

Producer-consumer

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}

The Critical section problem

- Consider a system consisting of n processes {P0,P1,......Pn-1}. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on.
- When one process is executing in its critical section, no other process is to be allowed to execute in its critical section.

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

Critical section

- The critical section problem is to design a protocol that the processes can use to cooperate.
- Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**.
- The critical section may be followed by an exit section.
- The remaining code is the remainder section.

Solution to Critical-Section Problem

Requirements:

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

Simple lock variable

Initial lock value is zero

```
        lock=0;
do{
        while(lock!=0);
          Lock=1; entry section
          Critical section;
          Lock = 0 ; exit section

}while(1);
```

Strict alternation

Process Pi:

Non critical section ;

while(turn!=i);   //entry section process pi enters into C.S

Critical section;

turn=j;   /*Exit section   //process pi is holding process pj critical section

Process Pj

Non- critical section;

while(turn!=j);   //entry section process pj enters into C.S

Critical section;

turn=i; //Exit section process pj holding process pi C.S

Progress is not guaranted because one process holding another process critical section.

Hardware solution

Test And Set Instruction

Definition:

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

Must be executed atomically

Hardware solution

Shared Boolean variable lock, initialized to false.

Solution:

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

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!

Process Pi

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

Process Pj

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

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()

  Originally called P() and V()

  Also called down() and up()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations. Two types of semaphores:binary semaphore (range is 0 to 1) and counting semaphore(range is -infinity to +infinity).
- (-ve) value indicates no of processes blocked.

Counting semaphore

- We know the no of processes that are blocked or suspended.

```
Down (semaphore s){
    s.value=s.value-1;
    if(s.value<0){
        Put process PCB in suspend list sleep();
    }
    Else{
        Return;
    }
}
```

Semaphore (up operation)

```
Up (semaphore s){
    s.value=s.value+1;
    if(s.value<=0){
        Select a process from suspend list;
        Wakeup();
    }
}
```

Counting semaphore

- In a certain computation the value of a counting semaphore (S) is initialized to 12, the following up & down operations are performed is the given order. 15P, 12V, 8P, 6V,2P,3V,8P,4V what is the current value of semaphore?

Binary semaphore

```
Down (semaphore s){
    if(s.value==1){
        s.value=0;
        }
        Else{
            Block this process and place in suspend list;
            Sleep();
        }
}
```

Binary semaphore

- We don’t know the no of blocked processes as long as there is any blocked process the value of S remains at 0.

```
up(semaphore s){
   If(suspend lis is empty)
   {
      s.value=1;
   }
   Else {
         Select process from suspend list;
         Wakeup();
   }
}
```

Binary semaphore

- Let B semaphore s=1, 10p, 8v, 16p, 9v, 2p, 4v operations then what is the value of S and length of queue S.L()?

9	1	17	8	10	6
0	0	0	0	0	0

value of s is 0 and queue length is 6.

Bounded buffer problem using semaphore

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.
- Semaphore empty=N // no of empty slots is the buffer
- Semaphore full =0 // no of full slots in the buffer
- Semaphore mutex=1 // used to ensure producer mutual exclusion between producer P & consumer C processes on the buffer.

Producer process

```
Void producer (void){
    Int item p
    While(1){
        produce_item(item p):
        down(empty)   // when buffer is full empty = 0 and producer is blocked
        down(mutex)
        buffer[in]=item p;
        In =(in+1 ) % n;
       up(mutex);
       up(full);
    }
}
```

Consumer process

```
Void consumer (void){
    Int item c;
    While(1){
      down(full)  // when buffer is free full = 0 so consumer will be blocked
      down(mutex)
      Int c=buffer[out]
     out=(out+1) % n
    up(mutex);
    up(empty);
    process_item(item c)
    }
}
```

Readers and writers problem

- 1R-1W (one reader one writer)
- NR-1W (multiple readers – one writer)
- NR-MW (multiple readers -writers)
- Int rc = 0 // readers count in dbms
- Semaphore mutex = 1 // used between readers to access ‘rc’ in a mutual exclusion manner.
- Semaphore db=1 // used between readers & writers to access DBMS in a M.E manner.

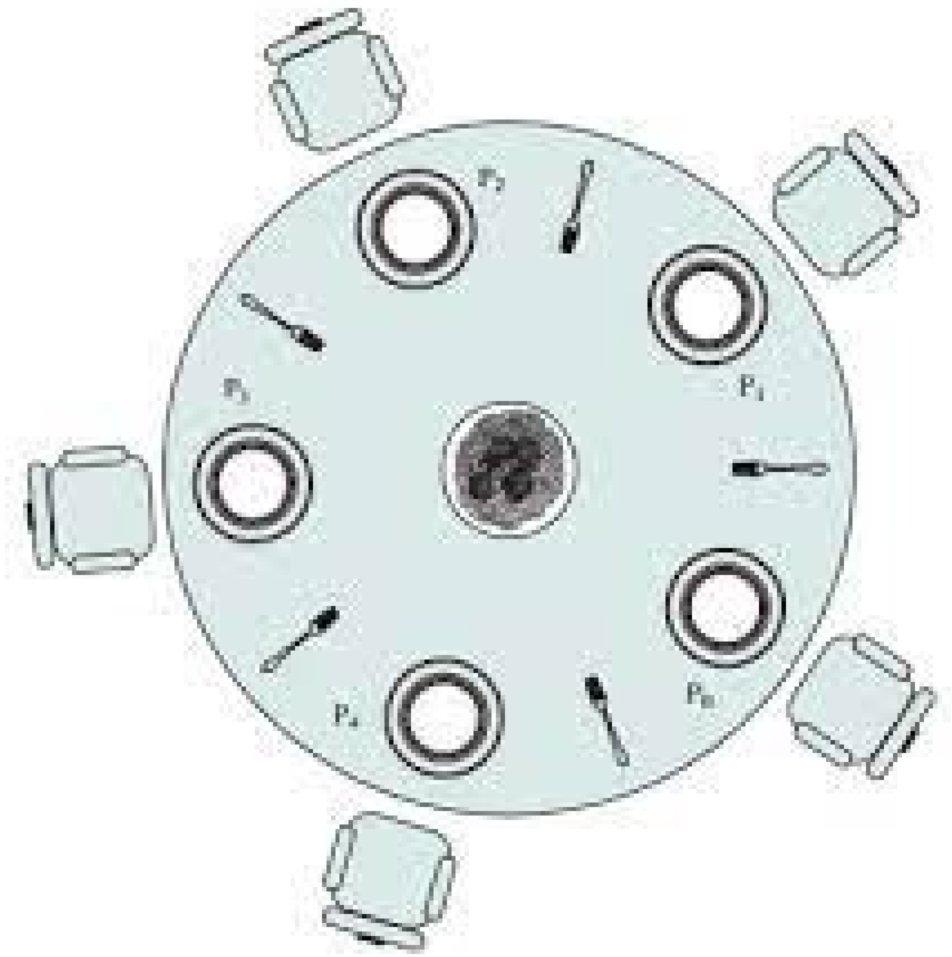
Readers - writers

```
Void reader (void){
   While(1){
     down(mutex);
     rc=rc+1;
     if(rc==1) down (db);
    up(mutex)
     Read DBMS
    down(mutex)
   rc=rc-1;
   if(rc==0) up(db);
   up(mutex)
   }
}
```

readers-writer

```
Void writer(void){
    While(1)
    {
    down(db);
    <write DBMS>
    up(db);
    }
}
```

Dhining philosopher



Dhining philosopher

```
Do{
    Wait(chopstick[i]);
    wait(chopstick[(i+1)%5]);
    ....
    //eat
    ....
    signal(chopstick[i]);
    signal(chopstick[(i+1)%5]);
    ....
    //think
    ....
}while(true);
```

Monitors

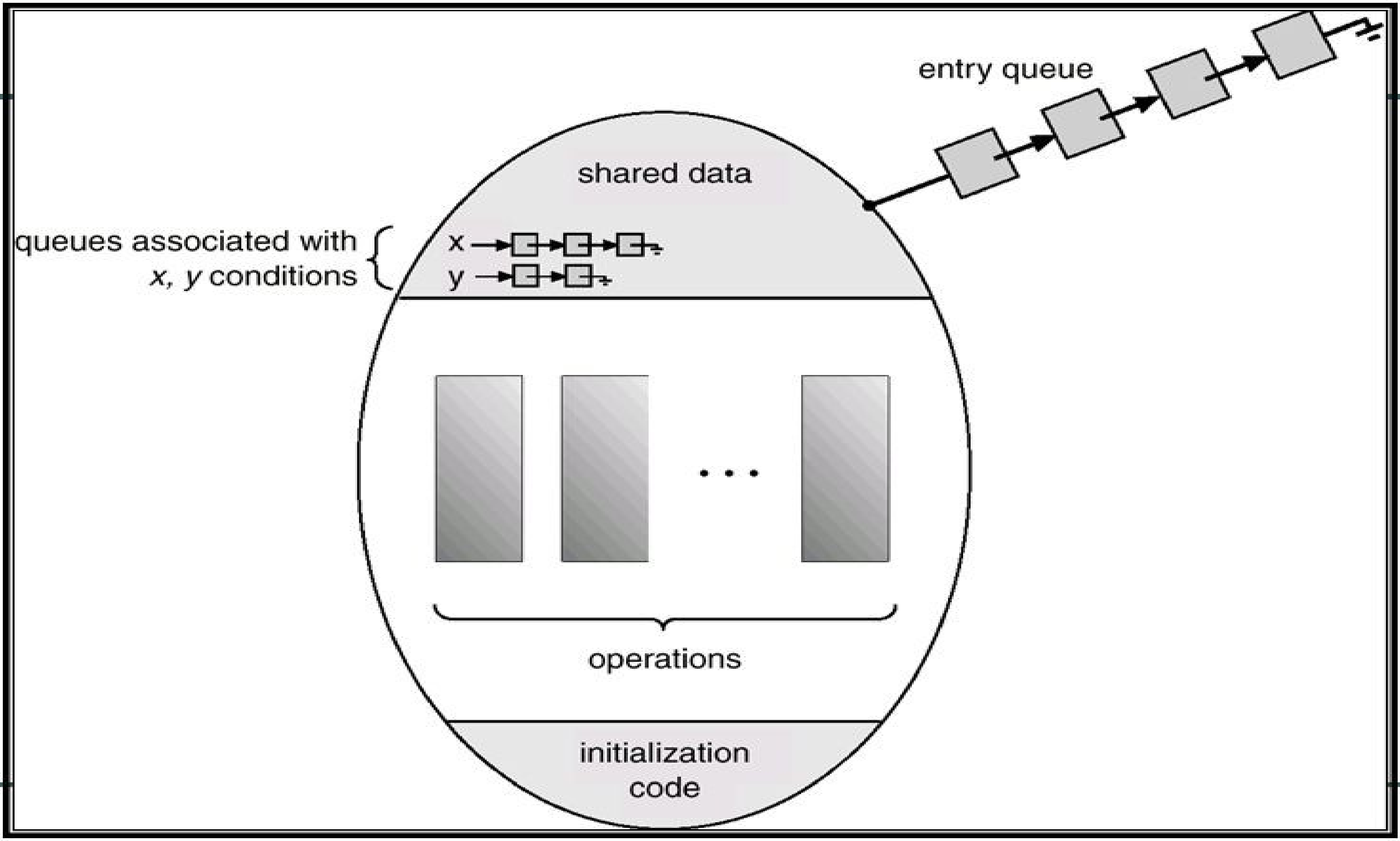
- A high level abstruction that provides a convenient and effective mechanism for process synchronization.
- A monitor type presents a set of programmer defined operations that provides mutual exclusion with the monitor.
- A procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.
- Similarly the local variables of a monitor can be accessed by only the local procedures.

Monitor

- The monitor construct ensures that only one process at a time can be active within the monitor.
- Condition construct condition x,y; the only operatons that can be invoked on a condition variable are wait() and signal();
- The operation x.wait(); mean that the process invoking this operation is suspended until another process invokes x.signal();
- The x.signal() operation resumes exactly one suspended process.

Monitor

Schematic View of a Monitor



monitor

```
Monitor   monitor name
{
   //shared variable declarations
   Procedure P1(.......){
   ..............
   }
   Procedure P2(......){
   .........
   }
   .
   .
   .
   Procedure Pn(.....){
   .......
   }
}
```

Dining philosophers solution using monitor

enum{THINKING, HUNGRY, EATING} state[5];

- Philosopher i can set the varilabe state[i]=EATING only if her two neighbors not eating: (state[(i+4)%5]!=EATING) and (state[(i+1)%5]!=EATING).
- Condition self[5]; in which philosopher i can delay herself when she is hungry but is unable to obtain the chopsticks she needs.

DP solution using monitor

```
Monitor dp{
    enum{THINKING, HUNGRY, EATING} state[5];
    Condition self[5];
    Void pickup(int i){
    state[i]=HUNGRY;
    test(i);
    If(state[i]!=EATING)
    self[i].wait();
    }
```

DP solution using monitor

```
Void putdown(int i){
  state[i]=THINKING;
 test((i+4)%5);
test((i+1)%5);
}
Void test(int i){
if((state[(i+4)%5]!=EATING) && (state[i]==HUNGRY) && (state[(i+1)%5]!=EATING))
    {
 state[i]=EATING;
 Self[i].signal();
}
}
```

DP solution using monitor

```
initialization_code(){
    for(int i=0; i<5; i++){
      state[i]=THINKING;
      }
}
}
```