UNIT-3

Process Synchronization

Our original solution allowed at most BUFFER_SIZE - 1 items in the buffer at the same time.

Suppose we want to modify the algorithm to remedy this deficiency. One possibility is to add an integer variable counter, initialized to 0.

→ counter is incremented every time we add a new item to the buffer.

→ It is decremented every time we remove one item from the buffer.

Code for producer process

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

Code for Consumer process

```
while (true)
{
    while (counter == 0);
        /* do nothing */
    next_consumed = buffer[out];
    out = (out+1) % BUFFER_SIZE;
    counter--;
    /* Consume the item in next_consumed */
}
```

"Counter ++" may be implemented in machine language

register$_1$ = Counter
register$_1$ = register$_1$ + 1
Counter = register$_1$

while register$_1$ is one of the local CPU register.
Similarly, the statement "Counter--" is implemented as follows

register$_2$ = Counter
register$_2$ = register$_2$ - 1
Counter = register$_2$

Again register$_2$ is one of the local CPU register.

The Concurrent execution of "Counter++" and "Counter--" is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order

T_0: producer execute register$_1$ = counter [register$_1$ = 5]
T_1: producer execute register$_1$ = register$_1$ + 1 [register$_1$ = 6]
T_2: Consumer execute register$_2$ = Counter [register$_2$ = 5]
T_3: Consumer execute register$_2$ = register$_2$ - 1 [register$_2$ = 4]
T_4: producer execute Counter = register$_1$ [Counter = 6]
T_5: Consumer execute Counter = register$_2$ [Counter = 4]

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.

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

The Critical Section Problem

→ 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~~

→ Critical section is the part of a program which tries to access shared resources.

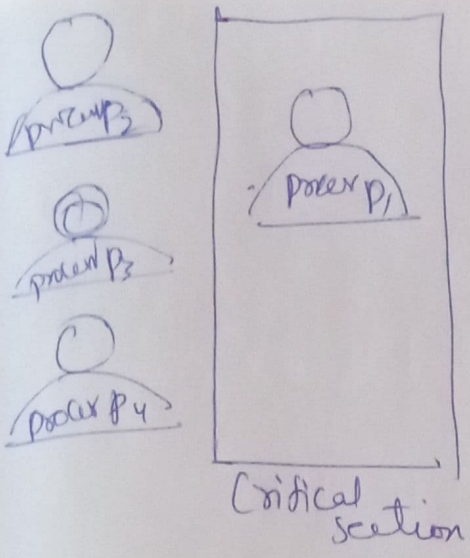
→ The section of code implementing this request is the entry section

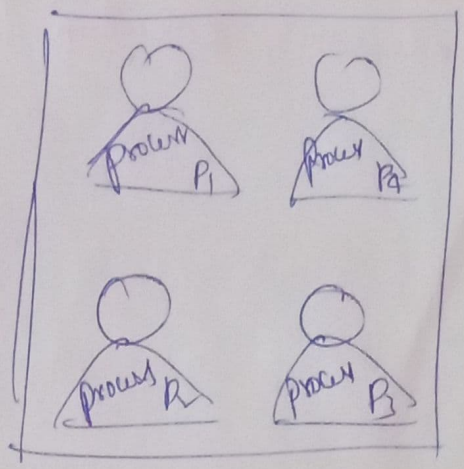
- The critical section may be followed by an exit section
- The remaining code is the remainder section

A solution to the critical-section problem must satisfy the following 3 requirements:

(1) Mutual Exclusion

One solution must provide mutual exclusion. By mutual exclusion, we mean that if one process is executing inside critical section then the other process must not enter in the critical section





X

Progress :- progress means that if one process doesn't need to execute into critical section then it should not stop other processes to get into the critical section

Bounded waiting :- we should be able to predict the waiting time for every process to get into the critical section. the process must not be endlessly waiting for getting into the critical section

Two general approaches are used to handle critical sections in operating systems :

- premptive kernels
- nonpreemptive kernel

* A preemptive kernel allows a process to be preempted while it is running in kernd mode.
* A non preemptive kernel does not allow a process running in kernel mode to be preempted;

Synchronization Hardware

In synchronization hardware, we explore several more solutions to the critical-section problem using techniques ranging from hard... to software based APIs available to applica... programmer.

These solutions are based on the premise of locking; however the design of such locks can be quite sophisticated.

→ These hardware features can make any programming task easier and improve system efficiency

→ System efficiency decreases when this manage... passing delays entry into each critical section.

→ So many Many modern computer systems therefore provide special hardware instructions that allow us that we can test and modify the the content of a word & to swap the Contents of two words atomically - i.e. as one interruptible unit.

The TestandSet() instruction can be defined as shown

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = true;
    return rv;
}
```

Definition of test_and_set instruction.

The essential characteristic is that this instruction is executed atomically. So, if two TestAndSet instructions are executed simultaneously (each on different CPU), they will be executed sequentially in some arbitrary order.

We can implement mutual exclusion by declaring a boolean variable lock, initialized to false.

If the machine supports the TestAndSet() instruction. The structure of process P1 is shown below.

```
do {
    while (test_and_set(&lock));
        /* do nothing */
        /* critical section */
    lock = false;
        /* remainder section */
}
while(true);
```

Fig: Mutual-exclusion with definition of test_and_set() instruction

The swap() instruction, in contrast to the test-and-set() instruction, operates on the contents of two words; it is executed atomically.

Mutual-exclusion can be provided as follows if the machine supports swap() instruction

=> Additionally, each process has a local boolean variable key.

```
int compare_and_swap(int *value, int expected, int new_value)
    int temp = *value)
    if(*value == expected)
        *value = new_value;
    return temp;
}
```

definition of Compare-and-swap()

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

mutual-exclusion implementation with Compare-and-swap()

Since these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement

In below code, we present another algorithm using TestAndSet() instruction that satisfies all the critical-section requirements

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&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);
```

Bounded waiting mutual exclusion with test_and_se

Semaphorey

def:
A semaphore is a shared integer variable apa from initialization we can only perfom two operations on semphore.

Those are wait() and signal()

~~wait~~ P.

The wait() operation ~~di~~ was diginally termed p (from the Dutch proberen, "to test"); signal() was diginally called call V (from Verhogen, "to increment)

The definition of wait() is as follows

```
wait(S){
    while(S<=0
      ;// no-op
    S--;
}.
```

spinlock

The definition of signal() is as follows

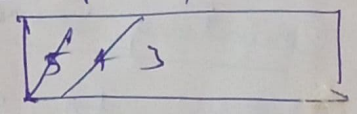
```
Signal(S){
    S++;
}.
```

Two types of Semaphores

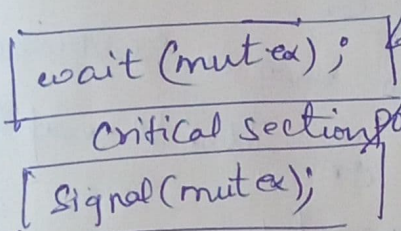
1. Binary Semaphore s=0/1 - used for mutual exclusive locks
2. Counting Semaphore S=0, 1, 2, ... - to control access to a resource.

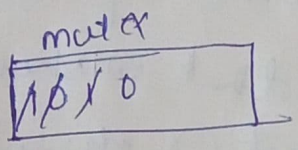
Semaphore mutex=1;

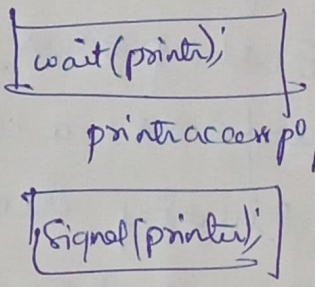
Semaphore printer=5;



do
{





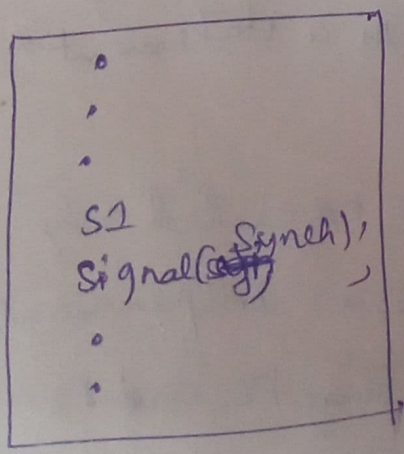


} while(true);

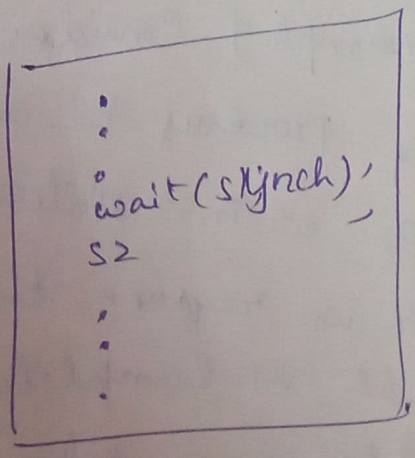
Semaphore s=0;

Synchronization

P1



P2



Explanation in next page.

Usage

Binary semaphores behave similarly to mutex locks.

→ It can be used instead for providing mutual exclusion.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.

Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count) when a process releases a resource it performs signal() operation (incrementing the count)

We can also use semaphores to solve various synchronization problems.

For example, Consider two concurrently running processes P1 and P2 with a statement S2 and P2 with a statement S1

Suppose we require that S2 be executed only after S1 has completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0.

P2 will execute S2 only after P1 has invoked signal(synch), which after statement S1 has been executed.

Semaphore producer–consumer problem

- The producer-consumer problem is a synchronization problem.
- There is a fixed size buffer and the producer produces items and enters them into the buffer.
- The consumer removes the items from the buffer and consumes them.
- The producer consumer problem can be resolved using semaphores.

A producer should not produce items into the buffer when the consumer is consuming an item from the buffer and vice versa. So the buffer should only be accessed by the producer or consumer at a time.

The producer and consumer processes share the following data structures:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to 1.

The empty and full semaphores count the number of empty and full buffers.

The semaphore empty is initialized to n.

The semaphore full is initialized to 0.

The Code for the producer process.

```
do{
    /* produce an item in next-produced */
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add next-produced to the buffer */
    ...
    signal(mutex);
    signal(full);
} while(true);
```

The Code for the Consumer process.

```
do{
    wait(full);
    wait(mutex);
    ...
    /* remove an item from buffer to next_consu
    ...
    signal(mutex);
    signal(empty);
    ...
    /* Consume the item in next-consumed */
    ...
} while(true);
```

We can interpret the code as the produ
producing full buffers for the Consume
or as the consumer producing empty
buffers for the producer.

Reader-writer problem

The reader-writer problem relates to an object such as a file that is shared b/w multiple processes. Some of these processes are readers, they want to read the data from the object.
Some of the processes are writers, they want to write into the object.

If two readers access the object at the same time there is no problem.
However, if two writers or a reader and writer access the object at same time there may be problems.
To solve this situation, a writer should get exclusive access to an object. This can be implemented using semaphore.
In the solution to the first reader writer problem, the reader process share the following data structures:

```
semaphore rw_mutex=1;  } initialized to 1
semaphore mutex=1;
int read_count=0;  ---> initialized to 0
```

~~mutex and~~ Writer process

The code that defines the writer process is given below:

```
do{
    wait(rw_mutex);
    ...
    /* writing is performed */
    ...
    signal(rw_mutex);
}while(true);
```

Dining-philosop

If a writer wants to access the object, wait operation is performed. After that no other writer can access the object. When a writer is done writing into the object. Signal operation is performed.

Reader process

The code that defines the reader process is given below:

```
do{
    wait(mutex);
    read_count++;
    if(read_count==1)
        wait(rw_mutex);
    Signal(mutex);
    ...
    /* reading is performed */;
    ...
    wait(mutex);
    readcount--;
    if(read_count==0)
        Signal(rw_mutex);
    Signal(mutex);
}while(true);
```

* The mutex semaphore ensures mutual exclusion and rw_mutex handles the writing mechanism.

* The variable ~~rc denotes~~ read_count denotes the no. of readers accessing the object

~~The vario~~ The mutex s

As soon as read_count becomes 1, wait operation is used on rw_mutex. This means that a writer cannot access the object anymore.

After the read operation is done, read_count is decremented.

When read_count becomes 0, signal operation is used on rw_mutex. So a writer can access the object now.

Dining philosophers problem

Consider 5 philosophers who spend their lives thinking. There are 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of rice for each of the philosophers and 5 chopsticks.

A philosopher needs both their right and left chopstick to eat. A hungry philosopher may only eat if there are both chopsticks available. Otherwise a philosopher puts down her chopstick and begin thinking again.

The dining-philosopher's problem is considered a classic synchronization problem neither because of its practical importance nor because Computer scientists dislike philosophers.

→ It demonstrates a large class of concurrency control problems.

→ It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

One simple solution is to represent each chopstick with a semaphore.

A philosopher tries to grab a chopstick by executing wait() operation on the semaphore. She releases her chopsticks by executing the signal() operation.

Thus, the shared data is

semaphore chopstick[5];

```
do{
   wait(chopstick(i));
   wait(chopstick[(i+1)%5]);
      ...
      /* eat for a while */
      ...
      Signal(chopstick[i]);
      Signal(chopstick[(i+1)%5];
      ...
      /* think for a while */
      ...
}while(true);
```

The above solution makes sure that no two neighboring philosophers can eat at the same time. But this solution lead to a deadlock.

This may happen if all the philosophers pick their left chopstick simultaneously. Then none of them can eat and deadlock occurs.

Some of the ways to avoid deadlock are as follow.

- There should be at most four philosopher on the table.

Monitors

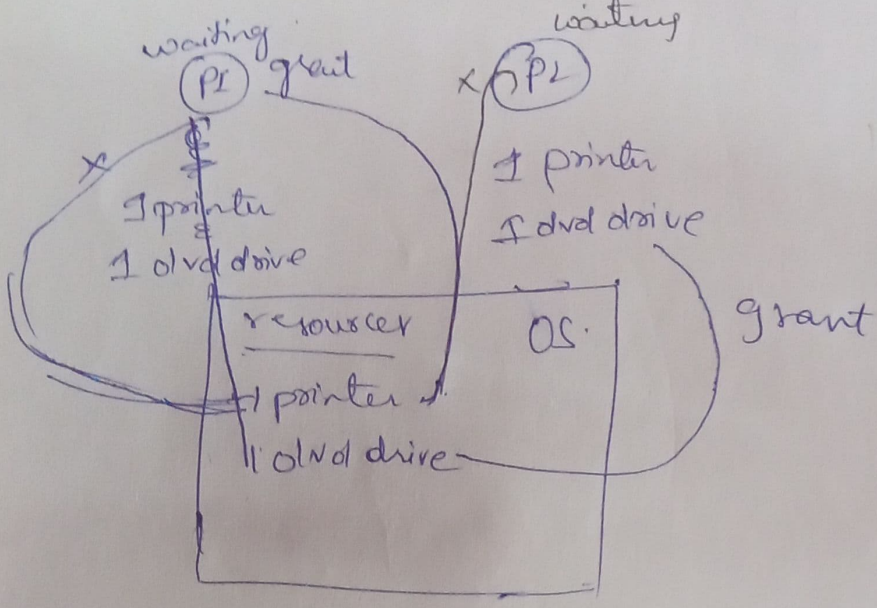
- An even philosopher should pick the right chopstick and then the left chopstick. While an odd philosopher should pick the left chopstick and then right chopstick.
- A philosopher should only be allowed to pick their chopstick if both are available at the same time.

Monitors

Dead locks

Resource request

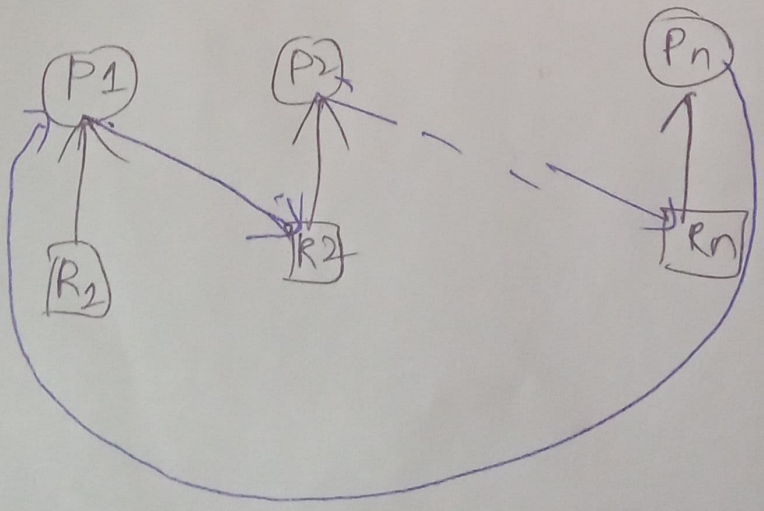
1. Available - it is allocated to process.
2. Not Available - it enters into blocked.



1. Request
2. Use
3. Release.

Necessary Condition for deadlock.

1. Mutual Exclusion.
2. Hold and wait
3. No premption
4. Circular wait



System Resource Allocation graph

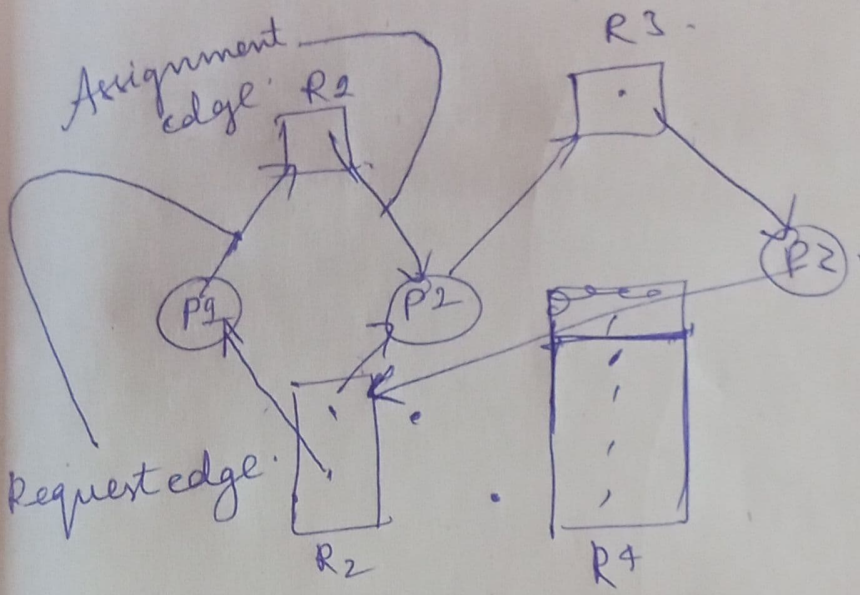
G = (V, E);

V is no empty set of vertices.

E is pair of vertices called edge.

V - process - Circle.

resources - rectangle



If Resource Allocation graph containing cycle (closed path) there is a possibility of deadlock.

P2 → R3 → P3 → R2 → P2

P2 → R3 → P3 → R2 → P1 → R1 → P2.

(graph Contains

→ Each resource have one instance, and having cycle. In that case, compulsory deadlock present (or) exists

→ Resources multiple instance - graph Contains cycle → deadlock may exists.

~~Cycle is~~