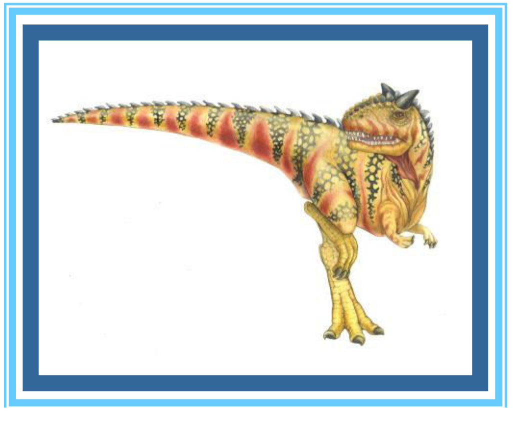
Process Synchronization



Critical Section Problem

At any given point in time, many kernel-mode processes may be active in OS, as a result kernel code is subject to race conditions.

Approaches to handle CS in OS:

- ***Preemptive Kernel*** *– A preemptive kernel allows a process to be preempted while it is running in kernel mode.*
- ***Non-Preemptive Kernels*** *– A non-preemptive kernel doesn't allow a process running in kernel mode to be preempted.*

Synchronization Hardware

- *Hardware features can make any programming task easier and improve efficiency.*
- ***Interrupt prevention*** *(no pre-emption) while a shared variable is being modified, could be a solution to critical section problem on single processor systems. Unfortunately this is not a solution on multiprocessor environment following problems could occur.*
  - Time consuming on multiprocessor.
  - System Efficiency will decrease.
  - System clock could be effected.

Synchronization Hardware

Locking: *Protecting critical regions through the use of* ***locks****.*

Synchronization Hardware(test_and_set Instruction)

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

- *Executed atomically.*
- *Returns original value of passed parameter.*
- *Set new value of passed parameter to TRUE.*

- *Boolean variable lock initialized to false*
- *If lock = false, return false, set lock =true*
- *If lock = true, return true, lock true*

Solution

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

Synchronization Hardware (Compare_and_swap())

```
int compare_and_swap(int *value, int expected, int
 new_value) {
        int temp = *value;

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

- *Boolean variable lock initialized to 0*

Solution

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

Synchronization Hardware bounded waiting

```
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);
```

Synchronization Hardware bounded waiting

Entry section

```
waiting[i] = true;
key = true;
while (waiting[i] && key)
key = test and set(&lock);
waiting[i] = false;
```

	P0	P1	P2	P3
Waiting [i]	F	F	F	F
Key				

lock

F

Critical Section

Exit section

```
 j = (i + 1) % n;
while ((j != i) && !waiting[j])
 j = (j + 1) % n;
 if (j == i)
 lock = false;
   else
waiting[j] = false;
```

TestAndSet

if lock=false, return false,lock=true

if lock=true,return true,lock=true

Hardware Solutions

- *Complicated Solutions.*
- *Generally inaccessible to application programmers.*

Mutex locks

- ***Mutex lock:*** *It a software tool to solve critical section problem and it is taken from (**Mut**ual **Ex**clusion).*
- *Mutex locks are used to protect critical section and to prevent race conditions.*
- *A process must **acquire the lock** before entering a critical section, it **releases the lock** when it exits the critical section.*
- *The **acquire()** function will acquire the lock and **release()** function will release the lock.*

Mutex locks

do{

acquire lock

critical section

release lock

remainder section

}while(true);

acquire(){

while (***!Available***);

available =false;

}

release(){

available =true;

}

Calls to acquire and release should be atomic usually implemented by hardware instructions.

Mutex locks disadvantage

- ***Busy waiting:*** *While a process is in its critical section any other process that tries to enter its critical section must loop continuously in the call to acquire(), this type of mutex lock is also called a spinlock because the process "spins" while waiting for lock to become available.*
- *This continual looping is clearly a problem in real multiprogramming systems where single CPU is shared among many processes.*
- *Busy waiting* ***waste CPU cycles*** *that some other process might be able to use productively.*

Busy WaitingAdvantage

- ***Busy waiting/Spinlock:*** No context switch is required, when a process must wait on lock, and context may take considerable time.
- On multiprocessor systems one thread can spin on one processor while another thread performs its critical section on another processor.

Semaphore

- *Semaphore tools provides a more sophisticated way (than mutex locks) for process to **synchronize** their activities.*
- *Semaphore is a **data type***
  - *Value (Integer)*
  - *Operations*
- *A **semaphore (S)** is an **integer variable** that, apart from initialization, is accessed only through two standard **atomic operations**: **wait(S)** and **signal(S).***
- *The wait() operation was originally termed as **P** and signal operation is termed as **V**.*

Semaphore: Operations

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

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

Semaphore: Applications:

- *Semaphore are used to solve the critical section problem*
- *Process Synchronization*
- *Resource management.*

Types of Semaphore

There are two main types of semaphores:

- ***Binary Semaphores:*** *This is also known as* ***mutex lock*** *and their value can range between 0 and 1.*
  - ***It is used to implement the solution of critical section problem with multiple processes.***
- ***Counting Semaphores:*** *Its value can range over an unrestricted domain. It* ***is used to control access to a resource*** *that has multiple instances.*

Types of Semaphore

- ***Counting semaphores*** *are used to coordinate the resource access, where the semaphore count is the number of available resources.*
- *Each process that wishes to use resource need to perform wait() on semaphore, when process release the resource it will perform signal() on semaphore.*
- *When count of the semaphore goes to 0 all resources are being used. After that wish to use a resource will block until count becomes greater than 0.*

Types of Semaphore

- ***Semaphore can be used to solve various synchronization problems.***

Example: Consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2. Suppose we want that S2 should be execute only after S1 has completed.

S1:	***wait (S);***
signal(S)	***S2***

Semaphores without busy waiting

- *With each semaphore there is an associated waiting queue.*
- *Each entry in a waiting queue has two data items:*
  - *value (of type integer)*
  - *a pointer to the next record in the list*
- *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.*

Semaphores without busy waiting

```
Typedef struct{
    int value;
    Struct process *list;
}semaphore;
```

```
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);
    }
}
```

Advantages of Semaphores

Some of the advantages of semaphores are as follows:

- They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
- There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.

Limitations of Semaphores

- **Deadlock** – *two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes. Let* ***S*** *and* ***Q*** *be two semaphores initialized to 1*

P_0	P_1
wait(S);	***wait(Q);***
wait(Q);	***wait(S);***
...	**...**
signal(S);	***signal(Q);***
signal(Q);	***signal(S);***

- ***Starvation*** – ***indefinite blocking***
  - *A process may never be removed from the semaphore queue in which it is suspended*
- ***Priority Inversion*** – *Scheduling problem when lower-priority process holds a lock needed by higher-priority process*
  - *Solved via* ***priority-inheritance protocol***

Semaphores Example - 1

A shared variable x, initialized to zero, is operated on by four concurrent processes W, X, Y, Z as follows. Each of the processes W and X reads x from memory, increments by one, stores it to memory, and then terminates. Each of the processes Y and Z reads x from memory, decrements by two, stores it to memory, and then terminates. Each process before reading x invokes the P operation (i.e., wait) on a counting semaphore S and invokes the V operation (i.e., signal) on the semaphore S after storing x to memory. Semaphore S is initialized to two. What is the maximum possible value of x after all processes complete execution?

(A) -2

(B) -1

(C) 1

(D) 2

```
Process P:
while (1) {
W:
    print '0';
    print '0';
X:
}
```

```
Process Q:
while (1) {
Y:
    print '1';
    print '1';
Z:
}
```

Synchronization statements can be inserted only at points W, X, Y and Z.

Which of the following will always lead to an output staring with '001100110011' ?

(A) ***P(S)*** at ***W***, ***V(S)*** at ***X***, ***P(T***) at ***Y***, ***V(T)*** at ***Z***, ***S and T initially 1***

(B) ***P(S)*** at ***W***, ***V(T)*** at ***X***, ***P(T)*** at ***Y***, ***V(S)*** at ***Z***, ***S initially 1, and T initially 0***

(C) ***P(S)*** at ***W***, ***V(T)*** at ***X***, ***P(T)*** at ***Y***, ***V(S)*** at ***Z***, ***S and T initially 1***

(D) ***P(S)*** at ***W***, ***V(S)*** at ***X***, ***P(T)*** at ***Y***, ***V(T)*** at ***Z***, ***S initially 1, and T initially 0***

Bounded Buffer Problem

- *There is a buffer of n slots and each slot is capable of storing* ***one unit*** *of data. There are two processes running, namely, producer and consumer, which are operating on the buffer.*
- *A producer tries to* ***insert*** *data into an* ***empty*** *slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. Two processes won't produce the expected output if they are being executed concurrently.*
- *There needs to be a way to make the producer and consumer work in an independent manner.*

Solution of Bounded Buffer Problem

- ***mutex***, *a* ***binary semaphore*** *which is used to acquire and release the lock.*
- ***empty***, *a* ***counting semaphore*** *whose initial value is the number of slots in the buffer, since, initially all slots are empty.*
- ***full***, *a* ***counting semaphore*** *whose initial value is 0*

The structure of the **producer** process

```
do  {
          //   produce an item in nextp

          wait (empty);
          wait (mutex);

          //  add the item to the  buffer

          signal (mutex);
          signal (full);

} while (TRUE);
```

The structure of the **consumer** process

```
do {
          wait (full);
          wait (mutex);

          //  remove an item from
          //  buffer to nextc

          signal (mutex);
          signal (empty);

          //  consume the item in nextc
} while (TRUE);
```

Solution of Bounded Buffer Problem

The structure of the **producer** process

```
do  {
        //   produce an item in nextp

        wait (empty);
        wait (mutex);

        //  add the item to the  buffer

        signal (mutex);
        signal (full);

} while (TRUE);
```

The structure of the **consumer** process

```
do {
        wait (full);
        wait (mutex);

        //  remove an item from
        //  buffer to nextc

        signal (mutex);
        signal (empty);

        //  consume the item in nextc
} while (TRUE);
```

Readers Writers Problem

A data set is shared among a number of concurrent processes

Readers *– only read the data set; they do not perform any updates*

Writers *– can both read and write.*

- ***Problem*** *– allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.*

Solution of Readers Writers Problem

Shared Data:

Semaphore ***mutex*** initialized to ***1***.

Semaphore ***wrt*** initialized to ***1***.

Integer ***readcount*** initialized to ***0***.

The structure of a ***writer*** process

```
while (true) {
        wait (wrt) ;
        write operation
        signal (wrt) ;
}
```

The structure of a ***reader*** process

```
while (true) {
    wait (mutex) ;
    readcount ++ ;
    if (readercount == 1)
        wait (wrt) ;
    signal (mutex)

        Read Operation

    wait (mutex) ;
    readcount  - - ;
    if (redacount  == 0)
        signal (wrt) ;
    signal (mutex) ;
}
```

Dining-Philosophers Problem

- *There are five silent philosophers (P1 – P5) sitting around a circular table, and they **eat and think alternatively**.*
- *There is a bowl of rice for each of the philosophers and 5 chopsticks (1 – 5).*
- *To be able to eat, a **philosopher needs to have chopstick in both his hands**. A hungry philosopher may only eat if there are both chopsticks available.*
- *After eating a philosopher puts down their chopstick and begin thinking again.*

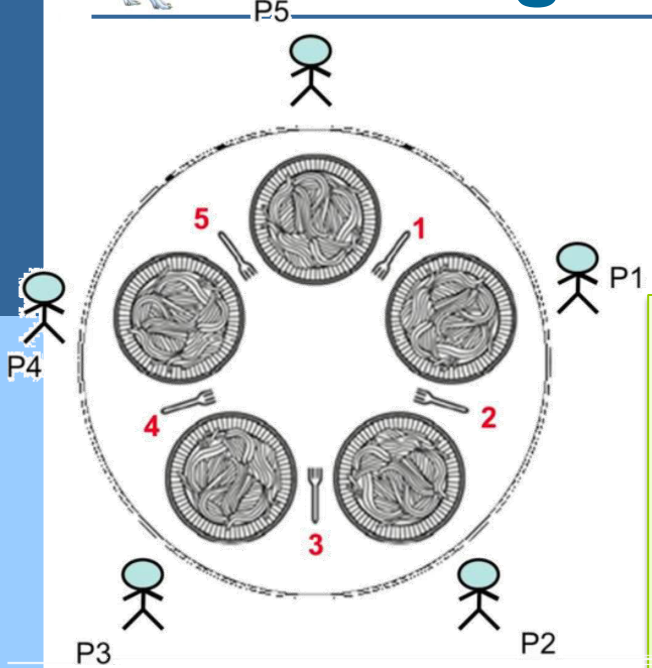
Dining-Philosophers Problem Solution

- *A solution of the Dining Philosophers Problem is to use a semaphore to represent a chopstick.*
- *A chopstick can be picked up by executing a* ***wait*** *operation on the* ***semaphore*** *and released by executing a* ***signal*** *semaphore.*
- *The structure of the chopstick is shown below:*

  semaphore chopstick [5];

- *Initially the elements of the chopstick are initialized to 1 as the chopsticks are on the table and not picked up by a philosopher.*

Dining-Philosophers Problem Solution



semaphore chopstick [5];

```
do {
        wait( chopstick[i] );
        wait( chopstick[ (i+1) % 5] );
  .
        Eating the Rice

        signal( chopstick[i] );
        signal( chopstick[ (i+1) % 5] );

        Thinking
.
} while(1);
```

Difficulty with the solution

The above solution makes sure that no two neighboring philosophers can eat at the same time. But this solution can 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 follows:*

- ***There should be at most four philosophers on the table.***
- ***An even philosopher should pick the right chopstick and then the left chopstick while an odd philosopher should pick the left chopstick and then the right chopstick.***
- ***A philosopher should only be allowed to pick their chopstick if both are available at the same time.***