**UNIT-II**

**Process Synchronization**

**Syllabus:**

The Critical-Section Problem, Peterson’s Solution, Synchronization Hardware, Semaphores, Monitors, Classical problems of synchronization, Case Study : Process Synchronization in Linux.

**Objectives:**

* To present the concept of 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 examine several classical process-synchronization problem
* To explore several tools those are used to solve process synchronization problems
* Concurrent access to shared data may result in data inconsistency.
* Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
* When 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 **race condition.**
* To guard against the race condition, we need to ensure that only one process at a time can be manipulating the variable counter. Hence processes must be synchronized.

**The Critical Section Problem**

* A critical section is a piece of code that accesses a shared resource (data structure or a device) that must not be concurrently accessed by more than one thread of execution.
* The execution of critical sections by the processes is mutually exclusive in time.
* The goal is to provide a mechanism by which only one instance of a critical section is executing for a particular shared resource.
* The critical-section problem is to design a protocol that the processes can use to cooperate.

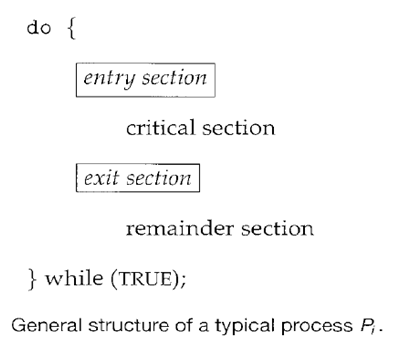
A Critical Section Environment contains:

**Entry Section**Code requesting entry into the critical section.

**Critical Section**Code in which only one process can execute at any one

***Exit Secti***on The end of the critical section, releasing or allowing others

***Remainder Section*** Rest of the code AFTER the critical section



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

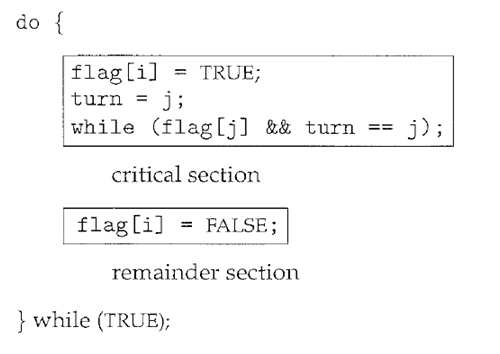
1. **Mutual Exclusion**. If process *Pi* is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress**. If no process is executing in its critical section and there exist some processesthat 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 areallowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted. (i.e. All requesters must eventually be let into the critical section).

**Peterson’s solution**

* A classic software based solution to the critical section problem is known as Peterson’s solution.
* It provides a good algorithmic description of solving the critical section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress and bounded waiting requirements.
* Peterson’s solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
* Peterson’s solution requires two data items to be shared between the two processes:
* The variable turn indicates whose turn it is to enter its critical section.
* The flag array is used to indicate if a process is ready to enter its critical section, flag[i] = true indicates that process Pi is ready.



The structure of process **Pi** in Peterson's solution.

1. Mutual Exclusion: P0and P1can never be in the critical section at the same time: If P0is inits critical section, then flag [0] is true and either flag [1] is false (meaning P1 has left its critical section) or turn is 0 (meaning P1 is just now trying to enter the critical section, but graciously waiting). In both cases, P1 cannot be in critical section when P0 is in critical section.

2. Progress: Each process can only be blocked at the while if the other process wants to usethe critical section (flag[ j ] = = true ), AND it is the other process's turn to use the critical section ( turn = = j ). If both of those conditions are true, then the other process ( j ) will be allowed to enter the critical section, and upon exiting the critical section, will set flag[ j ] to false, releasing process i. The shared variable turn assures that only one process at a time can be blocked, and the flag variable allows one process to release the other when exiting their critical section.

3. Bounded Waiting: As each process enters their entry section, they set the turn variable to bethe other processes turn. Since no process ever sets it back to their own turn, this ensures that each process will have to let the other process go first at most one time before it becomes their turn again.

**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

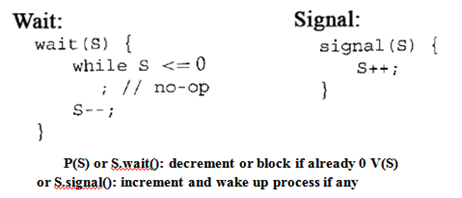
waiting[j] = FALSE;

// remainder section

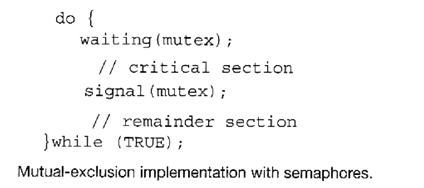
} while (TRUE);

**Semaphores**

* The various hardware based solutions can be difficult for application programmers to implement.
* **Semaphores** are most often used to synchronize operations (to avoid race conditions) when multiple processes access a common, non-shareable resource.
* *Semaphores* are integer variables for which only two (atomic) operations are defined, the **wait (P)** and **signal (V)** operations, whose definitions in pseudo code are shown in the following figure.



* To indicate a process has gained access to the resource, the process decrements the semaphore.
* Modifications to the integer value of the semaphore in the wait and signal operations must be executed indivisibly.
* When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
* Access to the semaphore is provided by a series of semaphore system calls.
* Semaphores can be used to deal with the *n-pro*cesses critical section problem, where the *n-processes* share a semaphore**mutex**(mutual exclusion) initialized to 1. Each process Piisorganized as shown:



**Implementation:**

wait (S) {

while S <= 0

; // no-op

S--;

}

signal (S) {

S++;

}

**Semaphore as General Synchronization Tool**

* Counting semaphore – integer value can range over an unrestricted domain
* Binary semaphore – integer value can range only between 0   
  and 1; can be simpler to implement
* Also known as mutex locks
* ,Can implement a counting semaphore S as a binary semaphore
* Provides mutual exclusion

semaphore mutex; // initialized to

do {

wait (mutex);

// Critical Section

signal (mutex);

// remainder section

} while (TRUE);

**Semaphore Implementation**

* Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
* Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
* Could now have busy waiting in critical section implementation
* But implementation code is short
* Little busy waiting if critical section rarely occupied
* Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

**Semaphore Implementation with no 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)
* pointer to 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.

Implementation of wait:

wait(semaphore \*S) {

S->value--;

if (S->value < 0) {

add this process to S->list;

block();

}

}

Implementation of signal:

signal(semaphore \*S) {

S->value++;

if (S->value <= 0) {

remove a process P from S->list;

wakeup(P);

}

}

**Deadlock and Starvation**

* 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

**Classical Problems of Synchronization**

* Bounded-Buffer Problem
* Readers and Writers Problem
* Dining-Philosophers Problem

**Bounded-Buffer Problem**

* *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.
* 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
* nProblem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time
* Shared Data
* Data set
* Semaphore mutex initialized to 1
* Semaphore wrt initialized to 1
* Integer readcount initialized to 0

The structure of a writer process

do {

wait (wrt) ;

// writing is performed

signal (wrt) ;

} while (TRUE);

The structure of a reader process

do {

wait (mutex) ;

readcount ++ ;

if (readcount == 1)

wait (wrt) ;

signal (mutex)

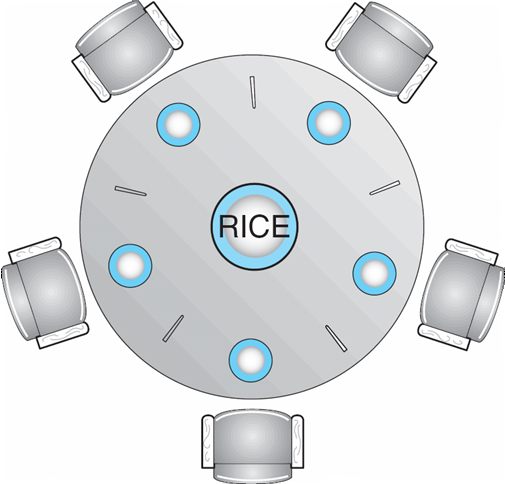
// reading is performed

wait (mutex) ;

readcount - - ;

if (readcount == 0)

signal (wrt) ;

 signal (mutex) ;

} while (TRUE);

**Dining-Philosophers Problem**

* Shared data
* Bowl of rice (data set)
* Semaphore chopstick [5] initialized to 1
* The structure of Philosopher *i*:

do {

wait ( chopstick[i] );

wait ( chopStick[ (i + 1) % 5] );

// eat

signal ( chopstick[i] );

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

// think

} while (TRUE);

**Problems with Semaphores**

Incorrect use of semaphore operations:

signal (mutex)

….

wait (mutex)

wait (mutex) …

wait (mutex)

Omitting of wait (mutex) or signal (mutex) (or both)

**Monitors**

A high-level abstraction that provides a convenient and effective mechanism for process synchronization

Only one process may be active within the monitor at a time

monitor monitor-name

{

// shared variable declarations

procedure P1 (…) { …. }

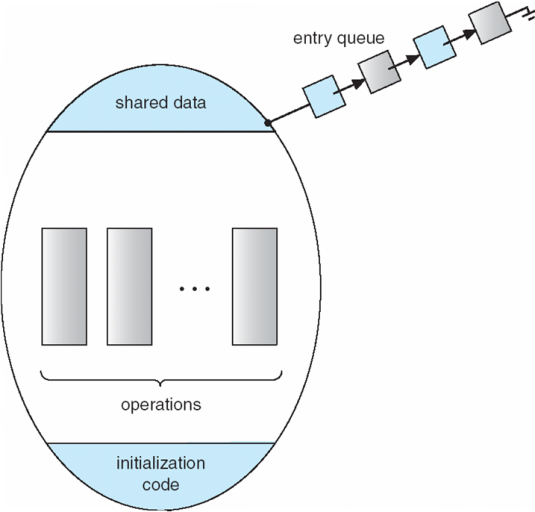
…

procedure Pn (…) {……}

Initialization code ( ….) { … }

…

}

****}

**Schematic view of a Monitor**

**Condition Variables**

condition x, y;

Two operations on a condition variable:

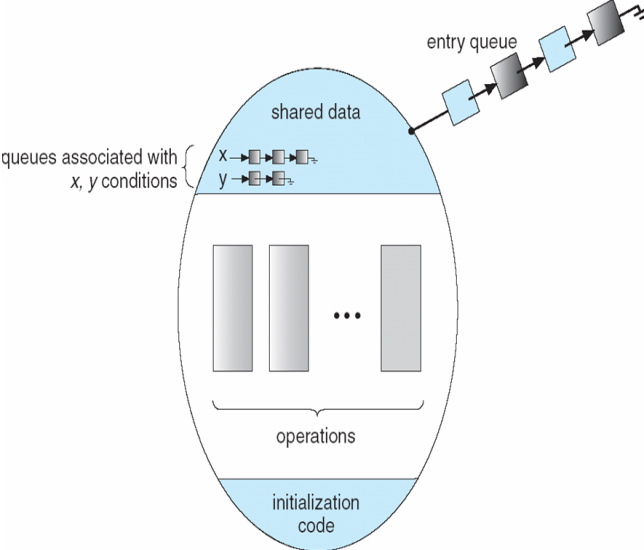
x.wait () – a process that invokes the operation is

suspended.

x.signal () – resumes one of processes (if any) that

invoked x.wait ()

**Monitor with Condition Variables**



**Solution to Dining Philosophers**

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;

}

void putdown (int i) {

state[i] = THINKING;

// test left and right neighbors

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 () ;

}

}

initialization\_code() {

for (int i = 0; i < 5; i++)

state[i] = THINKING;

}

}

Each philosopher *I* invokes theoperations pickup()

and putdown() in the following sequence:

DiningPhilosophters.pickup (i);

EAT

DiningPhilosophers.putdown (i);

**Monitor Implementation Using Semaphores**

**Variables**

semaphore mutex; // (initially = 1)

semaphore next; // (initially = 0)

int next-count = 0;

nEach procedure ***F*** will be replaced by

wait(mutex);

…

body of *F*;

…

if (next\_count > 0)

signal(next)

else

signal(mutex);

nMutual exclusion within a monitor is ensured.

**Monitor Implementation**

For each condition variable ***x***, we have:

semaphore x\_sem; // (initially = 0)

int x-count = 0;

nThe operation x.waitcan be implemented as:

x-count++;

if (next\_count > 0)

signal(next);

else

signal(mutex);

wait(x\_sem);

x-count--;

The operation x.signal can be implemented as:

if (x-count > 0) {

next\_count++;

signal(x\_sem);

wait(next);

next\_count--;

}

**A Monitor to Allocate Single Resource**

monitor ResourceAllocator

{

boolean busy;

condition x;

void acquire(int time) {

if (busy)

x.wait(time);

busy = TRUE;

}

void release() {

busy = FALSE;

x.signal();

}

initialization code() {

busy = FALSE;

}

}