|  |
| --- |
| **UNIT – III** |
| **Process Synchronization**: The Critical-Section Problem, Semaphores, Monitors, Message Passing, Classical IPC problems (Readers-Writers, Dining philosophers and producer & consumer problems). **Deadlocks**: Resources, Conditions for resource deadlocks, deadlock avoidance, deadlock prevention. Deadlock detection and recovery. |

1. **The Critical-Section Problem**

Consider system of n processes {p0, p1, … pn-1}

Each process has critical section segment of code. Process may be changing common variables, updating table, writing file, etc .When one process in critical section, no other may be in its critical section Critical section problem is to design protocol to solve this .Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section .Especially challenging with preemptive kernels

General structure of process pi is



**Solution to Critical-Section Problem**

. **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 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

**2.Semaphores:**

* Synchronization tool that does not require busy waiting
* Semaphore *S* – integer variable
* Two standard operations modify S: wait() and signal()
	+ Originally called P() andV()
* Less complicated
* Can only be accessed via two indivisible (atomic) operations

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 1

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

* 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

**Semaphore Implementation with no Busy waiting**

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
	+ Solved via **priority-inheritance protocol**

**3.Classical Problems of Synchronization**

* Classical problems used to test newly-proposed synchronization schemes
	+ 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
* Problem – allow multiple readers to read at the same time
	+ Only one single writer can access the shared data at the same time
	+ Several variations of how readers and writers are treated – all involve priorities
* 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**



* Philosophers spend their lives thinking and eating
* Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
	+ Need both to eat, then release both when done
* In the case of 5 philosophers
	+ 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);

**Monitors**

* A high-level abstraction that provides a convenient and effective mechanism for process synchronization
* *Abstract data type*, internal variables only accessible by code within the procedure
* Only one process may be active within the monitor at a time
* But not powerful enough to model some synchronization schemes

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 until x.signal ()
	+ x.signal () – resumes one of processes (if any) that invoked x.wait ()
		- If no x.wait () on the variable, then it has no effect on the variable

**Monitor with Condition Variables**



**Deadlocks**

**Deadlock: -**A deadlock consists of a set of blocked processes, each holding a resource and waiting to acquire a resource held by another process in the set.

**Example: - 1.**

****

* Traffic only in one direction.
* The resource is a one-lane bridge.
* If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)‏.
* Several cars may have to be backed up if a deadlock occurs.
* Starvation is possible.

**Example: -**

* + A system has 2 disk drives
	+ *P*1 and *P*2 each hold one disk drive and each needs the other one

**Conditions for resource deadlock: -**

* Deadlock can arise if four conditions hold simultaneously.
1. **Mutual exclusion:** only one process at a time can use a resource
2. **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
3. **No preemption:** a resource can be released only voluntarily by the process holding it after that process has completed its task
4. **Circular wait:** there exists a set {*P*0, *P*1, …, *P*0} of waiting processes such that *P*0 is waiting for a resource that is held by *P*1, *P*1 is waiting for a resource that is held by
	* 1. *P*2, …, *Pn*–1 is waiting for a resource that is held by
		*P*n, and *P*n is waiting for a resource that is held by *P*0

**Methods for Handling Deadlocks:-**

* **Prevention**
	+ Ensure that the system will *never* enter a deadlock state
* **Avoidance**
	+ Ensure that the system will *never* enter an unsafe state
* **Detection**
	+ Allow the system to enter a deadlock state and then recover
* **Do Nothing**
	+ Ignore the problem and let the user or system administrator respond to the problem; used by most operating systems, including Windows and UNIX

**Resource - Allocation Graph: -**

A set of vertices *V* and a set of edges *E*.

* + 1. V is partitioned into two types:
		- P={P1,P2,------Pn}, the set consisting of all the processes in the

System.

* + - R= {R1, R2, ----Rm}, the set consisting of all resource types in the

System.

* + 1. Request edge - directed edge P1->Rj.
		2. Assignment edge – directed edge Rj-> Pj.



Before P3 requested an instance of R2 After P3 requested an instance of R2



**Graph with A Cycle But No Deadlock: -**

Process P4 may release its instance of resource type R2. That resource

can then be allocated to P3, thereby breaking the cycle.

**Relationship of cycles to deadlocks:-**

* If a resource allocation graph contains no cycles Þ no deadlock.
* If a resource allocation graph contains a cycle and if only one

 instance exists per resource type Þ deadlock.

* If a resource allocation graph contains a cycle and if several instances exists per resource type Þ possibility of deadlock.

**Deadlock Prevention: -**

Heavener in his pioneering work showed that since all four of the conditions are necessary for deadlock to occur, it follows that deadlock might be prevented by denying any one of the conditions.

* **Elimination of “Mutual Exclusion” Condition**The mutual exclusion condition must hold for non-sharable resources. That is, several processes cannot simultaneously share a single resource. This condition is difficult to eliminate because some resources, such as the tap drive and printer, are inherently non-shareable. Note that shareable resources like read-only-file do not require mutually exclusive access and thus cannot be involved in deadlock.

* **Elimination of “Hold and Wait” Condition**There are two possibilities for elimination of the second condition. The first alternative is that a process request be granted all of the resources it needs at once, prior to execution. The second alternative is to disallow a process from requesting resources whenever it has previously allocated resources. This strategy requires that all of the resources a process will need must be requested at once. The system must grant resources on “all or none” basis. If the complete set of resources needed by a process is not currently available, then the process must wait until the complete set is available. While the process waits, however, it may not hold any resources. Thus the “wait for” condition is denied and deadlocks simply cannot occur. This strategy can lead to serious waste of resources. For example, a program requiring ten tap drives must request and receive all ten derives before it begins executing. If the program needs only one tap drive to begin execution and then does not need the remaining tap drives for several hours. Then substantial computer resources (9 tape drives) will sit idle for several hours. This strategy can cause indefinite postponement (starvation). Since not all the required resources may become available at once.

* **Elimination of “No-preemption” Condition**The non-preemption condition can be alleviated by forcing a process waiting for a resource that cannot immediately be allocated to relinquish all of its currently held resources, so that other processes may use them to finish. Suppose a system does allow processes to hold resources while requesting additional resources. Consider what happens when a request cannot be satisfied. A process holds resources a second process may need in order to proceed while second process may hold the resources needed by the first process. This is a deadlock. This strategy require that when a process that is holding some resources is denied a request for additional resources. The process must release its held resources and, if necessary, request them again together with additional resources. Implementation of this strategy denies the “no-preemptive” condition effectively.
**High Cost**   When a process release resources the process may lose all its work to that point. One serious consequence of this strategy is the possibility of indefinite postponement (starvation). A process might be held off indefinitely as it repeatedly requests and releases the same resources.

* **Elimination of “Circular Wait” Condition**The last condition, the circular wait, can be denied by imposing a total ordering on all of the resource types and then forcing, all processes to request the resources in order (increasing or decreasing). This strategy impose a total ordering of all resources types, and to require that each process requests resources in a numerical order (increasing or decreasing) of enumeration. With this rule, the resource allocation graph can never have a cycle.
For example, provide a global numbering of all the resources, as shown

|  |  |  |
| --- | --- | --- |
| 1 | ≡ | Card reader |
| 2 | ≡ | Printer |
| 3 | ≡ | Plotter |
| 4 | ≡ | Tape drive |
| 5 | ≡ | Card punch |

Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request first printer and then a tape drive (order: 2, 4), but it may not request first a plotter and then a printer (order: 3, 2). the problem with this strategy is that it may be impossible to find an ordering that satisfies everyone.

**Deadlock Avoidance: -**

This requires that the system has some information available up front. Each process declares the maximum number of resources of each type which it may need. Dynamically examine the resource allocation state to ensure that there can never be a circular-wait condition.

The system's resource-allocation state is defined by the number of available and allocated resources, and the maximum possible demands of the processes. When a process requests an available resource, the system must decide if immediate allocation leaves the system in a *safe state*.

The system is in a safe state if there exists a safe sequence of all processes:

Sequence < P1, P2, --- Pn > is safe for the current allocation state if, for each P*i*, the resources which P*i* can still request can be satisfied by

* The currently available resources plus
* The resources held by all of the P*j*'s, where *j* < *i*.

If the system is in a safe state, there can be no deadlock. If the system is in an unsafe state, there is the *possibility* of deadlock.

**Banker’s algorithm for deadlock avoidance: -**

Banker’s algorithm is a deadlock avoidance algorithm. It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not.

Consider there are n account holders in a bank and the sum of the money in all of their accounts is S. Every time a loan has to be granted by the bank, it subtracts the loan amount from the total money the bank has. Then it checks if that difference is greater than S. It is done because, only then, the bank would have enough money even if all the n account holders draw all their money at once.

Banker’s algorithm works in a similar way in computers. Whenever a new process is created, it must exactly specify the maximum instances of each resource type that it needs.

Let us assume that there are **n** processes and **m** resource types. Some data structures are used to implement the banker’s algorithm. They are:

* **Available:** It is an array of length **m**. It represents the number of available resources of each type. If **Available[j] = k**, then there are **k** instances available, of resource type **Rj**.
* **Max:** It is an **n x m** matrix which represents the maximum number of instances of each resource that a process can request. If **Max[i][j] = k**, then the process **Pi** can request at most **k** instances of resource type **Rj**.
* **Allocation:** It is an **n x m** matrix which represents the number of resources of each type currently allocated to each process. If **Allocation[i][j] = k**, then process **Pi** is currently allocated **k** instances of resource type **Rj**.
* **Need:** It is an **n x m** matrix which indicates the remaining resource needs of each process. If **Need[i][j] = k**, then process **Pi** may need **k** more instances of resource type **Rj** to complete its task.

Need[i][j] = Max[i][j] - Allocation [i][j]

#### Safety Algorithm:

****

**Example: -**

****

****

****