**Process Synchronization**

**Introduction**

* Concurrent access to shared data may result data inconsistency.
* Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

**Critical - Section Problem**

* The key to preventing trouble involving shared storage is find some way to prevent more than one process from reading and writing the shared data simultaneously.
* That part of the program where the shared memory is accessed is called the *Critical Section*.
* To avoid race condition, one must identify codes in Critical Sections  for each thread.

**Race Conditions**

* In operating systems, processes that are working together share some common storage (main memory, file etc.) that each process can read and write.
* When two or more processes are reading or writing some shared data simultaneously, the Race conditions will occur.
* Concurrently executing threads that share data need to synchronize their operations and processing in order to avoid race condition on shared data.
* Only one thread at a time should be allowed to examine and update the shared variable (Mutual Exclusion)
* The interrupts must be disabled to prevent another interrupt before the first one completes.
* The important feature of operating system is that, when one process is executing in its **critical section**, no other process is to be allowed to execute in it. That is, no two processes are executing in their critical sections at the same time.
* The Critical – Section problem is used 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**.

Do {

Entry section

Critical section

Exit section

Remainder section

} while (TRUE);

**Example**

|  |  |
| --- | --- |
| Producer:  while (count==Buffer\_Size);  do nothing  add an item to the buffer  ++count;  Buffer[in]=item;  in=(in +1)%Buffer\_Size; | Consumer:  while (count==0);  do nothing;  remove an item from buffer  --count;  Item = Buffer[out];  out=(out+1)%Buffer\_Size) |

**Properties of a Critical Section Code:**

* Codes that alter one or more variables that are possibly being referenced in “read- update-write” fashion by another thread.
* Codes alter any part of a data structure while it is possibly in use by another thread.

**Requirements for Solution Critical section Problem:**

1. **Mutual Exclusion**: if process P1 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 some process wish to enter their critical sections, then only those process that are not executing in their remainder sections can participate in the decision on which will enter its critical section next , and this selection cannot be postponed indefinitely.
3. **Bounded waiting**: there exists abound , or limit , 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

**Unsynchronized Producer-Consumer Problem:**

public class UnsynchronizedBuffer implements Buffer

{

private int buffer = -1; // shared by producer and consumer

public void set( int value ) throws Interrupted Exception

{

buffer = value;

} // end method set

public int **get**() throws InterruptedException

{

return buffer;

} // end method get

} // end class UnsynchronizedBuffer

**Solution for Critical – Section Problem**

**Peterson’s Solution**

It is a classic software –based solution, is restricted to two process that alternate execution between their critical sections and remainder sections .The processes are numbered P0 and P1 . The two processes share two variables:

int turn ;

Boolean flag[ 2];

The variable (**turn**) indicates whose turn 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 Pi is ready.

**Algorithm for Process Pi**

while (true)

{

Flag[i] = TRUE;

turn = j;

While (flag [ j ] && turn == j );

// critical section

Flag[i] = FALSE;

// remainder section

}

**Prove Solution is Correct**

* Property 1: each Pi enters its critical section only if either flag[j] == FALSE or turn == i.
* Properties 2 and 3: Process Pi can be prevented from entering the critical section only if it is stack in the while loop with the condition.
* flag[ j ] == TRUE and turn == j; this loop is the only one possible.
* If Pj is not ready to enter the critical section, then flag[ j ] == FALSE , and Pi can enter its critical section .

**Critical – Section Problems using Locks**

* Any solution to the critical – section problem requires a simple tool – a **Lock.**
* Race conditions are prevented by requiring that critical regions be protected by locks
* Process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section.

**Solution to the Critical-Section problem using Locks**

do

{

Acquire lock;

// critical section

Release lock

// remainder section

} while ( TRUE ) ;

**Synchronized Producer-Consumer Problem**

public synchronized void set( int value ) throws InterruptedException

{

while ( occupied )

{

wait();

} // end while

buffer = value; // set new buffer value

occupied = true;

notifyAll();

}

// end method set; releases lock on Synchronized Buffer

public synchronized int get() throws InterruptedException

{

while ( !occupied )

{

wait();

} // end while

occupied = false;

notifyAll();

Return buffer;

//selects all threads or processes in wait set and moves them to entry set

}

// end method get; releases lock on Synchronized Buffer

**Semaphore**

* Synchronization tool that does not *require* busy waiting.
* A semaphore *S* is a protected integer-valued variable.
* Two standard operations modify semaphore S:

acquire( ) and release( ) or wait( ) and signal( )

* + Busy-waiting implementation of these indivisible (atomic) operations:

wait( ){

while ( value <= 0); // empty loop body (no-op)

value--;}

signal( ){ value++; }

**What is a semaphore?**

Semaphore as general It is Synchronization Tool

A semaphore is hardware or a software tag variable whose value indicates the status of a common resource. Its purpose is to lock the resource being used. A process which needs the resource will check the semaphore for determining the status of the resource followed by the decision for proceeding. In multitasking operating systems, the activities are synchronized by using the semaphore techniques.

There are 2 types of semaphores:

* Counting semaphore – integer value can range over ordinary domain, it may have value to be greater than one, typically used to allocate resources from a pool of identical resources.
* Binary semaphore – integer value can range only between 0 and 1, also known as mutex locks :

Semaphore S = new semaphore ( );

S. wait( ) ; wait( ){ while ( value <= 0); // empty loop body (no-op) value--;}

// critical section

S. signal( ) ; signal( ){ value++; }

// remainder section;

**Semaphore Implementation**

* Must guarantee that no two processes can execute acquire ( ) and release ( ) on the same semaphore at the same time.
* Implementation of acquire (wait) and release (signal) could now have busy waiting in critical section implementation.
* Solving this by modify the definition of wait ( ) and release ( ) semaphore operations.
* When a process executes the wait ( ) operation and finds that the semaphore value is not positive, it must wait.
* The process blocks itself.

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 );

}

}

**Problems with Semaphores**

* + Correct use of semaphore operations may not be easy:
  + Suppose semaphore variable called mutex is initialized to 1.
  + What happens in the following?
* signal (mutex) … CS … wait (mutex)
* wait (mutex) … CS … wait (mutex)
  + The following will happen:
* Using wait and signal in a wrong sequence 🡪 synchronization problems.
* Omitting wait or signal 🡪 synchronization problems.

**What is the mutex?**

Mutex is the short form for ‘Mutual Exclusion object’. A mutex allows multiple threads for sharing the same resource. The resource can be file. A mutex with a unique name is created at the time of starting a program. A mutex must be locked from other threads, when any thread that needs the resource. When the data is no longer used / needed, the mutex is set to unlock.