DATE:

Chapter 8: Process Synchronization & Communication.

2075 Bhadra

Q. What is critical section problem? Why must the executing the critical section be mutually exclusive? Describe how semaphores can be used to solve the critical section problem?

Ans: Critical section is the part of a program which tries to access shared resources. That resource may be any resource in a computer like memory location, data structure, CPU or any IO device. The critical section cannot be executed by more then one process at the same time; operating system faces the difficulty in allowing and disallowing the processes from ~~entering~~ entering the critical section.

The critical section problem is used to design a set of protocols which can ensure that the Race condition among the processes will never arise.

Mutual exclusion means if a process is executing in its critical section, then no other process is allowed to execute in the critical section. Since, shared resources are kept in the critical section and if more than one process ~~is~~ access the critical section, the alteration caused by one may cause error in the execution of another. So, executing the critical section must be mutually exclusive.

```
int s = 1;
wait( Semaphore s)
{
    while (s == 0);
```

```
    s = s - 1;
}
signal (Semaphore s)
{
    s = s + 1;
}
```

Implementation of semaphore:

```
do
{
    wait(s); // critical section.
    signal(s); // remainder section.
} while (1);
```

Let there be two process P1 and P2 and a semaphore initialised as 1. Now if suppose P1 enters in its critical section then it will wait unit $s > 0$, this call only happen when P1 finishes its critical section and calls signal() operation on semaphore s. This way mutual exclusion is achieved and critical section problem is solved.

2074 Bhadra

Q. What is race condition? Explain how Sleep and Wakeup() solution is better than busy waiting solution for critical section problem.

Ans. When two or more processes are reading or writing some shared data and the final results depends on who runs precisely when, are called race condition.

In busy waiting solution, processes waiting to enter their critical sections waste processor time

checking to see if they can proceed. Busy waiting has the following disadvantages

- waste of processor time
- Possibility of deadlock/ starvation in systems with multipriority scheduling.

Sleep() and wakeup() eliminates this problem. By calling sleep() the calling process is suspended till being woken by other process called wakeup()

Q. What is TSL? Why it is used? Explain the major operations of semaphore with a simple implementation as a class.

Ans:

Test and Set lock (TSL) is a synchronization mechanism. It uses a test and set instruction to provide the synchronization among the processes executing concurrently.

It is used to implement mutual exclusion.

The major operations of semaphore are:

- wait(): called when a process wants access to a resource.
- signal(): called when a process is done using a resource.

```
class semaphore {
public:
    semaphore (int n): n-(n) {
    }
    void wait() {
        --n;
    }
```

DATE:

```
void signal() {
        ++n;
    }
private:
    std::atomic<int> n_;
};
```

2073 Bhadra

Q. Explain critical section problem. Why is it important for a thread to execute a critical section as quickly as possible?

Ans: A thread must acquire a lock prior to executing a critical section. The lock can be acquired by only one thread. So, in order to release the lock, so that the other thread could access that critical section, a thread must execute a critical section as quickly as possible.

Q. Define Semaphore and explain the major operations in semaphore including pseudocode.

Ans: Semaphore is a variable used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment. The major operations of a semaphore are:

- wait()
- signal()

Pseudocode.

```
int s = 1;
wait (Semaphore s)
{
```

```
    while (s==0);
    s=s-1;
}
signal (Semaphore s)
{
    s=s+1;
}
```

2073 Magh:

Q. What are the requirements of mutual exclusion? Solve producer - consumer problem using semaphore and message passing.

Ans. Requirements of mutual exclusion

1. No two processes may be simultaneously inside their critical region.
2. No assumption may be made about speeds or number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

Solution of Consumer-Producer problem using semaphore.

```
#define N 100
typedef int semaphore
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
void producer (void)
{
```

```
int item;
while (TRUE)
{
    item = produce_item();
    down(&empty);
    down(&mutex);
    insert_item(item);
    up(&mutex);
    up(&full);
}
}
void consumer(void)
{
    int item;
    while (TRUE)
    {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consumer_item(item);
    }
}.
```

2072 Ashwin:

Q. Solve producer - consumer problem using monitors.

Ans Monitors make solving the producer - consumer a little easier. Mutual exclusion is achieved by placing the critical section of a program inside a monitor. In the code below, the critical sections of

DATE:

the producer and consumer are inside the monitor, Producer Consumer. Once inside the monitor, a process is blocked by the Wait and Signal primitives if it cannot continue.

```
monitor ProducerConsumer
    condition full, empty;
    int count;
producer    enter();
{
    if (count == N)
```

```
monitor ProducerConsumer
{
    int itemCount = 0;
    condition full;
    condition empty;
    procedure add (item)
    {
        if (itemCount == BUFFER_SIZE)
        {
            wait (full);
        }
        putItemIntoBuffer (item);
        itemCount = itemCount + 1;
        if (itemCount == 1)
        {
            notify (empty);
        }
    }
```

```
procedure remove( )
{
    if (itemCount == 0)
    {
        wait(empty);
    }
    item = removeItemFromBuffer();
    itemCount = itemCount - 1;
    if (itemCount == BUFFER_SIZE - 1)
    {
        notify(full);
    }
    return item;
}
}

procedure producer( )
{
    while (true)
    {
        item = produceItem();
        ProducerConsumer.add(item);
    }
}

procedure consumer()
{
    while (true)
    {
        item = ProducerConsumer.remove()
        consumeItem(item);
    }
}
```

PAGE NO.:
DATE:

2072 Magh:

Q. Why processes need to be synchronized ? Explain Peterson's Solution and TSL instruction approaches used in mutual exclusion with busy waiting.

Ans:

Process synchronization means sharing system resources by processes in a such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Processes need to be synchronized for sharing of resources without interference using mutual exclusion.

Peterson's algorithm is a concurrent programming algorithm. It is used for mutual exclusion and allows two processes to share a single-use resource without conflict. It is given as.

```
#define FALSE 0
#define TRUE 1
#define N 2
int turn;
int interested[N];
void enter_region(int process);
{
    int other;
    other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while (turn == process && interested[other] == TRUE);
}
void leave_region(int process)
{
```

interested [process] - FALSE;
}

Before using the shared variables (i.e. before entering its critical region), each process calls enter_region with its own process number, 0 or 1, as parameter. This call will cause to wait, if needed be, until it is safe to enter. After it has finished with the shared variables, the process calls leave_region to indicate that it is done and to allow the other process to enter, if it so desires.

TSL Instruction:

```
enter region:
    TSL REGISTER LOCK
    CMP REGISTER, #0
    JNE enter region
    RET
leave region:
    MOVE LOCK, #0
    RET.
```

To use the TSL instruction, we will use a shared variable, lock to coordinate access to shared memory. When lock is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory. When it is done, the process set lock back to 0 using an ordinary move instruction.

The first instruction copies the old value of lock to the register and then set lock to 1. Then the old value is compared with 0. If it is nonzero,

the lock was already set, so the program just goes back to the beginning and tests it again. Sooner or later it will become 0 (when the process currently in its critical region is done with its critical region), and the subroutine returns, with the lock set. Clearing the lock is very simple. The program just stores a 0 in lock. No special synchronization instructions are needed.

2071 Magh

Q. Why do we need pipe() function? Define Semaphore and explain the major operations in semaphore. Can semaphores be used in distributed system? Explain why or why not.

Ans: We need pipe() function in order to create a connection between two processes, such that the standard output from one process becomes the standard input of the other ~~sys~~ process. Pipes are useful for communication between related processes (inter-process communication). Pipe can be used by the creating process, as well as all its child processes, for reading and writing.

Yes, semaphores can be used in distributed systems. A distributed system is a ~~new~~ network of processors interconnected by a communication network. The processors do not share memory and exchange information through messages. The lack of shared memory makes implementation of semaphores very difficult in a distributed system. A

implementation of semaphores in such a system must rely on message passing. To protect a critical section in a distributed system, several algorithms based on message passing have been proposed. Those algorithms indirectly implement binary semaphore in distributed system.

2070 Bhadra

Q. Explain all possible approaches to handle the situation "while one process is busy updating shared memory, no other process will enter its critical section and cause trouble".

Ans. The various approaches to handle the situation are:

i) Disabling Interrupts:

On a single-processor system, the simplest solution is to have each process disable all interrupts just after entering its critical region and renable them just before leaving it. With interrupts disabled, no clock interrups can occur. The CPU is only switched from process to process as a result of clock or other interrupts, after all, and with interrupts turned off the CPU will not be switched to another process.

This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts.

ii) Lock variables:

This approach is based on software. Consider having a single, shared (lock) variable, initially 0. When a process want to enter its critical region, it first test the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no

process in its critical region, and a 1 means that some process is in its critical region.

iii) Strict Alternation.

Two Process Solution:

```
P0: #define FALSE 0
    #define TRUE 1
    while (TRUE)
    {
        while(turn != 0);
        critical_region();
        turn = 1;
        noncritical_region();
    }
P1: #define FALSE 0
    #define TRUE 1
    while (TRUE)
    {
        while (turn != 1);
        critical_region();
        turn = 0;
        noncritical_region();
    }
```

A proposed solution to the critical region problem for Process '0' and Process '1'.

The integer variable turn, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory. Initially, process 0 inspects turn, finds it to be 0 and enters its critical region. Process 1 also finds it to be 0 and therefore sits in a tight loop continually

testing turn to see when it becomes 1.

iv) Peterson's Solution:

Peterson's algorithm is a concurrent programming algorithm for mutual exclusion that allows two or more processes to share a single-use resource without conflict, using only shared memory for communication.

v) TSL Instruction:

Test and Set Lock (TSL) is a synchronisation mechanism. It uses a test and set instruction to provide the synchronization among the process executing concurrently.

It is an instruction that returns the old value of a memory location and sets the memory location value to 1 as a single atomic operation. If one process is currently executing a test-and-set, no other process is allowed to begin another test-and-set until the first process test-and-set is finished.