USN : ______________________________

CMR Institute of Technology, Bangalore
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
II - INTERNAL ASSESSMENT

Semester: 4-CBCS 2018	Date: 22 Jun 2021
Subject: OPERATING SYSTEMS (18CS43)	
Faculty: Ms Savitha S	
Time: 01:00 PM - 02:30 PM	Max Marks: 50

ANSWER_ANY 5 Question(s)

Marks CO BT/CL

1a. What is true about thread? A. Thread switching does not need to interact with operating system. B. All threads can share same set of open files, child processes. C. Multiple threaded processes use fewer resources. D. All of the above

Ans:D

[1.0] 1 [2]

1b. A thread is also called ? A. heavyweight process. B. lightweight process. C. data segment process. D. overhead process

Ans: B

[1.0] 1 [2]

1c. Which of the following is not an advantage about thread? A. Threads minimize the context switching time. B. Use of threads provides concurrency within a process. C. kernel is single threaded D. All of the above

Ans:C

[1.0] 1 [2]

1d. Which of the following is true about kernal level thread? A. Implementation is by a thread library at the user level. B. Kernel-level threads are slower to create and manage. C. Multi-threaded applications cannot take advantage of multiprocessing. D. Both B and C

Ans:B

[1.0] 1 [2]

1e. Multithreading models are of_________ types?A. 2 B. 3 C. 4 D. 5

Ans B

[1.0] 1 [2]

1f. The kernel is _______ of user threads. A. a part of B. the creator of C. unaware of D.aware of

Ans C

[1.0] 1 [2]

1g. Multithreading on a multi : CPU machine ___________ A. decreases concurrency B. increases concurrency C. doesn't affect the concurrency D. can increase or decrease the concurrency

Ans: B

[1.0] 1 [3]

1h. The OS maintains all PCBs in? A. Process Scheduling Queues B. Job queue C. Ready queue D. Device queues

Ans:A

[1.0] 1 [2]

1i. The processes which are blocked due to unavailability of an I/O device constitute this queue. A. Process Scheduling Queues B. Job queue. C.Ready queue D. Device queues

Ans:D

[1.0] 1 [2]

1j. Which is not a type of Schedulers? A. Long-Term Scheduler B. Short-Term Scheduler C. Medium-Term Scheduler D. None of the above

Ans:D

[1.0] 1 [2]

2a. When the suspended process is moved to the secondary storage. This process is called? A. process mix. B. Swapping C. Swap-In D. Swap-**Out**

Ans:**B**

[1.0] 1 [2]

2b. A_________ is the mechanism to store and restore the state A. PCB B. Program Counter C. Scheduling information D. context switch

Ans: C

[1.0] 1 [2]

2c. Which of the following is false regarding First Come First Serve (FCFS)? A. FCFS performance is high B. average wait time is high. C. Its implementation is based on FIFO queue. D. FCFS is easy to understand and implement.

Ans:A

[1.0] 1 [3]

2d. Which of the following is non-preemptive algorithm? A. Priority Based Scheduling B. Shortest Remaining Time C. FCFS D. shortest job first

Ans:A

[1.0] 1 [3]

2e. In Round Robin Scheduling, Each process is provided a fix time to execute, it is called a? A. Batch Time B. Job Time C. Quantum D. Period

Ans:C

[1.0] 1 [2]

2f. Which of the following is preemptive algorithm? A. Round Robin Scheduling B. Shortest Remaining Time C. Multiple-Level Queues Scheduling D. Both A and B

Ans:D

[1.0] 1 [2]

2g. When several processes access the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called? A.Dynamic condition B.Race condition C.Essential condition D.Critical condition

Ans:**B**

[1.0] 1 [3]

2h. If a process is executing in its critical section, then no other processes can be executing in their critical section. This condition is called? A.Mutual exclusion B.Critical exclusion C.Synchronous exclusion D.Asynchronous exclusion

Ans:A

[1.0] 1 [2]

2i. Which of the following for Mutual exclusion can be provided by the: A.mutex locks B.binary semaphores C.both mutex locks and binary semaphores D.none of the mentioned

Ans:C

[1.0] 1 [3]

2j. Peterson's solution is restricted to ___ processes that alternate execution between their critical sections and remainder sections? A.one B.Two C.Three D.Four

Ans:**B**

[1.0] 1 [10]

3. Explain multithreaded models with diagrams

Support for threads may be provided ateither
1. The user level, for **user threads** or
2. By the kernel, for **kernel threads**.

User-threads are supported above the kernel and are managed without kernel support. Kernel- threads are supported and managed directly by the OS.

Three ways of establishing relationship between user-threads &kernel-threads:

1. Many-to-one model
2. One-to-one model and
3. Many-to-many model.

Many-to-One Model

Many user-level threads are mapped to one kernel thread.

Advantages:

- Thread management is done by the thread library in user space, so it is efficient.

Disadvantages:

- The entire process will block if a thread makes a blocking system-call.
- Multiple threads are unable to run in parallel on multiprocessors.

For example:

- Solaris green threads
- GNU portable threads.

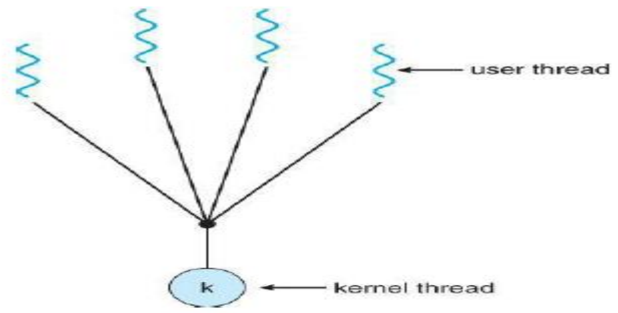


Fig: Many-to-one model

One-to-One Model

Each user thread is mapped to a kernel thread.

Advantages:

- It provides more concurrency by allowing another thread to run when a thread makes a blocking system-call.
- Multiple threads can run in parallel on multiprocessors.

Disadvantage:

- Creating a user thread requires creating the corresponding kernel thread.

For example:

- Windows NT/XP/2000, Linux

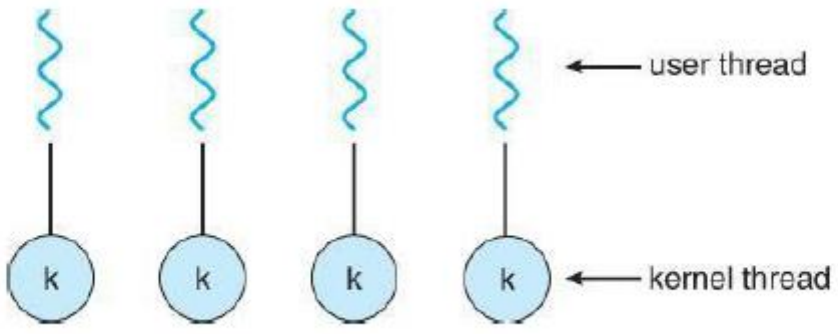


Fig: one-to-one model

Many-to-Many Model

 Many user-level threads are multiplexed to a smaller number of kernel threads.

Advantages:

▪ Developers can create as many user threads as necessary

▪ The kernel threads can run in parallel on a multiprocessor.

▪ When a thread performs a blocking system-call, kernel can schedule another thread for execution.

Two Level Model
A variation on the many-to-many model is the two level-model

Similar to M:N, except that it allows a user thread to be bound to kernelthread.

forexample:
▪ HP-UX

▪ Tru64 UNIX

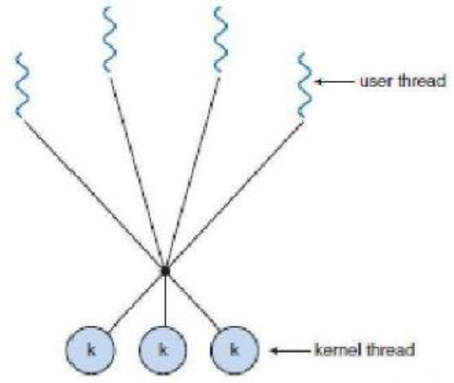


Fig: Many-to-many model

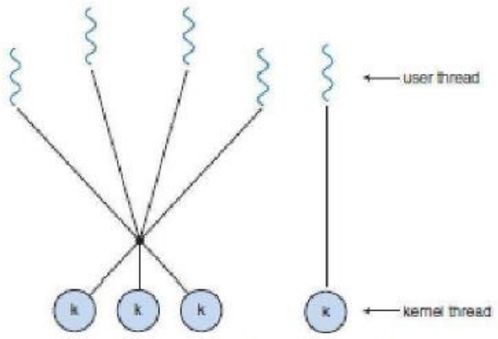


Fig: Two-level model

4. Illustrate petersons solution for critical section problem [10]

A classic software-based solution to the critical-section problem known as **Peterson's solution**. It addresses the requirements of mutual exclusion, progress, and bounded waiting.
It Is two process solution.

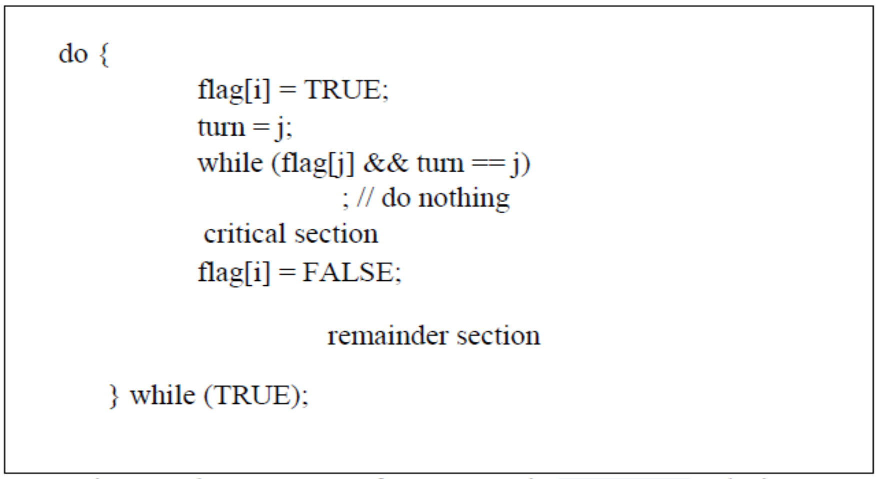
Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted

The two processes share two variables:

int turn;

Boolean flag[2]

turn: The variable turn indicates whose turn it is to enter its critical section. **Ex:** if turn == i, then process P_i is allowed to execute in its critical section
flag: The flag array is used to indicate if a process is ready to enter its critical section. **Ex:** if flag [i] is true, this value indicates that P_i is ready to enter its critical section.



```
do {
            flag[i] = TRUE;
            turn = j;
            while (flag[j] && turn == j)
                        ; // do nothing
             critical section
            flag[i] = FALSE;

                        remainder section

    } while (TRUE);
```

Figure: The structure of process P_i in Peterson's solution

:

To enter the critical section, process Pi first sets flag [i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so.

If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last, the other will occur but will be over written immediately.

The eventual value of turn determines which of the two processes is allowed to enter its critical sectionfirst
To prove that solution is correct, then we need to show that
1. Mutual exclusion ispreserved

2. Progress requirement is satisfied

3. Bounded-waiting requirement is met

1. To prove Mutual exclusion
Each pi enters its critical section only if either flag [j] == false or turn ==i.

If both processes can be executing in their critical sections at the same time, then flag [0] == flag [1]==true.

These two observations imply that Pi and Pj could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence, one of the processes (Pj) must have successfully executed the while statement, whereas Pi had to execute at least one additional statement ("turn==j").

However, at that time, flag [j] == true and turn == j, and this condition will persist as long as Pi is in its critical section, as a result, mutual exclusion is preserved.

2. To prove Progress and Bounded-waiting
A process Pi can be prevented from entering the critical section only if it is stuck 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.

If Pj has set flag [j] = true and is also executing in its while statement, then either turn === i or turn ===j.

- If turn == i, then Pi will enter the critical section.
- If turn== j, then Pj will enter the critical section.

However, once Pj exits its critical section, it will reset flag [j] = false, allowing Pi to enter its critical section.

If Pj resets flag [j] to true, it must also set turn to i.

Thus, since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (progress) after at most one entry by Pj (bounded waiting).

[10.0] 1 [3]

5. Illustrate how readers writers problem can be solved using semaphores

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.

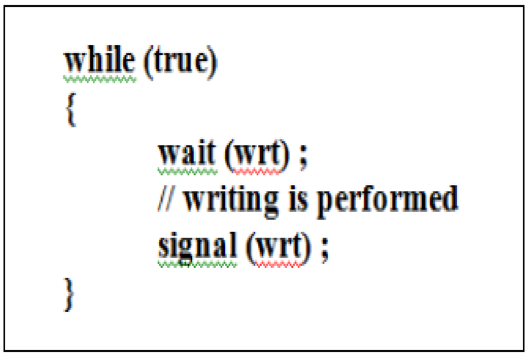
Shared Data

Dataset

Semaphore **mutex** initialized to 1.

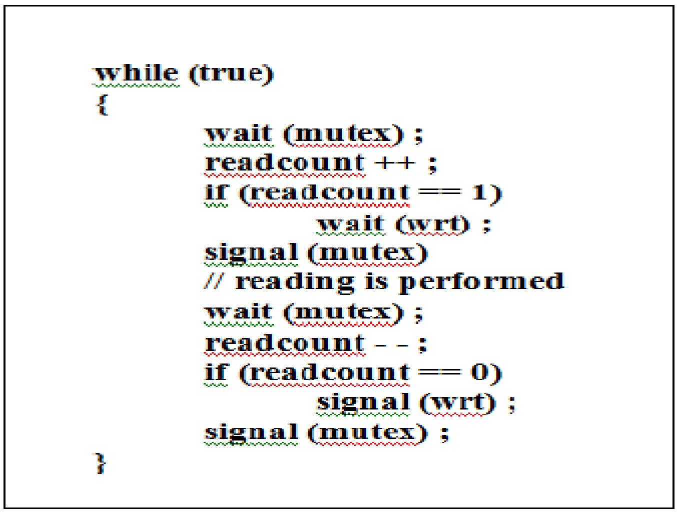
Semaphore **wrt** initialized to1.

Integer **readcount** initialized to 0.



```
while (true)
{
        wait (wrt) ;
        // writing is performed
        signal (wrt) ;
}
```

The structure of a writerprocess



```
while (true)
{
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
                wait (wrt) ;
        signal (mutex)
        // reading is performed
        wait (mutex) ;
        readcount - - ;
        if (readcount == 0)
                signal (wrt) ;
        signal (mutex) ;
}
```

The structure of a readerprocess

[10.0] 1 [3]

7. What is busy waiting in critical section problem? how semaphores are used to solve critical section? What are the advantages of semaphores

Solution for Busy Waiting problem:
Modify the definition of the wait() and signal()operations as follows: When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. Rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.
A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup() operation, which

the process from the waiting state to the ready state. The process is then placed in the ready queue.
To implement semaphores under this definition, define a semaphore as follows:
typedef struct
 { int value;
struct process *list;
} semaphore;
Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the

list of waiting processes and awakens that process. Now, the wait() semaphore operation can be defined as:

```
wait(semaphore *S) {
S->value--;
if (S->value < 0) {
add this process to S->list; block();
}
}
```

and the signal() semaphore operation can be defined as

```
signal(semaphore *S) {
S->value++;
if (S->value <= 0) {
remove a process P from S->list; wakeup(P);
}
}
```

The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P.

[10.0] 1 [3]