OPERATING SYSTEM
IT-41033

CHAPTER – 6
Q&A

6.2

There is a few reasons why disabling interrupts are poor choice of implementation of synchronization primitives in multicore/multiprocessor systems. If interrupts are disabled for all cores the following can occur:

Priority – If something urgent occurs kernel might need an interrupt. This cannot happen since every interrupts are disabled and kernel could not handle urgent action.

Lack of interrupts – OS could not use context-switching. I/O operations or any other operations which would require a trap in kernel.

Clock lag – If clock relies on interrupts it could start lagging

Early exit – If a user apps exits due to an exception or error and it doesn’t free/release synchronization primitives it could leave interrupts disabled.

6.3

Mutual exclusion – First while loop will last as long as the process is not the only process marked with flag waiting. After that it will flag itself as active and it will check that it is the only process marked as active. This will loop until condition is satisfied. After that if there were no other active processes, and if we have the turn or else whoever has it is idle, then proceed. Otherwise, repeat the whole sequence.

Progress – When a process/thread decides to enter critical section the decision will be made.

Waiting – When a process/thread does want to enter critical region, it will eventually enter the section as algorithm sets previous "occupant" of critical section to idle state and the turn variable "cycles".

All of the conditions are satisfied. In finite amount of time or steps thread/process which enters is decided, no process will forever await entry and only one process at the time can enter.

6.10

These operating systems provide different locking mechanisms depending on the application developers needs. Spinlocks are useful for multiprocessor systems where a thread can run in a busy loop rather than incurring the overhead of being put in a sleep queue. *Mutexes* are useful for locking resources. Solaris uses adaptive *mutexes*, meaning that *mutex* is implemented with a spinlock on multiprocessor machines. Semaphores and condition variables are more appropriate tools for synchronization when a resource must be held for a long period of time, since spinning is inefficient for a long duration.

6.16 Describe how volatile, nonvolatile, and stable storage differ in cost.

- Volatile storage refers to main and cache memory and is very fast. Volatile storage cannot survive system crashes or powering down the system.
- Non-volatile storage survives system crashes and powered-down systems. Disks an tapes are examples of non-volatile storage. USB devices using erasable program read-only memory (EPROM) have appeared providing non-volatile storage.
- Stable storage refers to storage that technically can never be lost as there are redundant backup copies of the data on disk.

6.19

Every kernel needs to track all active processes. This can be done via a circular double linked-list that is linux kernel. When a process is created it should be inserted in that list. OS needs to do next is following:

- Find a process which will be previous preceding process for new one that is process A.
- Process B is next succeeding process A. Then, it obtain B with A => next
- Process A => next should now point to the new process N, while N => previous which was not assigned previously should point to A.
- Process B => previous should point to the new process N, while N => next which was not assigned previously should point to B

A race condition could occur if previously listed items are not protected. For example, while one process was created and it wanted to insert itself to a position, it may have unlinked A and B, and correctly linked itself to A and B. If this is not synchronized, it could be overwritten by a second process which was created before first new process inserted itself in position.

Similarly to circular double linked list, kernel uses a tree structure. Which uses pointers and concepts of children. As those concepts have previous(parent) and next(child), this situation could repeat itself.

6.31

A checkpoint log record indicates that a log record and its modified data has been written to stable storage and that the transaction need not to be redone in case of a system crash.

System performance when no failure occurs – If no failures occur, the system must incur the cost of performing checkpoints that are unnecessary. Performing checkpoints less often will lead to better system performance.

The time it takes to recover from a system crash – The existence of a checkpoint record means that an operation will not have to be redone during system recovery. The more checkpoints were performed, the faster the recovery time is from a system crash.

The time it takes to recover from a disk crash – The existence of a checkpoint record means that n operation will not have to be redone during system recovery. The more checkpoints were performed, the faster the recovery time is from a disk crash.

Q. Define the atomic transaction.

A transaction is a series of read and write operations upon some data followed by a commit operation. If the series of operations in a transaction cannot be completed, the transaction must be aborted and the operations that did take place must be rolled back. It is important that the series of operations in a transaction appear as one indivisible operation to ensure the integrity of the data being updated.

Q. In consideration bounded buffer, suppose to modify the algorithm, write the code for procedure, consumer process that can be modified.

```
while (true) {
     /* produce an item in nextProduced */
     while (counter == BUFFER_SIZE)
        ; /* do nothing */
     buffer[in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

Producer process

```
while (true) {
     while (counter == 0)
        ; /* do nothing */
     nextConsumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;
     /* consume the item in nextConsumed */
}
```

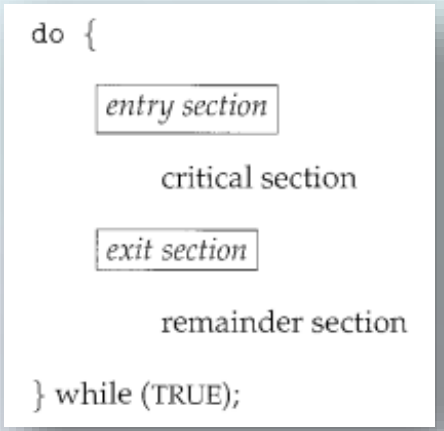
Consumer process

Q. What is the three requirements of solution to critical-section problem for satisfactions.

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 some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. **Bounded waiting:** There exists a bound, 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.



General structure of a typical process.

Q. Describe Peterson's solution and write codes for the structure of process P_i and P_j.

It is a software-based solution critical section problem.

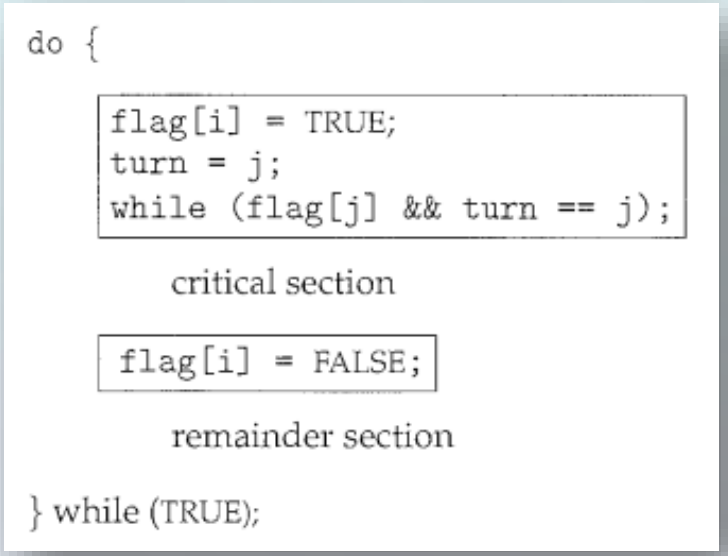
Two processes Pi and Pj need to access shared data safety, ensuring only one process is in its' critical section at a time.

They alternate between the critical section and a remainder section where they do other things.

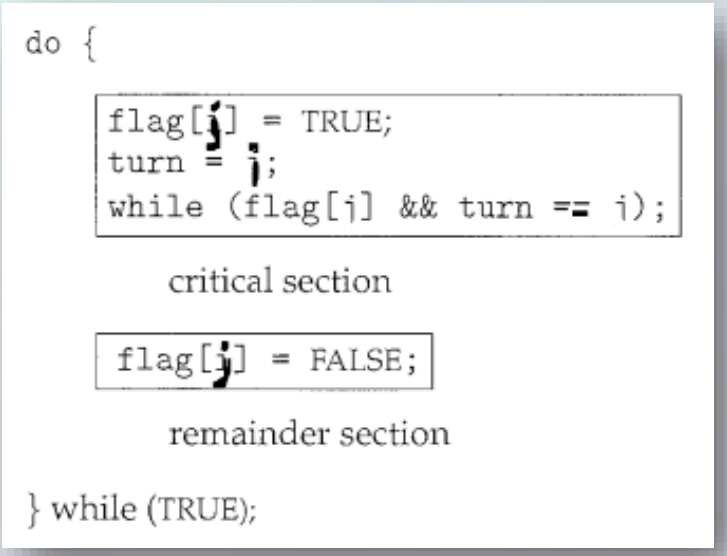
The processes P_i and P_j require two data items to be shared between two processes.

They are following:

- **int turn** => indicates whose turn it is to enter its critical section
- **boolean flag[2]** => use to indicate if a process is ready to enter its critical section



Structure of P_i in Peterson's solution



Structure of P_j in Peterson's solution

Q. Write codes for the definition of the *TestAndSet( )* instruction and mutual-exclusion implementation.

```
boolean TestAndSet(boolean *target) {
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```

Process

```
do {
   while (TestAndSet(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
} while (TRUE);
```

Q. Define deadlock.

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event is the execution of a *signal( )* operation. When such a state is reached, these processes are said to be deadlocked.

To consider a system consisting of two processes, P_0 and P_1, each accessing two semaphores, S and Q, set to the value 1:

P_0 executes *wait(S)* and P_1 executes *wait(Q)*.

When P_0 executes *wait(Q)*, it must wait until P_1 executes *signal(Q)*.

When P1 executes wait(S), it must wait until P_0 executes *signal(S)*.

These *signal( )* operations cannot be executed, P_0 and P_1 are deadlocked.

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
.	.
.	.
.	.
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Q. Write codes for the data structure of a reader and writer process.

In the solution to the readers-writers problem, the reader processes shared the following data structures: *semaphore mutex, wrt;*

int readcount;

The semaphores *mutex* and *wrt* are initialized to 1; *readcount* is initialized to 0. The semaphore *wrt* is common to both reader and writer processes. The *mutex* semaphore is used to ensure mutual exclusion when the variable *readcount* is updated. The *readcount* variable keeps track of how many processes are currently reading the object.

Q. Write codes for the data structure of a reader and writer process. (Cont…)

```
do {
   wait(wrt);
      . . .
   // writing is performed
      . . .
   signal(wrt);
} while (TRUE);
```

Writer 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);
```

Reader process

Q. Describe the several possible remedies to the deadlock to consider dining philosophers problem.

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosophers to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution, that is an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

Q. Explain and write code for dining philosophers solution using Monitors.

The solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To code this solution, it is needed to distinguish among three states in which may find a philosopher.

enum {THINKING, HUNGRY, EATING} state[5];

Philosopher i can set the variable state[i] = EATING only if her two neighbors are not eating:

state [(i+4) % 5] != EATING and state[(i+1) % 5] != EATING.

declaration their condition themselves: condition self[5];

Q. Explain and write code for dining philosophers solution using Monitors. (Cont…)

```
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((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;
    }
}
```

Dining philosophers solution using Monitors