Q1: What are the five major activities of an operating system in regard to process management?

Five major process management activities of an operating system are
a. creation and deletion of user and system processes.
b. suspension and resumption of processes.
c. provision of mechanisms for process synchronization.
d. provision of mechanisms for process communication.
e. provision of mechanisms for deadlock handling.

Q2: What are the three major activities of an operating system in regard to memory management?

Three major memory management activities of an operating system are
a. keeping track of which parts of memory are currently being used, and by whom.
b. deciding which processes are to be loaded into memory when memory space becomes available.
c. allocating and freeing memory space as needed.

Q3: What are the three major activities of an operating system in regard to secondary-storage management?

Three major secondary-storage management activities of an operating system are free-space management, storage allocation, disk scheduling.

1. What are the two main functions of an operating system?

An operating system must provide the user with an extended (i.e. virtual) machine, and it must manage the I/O devices and other system resources.

2. What is the difference between timesharing and multiprogramming systems?

Multiprogramming is the rapid switching of the CPU between multiple processes in memory. It is commonly used to keep the CPU busy while one or more processes are doing I/O.

3. What is multiprogramming?

Multiprogramming is the rapid switching of the CPU between multiple processes in memory. It is commonly used to keep the CPU busy while one or more processes are doing I/O.

4. What is spooling? Do you think that advanced personal computer will have spooling as a standard feature in the future?

Input spooling is the technique of reading in jobs, for example, from cards, onto the disk, so that when the currently executing processes are finished, there will be work waiting for the CPU. Output spooling consists of first copying printable files to disk before printing them, rather than printing directly as the output is generated. Input spooling on a personal computer is not very likely, but output spooling is.

5. On early computers, every byte of data read or written was handled by the CPU (i.e., there was no DMA). What implications does this have for multiprogramming?

The prime reason for multiprogramming is to give the CPU something to do while waiting for I/O to complete. If there is no DMA, the CPU is fully occupied doing I/O, so there is nothing to be gained (at least in terms of CPU utilization) by multiprogramming. No matter how much I/O a program does, the CPU will be 100% busy. This of course assumes the major delay is the wait while data are copied. A CPU could

do other work if the I/O were slow for other reasons (arriving on a serial line, for instance).

6. Why was timesharing not widespread on second-generation computers?

Second generation computers did not have the necessary hardware to protect the operating system from malicious user programs.

7. The family of computers idea was introduced in the 1960s with the IBM System/360 mainframes. Is this idea now dead as a doornail or does it live on?

It is still alive. For example, Intel makes Pentium I, II, and III, and 4 CPUs with a variety of different properties including speed and power consumption. All of these machines are architecturally compatible. They differ only in price and performance, which is the essence of the family idea.

8. Consider a system that has two CPUs and each CPU has two threads (hyperthreading). Suppose three programs, P0, PI, and P2, are started with run times of 5, 10 and 20mses, respectively. How long will it take to complete the execution of these programs? Assume that all three programs are 100% CPU bound, do not block during execution, and do not change CPUs once assigned.

It may take 20, 25 or 30 msec to complete the execution of these programs depending on how the operating system schedules them. If P0 and P1 are scheduled on the same CPU and P2 is scheduled on the other CPU, it will take 20 msec. If P0 and P2 are scheduled on the same CPU and P1 is scheduled on the other CPU, it will take 25 msec. If P1 and P2 are scheduled on the same CPU and P0 is scheduled on the other CPU,

it will take 30 msec. If all three are on the same CPU, it will take 35 msec.

9. One reason GUIs were initially slow to be adopted was the cost of the hardware needed to support them. How much video RAM is needed to support a 25 line x 80 row character monochrome text screen? How much for a 1024 x 768 pixel 24-bit color bitmap? What was the cost of this RAM at 1980 prices ($5/KB)? How much is it now?

A 25 × 80 character monochrome text screen requires a 2000-byte buffer. The 1024 × 768 pixel 24-bit color bitmap requires 2,359,296 bytes. In 1980 these two options would have cost $10 and $11,520, respectively. For current prices, check on how much RAM currently costs, probably less than $1/MB.

10. There are several design goals in building an operating system, for example, resource utilization, timeliness, robustness, and so on. Give an example of two design goals that may contradict one another.

Consider fairness and real time. Fairness requires that each process be allocated its resources in a fair way, with no process getting more than its fair share. On the other hand, real time requires that resources be allocated based on the times when different processes must complete their execution. A real time process may get a disproportionate share of the resources.

11. A computer has a pipeline with four stages. Each stage takes the same time to do its work, namely 1 nsec. How many instructions per second can this machine execute?

Every nanosecond one instruction emerges from the pipeline. This means the machine is executing 1 billion instructions per second. It

does not matter at all how many stages the pipeline has. A 10-stage pipeline with 1 nsec per stage would also execute 1 billion instructions per second. All that matters is how often a finished instruction pops out the end of the pipeline.

12. Which of the following instructions should be allowed only in kernel mode?

(a) Disable all interrupts.
(b) Read the time-of-day clock.
(c) Set the time-of-day clock.
(d) Change the memory map.

Choices (a), (c), and (d) should be restricted to kernel mode.

13. List some differences between personal computer operating systems and mainframe operating systems.

Personal computer systems are always interactive, often with only a single user. Mainframe systems nearly always emphasize batch or timesharing with many users, protection is much more of an issue on mainframe systems, as is efficient use of all resources.

14. Consider a computer system that has cache memory, main memory (RAM) and disk, and the operating system uses virtual memory. It takes 2 μsec to access a word from the cache, 10 nsec to access a word from the RAM, and 10 ms to access a word from the disk. If the cache hit rate is 95% and main memory hit rate (after a cache miss) is 99%, what is the average time to access a word?

Average access time = 0.95 × 2 nsec (word is cache)
+ 0.05 × 0.99 × 10 nsec (word is in RAM, but not in cache)

$$+ 0.05 \times 0.01 \times 10{,}000{,}000 \text{ nsec (word on disk only)}$$
$$= 5002.395 \text{ nsec} = 5.002395 \ \mu\text{sec}$$

15. What is a trap instruction? Explain its use in operating systems.

A trap instruction switches the execution mode of a CPU from the user mode to the kernel mode. This instruction allows a user program to invoke functions in the operating system kernel.

16. What is the key difference between a trap and an interrupt?

A trap is caused by the program and is synchronous with it. If the program is run again and again, the trap will always occur at exactly the same position in the instruction stream. An interrupt is caused by an external event and its timing is not reproducible.

17. When a user program makes a system call to read or write a disk file, it provides an indication of which file it wants, a pointer to the data buffer, and the count. Control is then transferred to the operating system, which calls the appropriate driver. Suppose that the driver starts the disk and terminates until an interrupt occurs. In the case of reading from the disk, obviously the caller will have to be blocked (because there are no data for it). What about the case of writing to the disk? Need the caller be blocking awaiting completion of the disk transfer?

Maybe. If the caller gets control back and immediately overwrites the data, when the write finally occurs, the wrong data will be written. However, if the driver first copies the data to a private buffer before returning, then the caller can be allowed to continue immediately. Another possibility is to allow the caller to continue and give it a signal when the buffer may be reused, but this is tricky and error prone.

18. Why is the process table needed in a timesharing system? Is it also needed in personal computer systems in which only one process exists, that process taking over the entire machine until it is finished?

The process table is needed to store the state of a process that is currently suspended, either ready or blocked. It is not needed in a single process system because the single process is never suspended.

19. Is there any reason why you might want to mount a file system on a nonempty directory? If so, what is it?

Mounting a file system makes any files already in the mount point directory inaccessible, so mount points are normally empty. However, a system administrator might want to copy some of the most important files normally located in the mounted directory to the mount point so they could be found in their normal path in an emergency when the mounted device was being checked or repaired.

20- What is the purpose of a system call in an operating system?

A system call allows a user process to access and execute operating system functions inside the kernel. User programs use system calls to invoke operating system services.

21. What is the essential difference between a block special file and a character special file?

Block special files consist of numbered blocks, each of which can be read or written independently of all the other ones. It is possible to seek to any block and start reading or writing. This is not possible with character special files.

22. In the example given in Fig. 1-17, the library procedure is called

read and the system call itself is called read. Is it essential that both of these have the same name? If not, which one is more important?

System calls do not really have names, other than in a documentation sense. When the library procedure read traps to the kernel, it puts the number of the system call in a register or on the stack. This number is used to index into a table. There is really no name used anywhere. On the other hand, the name of the library procedure is very important, since that is what appears in the program.

23. The client-server model is popular in distributed systems. Can it also be used in a single-computer system?

Yes it can, especially if the kernel is a message-passing system.

24. To a programmer, a system call looks like any other call to a library procedure. Is it important that a programmer know which library procedures result in system calls? Under what circumstances and why?

As far as program logic is concerned it does not matter whether a call to a library procedure results in a system call. But if performance is an issue, if a task can be accomplished without a system call the program will run faster. Every system call involves overhead time in switching from the user context to the kernel context. Furthermore, on a multiuser system the operating system may schedule another process to run when a system call completes, further slowing the progress in real time of a calling process.

25. Figure 1-23 shows that a number of UNIX system calls have no Win32 API equivalents. For each of the calls listed as having no Win32 equivalent, what are the consequences for a programmer of converting a UNIX program to run under Windows?

Several UNIX calls have no counterpart in the Win32 API:

Link: a Win32 program cannot refer to a file by an alternate name or see it in more than one directory. Also, attempting to create a link is a convenient way to test for and create a lock on a file.

Mount and umount: a Windows program cannot make assumptions about standard path names because on systems with multiple disk drives the drive name part of the path may be different.

Chmod: Windows programmers have to assume that every user can access every file.

Kill: Windows programmers cannot kill a misbehaving program that is not cooperating.

26. A portable operating system is one that can be ported from one system architecture to another without any modification. Explain why it is infeasible to build an operating system that is completely portable. Describe two high-level layers that you will have in designing an operating system that is highly portable.

Every system architecture has its own set of instructions that it can execute. Thus a Pentium cannot execute SPARC programs and a SPARC cannot execute Pentium programs. Also, different architectures differ in bus architecture used (such as VME, ISA, PCI, MCA, SBus, ...) as well as the word size of the CPU (usually 32 or 64 bit). Because of these differences in hardware, it is not feasible to build an operating system that is completely portable. A highly portable operating system will consist of two high-level layers---a machinedependent layer and a machine independent layer. The machine-dependent layer addresses the specifics of the hardware, and must be implemented separately for every architecture. This layer provides a uniform interface on which the machine-independent layer is built. The machine-independent layer has

to be implemented only once. To be highly portable, the size of the machinedependent layer must be kept as small as possible.

27. Explain how separation of policy and mechanism aids in building microkernel-based operating systems.

Separation of policy and mechanism allows OS designers to implement a small number of basic primitives in the kernel. These primitives are simplified, because they are not dependent of any specific policy. They can then be used to implement more complex mechanisms and policies at the user level.

28. Here are some questions for practicing unit conversions:
(a) How long is a microyear in seconds?
(b) Micrometers are often called microns. How long is a gigamicron?
(c) How many bytes are there in a I-TB memory?
(d) The mass of the earth is 6000 yottagrams. What is that in kilograms?

(a) A micro year is $10^{-6} \times 365 \times 24 \times 3600 = 31.536$ sec.
(b) 1000 meters or 1 km.
(c) There are 2^{40} bytes, which is 1,099,511,627,776 bytes.
(d) It is 6×10^{24} kg.

29. An alert reviewer notices a consistent spelling error in the manuscript of an operating systems textbook that is about to go to press. The book has approximately 700 pages, each with 50 lines of 80 characters each. How long will it take to electronically scan the text for the case of the master copy being in each of the levels of memory of Fig. 1-7? For internal storage methods, consider that the access time given in per character, for disk devices assume the time is per block of 1024 characters, and for tape assume the time given is to the start of the data with subsequent access at the same speed as disk access.

The manuscript contains 80 × 50 × 700 = 2.8 million characters. This is, of course, impossible to fit into the registers of any currently available CPU and is too big for a 1-MB cache, but if such hardware were available, the manuscript could be scanned in 2.8 msec from the registers or 5.8 msec from the cache. There are approximately 2700 1024-byte blocks of data, so scanning from the disk would require about 27 seconds, and from tape 2 minutes 7 seconds. Of course, these times are just to read the data. Processing and rewriting the data would increase the time.

30. Can the; count = write(fd, buffer, nbytes); Call return any value in count other than nbytes? If so, why?

If the call fails, for example because fd is incorrect, it can return −1. It can also fail because the disk is full and it is not possible to write the number of bytes requested. On a correct termination, it always returns nbytes.

31. For each of the following system calls, give a condition that causes it to fail: fork, exec, and unlink.

Fork can fail if there are no free slots left in the process table (and possibly if there is no memory or swap space left). Exec can fail if the file name given does not exist or is not a valid executable file. Unlink can fail if the file to be unlinked does not exist or the calling process does not have the authority to unlink it.

32. A file whose file descriptor is fd contains the following sequence of bytes: 3,1,4,1,5,9,2,6,5,3,5. The following system calls are made: lseek(fd, 3, SEEK_SET); read(fd, &buffer, 4);

where the lseek call makes a seek to byte 3 of the file. What does buffer contain after the read has completed?

It contains the bytes: 1, 5, 9, 2.

33. What is the difference between kernel and user mode? Explain how having two distinct modes aids in designing an operating system.

Operating System's were designed with two different major operative mode.

Kernel Mode:
In the kernel mode running codes can access to hardware directly and here we can see any CPU instruction and reference can put any memory address. I mean in this mode the codes run directly hardware with no restriction, it can be put and access memory directly. The Kernel mode is safety place of the operating system that hides important things from the users who wants to damage the system, operating system seperate kernel programs and user programs in the different modes.For instance, when we install a hardware driver(software) use kernel mode. In addition, Kernel mode crashes will be hard to solve it is not recoverable.

User Mode:
In the user mode running codes can not access to hardware directly(access thanks to system calls) and can not put any referance to memory directly. We can say it is normal mode that we use the programs like internet explorer, media player etc, in this mode.Most of the codes running, the computer works in this mode.It is not access directly any important place to Operating System so if there is a problem in here, it can be recoverable, only that application crash, not entire system.

I explained something what is kernel and user mode. They are enforced from CPU hardware and they are extremely important for operating system safety, because of the user can not use kernel mode that includes critical data structures operates machine, direct hardware and direct memory accessing, if there is any harm to system, the main system will be halt.Thanks to this distinct the user can use the user mode and operating system avoid to meet any harm. Because, the user can not access any important place.

Q4: What are the five major activities of an operating system in regard to file management?

Five major file management activities of an operating system are
a. creation and deletion of files.
b. creation and deletion of directories.
c. support of primitives for manipulation files and directories.
d. mapping of files and directories onto secondary storage.
e. backup of files on stable (nonvolatile) storage media.

Q5: What is the purpose of the command interpreter? Why is it usually separate from the kernel?

The command interpreter is the interface between the user and the operating system. Its function is simple: to get the next command statement and execute it.

Because that some operating systems include the command interpreter in the kernel.

Q6: List five services provided by an operating system. Explain how each provides convenience to the users. Explain in which cases it would be impossible for user-level programs to provide these services.

Five services that are provided by an operating system are
a. program execution. The OS provides for the loading of programs into memory and managing the execution of programs. The allocation of CPU time and memory could not be properly managed by the users' programs.

b. I/O operation. User-level programs should not be allowed to perform I/O operations directly for reasons of efficiency and security. Further, a simple I/O request by a user-level program typically entails a long sequence of machine language and device-control commands.

c. file-system manipulation. This service takes care of the housekeeping and management chores associated with creating, deleting, naming, and protecting files. User programs should not be allowed to perform these tasks for security reasons.

d. communications. The OS takes care of the complex subtasks involved in exchanging information between computer systems. It is neither desirable (because of the number and complexity of the lower-level subtasks involved) nor advisable (for security reasons) to let user-level programs handle communications directly.

e. error detection. This service includes the detection of certain errors caused by hardware failure (faults in memory or I/O devices) or by defective/malicious software. A prime directive for a computer system is the correct execution of programs, hence the responsibility of this service cannot be shifted to user-level programs.

Q7: What is the purpose of the system calls?

System calls provide the interface between processes and the operating system. These calls are usually available both from assembly-language programs and from higher-level language programs.

Q8: What are the main types of system calls? Describe their purpose.

Process control:

end, abort; load, execute; create process, terminate process; get process attributes, set process attributes; wait for time; wait event, signal event; allocate and free memory

File management:
Create file, delete file; open, close; read, write, reposition; get file attributes, set file attributes

Device management:
Request device, release device; read, write, reposition; get device attributes, set device attributes; Logically attach or detach devices

Information maintenance:
Get time or date, set time or date; get system data, set system data; get process, file, or device attributes; set process, file, or device attributes

Communications:
Create, delete communication connection; send, receive messages; transfer status information; Attach or detach remote devices

Q9: A total of how many processes would be created in the function fork test () given below, excluding the main process that called fork test()? How many times will "Whatsup" be printed. Briefly explain your answer.

```
void fork_test( ) {
    if (fork() != 0) {
        if (fork() != 0)
            fork();
        else
            printf("Hello 1 \n");
    } else {
        if (fork() != 0) {
            printf("Hello \n");
        } else {
            fork();
        }
    }
```

```
printf("Whatsup                                                    \n");
}
```

Five process create in this process excluding the fork test (). Whatsup line prints six times in this coding because each process print this line. fork test () method contain (fork()!=0) condition that is false in case of child process because child process fork() return 0 value. This condition is true for parent process because it return non zero value. If it return -1 then it contain error. First we consider parent process. It enter to this method and then check again condition that is false for child process and true for parent process. Parent process again call fork() method and make child process. Second (fork()!=0) condition that is false for child process run else portion of if condition and print hello line. Now we consider child process of 1st condition (fork()!=0) that is false for child process so we come to the else portion. In the else portion there is another (fork()!=0) condition. That is true for parent so parent process print hello line and this condition false for child process so it come to else portion and call fork() method and make child process. At the end each process print Whatsup line. As we know that is 6 process so it print 6 times Whatsup line

Q10: Describe what problems would happen when multiple threads will execute the following C statement; also justify your answer with proper reasons. { static int I; I++; }

1. Incorrect ordering: When two thread increment the counter but the result is 1 instead 2.

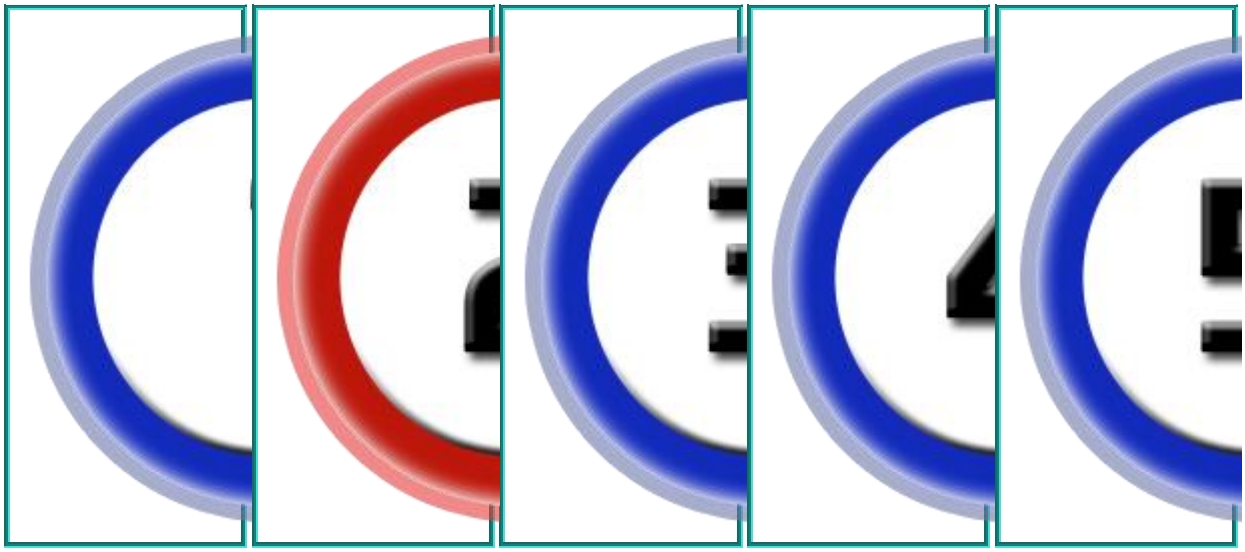
2. Concurrence issues: when executing static methods concurrently are static fields, which only exist one time. So, if the method reads and writes static fields, concurrence issues can occur In concurrent environment two or many threads can run concurrently, if all threads are trying to update some shared resource concurrently, there might be chance that threads trying to read shared resource get stale data and produce wrong outputs.

3. Deadlock problem: A deadlock occurs when each of two threads tries to lock a resource the other has already locked. Neither thread can make any further progress. Deadlock occur when two of many thread are waiting for two or more resources, where each thread need to get access on all resources to progress and

those resources are acquired by different threads and waiting for other resources to be release, which will not be possible ever. For example, two threads A and B need both resources X and Y to perform their tasks. If thread A acquires resource X and thread B acquires Y and now both threads are waiting for resources to be release by other thread. Which is not possible and this is called deadlock. You need to take care of the way of your synchronization if this is going to be a cause of deadlock.

4. Race condition problem: A race condition is a bug that occurs when the outcome of a program depends on which of two or more threads reaches a particular block of code first. Running the program many times produces different results, and the result of any given run cannot be predicted. In a multithreaded application, a thread that has loaded and incremented the value might be preempted by another thread which performs all three steps; when the first thread resumes execution and stores its value, it overwrites objCt without taking into account the fact that the value has changed in the interim. When writing multi-threaded applications, one of the most common problems experienced are race conditions. A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data A "race condition" exists when multithreaded (or otherwise parallel) code that would access a shared resource could do so in such as way as to cause unexpected results.

5. Visibility problem: Visibility of shared object could be different for different threads. This is one of the problems of multi-threading environment. Suppose there are two threads, one is writing any shared variable and other is reading from shared variable. When reading and writing will be occur in different threads, there is no guarantee that reader thread will see the value written by writer thread. This problem of multi-threading is known as visibility issue of shared resource. This problem can be resolved with proper synchronization of shared resource.



Q11: Reader/writer locks are specialized locks used to solve the readers/writers problem. Consider the following pseudo-code implementation of reader-writer locks which is a variant of the readers/writers solution discussed in lectures but implemented via semaphores. Note that readers must call the function AcquireReadLock before reading the data while writers must call AcquireWriteLock before modifying or updating the data. Once data access has been completed, the locks must be released by calling the ReleaseReadLock() and ReleaseWriteLock() functions respectively :

(a) Briefly explain why AcquireReadLock() and AcquireWriteLock() functions perform P and V operations on the mutex semaphore in the above code?

(b) Briefly explain whether readers or writers could be starved due to this implementation?

(c) Suggest a mechanism through which a no starvation policy could be implemented. In other words, suggest in words, how would you modify the code such that a starvation-free implementation results.

(a) Both AcquireReadLock() and AcquireWriteLock() functions perform P and V operations for mutual exclusion on mutex semaphore. As we know P operation on a semaphore means decrementing its value and V operation on a semaphore means

incrementing its value . P() make mutex value zero so that no one can enter in critical section. And when reader or writer out from the critical section then they perform v() on mutex and makes its value 1 so that other reader or writer can access.

(b) Yes, reader and writer both can face starvation. Reader starvation In ReleaseWriteLock() function When writer out it give priority to waiting writers as:

```
if(WAITINGWRITERS>0)                                    {
                                                    V(OkToWrite);
                                                ACTIVEWRITERS++;
                                                WAITINGWRITERS--;
}
```

So reader starvation occur.
Writer starvation AcquireReadLock() function when if condition will be false then else part execute and priority will be given to waiting reader. In other word when active writer not equal to zero because there are some writer then it is necessary to allow writer to access but in the given code priority is given to waiting reader.

```
if((ACTIVEWRITERS==0)                                   {
                                                    V(OkToRead);
                                                ACTIVEREADERS++;
}                                   else                {
                                                WAITINGREADERS++;
}
```

So writer starvation occur.

(c) To remove writer starvation we need to modify if condition in the AcquireReadLock() function as: if(ACTIVEWRITERS+WATINGWRITERS==0).

To remove reader starvation we need to modify the code in a way when fix number of writer access the database after this reader can access. Or when waiting readers increase from their fix limit then reader can access.

Q12: Review the Readers/Writers problem discussed in lecture 12, write the code for Reader() and Writer() functions, when readers are given priority over writers,

keeping the problem constraints in mind.

```
Reader () {
    lock.Acquire();
    while (AW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    } // while
    AR++;
    lock.Release();
    Access DB
    lock.Acquire();
    AR--;
    If (WR > 0) {
            okToRead.Broadcast(&lock);
    } else if (AR == 0 & WW > 0) {
    okToWrite.Signal(&lock);
    }
    lock.Release();
}
```

```
Writer () {
    lock.Acquire();
    while ((AR + WR + AW) > 0) {
            WW++;
            okToWrite.Wait(&lock);
            WW--;
    } // while
    AW++;
    lock.Release();
    Access DB
    lock.Acquire();
    AW--;
    If (WR > 0)
            okToRead.Broadcast(&lock);
    else if (WW > 0)
            okToWrite.Signal(&lock)
    lock.Release(); }
}
```

Q13: Which are the necessary conditions to hold the deadlocks?

There are four necessary conditions and all four of them must hold simultaneously for a deadlock to occur. There conditions are Mutual Exclusion, Hold & Wait, No preemption and Circular Wait.

I. Mutual Exclusion means that there must be at least one resource which can only be used by a single process at a time i.e. non-shareable mode. If any other process request for this resource, the requesting process must be delayed until the requested resource is released by the process already using it.

II. Hold and Wait means that a process is holding at least one resource and waiting for other resource(s) which is/are held by other process(es).

III. No preemption means that none of the processes can be forced to release the resources which are being held by them i.e. the processes will released the

resources voluntarily.

IV. Circular Wait. There must be a set of processes which are waiting for resources which are being held by the other process. E.g. P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, and so on upto Pn, and Pn is waiting for a resource held by P1.

Q14: Draw a resource graph to show the deadlock condition.

Following resource graph shows the deadlock condition. Here are three process P1, P2, P3 and four different Resources R1, R2, R3 and R4 each having 1, 1, 2 and 3 instances respectively (shown by filled circles inside resource rectangles).

Process requesting for an instance of resource is represented by directed edge from Process Pi to Resource Rj. Similarly, directed edge from resource instance towards any Process Pi represents that that particular instance of Resource Rj is allocated to Process Pi.

The graph shows that there are two cycles in this graph i.e. P1 → R1, R1 → P2, P2 → R2, R2 →P3, P3 →R3, R3 →P1 and P2 →R2, R2 →P3, P3 →R3, R3 →P2

None of the processes will release the resources that it has already acquired and all of these three will remain in deadlock state.

Q15: How is it possible to prevent the deadlocks?

There are four necessary conditions which must hold simultaneously for a deadlock to occur. So, deadlock can be prevented by implementing some methodology that guarantees that at least one of the four necessary conditions must not hold. By implementing this kind of strategy, it is possible to prevent a deadlock from occurring.

Relaxing first three conditions i.e. Mutual exclusion, Hold & wait, and no pre-emption are not always useful. There may be low utilization of resources & starvation.

Circular Wait is the ultimate choice. This is the most suitable condition i.e. ensuring that circular wait never holds. Simple methodology used for this purpose is order all the resources and each process always requests resources in increasing order. Hence, there may never be a process waiting for a low numbered resource holding a higher numbered resource, ensuring that occurrence of cycle is impossible. So, implementing that there will be no circular wait, deadlocks can be prevented.

Q16: Why it is considered that threads are harmful?

Threads are a difficult thing to handle for most programmers. Bugs can be introduced easily by the programmers while using threads and the most annoying this is that these are hard to find and even harder to fix. The things are painful for experts, so inexperienced programmers consider threads harmful.

While synchronization, it is required that threads must coordinate access to shared data with proper locks and missing a lock results in corruption of data. Threads also cause deadlocks if used inefficiently, carelessly. Main purpose of usage of threads is concurrency, but this is hard.

Hence threads are considered harmful and are avoided by normal programmers.

Q17: Could you give some reasons to use the threads in a beneficial way?

Threads can be used for concurrency. Responsiveness of applications is increased.

Threads should be used with care to increase responsiveness. Performance and utilization of multi-core, multi CPUs is increased with multiple threads running on multiple CPUs, each thread running in parallel on a different processor. Efficiency of high-end servers is increased.

Threads share memory and resources of the process to which they belong. Code sharing in allows an application to have several different threads of activity, all within the same address space.

Q18: Which are the classical issues related to the threads?

Synchronization & Deadlocks are issues to the threads and the classical problems are:

1. Bounded Buffer Problem
2. Readers and Writers Problem
3. Dining Philosophers Problem

Q19: There are p processes competing for r identical resource units. Each process needs a maximum of m resource units (where m ≤ r).
(a) Under what condition can no deadlock occur?
(b) Give an example of a request sequence leading to deadlock.

(a) Under following condition or rule, no deadlock occurs:

• Don't allocate any more resource to a process if the process is already using some resources. Let that process consume and release those resources first.

• If all the resources are consumed and further request comes, do not put the requesting process go into wait state. Let them consume and release pre-acquired resources first.

(b) Suppose that all the resources have been allocated to P processes or a subset of P processes i.e. P1, P2… Pn.

P1 needs one more resource, but no more resource is available, P1 goes into waiting.

P2 needs one more resource, but no more resource is available, P2 goes into waiting.

P3 needs one more resource, but no more resource is available, P3 goes into waiting.

.

.

Pn needs one more resource, but no more resource is available, Pn goes into waiting.

At this point all the processes are collectively holding all the resources, all the processes are in wait state, no process will release the acquired resources. So the deadlock occurs here.

Q20: Write a resource allocation algorithm which will prevent deadlock.

Following algorithm will p revent the deadlock:
P processes are requesting for r resources. Resource allocation algorithm does the following:

```
If (process P has ever acquired any resources before)
{
        Do not serve the request;
}
Else if (no resource is remaining)
{
        Do not serve the r equest;
}
Else
{
        If (no resource is free)
                                                {
        Wait until all resources r are free;
                                                }
    Grant process P individual access to r resources;
}
```

Q21: How should threads coordinate? And how should they wait for an event or state change?

A thread, while holding the lock, may change the variables constituting the condition expression. It should then call signals to notify a single waiting thread to resume execution or broadcast to wake all waiting threads. Signal and broadcast move waiting threads off the condition variable's queue and queue them on the

associated lock. Only after the signaling/broadcasting thread releases the lock may these threads resume execution. However, it is possible that other threads were previously waiting on the lock, so the waiting threads are not guaranteed that the condition expression is true. Rather, returning from wait is considered a hint that expression may have become true. After resuming, the thread must test the expression again before proceeding. Hence, threads commonly wait within a while loop that tests the condition expression

Q22: What are conditional critical regions? What are their limitations? And how can we overcome these limitations.

Conditional critical regions are the regions in a program that must be executed distinctly by one process and no other process can execute that region at the same time. When we use condition variables for locking purposes then the critical regions are referred to as conditional critical regions. The limitations of conditional critical region are: • With conditional critical regions, waiting occurs without any changes to shared state. • The resource concept is unreliable • The context switching is inefficient • The scheduling mechanism is too restrictive • These limitations can be removed using conditional variables.

Q23: What are condition variable? What are their limitations? And how can we overcome these limitations?

Condition variable are queues of waiting threads. Condition variables support three operations: wait, signal and broadcast. Logically, every condition variable is associated with one (or more) Boolean condition expressions and a mutex lock. Condition variables are used to complement locks by allowing a program to specify the order of execution. There are two limitations for condition variables: (1) the signal operation is explicit, so a programmer may forget to call signal when changing shared state, (2) condition variables do not nest within multiple levels of mutex locks because only the inner-most lock is released when waiting. As a result, deadlock may result if the blocked thread can only be woken by code that acquires a lock still held by the waiting thread. These limits are removed when we use conditional variables with transactional memory.

Q24: What is lost wakeup problem and how can it be resolved?

Signaling or broadcasting transaction may execute concurrently with a transaction that it woke up. If the waking transaction completes before the signaling transaction, it is called wakeup problem. The waking transaction may view shared state from before the signaling transaction, and hence before the logical condition it awaits is visibly changed. The wakeup problem may be resolved by deferred signal approach, or by speculative signal approach

Q25: What are the requirements and implications for the discussed two implementations of condition variables for transactional memory?

The requirements and implications for deferred signal approach are Commit Actions The requirements and implications for Speculative signal approach are Escape Actions, and Robust Conflicts Detections

Q26: Explain, in your own words, how the proposed approach manages to avoid deadlocks when unstructured locking is used.

The proposed approach says that the deadlock cannot occur when we have the requested lock and its future lock set. The future lockset is the set of locks when the lock is requested and all locks in between till a matching unlock is found. The proposed approach starts by tracking the simple effects. Simple effects may be the empty sequences of locks and unlocks events. Proposed approach uses a continuation effects i.e. the effect of code to its following expression. The proposed approach has the feature to add notes to the lock operation according to their continuation effects.

The continuation effects are calculated statically, but the future lock sets are computed at run time. When a function is called the continuation effects is pushed on to the stack up to the time the call is not fully executed. During the continuation effect, the locks are identified and added to the lock set. At runtime the lock operation unblocks whenever we find a lockset. This assists to avoid deadlocks.

The proposed approach gives only a single lock to every lock operation, this increase the degree of parallelism.

Now let us see constitutes of the proposed approach and its implementation in various programming language constructs.

Effects: Simple effects may be the empty sequences of locks and unlocks events. Continuation effect i.e. the effect of code to its following expression. Notes are added to the lock operation according to their continuation effects. The continuation effects are calculated statically, but the future lock sets are computed at run time. When a function is called the continuation effects is pushed on to the stack up to the time the call is not fully executed. During the continuation effect, the locks are identified and added to the lock set. At runtime the lock operation unblocks whenever we find a lockset. This assists to avoid deadlocks. The proposed approach gives only a single lock to every lock operation, this increase the degree of parallelism. The runtime system utilizes the dynamically computed future locksets so that each lock operation can only proceed when its future lockset is available to the requesting thread.

Function Calls: As we know that the continuation approach is intra-procedural so the unlock operation for a lock operation may not reside in the same function. This problem is resolved by adding the notes to function call according to the continuation effect. As static analysis says ensures that there is always an unlock operation for a lock, so the algorithm traverse the lockset and find the matching unlock terminates.

Conditionals: A demerit defining effects as ordered events is that, in case of conditional expressions/statements, it becomes less likely that the branches will have the same effect. Proposed approach can deal with it. Proposed approach traversed each branch and keep tracks of each branch, and algorithms calculates the lockset for each branch individually.

Loops and recursions: loops and recursions introduce additional problems. In the case of recursion and loops, function and its body name must express the same name. But this is no possible, due to the reason that the two effects cannot be structure-wise equivalent: the effect of a function name is contained in the effect of its body, due to the recursive call. To handle this problem, a summary is tagged to the function names showing the effect of their bodies.

Q27: How static analysis is performed and what are its limitations?

Static analysis is performed in the following four phases:

Pointer Analysis: First, the heap and stack state is formulated at each point of the program by doing a pointer analysis that is based on symbolic expressions. The analysis has been modified to treat the heap allocation in a context-sensitive manner. The out of first stage is a mapping for each expression to a set of abstract locations. Each abstract location may have any forms e.g. a formal parameter, a global variable or a heap-allocated location.

Effect Interference: we have a functional control flow graph in the analysis. Here a forward dataflow algorithm is run on the graph to compute the effect for each function. Nodes of the graph show input, current and output effects. The input effects are formulated from its front edges. The output effect is computed by appending the current effect to the input effect and is propagated to a node's successors until a fixed point is reached. Output effects that have no successors are joined to computer a function's effect.

Loops: at this phase, lock counts effects flowing from back edges must be equivalent to the input effect of the same node. Due to this bound, we can efficiently encode loop effects: from the entrance to exit the counts of each lock should match. We take effect of the entire loop. The empty effect on all branches compensates for the case where a loop is not executed.

Effect Optimization: function effect repetitions should be avoids. For this optimization techniques are applied. One technique is to find the common prefixes and suffixes so that the number of branches may be reduced. Another technique for removing the nested loop is to take the nested loops out. We also trigger the data flow algorithm, which is CPU-intensive, only for functions that are known to contain lock operations, to avoid additional overheads.

Limitations are as follows:

Non C code: C language can be tightly dealt with our static analysis. However, library code cannot be handled by our analysis. In our assumptions, we take the library functions' effect as empty effects. However, we can add user-defined notes

for library functions. There is another limitation with our analysis that it cannot handle non-local jumps (including signals) and inline assembly.

Pointer analysis: our analysis is not capable of handling pointers arithmetic that involves locks in it. Our analysis also fails to track heap allocation at recursive functions and loops. Another limitation is that we need that lock pointers are changed only before they are shared between threads, and that locks pointed by with at least two levels of indirection are not aliased at function calls.

Conditional Execution: if the locks and their matching unlock operation happens to be executed in separate conditional statements, and they have similar guards, in this case, our analysis will not accept the program.

q28: What is are message-oriented models?

Special types of processes communicate among each other with special paths like sockets, ports, or channels etc.
Process communication paths are static. Creating, deleting or changing connection among processes is difficult.
Processes work in their separate spaces and rarely access the shared data

Message-oriented models bear the following characteristics:
1. Process synchronization is made possible through queues that are attached with processes
2. When more than one processes operate on data, the data structure are passed by reference
3. Peripheral devices are also work like messages. Signal is sent to devices and an interrupt is generated by devices in response
4. Priorities for process are set statically at the time of system design
5. Mostly one process is served at a time and served completely before looking into the queue for next process

Messages-oriented models provide the following facilities:
1. It carries a handle for identifying a message, and message data portion
2. Messages communication is established through channels and ports.
3. Messages are sent and received through operations like SendMessage,

AwaitReply, WaitForMessage, and SendReply etc.