Question 1

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");
}
```

Answer:

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

Question 2

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++;
}
```

Answer:

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.

Question 3 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 :

```
Class ReaderWriterLock
{
  Semaphore mutex = 1;        // declaration of a semaphore
  OkToRead = 0;               // a flag to check it is OK to read the database
  OkToWrite = 0;              // a flag to check it it is OK to write the database int
  ACTIVEREADERS=0;            // number of readers holding the read lock and accessing the DB
  WAITINGREADERS=0;           // number of readers waiting to acquire read lock
  ACTIVEWRITERS=0;            // number of writers that have acquired write lock
                                    // (practically this will always be 1)
  WAITINGWRITERS=0;           // number of writers that are waiting for write lock
```

```
void AcquireReadLock() {
        P(mutex); //remember that P operation on a semaphore means decrement value
        if ((ACTIVEWRITERS == 0) {
                V(OkToRead); // remember that V operation on a semaphore
                                means incrementing its value
                ACTIVEREADERS++;
        } else {
                WAITINGREADERS++;
        }
        V(mutex);
        P(OkToRead);
}
void ReleaseReadLock() {
        P(mutex);
        ACTIVEREADERS--;
        if ((ACTIVEREADERS == 0) && (WAITINGWRITERS > 0)) {
                V(OkToWrite);
                ACTIVEWRITERS++;
                WAITINGWRITERS--;
        }
        V(mutex);
}
void AcquireWriteLock() {
        P(mutex);
        If (ACTIVEWRITERS + ACTIVEREADERS == 0) {
                V(OkToWrite);
                ACTIVEWRITERS++;
        } else {
                WAITINGWRITERS++;
        }
        V(mutex);
        P(OkToWrite);
}
void ReleaseWriteLock() {
        P(mutex);
        ACTIVEWRITERS--;
        if (WAITINGWRITERS > 0) {
                V(OkToWrite);
                ACTIVEWRITERS++;
                WAITINGWRITERS--;
        } else {
                while (WAITINGREADERS > 0) {
                        V(OkToRead);
                        ACTIVEREADERS++;
                        WAITINGREADERS--;
                }
        }
        V(mutex);
} } // end of class
```

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

Answer:
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) Briefly explain whether readers or writers could be starved due to this implementation?

Answer:
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) 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.

Answer:

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.

Question 4: 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.

Answer:

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

Question 5: Which are the necessary conditions to hold the deadlocks?

Answer:

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.

Question 6:

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.

Question 7:

How is it possible to prevent the deadlocks?

Answer:

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.

Question 8:

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.

Question 9:

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.

Question 10:

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

Question 11:

Consider the following processes and their related information:

Process	**Arrival Time**	**Burst Time**	**Priority**
P1	0	5	4
P2	2	8	2
P3	4	9	5
P4	6	7	1
P5	8	6	3

Use the appropriate information from the above given. Calculate response time, Waiting Time, and turnaround time and draw the Gant charts for following scheduling algorithms:

1. First come first Serve

P1	P2	P3	P4	P5	
0	5	13	22	29	35

Process	Arrival Time	Burst Time	First CPU Time	**Waiting Time**	**Response Time**	Finishing Time	**Turnaround Time**
P_1	0	5	0	**0**	**0**	5	**5**
P_2	2	8	5	**3**	**3**	13	**11**
P_3	4	9	13	**9**	**9**	22	**18**
P_4	6	7	22	**16**	**16**	29	**23**
P_5	8	6	29	**21**	**21**	35	**27**

2. Round Robin

P1	P2	P3	P4	P5	P1	P2	P3	P4	P5	P1

0	2	4	6	8	10	12	14	16	18	20	21

P2	P3	P4	P5	P2	P3	P4	P3	
21	23	25	27	29	31	33	34	35

Process	Arrival Time	Burst Time	First CPU Time	**Waiting Time**	**Response Time**	Finishing Time	**Turnaround Time**
P_1	0	5	0	**16**	**0**	21	**21**
P_2	2	8	2	**21**	**0**	31	**29**
P_3	4	9	4	**22**	**0**	35	**31**
P_4	6	7	6	**21**	**0**	34	**28**
P_5	8	6	8	**15**	**0**	29	**21**

3. Shortest Job First

P1	P2	P5	P4	P3	
0	5	13	19	26	35

Process	Arrival Time	Burst Time	First CPU Time	**Waiting Time**	**Response Time**	Finishing Time	**Turnaround Time**
P_1	0	5	0	**0**	**0**	5	**5**
P_2	2	8	5	**3**	**3**	13	**11**
P_3	4	9	26	**22**	**22**	35	**31**
P_4	6	7	19	**13**	**13**	26	**20**
P_5	8	6	13	**5**	**5**	19	**11**

4. Priority Scheduling

P1	P2	P4	P2	P5	P1	P3	
0	2	6	13	17	23	26	35

Process	Arrival Time	Burst Time	Priority	First CPU Time	**Waiting Time**	**Response Time**	Finishing Time	**Turnaround Time**
P_1	0	5	4	0	**21**	**0**	26	**26**
P_2	2	8	2	2	**7**	**0**	17	**15**
P_3	4	9	5	26	**22**	**22**	35	**31**
P_4	6	7	1	6	**0**	**0**	13	**7**
P_5	8	6	3	17	**9**	**9**	23	**15**

Hints:

1. The low numbers have high priority.
2. Time quantum where required is 2.
3. Use only required information for each of the algorithms. In a specific case the unrequired/irrelevant information may be skipped.

Question 12:

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");
}
```

Solution:-

Using the above code the putput of Whatsup will print total 6 times

As fork() is used to create new processes which known as child process. The child process is normally

copy of the parent, run a different branch of the program. Even this can run a completely different program.

After fork() child process and parent processes execute in similar. In the above program code the parent fort() process created when if(fork()!=0) executes, it prints the "whatsup" and after this every time when fork()called "whatsup" will print on the screen.

I write the code in unbunto window terminal and after given header file and the replace the fork() test function with main function I write the code give in the question and then compile it the execution and file creation process are display in the display screen short as given below;

Question 13:

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?

Answer:

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) Give an example of a request sequence leading to deadlock.
Answer:
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.

Question 14:
Write a resource allocation algorithm which will prevent deadlock.
Answer:
Following algorithm will prevent 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 request;
}
Else
{
        If (no resource is free)
        {
                Wait until all resources r are free;
        }
        Grant process P individual access to r resources;
}
```

Question 15:
How should threads coordinate? And how should they wait for an event or state change?

Answer:
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

Question 16:
How does thread synchronization problem change with transactional memory?
Answer:
Transactional memory changes the semantics of both waiting and signaling. For example, most condition variable implementations warn against “naked notifies”, or signaling without holding the condition mutex lock. However, signaling within a transaction raises the possibility that the signaling transaction may abort, so the condition it signaled never occurs. Furthermore, we observe that under some conflict resolution policies a recently woken transaction may cause a signaling thread to abort.

Question 17:
What are conditional critical regions? What are their limitations? And how can we overcome these limitations.
Answer:
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.

Question 18:
What are condition variable? What are their limitations? And how can we overcome these limitations?
Answer:
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.

Question 19:
What is lost wakeup problem and how can it be resolved?
Answer:
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

Question 20:
What are the requirements and implications for the discussed two implementations of condition variables for transactional memory?

Answer:
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

Question 21:
Explain, in your own words, how the proposed approach manages to avoid deadlocks when unstructured locking is used.

Answer:
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.

Question 22:
How static analysis is performed and what are its limitations?
Answer:
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.

Question 23:
What is are message-oriented models?

Mechanism of this model is passing messages/events among processes with an easy an efficient way. Such model also provides the facility of putting the messages into the queues unless they are served. Messages have the states of wait and ready etc. pre-emption for high priority messages is facilitated. Message-oriented models bear these features:

- 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:
- Process synchronization is made possible through queues that are attached with processes
- When more than one processes operate on data, the data structure are passed by reference
- Peripheral devices are also work like messages. Signal is sent to devices and an interrupt is generated by devices in response
- Priorities for process are set statically at the time of system design
- 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:

- It carries a handle for identifying a message, and message data portion
- Messages communication is established through channels and ports.
- Messages are sent and received through operations like SendMessage, AwaitReply, WaitForMessage, and SendReply etc.

Question 24:
What is Procedure-Oriented Model?

Mechanism of this model is quickly switching the context of processes by giving the protection and addressing mechanism. Processes use sharing of data extensively. Inter process communication is secured by semaphores, locks, monitors etc. Pre-emption of resources is only given to high priority processes after the resources are released by the using process. Procedure-oriented models bear these features:

- Processes access global data in a synchronized and controlled way.
- As a process and no communication channel associated with it so process creation is easy. Process deletion is also easy when a process holds no lock.
- Process is assigned a single task to complete and the process wanders at many location and states while changing its context.

Procedure-oriented models bear the following characteristics:

- Process synchronization in case of many resources is made possible through locks
- Process are made to share the data, and data access is given for shortest possible time and smallest possible part of resource
- Control of devices and interrupts from devices are also dealt with the help of locks

- Process are given priority dynamically, this priority is on the basis of short time of completion
- To save the context switching from conflicts, global naming is important

Procedure-oriented models provide the following facilities:

- They provide procedures. Procedures have algorithms, local data, parameters and results
- They provide synchronous and asynchronous procedure calls
- FORK, JOIN, WAIT, SIGNAL calls are provided for processes and protection of processes

Question 25:

What factor must be considered while choosing message-oriented or procedure-oriented system model?

We have seen that in the paper that both message-oriented and procedure-oriented models are counter parts of each other. System designed under any one model and be transformed into the other more with confidence. Same functionality can be achieved through the implementation of any of these two models. So why to choose and prefer over the other? For the answer of this question we will have to think two steps down. Process and synchronization facilities are things which make one or the other style more attractive or more tedious.

For example there is a system which is more tending towards utilizing the virtual memory mechanism to perform its tasks; here it is easier to distribute the memory into messages blocks and queue messages; but it is very difficult to protect the shared memory; so in such system it is a better decision to choose a message-oriented model.

There is another system which is more tending towards the usage of stack and block structured allocation. In such case we can choose the procedure-oriented model.

Following factors may also be considered while choosing one these models:

- Organization of real and virtual memory
- Space and size of the word that is used to save the state of each process context switch
- The ease with which process scheduling and dispatching can be accomplished
- The control and arrangement of peripheral devices and interrupts generated by these devices
- Architecture of the register that can be programmed
- Architecture of the instructions

Although the above factors can be considered while choosing one of the models, but it is to be noted that it is very rare that a programmer has a clear influence by one of them over the other. And the other thing even rarer is the fact that a programmer has a full knowledge of the two models and factors influencing them.

Question 26:

How are address translations and physical memory managed in the Exokernel?

For address translation, the virtual address space is divided into two portions. First portion keeps the application data and code. Virtual addresses in this portion are mapped and typically hold exception handling code and page-tables.

Following actions will take place On a TLB miss:

- Exokernel checks where the virtual address is located. In case of the standard user segment, the exception is dispatched directly to the application. In case of the second segment and guaranteed mapping, exokernal installs the TLB entry and continues; and in case of second segment and not guaranteed mapping exokernal forwards it to the application.
- The application searches the virtual address in its page-table structure and, if the access is not valid, an appropriate exception is generated. If the virtual address mapping is allowed,

the application constructs the appropriate TLB entry and its associated capability and invokes the appropriate exokernal routine.

- Exokernel then checks whether the access is allowed are not. If access is allowed then mapping is installed in the TLB and control is returned to the application. And if the access is not allowed an error is returned.
- At the last application does some cleanup work and then resumes its execution.

For efficient virtual memory management, TLB must be refilled fast. To do this , Exokernal use the caching for TLB entries in the kernel. Exokernal uses overlaying the hardware TLB with big software TLB (STLB) to absorb capacity misses. On a TLB miss, Exokernal first checks that the required mapping is in the STLB. If it is located there, Exokernal installs it and resumes execution; in the opposite case, the miss is forwarded to the application.

The STLB have 8 bytes entry and it has total 4096 entries. It has direct mapping and located in unmapped physical memory. When an STLB has a "hit", it takes 18 instructions. In the opposite case, performing an up call to application level on a TLB miss, a system call to install a new mapping is at least 3 to 6 microseconds more expensive. As we know that exokernel has a principle of exposing kernel bookkeeping structures, the STLB can be mapped to a capability that can search the entries efficiently.

Question 26:

How does Exokernel use hardware support to improve the performance?

Exokernel improve its performance by securely exposing the hardware. The main principal of the exokernel architecture is that the kernel should provide secure low-level primitives so that these primitives permit all hardware resources to be accessed as directly as possible. So the the major struggle of an exokernel designer is therefore to safely export all privileged instructions, hardware DMA capabilities, and machine resources. The exported resources are hardware i.e. physical memory, the CPU, disk memory, translation look-aside buffer (TLB), and addressing context identifiers. Due to this securely exposing hardware principle, an improvement occurs in the less tangible machine resources such as interrupts, exceptions, and cross-domain calls. Exokernel should not apply higher-level abstractions on these events. Most of the physical resources should be partitioned and sub-divided in a fine manner. The number, format, and current set of TLB mappings should be visible to library systems, and these should be replaceable by library operating systems. These should also be visible to and replaceable any "privileged" co-processor state. An exokernel must export privileged instructions to library operating systems so that they may be able to implement traditional operating system abstractions such as processes and address spaces. A system call can have an exported operation that checks the ownership of any resources involved. In negative wording, we can say that this principle states that an exokernel should avoid resource management. The resources should be managed to up to the required by protection (i.e., management of allocation, revocation, and ownership). We believe that using the hardware under this principle brings improvement in the performance because distributed, application-specific, and resource management is the best way to build efficient flexible systems.

Question 27:

Can we say that Exokernel is a micro-kernel? What abstractions does it support?

No, we cannot say that exokernel is a micro-kernel. In practice, exokernel system provides applications with greater flexibility and better performance than microkernel systems. The low level interface of Exokernel permits application-level software to manipulate resources very efficiently. Protected control transfer in exokernal is recoded 7 times faster than that of the current implementations of microkernel. Similarly, the exception dispatch is 5 time better than

that of the microkernel. Exokernel gives flexibility to the application level software, while microkernel does not provide this flexibility. For example, virtual memory is implemented at application level, where it can be tightly integrated with distributed shared memory systems and garbage collectors. Efficient protected control transfer in exokernel permits applications to build a large array of efficient IPC primitives by trading performance for additional functionality. As opposite to exokernel, microkernel systems do not give permission to un-trusted application software to build specialized IPC primitives because virtual memory and message passing services are implemented by the kernel and trusted servers. In the same way, microkernel does not provide the capability of modifying many other abstractions, such as page-table structures and process abstractions. Moreover, many of the hardware resources in microkernel systems, such as the network, screen, and disk, are encapsulated in heavyweight servers that cannot be bypassed or made suitable to application-specific needs.
Another aspect to be note is that, like microkernel systems, an exokernel can provide backward compatibility in three ways:

- Binary emulation of the operating system and its programs
- By implementing its hardware abstraction layer on top of an exokernel
- Re-implementing the operating system's abstractions on top of an exokernel

Just similar to microkernels, exokernels are designed to increase extensibility. And just opposite to traditional microkernels, an exokernel pushes the kernel interface much closer to the hardware, which allows for greater flexibility.

Question 28:
Abstractions are slow and are not tuned to all applications, how does Exokernel addresses this problem?
Most of the fixed high-level abstractions are slow. They put a resistance in the performance of application. This slow performance is due to the reason that there is no single way to abstract physical resources or to implement an abstraction that is best for all applications. When an operating system needs an abstraction to be implemented, the operating system has some restrictions to make trade-offs between read-intensive or write-intensive workloads, support for sparse or dense address spaces, etc. These tradeoffs put some penalties on performance on classes of applications. For example, relational databases and garbage collectors sometimes have very predictable data access patterns, and their performance suffers when a general purpose page replacement strategy such as LRU is imposed by the operating system. But we know that the performance improvement is very important factor in application-specific policies. To address this problem of efficiency, Exokernal uses the techniques of application-controlled file caching. And it has been measured that this technique can reduce application running time by as much as 45%.

Question 29:
Differentiate between Locks and Semaphores?

Answer:
Locks: locks are synchronization primitives used to protect shared data structures among threads. Locks can be implemented both ways i.e. busy-waiting and blocking (sleep). Semaphores: semaphores are synchronization primitives used to protect shared data structures among threads as well as can provide coordination among different threads. Semaphores are enhancement to locks and can be used as locks (i.e. mutexes).

Question 30:
Differentiate between Semaphores and Condition Variables?

Answer:
Semaphores: semaphores are synchronization primitives used to protect shared data structures among threads as well as can provide coordination among different threads. Semaphores are enhancement to locks and can be used as locks (i.e. mutexes). Condition Variables: condition variables are used to provide inter-thread coordination i.e. to coordinate events occurring in different threads. If one or more threads are waiting (blocked) for an event/signal to occur in another thread, we use condition variables to provide coordination among these threads.

Question 31:
How can we use Semaphores for thread reaping?

Answer:
Semaphores can be used for thread reaping. Here semaphore is used as mutex i.e. initialized with zero (count = 0). When the we want to wait for the new thread to complete (i.e. pthead_join()) we call P() operation, which causes the calling thread to block as count = -1. The thread who is being joined, would call V() operation when going to exit. Which would signal to the blocked thread and it would resume its execution.

Question 32:
Briefly discuss "Resource Ordering" technique for preventing deadlocks?

Answer:
Resource ordering is a practical and commonly used technique to prevent deadlocks. We assign a number to each resource (i.e. resource itself or the lock/semaphore associated with it) or we assign levels (i.e. these particular resources are categorized at level 1, these resources are at level 2 and so on). At each level there can be resources of multiple types. Rule: All threads will acquire the resources in same order (i.e. first level 1 resources, then level 2 resources and so on), that is, holding the resources taken from previous level, we take resources from next level.
If every thread follows that rule then there would be no chance of a deadlock, because there would be no circular wait situation.

Question 33:
Briefly discuss the problem of "Priority Inversion"?

Answer:
Priority Inversion problem occurs when:

- A high priority job waits for a low priority job while a medium priority job runs on the CPU.
- Or A high priority job keeps spinning in a loop waiting for a lower priority job to release a lock, but the lower priority job never gets a chance to run on the CPU.

Question 34:
Briefly discuss the disadvantages of "Threads" paradigm?

Answer:
Disadvantages of Threads paradigm are as follows:

1. It makes the overall code of the program, non-deterministic. It introduces Heisenbugs in the program. A Heisenbug is a type of bug that disappears or alters its behavior when you attempt to debug it.
2. In large, complex real life programs/problems, the programmer usually does not visualize properly that which synchronization mechanism should be deployed here in a particular situation. Hence, introducing wrong synchronization primitives in different scenarios, resulting in lots of undetectable bugs in the code.
3. As threads grow in numbers, for each thread kernel has to allocate memory internally for its stack. Hence the kernel's memory space is used for the purpose, it grows and soon reaches a limit beyond which it cannot expand. Secondly, when there are too many threads created, the benefit of parallelism achieved by overlapping I/O and CPU activities is destroyed / nullified.
4. As the number of threads grow, the overall thread management, done by a main controller thread, becomes a cumbersome job.
5. Context Switching overhead increases to a great extent, with the increase of no. of threads. Similarly thread scheduling and synchronization overhead increases with the increase in number of threads.

Question 35:
In lectures, we studied that there are three methods possible to achieve concurrency. What is the difference between second (Single Process, multiple events) and third (threads) method?

Answer:
Single process, multiple events approach:
In this approach, we have a single process, when we have to block, we do not block in fact we schedule an event for future firing, and in the call back of that event a registered function is used. The process in fact tries to save itself from getting blocked.
Drawbacks of this approach:

- The process has to maintain the state of each request/session.
- All requests/sessions can be in different states. So a FSM have to be maintained for each session/client.
- All these FSMs are maintained in a global table of a single process.

Threads approach:

Multiple execution states within a single address space. Instead of managing a global table for the states of every session, we offload all the session processing task to one execution stream (execution state) and another session processing to another execution stream and so on. Each execution stream maintains its data by itself.

Question 36:
What are the things included in a thread's context?

Answer:
Thread's context usually consists of:

- CPU registers
- Program Counter
- Stack Pointer

Question 37:
Who generates/creates the user-level thread? And do we require a system call to generate a user-level thread?

Answer:
User-level threads are created by a library (run time system). Whenever the program wants to create a thread, it calls the library, no system call is triggered, the library creates the thread itself.

Question 38:
What is the concept of "Scheduler Activation" as discussed in the lectures?

Answer:
Scheduler Activation:
Suppose kernel-level thread is going to block, if kernel somehow, notify it to the application / process [that this particular kernel thread is going to block]. The run time system (user-level thread library) checks that the user-level threads which are mapped to this blocking kernel thread, do not have some lock which other kernel level threads may need. If there are such lock(s), it temporarily gets these lock(s) back. This is known as scheduler activation. This scheduler activation is used on top of hybrid threads model.

Question 39:
What is the relationship between different threads in a single process? How one thread can wait for the termination of another thread?

Answer:
Different threads in a single process are peers of each other (i.e. there is no parent child relationship).
A thread can wait for the termination of another thread by calling the pthread_join() function, passing the joining thread's thread ID as argument in the function call.

Question 40:
In multithreaded programs, what are the two cases in which we need synchronization mechanisms?

Answer:
The two cases are:

1. to protect shared resources/data structures
2. to provide coordination among different threads of a process

Question 41:
In threads execution, what we mean by the term "arbitrary interleaving of executions"?

Answer:
Arbitrary interleaving of executions means we cannot predict the order of execution of threads, and execution can switch from one thread to another at arbitrary point of time. Secondly, we also cannot predict the rate of execution of instructions in different threads. So the goal is to write the multi-threaded program in such a way that any arbitrary interleaving of executions of threads does not make our program unpredictable.

Question 42:
In the "Too Much Milk" problem discussed in the lecture, what is the flaw in solution#2?

Answer:
Consider the situation, thread A runs, leaves note A then thread B runs, leaves note B, then thread A runs, checks condition (no Note from B), since note from B is present, hence fails then thread B runs, checks condition (no Note from A), since note from A is present hence fails, then thread A runs, removes note A, then thread B runs, removes note B. Hence, no one buys milk.

Question 43:
Consider the following ways of handling deadlock:

(1) Banker's algorithm

(2) Detect deadlock and kill thread, releasing all resources

(3) Reserve all resources in advance

(4) Restart thread and release all resources if thread needs to wait

(5) Resource ordering

(6) Detect deadlock and roll back thread's actions.

(a) One criterion to use in evaluating different approaches to deadlock is which approach permits the greatest concurrency. In other words, which approach allows the most threads to make progress without waiting when there is no deadlock? Give a rank order from 1 to 6 for each of the ways of handling deadlock just listed, where 1 allows the greatest degree of concurrency. Comment on your ordering.

(b) Another criterion is efficiency; in other words, which requires the least processor overhead. Rank order the approaches from 1 to 6, with 1 being the most efficient, assuming that deadlock is a very rare event. Comment on your ordering. Does your ordering change if deadlocks occur frequently?

Answer to Question (a)

Ser	Most to Least Degree	Comments
1	Restart thread and release all resources if thread needs to wait	If threads require waiting then resources should be released at once for use of others
2	Detect deadlock and roll back thread's actions	They cause waiting so concurrency is effected
3	Detect deadlock and kill thread, releasing all resources	"
4	Banker's algorithm	Create unwanted delay
5	Resource ordering	"
6	Reserve all resources in advance	Threads wait longer

Answer to Question (b)

Ser	Ordering	Comments
1	Reserve all resources in advance	
2	Resource ordering	
3	Restart thread and release all resources if thread needs to wait	
4	Detect deadlock and roll back thread's actions	
5	Detect deadlock and kill thread, releasing all resources	
6	Banker's algorithm	

In case of frequent deadlocks

Ser	Ordering	Comments
1	Reserve all resources in advance	
2	Resource ordering	
3	Banker's algorithm	
4	Detect deadlock and roll back thread's actions	
5	Detect deadlock and kill thread, releasing all resources	
6	Restart thread and release all resources if thread needs to wait	

Question 44:

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?

There will be no Deadlock if following conditions are avoided:-

- Mutual Exclusion: One or more than one resource must be held by a process in a non-sharable (exclusive) mode.
- Hold and Wait: A process holds a resource while waiting for another resource.
- No Preemption: There is only voluntary release of a resource - nobody else can make a process give up a resource.
- Circular Wait: Process A waits for Process B waits for Process C .... waits for Process A.

(b) Give an example of a request sequence leading to deadlock.

Following sequence will lead to deadlock

Request Sequence	Comments

process1: ...
open(file1,write);
open(file2,write);
…..
process2: ...
open(file2,write);
open(file1,write);
…..
	Deadlock will occur once executed concurrently

process1:if(C)send(process2,m);process2:
if(C)send(process1,m);
while(1){...
receive(process2,m);
send(process2,m);
... }
while(1){...
receive (process1,m);
send(process1,m);
... }
	Deadlock will occur when C will not be true

(c) Write a resource allocation algorithm which will prevent deadlock.

Resource allocation Algorithm to prevent deadlock

```
if process P has ever acquired any resources before, then
   refuse the request
   else if any resource R1, …Rn does not exist then
   refuse the request
  else
  {          if any resource R1, …Rn is not free then
                              wait until all resources R1, ... Rn are free
             end if
             grant process P exclusive access to resources R1, ... Rn
   }
   endif
```