CMR
INSTITUTE OF
TECHNOLOGY

USN | | | | | | | | | | |

Internal Assessment Test - III

Sub:	Programming Languages						Code:	10CS666
Date:	31/ 05 / 2017	Duration:	90 mins	Max Marks:	50	Sem: VI	Branch:	CSE

Answer Any FIVE FULL Questions

		Marks	OBE	
			CO	RBT
1	Briefly describe the keyword “Synchronised” in java. How is it used in concurrency?	[10]	CO2, CO3	L2
ans	Synchronized keyword in Java has to do with thread-safety, that is, when multiple threads read or write the same variable. This can happen directly (by accessing the same variable) or indirectly (by using a class that uses another class that accesses the same variable). The synchronized keyword is used to define a block of code where multiple threads can access the same variable in a safe way. Syntax-wise the synchronized keyword takes an Object as it's parameter (called a lock object), which is then followed by a { block of code }. When execution encounters this keyword, the current thread tries to "lock/acquire/own" (take your pick) the lock object and execute the associated block of code after the lock has been acquired. Any writes to variables inside the synchronized code block are guaranteed to be visible to every other thread that similarly executes code inside a synchronized code block using the same lock object. Only one thread at a time can hold the lock, during which time all other threads trying to acquire the same lock object will wait (pause their execution). The lock will be released when execution exits the synchronized code block. Synchronized methods: Adding synchronized keyword to a method definition is equal to the entire method body being wrapped in a synchronized code block with the lock object being this (for instance methods) and ClassInQuestion.getClass() (for class methods). - Instance method is a method which does not have static keyword. - Class method is a method which has static keyword. Technical			

Without synchronization, it is not guaranteed in which order the reads and writes happen, possibly leaving the variable with garbage.
(For example a variable could end up with half of the bits written by one thread and half of the bits written by another thread, leaving the variable in a state that neither of the threads tried to write, but a combined mess of both.)

It is not enough to complete a write operation in a thread before (wall-clock time) another thread reads it, because hardware could have cached the value of the variable, and the reading thread would see the cached value instead of what was written to it.
Conclusion

Thus in Java's case, you have to follow the Java Memory Model to ensure that threading errors do not happen.
In other words: Use synchronization, atomic operations or classes that use them for you under the hoods.

2	Why does implementing a lock require atomic operations supported by hardware ? Explain using the basic spin-lock as example.	[10]	CO3	L2

ans Locks typically require hardware support for efficient implementation. This support usually takes the form of one or more atomic instructions such as "test-and-set", "fetch-and-add" or "compare-and-swap". These instructions allow a single process to test if the lock is free, and if free, acquire the lock in a single atomic operation.
Uniprocessor architectures have the option of using uninterruptable sequences of instructions—using special instructions or instruction prefixes to disable interrupts temporarily—but this technique does not work for multiprocessor shared-memory machines. Proper support for locks in a multiprocessor environment can require quite complex hardware or software support, with substantial synchroniIn concurrent programming, an operation (or set of operations) is atomic, linearizable, indivisible or uninterruptible if it appears to the rest of the system to occur instantaneously. Atomicity is a guarantee of isolation from concurrent processes. Additionally, atomic operations commonly have a succeed-or-fail definition—they either successfully change the state of the system, or have no apparent effect.

In a concurrent system, processes can access a shared object at the same time. Because multiple processes are accessing a single object, there may arise a situation in which while one process is accessing the object, another process changes its contents. This example demonstrates the need for linearizability. In a linearizable system although operations overlap on a shared object, each operation appears to take place instantaneously. Linearizability is a strong correctness condition, which constrains what outputs are possible when an object is accessed by multiple processes concurrently. It is a safety property which ensures that operations do not complete in an unexpected or unpredictable manner. If a system is linearizable it allows a programmer to reason about the system.

Atomicity is often enforced by mutual exclusion, whether at the hardware level

building on a cache coherency protocol, or the software level using semaphores or locks. Thus, an atomic operation does not necessarily actually occur instantaneously. The benefit comes from the appearance: the system behaves as if each operation occurred instantly, separated by pauses. This makes the system consistent. Because of this, implementation details may be ignored by the user, except insofar as they affect performance. If an operation is not atomic, the user will also need to understand and cope with sporadic extraneous behaviour caused by interactions between concurrent operations, which by their nature are likely to be hard to reproduce and debug.zation issues.

3	What is preemption ? Why is it required ?	[10]	CO3	L2

ans In computing, preemption is the act of temporarily interrupting a task being carried out by a computer system, without requiring its cooperation, and with the intention of resuming the task at a later time. Such changes of the executed task are known as context switches. It is normally carried out by a privileged task or part of the system known as a preemptive scheduler, which has the power to preempt, or interrupt, and later resume, other tasks in the system.
The term preemptive multitasking is used to distinguish a multitasking operating system, which permits preemption of tasks, from a cooperative multitasking system wherein processes or tasks must be explicitly programmed to yield when they do not need system resources.

In simple terms: Preemptive multitasking involves the use of an interrupt mechanism which suspends the currently executing process and invokes a scheduler to determine which process should execute next. Therefore, all processes will get some amount of CPU time at any given time.

In preemptive multitasking, the operating system kernel can also initiate a context switch to satisfy the scheduling policy's priority constraint, thus preempting the active task. In general, preemption means "prior seizure of". When the high priority task at that instance seizes the currently running task, it is known as preemptive scheduling.

The term "preemptive multitasking" is sometimes mistakenly used when the intended meaning is more specific, referring instead to the class of scheduling policies known as time-shared scheduling, or time-sharing.

Preemptive multitasking allows the computer system to more reliably guarantee each process a regular "slice" of operating time. It also allows the system to rapidly deal with important external events like incoming data, which might require the immediate attention of one or another process.

At any specific time, processes can be grouped into two categories: those that are waiting for input or output (called "I/O bound"), and those that are fully utilizing the CPU ("CPU bound"). In early systems, processes would often "poll", or "busywait" while waiting for requested input (such as disk, keyboard or network input). During this time, the process was not performing useful work, but still maintained complete control of the CPU. With the advent of

interrupts and preemptive multitasking, these I/O bound processes could be "blocked", or put on hold, pending the arrival of the necessary data, allowing other processes to utilize the CPU. As the arrival of the requested data would generate an interrupt, blocked processes could be guaranteed a timely return to execution.

Although multitasking techniques were originally developed to allow multiple users to share a single machine, it soon became apparent that multitasking was useful regardless of the number of users. Many operating systems, from mainframes down to single-user personal computers and no-user control systems (like those in robotic spacecraft), have recognized the usefulness of multitasking support for a variety of reasons. Multitasking makes it possible for a single user to run multiple applications at the same time, or to run "background" processes while retaining control of the computer.

4	What are some of the ways of inter process communication ? Briefly explain them.	[10]	CO3	L1

ans A process can be of two type:

Independent process.
Co-operating process.

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently but in practical, there are many situations when co-operative nature can be utilised for increasing computational speed, convenience and modularity. Inter process communication (IPC) is a mechanism which allows processes to communicate each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other using these two ways:

Shared Memory
Message passing

An operating system can implement both method of communication. First, we will discuss the shared memory method of communication and then message passing. Communication between processes using shared memory requires processes to share some variable and it completely depends on how programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously and they share some resources or use some information from other process, process1 generate information about certain computations or resources being used and keeps it as a record in shared memory. When process2 need to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from other process as well as for delivering any specific information to other process.

	The IPC mechanism can be classified into pipes, first in, first out (FIFO), and shared memory. Pipes were introduced in the UNIX operating system. In this mechanism, the data flow is unidirectional. A pipe can be imagined as a hose pipe in which the data enters through one end and flows out from the other end. A pipe is generally created by invoking the pipe system call, which in turn generates a pair of file descriptors. Descriptors are usually created to point to a pipe node. One of the main features of pipes is that the data flowing through a pipe is transient, which means data can be read from the read descriptor only once. If the data is written into the write descriptor, the data can be read only in the order in which the data was written. The working principle of FIFO is very similar to that of pipes. The data flow in FIFO is unidirectional and is identified by access points. The difference between the two is that FIFO is identified by an access point, which is a file within the file system, whereas pipes are identified by an access point.			
5	When will a synchronisation mechanism be blocking ? How can this be addressed using CAS ?	[10]	CO3	L2
6	What are the 6 types of thread creation ? Explain any 2.	[10]	CO3	L2
ans	There is exactly one way to create a new thread in Java and that is to instantiate java.lang.Thread (to actually run that thread you also need to call start()). Everything else that creates threads in Java code falls back to this one way behind the cover (e.g. a ThreadFactory implementation will instantiate Thread objects at some point, ...). There are two different ways to specify which code to run in that Thread: Implement the interface java.lang.Runnable and pass an instance of the class implementing it to the Thread constructor. Extend Thread itself and override its run() method. The first approach (implementing Runnable) is usually considered the more correct approach because you don't usually create a new "kind" of Thread, but simply want to run some code (i.e. a Runnable) in a dedicated thread.			
7	Briefly describe the following: a) Memory coherence b) Monitors c) Race condition d) Interrupt e) Thread pool	[2x5]	CO2, CO3	L2
Ans	a) Memory coherence is an issue that affects the design of computer systems in which two or more processors or cores share a common area of memory. In multiprocessor (or multicore) systems, there are two or more processing elements working at the same time, and so it is possible that they simultaneously access the same memory location. Provided none of them changes the data in this location, they can share it indefinitely and cache it as they please. But as			

soon as one updates the location, the others might work on an out-of-date copy that, e.g., resides in their local cache. Consequently, some scheme is required to notify all the processing elements of changes to shared values; such a scheme is known as a memory coherence protocol, and if such a protocol is employed the system is said to have a coherent memory.

b) In concurrent programming, a monitor is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true. Monitors also have a mechanism for signalling other threads that their condition has been met. A monitor consists of a mutex (lock) object and condition variables. A condition variable is basically a container of threads that are waiting for a certain condition. Monitors provide a mechanism for threads to temporarily give up exclusive access in order to wait for some condition to be met, before regaining exclusive access and resuming their task.

c) A race condition or race hazard is the behavior of an electronic, software, or other system where the output is dependent on the sequence or timing of other uncontrollable events. It becomes a bug when events do not happen in the order the programmer intended. The term originates with the idea of two signals racing each other to influence the output first.

d) In system programming, an interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. An interrupt alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing. The processor responds by suspending its current activities, saving its state, and executing a function called an interrupt handler (or an interrupt service routine, ISR) to deal with the event. This interruption is temporary, and, after the interrupt handler finishes, the processor resumes normal activities. There are two types of interrupts: hardware interrupts and software interrupts.

e) In computer programming, a thread pool is a software design pattern for achieving concurrency of execution in a computer program. Often also called a replicated workers or worker-crew model, a thread pool maintains multiple threads waiting for tasks to be allocated for concurrent execution by the supervising program. By maintaining a pool of threads, the model increases performance and avoids latency in execution due to frequent creation and destruction of threads for short-lived tasks. The number of available threads is tuned to the computing resources available to the program, such as parallel processors, cores, memory, and network sockets.