END TERM Examination, May 2014
(Model Test Paper)
B.Tech (ECE) IV Semester

Paper Code: ETCS - 212 **Subject: Operating Systems**
Time: 3 hrs **Maximum Marks: 75**

Note: Attempt five questions including Q. No. 1 which is compulsory. Select one question from each unit.

Q1. Explain in brief: (2½ x 10 = 25)
i. Define "Thrashing".

Thrashing occurs when a system spends more time processing page faults than executing transactions. While processing page faults is necessary to in order to appreciate the benefits of virtual memory, thrashing has a negative effect on the system.
As the page fault rate increases, more transactions need processing from the paging device. The queue at the paging device increases, resulting in increased service time for a page fault. While the transactions in the system are waiting for the paging device, CPU utilization, system throughput and system response time decrease, resulting in below optimal performance of a system.
Thrashing becomes a greater threat as the degree of multiprogramming of the system increases.
ii. List the differences between Internal Fragmentation and external Fragmentation.

External Fragmentation*:* External Fragmentation happens when a dynamic memory allocation algorithm allocates some memory and a small piece is left over that cannot be effectively used. If too much external fragmentation occurs, the amount of usable memory is drastically reduced. Total memory space exists to satisfy a request, but it is not contiguous.
Internal Fragmentation*:* Internal fragmentation is the space wasted inside of allocated memory blocks because of restriction on the allowed sizes of allocated blocks. Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
iii. What do you know about "Spooling".

Acronym for ***s****imultaneous* ***p****eripheral* ***o****perations* ***on-line****, spooling* refers to putting jobs in a buffer, a special area in memory or on a disk where a device can access them when it is ready. Spooling is useful because devices access data at different rates. The buffer provides a waiting station where data can rest while the slower device catches up.
The most common spooling application is *print spooling.* In print spooling, documents are loaded into a buffer (usually an area on a disk), and then the printer pulls them off the buffer at its own rate. Because the documents are in a buffer where they can be accessed by the printer, you can perform other operations on the computer while the printing takes place in the background. Spooling also lets you place a number of print jobs on a queue instead of waiting for each one to finish before specifying the next one.

iv. Why does the page size is of 2_n.

Recall that paging is implemented by breaking up an address into a page and offset number. It is most efficient to break the address into X page bits and Y offset bits, rather than perform arithmetic on the address to calculate the page number and offset. Because each bit position represents a power of 2, splitting an address between bits results in a page size that is a power of 2.

v. List the necessary and sufficient conditions of deadlock.

Mutual Exclusion: The resources involved must be unshareable; otherwise, the processes would not be prevented from using the resource when necessary.

Hold and wait or partial allocation: The processes must hold the resources they have already been allocated while waiting for other (requested) resources. If the process had to release its resources when a new resource or resources were requested, deadlock could not occur because the process would not prevent others from using resources that it controlled.

No pre-emption: The processes must not have resources taken away while that resource is being used. Otherwise, deadlock could not occur since the operating system could simply take enough resources from running processes to enable any process to finish.

Resource waiting or circular wait: A circular chain of processes, with each process holding resources which are currently being requested by the next process in the chain, cannot exist. If it does, the cycle theorem (which states that "a cycle in the resource graph is necessary for deadlock to occur") indicated that deadlock could occur.

vi. **Explain the disadvantages of second attempt of Dekker's algorithm for process synchronization.**

It succeeds in preventing the conflict by enforcing mutual exclusion, meaning that only one process may use the resource at a time and will wait if another process is using it. This is achieved with the use of two "flags" and a "token". The flags indicate whether a process wants to enter the critical section (CS) or not; a value of 1 means TRUE that the process wants to enter the CS, while 0, or FALSE, means the opposite. The token, which can also have a value of 1 or 0, indicates priority when both processes have their flags set to TRUE. Dekker's algorithm will allow only a single process to use a resource if two processes are trying to use it at the same time. The highlight of the algorithm is how it solves this problem.

This algorithm can successfully enforce mutual exclusion but will constantly test whether the critical section is available and therefore wastes significant processor time. It creates the problem known as lockstep synchronization, in which each thread may only execute in strict synchronization. It is also non-expandable as it only supports a maximum of two processes for mutual exclusion.

vii. List the differences between Preemption and non-Preemption.

Preemption:

- Preemption means the operating system moves a process from running to ready without the process requesting it.
- Without preemption, the system implements ``run to completion (or yield or block)".
- The ``preempt" arc in the diagram.
- We do not consider yield (a solid arrow from running to ready).
- Preemption needs a clock interrupt (or equivalent).

 Preemption is needed to guarantee fairness.
 Found in all modern general purpose operating systems.
 Even non preemptive systems can be multiprogrammed (e.g., when processes block for I/O).

viii. What is “buffering”.

In computing, a buffer is a region of memory used to temporarily hold data while it is being moved from one place to another. Typically, the data is stored in a buffer as it is retrieved from an input device (such as a keyboard) or just before it is sent to an output device (such as a printer). However, a buffer may be used when moving data between processes within a computer. This is comparable to buffers in telecommunication. Buffers can be implemented in either hardware or software, but the vast majority of buffers are implemented in software. Buffers are typically used when there is a difference between the rate at which data is received and the rate at which it can be processed, or in the case that these rates are variable, for example in a printer spooler.

ix. List the disadvantages of Index allocation strategy.

 Large overhead for meta-data:
 Wastes space for unneeded pointers (most files are small!)
 Need to read indirect blocks of pointers to
 Calculate addresses (extra disk read)
 Keep indirect blocks cached in main memory

x. Define Multiprogramming.

A multiprogramming operating system is one that allows end-users to run more than one program at a time. The development of such a system, the first type to allow this functionality, was a major step in the development of sophisticated computers. The technology works by allowing the central processing unit (CPU) of a computer to switch between two or more running tasks when the CPU is idle. Multiprogramming is a rudimentary form of parallel processing in which several programs are run at the same time on a uniprocessor. Since there is only one processor, there can be no true simultaneous execution of different programs. Instead, the operating system executes part of one program, then part of another, and so on. To the user it appears that all programs are executing at the same time.

UNIT 1

Q2a Discuss about safe state and usefulness of safety algorithm for Deadlock avoidance (12½)

 A system is in a safe state only if there exists a safe sequence of processes P1, P2, …….., Pn where:
o For each Pi, the resources that Pi can still request can be satisfied by the currently available resources plus the resources help by all Pj, j<i.

 If a system is in safe state, there is no deadlock.
 If the system is deadlocked, it is in an unsafe state.
 If a system is in unsafe state, there is a possibility for a deadlock.
 Avoidance: making sure the system will not enter an unsafe state.

Safety *algorithm* (to check for a safe state):

1. Let work be an integer array of length m, initialized to available. Let finish be a boolean array of length n, initialized to false.

2. Find an i such that both:
o finish[i] == false

o need[i] <= work
If no such i exists, go to step 4
3. work = work + allocation[i]; finish[i] = true; Go to step 2

4. If finish[i] == true for all i, then the system is in a safe state, otherwise unsafe.

A state is considered safe if it is possible for all processes to finish executing (terminate). Since the system cannot know when a process will terminate, or how many resources it will have requested by then, the system assumes that all processes will eventually attempt to acquire their stated maximum resources and terminate soon afterward. This is a reasonable assumption in most cases since the system is not particularly concerned with how long each process runs. Also, if a process terminates without acquiring its maximum resources, it only makes it easier on the system. A safe state is considered to be the decision maker if it is going to process ready queue. Safe State ensures the Security.
Given that assumption, the algorithm determines if a state is **safe** by trying to find a hypothetical set of requests by the processes that would allow each to acquire its maximum resources and then terminate (returning its resources to the system). Any state where no such set exists is an **unsafe** state.
Example*:*
We can show that the state given in the previous example is a safe state by showing that it is possible for each process to acquire its maximum resources and then terminate.
1. P1 acquires 2 A, 1 B and 1 D more resources, achieving its maximum
□ [available resource: <3 1 1 2> - <2 1 0 1> = <1 0 1 1>]

□ The system now still has 1 A, no B, 1 C and 1 D resource available
2. P1 terminates, returning 3 A, 3 B, 2 C and 2 D resources to the system
□ [available resource: <1 0 1 1> + <3 3 2 2> = <4 3 3 3>]

□ The system now has 4 A, 3 B, 3 C and 3 D resources available
3. P2 acquires 2 B and 1 D extra resources, then terminates, returning all its resources
□ [available resource: <4 3 3 3> - <0 2 0 1>+<1 2 3 4> = <5 3 6 6>]

□ The system now has 5 A, 3 B, 6 C and 6 D resources
4. P3 acquires 1 B and 4 C resources and terminates
□ [available resource: <5 3 6 6> - <0 1 4 0> + <1 3 5 0> = <6 5 7 6>

□ The system now has all resources: 6 A, 5 B, 7 C and 6 D
5. Because all processes were able to terminate, this state is safe

These requests and acquisitions are *hypothetical*. The algorithm generates them to check the safety of the state, but no resources are actually given and no processes actually terminate. Also note that the order in which these requests are generated – if several can be fulfilled – doesn't matter, because all hypothetical requests let a process terminate, thereby increasing the system's free resources.
Q2 b) Describe the state transition model in which few states resides in secondary memory. Also describe the use of these states. (12½)
In a multitasking computer system, processes may occupy a variety of states. These distinct states may not actually be recognized as such by the operating system kernel, however they are a useful abstraction for the understanding of processes.

Created:
(Also called **New**) When a process is first created, it occupies the "created" or "new" state. In this state, the process awaits admission to the "ready" state. This admission will be approved or delayed by a long-term, or admission, scheduler. Typically in most desktop computer systems, this admission will be approved automatically, however for real-time operating systems this admission may be delayed. In a real time system, admitting too many processes to the "ready" state may lead to oversaturation and over contention for the systems resources, leading to an inability to meet process deadlines.
Ready and waiting:
A "ready" or "waiting" process has been loaded into main memory and is awaiting execution on a CPU (to be context switched onto the CPU by the dispatcher, or short-term scheduler). There may be many "ready" processes at any one point of the system's execution—for example, in a one-processor system, only one process can be executing at any one time, and all other "concurrently executing" processes will be waiting for execution.

A *ready queue* or run queue is used in computer scheduling. Modern computers are capable of running many different programs or processes at the same time. However, the CPU is only capable of handling one process at a time. Processes that are ready for the CPU are kept in a queue for "ready" processes. Other processes that are waiting for an event to occur, such as loading information from a hard drive or waiting on an internet connection, are not in the ready queue.
Running:
A process moves into the running state when it is chosen for execution. The process's instructions are executed by one of the CPUs (or cores) of the system. There is at most one running process per CPU or core.
Blocked (Waiting)*:*
A process that is blocked on some event (such as I/O operation completion or a signal). A process may be blocked due to various reasons such as when a particular process has exhausted the CPU time allocated to it or it is waiting for an event to occur.
Terminated:
A process may be terminated, either from the "running" state by completing its execution or by explicitly being killed. In either of these cases, the process moves to the "terminated" state. The underlying program is no longer executing, but the process remains in the process table as a *zombie process* until its parent process calls the wait system call to read its exit status, at which point the process is removed from the process table, finally ending the process's lifetime. If the parent fails to call wait, this continues to consume the process table entry.
UNIT 2

Q4. Consider the following process: (12½)

PROCESS	**ARRIVAL TIME**	**SERVICE TIME**
P1	0	8
P2	1	4
P3	2	9
P4	3	5