Question paper follows same format as last year, but some mark allocation is different.

6 questions

Q1: 20 – Chp4  
Q2: 20 – Chp5  
Q3: 20 – Chp6  
Q4: 15 – Chp 7  
Q5: Theretical prac aspects – stack diagrams + floating points numbers  
Q6: Cold code

**Chapter 4 – threads**

**Multithreaded server architecture**

When a client sends a request to the server, the server has a listening request. It then spawns a 2nd/new thread to handle that request. It then resumes listening to client requests.   
(this is a diagram)

**Potential Q: why do we want to have a multithreaded system:**  
**Benefits – there are 4 things – know them well**

**Responsiveness:** When a process runs, certain aspects of the process can be blocked (ie network IO waiting), but a 2nd thread can be still doing stuff.

**Resource sharing:** Threads can easily share info among themselves. It is difficult to share info among processes.

**Economy:** Creating and context switching among multiple threads is cheaper than processes.

**Scale:** Take full advantage of multiprocessor environment. Don’t waste the Cores.

**MultiCore Programming**

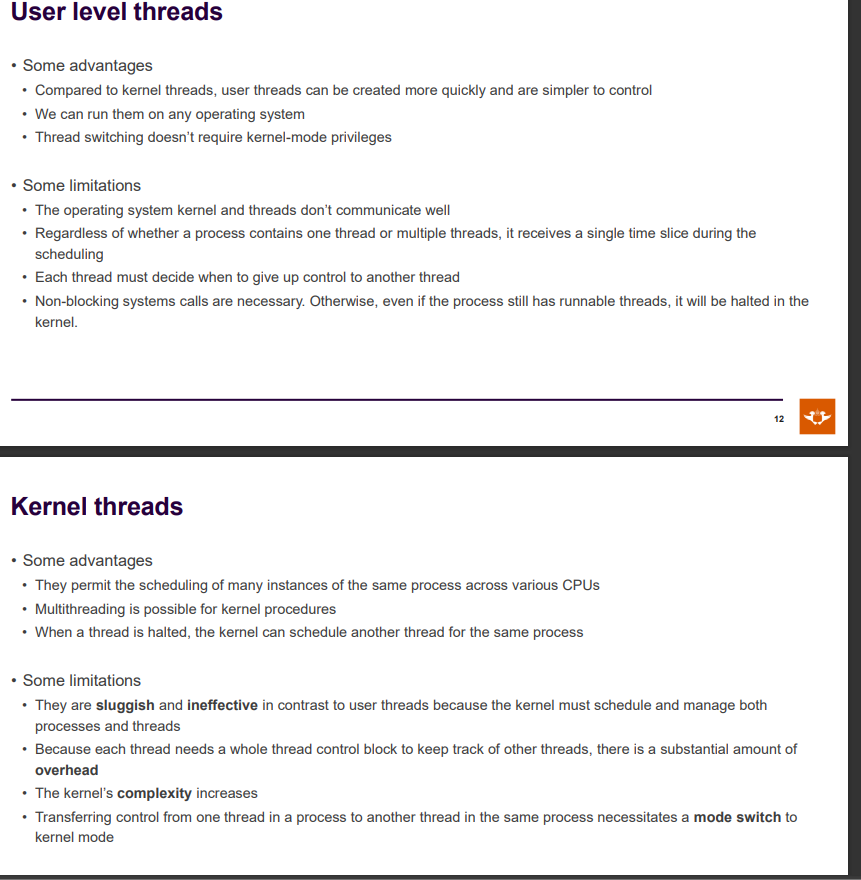
**Parallelism**: Describes where you have more than one thread that runs simultaneously. Multiuple tasks executing simultaneously (Concurrency vs Parrallelism diagram is good eg)

**Concurrency**: Multiple threads making progress in the system. (Timeslicing with single core.)

**User threads and kernal threads (Know difference and advantages)**

Threads exist in different architecuter of system.

Threads can start in applications: **User level threads**: These exist when the kernal does not support threading. Support is built in user mode.  
ADVANTAGES **USER THREADS**:  
-They are simpler to control easier to create (compared to kernal threads)-Can run on any OS  
-Thread switching does not require kernal mode privileges   
LIMITATIONS NOT MENTIONED IN CLASS



**Kernal Threads**

Can schedule on different CPUS.

Multithreading is possible for kernal procedures.

When a thread is halted, kernal can schedule another thread for same process.

**Chapter 5**

Synchronization and when process or threads work together and share info, and the issue that arise with that sharing of info. Such an issue can cause a race condition: sharing a global variable among threads or processes. It describes a counter variable to describe the number of things consumed or/and produced.

**Race Condition**

Incrementing and de-incrementing eg.Counter, takes a number of eg.3 levels of low level instructions. It must be noted they can be interrupted during that 3 steps. This causes the race condition.

The separate process gets CPU time then. It loads the global variable (counter) into its own register.

Where the global variable is not what we need/expected? **Look at slide.**

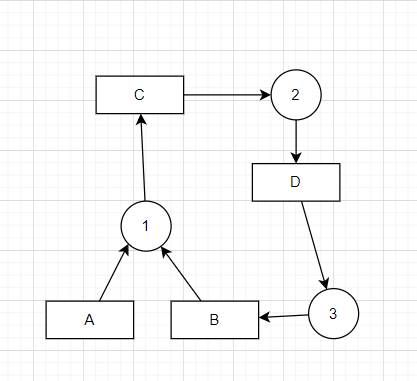
**Critical section problem**

There are **4 generalised areas in code:**1.Entry section: when we make decision of who can enter CS.  
2.Acutal critical section  
3.Exit section: tell processors that ur done  
4.Remainder Section: while(true) – does not make a choice of if entering CS. (As long as not the other 3, it’s the remainder)

Mutex lock (acquire and lock) and Semaphore (block and wakeup)

**Eg Class Scenario – Resource Allocation Graphs**

Process 1 holds A and holds B and Requests C  
Process 2 holds C and Requests D  
Process 3 holds D and Requests B

  
  
This is a deadlock.

**Chapter 6 – CPU Scheduling**

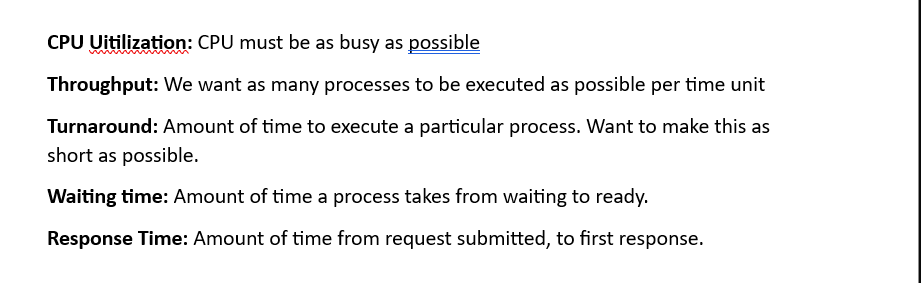
There are various algorithms that are being used when looking at how to chedule. These algorithms typically have:

**Non-preemtive:** Cant be interrupted. Must voluntarily let go of CPU.  
-Finished running  
-needs something like eg Hardware space

**Preemtive:** Not voluntary. Something else can interrupt running process.

There are algorithms that are preemtive and non-preemtive.   
Flag to highlight preemtive is the word **quanta.** (unit of time, max number of times can run per time slice)

**CPU Scheduler?**

****

* CPU Utilization: Want to maximise
* Throughput: How many processes complete per unit time? Maximise
* Turnaround time: Time to execute a process. Minimise
* Waiting Time: Time wasted in waiting queue. Minimise
* Response time: Time between request in and first response out. Minimise

**Priority Scheduling**

A problem is certain processes with high priority are only ones looked at. Low priority processes can starve (don’t get any CPU time).

This is solved by the principle of **aging.** This is where you take low priority processes and increase their priority, so they eventually get CPU time.

**Time quantum and context switch time**

A problem when deeling with preemtive scheduling is the size of quanta. The bigger the size, the slower the system will be. As process with get huge amount of CPU time, before another gets it.

IF it’s a small size, the problem is there are too many/much more context switches which take up a lot of CPU time. (smaller quanta size, more wasted CPU time/overhead).

**Multiple processor scheduling**

When you have multiple processors, you can have affinity.  
Remember processors have cache memory. So when a context switch occurs, the cache memory from that process is not always lost, so its better in that case for the process to run on that original processor.

**NUMA and CPU Scheduling (NON uniform memory architecture)**

The above problem is blown up in scale with NUMA. Its where there are multiple CPUS on a large motherboard with lots of memory. The distance of memory to each CPU is different, and the distance has a cost in the access time. So its better for a CPU to access and use memory close to it. This is what must be considered when scheduling processors.

**NBNB running out of time:**

**CPU scheduling alrgorithm**

**Understand: Priority scheduling, round robin, quant ect**

**Chapter 7**

All about memory.

**Address binding**

Main memory and the way we access it differs based on where we are in the processes life:

**Compiler** combines finished code to relocatable address (eg 14bites from beginning of this memory section)

**Linker** then binds these relocatable addresses to absolute addresses.

**Multiple partition allocation**

**Swapping**

Continuous memory allocation.  
…

**Dynamic storage allocation problem**

To find the best whole for algorithm

First-Fit:

Best fit:

Worst Fit:

**Paging example**

**Logical to physical (that demonstration)   
Always use 8-bits** for binary logical + binary physical