Part A

**Real-time system**: **(1)** Generally a control system, often embedded. Takes in info from env, processes it, and generates a response. **(2)** Reacts, responds, and alters its actions to affect the env in which it is placed. **(3)** Implies that there is smth significant and important about its response time. Doesn’t need to be lightning fast. Just has to be “timely”, which varies from app to app. **(4)** Has a guaranteed, calculatable (deterministic), worst case response time to an event under its control. Can also have min response time. Not just max response time. Min-max window tolerance. **(5)** A system where the correct answer at the wrong time is the wrong answer.

**HARD** real time system: failure to meet specified worst case response time leads to overall system failure. Time sensitive. **Eg,** Engine management system, pacemaker, car ABS, autopilot, DSP audio filters.

**SOFT** real time system: failure to meet specified response time merely results in system degradation, not necessarily system failure. Typically quotes an avg response time against which degradation can be judged. **Eg.** Elevator. Response time varies depending on time of day and number of passengers. Maybe acceptable if 95% requests are met within 30 seconds, averaged over a day. **Eg.** Cruise control. +- 2 kph 95% of the time.

**EVENT** driven: input sensor detects an event. **(1)** Interrupt: ISR. **Pros:** predictable and deterministic. Response time is independent of other sensors. Can be prioritized in hardware. Lots more spare processing power for CPU to do other stuff. **Cons:** More complex sw. More expensive hardware. Needs sensors to generate interrupt. Needs priority hardware. Needs system level code (interrupt handler). Impossible to debug because asynchronous sensor interrupts with unpredictable frequency and timing. Interrupt handlers make it multi-threaded code. **(2)** Polling. Literally while loop polling. **Pros:** Easier to write, no need for ISR, single threaded, easily debugged. **Cons:** events can be missed within polling loop. Difficulty in prioritizing those sensors. More sensors degrades overall system response time since takes longer to poll. Uses tons of CPU time.

**Time** driven: periodic interval actions. **Eg.** Task A every 30 ms. Task B every 40 ms. Hardware timer is implemented. Rate Monotonic Scheduling (RMS).

Safety with: **(1)** Interlocks. Electro-mechanical (Eg. Keys, door, latch). **(2)** Redundancy. Failure of one system can lead to 2nd system taking over (Eg. Dual port ssd)

**Time slicing**: quickly swapping in and out processes to be run by the scheduler. Pre-emption. Allows the processor's time to be shared between a number of tasks, giving the illusion that it is dealing with these tasks in parallel. An attempt to create **concurrency** by forcing a CPU/Core to swap rapidly between tasks. If the swapping is done frequently, it looks like all the tasks are executing at the same time. Time-slicing works by dividing up a CPU’s processing power in 'packets' of time (called **‘time slices’**) – typically between 1-10mS. The operating system then allocates these packets to programs or threads that are able to make use of them. Interrupts via **Real Time Clock.** Swapping is called **context-switching**. Good for interactive processes (gui).

**Single tasking** (linear).

**Multi tasking** (Concurrent). Running several pieces of system code at same time. **Pros:** Utilizes CPU power. Flexible, distributed across servers. Scalable, run multiple times as separate processes (Eg. 10 word docs). **Cons:** decomposing/architecting into appropriate concurrent tasks. Sharing data. Synchronizing tasks. Debugging + Testing. **Parallel programming** (specialized branch of concurrent programming). Focuses on design of algorithms that make software run faster in a multi-core system. Eg. Quicksort.

**Speed-up** (Speedup = Execution time (1 core) / Execution time (p cores)). **Ahmdahl’s law** (Speedup = 1/(1-proportion that can be sped up)), Can also add proportion / speedup, 1/((1-p)+p/s). Types of multitasking:

* Pseudo-multitasking: Faking it. Infinite loop of fast functions creates illusion that all tasks run concurrently. Simple. No complex OS needed. Inter-task communication using global vars. Task synchronization is automatic. But needs to be brief. No task protection from crash. No priority.
* Multiple CPU on backplane. VME bus. High performance and expensive. Not flexible and vulnerable to power supply failure.
* Distributed systems: Nodes connected to network. Delay to speed of network, size of packet, number of nodes.
* Cluster/grid computing. Useful for parallel processing large volumes of data. Eg. Web serving.
* Heterogenous CPU architectures: dissimilar CPUs doing different things on one interconnect (eg cpu, gpu, dps, FPGA)
* Homogenous CPU architectures: Quad core cpus from intel. Multi-processing.

**Granularity:** degree of parallelism that exists in a system. (Process splitting) Crude <-> (Thread splitting, parallel executing code) fine grained dust. May have communication + synchronization problems. Race conditions.

**Process:** program. Its own variables. Process boundary.

* Thread: Code within program that runs independently. Access to same global variables in same process.

**Data pool**. Shared memory. No pointers allowed b/c process boundary duh.

**Pipeline:** sequential FIFO between two programs. Ensure that data is not overwritten or re-read by accident. Read is destructive

**Mailbox:** Message queue. FIFO. Linked list. Usually 32 bit unsigned int in windows.

**Socket:** provides API (application programming interface) to facilitate communication between client and server. Basically a pipeline but over a network. Port #. IP. The **server** creates socket, opens port on socket, listens for incoming requests on port, accepts client connection, communicates with client using low-level R/W ops. The **client** creates socket, connects to server using IP and port, communicates using R/W similar to pipeline, closes connection. **RPC** (remote procedure calls), basically proxy function. **CORBA** (Common Object Request Brokerage Architecture), Microsoft makes DCOM. And .Net.

* **CORBA / DCOM:** model for communication between objects in a distributed object architecture, that is, objects existing either on the same machine or on different machines located anywhere in the world. In distributed computing, an object request broker (ORB) is installed middleware that sits between client and server) which allows program calls to be made from one computer to another via a computer network, providing location transparency through remote procedure calls.
* CORBA protocols allows applications to be developed in such a way that a process is able to locate any object in the world and talk to it (i.e. send it messages) as if it were talking to that object directly i.e. as if the object were part of the same program running on the same machine.
* First the developer uses an “IDL” compiler to create proxy (or dummy) objects locally on both the client and server machines with interfaces identical to the real objects on the remote machine (same function names, arguments, returned data etc.)
* The client application communicates with the local proxy object, which in turn communicates with an ORB (object request broker – a bit like a telephone directory). Through the ORB, objects written in different languages on different machines with different operating systems can communicate with each other.
* The proxy object on the client machine then communicates with the proxy on the server (using technologies like sockets/RPC). Thus it is the proxy on the server that carries out the actions of the client ON the server.

**Mutual Exclusion**: access to non-sharable resources that can be shared only if all tasks co-operate so that one has access at any one time. **Spin locking** is bad, wastes CPU time a lot.

**Atomic**: indivisible. **TAS (Test and Set**). Reads value of flag, sets value to 1 regardless, indicates to program previous value of flag without releasing control of the memory address and data buses. If TAS ==0, it was free. If tas == 1, try again later. **Atomic\_flag** only works between threads, not across process boundaries.

**Event:** To unblock threads that are already waiting. Self-resetting.

**Condition:** When unblocking threads for a finite amount of time, and controlling resetting. **Rendezvous:** or semaphore for counting or queuing. When specified set of threads wait for each other. No need for detecting or signaling

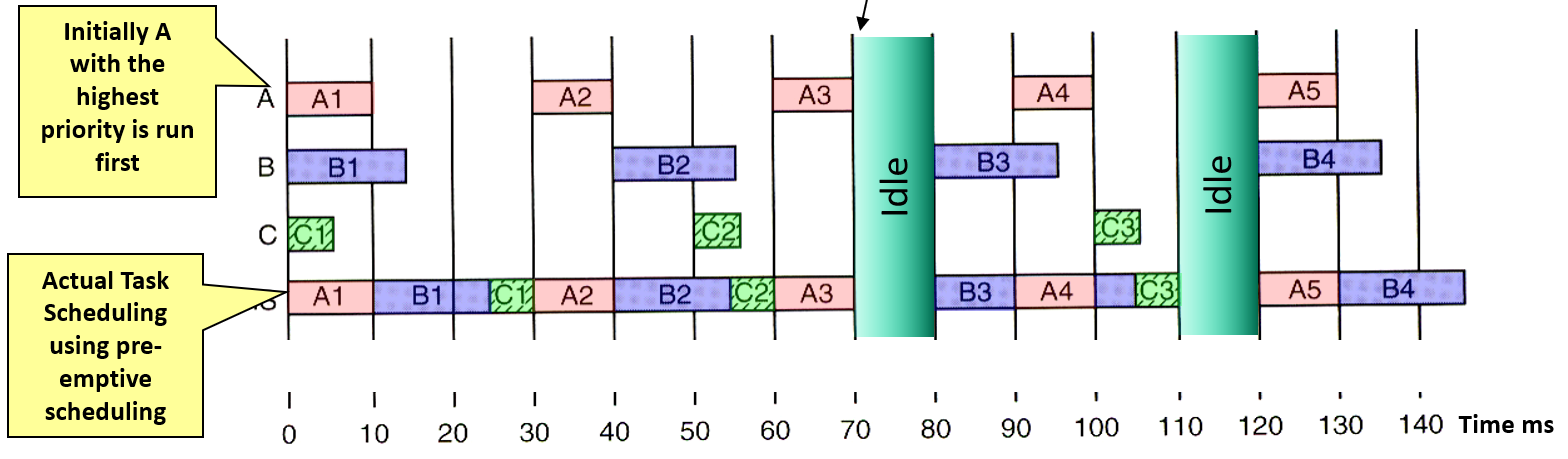
**Producer/Consumer**: problems: data corruption (updating datapool at same time being read). Thread synchronization (producer/consumer doesn’t know the other has updated datapool), producer can generate faster than consumption, or consumption faster than produced. Single consumer single producer Solution: Two semaphores, PS1 (init 0), CS1 (init 1). Producer while loop: cs1 wait, update datapool, ps1 signal. Consumer while loop: ps1 wait, read datapool, cs1 signal.

* One producer several consumers: add more cs and ps into producer loop. Each consumer singly has a ps wait, read, cs signal
* Multiple producers, one consumer: Polling, using read() on semaphore. Multiple threads within consumer to deal with each producer with each datapool.

**Reader/Writers:** Reader thread + writer thread. Multiple readers. One Writer.<shared\_mutex>, exclusive lock: lock, try\_lock, unlock. Shared locking, lock\_shared, try\_lock\_Shared, unlock\_shared. However, need to prevent writer starvation. When there is already one or more reader threads inside the resource and a writer thread arrives and wants access, then further reader threads should be blocked from immediate admittance to the resource until after the writer has been admitted and left. In this way, a writer thread only has to wait for the current reader threads already active within the resource, to leave before it gains access. It does not have to wait for reader threads that come along and try to gain admittance at the same time the writer is waiting.

**Sleeping Barbers:** One barber, one chair, lots of waiting chairs. If no customers, sleep. Otherwise customer wakes up barber. If barber is cutting, and new customer, they will wait in chair. If waiting chairs are full, customer leaves in disgust. 2 semaphores set to 0 init.

**Deadlock**: Circular dependence on waiting for resource.

* **Four conditions**: **1**. At least one resource must be non-sharable. **2**. Pre-emption is not possible, once resource is given to thread, cannot be taken back. **3**. One thread must be holding at least one resource while requesting use of another, held by another thread. **4**. Circular wait dependency.
* **Directed graph:** T1 -> R1 -> T2. T1 Waiting for R1 but blocked. T2 allocated R1, not blockers.
* **4 solutions**: Ignorance (ostrich). Avoidance. Detection + Recovery. Prevention.
  + Ostrich. Do nothing.
  + Avoidance 1: Resource request ordering. Acquire multiple resources in same strict order. Needs designers to agree and be aware of other threads.
  + Avoidance 2: Give up resources. After timeout, just give up. Cons: starvation. If it gives up a resource it already has acquired, it might not get any back, if another thread acquires in between.
  + Avoidance 3: Treat multiple resources as one. Just use one mutex for all resources. Cannot acquire individual resource, needs all even if doesn’t need all. Requires co-operation between all threads.
  + Detection + Recovery: uses OS to break deadlock. Rarely used in real-time due to safety and timing concerns.
    - Pre-emptive take back. OS forcibly takes back resource from process and gives it to another. However, no guarantee that the resource will be in the same state later.
    - Roll back. Checkpoints or snapshots established. Rewind to earlier state to repeat operations. However, complex procedure that can involve undoing successful actions. Can lead to further deadlock.
    - Process killing. Brute force killing process which releases all its resources. Killing database or e-commerce app is bad.
  + Prevention: **Bankers algo**. Rarely used in real-time due to small number of fixed processes and shared resources which can be easily avoided with careful programming. Limitations: processes have to declare upfront at runtime the number and type of resource requested.
    - OS prevents deadlock by always maintaining itself in a safe state where deadlock cannot happen. Good for batch programming. But bad for interactive or real-time systems.
* **Dining Philosopher’s problem**. Five philosophers with own rice bowl and two chopsticks shared with neighbours. Chopsticks are resources. Easiest way is to use wait\_for\_multiple\_objects kernel call to find them open and get them all at once. This avoids deadlock. But starvation can still happen, as a process may never find all the resources it needs that are free. Create array of handles (2) with mutexes. Then use kernel call.
* **Starvation**: infinite waiting for resource to be open.
* **Priority Inversion**. Common problem. High prio task blocked by low prio task which has resource lock… Needs to wait for low prio. Solve **using priority inheritance**, blocker task gets super high prio.
* **Rate Monotonic Scheduling (RMS)**. Period (time between successive triggering). Deadline (Time available to produce response). Compute time (CPU time needed) Deadline < trigger period, but often we can assume trigger period == deadline.
  + Schedule based on **shortest deadline = highest prio**. Priority: inverse of trigger period.
  + Two methods, math and draw it out (timeline graph).
  + Math: Sum of all tasks (compute time / deadline or period) < n(2^(1/n)-1). Must be strictly less than 100% though. Fairly obvious fact that if the tasks between them, consume more than 100% of CPU time, at some point they will fail to meet their deadlines
    - 1 task: 100%. 2 tasks: 82.8%. 3 tasks: 77.97%. inf: 69.3 (log\_e(2)).
    - Harmonic <= 100% good enough. Eg. 20, 40, 80 ms tasks. Total is <= 100% so OK. Reason why harmonic tasks can go up to 100% is that the pattern of scheduling is repetitive. That is, task scheduling requirements for each task do not vary w.r.t each other after all tasks have completed one deadline period (at 80ms when C repeats).
  + 

Part B

* Traditional SW Life Cycle (iterative):
  + System Engineering – Overview of system (Where, what kind, environment, timeline, budget, performance requirements)
  + Analysis – Identifying WHAT needs to happen, not HOW it will happen (Who are users/stakeholders, their functional requirements. Simple models of user interaction with system and info processing that takes place within it). Document functionality.
  + Design, code - Focusing on HOW (implementation decisions), turning analysis models to detailed design models, then to code. Answer questions related to approach, UI design, database, hw interface, etc.
  + Testing
  + Maintenance
* Iterative: iterations of waterfall model. Ship as partial release.
  + Pros: immediate short term feedback on what was done correctly and wrong. Fix bugs and misunderstandings. Get paid for each release, good for cash flow. Staff fully utilized (some testing, some coding, some analyzing)
* UML: Unified Markup Language
  + For large teams, 6+ months, complex systems (Often mandated in safety critical applications eg medical devices, power stations, military)
  + Use case diagram. User interactions.
  + Class diagram:
  + Sequence diagram: interaction of objects
  + State charts: basically state transition diagrams
* Waterfall method: suitable for engineering projects where requirements are specified up front and when cost of change is prohibitive. But not flexible enough for software.
* Agile: stories (As a USER I want to DO THIS so that I can ACHIEVE THAT). Good for projects when requirements and technology are not well understood or are changing. Cons: project management is non-existent, documentation is source code itself. Dependent that sw will be ready when it’s ready and will cost whatever it costs.
* Sprint/Release (1-3 weeks), Stake holders / users, talk to domain experts, Requirements analysis
  + Design, implement, test a version.
  + Scrum master. What they have done since last scrum, what intends to be done before next scrum, obstacles. Retrospectives after a sprint. What went well, what did not.
* MoSCoW (Cost estimation)
  + Must have. Essential. Cannot function without these features
  + Should have. Important. After must haves.
  + Could have. Nice to have but system could work without.
  + Won’t have. Wish list features.

**Use case**: process or procedure, describing in detail a user’s interaction with the system (eg. A library) for a specified, identifiable purpose (Eg. Borrowing book.)

Use case **scenario**: specific instance of a use case that is played out between actor and system at run-time. Document using text descriptions, flow charts, structured pseudo-code and algorithms, formulas, lookup tables, decision trees.

**Includes** (dependency)

**Extends** (optional behaviour)

**Architecture:** set of components from which a complex system is composed. Blueprint for design. Once implemented, very difficult and costly to change.

**Structural Model.**

1. Identify run time obj. Create/invent run time objects which the system will be composed of
2. Delegate roles and responsibilities to objects. Eg. Linked list. Role: store and retrieve data. Responsibility: perform the following tasks: insert, delete, find.
   1. Roles and responsibilities eventually mapped to member functions
3. Identify Obj data. (member variables)
4. Identify relationships with other obj
5. Identify relationship types with other obj
6. Identify object multiplicities in relationship (how many of floors does skyscraper have)

**Object identification techniques:**

1. Underline use-case nouns. (Actors, objects of interest, uninteresting objects, attributes / data of an object)
2. Verb analysis (Roles and responsibilities). Actions, things that happen. Infer behaviour.
3. Identify collabs between obj.
4. Identify active objects. Responsible for initiating actions in the system.
5. Identify passive objects that provide service to other obj.
6. Identify real-world objects from problem domain. (Gas station -> gas transaction. Library -> books / members. Banks -> bank account)
7. Identify transaction obj. (Eg. Button, elevator controller, request transport, floor request obj.)
8. Identify physical objects in system.
9. Identify persistent obj.
10. Identify GUI visual elements.