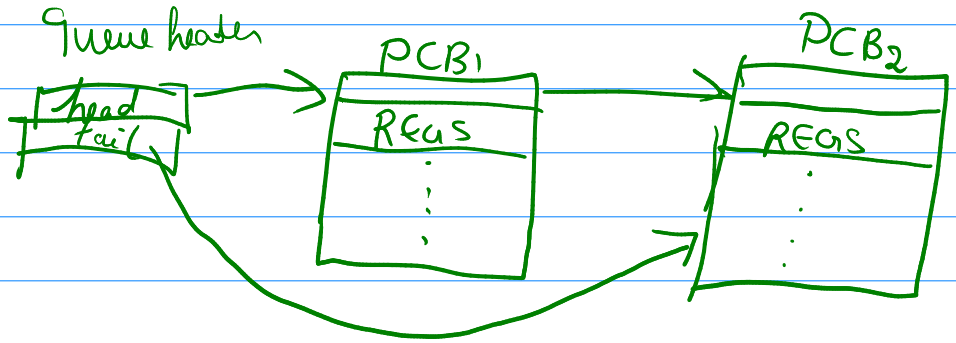
Scheduling

Job queue - set of all process

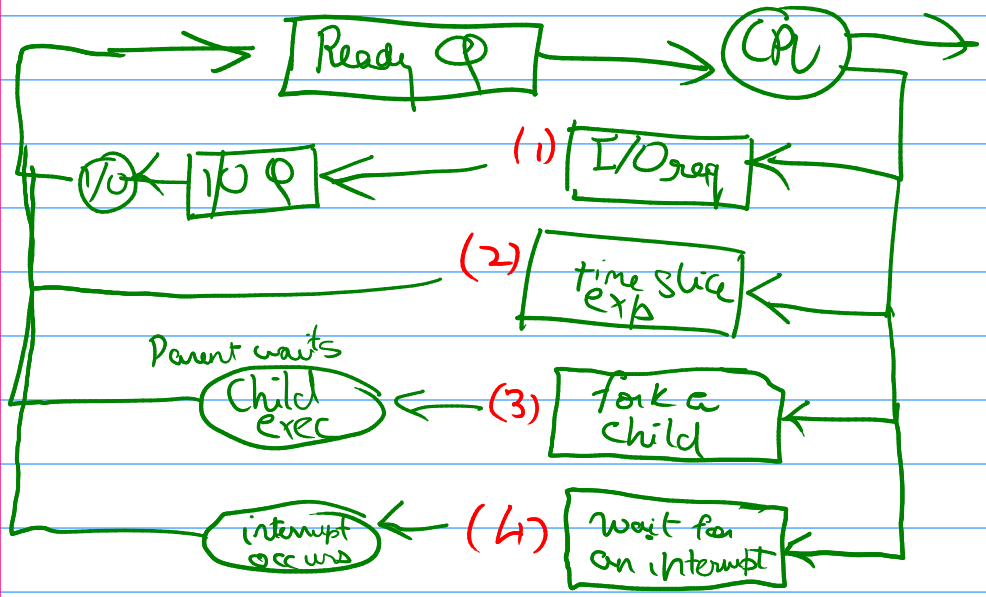
Ready queue - Proc. Ready & waiting to execute
↳ Linked list

Device queue -
Every I/O device has device queue

Ready Queue



Q Diagram



Short term
Sometimes only in a sys
→ Scheduler
→ context switching very fast
(milliseconds)
→ invoked

Long term
Which process
→ Should be in ready state?
→ Seconds, minutes invoked in

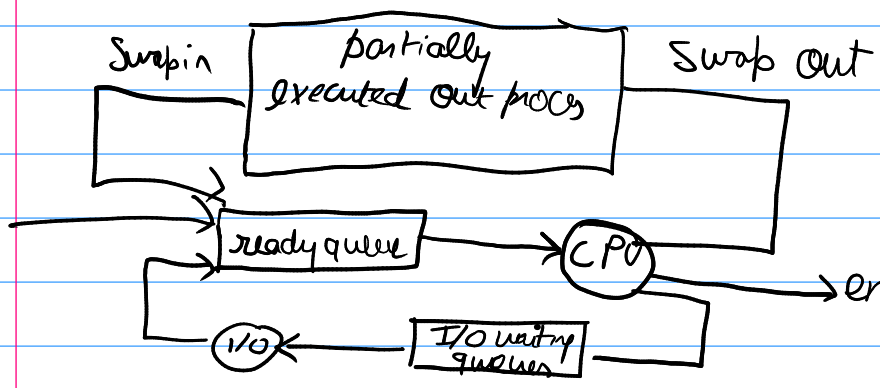
Processes

I/O bound
spends more time doing I/O than computations

CPU bound
Spends more time doing computations
few very long CPU bursts

good mix e.g.

Medium term
Middle level Scheduler



remove process from main mem & store on disk (Swap)

Threads

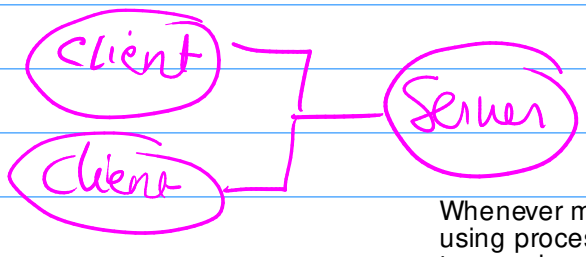
Processes - overheads

Degree of multiprogramming initially determined by no. of processes executing @ a time.

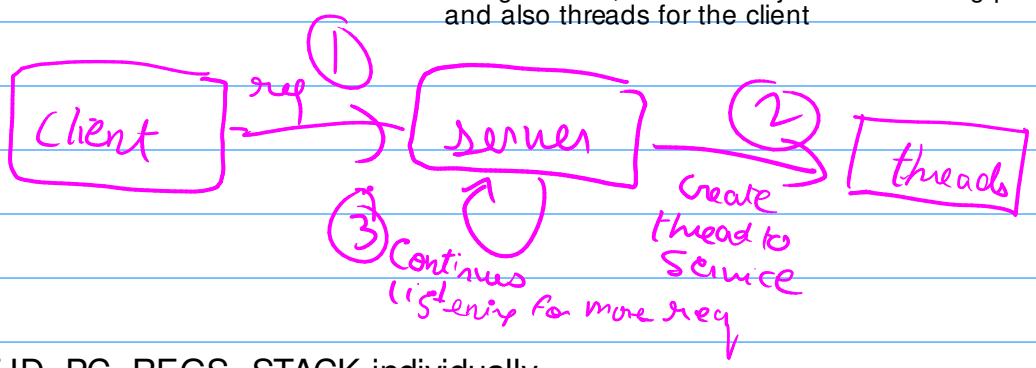
⇒ context switching between threads is easier

One house to the other (processes) — One room to another (threads)

Threads share certain resources with other threads in the same process

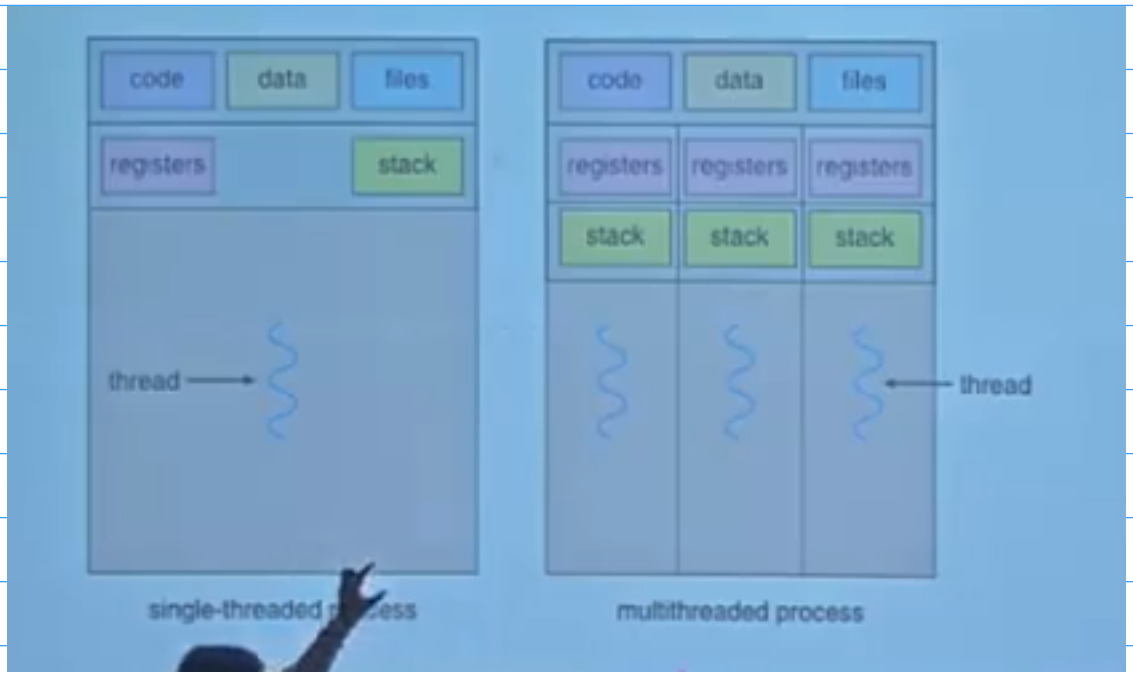


Whenever multiple clients request the server using processes, there'll be each process for a client too much overhead + one listening process
Using threads, we can use just one listening process, and also threads for the client



T-ID, PC, REGS, STACK individually

code, data, OS resources like open files are **shared**



Each thread can do different tasks, so you need different stacks for that

Benefits

responsiveness - may allow continued execution if part of process is blocked.
Eg. UI in interactive components, if one is displaying other is waiting for button
Resource sharing - shares resources of processes easier than shared mem and message passing
Economy - cheaper than creating process, thread swirching lower overhead
Scalability - one process one cpu, but with threads, each thread itself can get a cpu

Latest - two threads per core

Multicore Programming

multicore - >1 core on same cpu chip,
intra-chip communication is faster than inter-chip communication

Programming
1) identifying tasks and map them to core
2) simpler computations with high freq
don't need to be partitioned into a task and block a core
it can be in the same core since it is simple - balance

3) Data splitting while sharing (or not splitting xD) b/w threads and
4) Data consistency needs to be maintained

5) Testing and debugging this shit ...