* How long should a timeslice be?

⇒ We have to keep in mind to balance benefits and overhead.

sc1: → for I/O bound

sc2 → for CPU "

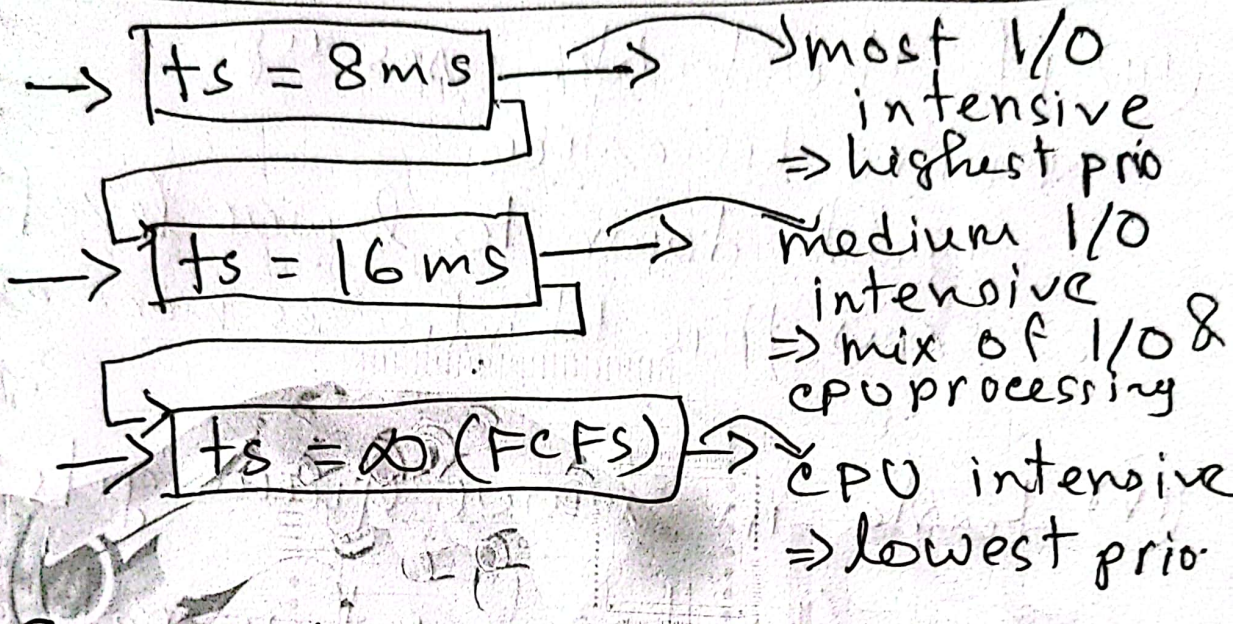
* discuss these scenario with example

* Runqueue DS: If we want I/O and CPU Bound tasks to have different timeslice values, then...

→ same runqueue, check type to apply different timeslice

→ two different structures each runqueues associated a different kind of policy suitable for CPU vs. I/O boun

=> Dealing with different timeslice values. (multiqueue DS)



⊕ timeslicing benefits provided for I/O bound tasks

⊕ timeslicing overheads avoided for CPU bound tasks

But How do we know if a task is CPU or I/O intensive.

=> How do we know how I/O intensive a task is?

=> History based heuristics.

(i) tasks enter topmost queue

(ii) if task yields voluntarily
⇒ 'good choice', keep the task at this level

(iii) if the task uses up entire timeslice
⇒ push down to lower level (as this it indicates more cpu

(iv) task in lower queue gets priority boost when releasing CPU ~~to~~ due to I/O waits.

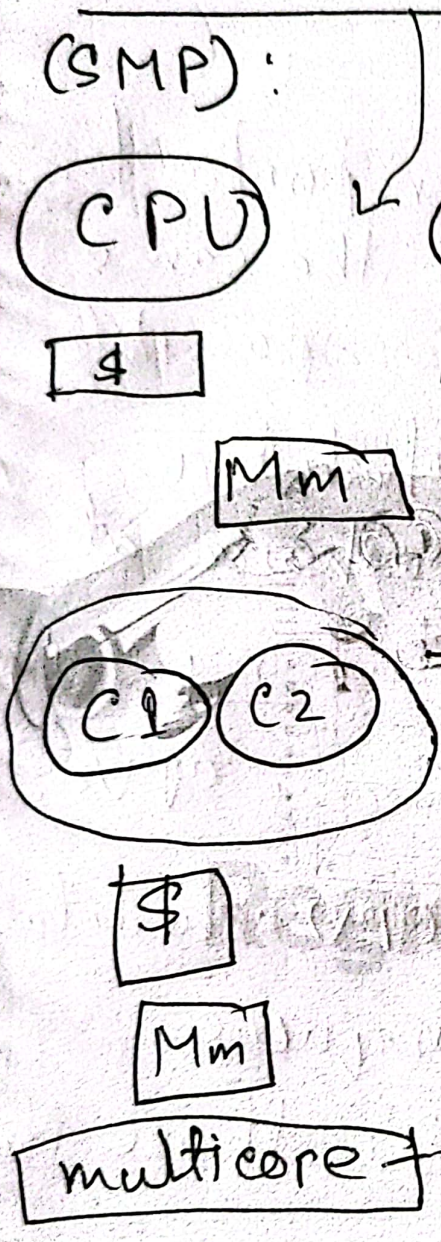
this updated DS called multi-level feedback queue (MLFQ)

— Fernando. Corbato, turing Award winner.

MLFQ != Priority queue.

*Scheduling on Multi-CPU system

① Shared Memory Multi-processor (SMP):



→ CPUs and their private/on-chip caches (L1, L2)

→ last level cache (LLC)

→ Memory (DRAM)

→ CPU multiple cores

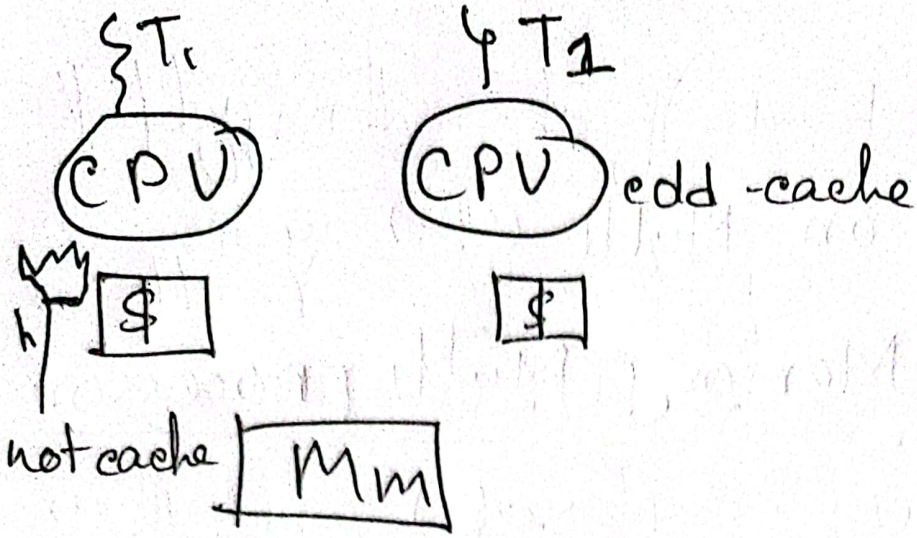
→ each cores have private L1, L2...

→ shared LLC

→ memory

→ Fig

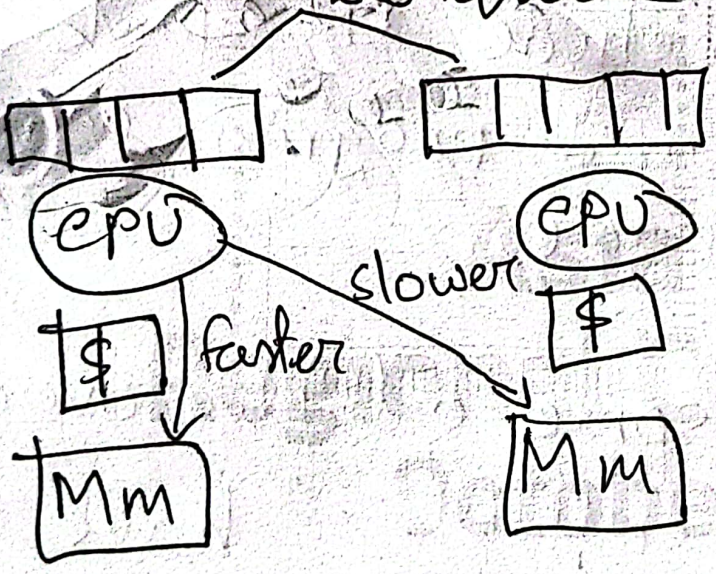
purpose: we want to achieve with a scheduling on multi-CPU system executed before because it is more likely that it's caches will be hot.



⇒ cache - affinity important
⇒ keep tasks on the same CPU as much as possible.
⇒ hierarchical scheduler architecture, where at the top a load balancing comp. divides the tasks among CPUs. and then a per CPU scheduler with a per-CPU runqueue repeatedly schedules those tasks on a given CPU as much as p

* Non-uniform memory access (NUMA)

- multiple memory nodes
- memory nodes closer to a "socket" of multiple processors (subset of CPU)
- access to local mm node faster than access to remote mm node. (lb aware



Note: the mms are interconnected via some sort of interconnect
→ for intel it's QPI

⇒ keep tasks on CPU closer to mm node where their state is

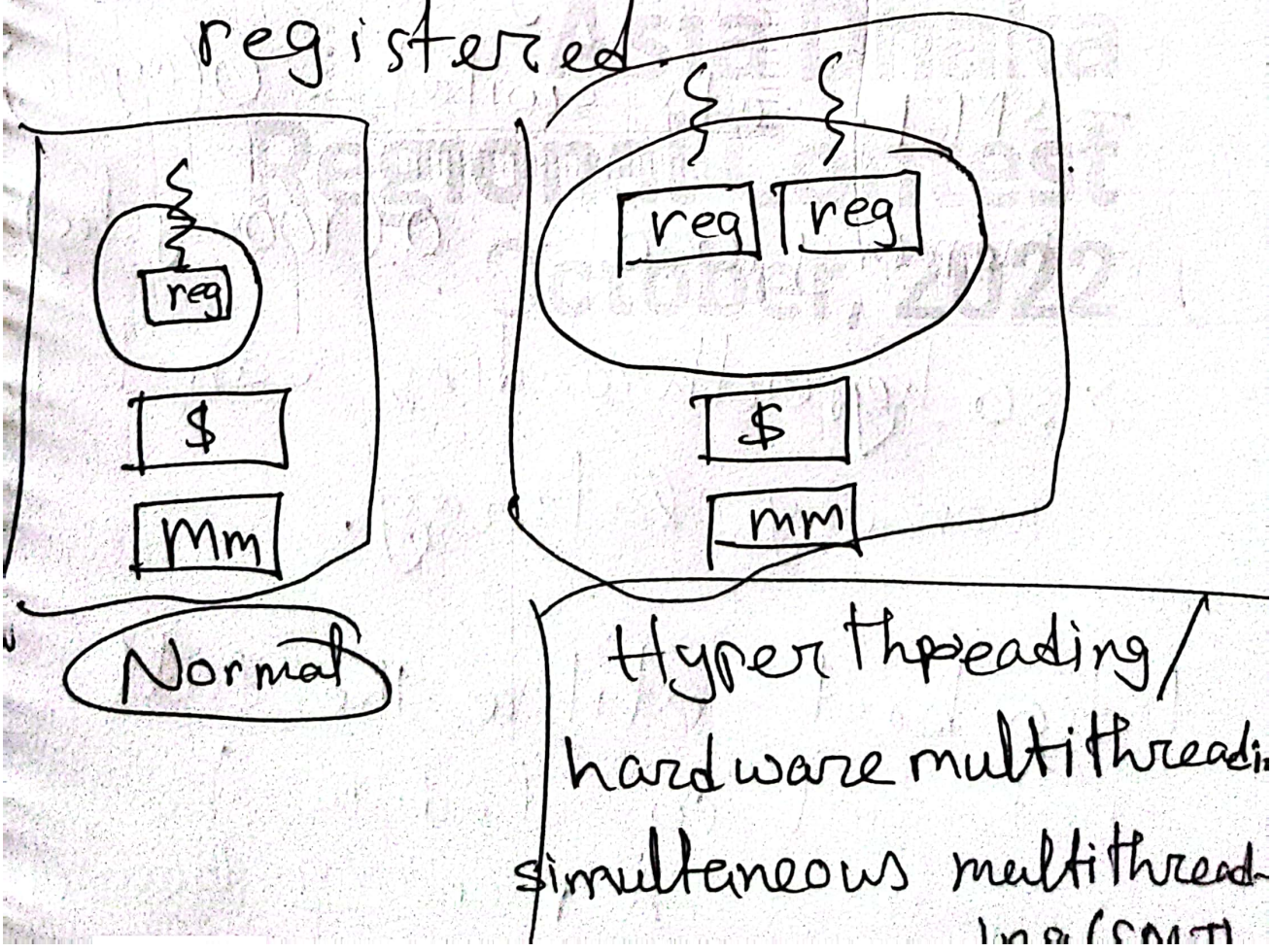
⇒ known as NUMA-aware scheduling.

* Hyperthreading (SMT)

→ The reason why we have to context switch among threads is because the CPU has one set of registers to describe the active execution context, the thread that's currently running on the CPU. These include the stack pointer and program counter in particular.

→ to hide some of the overhead with ctx switching, have CPU that have multiple set of registers can describe the context of a seperate thread. This mechanism is refered to as hyperthreading.

⁂ There's still just one CPU, so only one of these threads can execute at a particular moment of time. However the ctx swi. is very fast between these threads as the CPU only needs to switch from one set of registers to another set of registered



=) The OS sees each hardware context as a seperate virtual CPU onto which it can schedule threads by loading the reg. with the thread context concurrer

=> if (t_idle > 2 * t_ctx_switch)
Then context switch to hide latency

- SMT. ctx-switch - O(cycles)
- memory load - O(100 cycles)

=> so hyperthreading can hide memory latency.

* to best utilize the full potential co-schedule compute- and memory-bound

This allows

→ avoid / limit contention on processor pipeline

→ all components CPU & memory well utilized