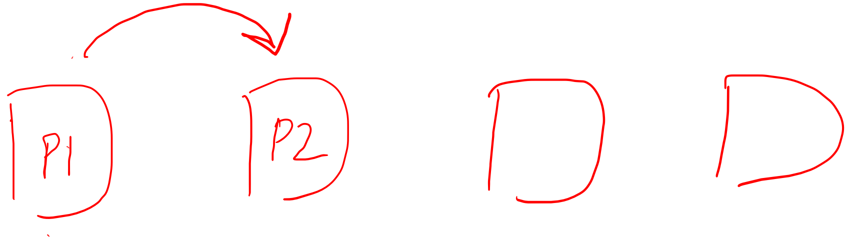
Design and Engineering of Computer Systems

Lecture 9:
CPU scheduling policies

Mythili Vutukuru

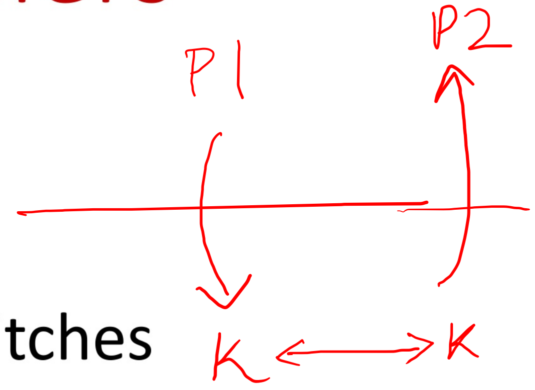
IIT Bombay

OS scheduler



- OS scheduler schedules process on CPU cores
  - One process at a time per CPU core
  - Processes and kernel-level threads are scheduled similarly
- Scheduling policy: which one of the ready/runnable processes should be run next on a given CPU core?
  - Mechanism of context switching (save context of old process in its kernel stack/PCB and restore context of new process) is independent of policy
- Simple scheduling policies have good theoretical guarantees, but not practical for real operating systems
  - Real-life schedulers are very complex, involve many heuristics

Preemptive vs. non preemptive schedulers

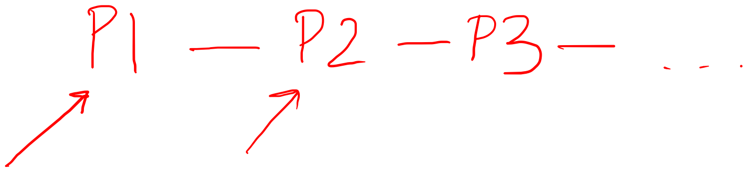


- When is the OS scheduler invoked to trigger a context switch?
  - Only when a process is in kernel mode for a trap
- Non-preemptive scheduler performs only voluntary context switches
  - Process makes blocking system call
  - Process has exited or has been terminated
- Preemptive scheduler performs involuntary context switches also
  - Process can be context switched out even if process is still runnable/ready
  - OS can ensure that no process runs for too long on CPU, starving others
- Timer interrupts: special interrupts that go off periodically to trap to OS
  - Used by OS to get back control, trigger involuntary context switches
- Modern systems use preemptive schedulers
  - Process can be context switched out any time in its execution

Goals of CPU scheduling policy

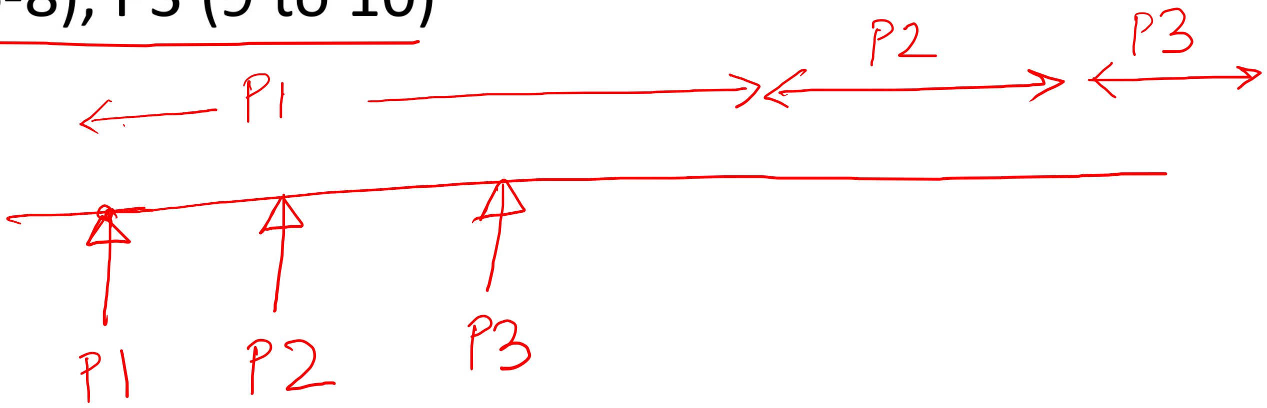
- Maximize utilization: efficient use of CPU hardware
- Minimize completion time of a process (time from process creation to completion)
- Minimize response time of a process (time from process creation to first time it is run)
  - Important for interactive processes
- Fairness: all processes get a fair share of CPU
  - Can account for priorities also
- Low overhead of scheduling policy
  - Scheduler does not take too long to make a decision (even with large #processes)
  - Scheduler does not cause too many context switches (~1 microsecond to switch)

Simplest policy: First In First Out



- Newly created processes are put in a FIFO queue, scheduler runs them one after another from queue
- Non-preemptive: process allowed to run till it terminates or blocks
  - When process unblocks, the next run is separate “job”, added to queue again
- Problem: short processes can get stuck behind big processes
  - Response time of interactive processes may be poor
- Example schedule: P1 (1-5), P2 (6-8), P3 (9 to 10)

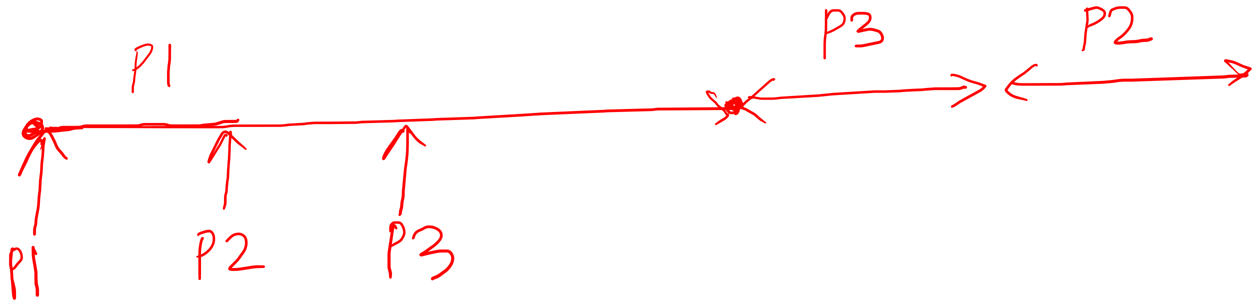
Process	CPU time needed (units)	Arrives at end of time unit
P1	5	0
P2	3	1
P3	2	3



Shortest Job First (SJF)

- Assume CPU burst of a process (amount of time a process runs on CPU until termination/blocking) is known
- Pick process with smallest CPU burst to run next, non-preemptive
  - Store processes in a heap-like data structure, extract process with min CPU burst
- Provably optimal average completion time when all processes arrive at the same time
  - But short processes that arrive late can still get stuck behind long ones
- Example schedule: P1 (1-5), P3 (6-7), P2 (8-10)

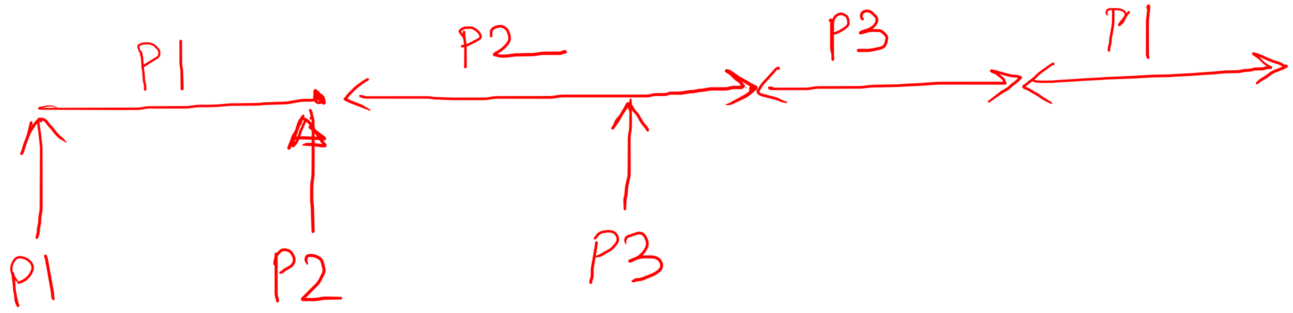
Process	CPU burst	Arrival time
P1	5	0
P2	3	1
P3	2	3



Shortest Remaining Time First (SRTF)

- Preemptive version of SJF
- A newly arrived process can preempt a running process, if CPU burst of new process is shorter than remaining time of running process
  - Avoids problem of short process getting stuck behind long one
- Example schedule: P1 runs for 1 unit, P2 (2-4), P3 (5-6), P1 (7-10)

Process	CPU burst	Arrival time
P1	5 4	0
P2	3 1	1
P3	2	3

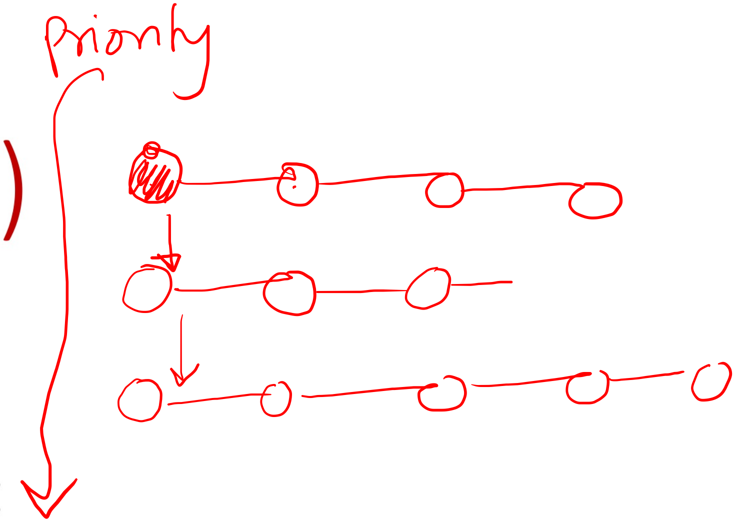


Round Robin (RR) / Weighted Fair Queueing (WFQ)

P1 —— P2 —— P3. . . . .

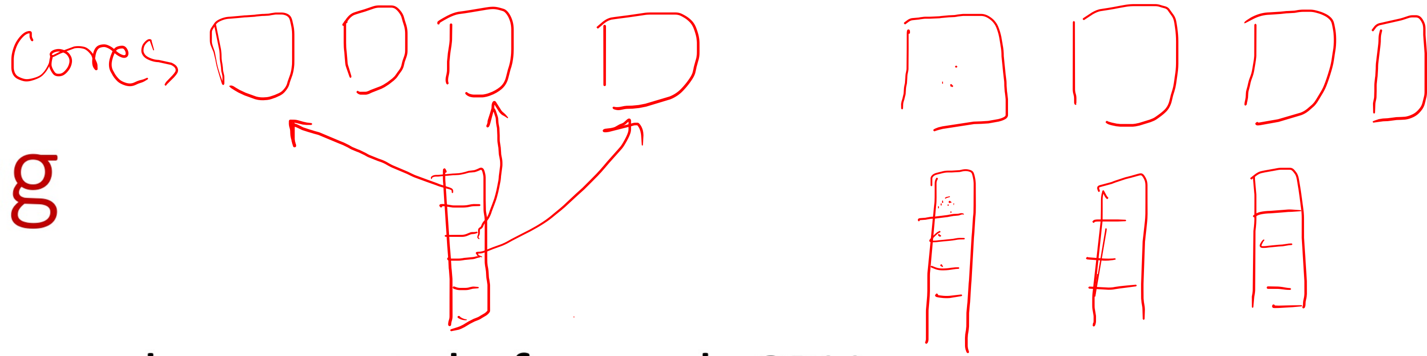
- Processes are run one after the other for a time slice each, fairly sharing CPU
- Can also assign different weights or priorities to processes (can be set by users)
  - Time slice will be in proportion to the weight or priority
- Preemptive policy: timer interrupt allows enforcing context switch after time slice
- Good for fairness and response time
  - Time slice should not be too big, for good responsiveness
- Real life schedulers may not be able to enforce time slice exactly
  - What if timer interrupt is not exactly aligned with time slice?
  - What if process blocks before its time slice?
- Practical modification: keep track of run time of process, schedule process that has used least fraction of its fair share
  - Compensate excess/deficit running time in future time slices
- Linux scheduler is a variant of weighted fair queueing

Multi-level feedback queue (MLFQ)

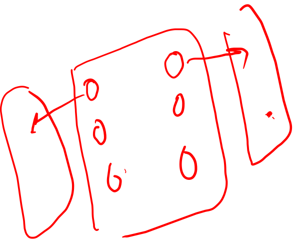


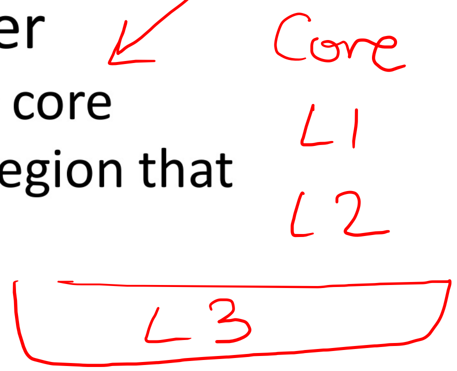
- Multiple queues, one for each priority level
  - Schedule processes from highest priority queue to lowest
  - Use round robin scheduling for processes within same priority level
- Priority set by user or OS, but decays with age
  - Job that uses up its time slice at a priority level goes to lower priority level
  - Why? Ensures short I/O-bound processes that don't use their full slice get priority over long CPU-bound processes that use their fair share all the time
- Periodically reset all processes to highest priority level to avoid starvation of low priority or CPU-bound processes

Multicore scheduling



- Scheduling decision needs to be made separately for each CPU core
- Do we bind a process to a particular CPU core always, or do we let a process run on any CPU core that is free?
  - Is the queue of ready processes common to all cores, or maintained per core?
- Ensuring a process runs on the same core as far as possible is better
  - Cache locality: process-related memory is likely to be in CPU caches of the core
  - In NUMA systems, better to run process on core that is close to the RAM region that has process memory
  - Per-core queue of ready processes avoids synchronization across cores
- But, we must be flexible too
  - If CPU core overloaded, some of its processes must move to another core
  - Load balancing across cores to ensure uniform workload distribution





Summary

- In this lecture:
  - Scheduling policy: goals ✓
  - Example policies: FIFO, SJF, SRTF, RR, WFQ, MLFQ ✓
  - Considerations for multicore systems ✓
- Think of examples of scheduling policies in real life systems. Do you see queues in daily life? What kind of scheduling policies do they use?