CPU Scheduling Policies

Mythili Vutukuru

CSE, IIT Bombay

OS scheduler

PCB keeps the information about the processes running in the CPU. These information includes pid, state, pc value, memory management ifno, and I/O status.

- OS scheduler schedules process on CPU
  - One process at a time per core
  - Multiple processes can run in parallel on multiple cores
- Scheduling policy: which one of the ready/runnable processes should be run next on a given CPU core? Which one should be given priority to run next.
  - Mechanism of context switching (save context of old process in its kernel stack/PCB and restore context of new process) is independent of policy

context switching occurs independent of the policy to choose the next process to run.

- Simple scheduling policies have good theoretical guarantees, but not practical for real operating systems
  - Real-life schedulers are very complex, involve many heuristics

kind of experience

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, but not on every 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

every process should get some fair share.

- Modern systems use preemptive schedulers
  - Process can be context switched out any time in its execution

every process has to give up CPU after certain time of execution and let other processes run in the CPU.

Timer interrupts

- Scheduler (OS) can run only when process is in kernel mode
- What if process never goes to kernel mode? How to perform pre-emptive scheduling?
- Timer interrupts: special interrupts that go off periodically
- Used by OS to get back in control periodically
  - Increment clock ticks, do other book keeping
  - Run scheduling algorithm if required to do involuntary context switch of currently running process
- Hardware support in the form of timer interrupts essential to implementing pre-emptive scheduling policies

Hardware supports the implementation of timer interrupt.

Goals of CPU scheduling policy

- Maximize utilization: efficient use of CPU hardware
- Minimize completion time / turnaround 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 (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
  - That is, if a process comes back after I/O wait, it counts as a fresh CPU burst (CPU burst = the CPU time used by a process in a continuous stretch)
- Example schedule: P1 (1-5), P2 (6-8), P3 (9 to 10)

Process	CPU time needed (units)	Arrives at end of time unit	Execution time slots
P1	5	0	1-5
P2	3	1	6-8
P3	2	3	9-10

Problem with FIFO

- Example: three processes arrive at t=0 in the order A,B,C
- Problem: convoy effect (small processes get stuck behind long processes)
- Average turnaround times tend to be high

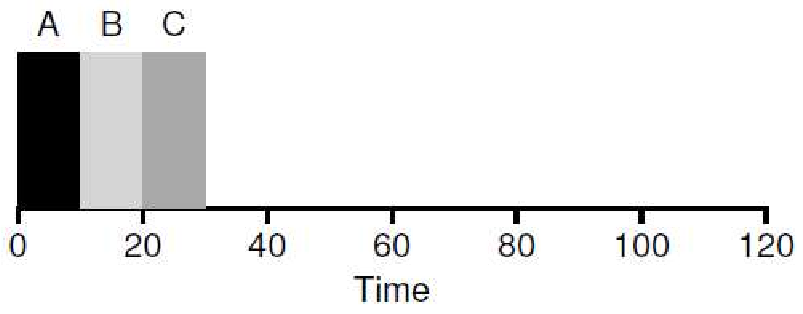


Figure 7.1: **FIFO Simple Example**

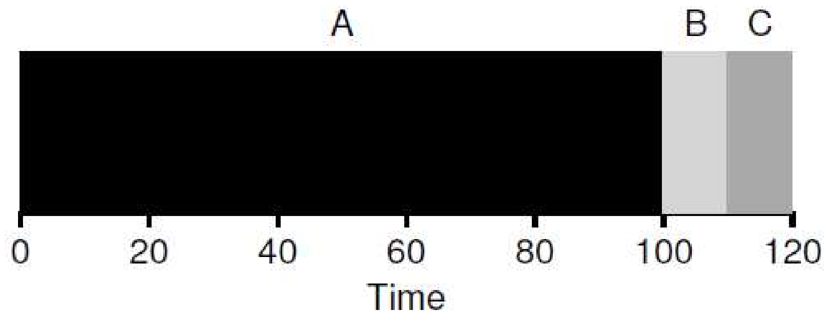


Figure 7.2: **Why FIFO Is Not That Great**

Image credit: OSTEP

Shortest Job First (SJF)

- Assume CPU burst of a process (amount of time a process runs on CPU until termination/blocking) is known apriori
- Pick process with smallest CPU burst to run next, non-preemptive
  - Store PCBs in a heap-like data structure, extract process with min CPU burst
- Example schedule: P1 (1-5), P3 (6-7), P2 (8-10)

Process	CPU burst	Arrival time	Execution time slots
P1	5	0	1-5
P2	3	1	8-10
P3	2	3	6-7

Problem with SJF

- Provably optimal when all processes arrive together
  - Theoretically guaranteed to have the lowest average turnaround time across all policies (under certain assumptions)
- SJF is non-preemptive, so short jobs can still get stuck behind long ones.

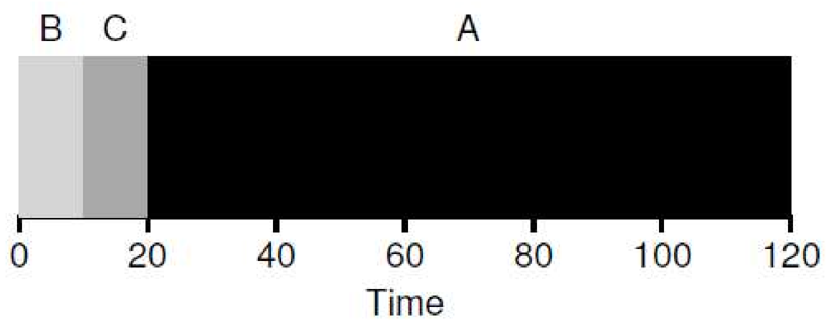


Figure 7.3: **SJF Simple Example**

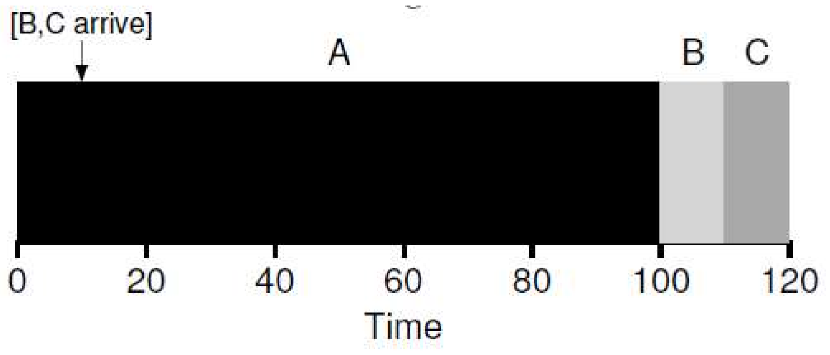


Figure 7.4: **SJF With Late Arrivals From B and C**

Image credit: OSTEP

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	Execution time slots
P1	5	0	1, preempted, then 7-10
P2	3	1	2-4
P3	2	3	5-6

Round Robin (RR)

- Every process executes for a fixed quantum slice
  - Slice not too small (to amortize cost of context switch)
  - Slice not too big (to provide good responsiveness)
- Preemptive policy
  - Timer interrupt used to enforce periodic scheduling
- Good for response time, fairness
- Bad for turnaround time
- xv6 is a simple RR scheduler

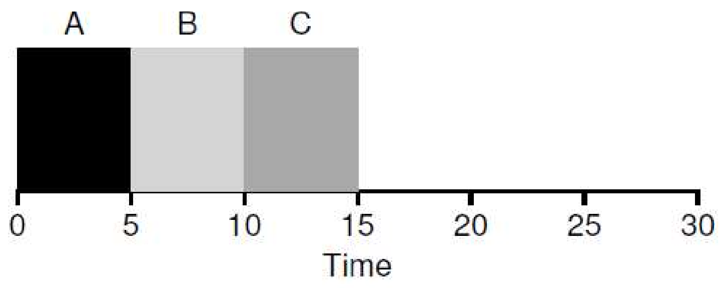


Figure 7.6: **SJF Again (Bad for Response Time)**

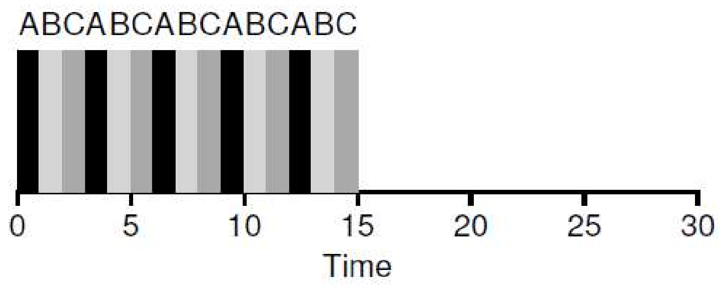


Figure 7.7: **Round Robin (Good for Response Time)**

Image credit: OSTEP

Weighted Fair Queueing (WFQ)

- Round robin with different weights or priorities to processes
  - Decided by scheduler or can be set by users
  - Time slice will be in proportion to the weight or priority
- 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
  - CFS (completely fair scheduler) uses red-black trees to keep track of the fair share run time of processes, schedules the one with least run time

Multi-level feedback queue (MLFQ)

- Another practical algorithm, with realistic assumptions
- What problem does it solve?
- Ideally, we want to optimize for turnaround time like SJF, give priority to shorter jobs (less time on CPU, more time on I/O)
  - But we do not know running time beforehand
- Also ensure low response time like round robin
- How to optimize for both turnaround time and response time without knowing job size apriori?

Overview of 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
- What is priority? Set by user but decays with age. Why?

Because it already has run for time.

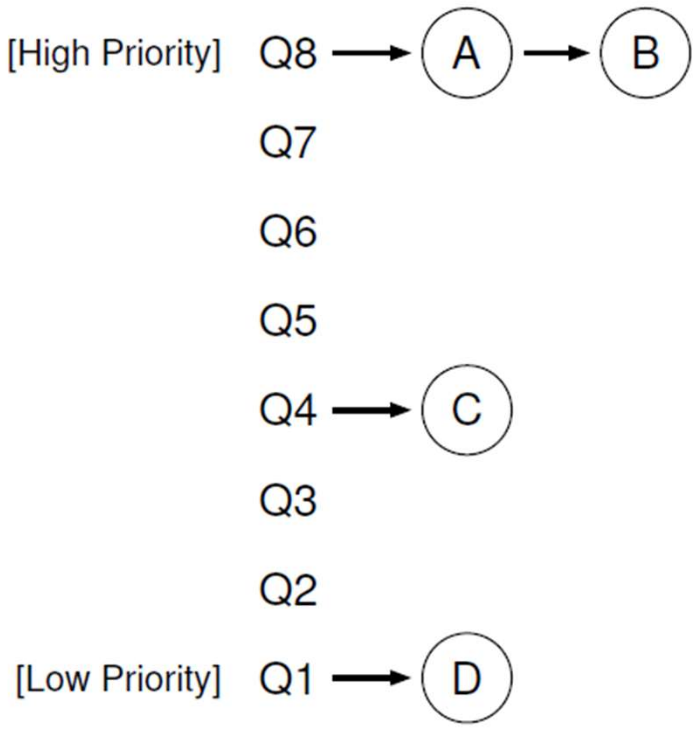


Figure 8.1: **MLFQ Example**

Image credit: OSTEP

Priority decays with age

- Job that uses up its time slice at a priority level goes to lower priority
- Why? Ensures short I/O-bound processes get priority over long CPU-bound processes, but without knowing CPU burst apriori

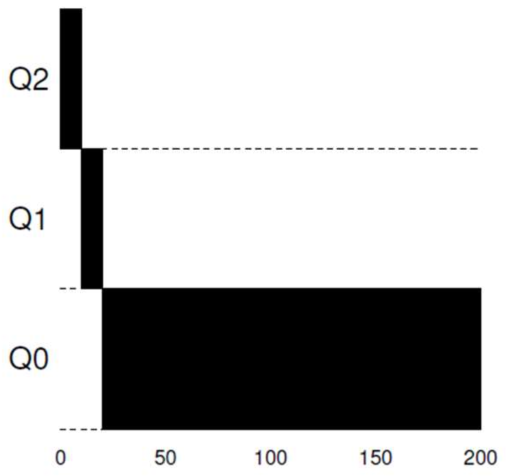


Figure 8.2: **Long-running Job Over Time**

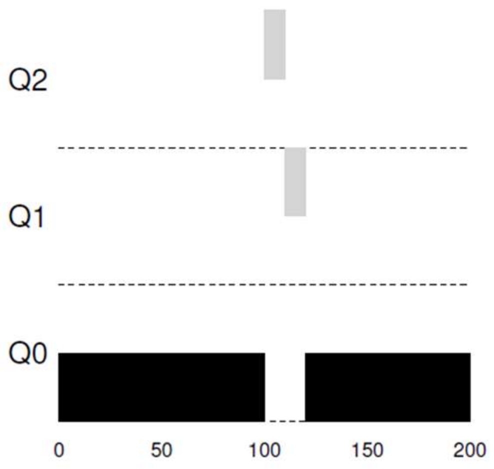


Figure 8.3: **Along Came An Interactive Job**

Image credit: OSTEP

Avoiding starvation

- Periodically reset all processes to highest priority level to avoid starvation of low priority or CPU-bound processes

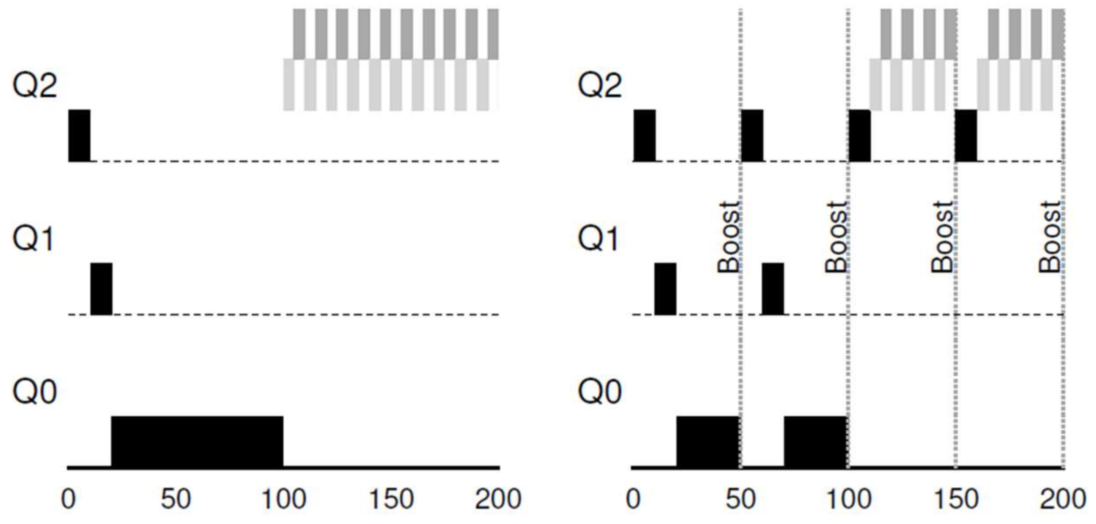


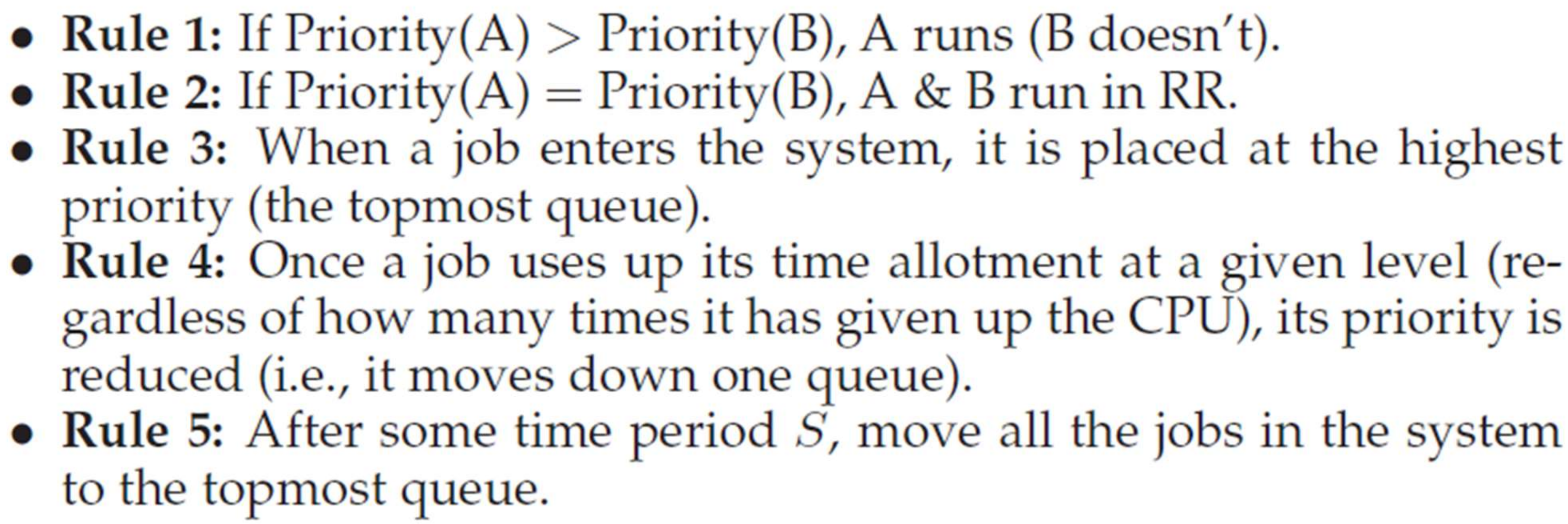
Figure 8.5: **Without (Left) and With (Right) Priority Boost**

Image credit: OSTEP

Other considerations

- What if a job always gives up CPU just before its time slice ends? Do we keep it always at highest priority?
- We can consider total time spent by a job at a given priority level, not time in just one execution
- Time slice can vary with priority level
  - Longer time slice for lower priority long running jobs
- How to parameterize? How many priority levels? What is time slice?
  - No easy answers, must be tuned based on workload
  - Every OS comes with some default parameters

MLFQ summary



- **Rule 1:** If Priority(A) $>$ Priority(B), A runs (B doesn't).
- **Rule 2:** If Priority(A) $=$ Priority(B), A & B run in RR.
- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

Image credit: OSTEP

So, what have we achieved with MLFQ?

- Prioritize short, I/O bound processes over long, CPU bound processes
- Why? System works best when short processes don’t have to wait for too long behind longer processes (that’s why SJF is optimal!)
- How is this done? I/O bound processes will stay at highest priority level
- We do this without knowing the run times apriori, so practical to work in real life scenarios
- Real life schedulers cannot make assumptions on knowing run times etc, but still try to achieve good properties of theoretically optimal schedulers

Multicore scheduling

- Scheduling decision needs to be made separately for each 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?
- Ensuring a process runs on the same core as far as possible is better
  - Avoids coordination overheads across cores, better CPU cache performance
- 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