Sistemas Operativos: Process Scheduling

Pedro F. Souto (pfs@fe.up.pt)

March 6, 2016

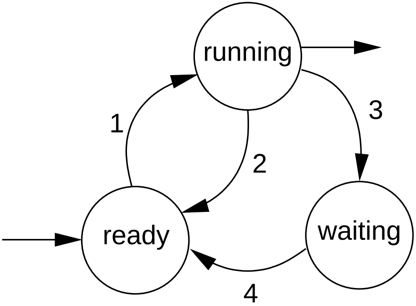
Roadmap

Process Scheduling

Further Reading

Process Scheduling

Problem: when there are more than one process ready for executing, which one should execute?



Solução: the OS, more specifically the *scheduler*, executes a scheduling algorithm to decide
I.e., the scheduler determines to which process the CPU should be assigned.

Note So far we have focused on the **mechanisms**, i.e. the **how**, used by the OS to implement the process abstraction, we will now look into **policies**, i.e. the **when** and **what**

Workload Assumptions

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. Once started, each job runs to completion
4. All jobs only use the CPU (i.e., they do not perform I/O)
5. The run-time of each job is known

Job This is usually a process

Note All these assumptions are unrealistic. We'll drop them one-by-one

Scheduling Metric

Metric Parameter used to measure something.

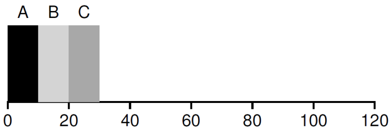
Turnaround time, $T_{turnaround}$ time interval between the **arrival**, T_{arr} of a process and its **completion**, T_{comp}

$$T_{turnaround} = T_{comp} - T_{arr}$$

- Many other metrics make sense for scheduling. E.g. **fairness**
- This may not be the best metric for interactive workloads.

First Come First Served (FCFS) or FIFO

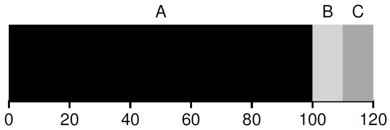
- As its name suggests, the first job to arrive is the first to be served, i.e. according to its order in some queue.
- E.g. 3 jobs, A, B and C, each with 10 time units execution time, arrive in that order:



$$T_A = 10 \qquad T_B = 20 \qquad T_C = 30$$

$$T_{av} = \frac{T_A + T_B + T_C}{3} = 20$$

- But what if we drop the first assumption and $T_A = 100$?



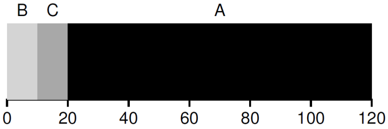
$$T_A = 100 \qquad T_B = 110 \qquad T_C = 120$$

$$T_{av} = \frac{T_A + T_B + T_C}{3} = 110$$

- This is known as the **convoy** effect. How can we avoid it?

Shortest Job First (SJF)

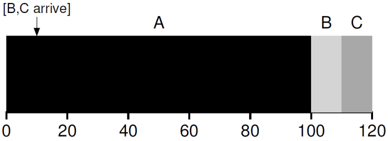
- To avoid penalizing the shortest jobs, these are run before longer jobs



$$T_A = 120 \qquad T_B = 10 \qquad T_C = 20$$

$$T_{av} = \frac{T_A + T_B + T_C}{3} = 50$$

- Can be shown to be **optimal**, given the assumptions.
- But what if we drop the second assumption and B and C arrive 10 time units after A?



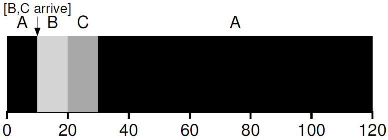
$$T_A = 100 \qquad T_B = 110 \qquad T_C = 120$$

$$T_{av} = \frac{T_A + T_B + T_C}{3} = 110$$

- How can we improve this?

Shortest Time to Completion First (STCF)

- Need to drop 3rd assumption, i.e. allow the arrival of a process to **preempt** a running process
  - SJF is **non-preemptive**



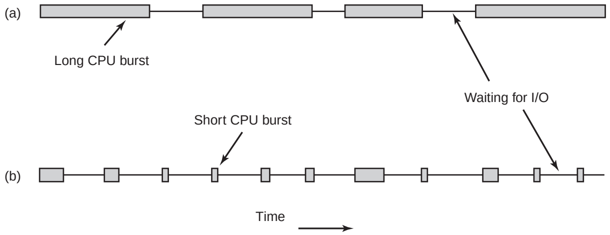
$$T_A = 120 \qquad T_B = 10 \qquad T_C = 20$$

$$T_{av} = \frac{T_A + T_B + T_C}{3} = 50$$

- Can be shown to be **optimal**, given the assumptions.

What about I/O? (1/2)

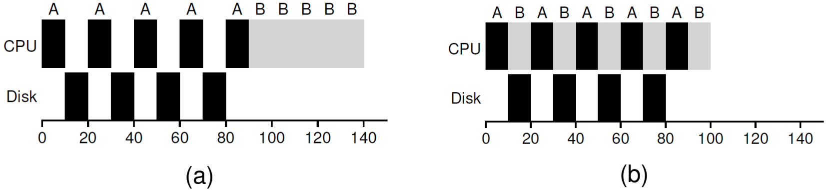
- During their execution, processes (and *threads*) alternate
  - bursts of CPU usage
  - with periods waiting for I/O
- Depending on the relative size of these periods, processes can be classified as: *CPU-bound* (a) or *IO-bound* (b)



- Therefore, let's drop assumption 4
  - All jobs only use the CPU (i.e., they do not perform I/O)

What about I/O? (2/2)

- What to do if a process blocks when it initiates an I/O operation?
- If it does not schedule another process to run, the CPU will be wasted (Fig. a)



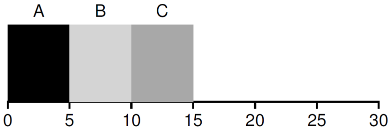
- Thus, the scheduler should (Fig. b):
  1. Schedule a ready process to run, when another blocks on an I/O operation
  2. Pick the best process to run, when completion of an I/O operation unblocks a process
- This allows overlapping processing and I/O improving a system's global performance

Response Time: a new metric

- The turnaround time may be a good metric for early **batch** computer systems, but not so for **time sharing** systems
- For these systems interactive performance is also important, and a better metric for that is **response time**, T_{resp}, the time interval between the arrival of a job and the first time it is schedule to run, T_{run}:

$$T_{resp} = T_{run} - T_{arr}$$

- Neither SJF nor STCF are particularly good with this metric
  - Often they run jobs until completion before starting to run other jobs for the first time



$$T_A = 0 \quad T_B = 5 \quad T_C = 10$$

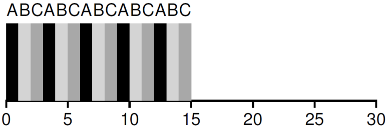
$$T_{av} = \frac{T_A + T_B + T_C}{3} = 5$$

- The response time would be much worse, if the execution time was 100 rather than 10 time units

Round-Robin (RR)

- RR runs each job for a **time-slice/(scheduling) quantum** rather than until completion (drop 5th assumption)
- When a process runs until the end of its slice, the scheduler:
  1. puts it at the end of the run queue;
  2. picks the first job in the run queue

  thus jobs run in turns, or round-robin
- The length of a time-slice must be multiple of the timer-interrupt period



$$T_A = 0 \qquad T_B = 1 \qquad T_C = 2$$

$$T_{av} = \frac{T_A + T_B + T_C}{3} = 1$$

- By reducing the time-slice we can improve the response time
  - At the cost of additional context-switches, which are not free
- Also, RR turnaround time tends to be poor:
  - RR is inherently **fair**, and this usually hurts performance

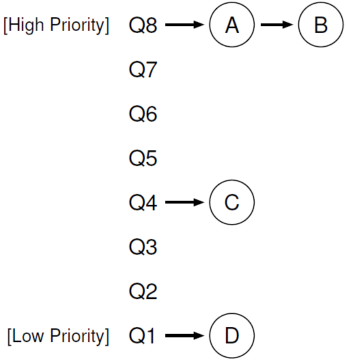
Real Workloads

- Most real workloads are a mix of CPU-bound and interactive jobs
- Round-robin cannot be tuned for both:

  CPU-bound jobs the quantum should be large
  Interactive jobs the quantum should be short
- Furthermore, most of the times the OS does not know whether a job is CPU-bound or interactive
  - Many jobs alternate between phases in which they are CPU-bound and phases in which they are not
- Need a scheduler that is able to handle this kind of workload

Multi-Level Feedback Queue (MLFQ)

- Use multiple **queues** each with a different priority level
- Each job has a priority, which may **change** with time
- Basic scheduling rules:
  1. If priority(A) > priority(B), then A runs
  2. If priority(A) = priority(B), then A and B run in RR

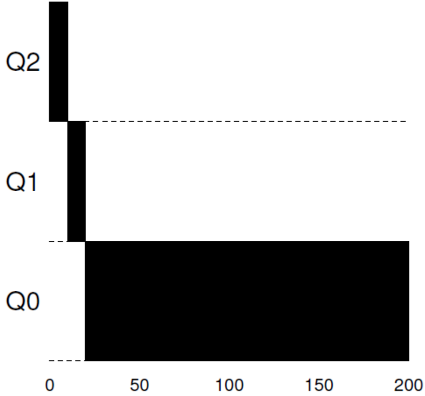


Key How should the priority of a job change?

3. A job starts at the highest priority
4a. If a job uses up its entire time slice, its priority is reduced
4b. If a job gives up the CPU before its time slice is up, it keeps its priority

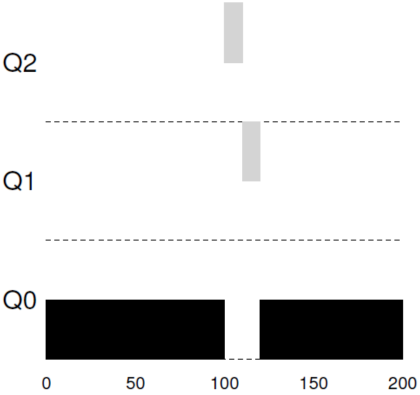
MLFQ Examples

Single Long-running job



- After $n - 1$ time-slices, where n is the number of levels, a long-running will sink into the lowest priority queue

Then, a short job comes

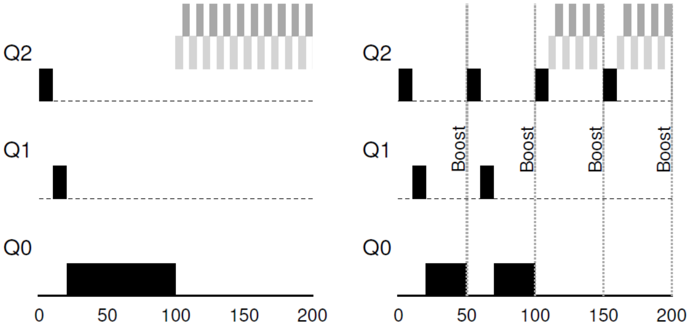


- An incoming short job preempts the longer-running job

MLFQ: Avoiding Starvation

Issue: Starvation If short running jobs keep arriving, long running jobs may **starve**, i.e. will not get a chance to run

Fix: Periodically boost the priority of a long running job to the highest priority

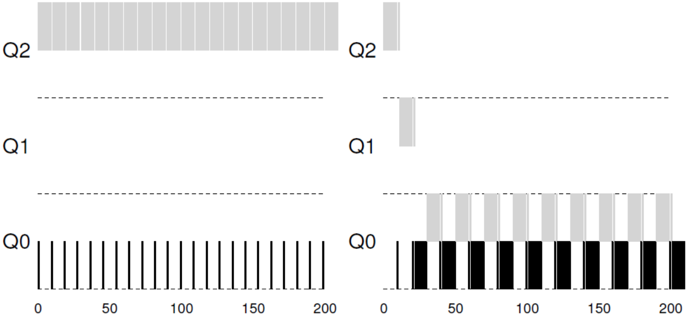


Rule 5. After some time *S*, move all the jobs in the system to the topmost queue

MLFQ: Avoiding Gaming

Issue: Gaming A process may give up the CPU just shortly before the end of the time slice, and therefore keep the highest priority

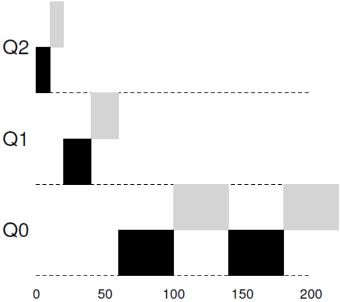
Fix: Accumulate CPU cycles across time-slices, and reduce priority when it uses up its allotment for the current level.



Rule 4: Once a job uses up its time alllotment at a given level, its priority is reduced

MLFQ Conclusion

1. If priority(A) > priority(B), A runs
2. If priority(A) = priority(B), A and B take turns (RR)
3. When a job enters the system, it is placed at the highest priority (topmost queue)
4. Once a job uses up its time allotment at a given level, its priority is reduced
5. After some time period S, move all the jobs to the topmost queue



Variations Some MLFQ schedulers allow for different time-slices at different priority levels

Criticism Too many **voodo constants** to tune

Roadmap

Further Reading

OSTEP

- Scheduling: Introduction
- Scheduling: MLFQ

Sistemas Operativos

- Secções 4.1, 4.2, 4.3 e 4.4

Modern Operating Systems, 2nd. Ed.

- Sections 2.5 e 10.3

Operating Systems Concepts

- Sections 5.1, 5.2 , 5.3 (and 5.4)