Operating Systems

CPU Scheduling

Thomas Ropars

thomas.ropars@imag.fr

Équipe ERODS – LIG/IM2AG/UJF

2015

References

The content of these lectures is inspired by:

- The lecture notes of Renaud Lachaize.
- The lecture notes of Prof. David Mazières.
- The lectures notes of Arnaud Legrand.
- *Operating Systems: Three Easy Pieces* by R. Arpaci-Dusseau and A. Arpaci-Dusseau

Other references:

- *Modern Operating Systems* by A. Tanenbaum
- *Operating System Concepts* by A. Silberschatz et al.

Agenda

Agenda

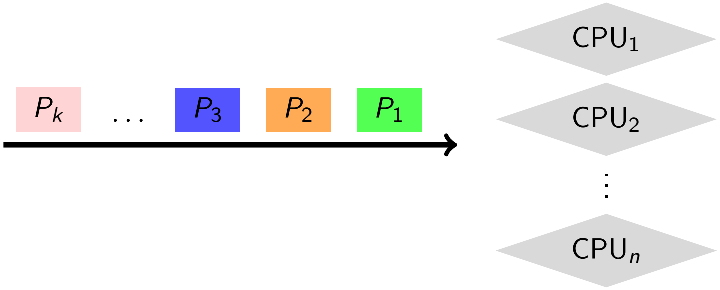
The problem

Textbook algorithms

Multi-level feedback queues

Multiprocessor scheduling

CPU scheduling



The scheduling problem:

- The system has k processes ready to run
- The system has $n \geq 1$ CPUs that can run them

Which process should we assign to which CPU(s)?

About threads and multiprocessors

Thread scheduling

When the operating system implements kernel threads, scheduling is applied to threads

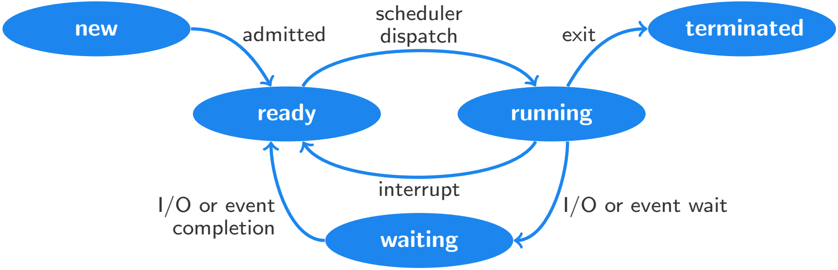
- The following slides discuss process scheduling but also applies to kernel threads.

Multiprocessors

Having multiple CPUs available to schedule processes increases the complexity of the scheduling problem

- In a first step, we consider scheduling on a single CPU

Process state



Process state (in addition to new/terminated):

- Running: currently executing (or will execute on kernel return)
- Ready: can run, but kernel has chosen a different process to run
- Waiting: needs external event (e.g., end of disk operation, signal on condition variable) to proceed

When to schedule?

Which process should the kernel run?

- If non runnable, run idle loop, if a single process runnable, run this one
- If more than one runnable process, must make scheduling decision

When is a scheduling decision taken?

1. A process switches from running to waiting state
2. A process switches from running to ready state
3. A process switches from new/waiting to ready state
4. A process exits

Note that early schedulers were non-preemptive (e.g., windows 3.x). It means that no scheduling decision was taken until the running CPU was explicitly releasing the CPU (case 1, 4 or 2 with yield()).

Preemption

A process can be preempted when kernel gets control. There are several such opportunities:

- A running process can transfer control to kernel through a trap (System call (including exit), page fault, illegal instruction, etc.)
  - May put current process to wait – e.g., read from disk
  - May make other processes ready to run – e.g., fork, mutex release
  - May destroy current process
- Periodic timer interrupt
  - If running process used up time quantum, schedule another
- Device interrupt (e.g., disk request completed, or packet arrived on network)
  - A previously waiting process becomes ready
  - Schedule if higher priority than current running process

Context switching

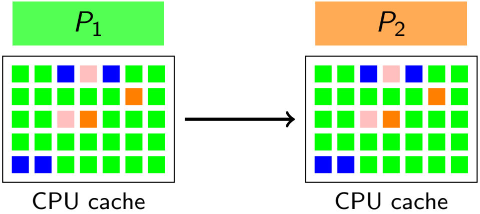
Changing the running process implies a context switch. This operation is processor dependent but it typically includes:

- Save/restore general registers
- Save/restore floating point or other special registers
- Switch virtual address translations (e.g., pointer to root of paging structure)
- Save/restore condition codes (flags)
- Save/restore program counter

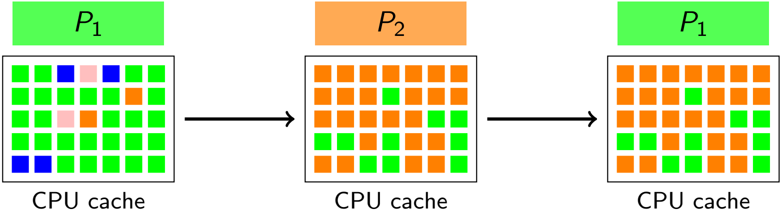
A context switch has a non negligible cost:

- In addition to saving/restoring registers, it may induce TLB flush/misses, cache misses, etc. (different working set).

Context switch cost: cache misses



Context switch cost: cache misses



Scheduling criteria

Main performance metrics:

- Throughput: Number of processes that complete per time unit (higher is better)
- Turnaround time: Time for each process to complete (lower is better)
- Response time: Time from request to first response (e.g., key press to character echo) (lower is better)

Secondary goals:

- CPU utilization: Fraction of time that the CPU spends doing productive work (i.e., not idle) (to be maximized)
- Waiting time: Time that each process spends waiting in ready queue (to be minimized)

Scheduling policy

The problem is complex because their can be multiple (conflicting) goals:

- Fairness – prevent starvation
- Priority – reflect relative importance of processes
- Deadlines – must do x by a certain time
- Reactivity – minimize response time
- Efficiency – minimize the overhead of the scheduler itself

Scheduling policy

The problem is complex because their can be multiple (conflicting) goals:

- Fairness – prevent starvation
- Priority – reflect relative importance of processes
- Deadlines – must do x by a certain time
- Reactivity – minimize response time
- Efficiency – minimize the overhead of the scheduler itself

There is no universal policy

- Many goals – cannot optimize for all
- Conflicting goals (e.g., throughput or priority versus fairness)

Agenda

The problem

Textbook algorithms

Multi-level feedback queues

Multiprocessor scheduling

How to pick up which process to run?

Why not picking first runnable process in the process table?

How to pick up which process to run?

Why not picking first runnable process in the process table?

- Expensive
- Weird priorities (low PIDs have higher priority?)

How to pick up which process to run?

Why not picking first runnable process in the process table?

- Expensive
- Weird priorities (low PIDs have higher priority?)

- We need to maintain a set of *ready* processes

How to pick up which process to run?

Why not picking first runnable process in the process table?

- Expensive
- Weird priorities (low PIDs have higher priority?)

- We need to maintain a set of *ready* processes

What policy?

- FIFO?
- Priority?

First Come, First Served (FCFS)

Description

- Idea: run jobs in order of arrival
- Implementation: a FIFO queue (simple)

Example

3 processes: P_1 needs 24 sec, P_2 and P_3 need 3 sec. P_1 arrives just before P_2 and P_3.

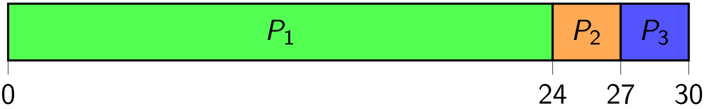
First Come, First Served (FCFS)

Description

- Idea: run jobs in order of arrival
- Implementation: a FIFO queue (simple)

Example

3 processes: P_1 needs 24 sec, P_2 and P_3 need 3 sec. P_1 arrives just before P_2 and P_3.



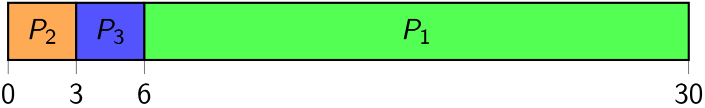
Performance

- Throughput: 3 jobs / 30 sec = 0.1 jobs/sec
- Turnaround Time: $P_1 : 24$, $P_2 : 27$, $P_3 : 30$ (Avg = 27)

Can we do better?

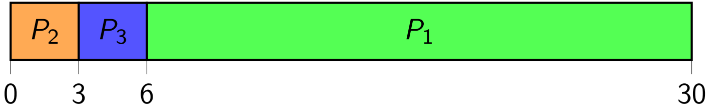
Can we do better?

Suppose we would schedule first P_2 and P_3, and then P_1.



Can we do better?

Suppose we would schedule first P_2 and P_3, and then P_1.



Performance

- Throughput: 3 jobs / 30 sec = 0.1 jobs/sec
- Turnaround Time: $P_1 : 30$, $P_2 : 3$, $P_3 : 6$ (Avg = 13)

Lessons learned

- Scheduling algorithm can reduce turnaround time
- Minimizing waiting time can improve turnaround time and response time

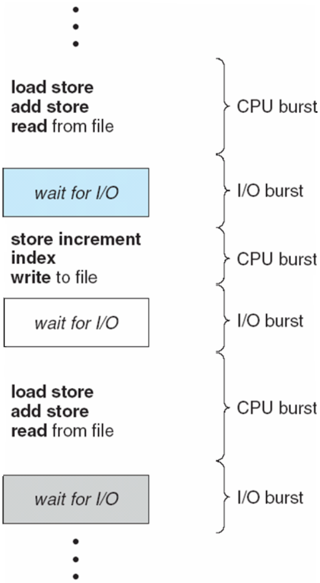
Computation and I/Os

Most jobs contain computation and I/O (disk, network)

- Burst of computation and then wait on I/O

To maximize throughput, we must optimize

- CPU utilization
- I/O device utilization

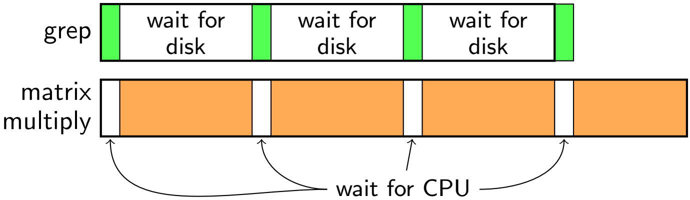


Computation and I/Os

The idea is to overlap I/O & computation from multiple jobs

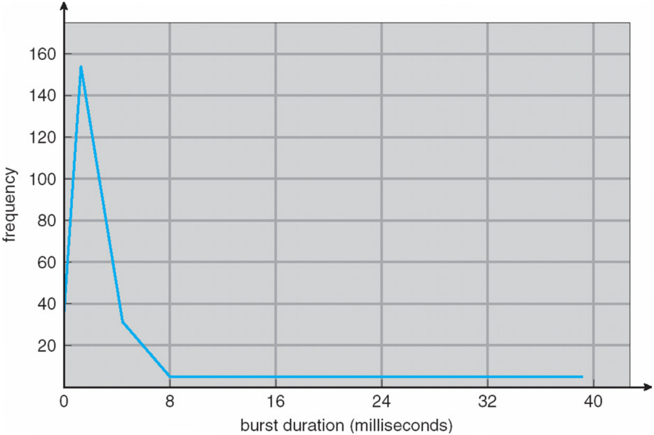
Example

Disk-bound grep + CPU-bound matrix multiply



- With perfect overlapping, the throughput can be almost doubled

Duration of CPU bursts



- In practice, many workloads have short CPU bursts
- What does this mean for FCFS?

Back to FCFS: the convoy effect

Consider our previous example with a disk-bound and a cpu-bound application. What is going to happen with FCFS?

Imagine now there are several IO-bound job and one CPU-bound job . . .

Back to FCFS: the convoy effect

Definition

A number of relatively-short potential consumers of a resource get queued behind a heavyweight resource consumer

Consequences

- CPU bound jobs will hold CPU until exit or I/O (but I/O rare for CPU- bound threads)
- Long period with CPU held and no I/O request issued
- Poor I/O device utilization

Back to FCFS: the convoy effect

Definition

A number of relatively-short potential consumers of a resource get queued behind a heavyweight resource consumer

Consequences

- CPU bound jobs will hold CPU until exit or I/O (but I/O rare for CPU- bound threads)
- Long period with CPU held and no I/O request issued
- Poor I/O device utilization

Simple hack

- Run process whose I/O completed
- New problems?

Back to FCFS: the convoy effect

Definition

A number of relatively-short potential consumers of a resource get queued behind a heavyweight resource consumer

Consequences

- CPU bound jobs will hold CPU until exit or I/O (but I/O rare for CPU- bound threads)
- Long period with CPU held and no I/O request issued
- Poor I/O device utilization

Simple hack

- Run process whose I/O completed
- New problems? What if after the IO it has a long CPU burst?

Shortest Job First (SJF)

Idea

- Schedule the job whose next CPU burst is the shortest

2 versions:

- Non-preemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
- Preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt (Known as the *Shortest-Remaining-Time-First* or SRTF)

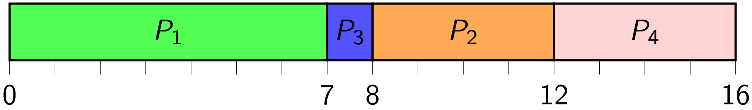
The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes.

- Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process

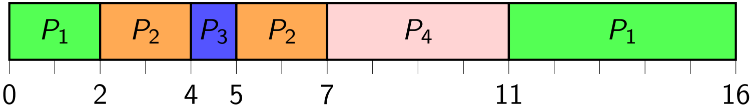
Examples

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- Non-preemptive



- Preemptive



Average turnaround time: FCFS= 8.75; SJF = 8; SRTF = 7

SJF limitations

- It can lead to unfairness or even starvation
  - A job with very short CPU and I/O bursts will be run very often
  - A job with very long CPU bursts might never get to run

SJF limitations

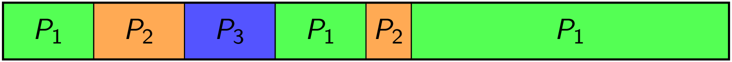
- It can lead to unfairness or even starvation
  - A job with very short CPU and I/O bursts will be run very often
  - A job with very long CPU bursts might never get to run
- In practice, we can't predict the future
- But we can estimate CPU burst length based on the past
  - Exponentially weighted average is a good idea
  - Idea: Predict future burst based on past burst with more weight to recent bursts.
  - (See textbooks for details, e.g., Silberschatz et al.)
  - Hard to apply to interactive jobs

Round Robin (RR) Scheduling

Description

- Similar to FCFS scheduling, but timer-based preemption is added to switch between processes.
- Time slicing: RR runs a job for a time slice (sometimes called a scheduling quantum) and then switches to the next job in the run queue.
- If the running process stops running (wait or terminate) before the end of the time slice, the scheduling decision is taken immediately

Example



Round Robin (RR) Scheduling

Solution to fairness and starvation

- Implement the ready list as a FIFO queue
- At the end of the time slice, put the running process back at the end of the queue
- *Most systems implement some flavor of this*

Advantages

- Fair allocation of CPU across jobs
- Low variations in waiting time even when job lengths vary
- Good for responsiveness if small number of jobs (and time quantum is small)

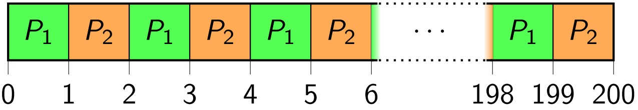
What are the drawbacks?

RR Scheduling: Disadvantages

RR performs poorly with respect to Turnaround Time (especially if the time quantum is small).

Example

Let's consider 2 jobs of length 100 with a time quantum of 1:

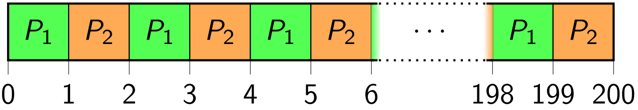


RR Scheduling: Disadvantages

RR performs poorly with respect to Turnaround Time (especially if the time quantum is small).

Example

Let's consider 2 jobs of length 100 with a time quantum of 1:



Even if context switches were for free:

- Avg turnaround time with RR: 199.5
- Avg turnaround time with FCFS: 150

Time quantum

How to pickup a time quantum?

- Should be much larger than context switch cost
- Majority of bursts should be shorter than the quantum
- But not so large system reverts to FCFS
- Typical values: 1–100 ms (often $\sim$ 10 ms)

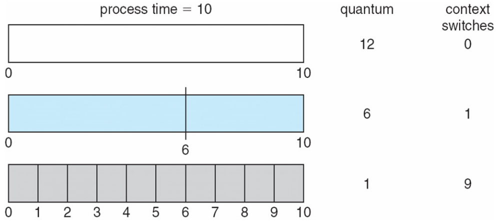


Figure: Impact of time quantum of the number of context switches

Priority scheduling

Principle

- Associate a numeric priority with each process
  - E.g., smaller number means higher priority (Unix)
- Give CPU to process with highest priority (can be done preemptively or non-preemptively)

Note SJF is a priority scheduling where priority is the predicted next CPU burst time.

Problem of starvation

- Low priority processes may never execute
- Solution: Aging – increase the priority of a process as it waits

Agenda

Multi-level feedback queues (MLFQ) scheduling

To be read: Operating Systems: Three Easy Pieces – chapter 8

Goals

- Optimize turnaround time (as SJF but without a priori knowledge of next CPU burst length)
- Make the system feel responsive to interactive users (as RR does)

Multi-level feedback queues (MLFQ) scheduling

To be read: Operating Systems: Three Easy Pieces – chapter 8

Goals

- Optimize turnaround time (as SJF but without a priori knowledge of next CPU burst length)
- Make the system feel responsive to interactive users (as RR does)

Basic principles

- A set of queues with different priority
- At any moment, a *ready* job is in at most one queue

Multi-level feedback queues (MLFQ) scheduling

To be read: Operating Systems: Three Easy Pieces – chapter 8

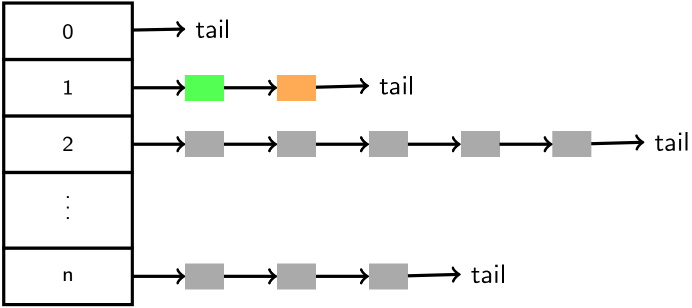
Goals

- Optimize turnaround time (as SJF but without a priori knowledge of next CPU burst length)
- Make the system feel responsive to interactive users (as RR does)

Basic principles

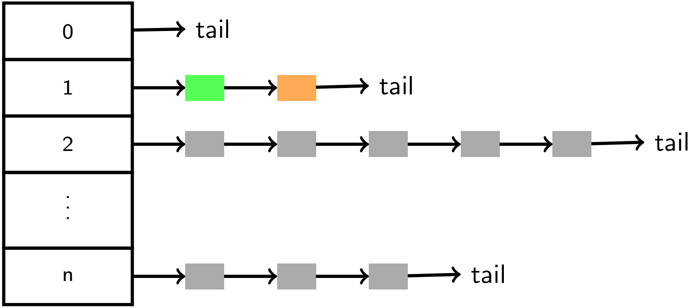
- A set of queues with different priority
- At any moment, a *ready* job is in at most one queue
- Basic scheduling rules:
  - Rule 1: If priority(A) $>$ priority(B), then A runs (B doesn't)
  - Rule 2: If priority(A) $==$ priority(B), RR is applied

MLFQ scheduling



Problem?

MLFQ scheduling



Problem?

- Starvation: Only the processes with the highest priority run
- How to change priorities over time?

MLFQ scheduling: managing priorities

First try

Additional rules

- Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue)
  - Everybody gets a chance to be considered as high priority job (first assume all jobs are short-running).
- Rule 4a: If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue)
  - The priority of CPU-intensive jobs decreases rapidly (this tries to simulate SJF).
- Rule 4b: If a job gives up the CPU before the time slice is up, it stays at the same priority level.
  - Short CPU bursts are typical of interactive jobs, so keep them with high priority for responsiveness
  - More generally, optimize overlapping between I/Os and computation

MLFQ scheduling: managing priorities

Second try

Weaknesses of the current solution

MLFQ scheduling: managing priorities

Second try

Weaknesses of the current solution

- Risk of starvation for CPU-bound jobs if too many IO-bound jobs
- A user can “trick” the system: put a garbage IO just before the end of the time slice to keep high priority
- What if a program changes its behavior over time?

MLFQ scheduling: managing priorities

Second try

Weaknesses of the current solution

- Risk of starvation for CPU-bound jobs if too many IO-bound jobs
- A user can "trick" the system: put a garbage IO just before the end of the time slice to keep high priority
- What if a program changes its behavior over time?

Priority Boost

- Rule 5: After some time period S, move all the jobs in the system to the topmost queue.
  - Avoids starvation
  - Deals with the case of an application changing from CPU-bound to IO-bound

MLFQ scheduling: managing priorities

Third try

Better accounting

We replace rules 4a and 4b by the following single rule:

- Rule 4: Once a job uses up its time slice 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).
  - The scheduler keeps track of how much CPU time each job uses
  - Impossible to use some "gaming strategy" to keep high priority

MLFQ scheduling: configuration

Several parameters of MLFQ can be tuned. There is no single good configuration.

- How many queues?
  - E.g., 60 queues
- How long should be the time slice in each queue?
  - Some systems use small time slices for high priority queues, and big time slices for low priority.
- How often should priority boost be run ?
  - E.g., every 1 second

Agenda

Multiprocessor scheduling

Why can't we simply reuse what we have just seen?

Multiprocessor scheduling

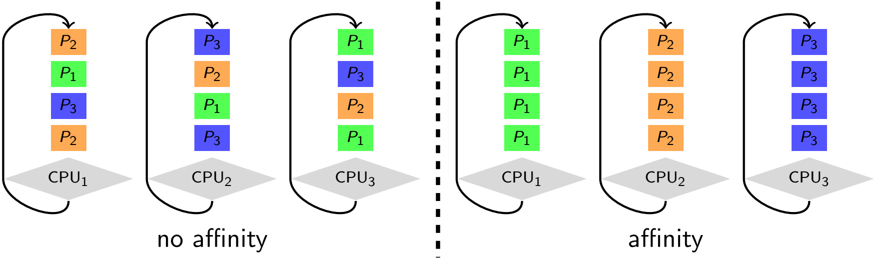
Why can't we simply reuse what we have just seen?

- The problem is more complex: We need to decide which process to run on which CPU.
- Migrating processes from CPU to CPU is very costly: It will generate a lot of cache misses

Multiprocessor scheduling

Affinity scheduling

- Typically one *scheduler* per CPU
- Risk of load imbalance
  - Do cost-benefit analysis when deciding to migrate



References for this lecture

- *Operating Systems: Three Easy Pieces* by R. Arpaci-Dusseau and A. Arpaci-Dusseau
  - Chapter 7: CPU scheduling
  - Chapter 8: Multi-level feedback
  - Chapter 10: Multi-CPU scheduling
- *Operating System Concepts* by A. Silberschatz et al.
  - Chapter 5: CPU scheduling