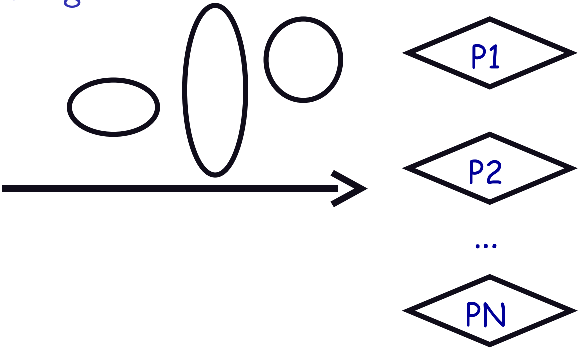
CPU Scheduling

Operating System Design – MOSIG 1

Instructor: Arnaud Legrand
Class Assistants: Benjamin Negrevergne, Sascha Hunold

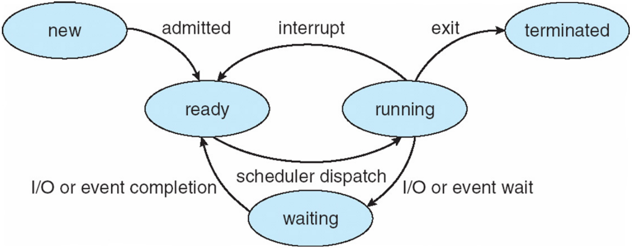
November 16, 2010

CPU Scheduling



- **The scheduling problem:**
  - Have K jobs ready to run
  - Have $N \geq 1$ CPUs
  - Which jobs to assign to which CPU(s)
- **When do we make decision?**

CPU Scheduling



- **Scheduling decisions may take place when a process:**
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Exits
- **Non-preemptive schedules use 1 & 4 only**
- **Preemptive schedulers run at all four points**

Criteria: Intuitive Notion

CPU utilization (max) percent usage of CPU.

Criteria: Intuitive Notion

CPU utilization (max) percent usage of CPU. Only useful computations (mix CPU, I/O; preemption overhead).

Throughput (max) *average* number of tasks that complete their execution per time-unit.

Criteria: Intuitive Notion

CPU utilization (max) percent usage of CPU. Only useful computations (mix CPU, I/O; preemption overhead).

Throughput (max) *average* number of tasks that complete their execution per time-unit.

Turnaround Time/Response Time/Flow (min) amount of time it takes between the task arrival and its completion.

Waiting Time (min) amount of time spent waiting for being executed.

Criteria: Intuitive Notion

CPU utilization (max) percent usage of CPU. Only useful computations (mix CPU, I/O; preemption overhead).

Throughput (max) *average* number of tasks that complete their execution per time-unit.

Turnaround Time/Response Time/Flow (min) amount of time it takes between the task arrival and its completion.

Waiting Time (min) amount of time spent waiting for being executed.

Slowdown/Stretch (min) slowdown factor encountered by a task relative to the time it would take on an unloaded system.

The previous quantities are task- or CPU-centric and need to be aggregated into a single objective function.

- max (the worst case)
- average: arithmetic (i.e. sum) or something else...
- variance (to be "fair" between the tasks).

Criteria: Classical Definitions

A given task T_i is defined by:

- processing time p_i
- release date r_i
- (number of required processors q_i)
- (deadline d_i)

Then, depending on the scheduling decision, we obtain its completion time C_i

Completion Time

- Makespan: $C_{\max} = \max_i C_i$
  This metric is relevant when scheduling a *single* application (made of several synchronized process).
- Total (or average) Completion Time: $SC = \sum_i C_i$

Criteria: Classical Definitions

A given task T_i is defined by:

- processing time p_i
- release date r_i
- (number of required processors q_i)
- (deadline d_i)

Then, depending on the scheduling decision, we obtain its completion time C_i

Response Time

$$F_i = C_i - r_i$$

- Maximum Flow Time: $F_{\max} = \max_i F_i$
- Total Completion Time: $SF = \sum_i F_i = SC - \sum_i r_i$

Criteria: Classical Definitions

A given task T_i is defined by:

- processing time p_i
- release date r_i
- (number of required processors q_i)
- (deadline d_i)

Then, depending on the scheduling decision, we obtain its completion time C_i

Waiting Time

$$W_i = C_i - r_i - p_i$$

- Maximum Waiting time: $W_{\max} = \max_i W_i$
- Total Waiting Time: $SW = \sum_i W_i = SF - \sum_i p_i$

Criteria: Classical Definitions

A given task T_i is defined by:

- processing time p_i
- release date r_i
- (number of required processors q_i)
- (deadline d_i)

Then, depending on the scheduling decision, we obtain its completion time C_i

Slowdown

$$S_i = \frac{C_i - r_i}{p_i}$$

- Maximum Stretch: $S_{\max} = \max_i S_i$
- Total Stretch: $SS = \sum_i S_i$

Outline

Optimizing largest response time

Optimizing throughput

Optimizing average response time

Avoiding starvation

Coming up with a compromise

Recap

Outline

Optimizing largest response time

Optimizing throughput

Optimizing average response time

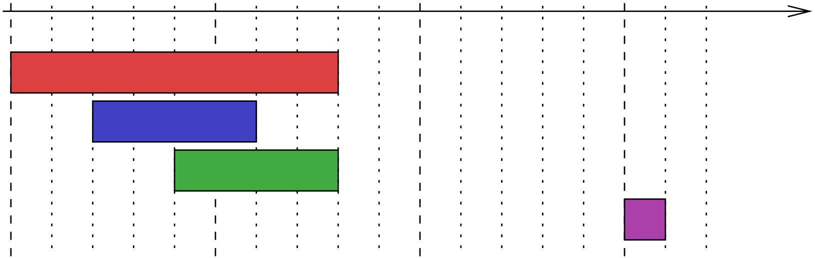
Avoiding starvation

Coming up with a compromise

Recap

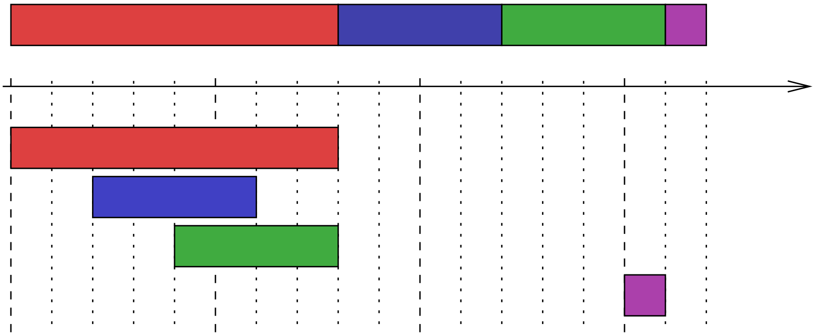
Let's play with a small example

We wish to find a schedule (possibly using preemption) that has the smallest possible max flow ($\max_i C_i - r_i$).



Let's play with a small example

We wish to find a schedule (possibly using preemption) that has the smallest possible max flow ($\max_i C_i - r_i = 12$).

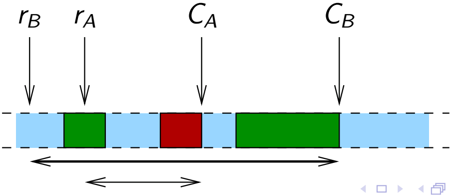


First-Come First-Served seems to be optimal.

FCFS is optimal: sketch of the proof

Proof:

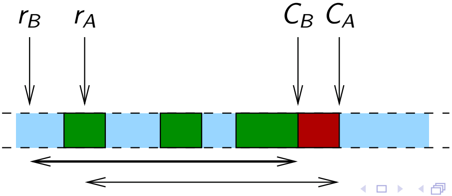
- Let us consider an optimal schedule σ. Let us assume that there are two jobs A and B that are not scheduled according to the FCFS policy, i.e. $r_B < r_A$ and $C_A < C_B$.



FCFS is optimal: sketch of the proof

Proof:

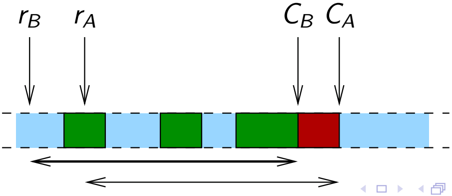
- Let us consider an optimal schedule σ. Let us assume that there are two jobs A and B that are not scheduled according to the FCFS policy, i.e. $r_B < r_A$ and $C_A < C_B$.
- By scheduling B before A, we do not increase $\max(C_A - r_A, C_B - r_B)$.



FCFS is optimal: sketch of the proof

Proof:

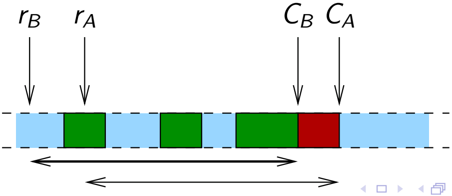
- Let us consider an optimal schedule σ. Let us assume that there are two jobs A and B that are not scheduled according to the FCFS policy, i.e. $r_B < r_A$ and $C_A < C_B$.
- By scheduling B before A, we do not increase $\max(C_A - r_A, C_B - r_B)$.
- Therefore, by scheduling A and B according to the FCFS policy, we get a new schedule σ' that is still optimal.



FCFS is optimal: sketch of the proof

Proof:

- Let us consider an optimal schedule σ. Let us assume that there are two jobs A and B that are not scheduled according to the FCFS policy, i.e. $r_B < r_A$ and $C_A < C_B$.
- By scheduling B before A, we do not increase $\max(C_A - r_A, C_B - r_B)$.
- Therefore, by scheduling A and B according to the FCFS policy, we get a new schedule σ' that is still optimal.
- By proceeding similarly for all pairs of jobs, we prove that FCFS is optimal. □



FCFS is optimal: sketch of the proof

Proof:

- Let us consider an optimal schedule σ. Let us assume that there are two jobs A and B that are not scheduled according to the FCFS policy, i.e. $r_B < r_A$ and $C_A < C_B$.
- By scheduling B before A, we do not increase $\max(C_A - r_A, C_B - r_B)$.
- Therefore, by scheduling A and B according to the FCFS policy, we get a new schedule σ' that is still optimal.
- By proceeding similarly for all pairs of jobs, we prove that FCFS is optimal. □

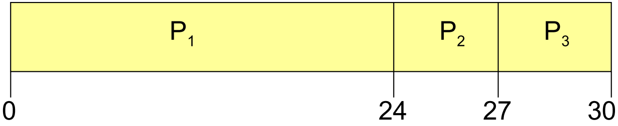
We do not even need to preempt jobs! Note that when you have *more than one processor*, things are more complicated:

Bad News NP-complete with no preemption.

Good news Polynomial algorithm with preemption but it is much more complicated than FCFS.

FCFS: other criteria

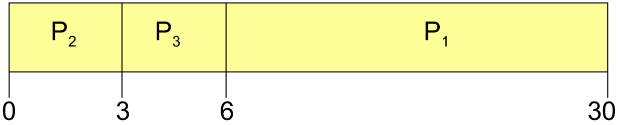
- **The FCFS scheduling policy is non-clairvoyant, easy to implement, and does not use preemption.**
- **The FCFS policy is optimal for minimizing** $\max F_i$**. It minimizes the "response time"!**
  Yet, would you say it is "reactive" ?
- **Run jobs in order that they arrive**
  - E.g.., Say P_1 needs 24 sec, while P_2 and P_3 need 3.
  - Say P_2, P_3 arrived immediately after P_1, get:



- **Throughput: 3 jobs / 30 sec = 0.1 jobs/sec**
- **Turnaround Time:** $P_1 : 24$**,** $P_2 : 27$**,** $P_3 : 30$
  - Average TT: $(24 + 27 + 30)/3 = 27$
- **Can we do better?**

FCFS continued

- **We would accept to sacrifice some jobs to get something more "reactive".**
- **Suppose we scheduled P_2, P_3, then P_1**
  - Would get:



- **Throughput: 3 jobs / 30 sec = 0.1 jobs/sec**
- **Turnaround time:** $P_1 : 30$, $P_2 : 3$, $P_3 : 6$
  - Average TT: $(30 + 3 + 6)/3 = 13$ – much less than 27
- **Lesson: scheduling algorithm can reduce TT**
- **What about throughput?**

Outline

Optimizing largest response time

Optimizing throughput

Optimizing average response time

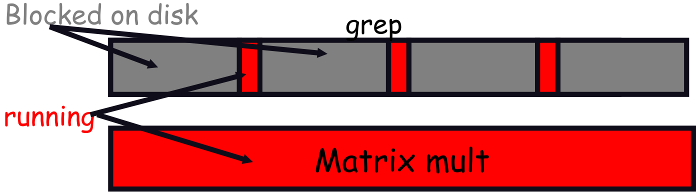
Avoiding starvation

Coming up with a compromise

Recap

View CPU and I/O devices the same

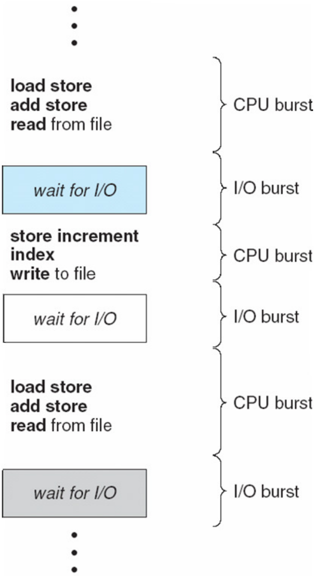
- **CPU is one of several devices needed by users' jobs**
  - CPU runs compute jobs, Disk drive runs disk jobs, etc.
  - With network, part of job may run on remote CPU
- **Scheduling 1-CPU system with n I/O devices like scheduling asymmetric $n + 1$-CPU multiprocessor**
  - Result: all I/O devices + CPU busy $\Longrightarrow$ n+1 fold speedup!



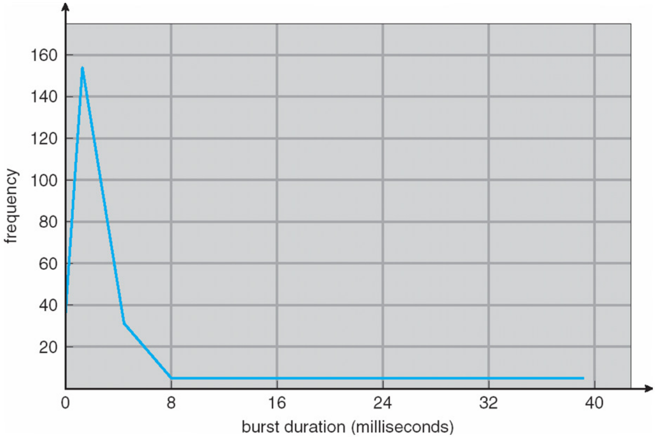
  - Overlap them just right? throughput will be almost doubled

Bursts of computation & I/O

- **Jobs contain I/O and computation**
  - Bursts of computation
  - Then must wait for I/O
- **To Maximize throughput**
  - Must maximize CPU utilization
  - Also maximize I/O device utilization
- **How to do?**
  - Overlap I/O & computation from multiple jobs
  - Means *response time* very important for I/O-intensive jobs: I/O device will be idle until job gets small amount of CPU to issue next I/O request



Histogram of CPU-burst times



- **What does this mean for FCFS?**

FCFS Convoy effect

- **CPU bound jobs will hold CPU until exit or I/O (but I/O rare for CPU-bound thread)**
  - long periods where no I/O requests issued, and CPU held
  - Result: poor I/O device utilization
- **Example: one CPU-bound job, many I/O bound**
  - CPU bound runs (I/O devices idle)
  - CPU bound blocks
  - I/O bound job(s) run, quickly block on I/O
  - CPU bound runs again
  - I/O completes
  - CPU bound job continues while I/O devices idle
- **Simple hack: run process whose I/O completed?**
  - What is a potential problem?

Outline

Optimizing largest response time

Optimizing throughput

Optimizing average response time

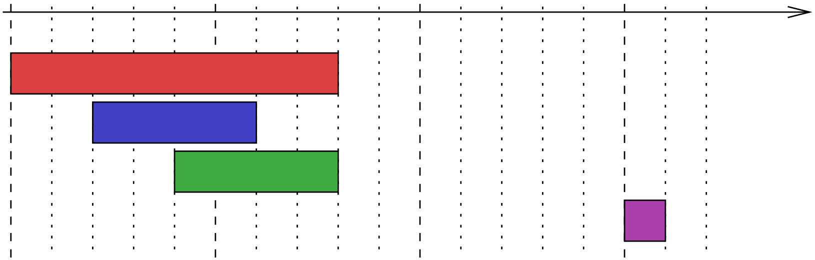
Avoiding starvation

Coming up with a compromise

Recap

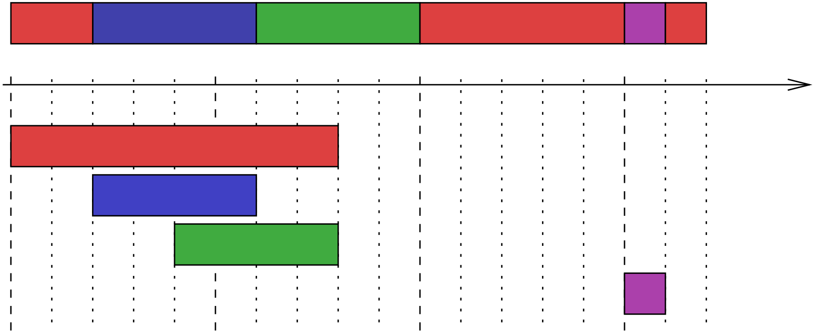
Let's play with a small example

We wish to find a schedule (possibly using preemption) that has the smallest possible sum flow ($\sum_i C_i - r_i$).



Let's play with a small example

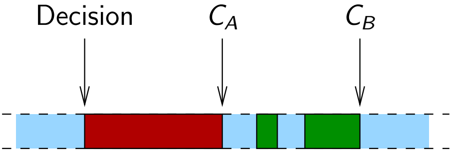
We wish to find a schedule (possibly using preemption) that has the smallest possible sum flow ($\sum_i C_i - r_i = 28$).



Shortest Remaining Processing Timer first seems to be optimal.

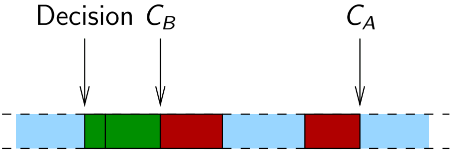
SRPT is optimal: sketch of the proof

- Let us consider an optimal schedule σ. Let us assume that there are two jobs A and B that are not scheduled according to the SRPT policy, i.e. $C_A < C_B$ and at some point there were more work to finish A than to finish B.



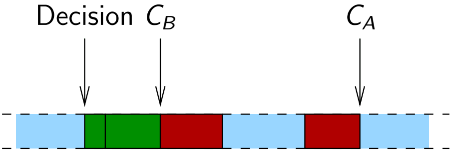
SRPT is optimal: sketch of the proof

- Let us consider an optimal schedule σ. Let us assume that there are two jobs A and B that are not scheduled according to the SRPT policy, i.e. $C_A < C_B$ and at some point there were more work to finish A than to finish B.
- By scheduling B before A, we strictly decrease $C_A + C_B$ and thus we strictly decrease the total flow.



SRPT is optimal: sketch of the proof

- Let us consider an optimal schedule σ. Let us assume that there are two jobs A and B that are not scheduled according to the SRPT policy, i.e. $C_A < C_B$ and at some point there were more work to finish A than to finish B.
- By scheduling B before A, we strictly decrease $C_A + C_B$ and thus we strictly decrease the total flow.
- Therefore, the original schedule was not optimal! The only optimal schedule is thus SRPT. □



SRPT is optimal: sketch of the proof

- Let us consider an optimal schedule σ. Let us assume that there are two jobs A and B that are not scheduled according to the SRPT policy, i.e. $C_A < C_B$ and at some point there were more work to finish A than to finish B.
- By scheduling B before A, we strictly decrease $C_A + C_B$ and thus we strictly decrease the total flow.
- Therefore, the original schedule was not optimal! The only optimal schedule is thus SRPT. □

Here, preemption is required!

Bad News NP-complete for multiple processors or with no preemption.

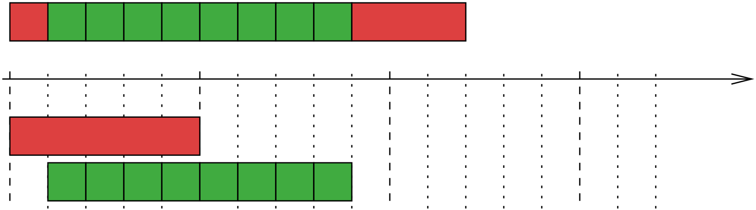
Good News Algorithm with logarithmic competitive ratio on multiple processors exists.

Comments

- Scheduling small jobs first is good for "reactivity" but it requires to know the size of the jobs (i.e. clairvoyant).
- Scheduling small jobs first is good for the average response time but some jobs may be left behind...

Comments

- Scheduling small jobs first is good for "reactivity" but it requires to know the size of the jobs (i.e. clairvoyant).
- Scheduling small jobs first is good for the average response time but large jobs may be left behind...



- Do you know an algorithm where job cannot starve?

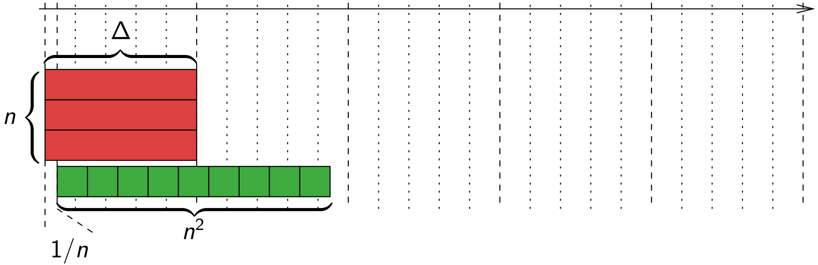
FCFS is Δ-competitive for $\sum F_i$

Δ: ratio of the sizes of the largest and smallest job.

Let's prove FCFS is at most Δ-competitive.

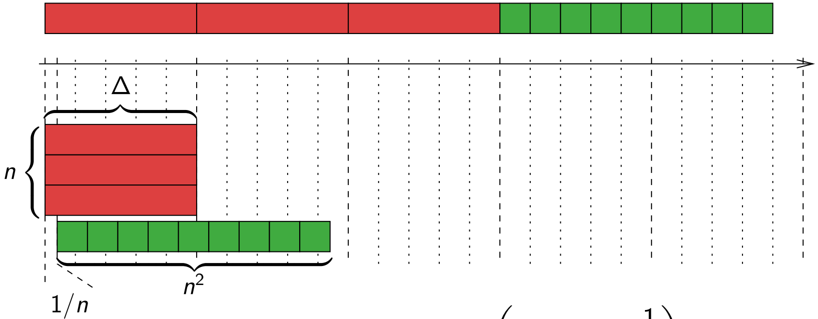
FCFS is Δ-competitive for $\sum F_i$

Δ: ratio of the sizes of the largest and smallest job.
Let's prove FCFS is at most Δ-competitive.



FCFS is Δ-competitive for $\sum F_i$

Δ: ratio of the sizes of the largest and smallest job.
Let's prove FCFS is at most Δ-competitive.

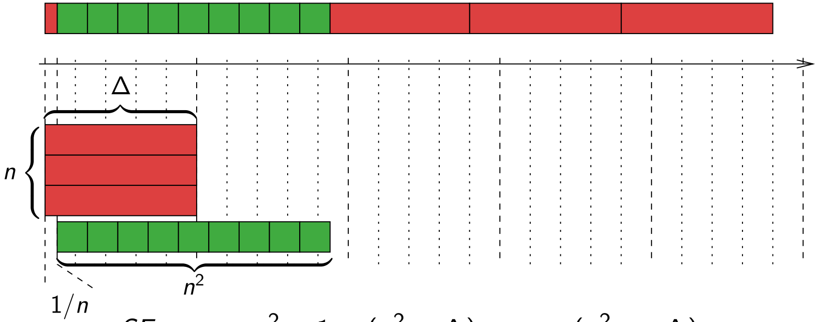


$$SF_{FCFS} = \Delta + \ldots + n\Delta + n^2\left(1 + n\Delta - \frac{1}{n}\right)$$
$$= \frac{2n^3\Delta + n^2(2+\Delta) + n(\Delta - 2)}{2}$$

FCFS is Δ-competitive for $\sum F_i$

Δ: ratio of the sizes of the largest and smallest job.

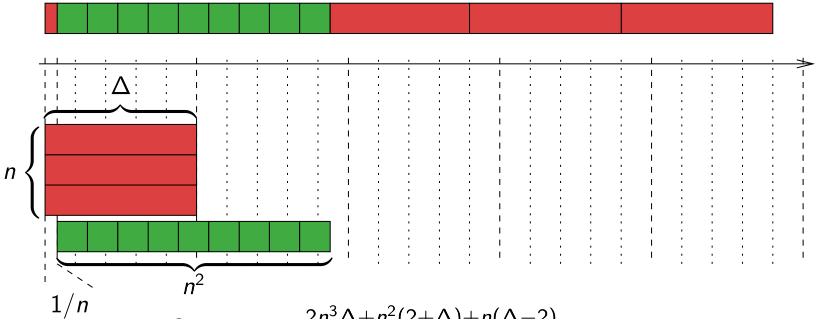
Let's prove FCFS is at most Δ-competitive.



$$SF_{SRPT} = n^2 \times 1 + (n^2 + \Delta) + \ldots + (n^2 + n\Delta)$$
$$= n^3 + n^2 + \frac{n(n+1)}{2}\Delta$$

FCFS is Δ-competitive for $\sum F_i$

Δ: ratio of the sizes of the largest and smallest job.
Let's prove FCFS is at most Δ-competitive.



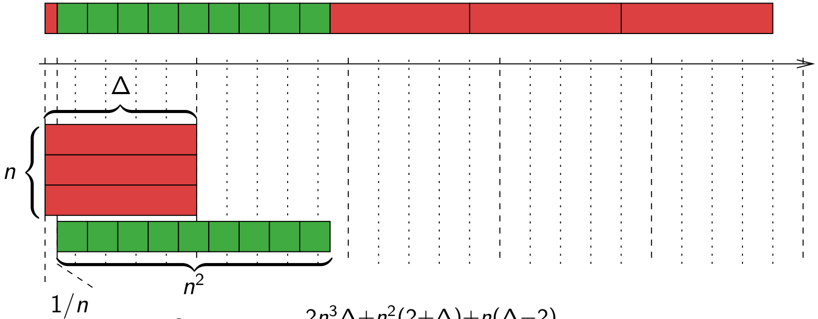
$$\varrho_{FCFS}(\Delta) \geqslant \frac{SF_{FCFS}}{SF_{SRPT}} = \frac{\frac{2n^3\Delta+n^2(2+\Delta)+n(\Delta-2)}{2}}{n^3+n^2+\frac{n(n+1)}{2}\Delta}$$

$$= \frac{2n^3\Delta+n^2(2+\Delta)+n(\Delta-2)}{2n^3+2n^2+n(n+1)\Delta} \xrightarrow[n\to+\infty]{} \Delta$$

FCFS is Δ-competitive for $\sum F_i$

Δ: ratio of the sizes of the largest and smallest job.

Let's prove FCFS is at most Δ-competitive.



$$\varrho_{FCFS}(\Delta) \geqslant \frac{SF_{FCFS}}{SF_{SRPT}} = \frac{\frac{2n^3\Delta+n^2(2+\Delta)+n(\Delta-2)}{2}}{n^3+n^2+\frac{n(n+1)}{2}\Delta}$$

$$= \frac{2n^3\Delta+n^2(2+\Delta)+n(\Delta-2)}{2n^3+2n^2+n(n+1)\Delta} \xrightarrow[n\to+\infty]{} \Delta$$

FCFS is at exactly Δ-competitive.

Optimizing the average response time with no starvation?

Theorem

Consider any online algorithm whose competitive ratio for average flow minimization satisfies $\varrho(\Delta) < \Delta$.

There exists for this algorithm a sequence of jobs leading to starvation, and for which the maximum flow can be as far as we want from the optimal maximum flow.

The starvation issue is inherent to the optimization of the average response time.
Still, we would like something "reactive" and we like the idea that short jobs have a higher priority.

Recap SJF/SRPT limitations

- **The previous analysis relies on a model (i.e. a simplification of reality) and is thus limited.**
- **Actually, in practice, doesn't always minimize average turnaround time**
  - Example where turnaround time might be suboptimal?

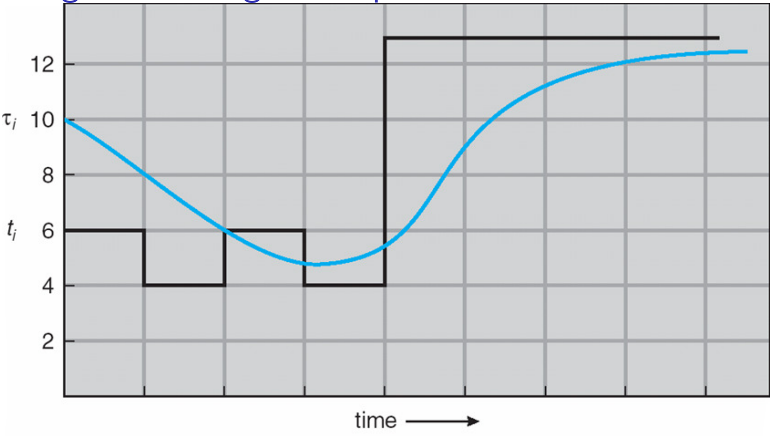
Recap SJF/SRPT limitations

- **The previous analysis relies on a model (i.e. a simplification of reality) and is thus limited.**
- **Actually, in practice, doesn't always minimize average turnaround time**
  - Example where turnaround time might be suboptimal?
    If applications are made of jobs/tasks that have dependencies (synchronizations), if more than 1 CPU, ...
- **Can lead to unfairness or starvation**

Recap SJF/SRPT limitations

- **The previous analysis relies on a model (i.e. a simplification of reality) and is thus limited.**
- **Actually, in practice, doesn't always minimize average turnaround time**
  - Example where turnaround time might be suboptimal?
    If applications are made of jobs/tasks that have dependencies (synchronizations), if more than 1 CPU, ...
- **Can lead to unfairness or starvation**
- **In practice, can't actually predict the future**
- **But can estimate CPU burst length based on past**
  - Exponentially weighted average a good idea
  - t_n actual length of proc's n^{th} CPU burst
  - τ_{n+1} estimated length of proc's $n+1^{\text{st}}$
  - Choose parameter α where $0 < \alpha \leq 1$
  - Let $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

Exp. weighted average example



CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Outline

Optimizing largest response time

Optimizing throughput

Optimizing average response time

Avoiding starvation

Coming up with a compromise

Recap

Round robin (RR) scheduling

- **Solution to fairness and starvation**
  - Preempt job after some time slice or *quantum*
  - When preempted, move to back of FIFO queue
  - (Most systems do some flavor of this)
- **Advantages:**
  - Fair allocation of CPU across jobs
  - Low average waiting time when job lengths vary
  - Good for responsiveness if small number of jobs
- **Disadvantages?**

RR disadvantages

- **Varying sized jobs are good**
  . . . but what about same-sized jobs?
- **Assume 2 jobs of time=100 each:**

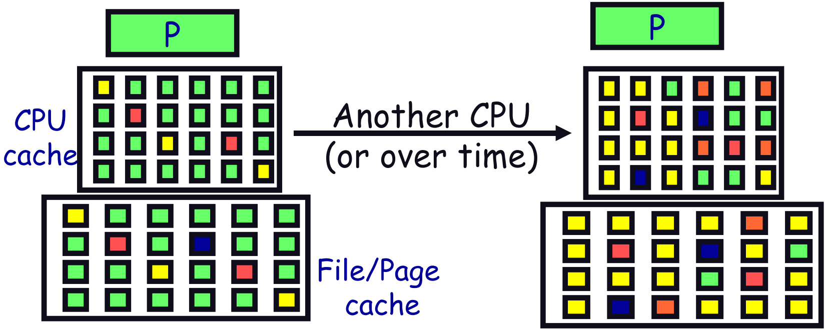
  - What is average completion time?
  - How does that compare to FCFS?
- **In the previous algorithms (FCFS, SRPT), we have never produced a schedule with** $\dots A \dots B \dots A \dots B \dots$**. Intuitively alternating jobs is not a good idea for minimizing completion time.**
- **Preemption should not be blindly used to ensure fairness. It should help to deal with the online non-clairvoyant setting.**

Context switch costs

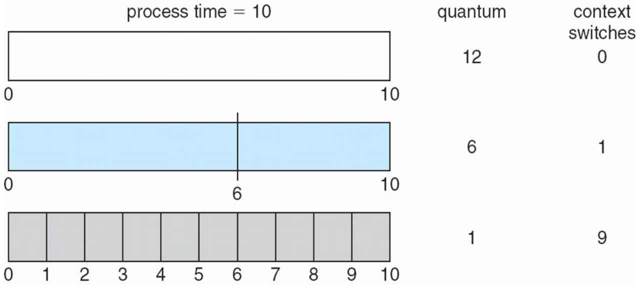
- **What is the cost of a context switch? (recall from previous lectures)**

Context switch costs

- **What is the cost of a context switch? (recall from previous lectures)**
- **Brute CPU time cost in kernel**
  - Save and restore resisters, etc.
  - Switch address spaces (expensive instructions)
- **Indirect costs: cache, buffer cache, & TLB misses**

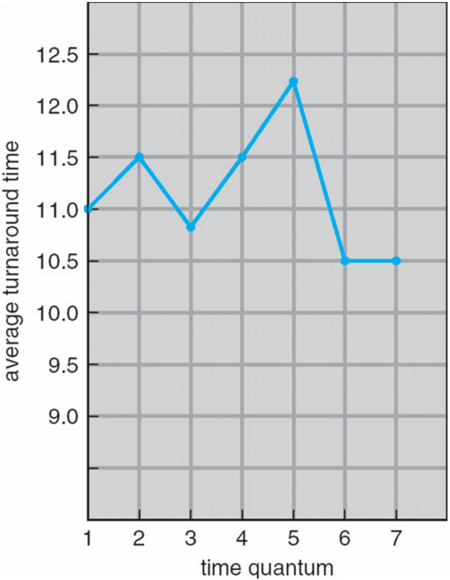


Time quantum



- **How to pick quantum?**
  - Want much larger than context switch cost
  - Majority of bursts should be less than quantum
  - But not so large system reverts to FCFS
- **Typical values: 10–100 msec**

Turnaround time vs. quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Two-level scheduling

- **Switching to swapped out process very expensive**
  - Swapped out process has most pages on disk
  - Will have to fault them all in while running
  - One disk access costs ~10ms. On 1GHz machine, 10ms = 10 million cycles!
- **Context-switch-cost aware scheduling**
  - Run in-core subset for “a while”
  - Then swap some between disk and memory
- **How to pick subset? How to define “a while”?**
  - View as scheduling *memory* before CPU
  - Swapping in process is cost of memory “context switch”
  - So want “memory quantum” much larger than swapping cost

Outline

Optimizing largest response time

Optimizing throughput

Optimizing average response time

Avoiding starvation

Coming up with a compromise

Recap

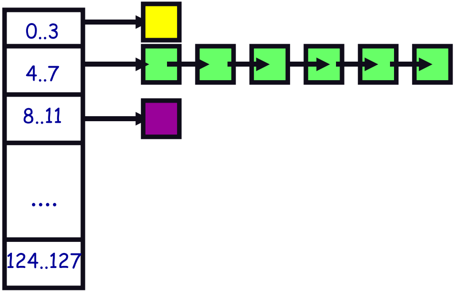
Priority scheduling

- **Associate a numeric priority with each process**
  - E.g., smaller number means higher priority (Unix/BSD)
  - Or smaller number means lower priority (Pintos)
- **Give CPU to the 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**
- **Starvation – low priority processes may never execute**
- **Solution?**

Priority scheduling

- **Associate a numeric priority with each process**
  - E.g., smaller number means higher priority (Unix/BSD)
  - Or smaller number means lower priority (Pintos)
- **Give CPU to the 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**
- **Starvation – low priority processes may never execute**
- **Solution?**
  - Aging - increase a process's priority as it waits

Multilevel feeedback queues (BSD)



- **Every runnable process on one of 32 run queues**
  - Kernel runs process on highest-priority non-empty queue
  - Round-robins among processes on same queue
- **Process priorities dynamically computed**
  - Processes moved between queues to reflect priority changes
  - If a process gets higher priority than running process, run it
- **Idea: Favor interactive jobs that use less CPU**

Process priority

- $\texttt{p\_nice}$ – **user-settable weighting factor**
- $\texttt{p\_estcpu}$ – **per-process estimated CPU usage**
  - Incremented whenever timer interrupt found proc. running
  - Decayed every second while process runnable

$$\texttt{p\_estcpu} \leftarrow \left(\frac{2 \cdot \mathrm{load}}{2 \cdot \mathrm{load} + 1}\right) \texttt{p\_estcpu} + \texttt{p\_nice}$$

  - Load is sampled average of length of run queue plus short-term sleep queue over last minute
- **Run queue determined by** $\texttt{p\_usrpri}/4$

$$\texttt{p\_usrpri} \leftarrow 50 + \left(\frac{\texttt{p\_estcpu}}{4}\right) + 2 \cdot \texttt{p\_nice}$$

(value clipped if over 127)

Sleeping process increases priority

- `p_estcpu` **not updated while asleep**
  - Instead `p_slptime` keeps count of sleep time
- **When process becomes runnable**

$$\texttt{p\_estcpu} \leftarrow \left(\frac{2 \cdot \text{load}}{2 \cdot \text{load} + 1}\right)^{\texttt{p\_slptime}} \times \texttt{p\_estcpu}$$

  - Approximates decay ignoring nice and past loads
- **These are uggly hacks.**
  - The BSD time quantum: 1/10 sec (since ∼1980)
  - Empirically longest tolerable latency
  - Computers now faster, but job queues also shorter

Limitations of BSD scheduler

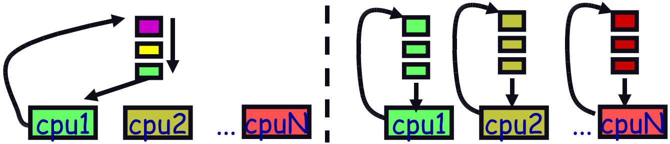
- **Hard to have isolation / prevent interference**
  - Priorities are absolute
- **Can't donate priority (e.g., to server on RPC)**
- **No flexible control**
  - E.g., In monte carlo simulations, error is 1/sqrt(N) after N trials
  - Want to get quick estimate from new computation
  - Leave a bunch running for a while to get more accurate results
- **Multimedia applications**
  - Often fall back to degraded quality levels depending on resources
  - Want to control quality of different streams

Real-time scheduling

- **Two categories:**
  - *Soft real time*—miss deadline and CD will sound funny
  - *Hard real time*—miss deadline and plane will crash
- **System must handle periodic and aperiodic events**
  - E.g., procs A, B, C must be scheduled every 100, 200, 500 msec, require 50, 30, 100 msec respectively
  - *Schedulable* if $\sum \frac{CPU}{\mathrm{period}} \leq 1$ (not counting switch time)
- **Variety of scheduling strategies**
  - E.g., first deadline first (works if schedulable)
- **Linux is finaly slightly moving from priority scheduling to deadline scheduling**

Multiprocessor scheduling issues

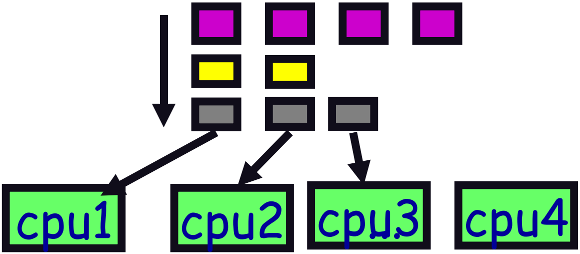
- **Must decide on more than which processes to run**
  - Must decide on which CPU to run which process
- **Moving between CPUs has costs**
  - More cache misses, depending on arch more TLB misses too
- **Affinity scheduling—try to keep threads on same CPU**



  - But also prevent load imbalances
  - Do *cost-benefit* analysis when deciding to migrate

Multiprocessor scheduling (cont)

- **Want related processes scheduled together**
  - Good if threads access same resources (e.g., cached files)
  - Even more important if threads communicate often, otherwise must context switch to communicate
- **Gang scheduling—schedule all CPUs synchronously**
  - With synchronized quanta, easier to schedule related processes/threads together



Outline

Optimizing largest response time

Optimizing throughput

Optimizing average response time

Avoiding starvation

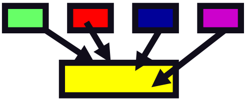
Coming up with a compromise

Recap

CPU Scheduling Recap

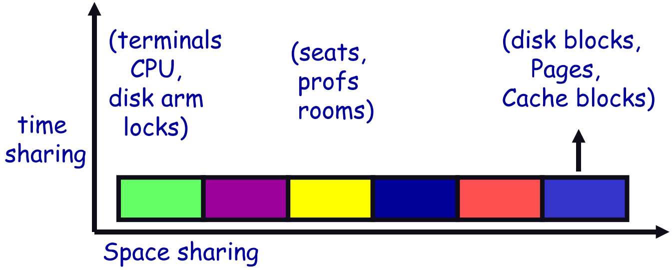
- **Goal: High throughput**
  - Minimize context switches to avoid wasting CPU, TLB misses, cache misses, even page faults.
- **Goal: Low latency**
  - People typing at editors want fast response
  - Network services can be latency-bound, not CPU-bound
- **Algorithms**
  - Round-robin
  - Priority scheduling
  - Shortest process next (if you can estimate it)
  - Fair-Share Schedule (try to be fair at level of users, not processes)
  - Fancy combinations of the above

The universality of scheduling



- **General problem: Let m requests share n resources**
  - Always same issues: fairness, prioritizing, optimization
- **Disk arm: which read/write request to do next?**
  - Optimal: close requests = faster
  - Fair: don't starve far requests
- **Memory scheduling: whom to take page from?**
  - Optimal: past=future? take from least-recently-used
  - Fair: equal share of memory
- **Printer: what job to print?**
  - People = fairness paramount: uses FIFO rather than SJF
  - Use "admission control" to combat long jobs

How to allocate resources



- **Space sharing (sometimes): split up. When to stop?**
- **Time-sharing (always): how long do you give out piece?**
  - Pre-emptable (CPU, memory) vs. non-preemptable (locks, files, terminals)

Postscript

- **In principle, scheduling decisions can be arbitrary & shouldn't affect program's results**
  - Good, since rare that "the best" schedule can be calculated
- **In practice, schedule does affect correctness**
  - Soft real time (e.g., mpeg or other multimedia) common
  - Or after 10s of seconds, users will give up on web server
- **Unfortunately, algorithms strongly affect system throughput, turnaround time, and response time**
- **The best schemes are adaptive. To do absolutely best we'd have to predict the future.**
  - Most current algorithms tend to give the highest priority to the processes that need the least CPU time
  - Scheduling has gotten increasingly *ad hoc* over the years. 1960s papers very math heavy, now mostly "tweak and see"