scheduling 3

Changelog

Changes not seen in first lecture:

4 Feb 2020: MLFQ example: number priorities so largest number is highest priority

4 Feb 2020: lottery scheduler assignment: replace "how long processes run" with "how often processes scheduled' to better match assignment writeup

4 Feb 2020: CFS: A's long sleep: show overriding of natural virtual time with strikeout

4 Feb 2020: added CFS exercise explanation slides

last time

first-come first-served and round robin

round robin time quantum

- low = fair but bad throughput
- high = unfair (order) but good throughput

priority scheduling

- run some threads first, no matter what
- often implies starvation

shortest remaining time first (SRTF): mean turnaround time

started multi-level feedback queues

- approximate SRTF via priority buckets
- priority $\implies$ CPU burst estimate + time quantum
- exceeds time quantum? adjust priority

multi-level feedback queues

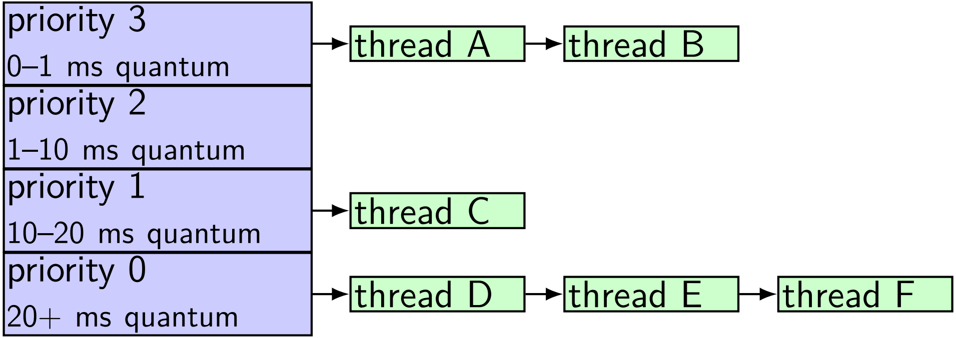
classic strategy based on priority scheduling

combines update time estimates and running shorter times first

key idea: current priority $\approx$ current time estimate

small(ish) number of time estimate “buckets”

multi-level feedback queues: setup



goal: place processes at priority level based on CPU burst time

- just a few priority levels — can't guess CPU burst precisely anyways

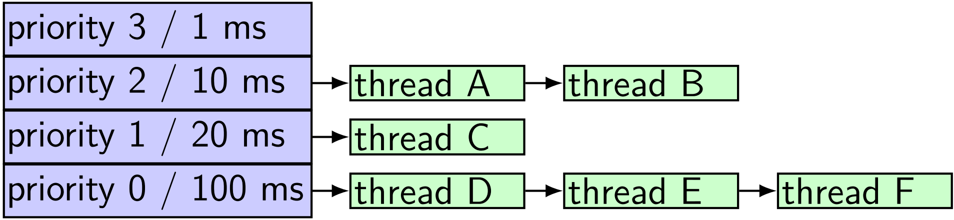
dynamically adjust priorities based on observed CPU burst times

priority level → allowed/expected time quantum

- use more than 1ms at priority 3? — you shouldn't be there
- use less than 1ms at priority 0? — you shouldn't be there

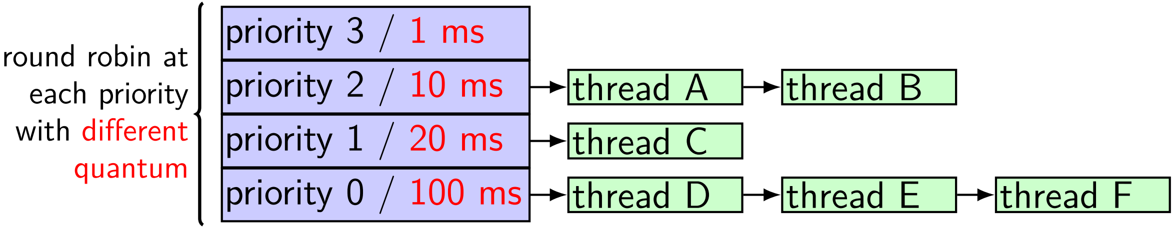
taking advantage of history

idea: priority = CPU burst length



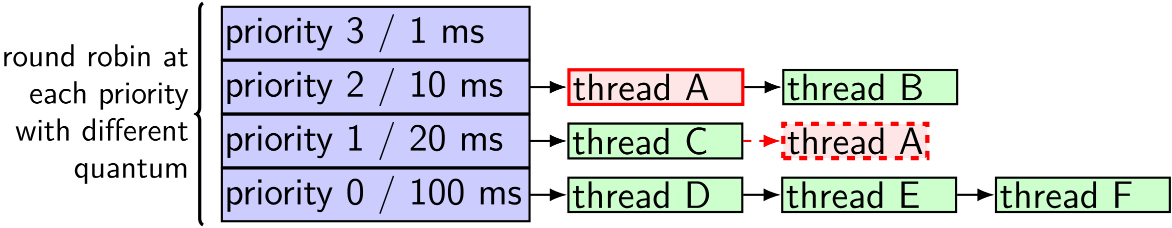
taking advantage of history

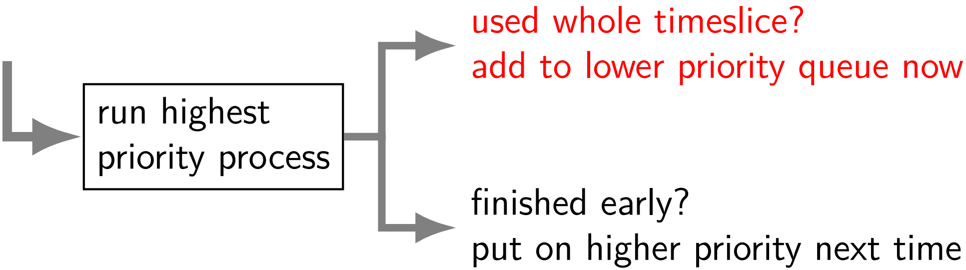
idea: priority = CPU burst length



taking advantage of history

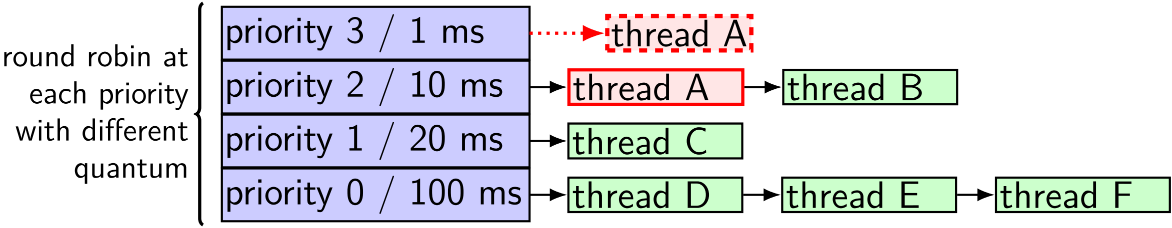
idea: priority = CPU burst length

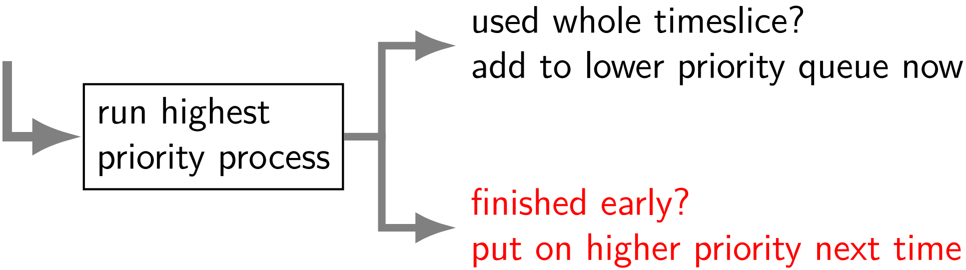




taking advantage of history

idea: priority = CPU burst length





multi-level feedback queue idea

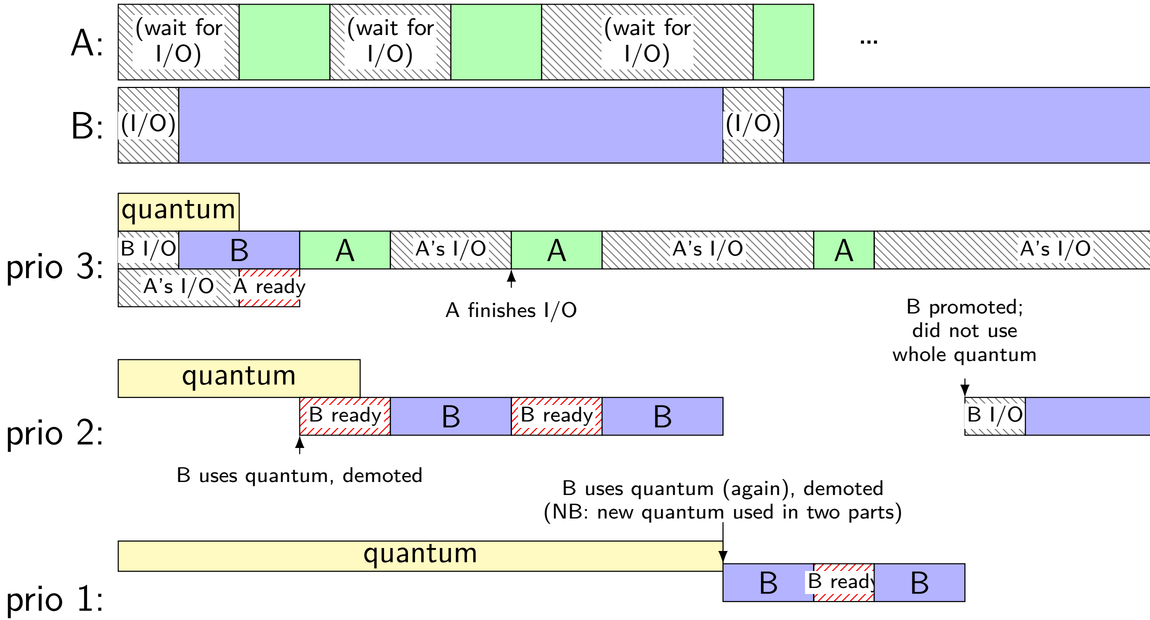
higher priority = shorter time quantum (before interrupted)

adjust priority *and* timeslice based on last timeslice

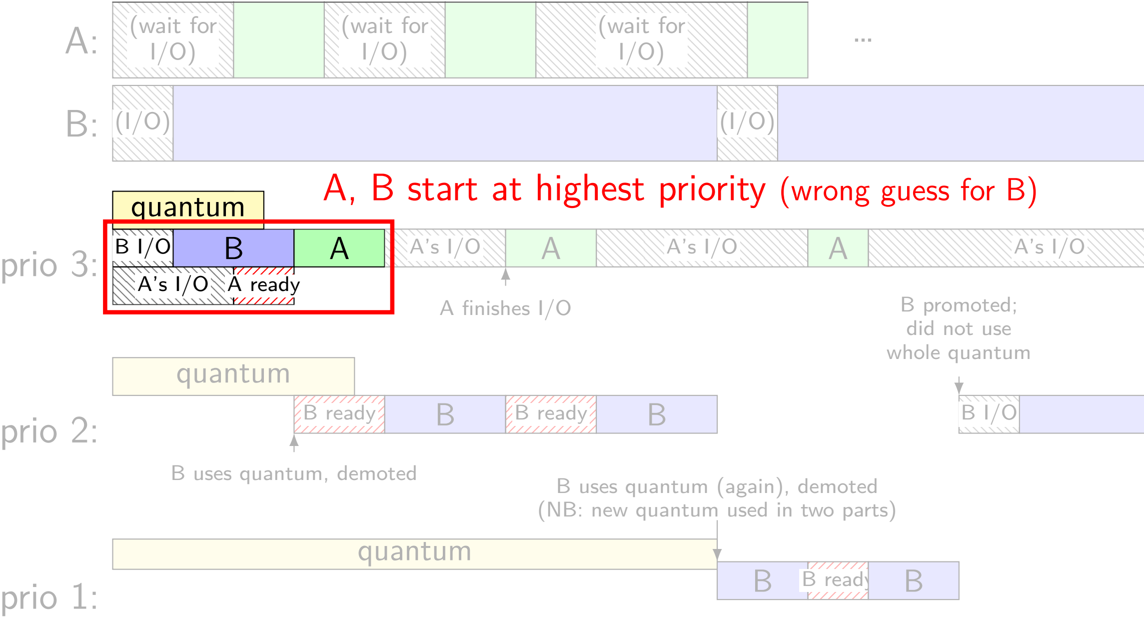
intuition: thread always uses same CPU burst length?
ends up at "right" priority

- rises up to queue with quantum just shorter than it's burst
- then goes down to next queue, then back up, then down, then up, etc.

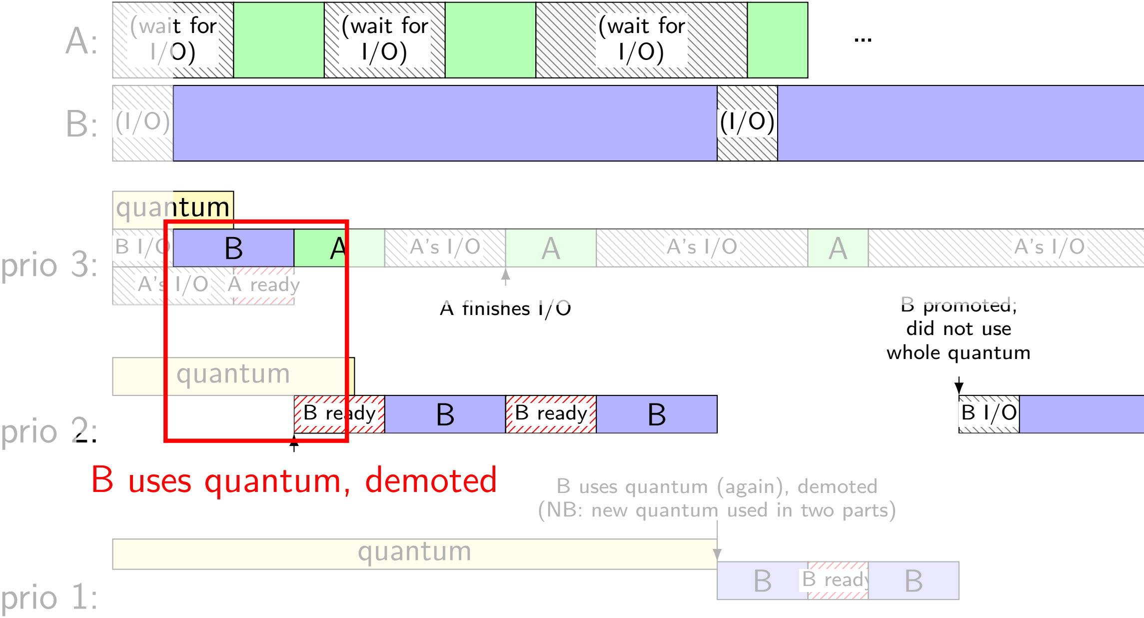
MLFQ example



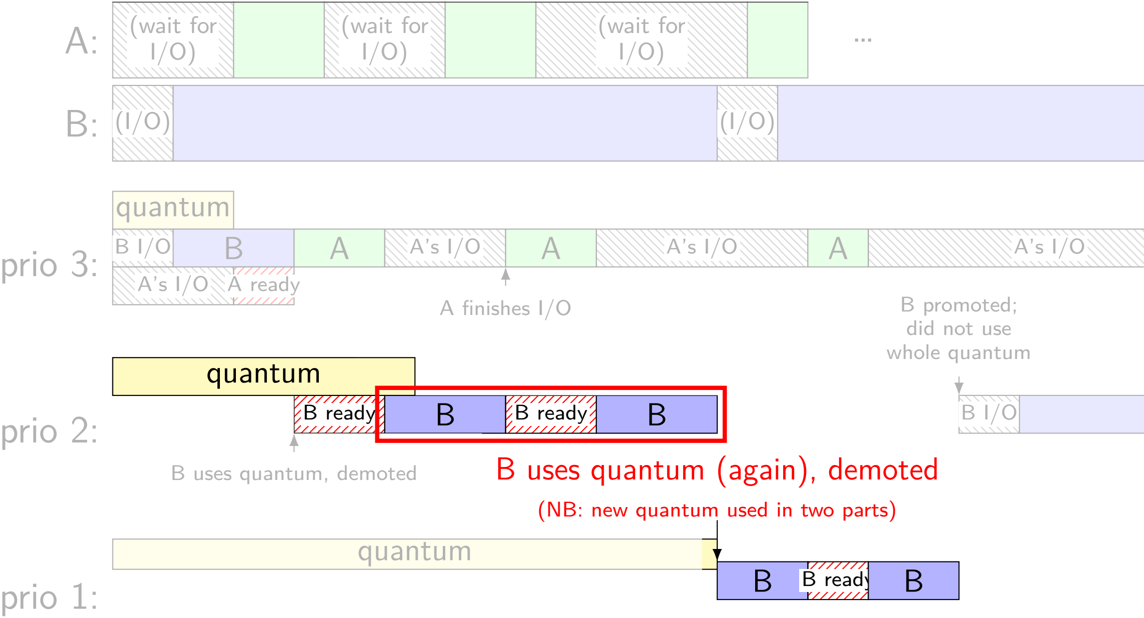
MLFQ example



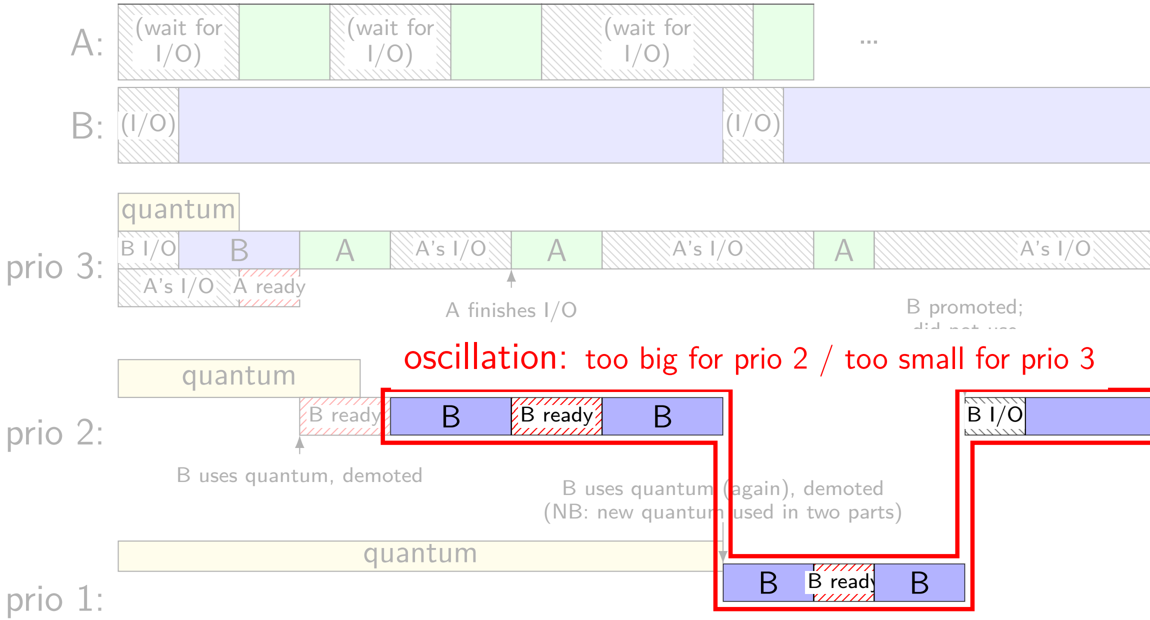
MLFQ example



MLFQ example



MLFQ example



cheating multi-level feedback queuing

algorithm: don't use entire time quantum? priority increases

getting all the CPU:

```
while (true) {
  useCpuForALittleLessThanMinimumTimeQuantum();
  yieldCpu();
}
```

multi-level feedback queuing and fairness

suppose we are running several programs:

A. one very long computation that doesn't need any I/O
B1 through B1000. 1000 programs processing data on disk
C. one interactive program

how much time will A get?

multi-level feedback queuing and fairness

suppose we are running several programs:

A. one very long computation that doesn't need any I/O
B1 through B1000. 1000 programs processing data on disk
C. one interactive program

how much time will A get?

almost none — starvation

intuition: the B programs have higher priority than A
because it has smaller CPU bursts

MLFQ variations

version of MLFQ I described is in Anderson-Dahlin

problems:

starvation

worse than with real SRTF — based on *guess*, not real remaining time

oscillation not great for predictability

variation to prevent starvation

Apraci-Dusseau presents version of MLFQ w/o starvation

two changes:

don't increase priority when whole quantum not used

- instead keep the same — more stable

periodically increase priority of *all threads*

- allow compute-heavy threads to run a little
- still deals with thread's behavior changing over time
- replaces finer-grained upward adjustments

FreeBSD scheduler

current default FreeBSD scheduler based on MLFQ idea

...but: time quantums don't depend on priority

computes *interactivity score* $\sim \dfrac{\text{I/O wait}}{\text{I/O wait} + \text{runtime}}$

note: deliberately not estimating remaining time

(using "recent" history of thread)

thread priorities set based on interactivity score

conflicting goals for interactivity heuristics

efficiency

- avoid scanning all threads every few milliseconds

figure out new programs quickly

adapt to changes/spikes in program behavior

avoid pathological behavior

- starvation, hanging when new compute-intensive program starts, etc.

exercise: how to handle each of these well?

- what does MLFQ do well?

fair scheduling

what is the fairest scheduling we can do?

intuition: every thread has an equal chance to be chosen

random scheduling algorithm

“fair” scheduling algorithm: choose uniformly at random

good for “fairness”

bad for response time

bad for predictability

proportional share

maybe every thread isn't equal

if thread A is twice as important as thread B, then…

proportional share

maybe every thread isn't equal

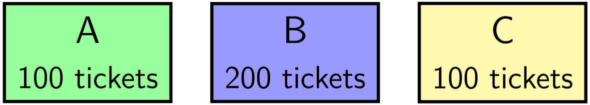
if thread A is twice as important as thread B, then...

one idea: thread A should run twice as much as thread B

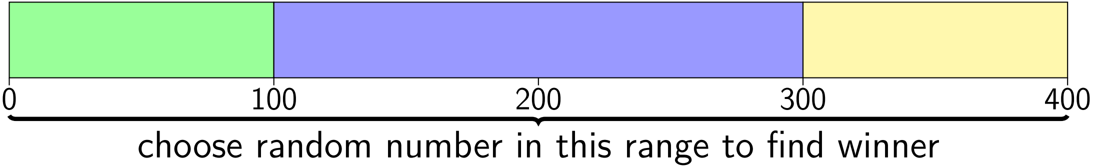
proportional share

lottery scheduling

every thread has a certain number of lottery tickets:



scheduling = lottery among ready threads:



simulating priority with lottery

A (high priority)	B (medium priority)	C (low priority)
1M tickets	1K tickets	1 tickets

very close to strict priority

...or to SRTF if priorities are set/adjusted right

simulating priority with lottery

A (high priority)	B (medium priority)	C (low priority)
1M tickets	1K tickets	1 tickets

very close to strict priority

...or to SRTF if priorities are set/adjusted right

lottery scheduling assignment

assignment: add lottery scheduling to xv6

extra system call: `settickets`

also counting of how often processes scheduled (for testing)

lottery scheduling assignment

assignment: add lottery scheduling to xv6

extra system call: `settickets`

also counting of how often processes scheduled (for testing)

simplification: okay if scheduling decisions are linear time

- there is a faster way

not implementing preemption before time slice ends

- might be better to run new lottery when process becomes ready?

is lottery scheduling actually good?

seriously proposed by academics in 1994 (Waldspurger and Weihl, OSDI'94)

- including ways of making it efficient
- making preemption decisions (other than time slice ending)
- if processes don't use full time slice
- handling non-CPU-like resources
- …

elegant mecahnism that can implement a variety of policies

but there are some problems…

exercise

thread A: 1 ticket, always runnable

thread B: 9 tickets, always runnable

over 10 time quantum
what is the probability A runs for at least 3 quanta?

- i.e. 3 times as much as "it's supposed to"
- chosen 3 times out of 10 instead of 1 out of 10

exercise

thread A: 1 ticket, always runnable

thread B: 9 tickets, always runnable

over 10 time quantum
what is the probability A runs for at least 3 quanta?

> i.e. 3 times as much as "it's supposed to"
> chosen 3 times out of 10 instead of 1 out of 10

approx. 7%

A runs w/in 10 times...

0 times	34%
1 time	39%
2 time	19%
3 time	6%
4 time	1%
5+ time	$<1\%$

(binomial distribution...)

aside: measuring fairness (1)

first question: what needs to be divided fairly?

problem: what about programs waiting for I/O?

answer 1:
don't consider what happens when program waiting for I/O

answer 2:
give program credit for time not running while waiting for I/O

aside: measuring fairness (1)

first question: what needs to be divided fairly?

problem: what about programs waiting for I/O?

answer 1:
don't consider what happens when program waiting for I/O

answer 2:
give program credit for time not running while waiting for I/O

aside: measuring fairness (1)

first question: what needs to be divided fairly?

problem: what about programs waiting for I/O?

answer 1:
don't consider what happens when program waiting for I/O

answer 2:
give program credit for time not running while waiting for I/O

aside: measuring fairness (2)

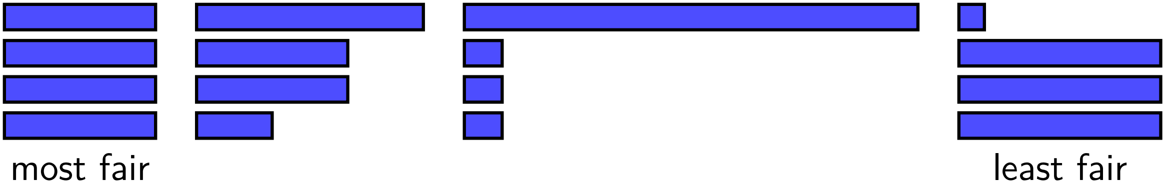
one way: max-min fairness

choose schedule that maximizes the minimum resource given to anyone

aside: measuring fairness (2)

one way: max-min fairness

choose schedule that maximizes the minimum resource given to anyone



lottery scheduler and interactivity

suppose two processes A, B, each have same # of tickets

process A is CPU-bound,

process B does lots of I/O (but has enough work to use 50% of the CPU)

lottery scheduler: run equally when both can run

result: B runs less than A

50% when both runnable

lottery scheduler and interactivity

suppose two processes A, B, each have same # of tickets

process A is CPU-bound,

process B does lots of I/O (but has enough work to use 50% of the CPU)

lottery scheduler: run equally when both can run

result: B runs less than A

- 50% when both runnable

is this fair?

- yes, it evenly splits up time when both programs runnable
- no, the programs don't get equal CPU time

recall: proportional share randomness

lottery scheduler: variance was a problem

- consistent over the long-term
- inconsistent over the short-term

want something more like weighted round-robin

- run one, then the other
- but run some things more often (depending on weight/# tickets)

deterministic proportional share scheduler

Linux's scheduler is a deterministic proportional share scheduler

…which gives processes credit when not runnable

Linux's Completely Fair Scheduler (CFS)

Linux's default scheduler is a proportional share scheduler…

…without randomization (consistent)

…with $O(\log N)$ scheduling decision

(handles many threads/processes)

…which favors interactive programs

…which adjusts timeslices dynamically

shorter timeslices if many things to run

Linux's Completely Fair Scheduler (CFS)

Linux's default scheduler is a proportional share scheduler...

...without randomization (consistent)

...with $O(\log N)$ scheduling decision

(handles many threads/processes)

...which favors interactive programs

...which adjusts timeslices dynamically

shorter timeslices if many things to run

CFS: tracking runtime

each thread has a *virtual runtime* ($\sim$ how long it's run)

incremented when run based how long it runs

scheduling decision: run thread with lowest virtual runtime

data structure: balanced tree

CFS: tracking runtime

each thread has a *virtual runtime* ($\sim$ how long it's run)

incremented when run based how long it runs

more/less important thread? multiply adjustments by factor

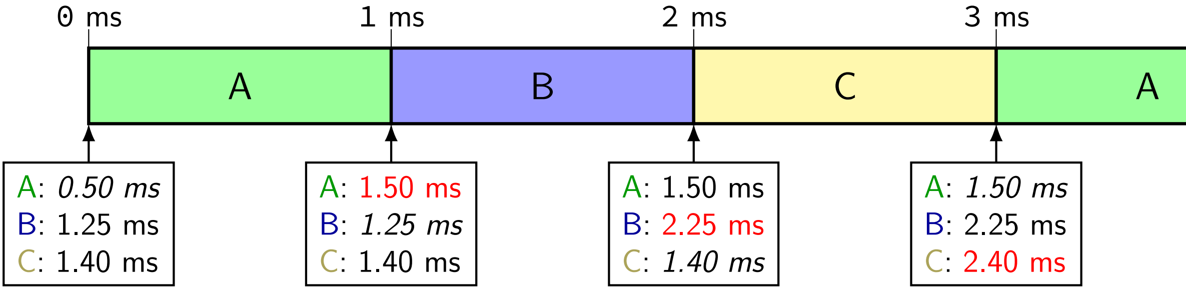
adjustments for threads that are new or were sleeping

- too big an advantage to start at runtime `0`

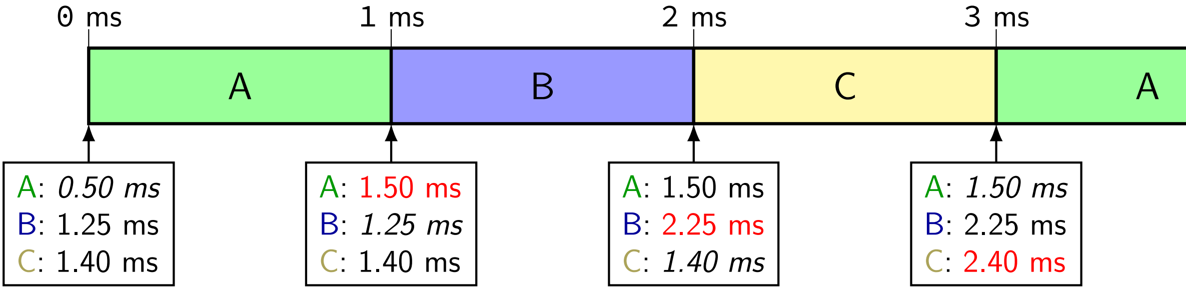
scheduling decision: run thread with lowest virtual runtime

- data structure: balanced tree

virtual time, always ready, 1 ms quantum



virtual time, always ready, 1 ms quantum

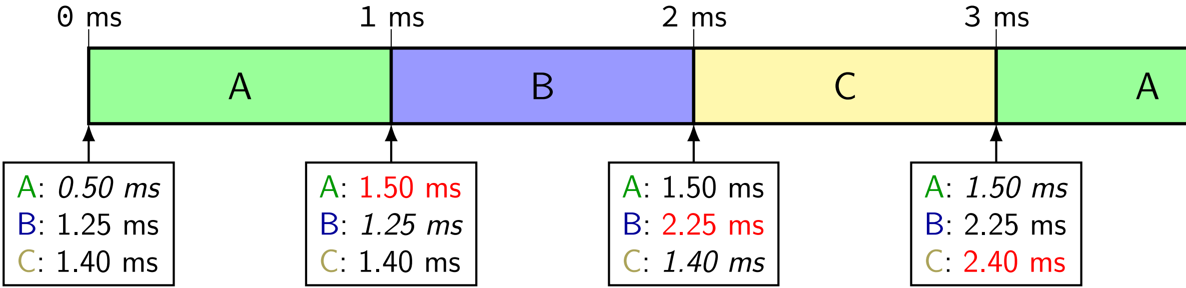


at each time:

update current thread's time

run thread with lowest total time

virtual time, always ready, 1 ms quantum



at each time:
update current thread's time
run thread with lowest total time

same effect as round robin
if everyone uses whole quantum

what about threads waiting for I/O, ...?

should be advantage for processes not using the CPU as much

- haven't used CPU for a while — deserve priority now
- ...but don't want to let them hog the CPU

Linux solution: newly ready task time = max of

- its prior virtual time
- a little less than minimum virtual time (of already ready tasks)

what about threads waiting for I/O, ...?

should be advantage for processes not using the CPU as much

- haven't used CPU for a while — deserve priority now
- ...but don't want to let them hog the CPU

Linux solution: newly ready task time = max of

- its prior virtual time
- a little less than minimum virtual time (of already ready tasks)

not runnable briefly? still get your share of CPU

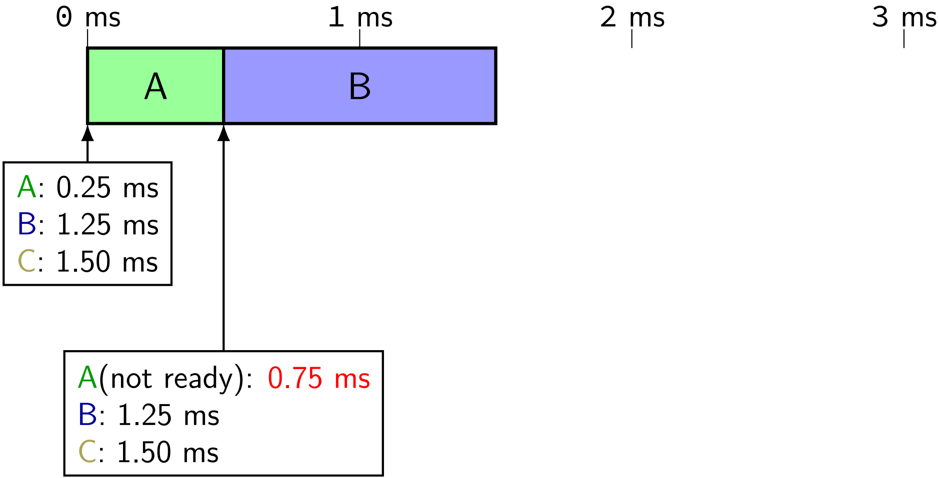
- (catch up from prior virtual time)

not runnable for a while? get bounded advantage

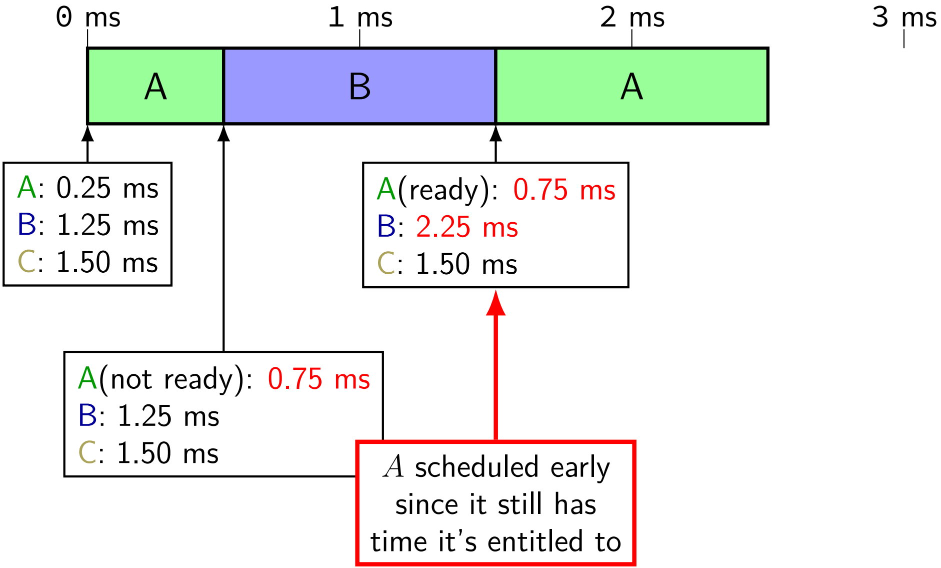
A doesn't use whole time...

0 ms　　1 ms　　2 ms　　3 ms

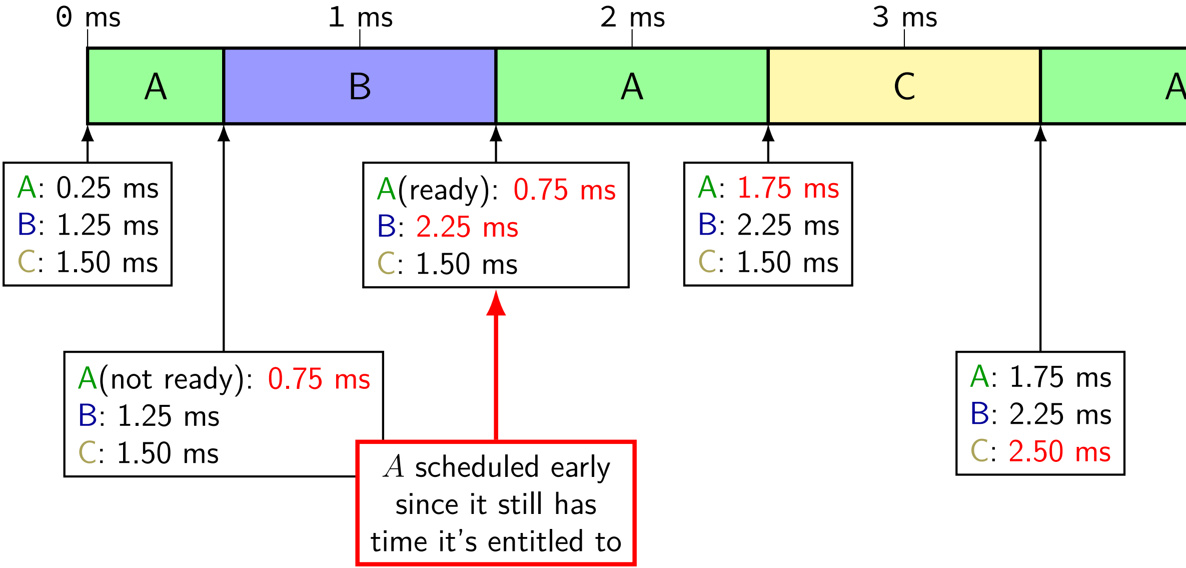
A doesn't use whole time...



A doesn't use whole time...



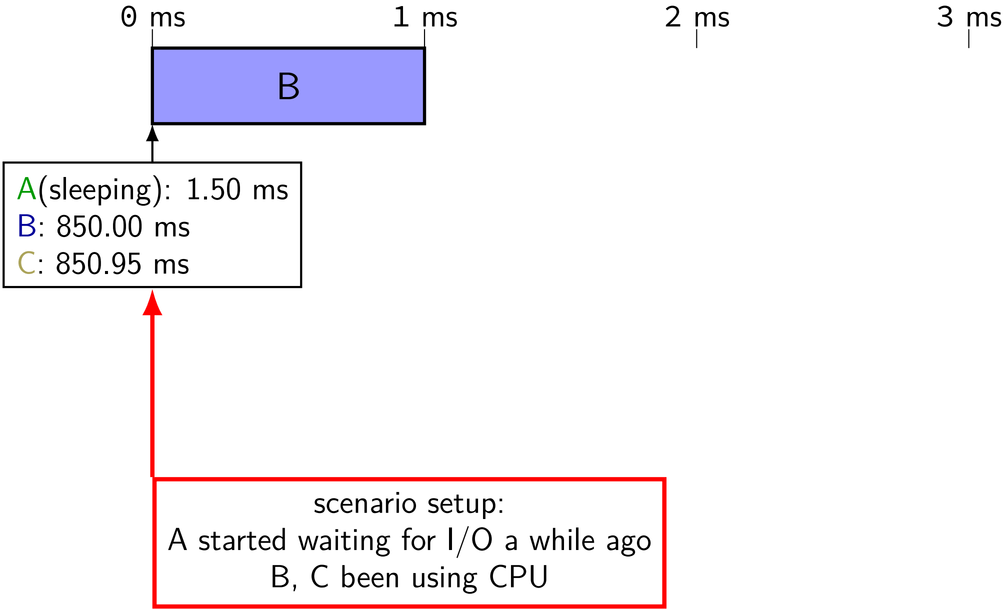
A doesn't use whole time...



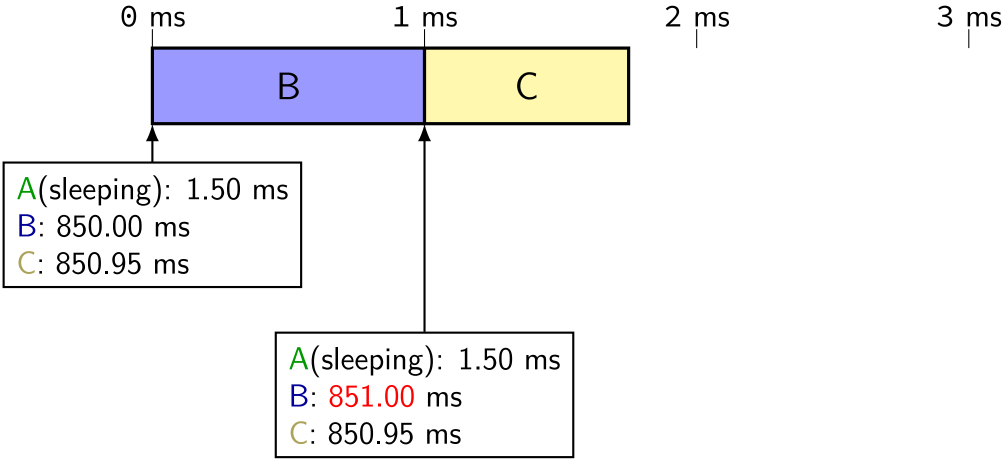
A's long sleep...

0 ms 1 ms 2 ms 3 ms

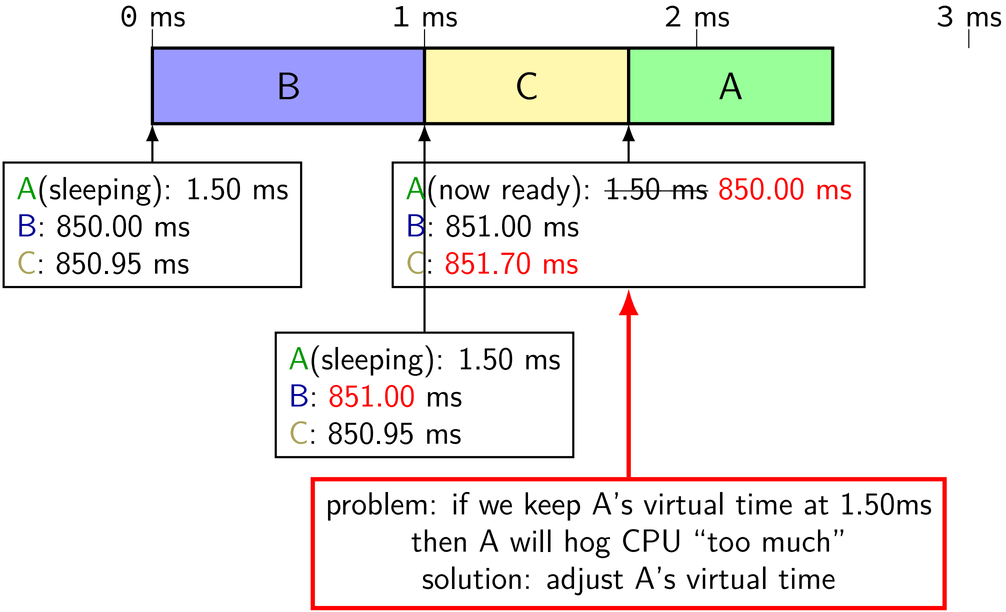
A's long sleep...



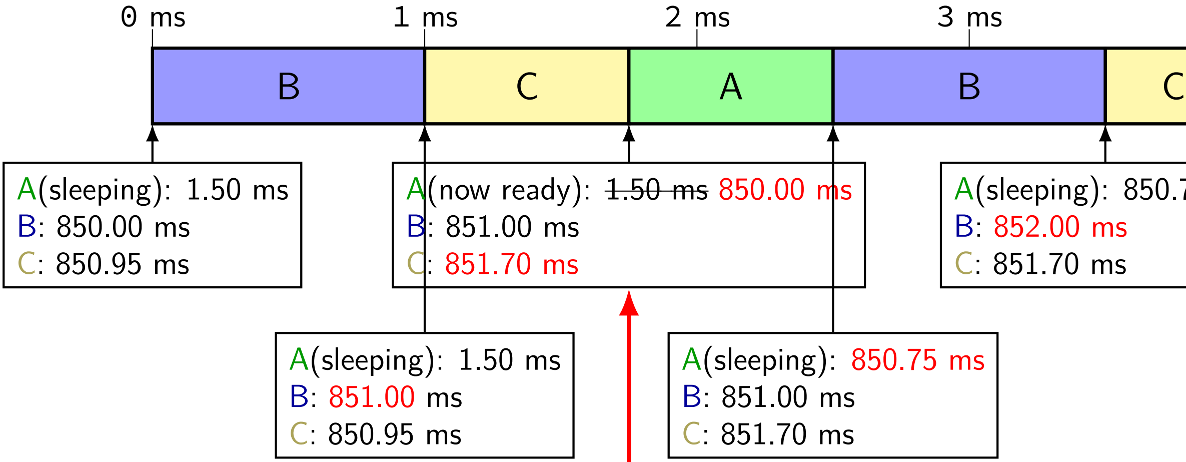
A's long sleep...



A's long sleep...



A's long sleep...



problem: if we keep A's virtual time at 1.50ms
then A will hog CPU "too much"
solution: adjust A's virtual time

handling *proportional* sharing

solution: multiply used time by weight

e.g. 1 ms of CPU time costs process 2 ms of virtual time

higher weight $\implies$ process less favored to run

CFS quantum lengths goals

first priority: constrain minimum quantum length (default: 0.75ms)

avoid too-frequent context switching

second priority: run every process “soon” (default: 6ms)

avoid starvation

CFS quantum lengths goals

first priority: constrain minimum quantum length (default: 0.75ms)

avoid too-frequent context switching

second priority: run every process "soon" (default: 6ms)

avoid starvation

quantum $\approx$ max(fixed window / num processes, minimum quantum)

CFS: avoiding excessive context switching

conflicting goals:

schedule newly ready tasks immediately

(assuming less virtual time than current task)

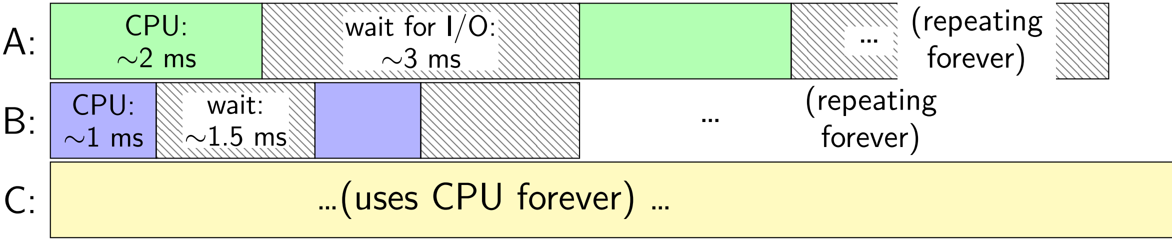
avoid excessive context switches

CFS rule:
if virtual time of new task $<$ current virtual time by threshold

default threshold: 1 ms

(otherwise, wait until quantum is done)

CFS exercise



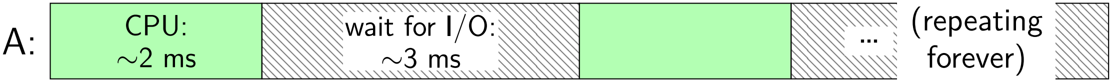
suppose programs A, B, C with alternating CPU + I/O as above

with CFS (and equal weights), about what portion of CPU does program A get?

your answer might depend on scheduler parameters

recall: limit on 'advantage' of programs waking from sleep

CFS exercise: maximum time for A



A running alone: A runs 2/5ths of the time

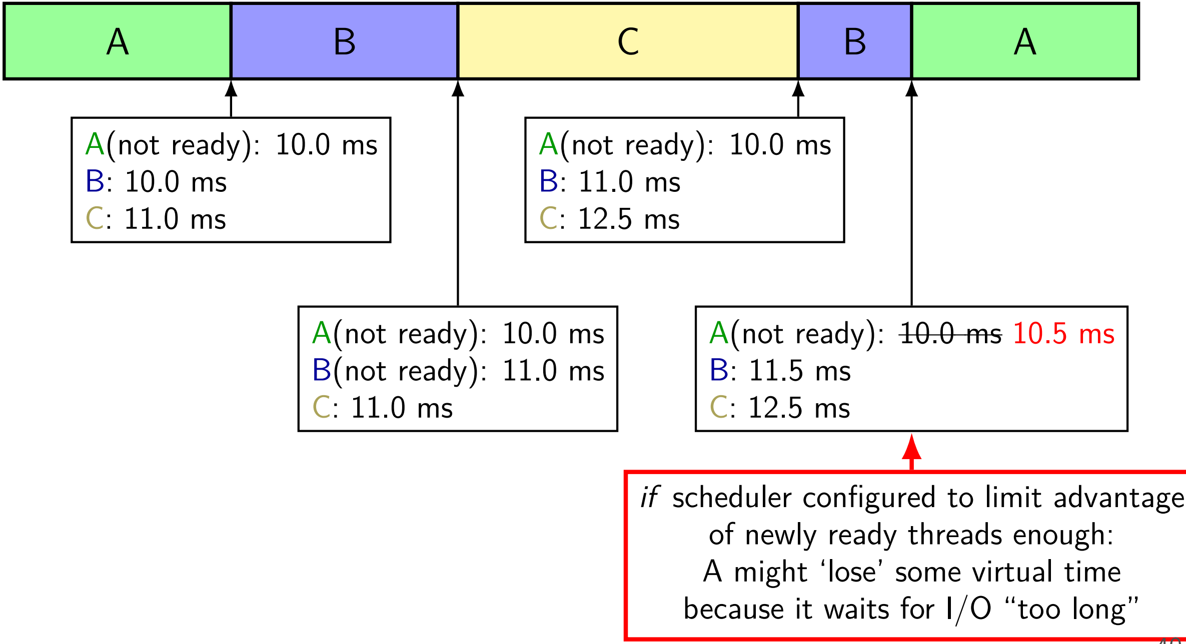
A, B, C sharing fairly: each runs 1/3rd of the time

result: A runs at most 1/3rd of the time...

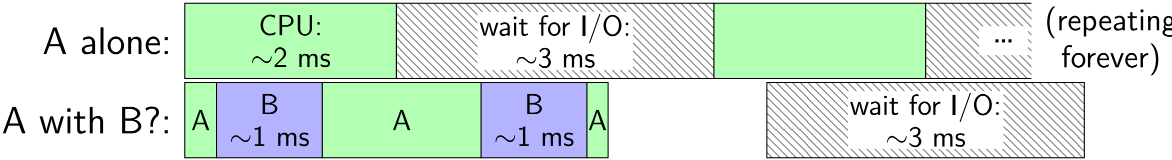
unless B somehow doesn't get its full share because of I/O

(because of being interrupted by A too much?)

CFS exercise: A disadvantage from sleep



CFS exercise: A interrupted by B?



combined with A losing 'banked' virtual time (so B can interrupt it sometimes),
could prevent A from running more

A interrupted by B a bunch sometimes...?

- depends on B's virtual time, etc.

might not start I/O as often

might not be able to run 1/3rd of the time

- assuming I/O is like disk, not keyboard

other CFS parts

dealing with multiple CPUs

handling groups of related tasks

special 'idle' or 'batch' task settings

...

CFS versus others

very similar to *stride scheduling*

- presented as a deterministic version of lottery scheduling
- Waldspurger and Weihl, "Stride Scheduling: Deterministic Proportional-Share Resource Management" (1995, same authors as lottery scheduling)

very similar to *weighted fair queuing*

- used to schedule network traffic
- Demers, Keshav, and Shenker, "Analysis and Simulation of a Fair Queuing Algorithm" (1989)

a note on multiprocessors

what about multicore?

extra considerations:

want two processors to schedule without waiting for each other

want to keep process on same processor (better for cache)

what process to preempt when three+ choices?

4.4BSD scheduler

4.4BSD / FreeBSD pre-2003 scheduler was a variation on MLFQ

64 priority levels, 100 ms quantum

same quantum at every priority

priorities adjusted periodically

- in retrospect not good for performance — iterate through all threads
- part of why FreeBSD stopped using this scheduler

priority of threads that spent a lot of time waiting for I/O increased

priority of threads that used a lot of CPU time decreased

real-time

so far: “best effort” scheduling

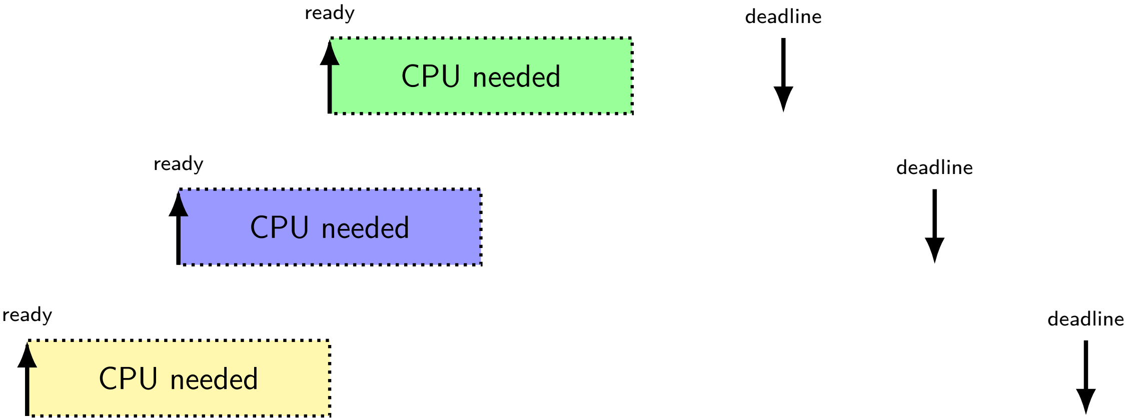
- best possible (by some metrics) given some work

alternate model: need gaurnetees

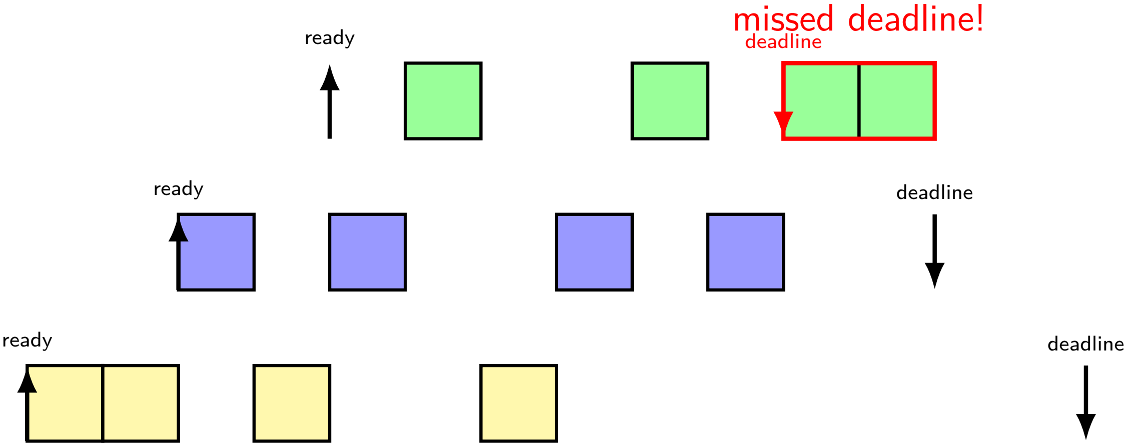
deadlines imposed by real-world

- process audio with 1ms delay
- computer-controlled cutting machines (stop motor at right time)
- car brake+engine control computer
- ...

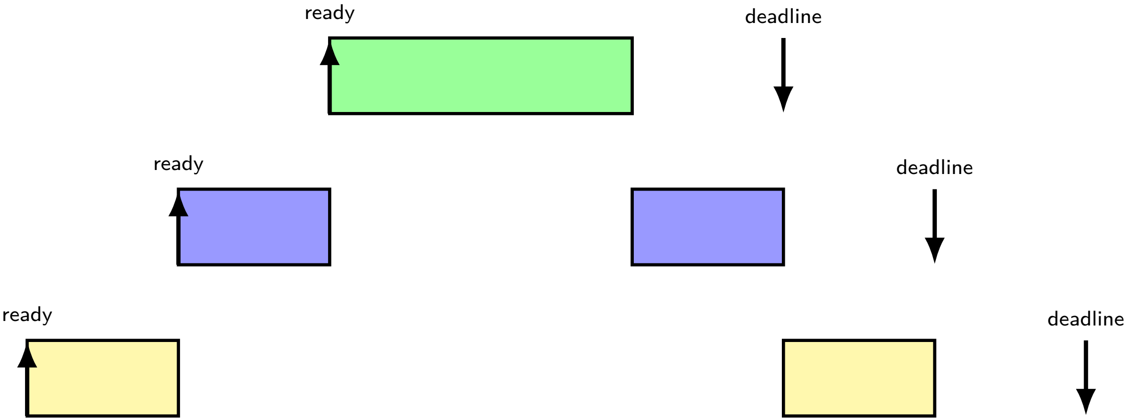
real time example: CPU + deadlines



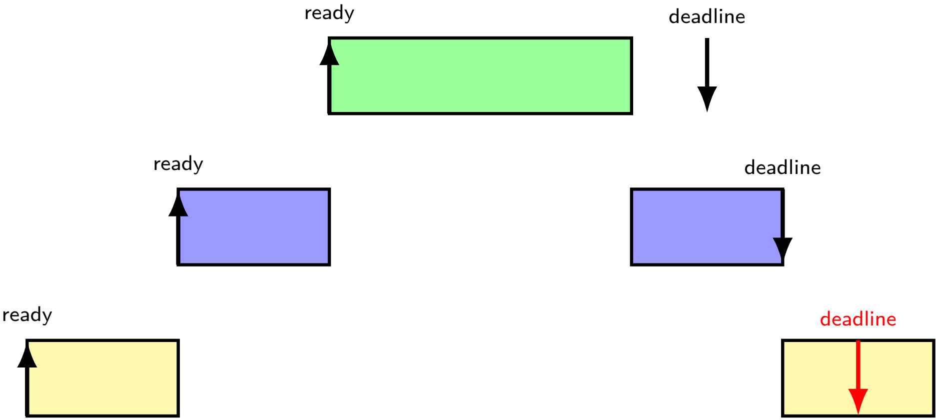
example with RR



earliest deadline first



impossible deadlines



no way to meet all deadlines!

admission control

given *worst-case* runtimes, start times, deadlines, scheduling algorithm,...

figure out whether it's possible to gaurentee meeting deadlines

- details on how — not this course (probably)

if not, then

- change something so they can?
- don't ship that device?
- tell someone at least?

earliest deadline first and...

earliest deadline first does *not* (even when deadlines met)

- minimize response time
- maximize throughput
- maximize fairness

exercise: give an example

other real-time schedulers

typical real time systems: *periodic tasks with deadlines*

  “*rate monotonic*”

commonly approximate EDF with lower period = higher priority

  easier to implement than true EDF

well-known method to determine if schedule is admissible

  = won't exceed deadline (under some assumptions)