changelog

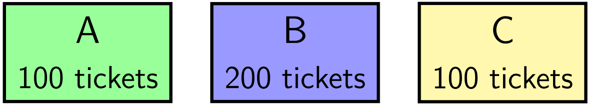
reorganize CFS slides to put text describing algorithm after examples
and move proportaionl share discussion till later

last time

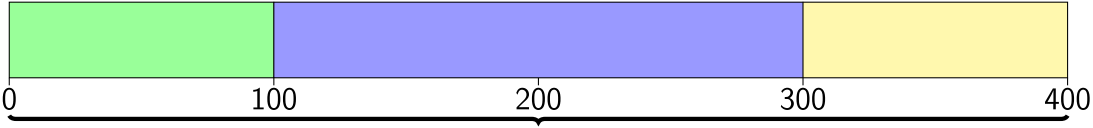
- throughput, turnaround time, fairness
  - throughput: time running actual programs (not scheduling stuff)
  - turnaround time $\sim$ responsiveness, maybe?
- first-come, first-served and round-robin
  - run threads in order they are listed
  - round robin: and stop running after time quantum
- time quantum — how long to let run?
  - longer: less context switches; shorter: probably more fair?
  - typical 1–100 ms
  - control context switch overhead
- optimizing turnaround time: shorter things first
- priority scheduling and starvation
- proportional share scheduling as priority compromise

lottery scheduling

every thread has a certain number of lottery tickets:



scheduling = lottery among ready threads:



choose random number in this range to find winner

simulating priority with lottery

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

very close to strict priority

lottery scheduling assignment

assignment: add lottery scheduling to xv6

extra system call: `settickets`

also counting of how often threads scheduled (for testing)

lottery scheduling assignment

assignment: add lottery scheduling to xv6

extra system call: `settickets`

also counting of how often threads 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 threads 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...)

minimizing turnaround time

recall: first-come, first-served best order:
had shortest CPU bursts first

$\rightarrow$ scheduling algorithm: 'shortest job first' (SJF)

$=$ same as priority where CPU burst length determines priority

...but without preemption for now

- priority $=$ job length doesn't quite work with preemption
- (preview: need priority $=$ remaining time)

a practical problem

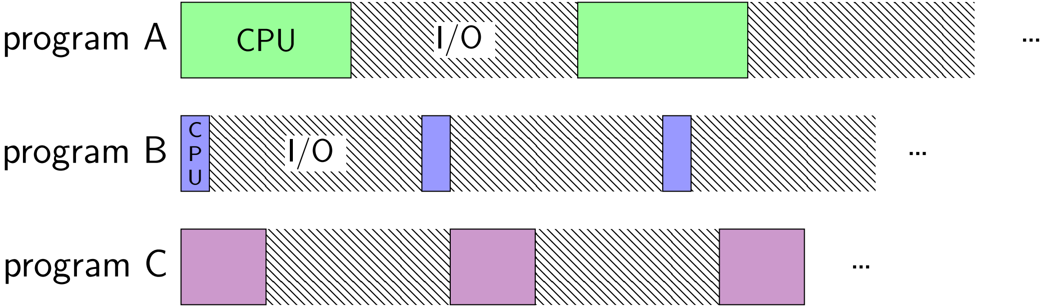
so we want to run the shortest CPU burst first

how do I tell which thread that is?

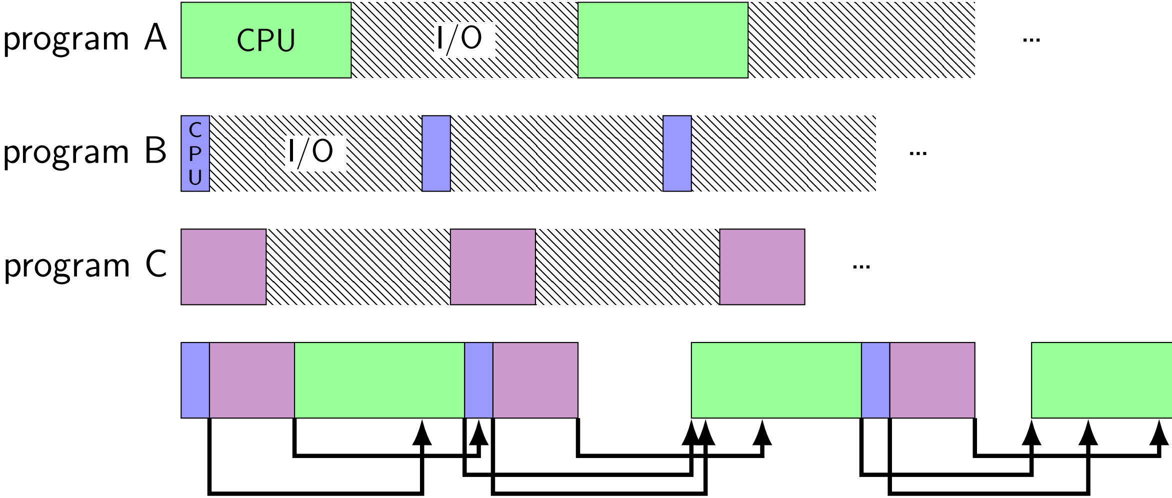
we'll deal with this problem later

...kinda

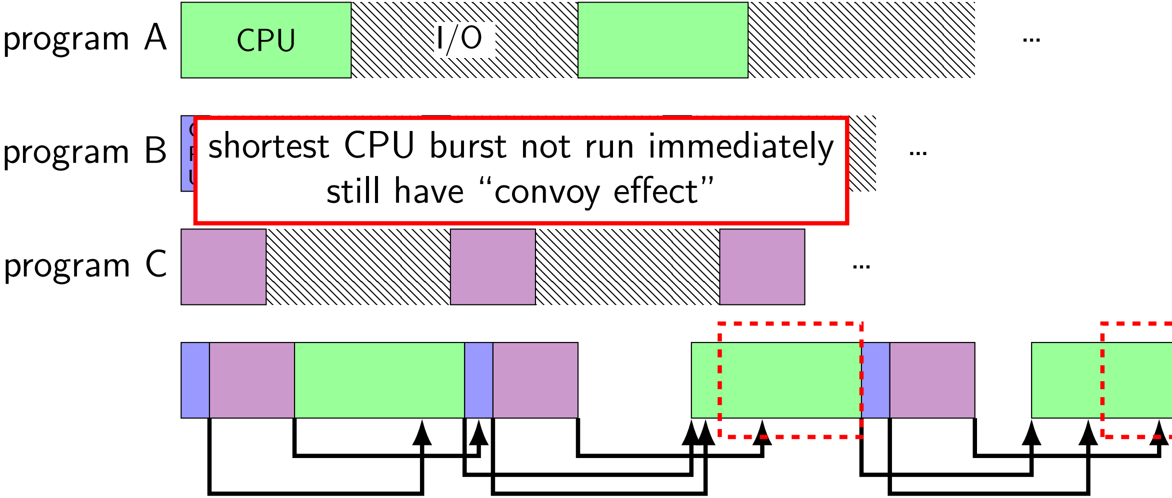
alternating I/O and CPU: SJF



alternating I/O and CPU: SJF



alternating I/O and CPU: SJF



preemption: definition

stopping a running program while it's still runnable

example: FCFS did not do preemption. RR did.

what we need to solve the problem:
'accidentally' ran long task, now need room for short one

adding preemption (1)

what if a long job is running, then a short job interrupts it?

- short job will wait for too long

solution is preemption — reschedule when new job arrives

- new job is shorter — run now!

adding preemption (2)

what if a long job is *almost done* running, then a medium job interrupts it?

- recall: priority = job length
- long job waits for medium job
- …for longer than it would take to finish
- worse than letting long job finish

adding preemption (2)

what if a long job is *almost done* running, then a medium job interrupts it?

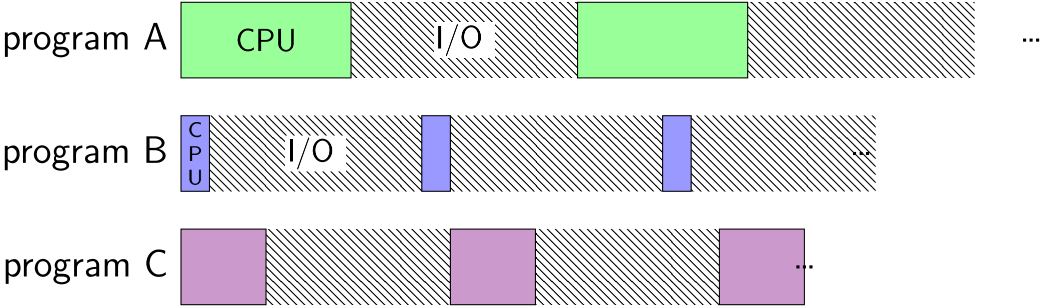
- recall: priority = job length
- long job waits for medium job
- …for longer than it would take to finish
- worse than letting long job finish

solution: priority = remaining time

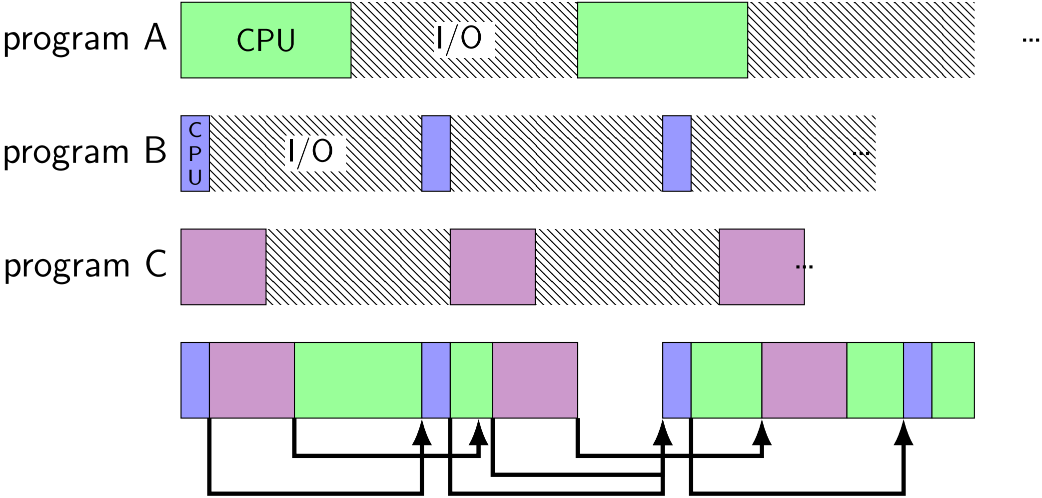
called shortest *remaining time* first (SRTF)

- prioritize by what's left, not the total

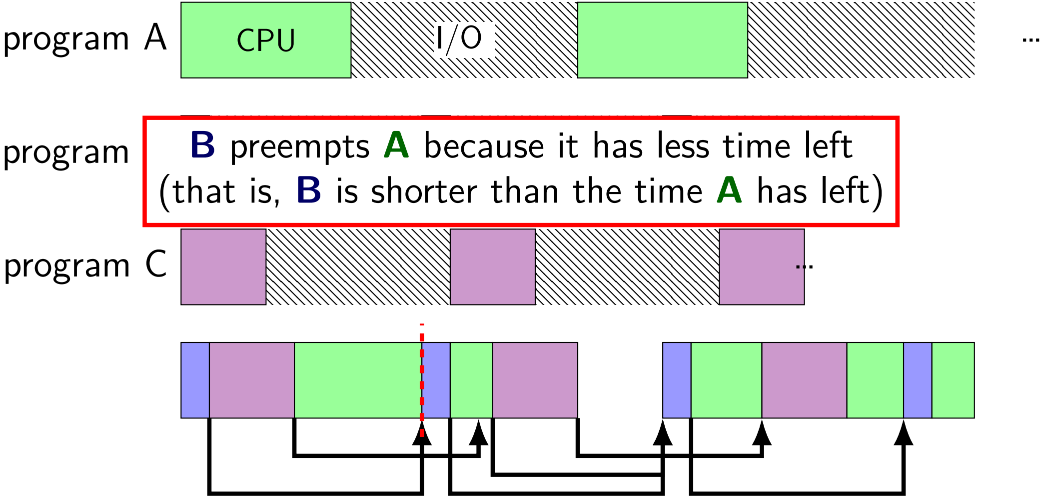
alternating I/O and CPU: SRTF



alternating I/O and CPU: SRTF

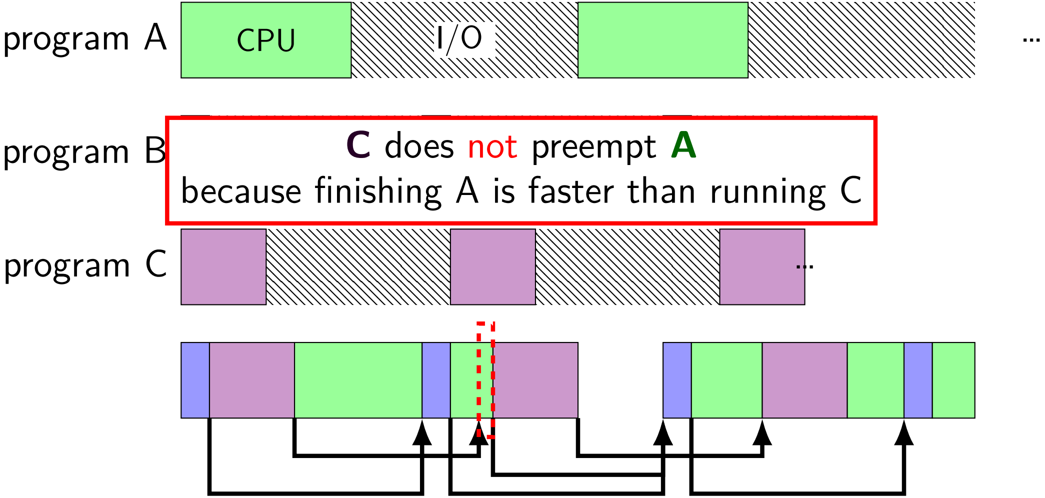


alternating I/O and CPU: SRTF



B preempts **A** because it has less time left
(that is, **B** is shorter than the time **A** has left)

alternating I/O and CPU: SRTF



C does not preempt **A**
because finishing A is faster than running C

SRTF, SJF are optimal (for turnaround time)

SJF minimizes turnaround time/waiting time
...if you disallow preemption/leaving CPU deliberately idle

SRTF minimizes turnaround time/waiting time
...if you ignore context switch costs

aside on names

we'll use:

SRTF for preemptive algorithm with remaining time

SJF for non-preemptive with total time=remaining time

might see different naming elsewhere/in books, sorry...

knowing job (CPU burst) lengths

seems hard

sometimes you can ask

- common in batch job scheduling systems

and maybe you'll get accurate answers, even

the SRTF problem

want to know CPU burst length

well, how does one figure that out?

the SRTF problem

want to know CPU burst length

well, how does one figure that out?

e.g. not any of these fields

```
uint sz;                        // Size of process memory (bytes)
pde_t* pgdir;                   // Page table
char *kstack;                   // Bottom of kernel stack for this pr
enum procstate state;           // Process state
int pid;                        // Process ID
struct proc *parent;            // Parent process
struct trapframe *tf;           // Trap frame for current syscall
struct context *context;        // swtch() here to run process
void *chan;                     // If non-zero, sleeping on chan
int killed;                     // If non-zero, have been killed
struct file *ofile[NOFILE];     // Open files
struct inode *cwd;              // Current directory
char name[16];                  // Process name (debugging)
```

predicting the future

worst case: need to run the program to figure it out

but heuristics can figure it out

- (read: often works, but no gaurentee)

key observation: CPU bursts now are like CPU bursts later

- intuition: interactive program with lots of I/O tends to stay interactive
- intuition: CPU-heavy program is going to keep using CPU

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”

will show one version; lots of small variations

multi-level feedback queues

classic strategy based on priority scheduling

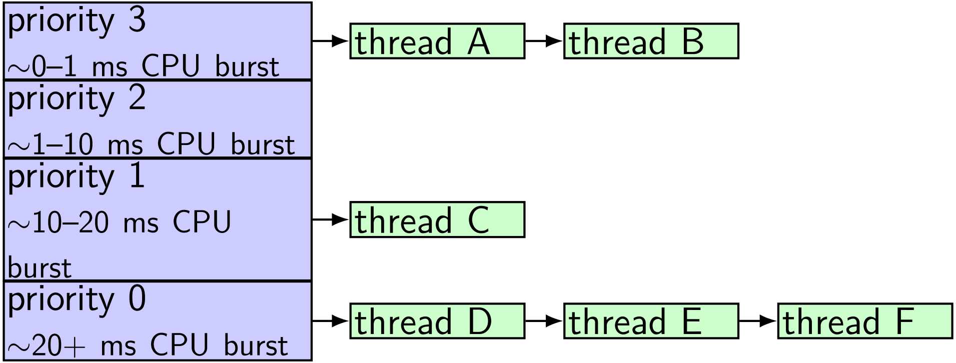
combines update time estimates and running shorter times first

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

small(ish) number of time estimate "buckets"

will show one version; lots of small variations

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

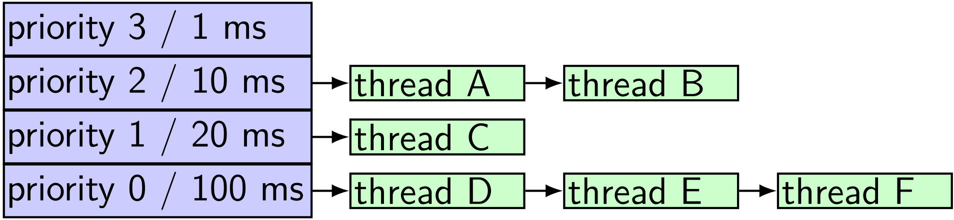
dynamically adjust priorities based on observed CPU burst times

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

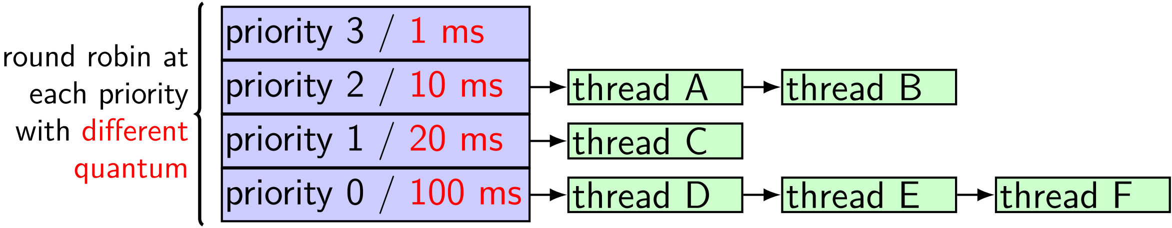
taking advantage of history

idea: priority = CPU burst length



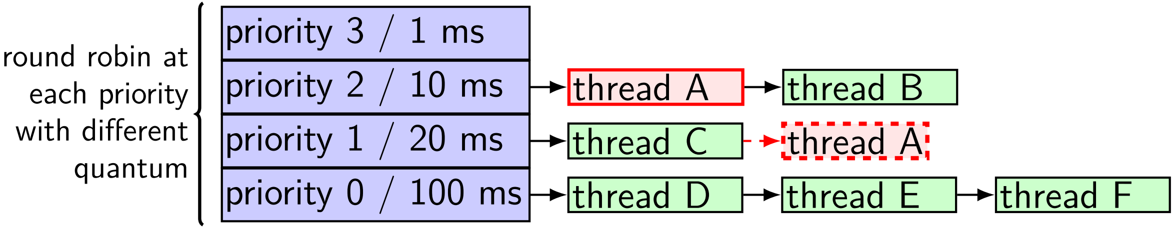
taking advantage of history

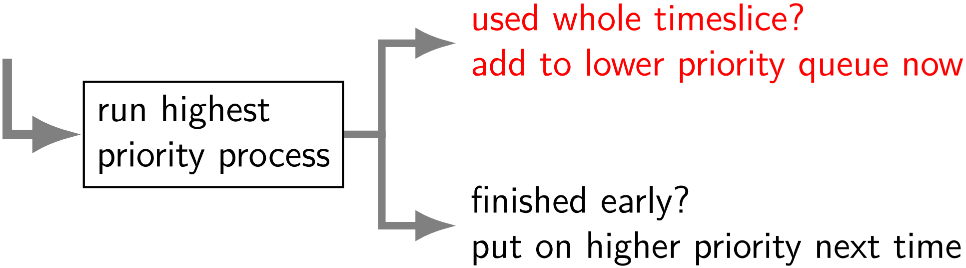
idea: priority = CPU burst length



taking advantage of history

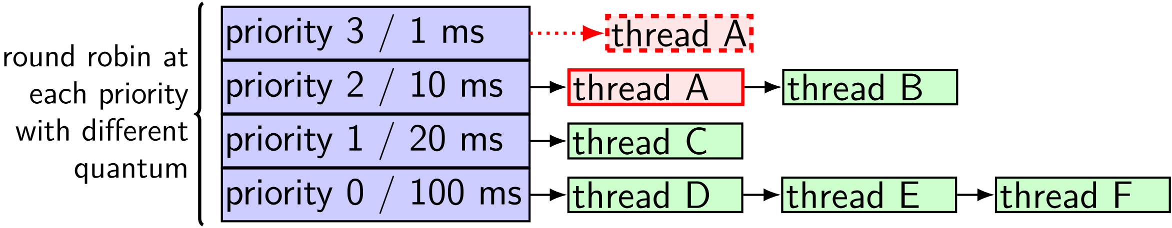
idea: priority = CPU burst length

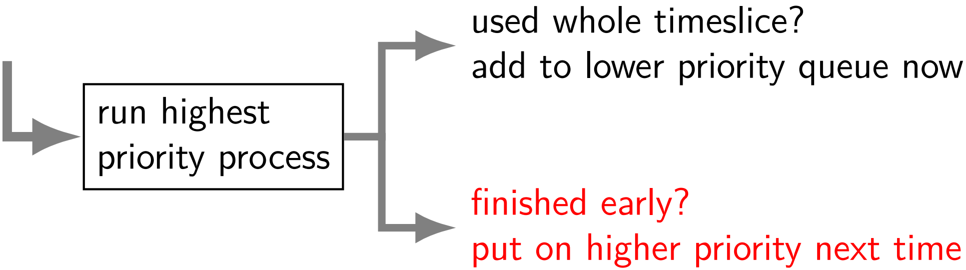




taking advantage of history

idea: priority = CPU burst length





multi-level feedback queue idea

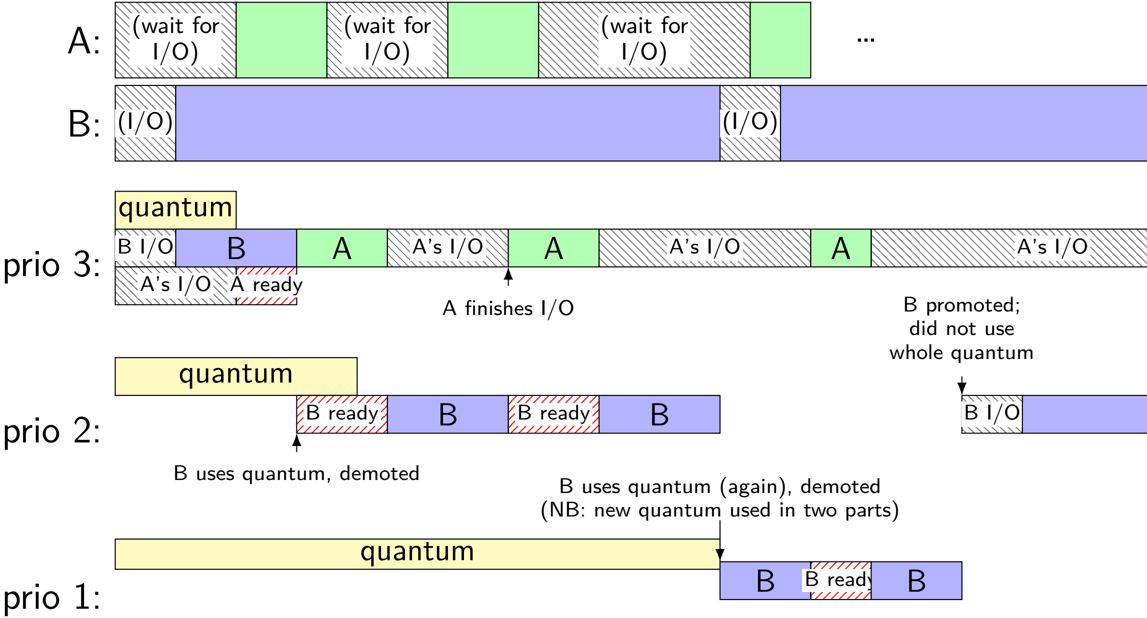
higher priority = shorter time quantum (before interrupted)

adjust priority *and* timeslice based on last timeslice

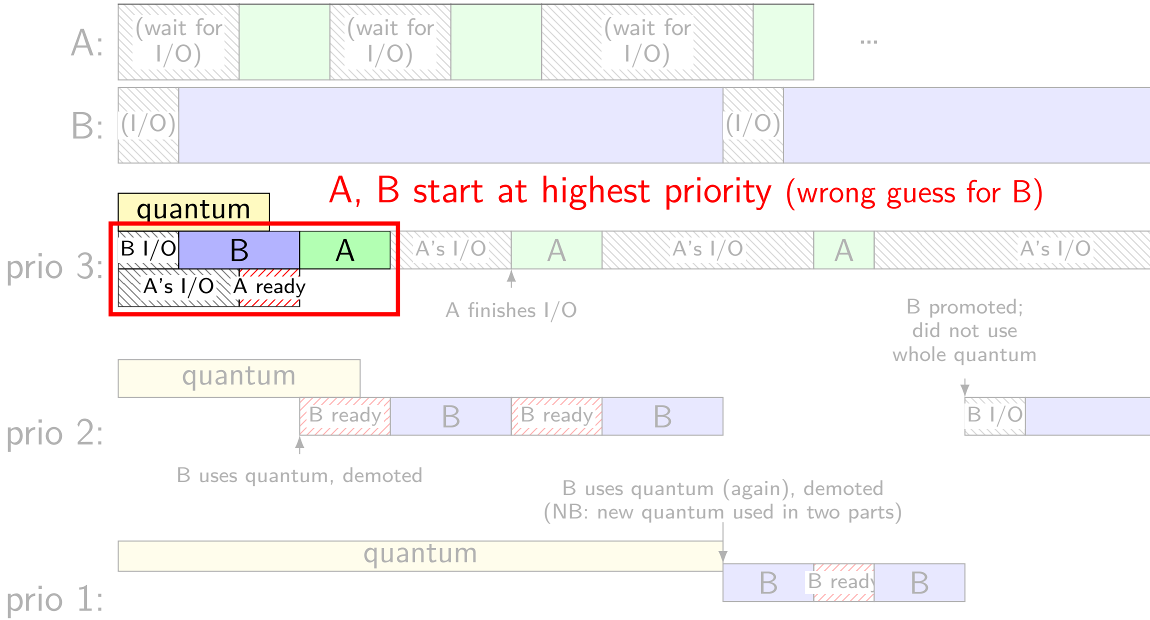
intuition: thread always uses same CPU burst length?
ends up at "right" priority

- 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.

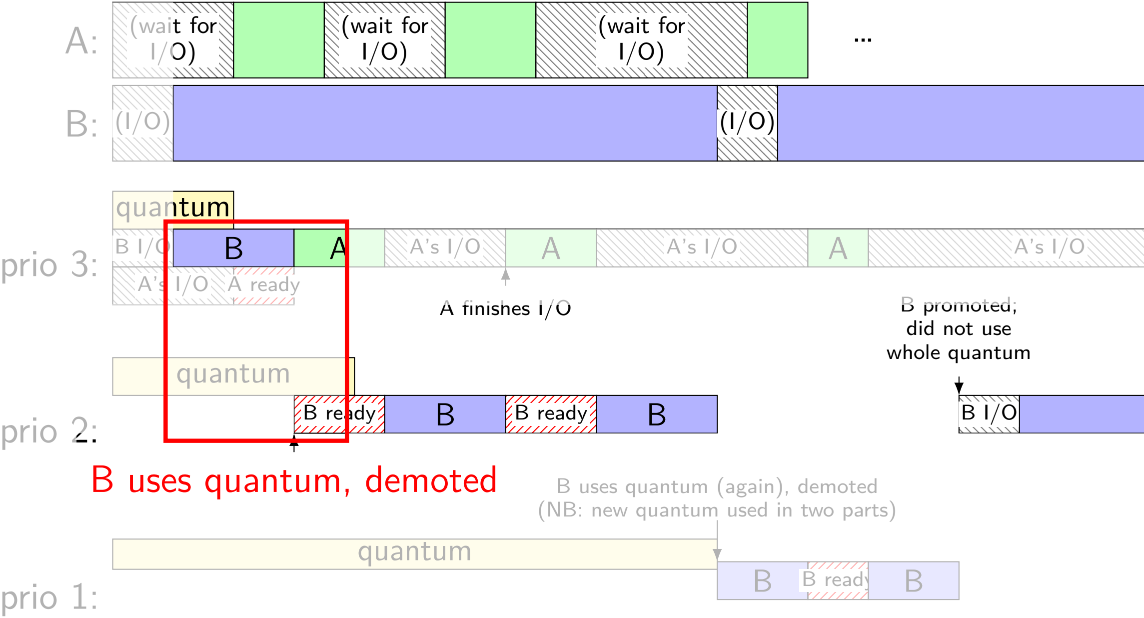
MLFQ example



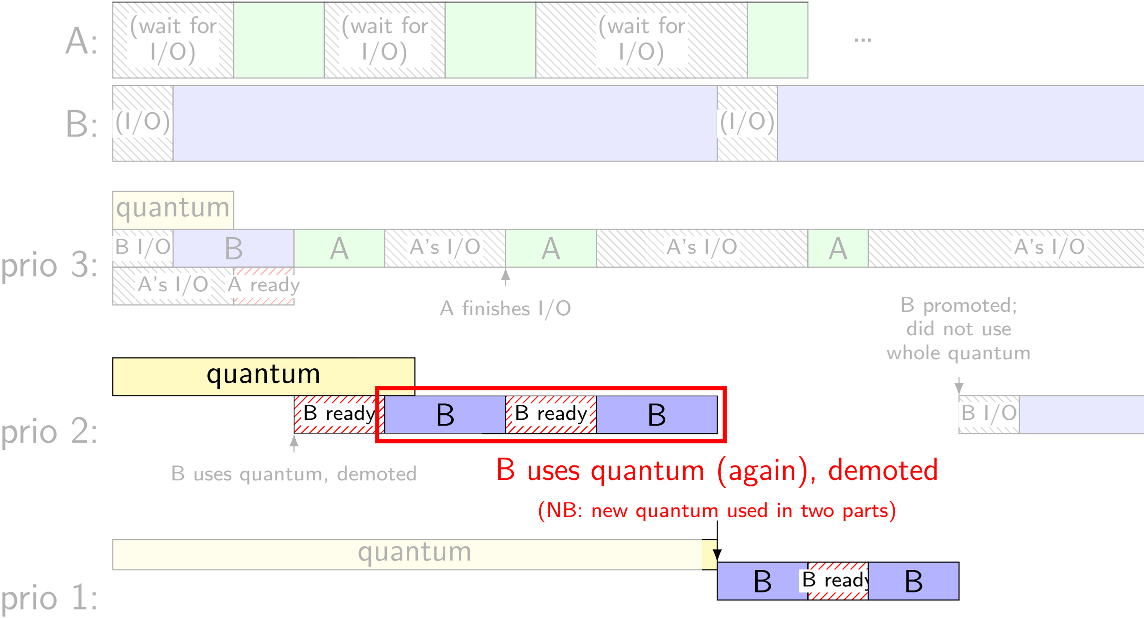
MLFQ example



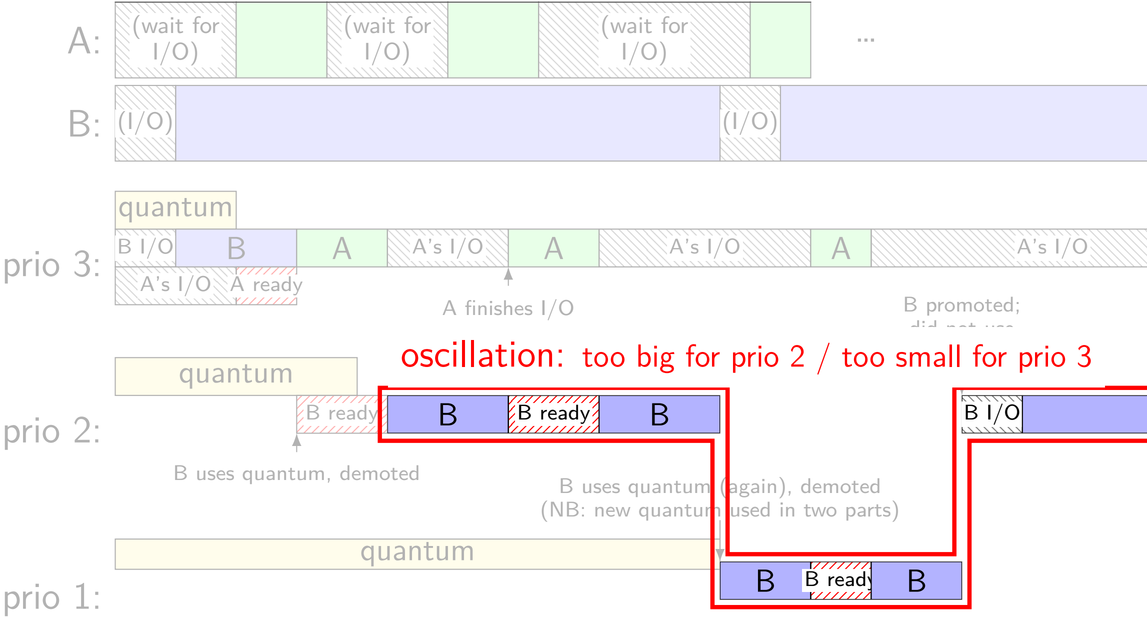
MLFQ example



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

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?

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

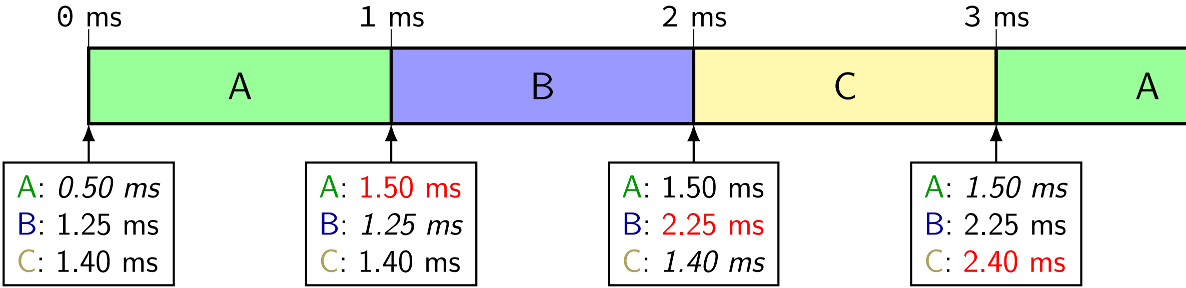
each thread has a *virtual runtime* ($\sim$ how long it's run)

incremented when run based how long it runs

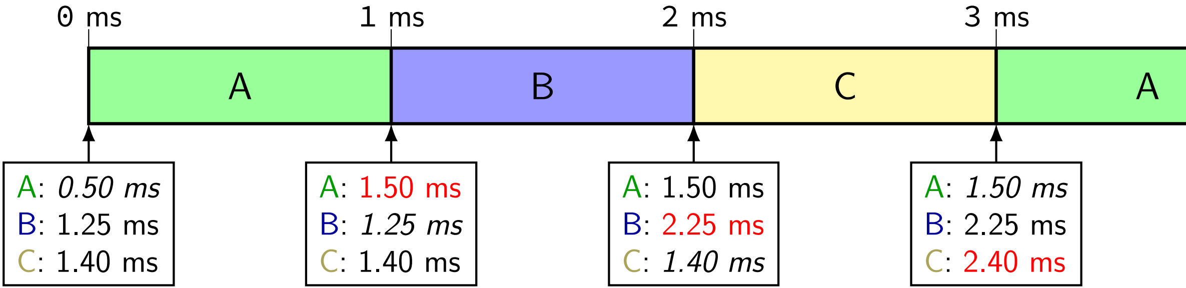
scheduling decision: run thread with lowest virtual runtime

- data structure: balanced tree

virtual time, always ready, 1 ms quantum



virtual time, always ready, 1 ms quantum

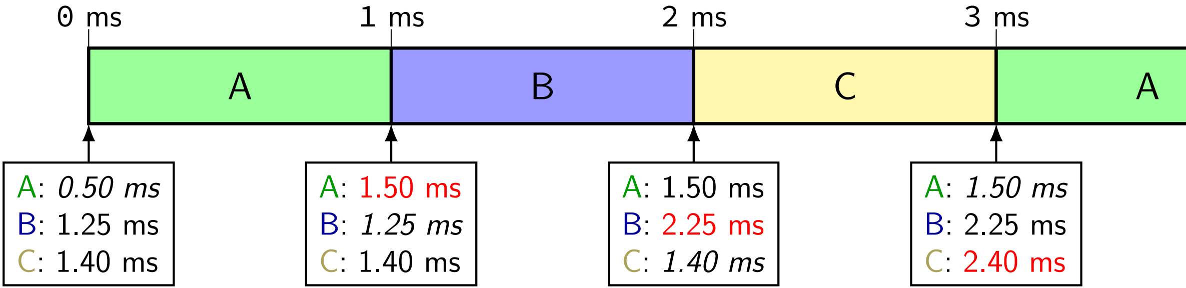


at each time:

update current thread's time

run thread with lowest total time

virtual time, always ready, 1 ms quantum

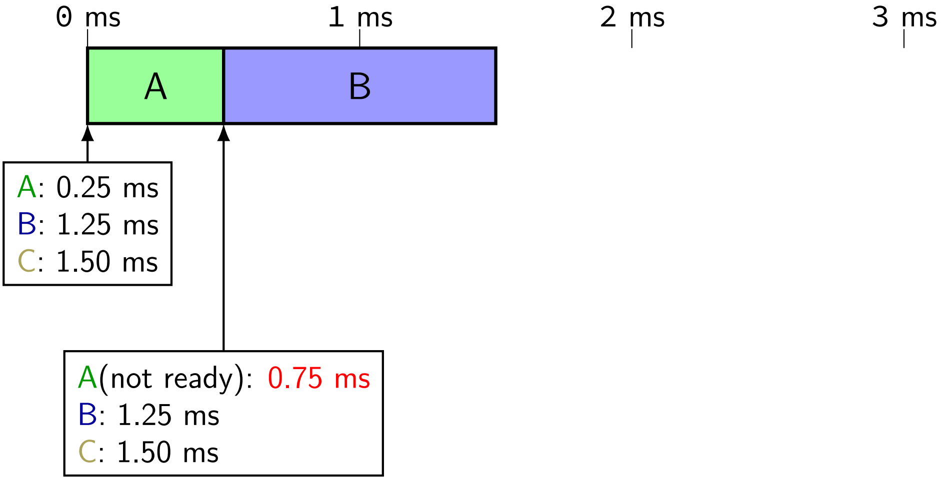


at each time:
update current thread's time
run thread with lowest total time

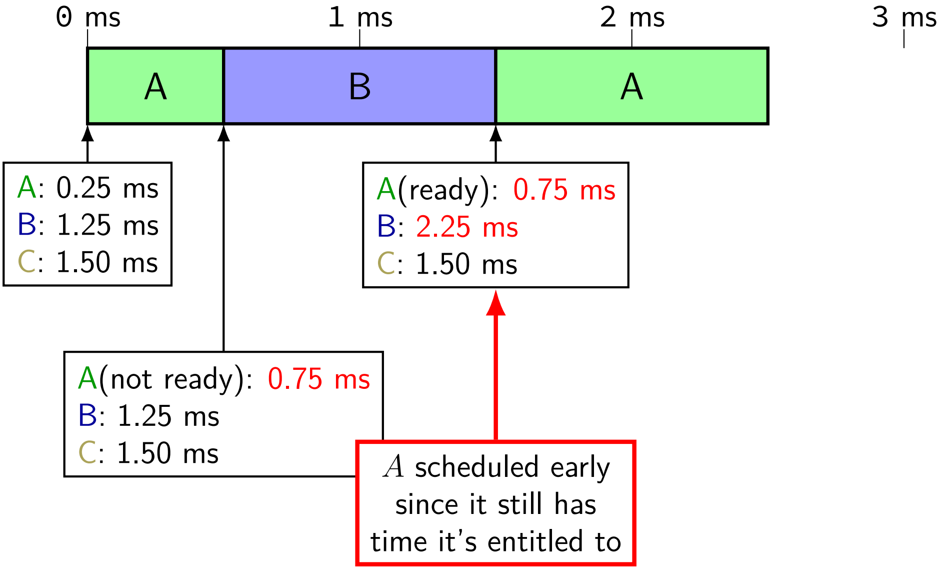
same effect as round robin
if everyone uses whole quantum

A doesn't use whole time...

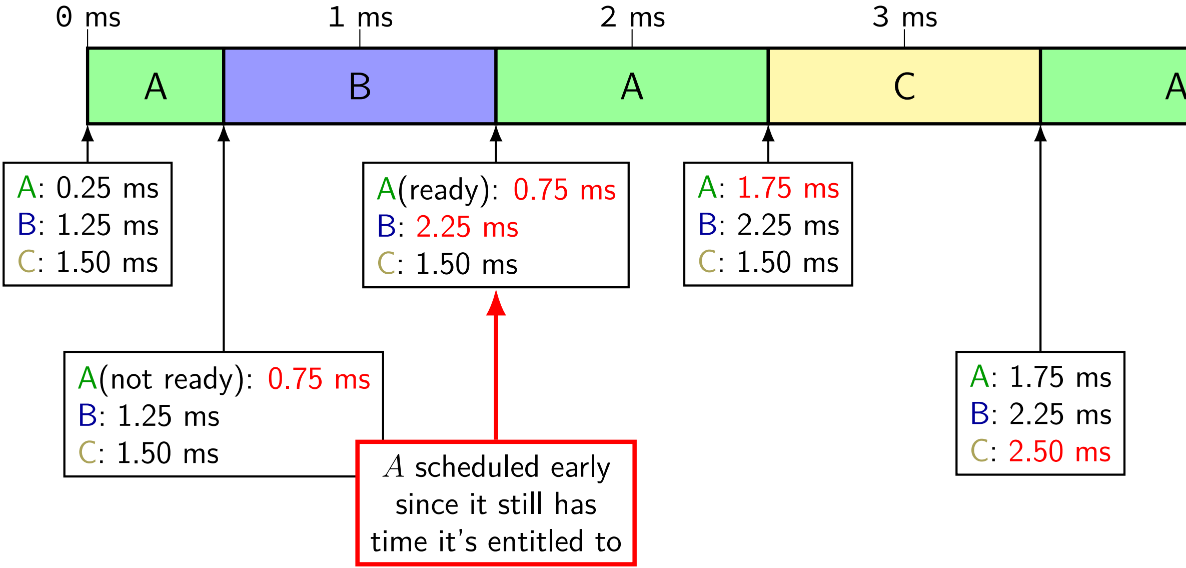
A doesn't use whole time…



A doesn't use whole time...



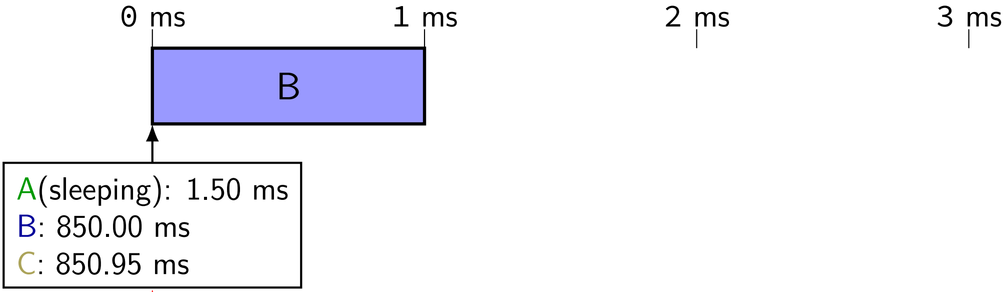
A doesn't use whole time...



A's long sleep...

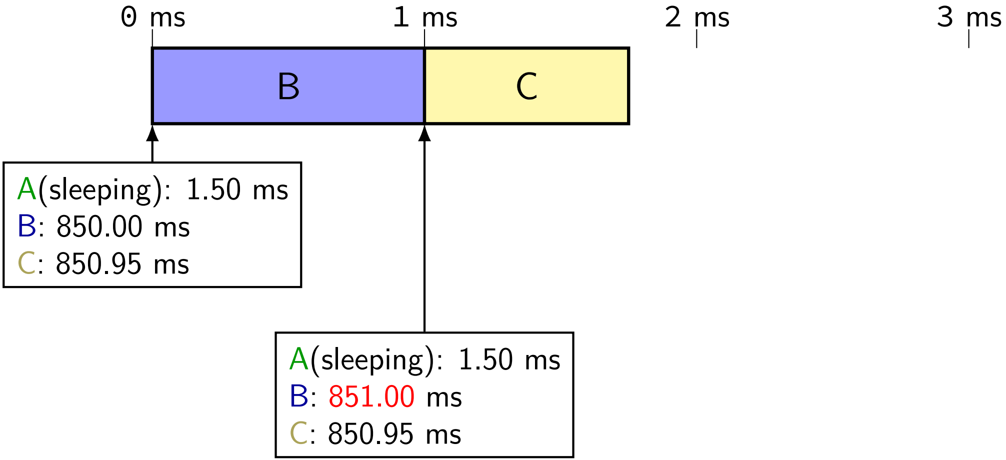
0 ms 1 ms 2 ms 3 ms

A’s long sleep...

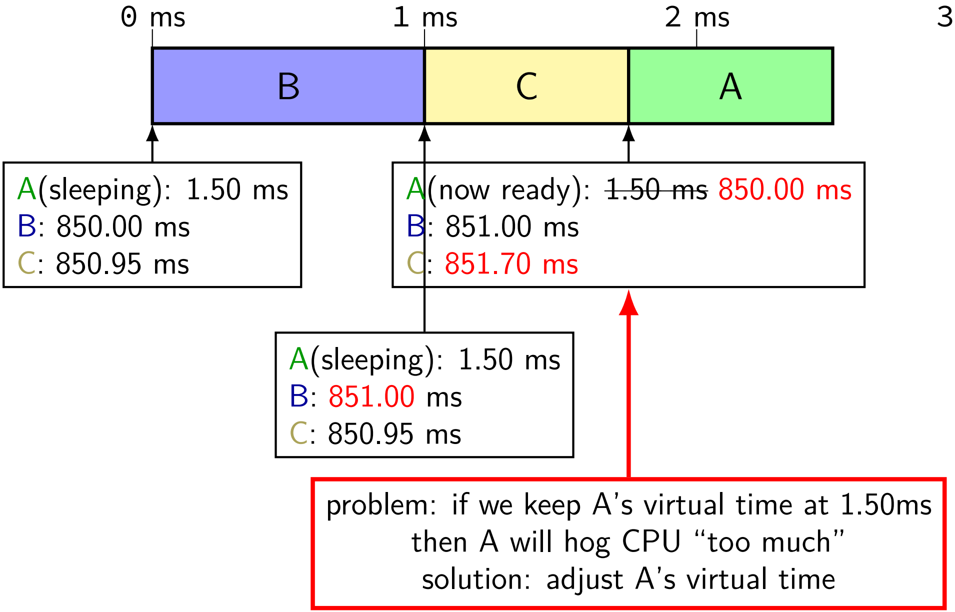


scenario setup:
A started waiting for I/O a while ago
B, C been using CPU

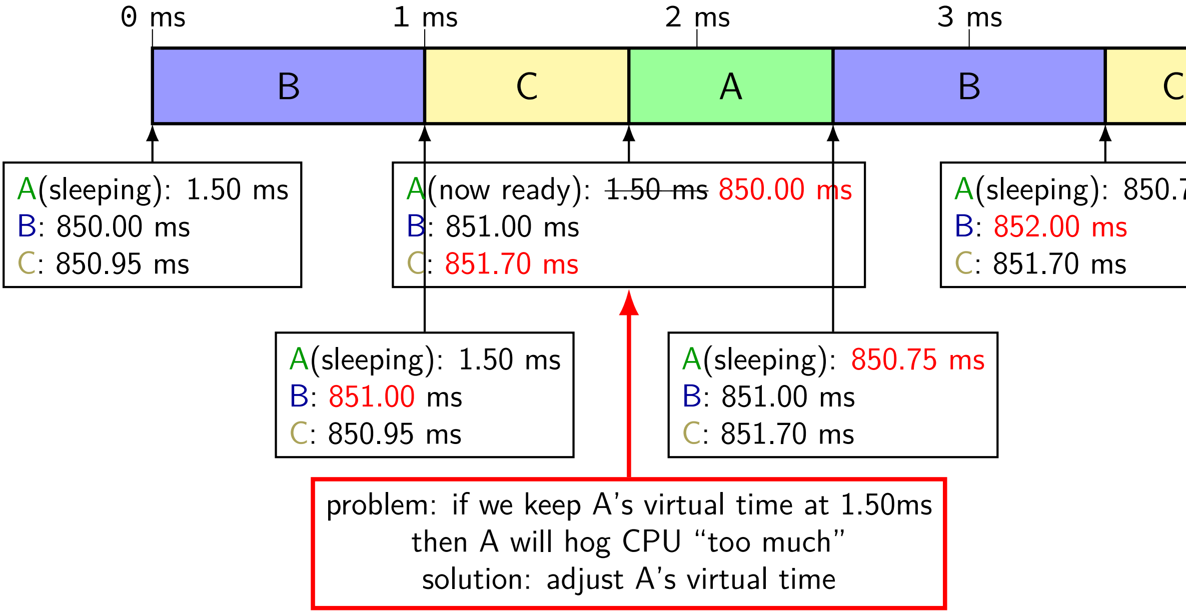
A's long sleep...



A's long sleep...



A's long sleep...



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

CFS: tracking runtime

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

incremented when run based how long it runs

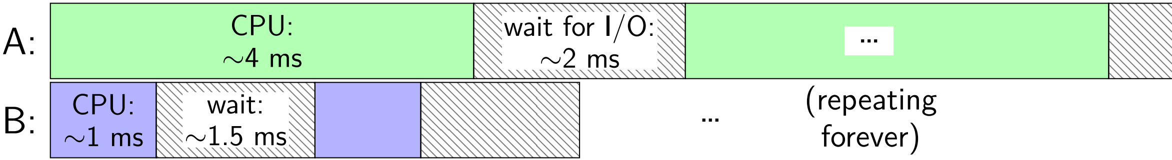
adjustments for threads that are new or were sleeping

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

scheduling decision: run thread with lowest virtual runtime

- data structure: balanced tree

CFS exercise (0c)



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

with CFS (and equal weights) and **no adjustments to virtual time for programs waking up from sleep**, **about** what portion of CPU does program A get?

exercise solution

if A, B, were running alone, could get at most 1/2 the CPU

B can't use that much time

so B will run 2/5ths of the time (the most it can)

so B will almost always have lower virtual time than A

A will get the remaining about 3/5ths

exception: time both A and B are both doing I/O

exception: extra time A gets to run if no preemption during its time quantum?

backup sides

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 variant 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

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)

real-time

so far: “best effort” scheduling

best possible (by some metrics) given some work

alternate model: need gaurnetees

deadlines imposed by real-world

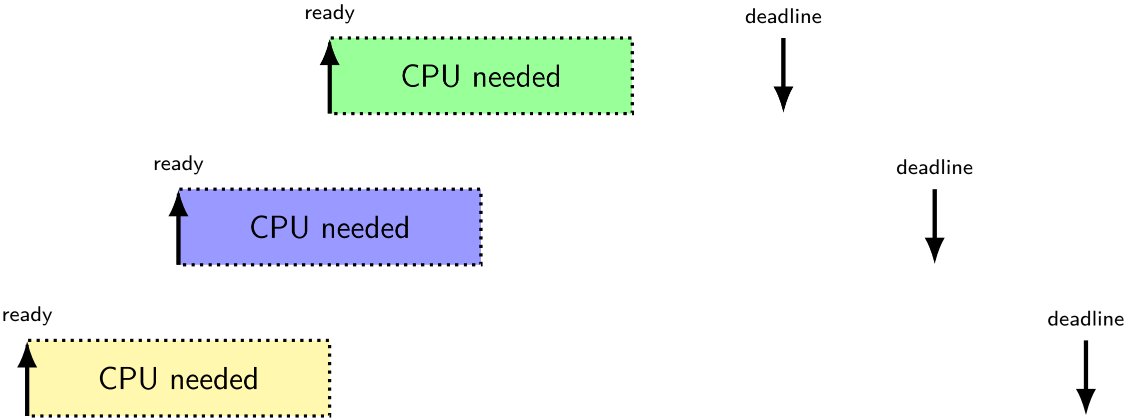
process audio with 1ms delay

computer-controlled cutting machines (stop motor at right time)

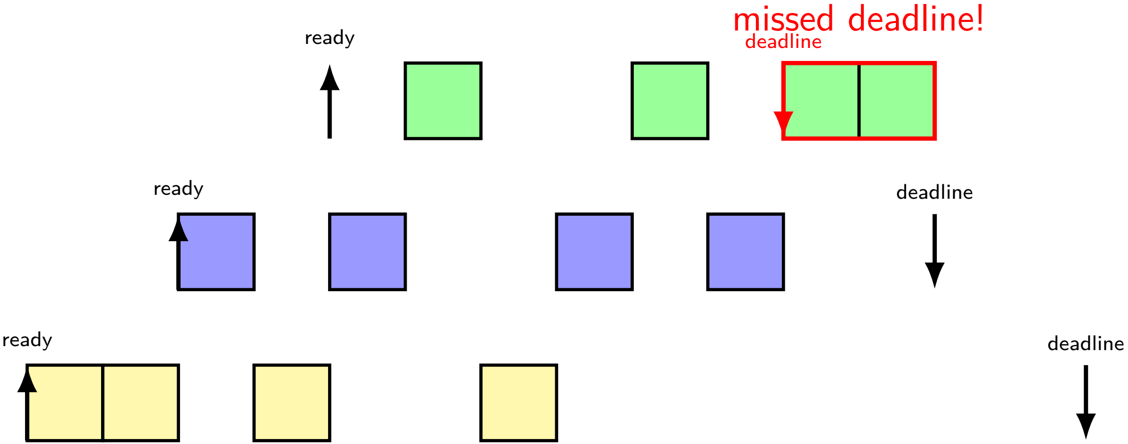
car brake+engine control computer

...

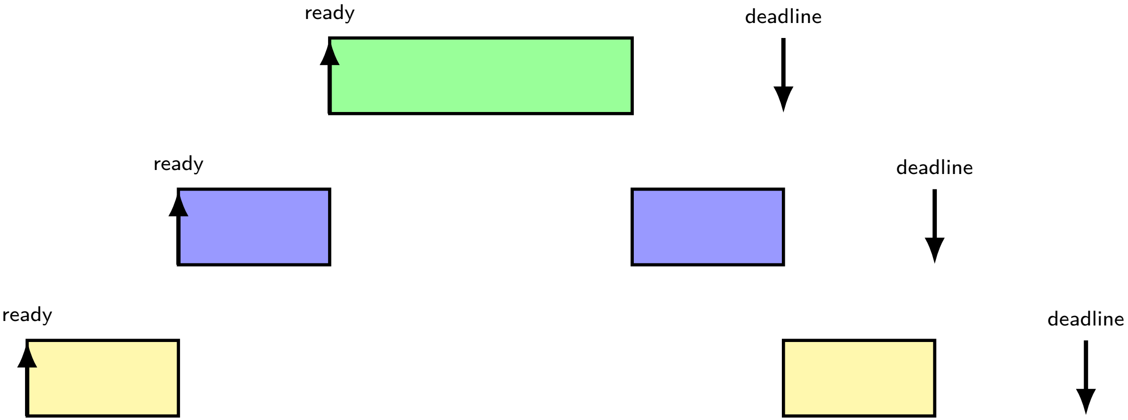
real time example: CPU + deadlines



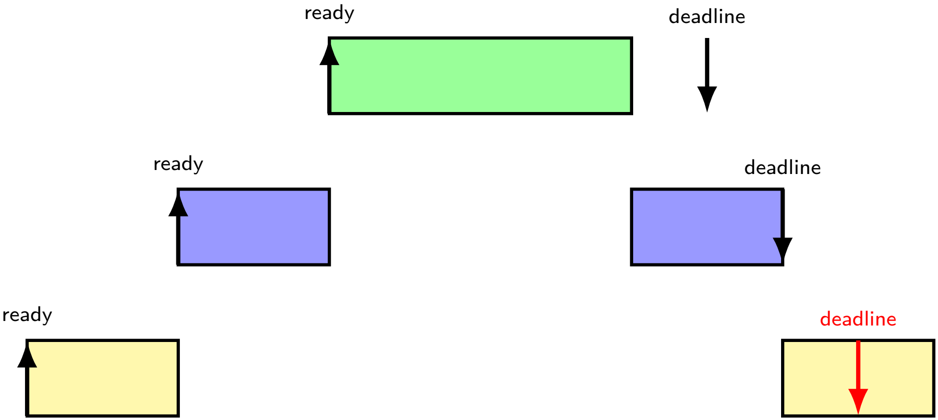
example with RR



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)

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

most fair

least fair

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

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)

backup slides