scheduling 3: MLFQ / proportional share

last time

CPU burst concept

scheduling metrics

- turnaround and wait time
- throughput
- fairness

first-come, first-served (FCFS) and round-robin (RR)

SJF (shortest job first) and SRTF (shortest remaining time first)

priority scheduling

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

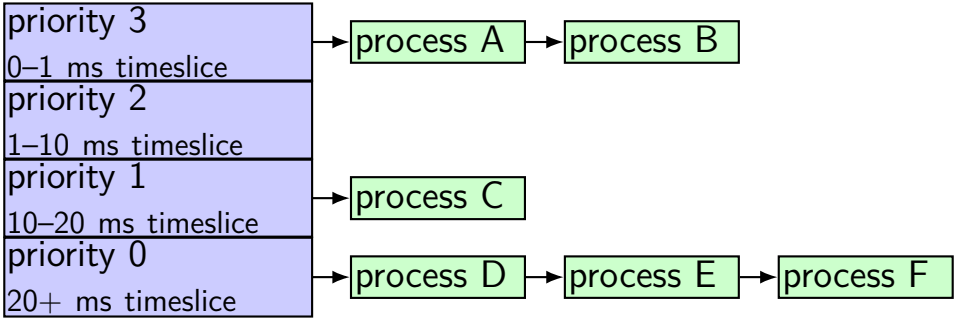
(read: often works, but no gaurentee)

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: setup



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

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

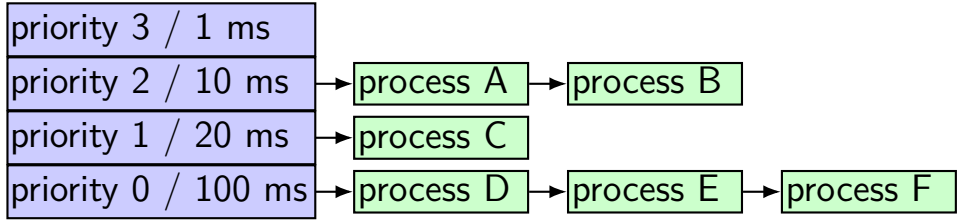
dynamically adjust priorities based on observed CPU burst times

priority level $\rightarrow$ 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

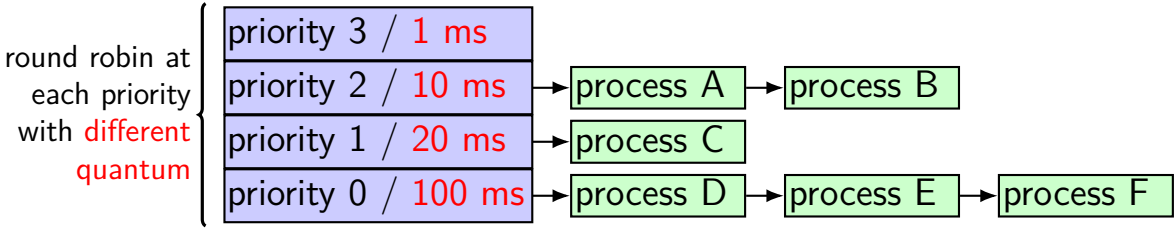
taking advantage of history

idea: priority = CPU burst length



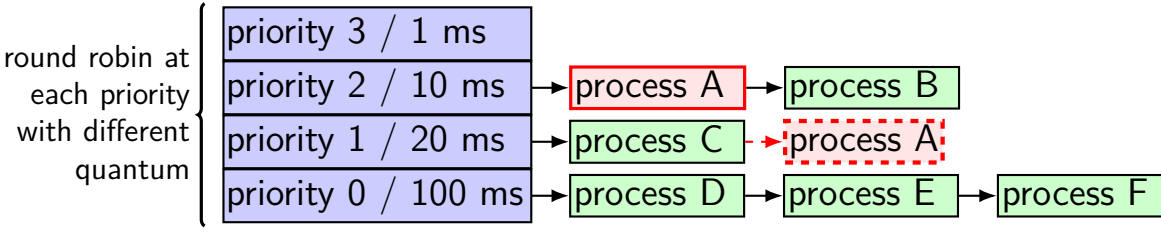
taking advantage of history

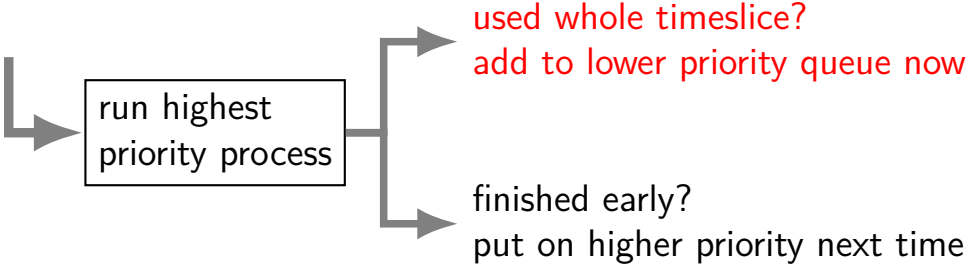
idea: priority = CPU burst length



taking advantage of history

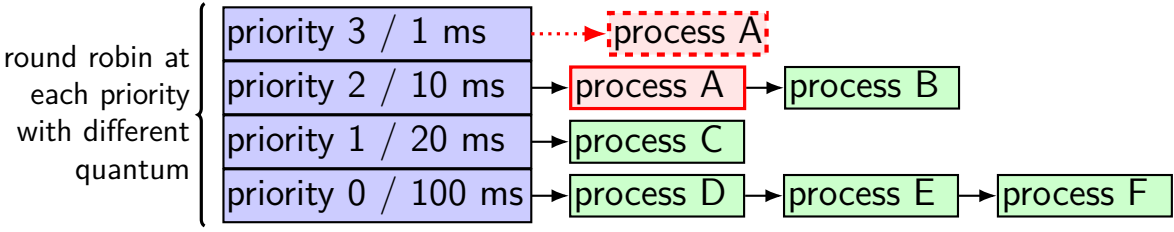
idea: priority = CPU burst length

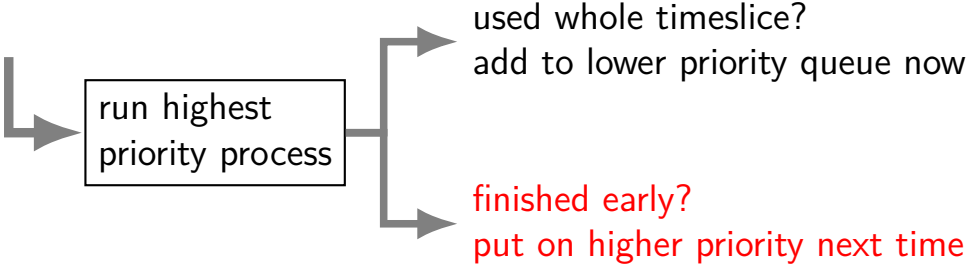




taking advantage of history

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: process 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.

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

providing fairness

an additional heuristic: avoid starvation

track processes that haven't run much recently

...and run them earlier than they "should" be

conflicts with SJF/SRTF goal

...but typically done by multi-level feedback queue implementations

other heuristics?

MFQ assumption: past CPU burst $\approx$ next one

could have other models of CPU bursts

- based on length of time since last runnable?
- fit to some statistical distribution?
- based on what I/O devices are open?

lots of possible scheduling heuristics...

policy versus mechanism

MFQ: example of implementing SJF-like policy with priority mechanism

common theme: one mechanism (e.g. priority) supporting many policies

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

aside: measuring fairness

one way: max-min fairness

choose schedule that maximizes the minimum resources (CPU time) given to any thread

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

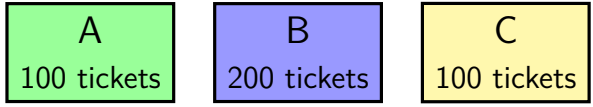
if thread A is twice as important as thread B, then...

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

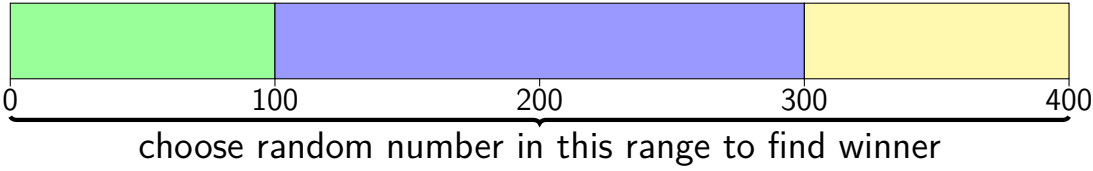
proportional share

lottery scheduling

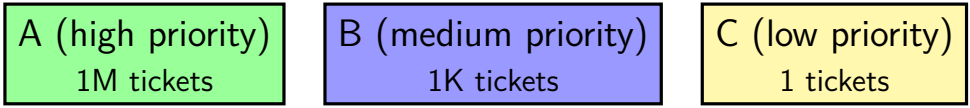
every thread has a certain number of lottery tickets:



scheduling = lottery among ready threads:



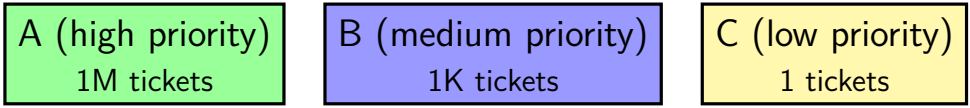
simulating priority with lottery



very close to strict priority

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

simulating priority with lottery



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 long processes run (for testing)

lottery scheduling assignment

assignment: add lottery scheduling to xv6

extra system call: `settickets`

also counting of how long processes run (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

process A: 1 ticket, always runnable

process 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

process A: 1 ticket, always runnable

process 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...)

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

lottery scheduler: run equally when both can run

result: B runs less than A

- 50% when both runnable
- 0% of the time when only A runnable (waiting on I/O)

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

lottery scheduler: run equally when both can run

result: B runs less than A

- 50% when both runnable
- 0% of the time when only A runnable (waiting on I/O)

is this fair? depends who you ask

one idea: B should get more tickets for waiting

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

...with a different solution to interactivity problem

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

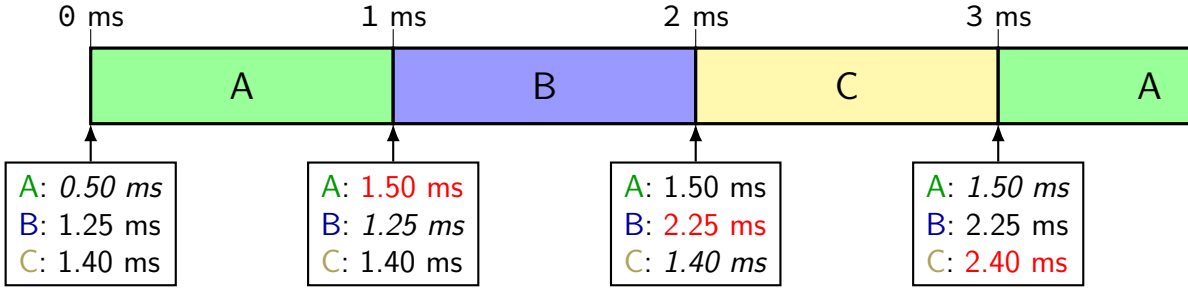
adjustments for threads that are new or were sleeping

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

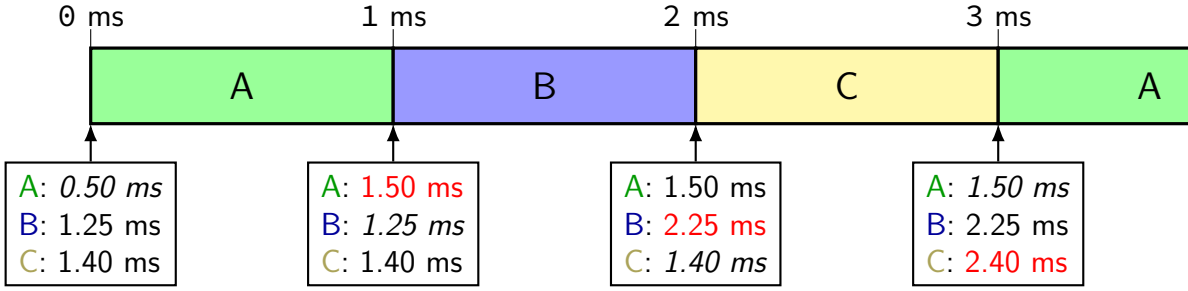
scheduling decision: run thread with lowest virtual runtime

- data structure: balanced tree

virtual time, always ready, 1 ms quantum



virtual time, always ready, 1 ms quantum

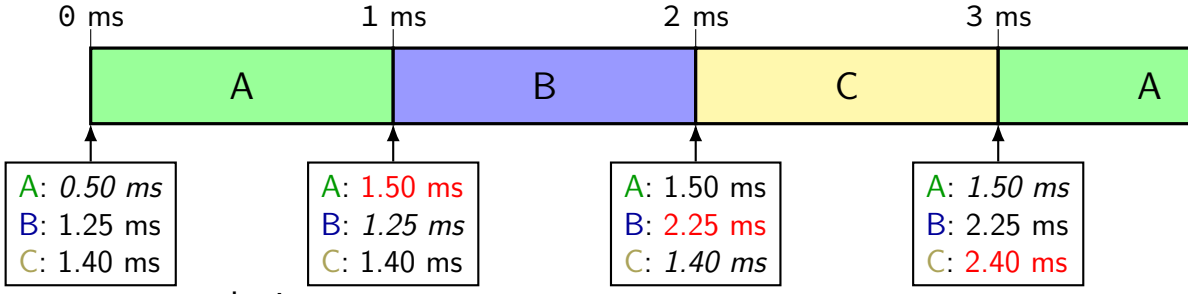


at each time:

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)

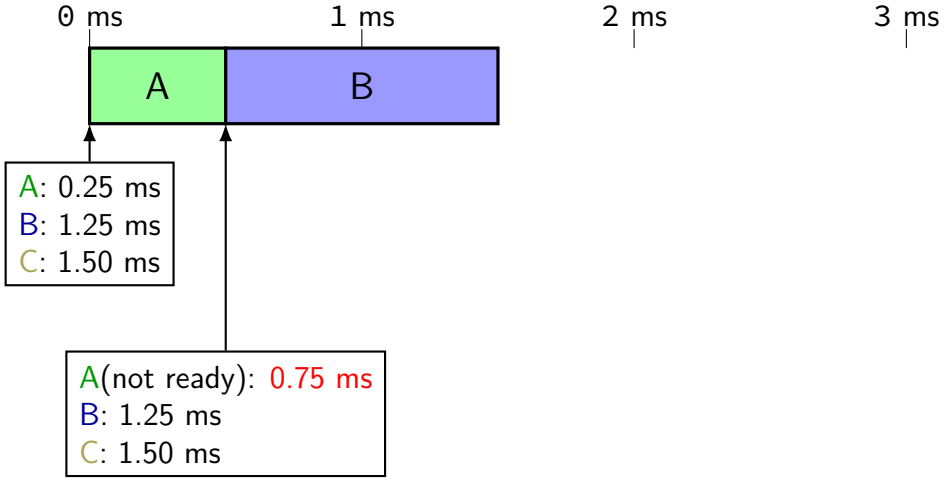
not runnable briefly? still get your share of CPU

- (catch up from prior virtual time)

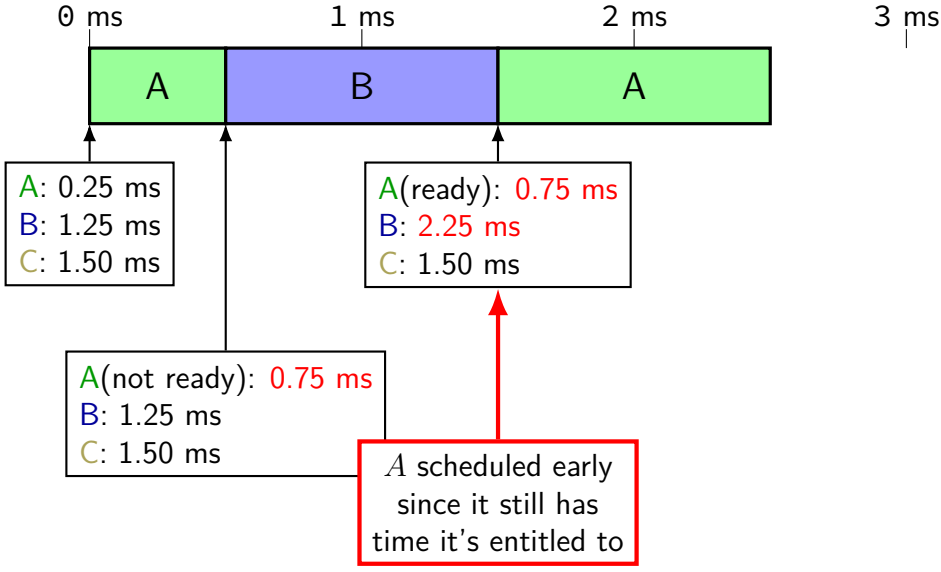
not runnable for a while? get bounded advantage

A doesn't use whole time...

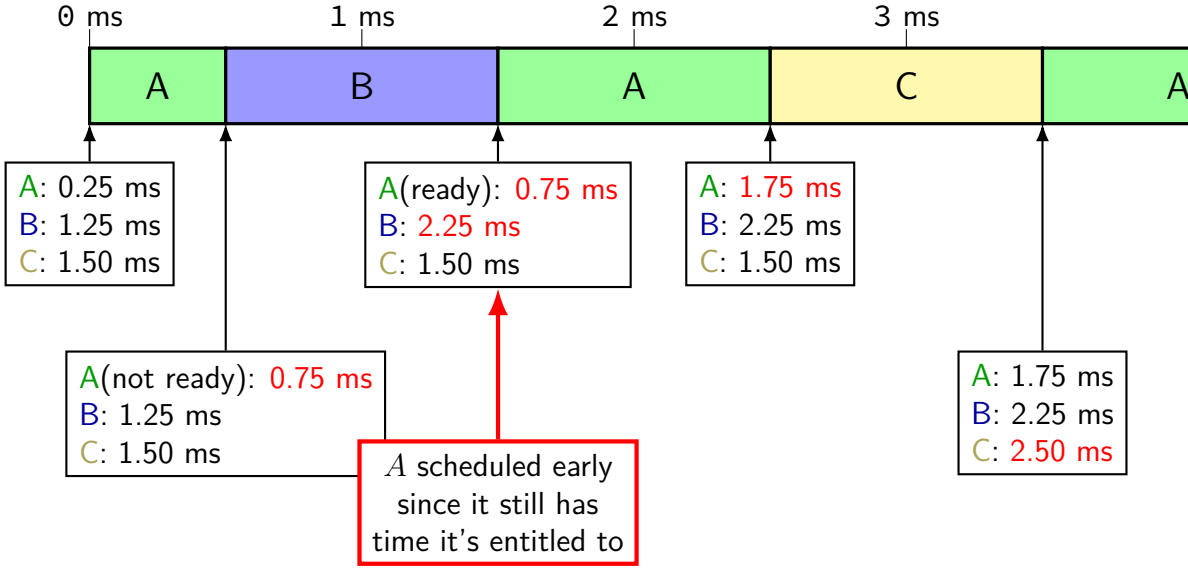
A doesn't use whole time...



A doesn't use whole time...

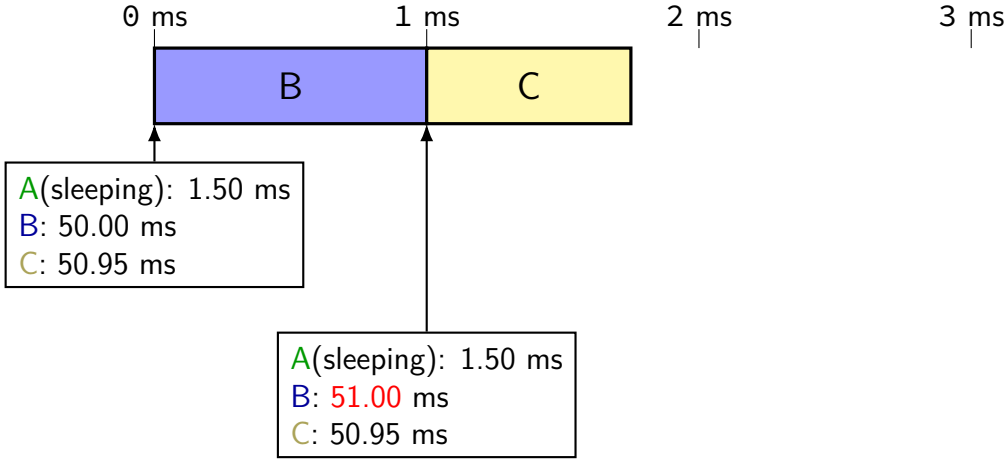


A doesn't use whole time...

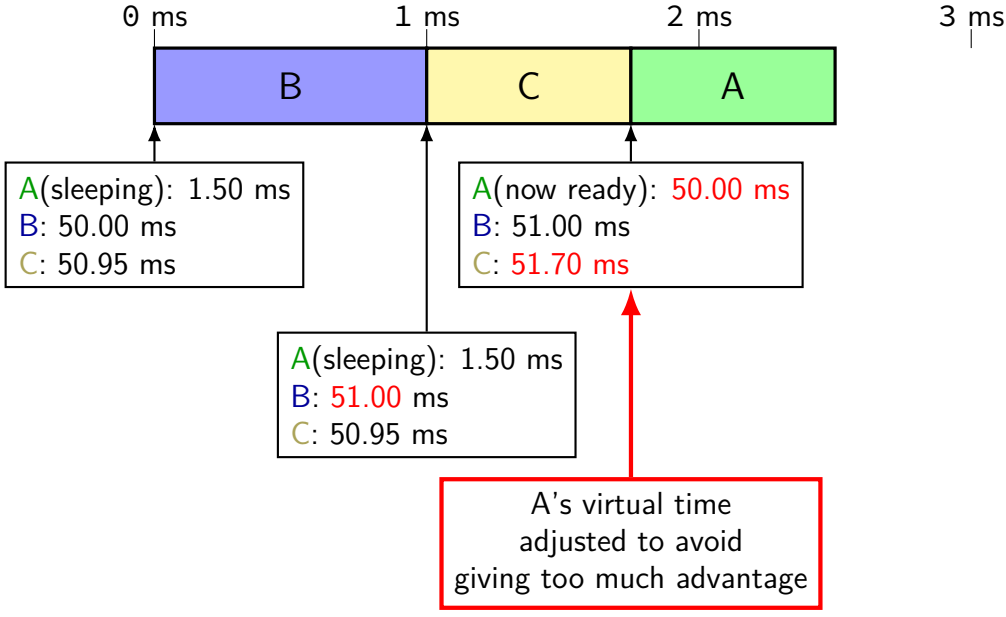


A's long sleep...

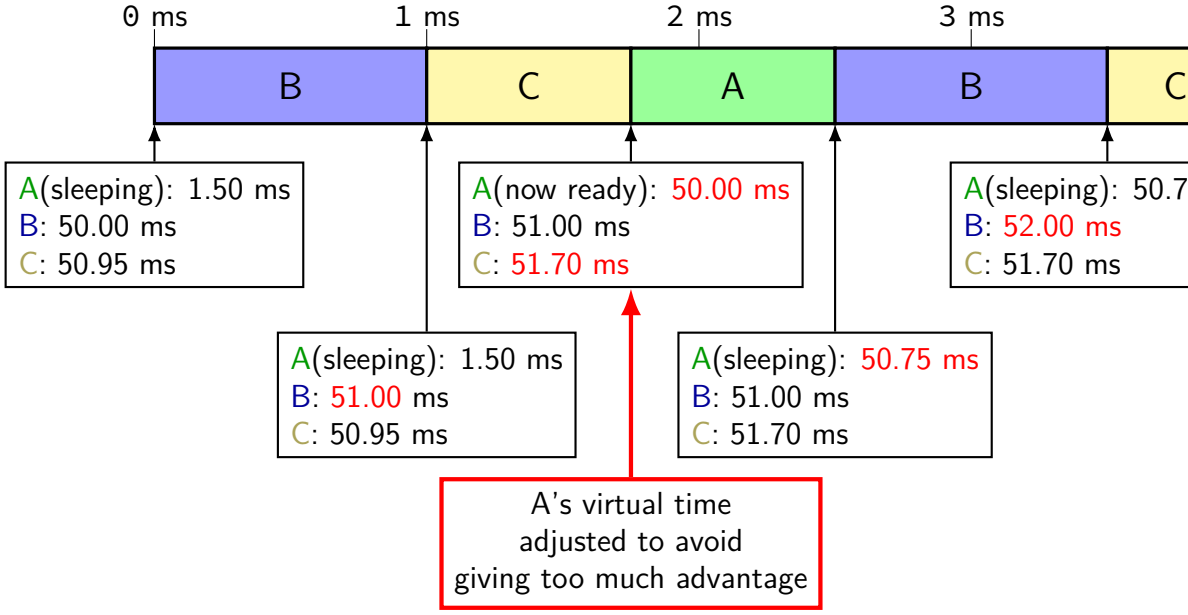
A's long sleep...



A's long sleep...



A's long sleep...



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)

other CFS parts

dealing with multiple CPUs

handling groups of related tasks

special ‘idle’ or ‘batch’ task settings

...

CFS versus others

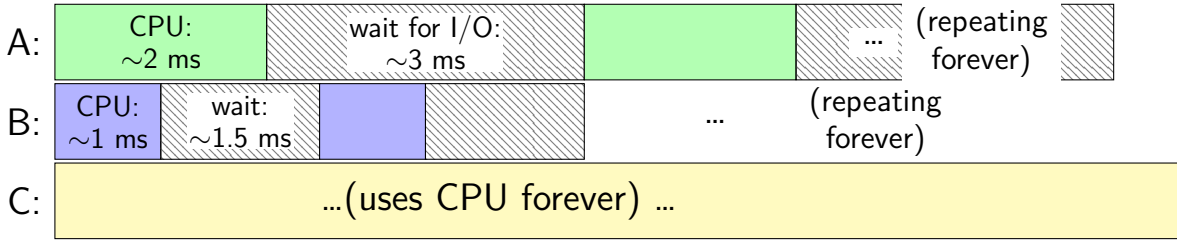
very similar to *stride scheduling*

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)

CFS exercise



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

with CFS, 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

real-time

so far: "best effort" scheduling

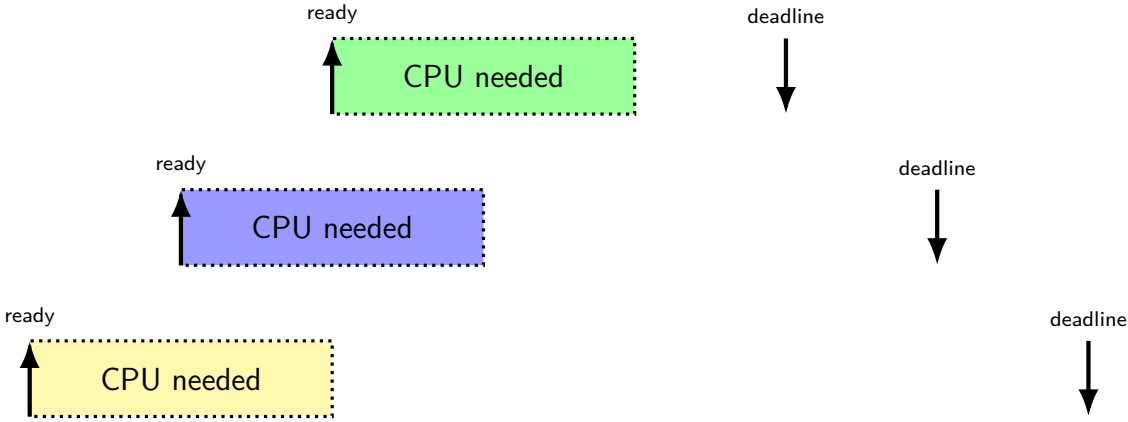
- best possible (by some metrics) given some work

alternate model: need gaurnetees

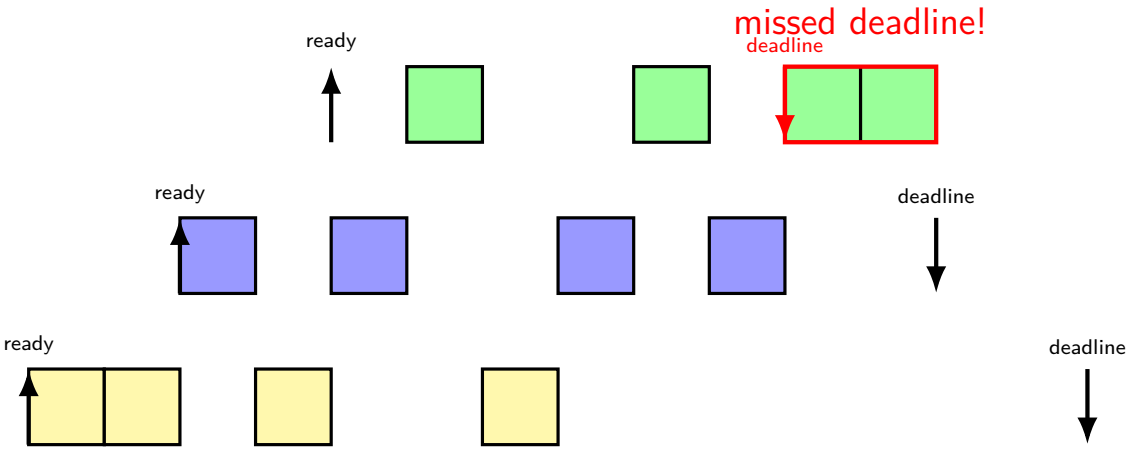
deadlines imposed by real-world

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

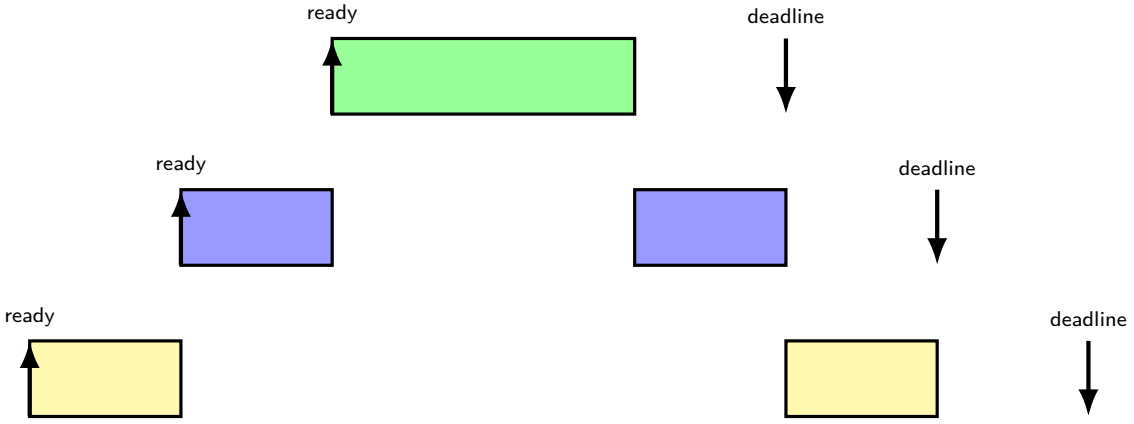
real time example: CPU + deadlines



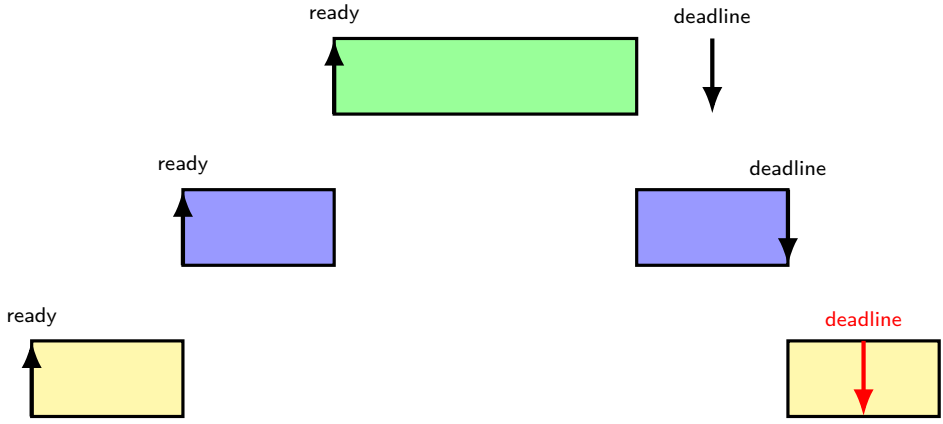
example with RR



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

which scheduler should I choose?

I care about...

CPU throughput: first-come first-serve

average response time: SRTF approximation

I/O throughput: SRTF approximation

fairness — long-term CPU usage: something like Linux CFS

fairness — wait time: something like RR

deadlines — earliest deadline first

favoring certain users: strict priority

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?