scheduling 2 — FCFS, RR, priority, SRTF

last time

xv6 scheduler design

- separate scheduler thread
- disable interrupts while changing thread states

threads versus processes

- thread: part on processor core
- xv6: each process has exactly one thread

CPU bursts

scheduling metrics

- turnaround time: becomes runnable to becomes not-running/runnable
- wait time: turnaround time minus time spent running
- throughput: amount of useful work done per unit time
- fairness
- ...other, subjective/tricky to quantify metrics?

metrics today

big focus on minimizing mean/total turnaround time

- thread becomes ready until thread done being ready

imperfect approximation of interactivity/responsiveness on desktop

question: why imperfect?

two trivial scheduling algorithms

first-come first served (FCFS)

round robin (RR)

scheduling example assumptions

multiple programs become ready at almost the same time

- alternately: became ready while previous program was running

...but in some order that we'll use

- e.g. our ready queue looks like a linked list

two trivial scheduling algorithms

first-come first served (FCFS)

round robin (RR)

first-come, first-served

simplest(?) scheduling algorithm

no preemption — run program until it can't

- suitable in cases where no context switch
- e.g. not enough memory for two active programs

first-come, first-served (FCFS)

(AKA "first in, first out" (FIFO))

thread	CPU time needed
A	24
B	4
C	3

first-come, first-served (FCFS)

(AKA "first in, first out" (FIFO))

thread	CPU time needed
A	24
B	4
C	3

A $\sim$ CPU-bound
B, C $\sim$ I/O bound or interactive

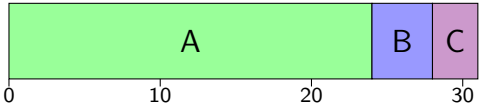
first-come, first-served (FCFS)

(AKA "first in, first out" (FIFO))

thread	CPU time needed
A	24
B	4
C	3

A $\sim$ CPU-bound
B, C $\sim$ I/O bound or interactive

arrival order: **A**, **B**, **C**



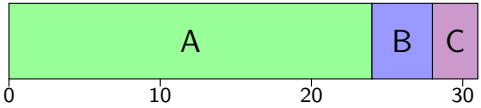
first-come, first-served (FCFS)

(AKA "first in, first out" (FIFO))

thread	CPU time needed
A	24
B	4
C	3

A ∼ CPU-bound
B, C ∼ I/O bound or interactive

arrival order: **A**, **B**, **C**



wait times: (mean=17.3)
0 (**A**), 24 (**B**), 28 (**C**)
turnaround times: (mean=27.7)
24 (**A**), 28 (**B**), 31 (**C**)

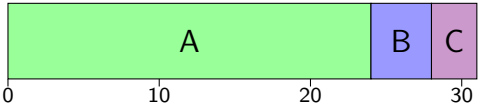
first-come, first-served (FCFS)

(AKA "first in, first out" (FIFO))

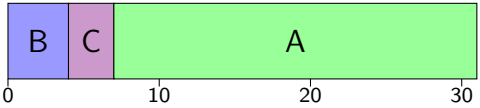
thread	CPU time needed
A	24
B	4
C	3

A ∼ CPU-bound
B, C ∼ I/O bound or interactive

arrival order: **A**, **B**, **C**



arrival order: **B**, **C**, **A**



wait times: (mean=17.3)
0 (**A**), 24 (**B**), 28 (**C**)
turnaround times: (mean=27.7)
24 (**A**), 28 (**B**), 31 (**C**)

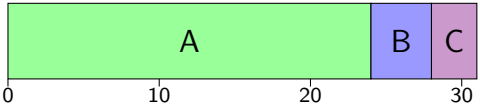
first-come, first-served (FCFS)

(AKA "first in, first out" (FIFO))

thread	CPU time needed
A	24
B	4
C	3

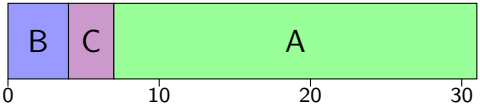
A ∼ CPU-bound
B, C ∼ I/O bound or interactive

arrival order: **A**, **B**, **C**



wait times: (mean=17.3)
0 (**A**), 24 (**B**), 28 (**C**)
turnaround times: (mean=27.7)
24 (**A**), 28 (**B**), 31 (**C**)

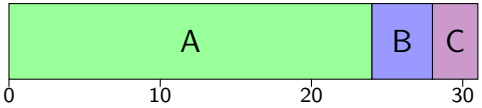
arrival order: **B**, **C**, **A**



wait times: (mean=3.7)
7 (**A**), 0 (**B**), 4 (**C**)
turnaround times: (mean=14)
31 (**A**), 4 (**B**), 7 (**C**)

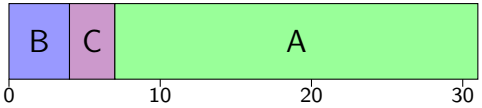
FCFS orders

arrival order: **A**, **B**, **C**



wait times: (mean=17.3)
0 (**A**), 24 (**B**), 28 (**C**)
turnaround times: (mean=27.7)
24 (**A**), 28 (**B**), 31 (**C**)

arrival order: **B**, **C**, **A**



wait times: (mean=3.7)
7 (**A**), 0 (**B**), 4 (**C**)
turnaround times: (mean=14)
31 (**A**), 3 (**B**), 7 (**C**)

"convoy effect"

two trivial scheduling algorithms

first-come first served (FCFS)

round robin (RR)

round-robin

simplest(?) preemptive scheduling algorithm

run program until either

- it can't run anymore, or
- it runs for too long (exceeds "time quantum")

requires good way of interrupting programs

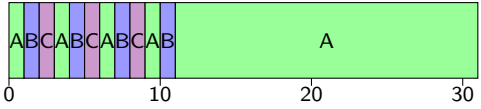
- like xv6's timer interrupt

requires good way of stopping programs whenever

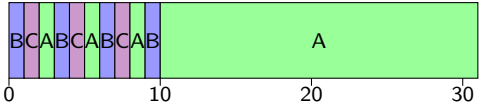
- like xv6's context switches

round robin (RR) (varying order)

time quantum $= 1$,
order **A**, **B**, **C**

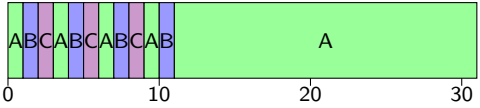


time quantum $= 1$,
order **B**, **C**, **A**



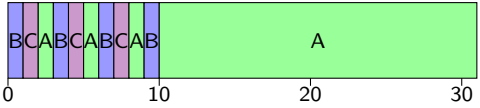
round robin (RR) (varying order)

time quantum = 1,
order **A**, **B**, **C**



waiting times: (mean=6.7)
7 (**A**), 7 (**B**), 6 (**C**)
turnaround times: (mean=17)
31 (**A**), 11 (**B**), 9 (**C**)

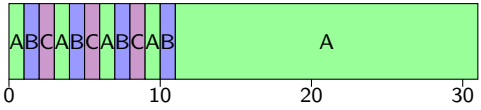
time quantum = 1,
order **B**, **C**, **A**



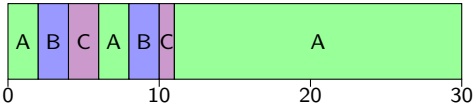
waiting times: (mean=6)
7 (**A**), 6 (**B**), 5 (**C**)
turnaround times: (mean=16.3)
31 (**A**), 10 (**B**), 8 (**C**)

round robin (RR) (varying time quantum)

time quantum $= 1$,
order **A**, **B**, **C**

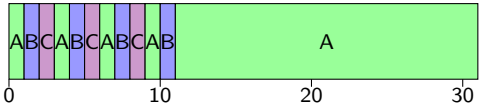


time quantum $= 2$,
order **A**, **B**, **C**



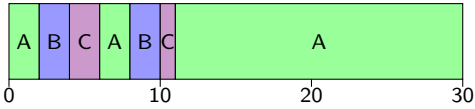
round robin (RR) (varying time quantum)

time quantum = 1,
order **A**, **B**, **C**



waiting times: (mean=6.7)
7 (**A**), 7 (**B**), 6 (**C**)
turnaround times: (mean=17)
31 (**A**), 11 (**B**), 9 (**C**)

time quantum = 2,
order **A**, **B**, **C**



waiting times: (mean=7)
7 (**A**), 6 (**B**), 8 (**C**)
turnaround times: (mean=17.3)
31 (**A**), 10 (**B**), 11 (**C**)

round robin idea

choose fixed time quantum Q

- unanswered question: what to choose

switch to next process in ready queue after time quantum expires

this policy is what xv6 scheduler does

- scheduler runs from timer interrupt (or if process not runnable)
- finds next runnable process in process table

round robin and time quantums

many context switches
(lower throughput)

few context switches
(higher throughput)

order doesn't matter
(more fair)

first program favored
(less fair)

smaller quantum: more fair, worse throughput

round robin and time quantums

many context switches
(lower throughput)

few context switches
(higher throughput)

order doesn't matter
(more fair)

first program favored
(less fair)

smaller quantum: more fair, worse throughput

FCFS = RR with infinite quantum

more fair: at most $(N-1)Q$ time until scheduled if N total processes

aside: context switch overhead

typical context switch: $\sim$ 0.01 ms to 0.1 ms

- but tricky: lot of indirect cost (cache misses)
- (above numbers try to include likely indirect costs)

choose time quantum to manage this overhead

current Linux default: between $\sim$0.75 ms and $\sim$6 ms

- varied based on number of active programs
- Linux's scheduler is more complicated than RR

historically common: 1 ms to 100 ms

- 1% to 0.1% ovherhead?

round robin and time quantums

many context switches
(lower throughput)

few context switches
(higher throughput)

order doesn't matter
(more fair)

first program favored
(less fair)

smaller quantum: more fair, worse throughput

FCFS = RR with infinite quantum

more fair: at most $(N-1)Q$ time until scheduled if N total processes

but what about turnaround/waiting time?

exercise: round robin quantum

if there were no context switch overhead, *decreasing* the time quantum (for round robin) would cause mean turnaround time to _____.

A. always decrease or stay the same

B. always increase of stay the same

C. increase or decrease or stay the same

D. something else?

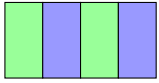
increase mean turnaround time

A: 1 unit CPU burst
B: 1 unit

Q = 1

mean turnaround time =
$(1 + 2) \div 2 = 1.5$

Q = 1/2



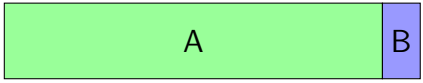
mean turnaround time =
$(1.5 + 2) \div 2 = 1.75$

decrease mean turnaround time

A: 10 unit CPU burst

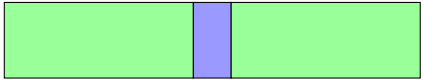
B: 1 unit

Q = 10



mean turnaround time =
$(10 + 11) \div 2 = 10.5$

Q = 5



mean turnaround time =
$(6 + 11) \div 2 = 8.5$

stay the same

A: 1 unit CPU burst
B: 1 unit

$Q = 10$ 

$Q = 1$ 

FCFS and order

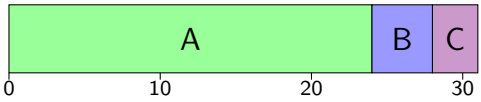
earlier we saw that with FCFS, arrival order mattered

big changes in turnaround/waiting time

let's use that insight to see how to optimize mean/total turnaround times

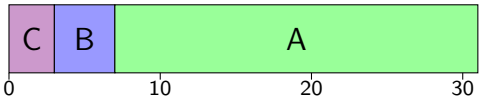
FCFS orders

arrival order: **A**, **B**, **C**



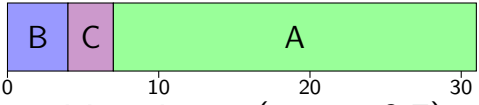
waiting times: (mean=17.3)
0 (**A**), 24 (**B**), 28 (**C**)
turnaround times: (mean=27.7)
24 (**A**), 28 (**B**), 31 (**C**)

arrival order: **C**, **B**, **A**



waiting times: (mean=3.3)
7 (**A**), 3 (**B**), 0 (**C**)
turnaround times: (mean=13.7)
31 (**A**), 7 (**B**), 3 (**C**)

arrival order: **B**, **C**, **A**



waiting times: (mean=3.7)
7 (**A**), 0 (**B**), 4 (**C**)
turnaround times: (mean=14)
31 (**A**), 4 (**B**), 7 (**C**)

order and turnaround time

best total/mean turnaround time = run shortest CPU burst first

worst total/mean turnaround time = run longest CPU burst first

intuition (1): "race to go to sleep"

intuition (2): minimize time with two threads waiting

order and turnaround time

best total/mean turnaround time = run shortest CPU burst first

worst total/mean turnaround time = run longest CPU burst first

intuition (1): "race to go to sleep"

intuition (2): minimize time with two threads waiting

later: we'll use this result to make a scheduler that minimizes mean turnaround time

diversion: some users are more equal

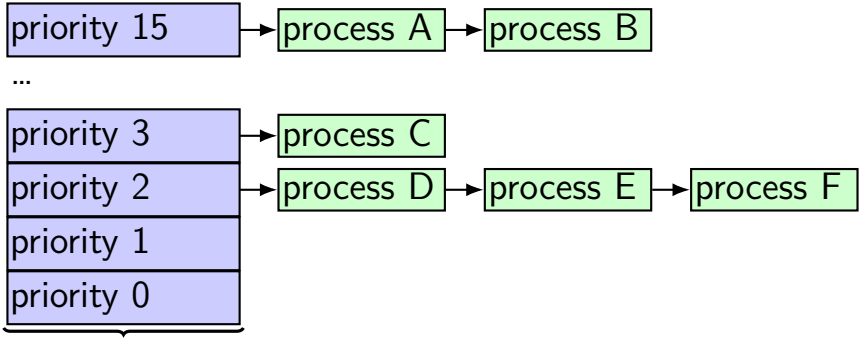
shells more important than big computation?

- i.e. programs with short CPU bursts

faculty more important than students?

scheduling algorithm: schedule shells/faculty programs first

priority scheduling



choose process from ready queue for highest priority

within each priority, use some other scheduling (e.g. round-robin)

could have each process have unique priority

priority scheduling and preemption

priority scheduling can be preemptive

i.e. higher priority program comes along — stop whatever else was running

exercise: priority scheduling (1)

Suppose there are two processes:

thread A

- highest priority
- repeat forever: 1 unit of I/O, then 10 units of CPU, ...

thread Z

- lowest priority
- 4000 units of CPU (and no I/O)

How long will it take thread Z complete?

exercise: priority scheduling (2)

Suppose there are three processes:

thread A
- highest priority
- repeat forever: 1 unit of I/O, then 10 units of CPU, ...

thread B
- second-highest priority
- repeat forever: 1 unit of I/O, then 10 units of CPU, ...

thread Z
- lowest priority
- 4000 units of CPU (and no I/O)

How long will it take thread Z complete?

starvation

programs can get “starved” of resources

never get those resources because of higher priority

big reason to have a ‘fairness’ metric

something almost all definitions of fairness agree on

minimizing turnaround time

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

→ 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

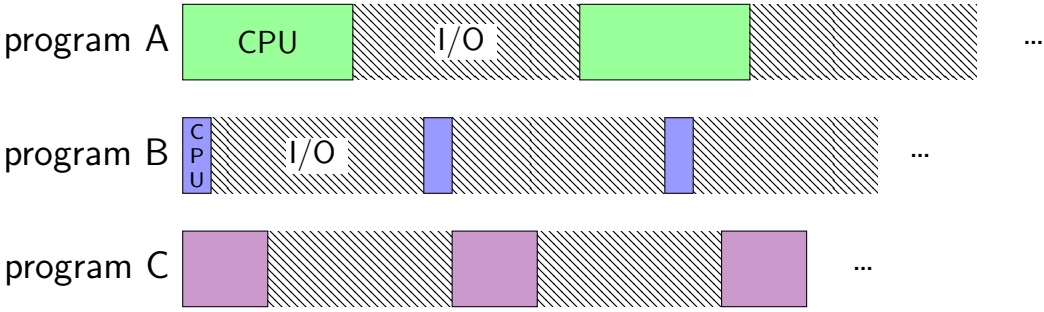
so we want to run the shortest CPU burst first

how do I tell which thread that is?

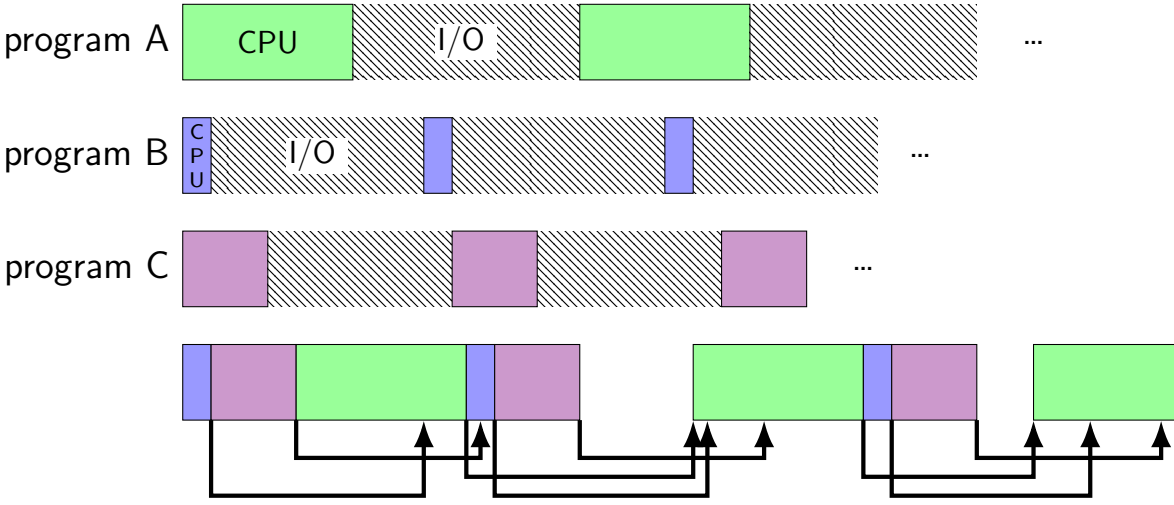
we'll deal with this problem later

...kinda

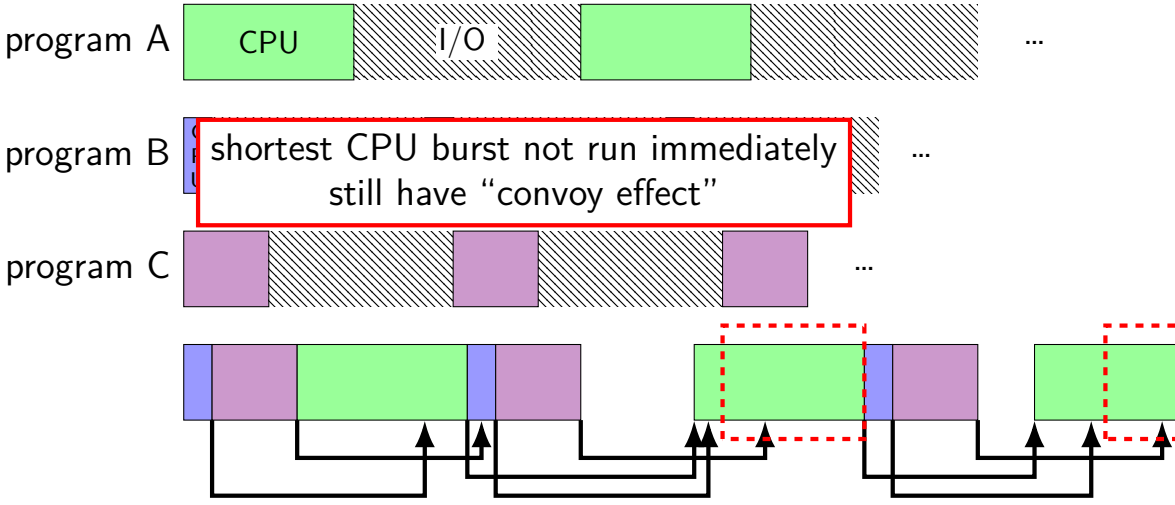
alternating I/O and CPU: SJF



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?

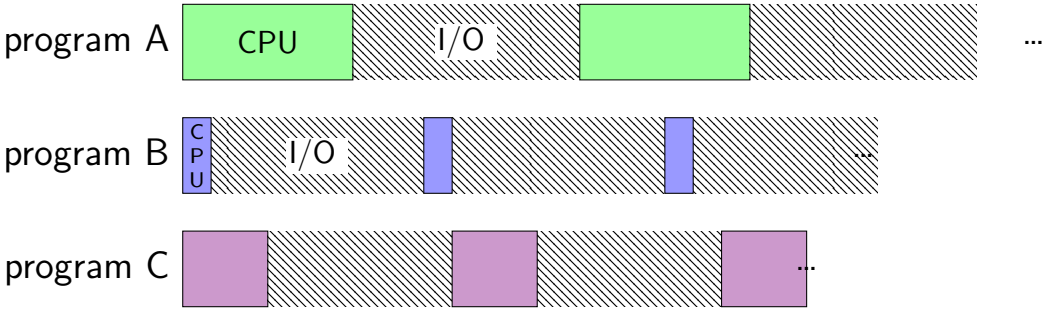
- recall: priority = job length
- long job waits for medium job
- ...for longer than it would take to finish
- worse than letting long job finish

solution: priority = remaining time

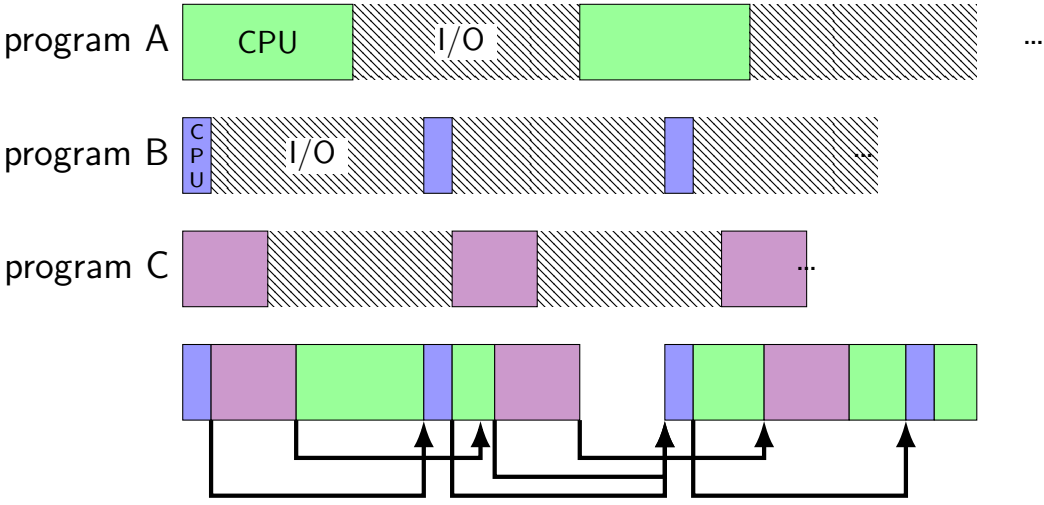
called shortest *remaining time* first (SRTF)

- prioritize by what's left, not the total

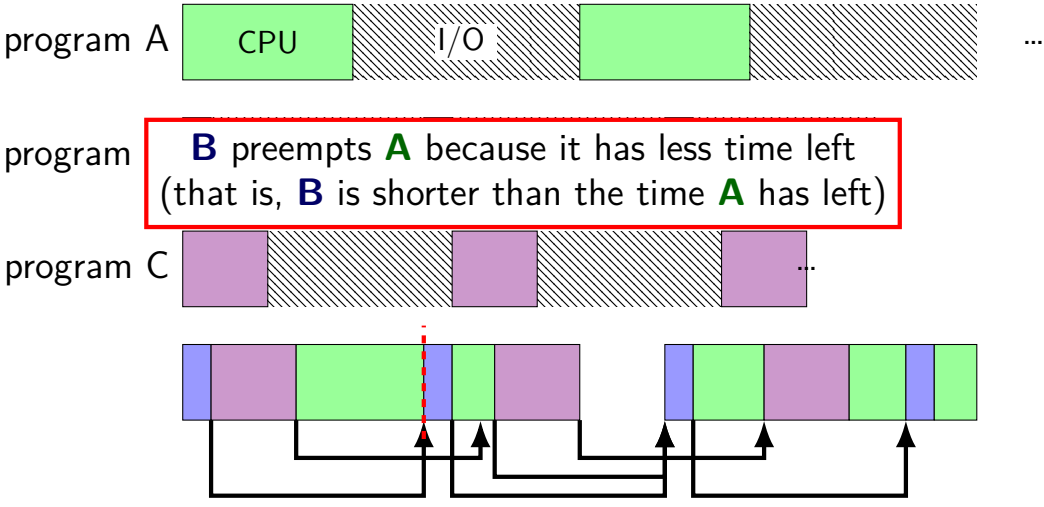
alternating I/O and CPU: SRTF



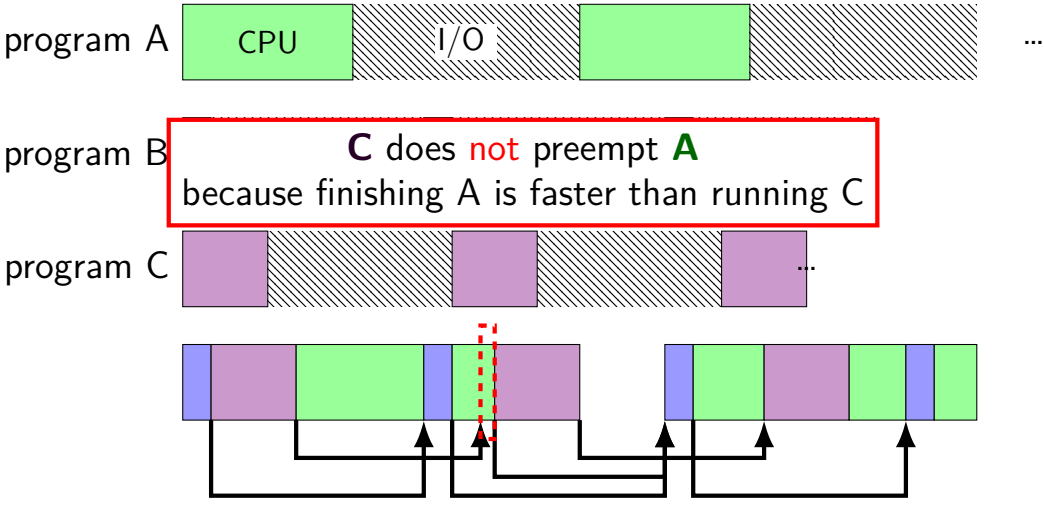
alternating I/O and CPU: SRTF



alternating I/O and CPU: SRTF



alternating I/O and CPU: SRTF



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

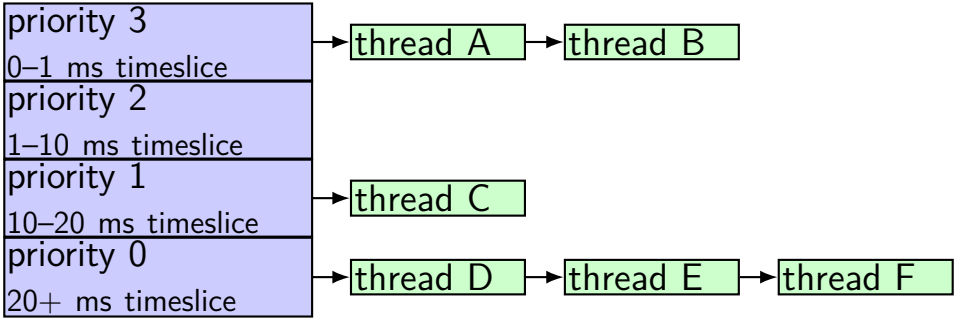
classic strategy based on priority scheduling

combines update time estimates and running shorter times first

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

small(ish) number of time estimate "buckets"

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

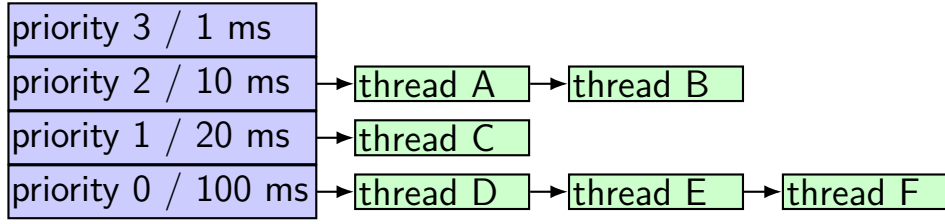
dynamically adjust priorities based on observed CPU burst times

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

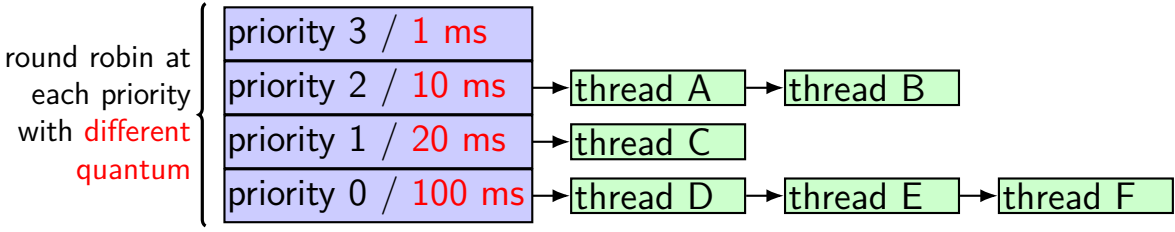
taking advantage of history

idea: priority = CPU burst length



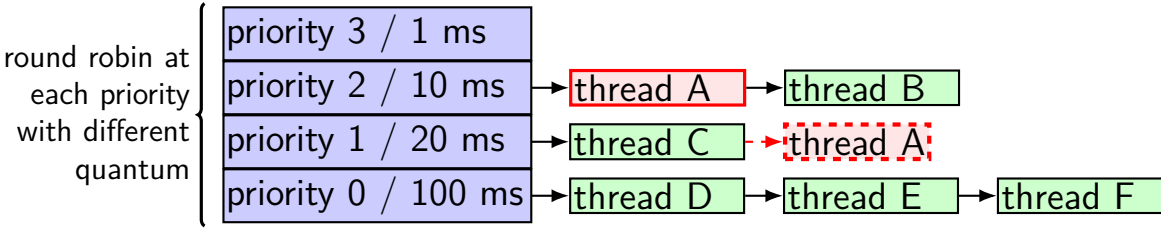
taking advantage of history

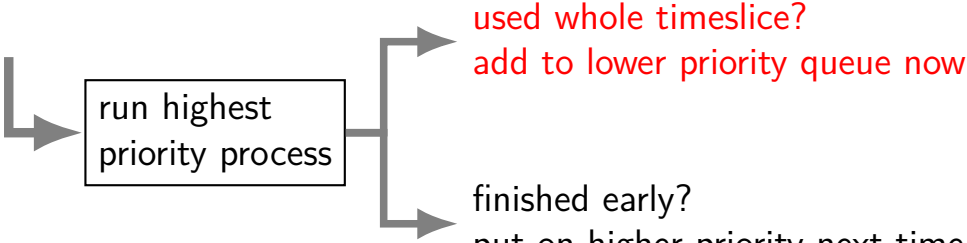
idea: priority = CPU burst length



taking advantage of history

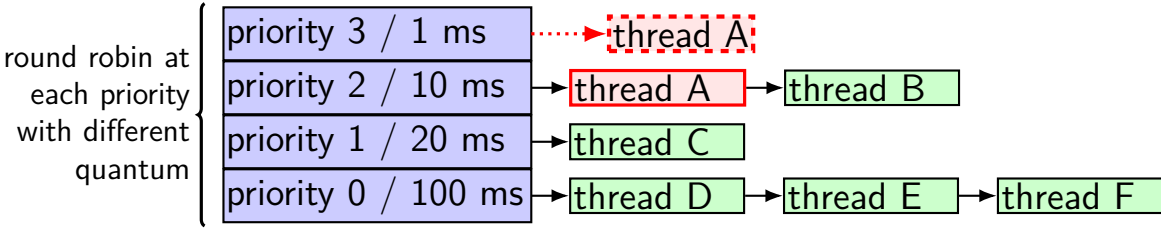
idea: priority = CPU burst length

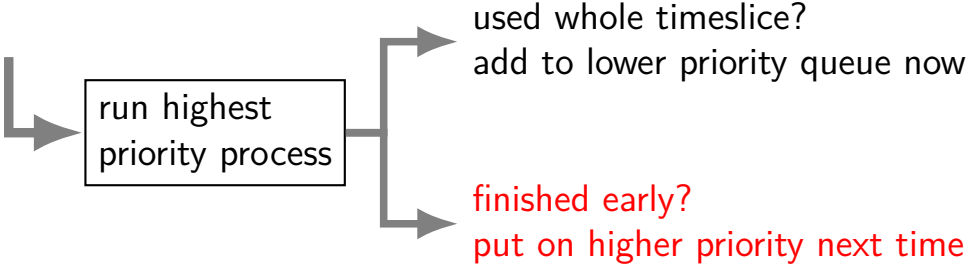




taking advantage of history

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: 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.