POSIX API 3 / Scheduling 1

last time

pipes (finish)

process states in xv6

the scheduler thread, switching to and from

scheduling queues

I/O and CPU bursts

note on VM

I made mistake in producing the VM...

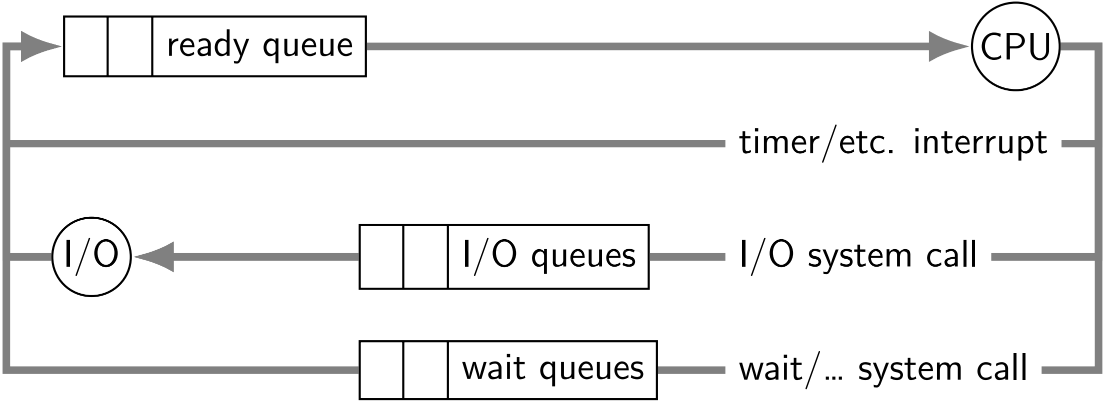
though I started from a clean VM but...

some excess files on VM from a last semester submission testing

please delete them (fat* and life directories)

questions/concerns? email/talk to me privately

recall: scheduling queues



CPU and I/O bursts

...

compute
start read
(from file/keyboard/...)

wait for I/O

compute on read data
start read

wait for I/O

compute on read data
start write

wait for I/O

...

program alternates between computing and waiting for I/O

examples:

shell: wait for keypresses

drawing program: wait for mouse presses/etc.

web browser: wait for remote web server

...

CPU bursts and interactivity (one c. 1966 shared system)

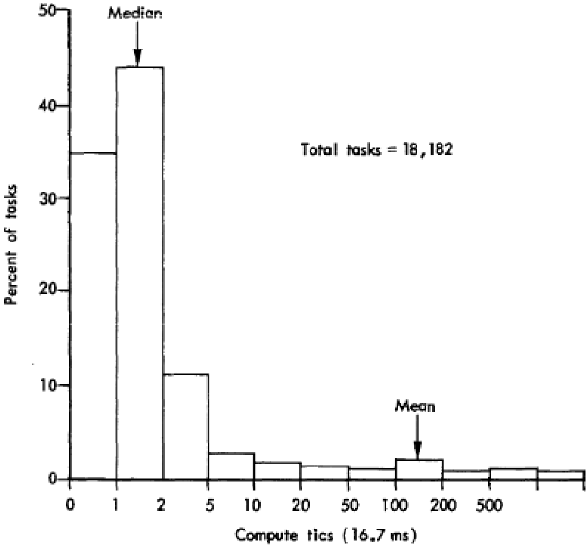
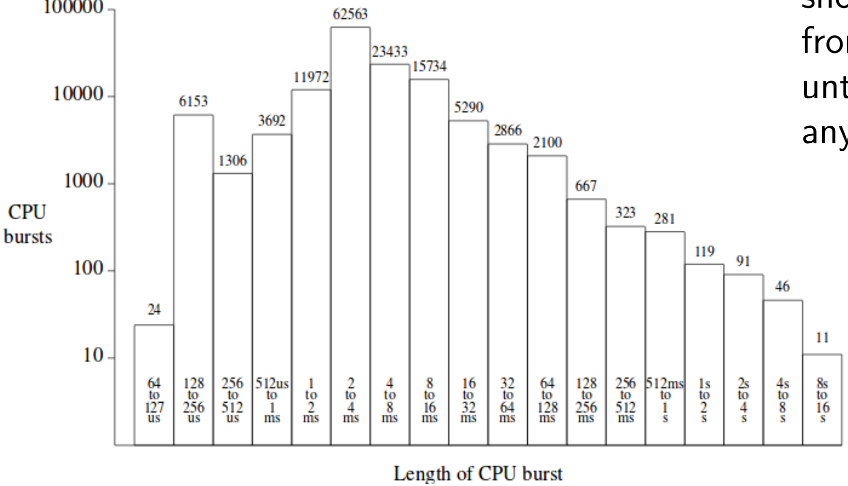


Figure 11—Compute time per task

shows compute time
from command entered
until next command prompt

from G. E. Bryan, "JOSS: 20,000 hours at a console—a statistical approach" in Proc. AFIPS 1967 FJCC

CPU bursts and interactivity (one c. 1990 desktop)



shows CPU time from RUNNING until not RUNNABLE anymore

from Curran and Stumm, "A Comparison of basic CPU Scheduling Algoirithms for Multiprocessor Unix"

CPU bursts

observation: applications alternate between I/O and CPU

- especially interactive applications
- but also, e.g., reading and writing from disk

typically short "CPU bursts" (milliseconds) followed by short "IO bursts" (milliseconds)

scheduling CPU bursts

our typical view: ready queue, bunch of CPU bursts to run

to start: just look at running what's currently in ready queue best

- same problem as 'run bunch of programs to completion'?

later: account for I/O after CPU burst

an historical note

historically applications were less likely to keep all data in memory

historically computers shared between more users

meant *more* applications alternating I/O and CPU

context many scheduling policies were developed in

scheduling metrics

response time (Anderson-Dahlin) AKA turnaround time (Arpaci-Dusseau) (want *low*)

- (what Arpaci-Dusseau calls response time is slightly different — more later)
- what user sees: from *keypress* to *character on screen*
- (submission until job finsihed)

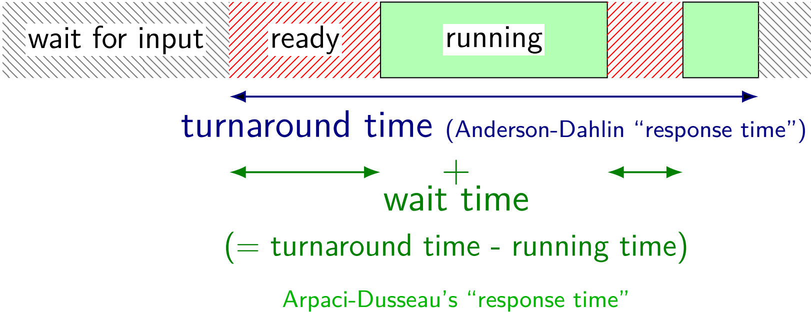
throughput (want *high*)

- total work per second
- problem: overhead (e.g. from context switching)

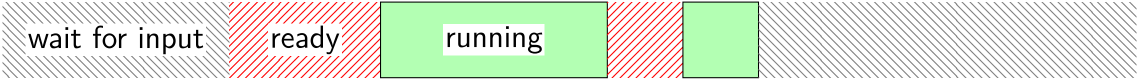
fairness

- many definitions
- all conflict with best average throughput/turnaround time

turnaround and wait time



turnaround and wait time



turnaround time (Anderson-Dahlin “response time”)

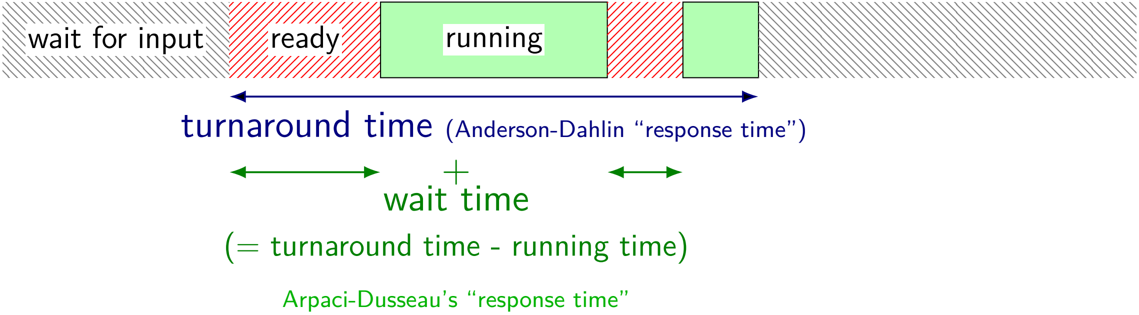
+

wait time

(= turnaround time - running time)

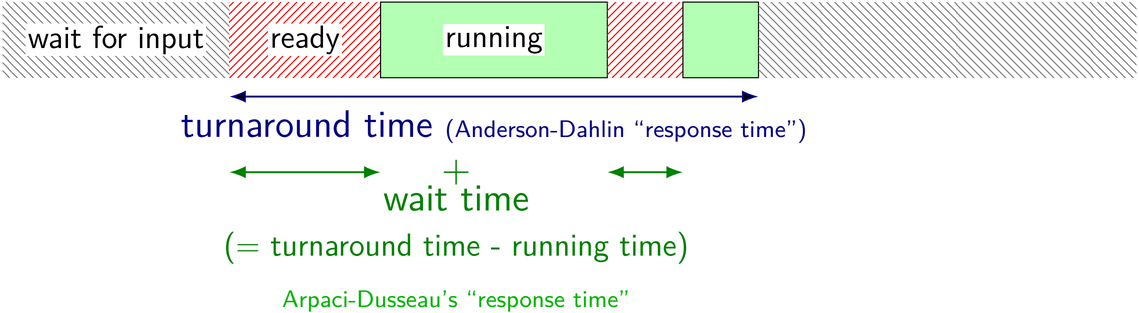
Arpaci-Dusseau’s “response time”

turnaround and wait time



common measure: *mean* turnaround time or *total* turnaround time

turnaround and wait time



common measure: *mean* turnaround time or *total* turnaround time

same as optimizing total/mean waiting time

turnaround time and I/O

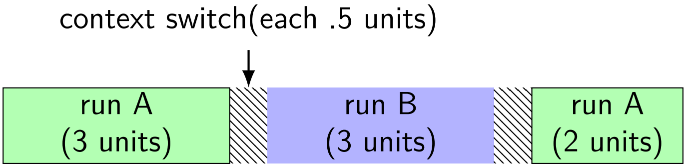
scheduling CPU bursts?

- turnaround time ≈ time to start next I/O
- important for fully utilizing I/O devices
- closed loop: faster turnaround time → program requests CPU sooner

scheduling batch program on cluster?

- turnaround time ≈ how long does user wait
- once program done with CPU, it's probably done

throughput



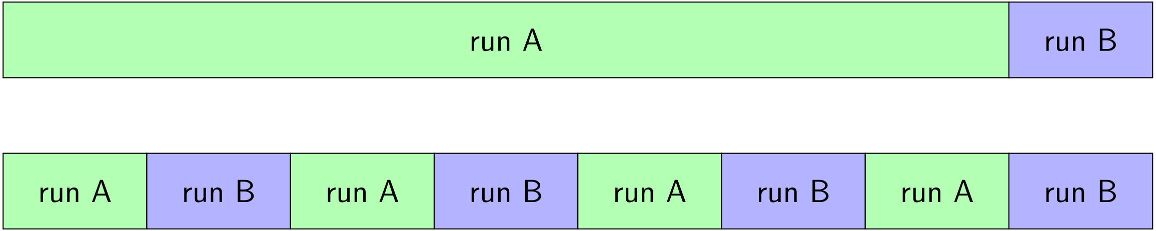
throughput: useful work done per unit time

non-context switch CPU utilization $= \frac{3+3+2}{3+.5+3+.5+2} = 88\%$

also other considerations:

- time lost due to cold caches
- time lost not starting I/O early as possible
- ...

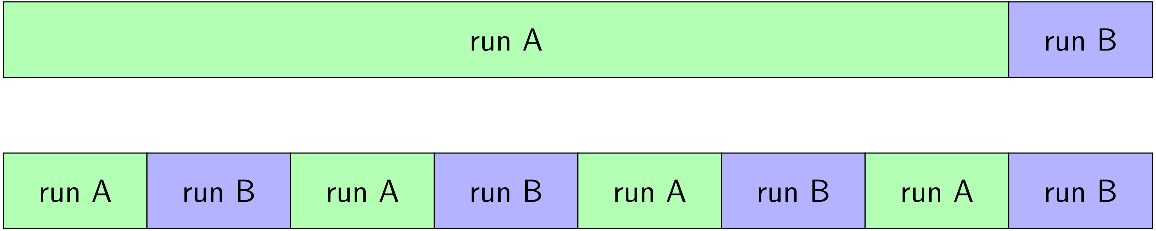
fairness



assumption: one program per user

two timelines above; which is fairer?

fairness



assumption: one program per user

two timelines above; which is fairer?

easy to answer — but formal definition?

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))

process	CPU time needed
A	24
B	4
C	3

first-come, first-served (FCFS)

(AKA “first in, first out” (FIFO))

process	CPU time needed
A	24
B	4
C	3

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

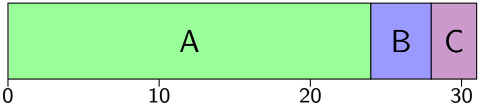
first-come, first-served (FCFS)

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

process	CPU time needed
A	24
B	4
C	3

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

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



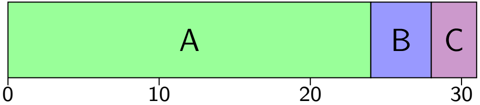
first-come, first-served (FCFS)

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

process	CPU time needed
A	24
B	4
C	3

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

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**)

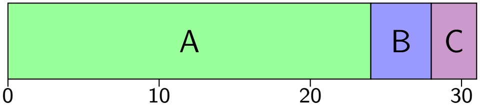
first-come, first-served (FCFS)

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

process	CPU time needed
A	24
B	4
C	3

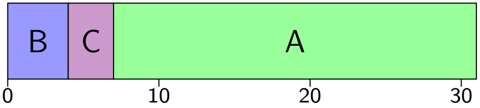
A ∼ CPU-bound
B, C ∼ I/O bound or interactive

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: **B**, **C**, **A**



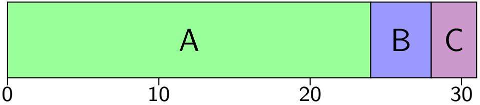
first-come, first-served (FCFS)

(AKA “first in, first out” (FIFO))

process	CPU time needed
A	24
B	4
C	3

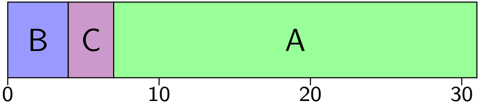
A $\sim$ CPU-bound
B, C $\sim$ I/O bound or interactive

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



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

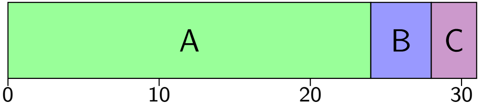
arrival order: **B**, **C**, **A**



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

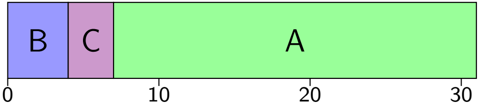
FCFS orders

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



waiting 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**



waiting 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

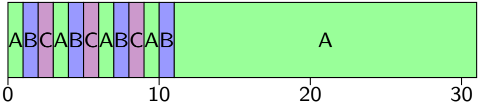
- like xv6's timer interrupt

requires good way of stopping programs whenever

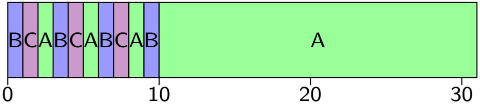
- like xv6's context switches

round robin (RR) (varying order)

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

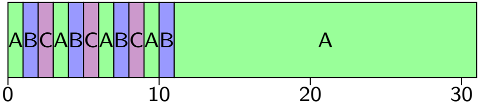


time quantum $= 1$,
order **B**, **C**, **A**



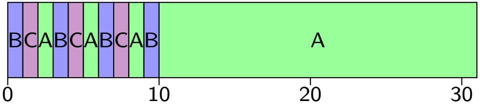
round robin (RR) (varying order)

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**)

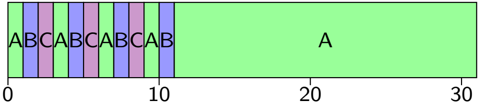
time quantum = 1,
order **B**, **C**, **A**



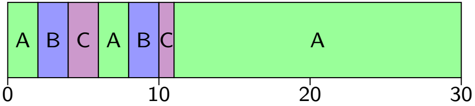
waiting times: (mean=6)
7 (**A**), 6 (**B**), 5 (**C**)
turnaround times: (mean=16.3)
31 (**A**), 10 (**B**), 8 (**C**)

round robin (RR) (varying time quantum)

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

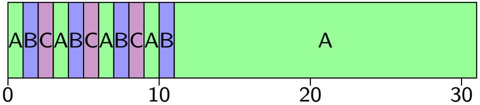


time quantum = 2,
order **A**, **B**, **C**



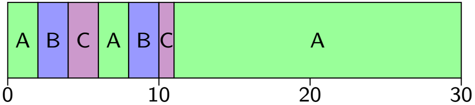
round robin (RR) (varying time quantum)

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

RR with short quantum	⟵⟶	FCFS
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)

RR with short quantum ⟷ FCFS

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

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)

RR with short quantum ⟷ FCFS

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 average turnaround time to ______________.

A. always decrease or stay the same

B. always increase of stay the same

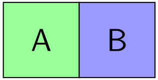
C. increase or decrease or stay the same

D. something else?

increase turnaround time

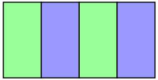
A: 1 unit CPU burst
B: 1 unit

Q = 1



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

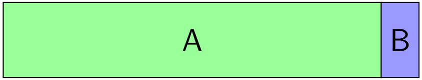
Q = 1/2



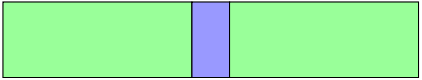
mean turnaround time =
$(1.5 + 2) \div 2 = 1.75$

decrease turnaround time

A: 10 unit CPU burst
B: 1 unit

Q = 10 

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

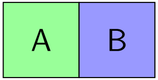
Q = 5 

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

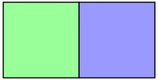
stay the same

A: 1 unit CPU burst
B: 1 unit

Q = 10



Q = 1



FCFS and order

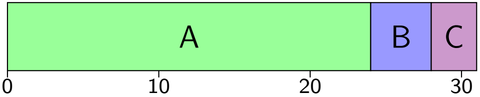
earlier we saw that with FCFS, arrival order mattered

big changes in turnaround/waiting time

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

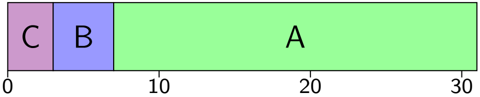
FCFS orders

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



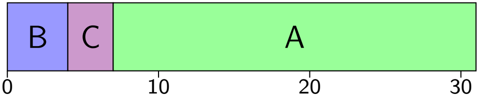
waiting times: (mean=17.3)
0 (**A**), 24 (**B**), 28 (**C**)
turnaround times: (mean=27.7)
24 (**A**), 28 (**B**), 31 (**C**)

arrival order: **B**, **C**, **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 turnaround time = run shortest CPU burst first

worst turnaround time = run longest CPU burst first

intuition: “race to go to sleep”

diversion: some users are more equal

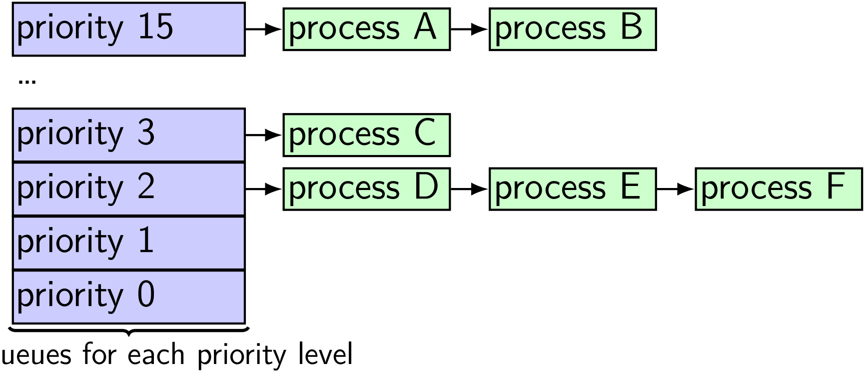
shells more important than big computation?

- 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:

process A

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

process Z

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

How long will it take process Z complete?

exercise: priority scheduling (2)

Suppose there are three processes:

process A

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

process B

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

process Z

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

How long will it take process Z complete?

starvation

programs can get “starved” of resources

never get those resources because of higher priority

big reason to have a ‘fairness’ metric

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

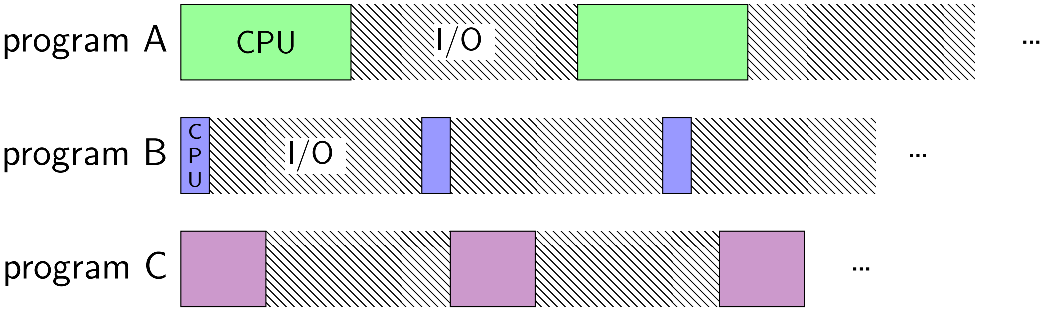
so we want to run the shortest CPU burst first

how do I tell which thread that is?

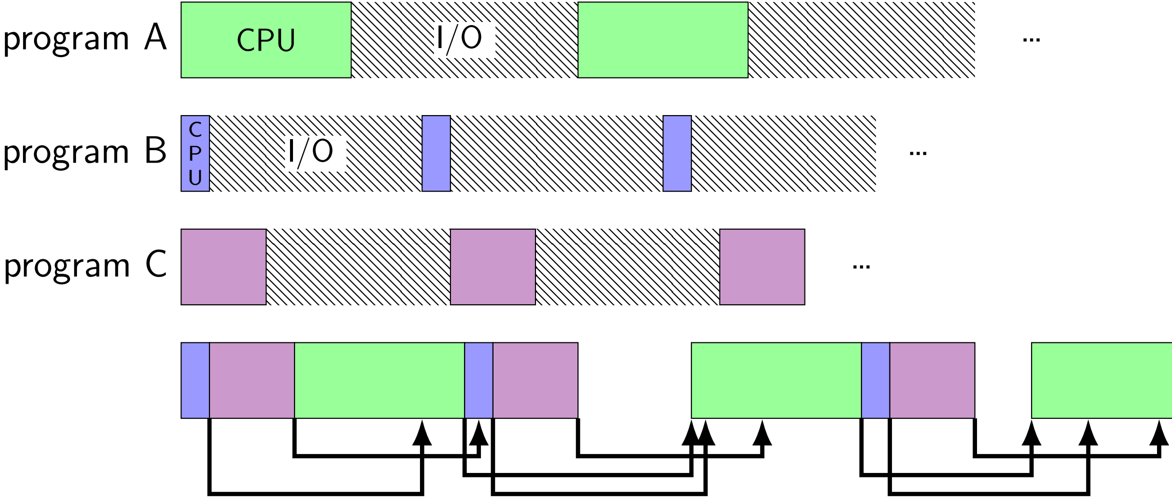
we'll deal with this problem later

...kinda

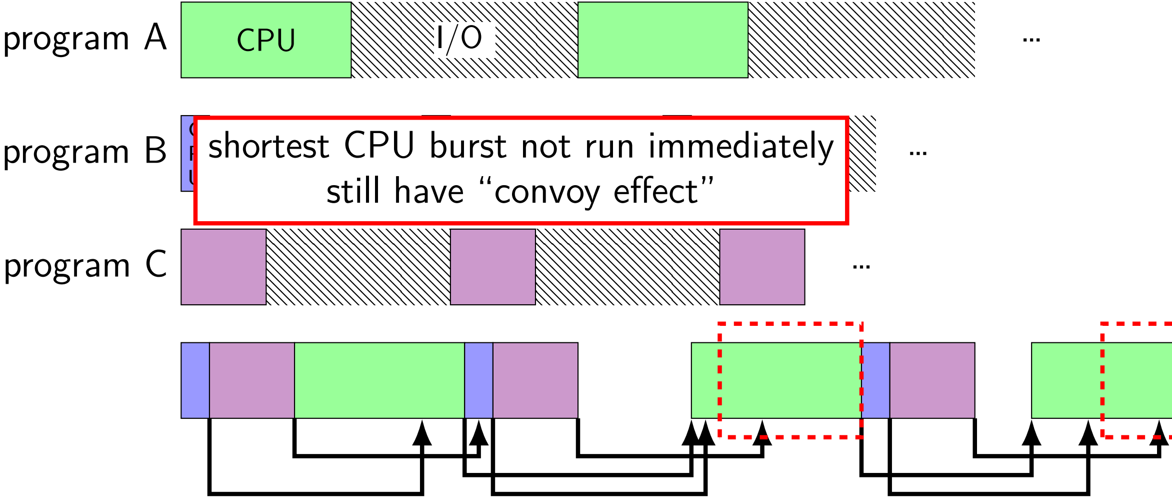
alternating I/O and CPU: SJF



alternating I/O and CPU: SJF



alternating I/O and CPU: SJF



shortest CPU burst not run immediately
still have "convoy effect"

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

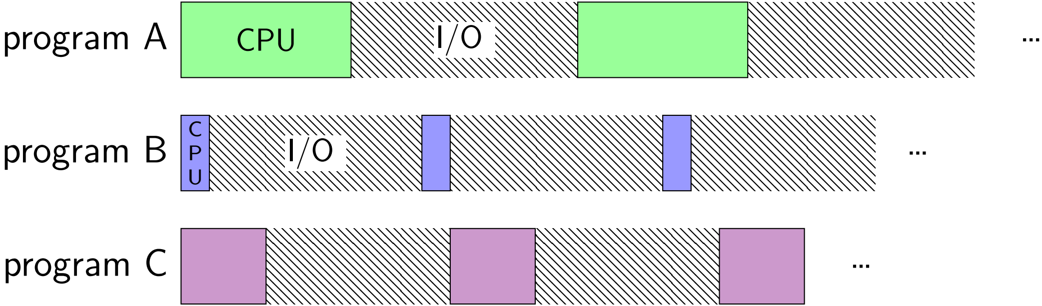
so we want to run the shortest CPU burst first

how do I tell which thread that is?

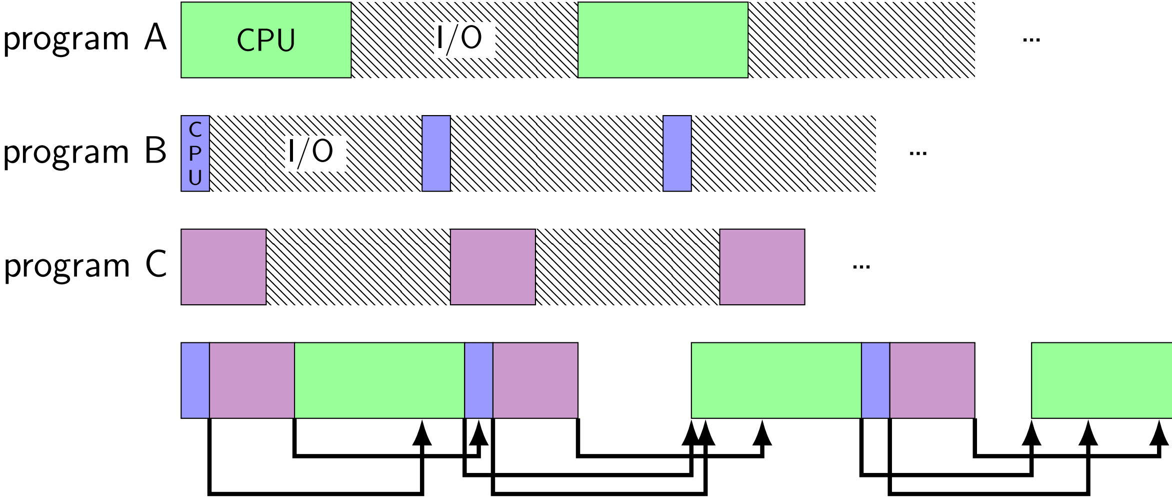
we'll deal with this problem later

...kinda

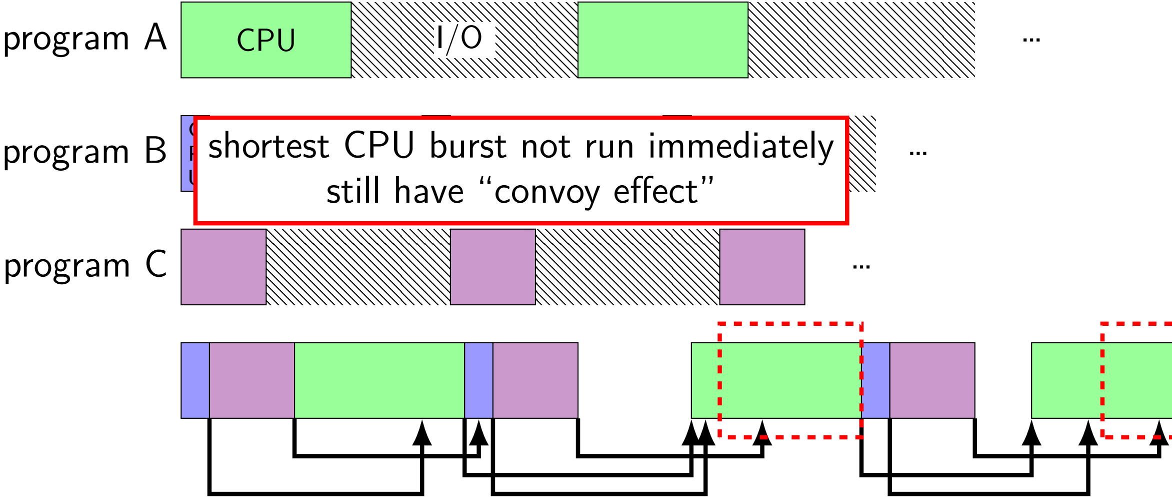
alternating I/O and CPU: SJF



alternating I/O and CPU: SJF



alternating I/O and CPU: SJF



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?

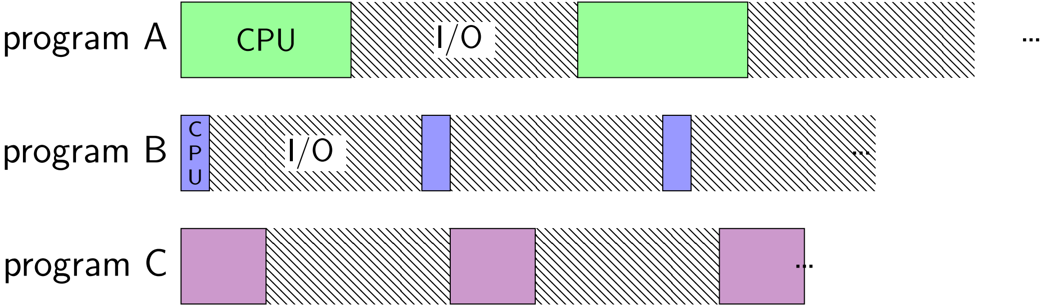
- recall: priority = job length
- long job waits for medium job
- ...for longer than it would take to finish
- worse than letting long job finish

solution: priority = remaining time

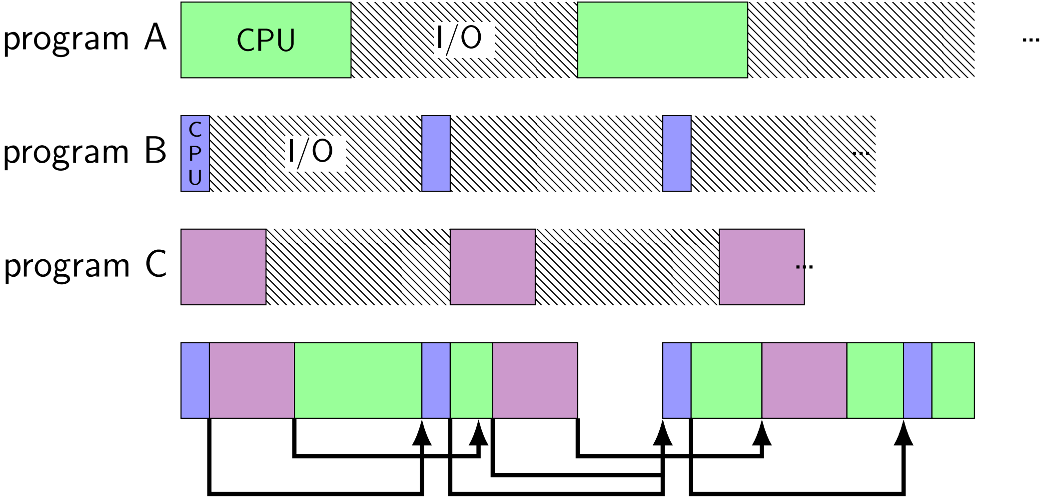
called shortest *remaining time* first (SRTF)

- prioritize by what's left, not the total

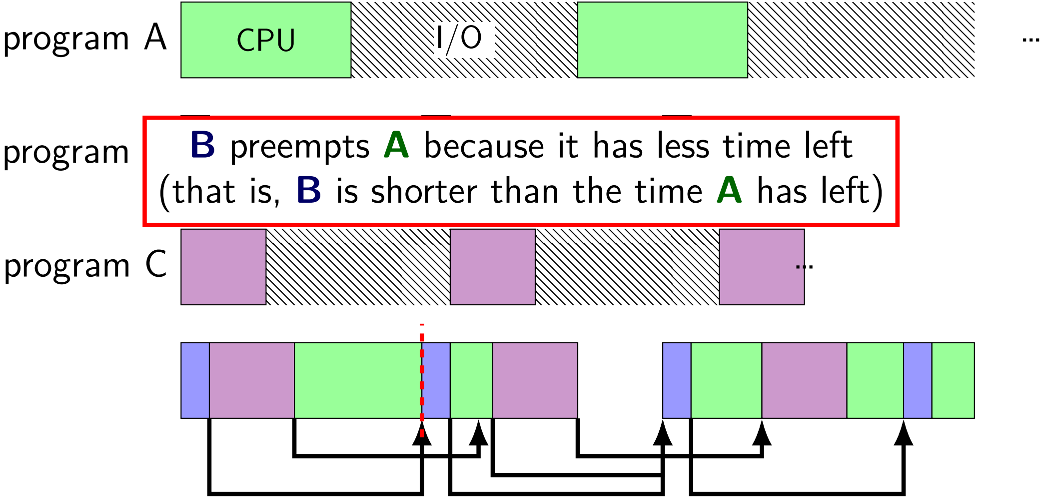
alternating I/O and CPU: SRTF



alternating I/O and CPU: SRTF

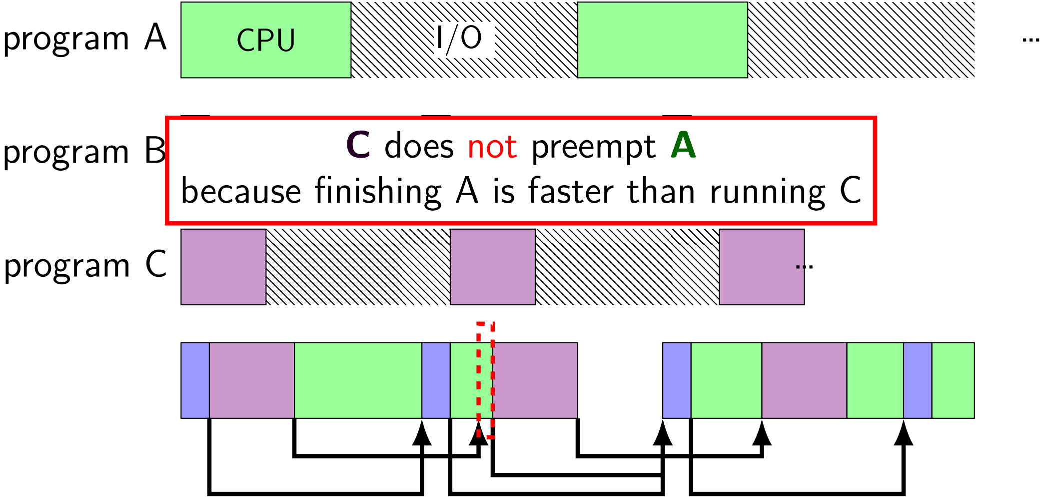


alternating I/O and CPU: SRTF



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 response time)

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

SRTF minimizes response 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 lengths

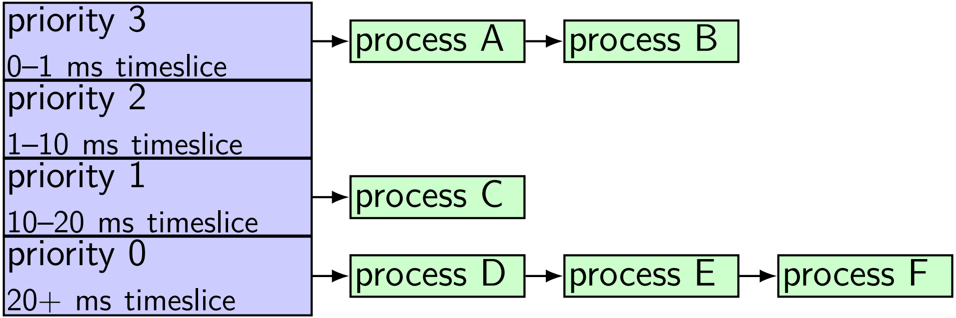
seems hard

sometimes you can ask

- common in batch job scheduling systems

and maybe you'll get accurate answers, even

approximating SJF with priorities



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

priority level = allowed time quantum

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

the SJF/SRTF problem

so, bucket implies CPU burst length

well, how does one figure that out?

the SJF/SRTF problem

so, bucket implies 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

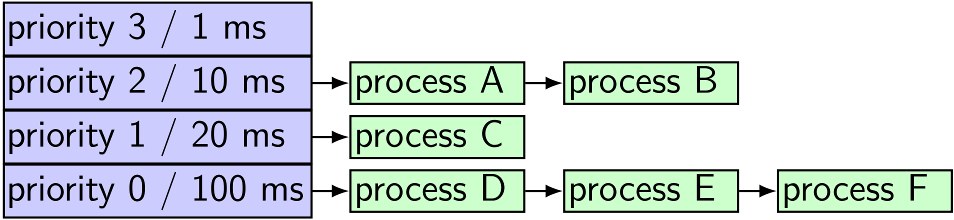
worst case: need to run the program to figure it out

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

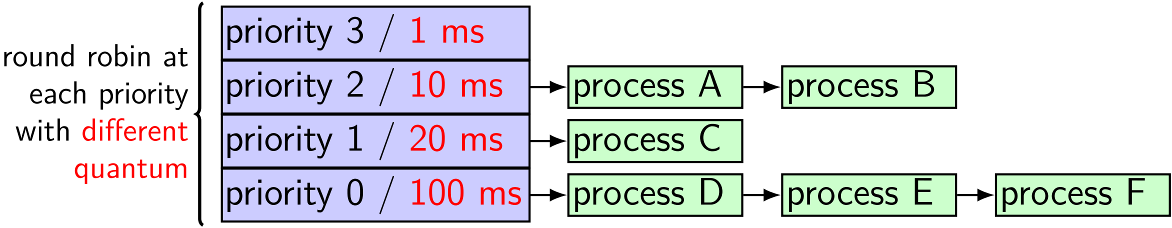
taking advantage of history

idea: priority = CPU burst length



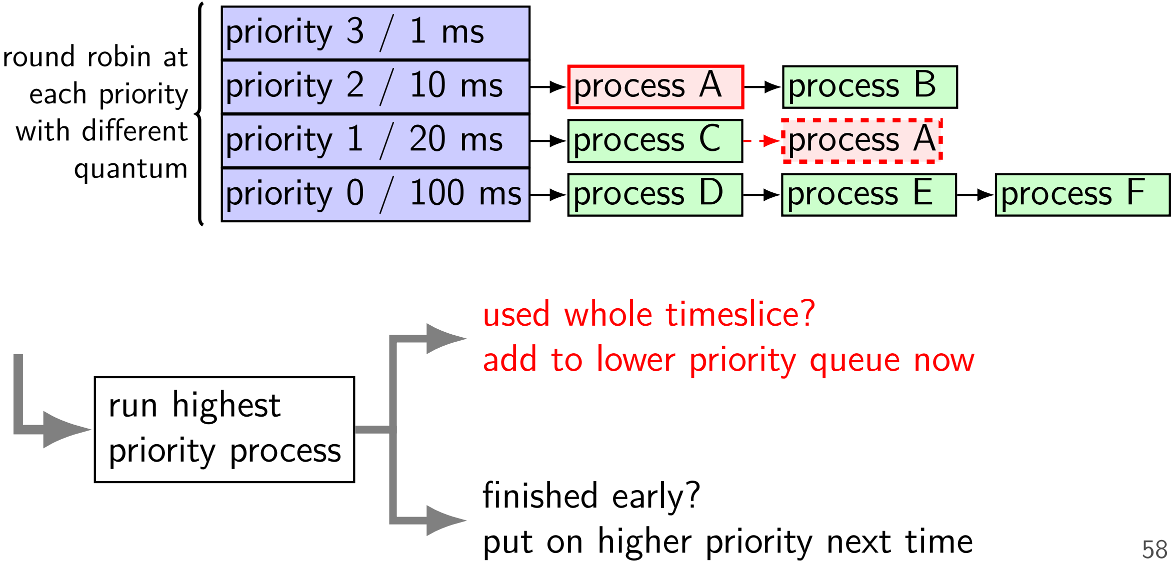
taking advantage of history

idea: priority = CPU burst length



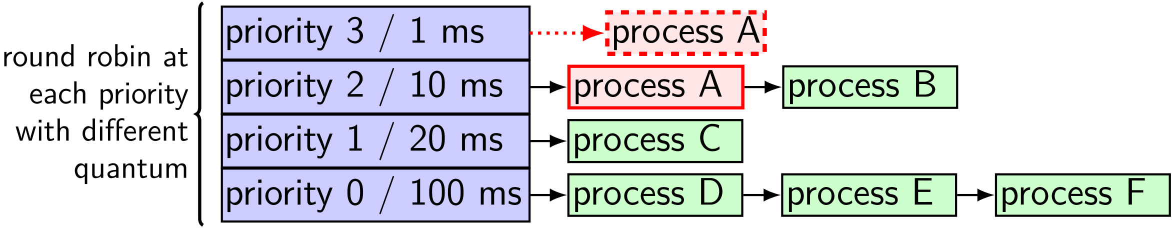
taking advantage of history

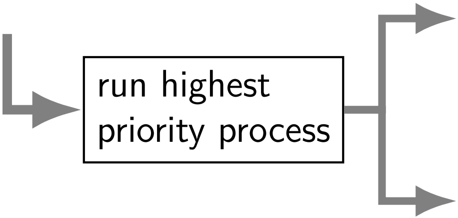
idea: priority = CPU burst length



taking advantage of history

idea: priority = CPU burst length





used whole timeslice?
add to lower priority queue now

finished early?
put on higher priority next time

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