scheduling 2

last time

xv6 scheduler design

- global array of processes (process table)
- seperate scheduler thread (per core) for convenience)
- lock to control changes to process table
- implicit queues (search process table for state)

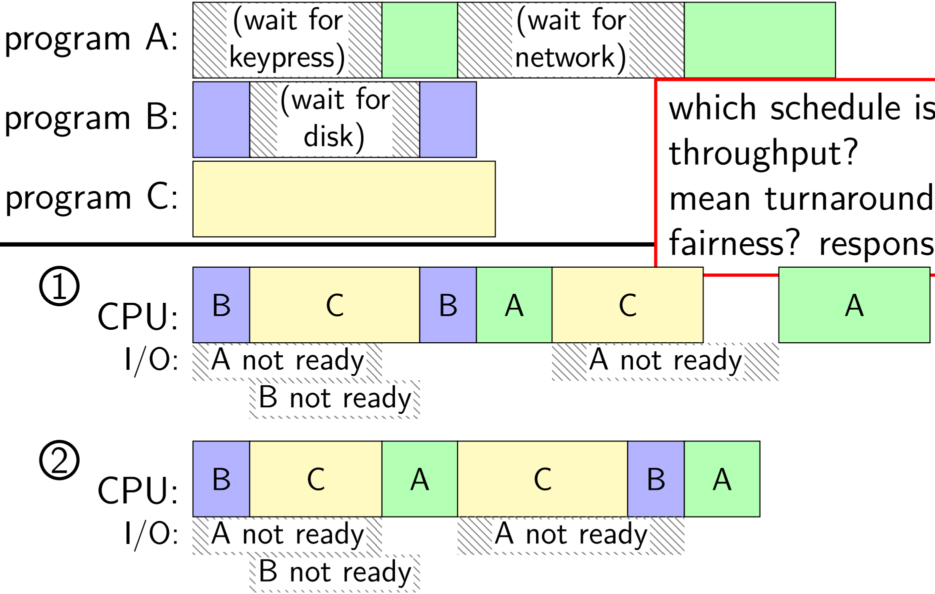
CPU and I/O bursts

scheduling policy = what to remove from queues

scheduling metrics

- throughput — useful work per unit time
- turnaround time — time from when becomes runnable to finishes running
- fairness
- …?

metrics example/exercise (2)



which schedule is better for:
throughput?
mean turnaround time?
fairness? responsiveness?

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)

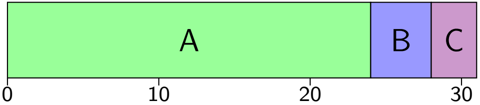
(AKA "first in, first out" (FIFO))

thread	CPU time needed
A	24
B	4
C	3

A ~ CPU-bound

B, C ~ I/O bound or interactive

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



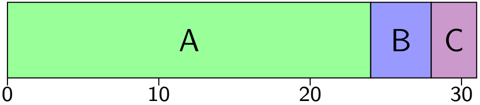
first-come, first-served (FCFS)

(AKA “first in, first out” (FIFO))

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**



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

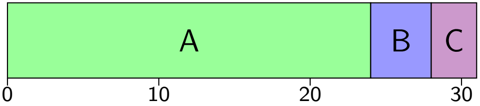
first-come, first-served (FCFS)

(AKA “first in, first out” (FIFO))

thread	CPU time needed
A	24
B	4
C	3

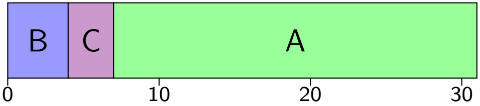
A ∼ CPU-bound
B, C ∼ I/O bound or interactive

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



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

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



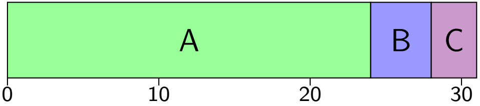
first-come, first-served (FCFS)

(AKA “first in, first out” (FIFO))

thread	CPU time needed
A	24
B	4
C	3

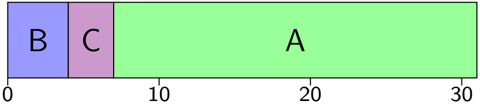
A $\sim$ CPU-bound
B, C $\sim$ I/O bound or interactive

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



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

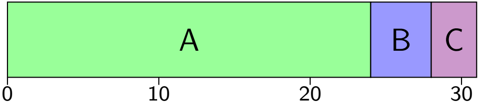
arrival order: **B**, **C**, **A**



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

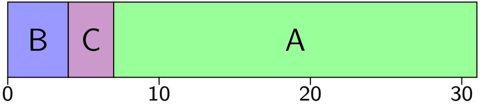
FCFS orders

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



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

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



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**

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

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

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**

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

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

RR with
short quantum

FCFS

smaller quantum: more fair, worse throughput

round robin and time quantums

many context switches
(lower throughput)

order doesn't matter
(more fair)

RR with
short quantum

few context switches
(higher throughput)

first program favored
(less fair)

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

- 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)

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 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 or stay the same

C. increase or decrease or stay the same

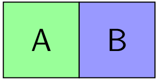
D. something else?

increase mean turnaround time

A: 1 unit CPU burst

B: 1 unit

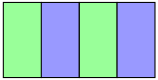
Q $=$ 1



mean turnaround time $=$

$(1 + 2) \div 2 = 1.5$

Q $=$ 1/2



mean turnaround time $=$

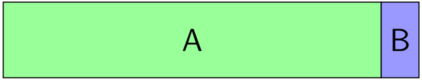
$(1.5 + 2) \div 2 = 1.75$

decrease mean turnaround time

A: 10 unit CPU burst

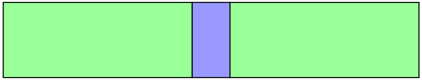
B: 1 unit

Q = 10



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

Q = 5



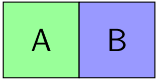
mean turnaround time $=$ $(6 + 11) \div 2 = 8.5$

stay the same

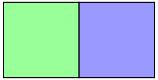
A: 1 unit CPU burst

B: 1 unit

$Q = 10$



$Q = 1$



FCFS and order

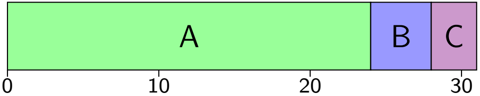
earlier we saw that with FCFS, arrival order mattered

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**



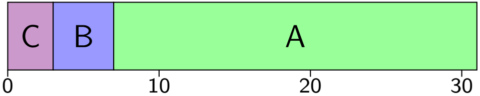
waiting times: (mean=17.3)

0 (**A**), 24 (**B**), 28 (**C**)

turnaround times: (mean=27.7)

24 (**A**), 28 (**B**), 31 (**C**)

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



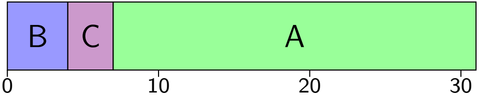
waiting times: (mean=3.3)

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

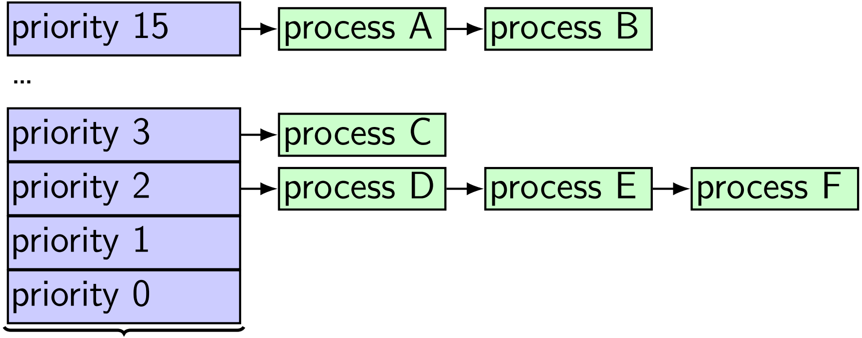
shells more important than big computation?

- i.e. programs with short CPU bursts

faculty more important than students?

scheduling algorithm: schedule shells/faculty programs first

priority scheduling



ready queues for each priority level

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

fair scheduling

what is the fairest scheduling we can do?

intuition: every thread has an equal chance to be chosen

random scheduling algorithm

“fair” scheduling algorithm: choose uniformly at random

good for “fairness”

bad for response time

bad for predictability

proportional share

maybe every thread isn't equal

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

proportional share

maybe every thread isn't equal

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

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

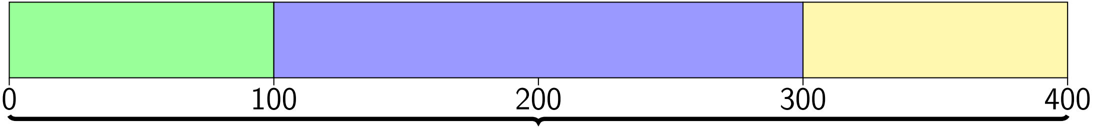
proportional share

lottery scheduling

every thread has a certain number of lottery tickets:

A	B	C
100 tickets	200 tickets	100 tickets

scheduling = lottery among ready threads:



choose random number in this range to find winner

simulating priority with lottery

A (high priority)
1M tickets

B (medium priority)
1K tickets

C (low priority)
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 processes scheduled (for testing)

lottery scheduling assignment

assignment: add lottery scheduling to xv6

extra system call: `settickets`

also counting of how often processes scheduled (for testing)

simplification: okay if scheduling decisions are linear time

- there is a faster way

not implementing preemption before time slice ends

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

is lottery scheduling actually good?

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

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

elegant mecahnism that can implement a variety of policies

but there are some problems…

exercise

thread A: 1 ticket, always runnable

thread B: 9 tickets, always runnable

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

- i.e. 3 times as much as “it’s supposed to”
- chosen 3 times out of 10 instead of 1 out of 10

exercise

thread A: 1 ticket, always runnable

thread B: 9 tickets, always runnable

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

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

A runs w/in 10 times...

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

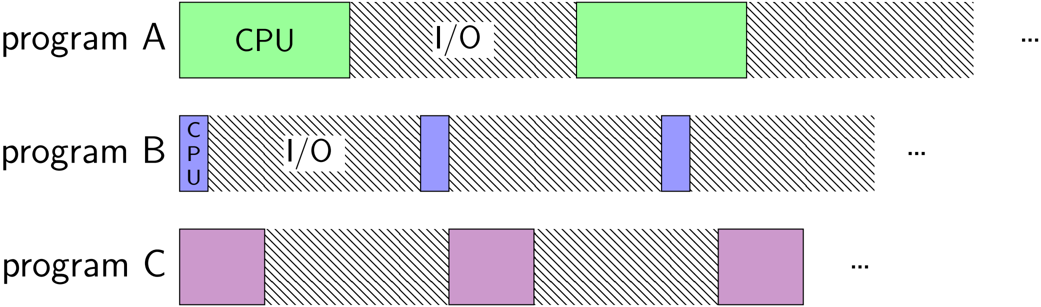
so we want to run the shortest CPU burst first

how do I tell which thread that is?

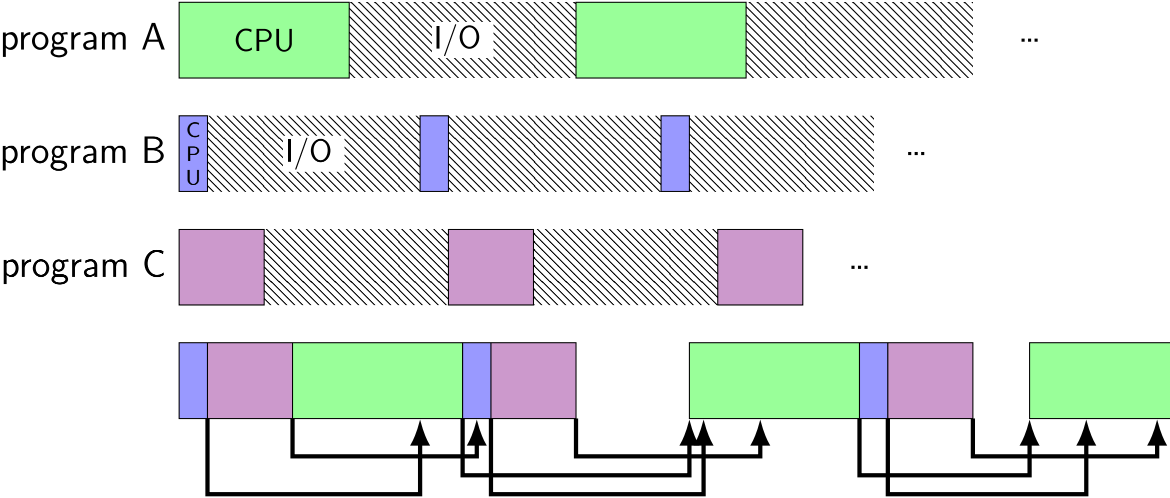
we'll deal with this problem later

...kinda

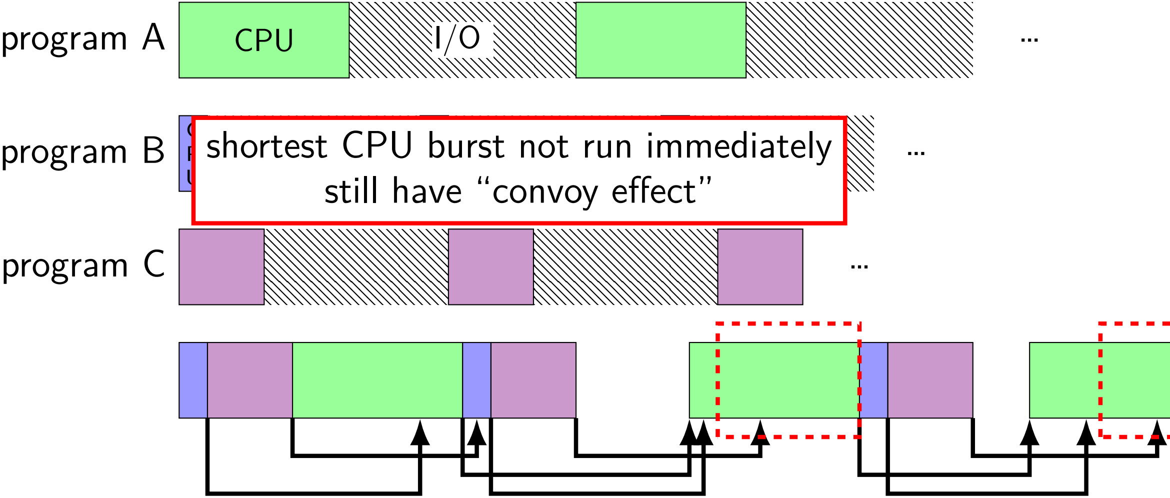
alternating I/O and CPU: SJF



alternating I/O and CPU: SJF



alternating I/O and CPU: SJF



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