Changelog

Changes made in this version not seen in first lecture:

13 September: replace 'multi-level queue' with 'multi-level feedback queue' throughout

last time

the xv6 scheduler
- loop through process table
- dedicated thread for scheduler
- swtch to scheduler to give up CPU
- scheduler switches back to you

scheduling metrics
- throughput, response time, fairness

first-come first-served (FCFS), round robin (RR)
- simple non-preemptive, preemptive scheduler

priority scheduling

shortest job first

scheduler HW timing note

on extensions and late policies

there is a late policy

- -10% 72 hours; -20% week

I generally don't do extensions for the whole class

- (exceptions: problems with submission system/weather/etc.)
- if someone made sure they completed the assignment on time...

I might do late penalty adjustments

execv and const

```
int execv(const char *path, char *const *argv);
```

`argv` is a pointer to constant pointer to char

probably should be a pointer to constant pointer to *constant* char

...this causes some awkwardness:

```
const char *array[] = { /* ... */ };
execv(path, array); // ERROR
```

solution: cast

```
const char *array[] = { /* ... */ };
execv(path, (char **) array); // or (char * const *)
```

shell HW Q&A time

minimizing response 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

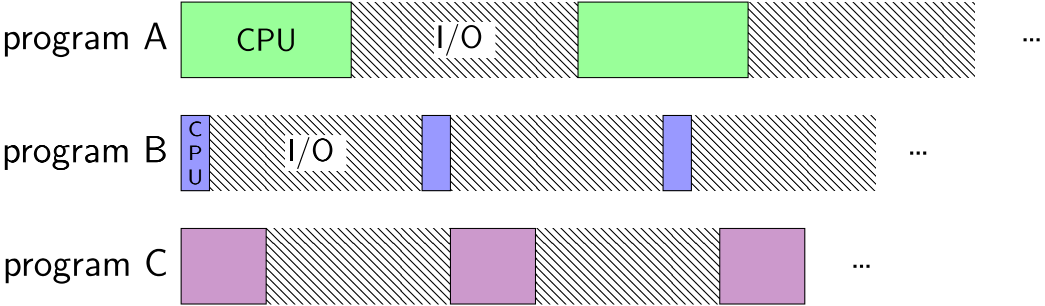
so we want to run the shortest CPU burst first

how do I tell which thread that is?

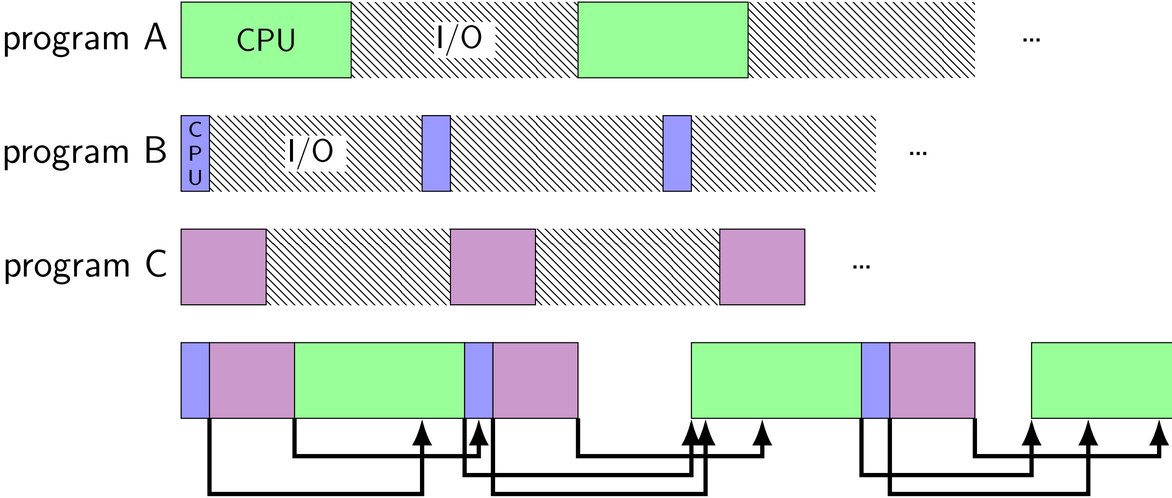
we'll deal with this problem later

...kinda

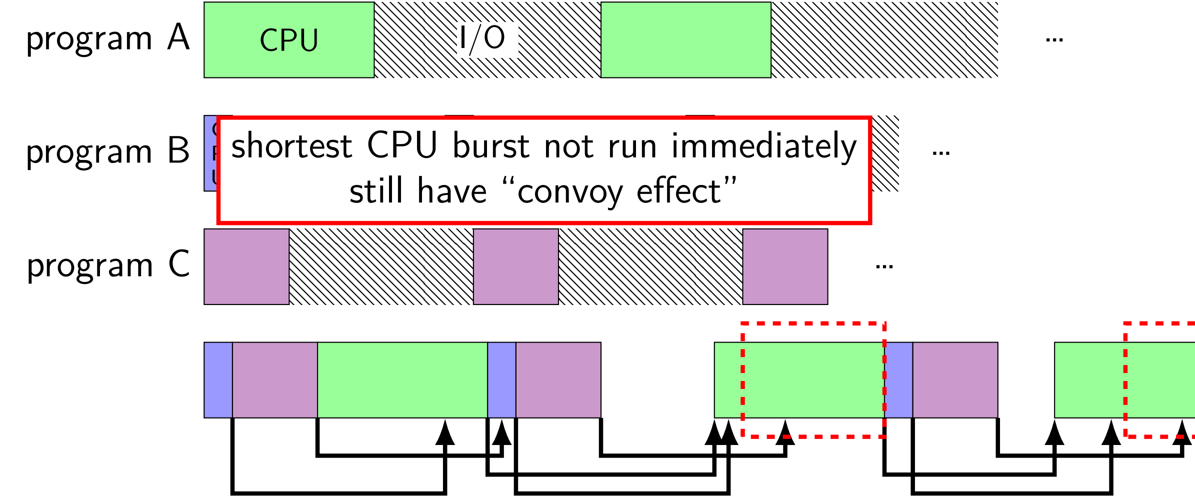
alternating I/O and CPU: SJF



alternating I/O and CPU: SJF



alternating I/O and CPU: SJF



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

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?

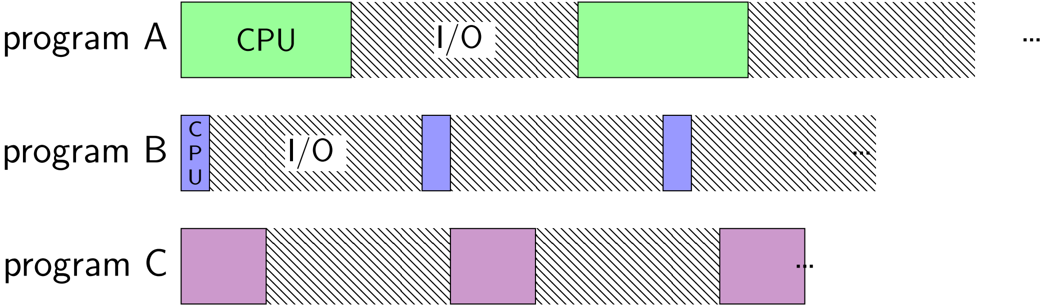
- recall: priority = job length
- long job waits for medium job
- ...for longer than it would take to finish
- worse than letting long job finish

solution: priority = remaining time

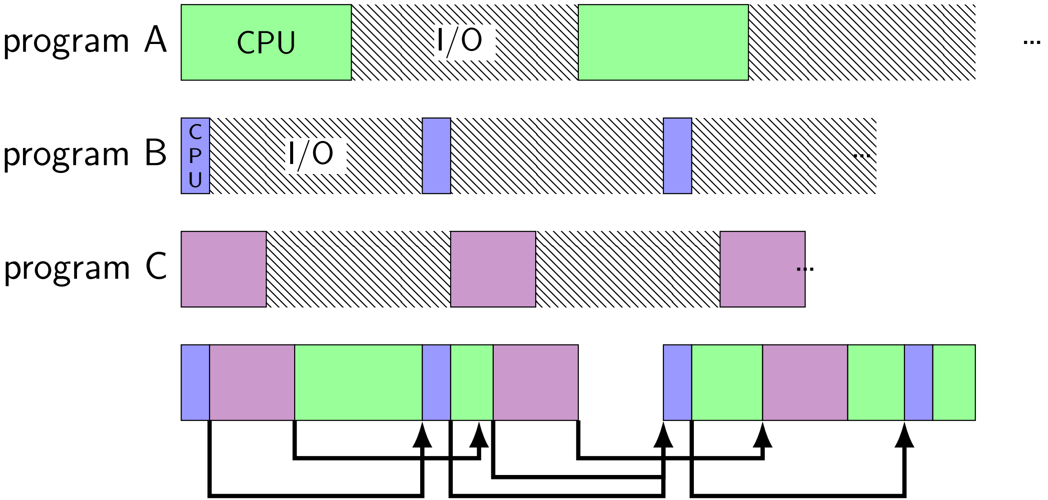
called shortest *remaining time* first (SRTF)

- prioritize by what's left, not the total

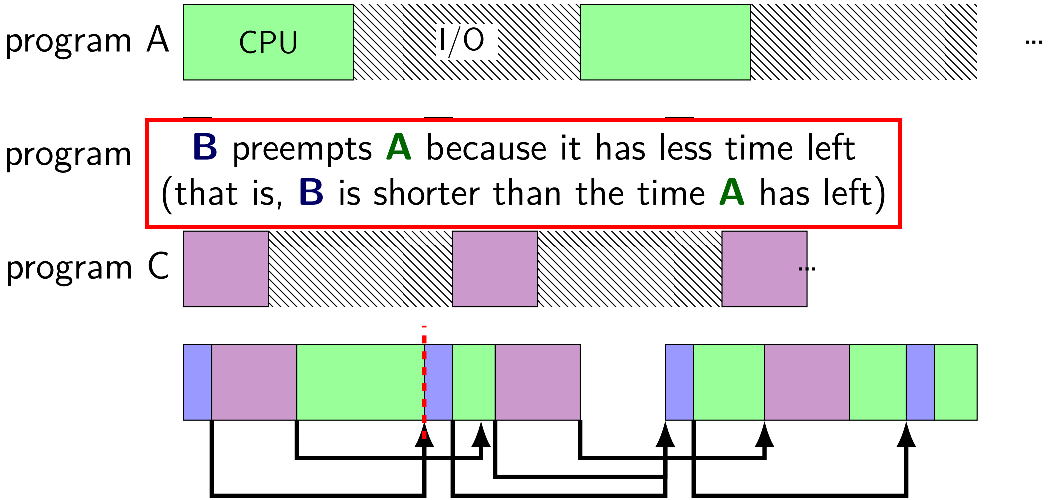
alternating I/O and CPU: SRTF



alternating I/O and CPU: SRTF

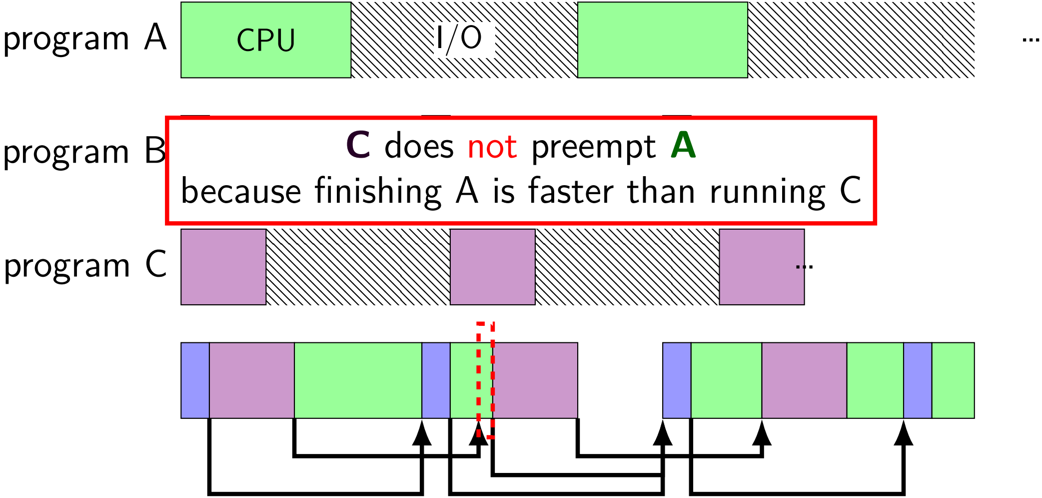


alternating I/O and CPU: SRTF



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



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

knowing job lengths

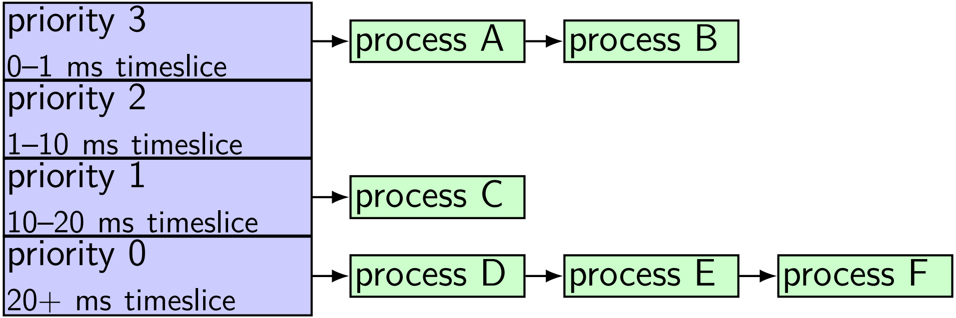
seems hard

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)

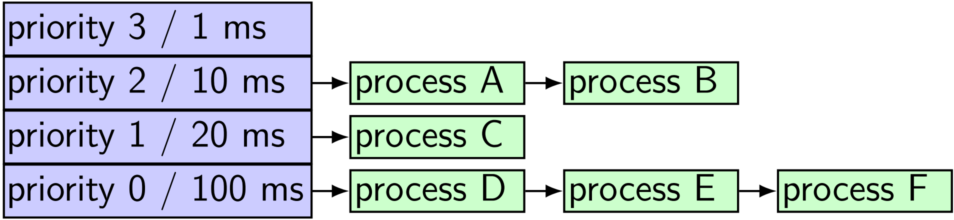
key observation: CPU bursts now are like CPU bursts later

intuition: interactive program with lots of I/O tends to stay interactive

intuition: CPU-heavy program is going to keep using CPU

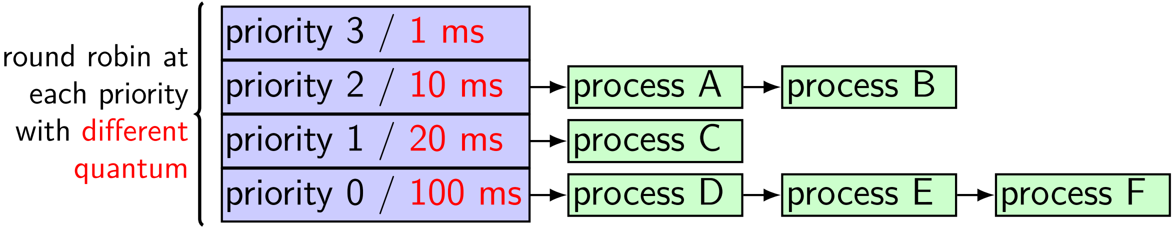
taking advantage of history

idea: priority = CPU burst length



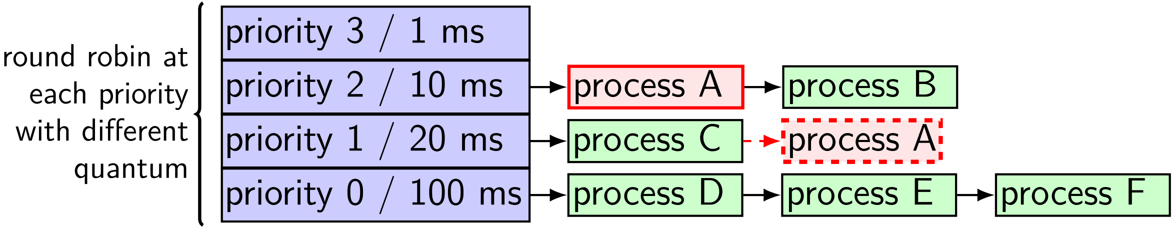
taking advantage of history

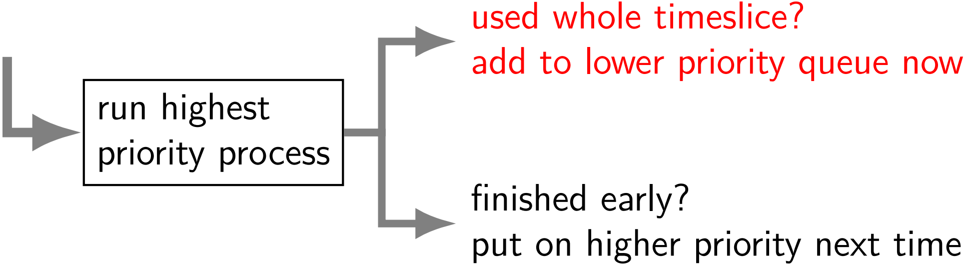
idea: priority = CPU burst length



taking advantage of history

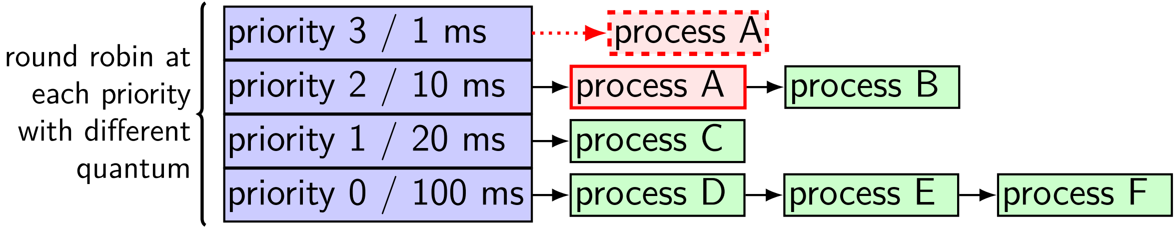
idea: priority = CPU burst length

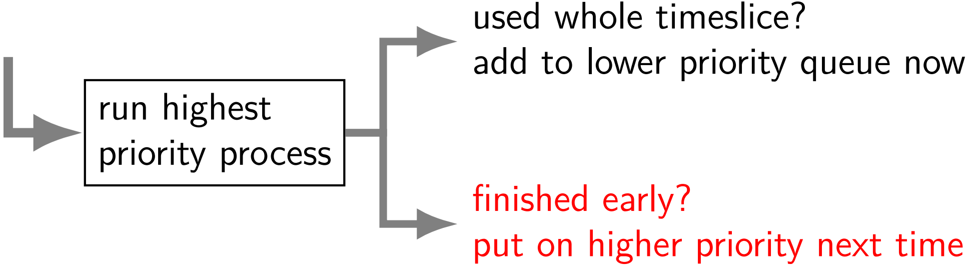




taking advantage of history

idea: priority = CPU burst length





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

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

most fair | least fair

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

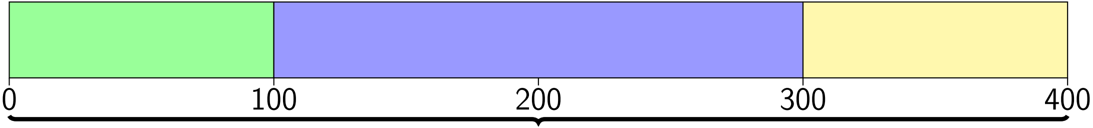
proportional share

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

very close to strict priority

...or to SJF if priorities are set 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%

backup slides

exercise

```c
pid_t p = fork();
int pipe_fds[2];
pipe(pipe_fds);
if (p == 0) { /* child */
  close(pipe_fds[0]);
  char c = 'A';
  write(pipe_fds[1], &c, 1);
  exit();
} else { /* parent */
  close(pipe_fds[1]);
  char c;
  int count = read(pipe_fds[0], &c, 1);
  printf("read %d bytes\n", count);
}
```

The child is trying to send the character A to the parent.

But the above code outputs `read 0 bytes` instead of `read 1 bytes`.

What happened?

exercise solution

pipe() is after fork — two pipes, one in child, one in parent

exercise

```
int pipe_fds[2]; pipe(pipe_fds);
pid_t p = fork();
if (p == 0) {
  close(pipe_fds[0]);
  for (int i = 0; i < 10; ++i) {
    char c = '0' + i;
    write(pipe_fds[1], &c, 1);
  }
  exit();
}
close(pipe_fds[1]);
char buffer[10];
ssize_t count = read(pipe_fds[0], buffer, 10);
for (int i = 0; i < count; ++i) {
  printf("%c", buffer[i]);
}
```

Which are possible outputs (if pipe, read, write, fork don't fail)?

A. 0123456789 B. 0 C. (nothing)

D. A and B E. A and C F. A, B, and C

exercise

```
int pipe_fds[2]; pipe(pipe_fds);
pid_t p = fork();
if (p == 0) {
  close(pipe_fds[0]);
  for (int i = 0; i < 10; ++i) {
    char c = '0' + i;
    write(pipe_fds[1], &c, 1);
  }
  exit();
}
close(pipe_fds[1]);
char buffer[10];
ssize_t count = read(pipe_fds[0], buffer, 10);
for (int i = 0; i < count; ++i) {
  printf("%c", buffer[i]);
}
```

Which are possible outputs (if pipe, read, write, fork don't fail)?

A. 0123456789 B. 0 C. (nothing)

D. A and B E. A and C F. A, B, and C

partial reads

read returning 0 always means end-of-file

- by default, read always waits *if no input available yet*
- but can set read to return *error* instead of waiting

read can return less than requested if not available

- e.g. child hasn't gotten far enough