Scheduling 3 / Threading 0

last time

scheduling metrics: turnaround/wait time, throughput, fairness

simple scheduling policies: FCFS, RR

optimizing for turnaround tinme: SJF/SRTF

approximating SRTF: MFQ

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

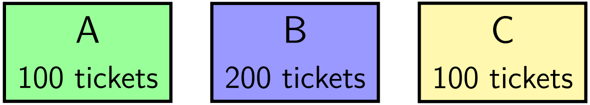
if thread A is twice as important as thread B, then…

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

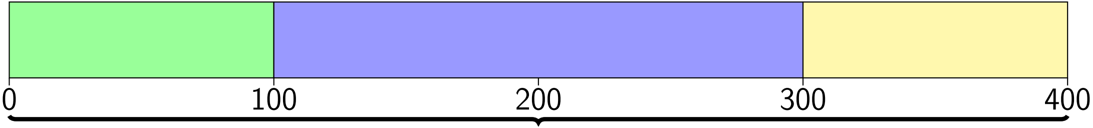
proportional share

lottery scheduling

every thread has a certain number of lottery tickets:



scheduling = lottery among ready threads:



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

...or to SJF if priorities are set right

simulating priority with lottery

A (high priority)
1M tickets

B (medium priority)
1K tickets

C (low priority)
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%

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

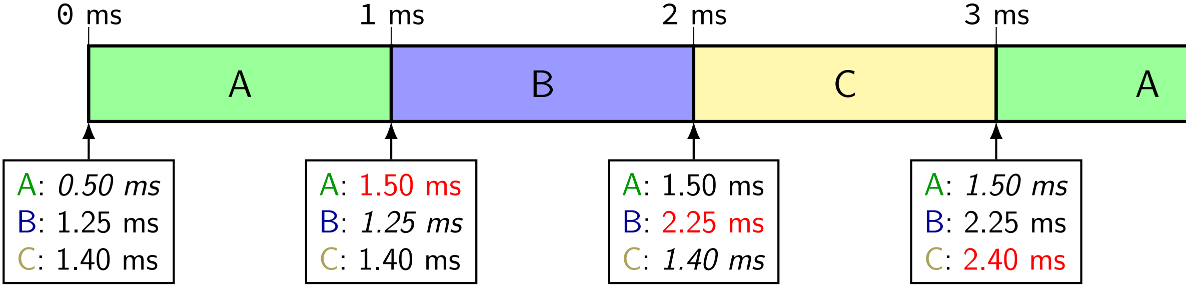
adjustments for threads that are new or were sleeping

- 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



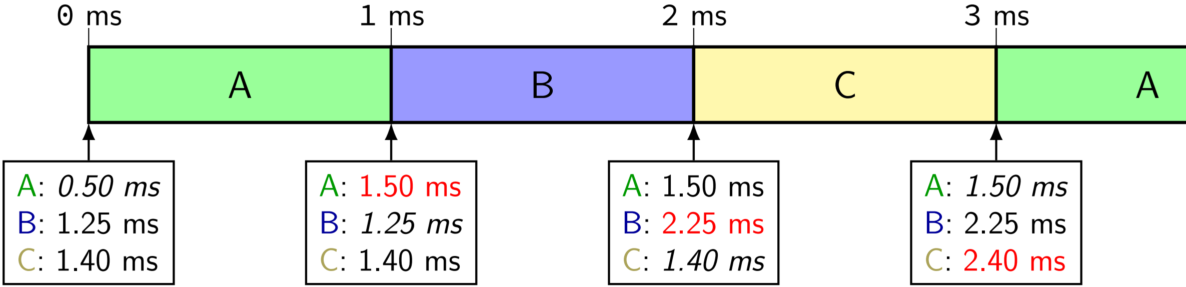
A: *0.50 ms*
B: 1.25 ms
C: 1.40 ms

A: 1.50 ms
B: *1.25 ms*
C: 1.40 ms

A: 1.50 ms
B: 2.25 ms
C: *1.40 ms*

A: *1.50 ms*
B: 2.25 ms
C: 2.40 ms

virtual time, always ready, 1 ms quantum



A: *0.50 ms*
B: 1.25 ms
C: 1.40 ms

A: 1.50 ms
B: *1.25 ms*
C: 1.40 ms

A: 1.50 ms
B: 2.25 ms
C: *1.40 ms*

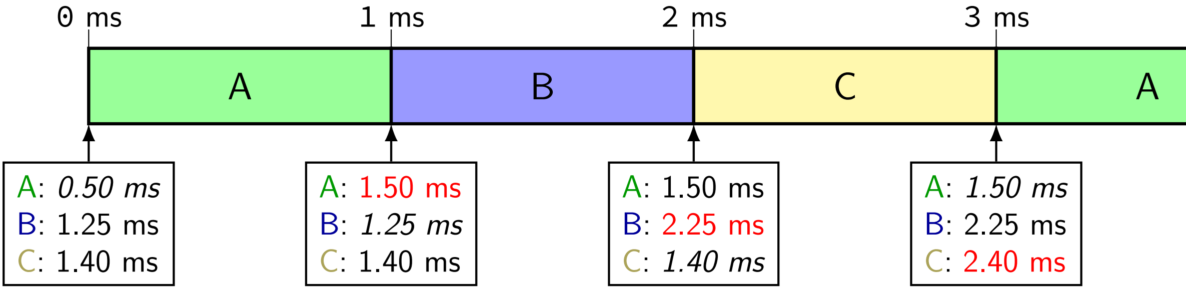
A: *1.50 ms*
B: 2.25 ms
C: 2.40 ms

at each time:

update current thread’s time

run thread with lowest total time

virtual time, always ready, 1 ms quantum



A: *0.50 ms*
B: 1.25 ms
C: 1.40 ms

A: 1.50 ms
B: *1.25 ms*
C: 1.40 ms

A: 1.50 ms
B: 2.25 ms
C: *1.40 ms*

A: *1.50 ms*
B: 2.25 ms
C: 2.40 ms

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)

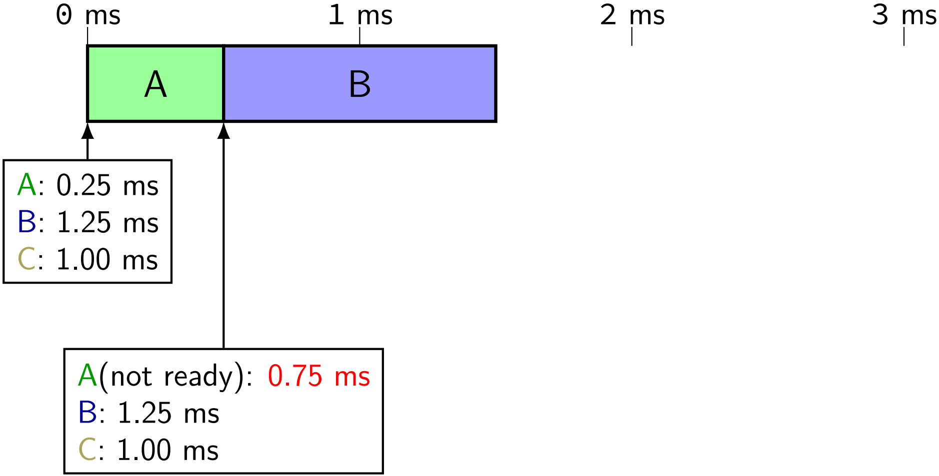
not runnable briefly? still get your share of CPU

- (catch up from prior virtual time)

not runnable for a while? get bounded advantage

A doesn't use whole time…

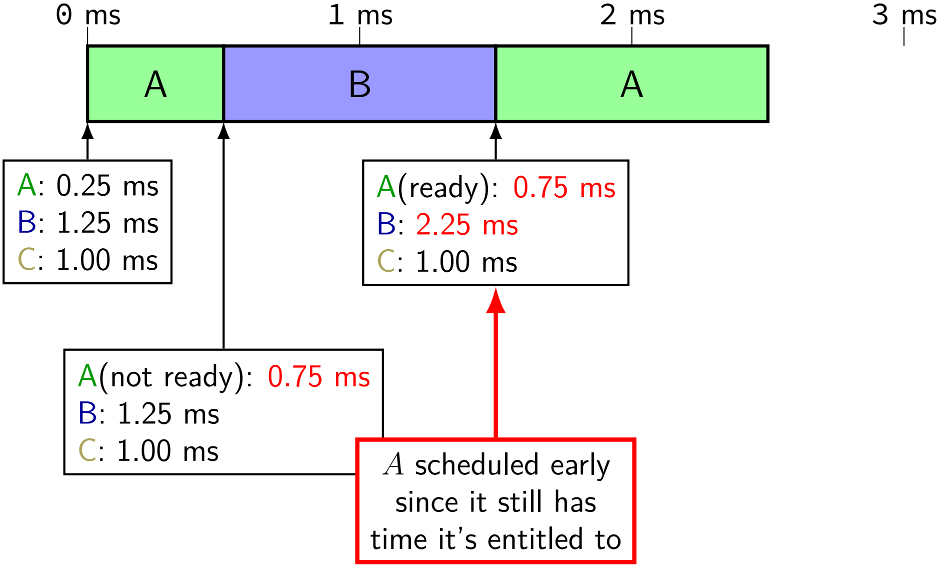
A doesn’t use whole time...



A: 0.25 ms
B: 1.25 ms
C: 1.00 ms

A(not ready): 0.75 ms
B: 1.25 ms
C: 1.00 ms

A doesn't use whole time...



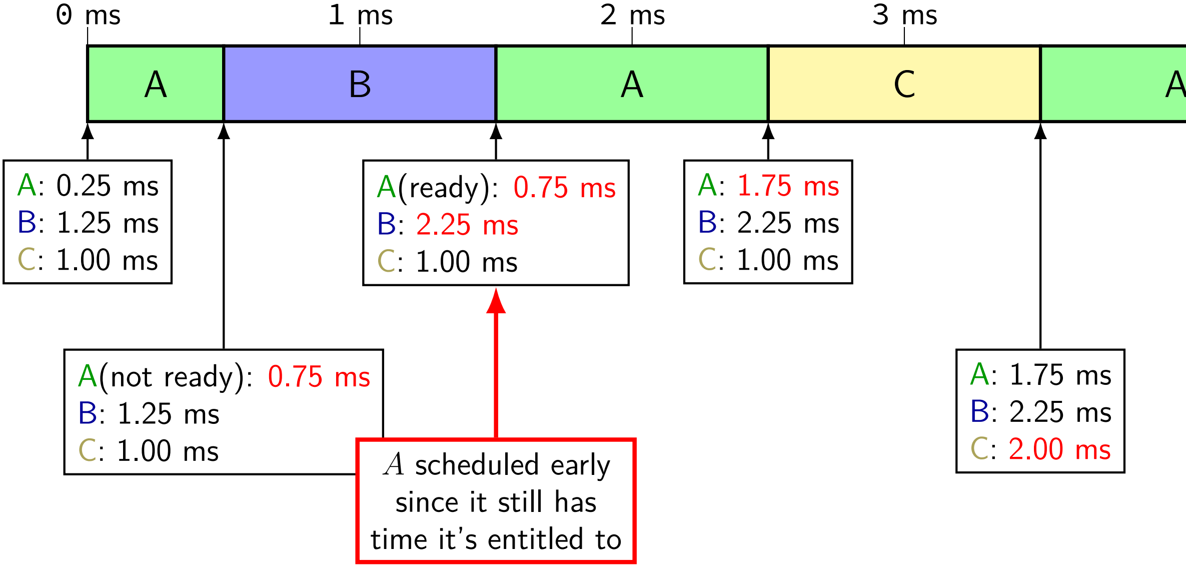
A: 0.25 ms
B: 1.25 ms
C: 1.00 ms

A(not ready): 0.75 ms
B: 1.25 ms
C: 1.00 ms

A(ready): 0.75 ms
B: 2.25 ms
C: 1.00 ms

A scheduled early since it still has time it's entitled to

A doesn't use whole time...



A: 0.25 ms
B: 1.25 ms
C: 1.00 ms

A(not ready): 0.75 ms
B: 1.25 ms
C: 1.00 ms

A(ready): 0.75 ms
B: 2.25 ms
C: 1.00 ms

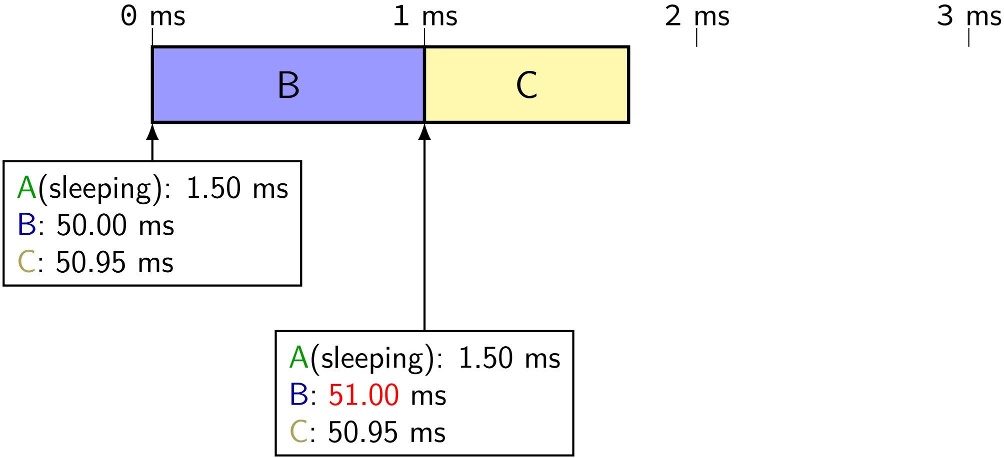
A: 1.75 ms
B: 2.25 ms
C: 1.00 ms

A: 1.75 ms
B: 2.25 ms
C: 2.00 ms

A scheduled early since it still has time it's entitled to

A's long sleep...

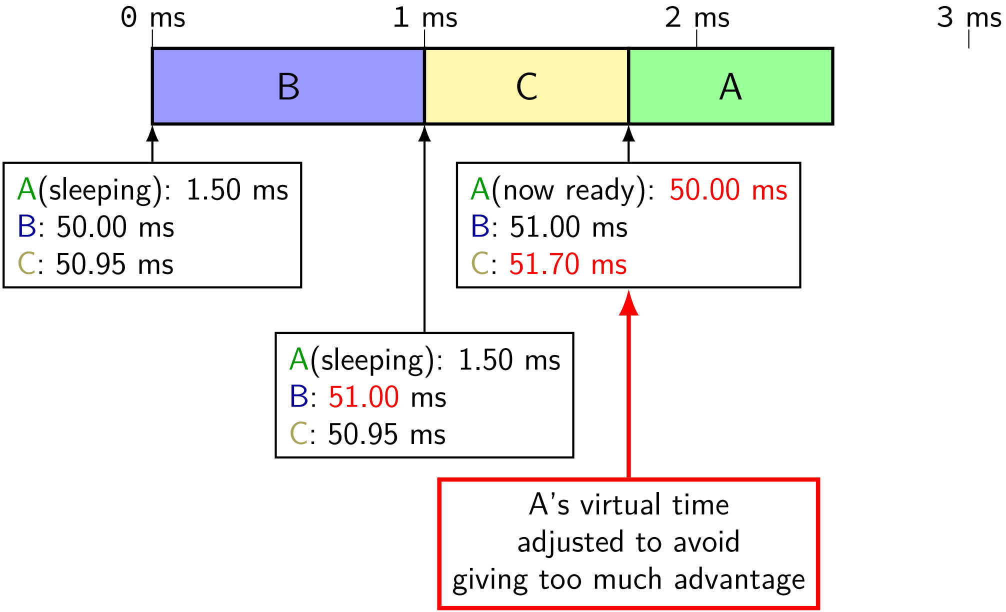
A's long sleep...



A(sleeping): 1.50 ms
B: 50.00 ms
C: 50.95 ms

A(sleeping): 1.50 ms
B: 51.00 ms
C: 50.95 ms

A's long sleep...



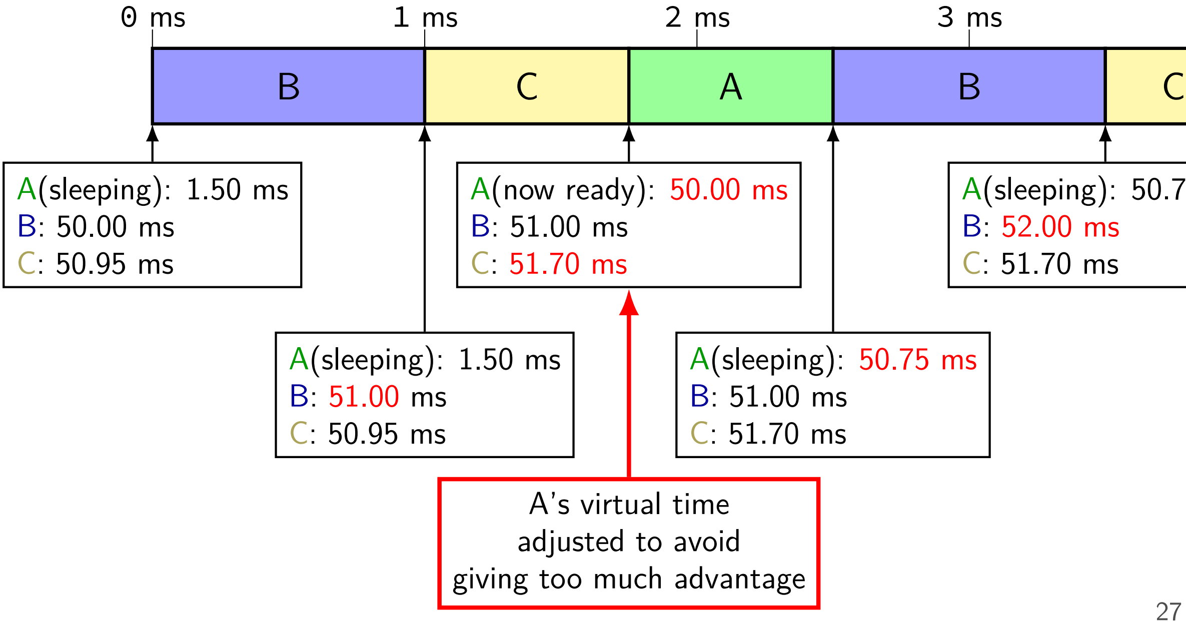
A(sleeping): 1.50 ms
B: 50.00 ms
C: 50.95 ms

A(sleeping): 1.50 ms
B: 51.00 ms
C: 50.95 ms

A(now ready): 50.00 ms
B: 51.00 ms
C: 51.70 ms

A's virtual time
adjusted to avoid
giving too much advantage

A’s long sleep...



A(sleeping): 1.50 ms
B: 50.00 ms
C: 50.95 ms

A(sleeping): 1.50 ms
B: 51.00 ms
C: 50.95 ms

A(now ready): 50.00 ms
B: 51.00 ms
C: 51.70 ms

A(sleeping): 50.75 ms
B: 51.00 ms
C: 51.70 ms

A(sleeping): 50.7
B: 52.00 ms
C: 51.70 ms

A’s virtual time
adjusted to avoid
giving too much advantage

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)

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?

real-time

so far: “best effort” scheduling

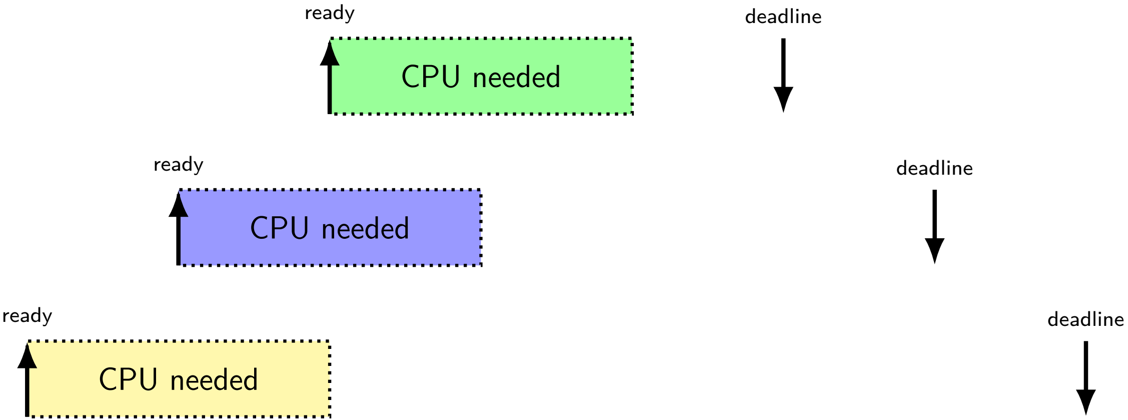
- best possible (by some metrics) given some work

alternate model: need gaurnetees

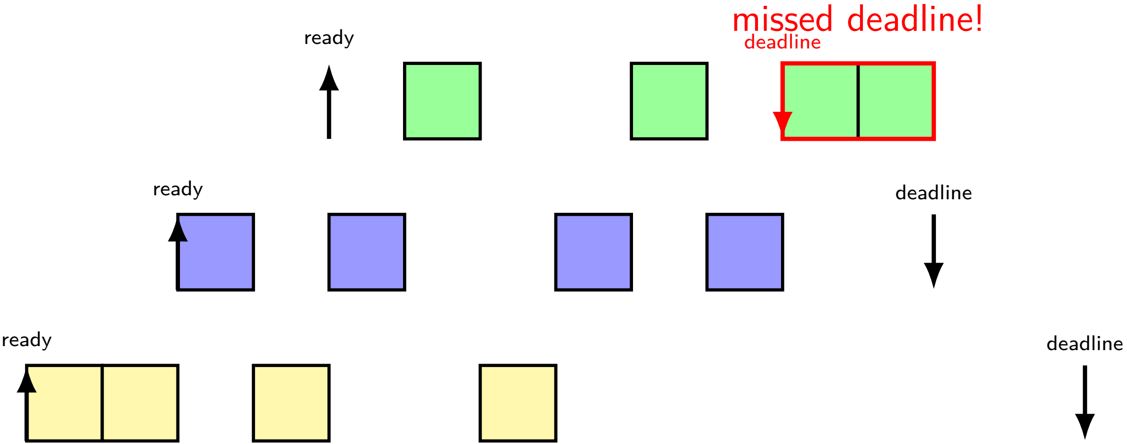
deadlines imposed by real-world

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

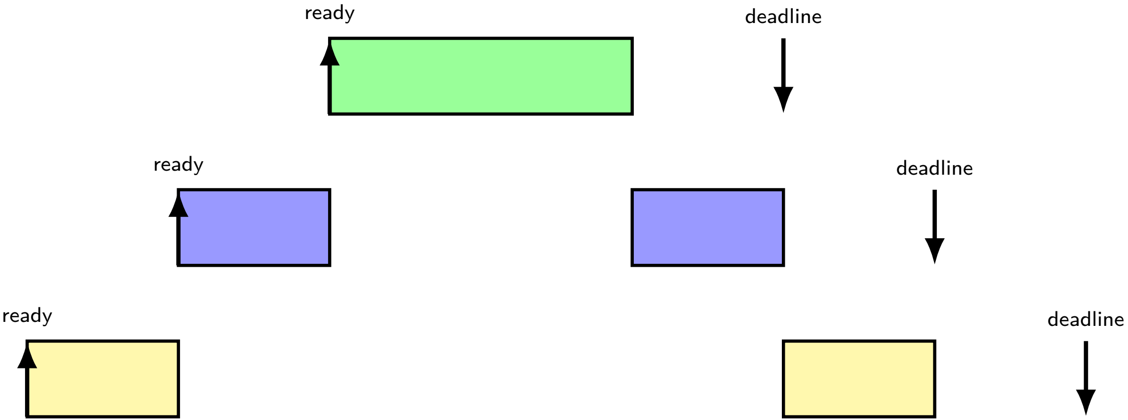
real time example: CPU + deadlines



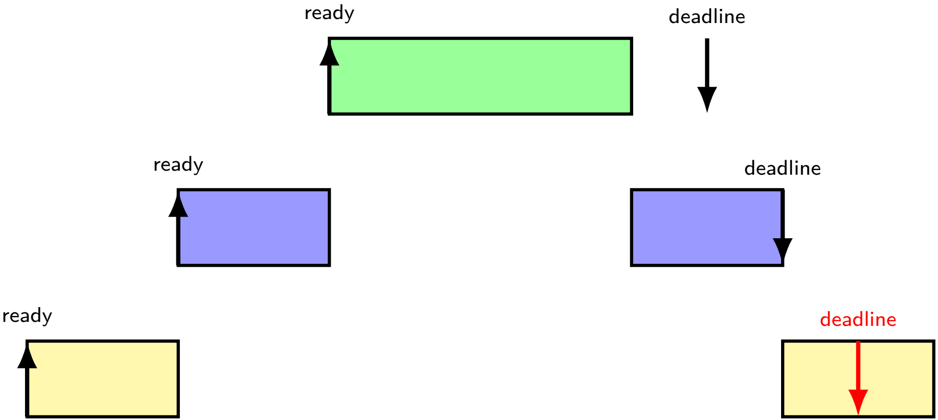
example with RR



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

threads versus processes

for now — each process has one thread

Anderson-Dahlin talks about thread scheduling

thread = part that gets run on CPU
- saved register values (including own stack pointer)
- save program counter

rest of process
- address space
- open files
- current working directory
- …

xv6 processes versus threads

xv6: one thread per process

so part of the process control block
is really a *thread control block*

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  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)
};
```

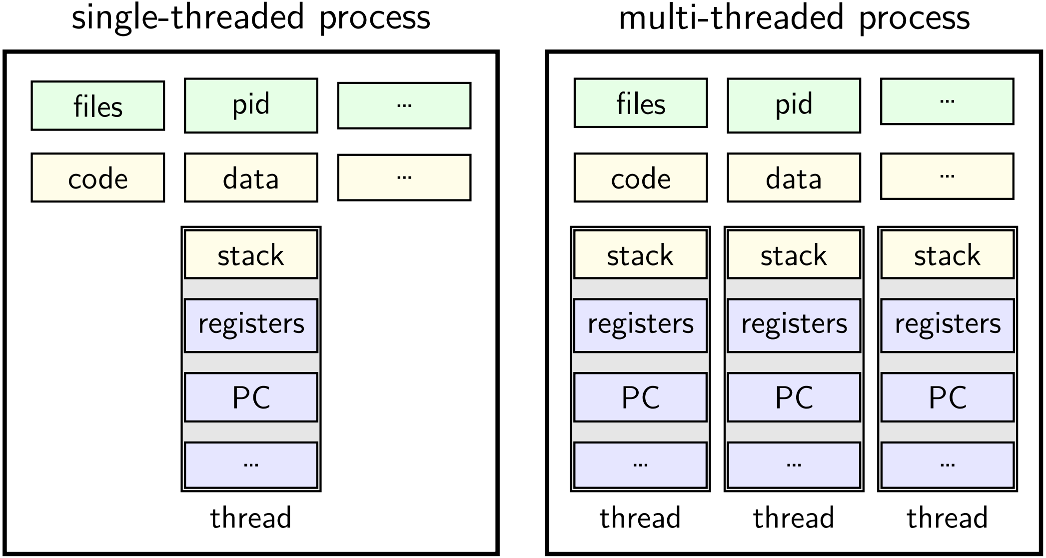
xv6 processes versus threads

xv6: one thread per process

so part of the process control block
is really a *thread control block*

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  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)
};
```

single and multithread processes



thread versus process state

thread state — kept in **thread control block**

- registers (including program counter)
- other information?

process state — kept in **process control block**

- address space (memory layout)
- open files
- process id
- …

Linux idea: task_struct

Linux model: single "task" structure = thread

pointers to address space, open file list, etc.

pointers can be shared — if same process

`fork()`-like system call "clone": choose what to share

- `clone(CLONE_FILES, ...)` — new process **sharing** open files
- `clone(CLONE_VM, ...)` — new process **sharing** address spaces

Linux idea: task_struct

Linux model: single "task" structure = thread

pointers to address space, open file list, etc.

pointers can be shared — if same process

`fork()`-like system call "clone": choose what to share

- `clone(CLONE_FILES, ...)` — new process **sharing** open files
- `clone(CLONE_VM, ...)` — new process **sharing** address spaces

advantage: no special logic for threads (mostly)

aside: alternate threading models

we'll talk about kernel threads

OS scheduler deals directly with threads

alternate idea: library code handles threading

kernel doesn't know about threads w/in process

hierarchy of schedulers: one for processes, one within each process

not currently common model — awkward with multicore

why threads?

concurrency: different things happening at once
- one thread per user of web server?
- one thread per page in web browser?
- one thread to play audio, one to read keyboard, ...?
- ...

parallelism: do same thing with more resources
- multiple processors to speed-up simulation (life assignment)

pthread_create

```
void *ComputePi(void *argument) { ...  }
void *PrintClassList(void *argument) { ...  }
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```

run ComputePi and PrintClassList at the same time

also run “more code”

pthread_create

thread identifier — used to perform operations on thread later

```
void
void
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```

run ComputePi and PrintClassList at the same time

also run “more code”

pthread_create

function to run — thread starts here, terminate if function returns

```
vo
void  PrintClassList(void  argument) { ... }
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```

run ComputePi and PrintClassList at the same time

also run “more code”

pthread_create

thread attributes (extra settings) and function argument

```
void *C
void *P
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```

run ComputePi and PrintClassList at the same time

also run “more code”

a threading race

```
#include <pthread.h>
#include <stdio.h>
void *print_message(void *ignored_argument) {
    printf("In␣the␣thread\n");
    return NULL;
}
int main() {
    printf("About␣to␣start␣thread\n");
    pthread_t the_thread;
    pthread_create(&the_thread, NULL, print_message, NULL);
    printf("Done␣starting␣thread\n");
    return 0;
}
```

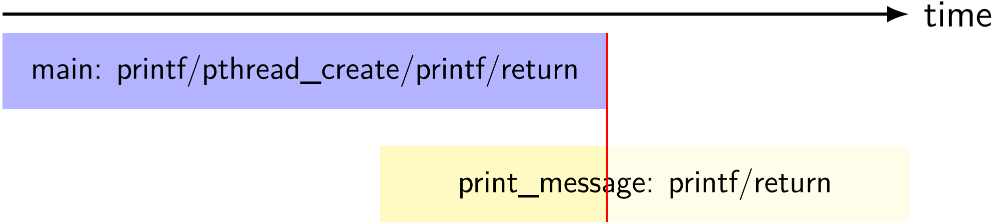
My machine: outputs `In the thread` about 4% of the time.

What happened?

a race

returning from main exits the entire process (all threads)

race: main's return 0 or print_message's printf first?



return from main
ends all threads
in the process

fixing the race (version 1)

```
#include <pthread.h>
#include <stdio.h>
void *print_message(void *ignored_argument) {
    printf("In␣the␣thread\n");
    return NULL;
}
int main() {
    printf("About␣to␣start␣thread\n");
    pthread_t the_thread;
    pthread_create(&the_thread, NULL, print_message, NULL);
    printf("Done␣starting␣thread\n");
    pthread_join(the_thread, NULL);  /* WAIT FOR THREAD */
    return 0;
}
```

fixing the race (version 2; not recommended)

```
#include <pthread.h>
#include <stdio.h>
void *print_message(void *ignored_argument) {
    printf("In␣the␣thread\n");
    return NULL;
}
int main() {
    printf("About␣to␣start␣thread\n");
    pthread_t the_thread;
    pthread_create(&the_thread, NULL, print_message, NULL);
    printf("Done␣starting␣thread\n");
    pthread_exit(NULL);
}
```

pthread_join, pthread_exit

`pthread_join`: wait for thread, returns its return value

- like `waitpid`, but for a thread
- return value is pointer to anything

`pthread_exit`: exit current thread, returning a value

- like `exit` or returning from main, but for a single thread
- same effect as returning from function passed to pthread_create