Scheduling 3 / Threads

last time

shortest job first/shortest remaining time first
- response time optimizing
- SJF — without preemption
- SRTF — with preemption

multi-level feedback scheduling
- priority $\sim$ quantum length
- process uses whole quantum? move down in priority
- process uses less than quantum? move up in priority (next time it runs)
- maybe extra work to avoid starvation

proportional share scheduling
- 2x share — 2x CPU time
- lottery scheduling — weighted random
- set weights to approximate priority or whatever wanted

lottery scheduler assignment

track "ticks" process runs

- = number of times scheduled
- simplification: don't care if process uses less than timeslice

new system call: getprocesesinfo

- copy info from process table into user space

new system call: settickets

- set number of tickets for current process
- should be inherited by fork

scheduler: choose pseudorandom weighted by tickets

- caution! no floating point

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

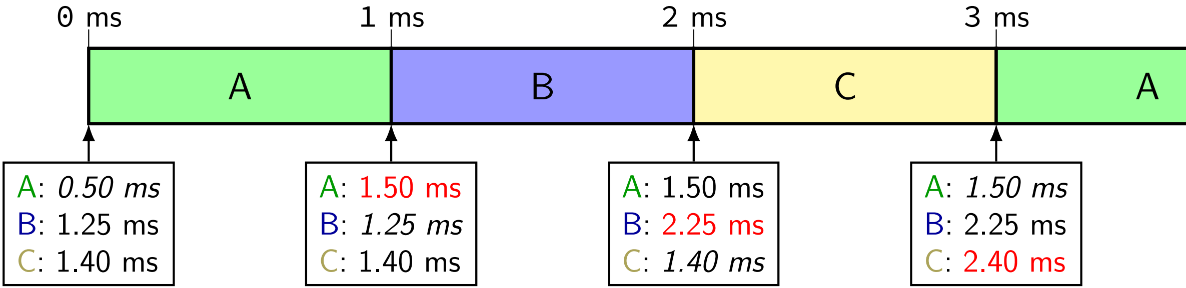
adjustments for threads that are new or were sleeping

- too big an advantage to start at runtime `0`

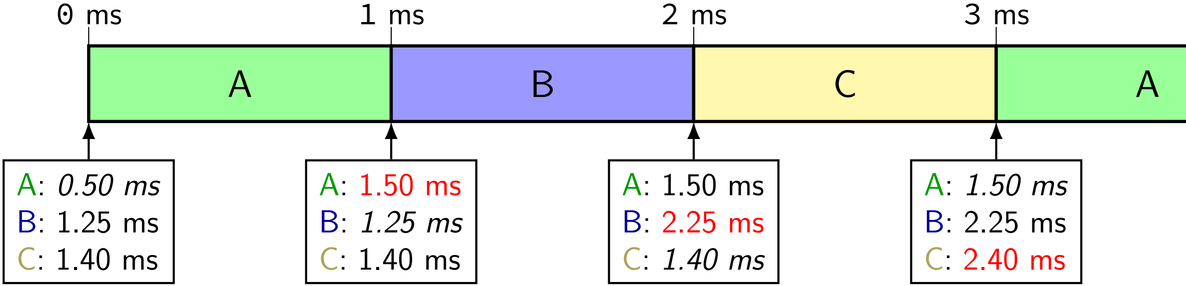
scheduling decision: run thread with lowest virtual runtime

- data structure: balanced tree

virtual time, always ready, 1 ms quantum



virtual time, always ready, 1 ms quantum

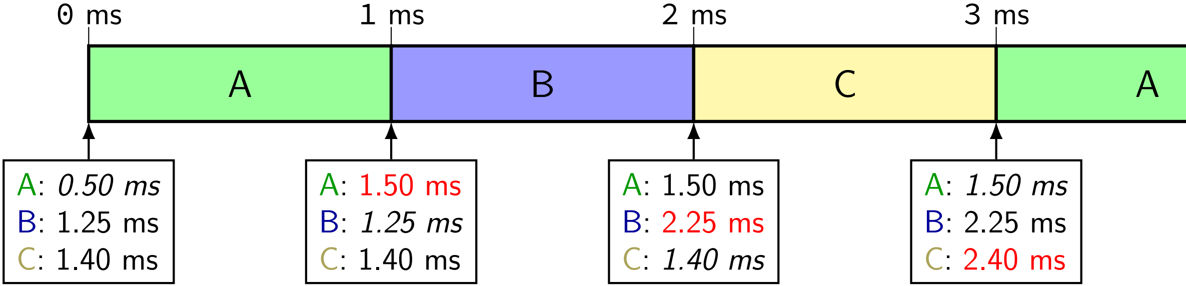


at each time:

update current thread's time

run thread with lowest total time

virtual time, always ready, 1 ms quantum



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)

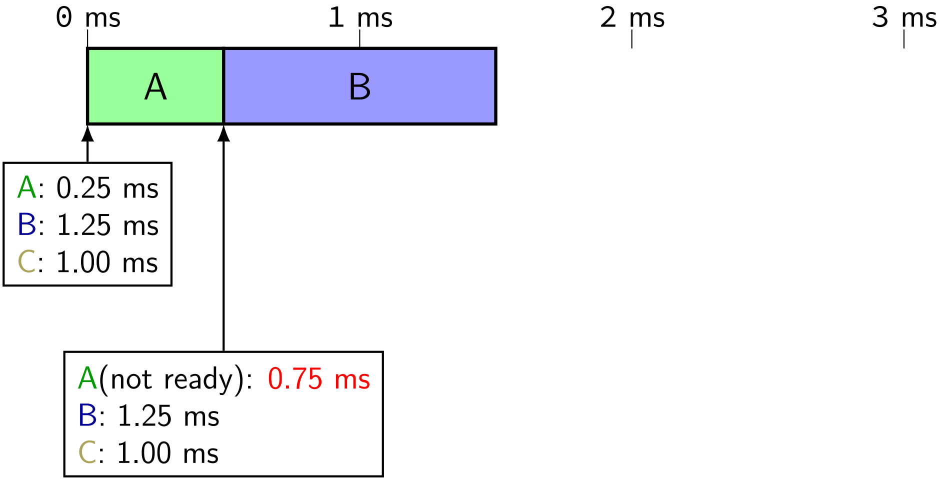
not runnable briefly? still get your share of CPU

- (catch up from prior virtual time)

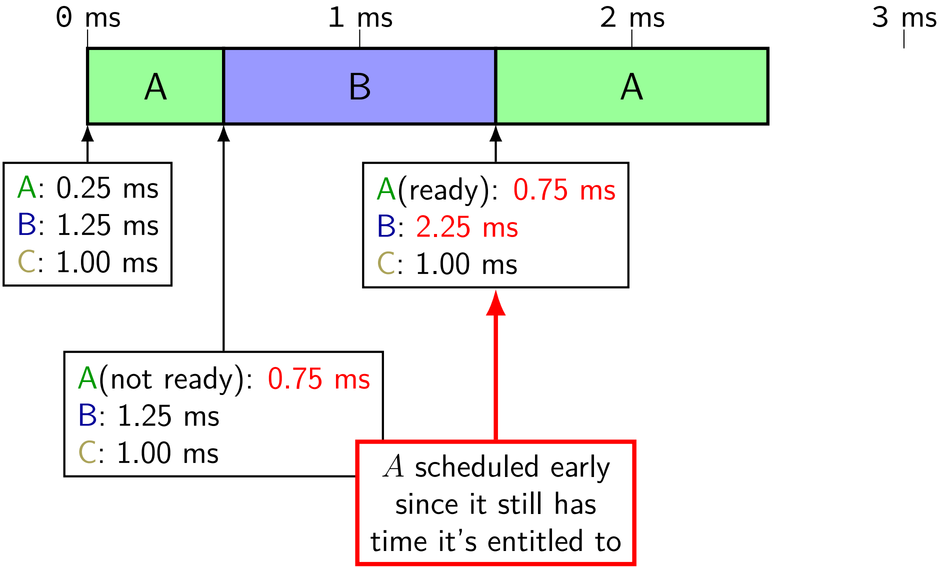
not runnable for a while? get bounded advantage

A doesn't use whole time...

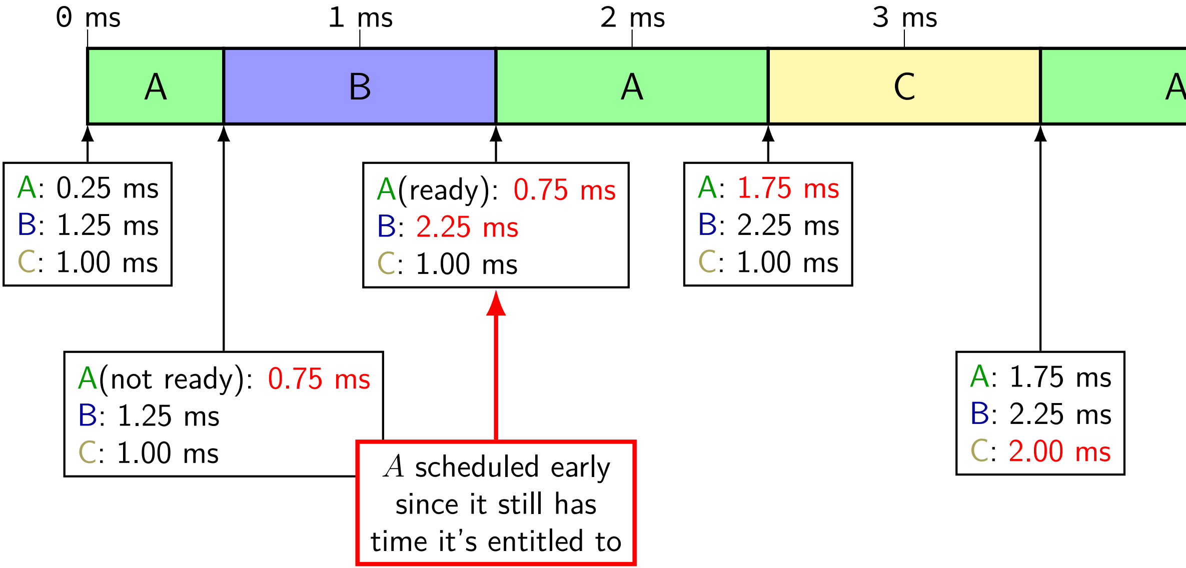
A doesn't use whole time...



A doesn't use whole time...



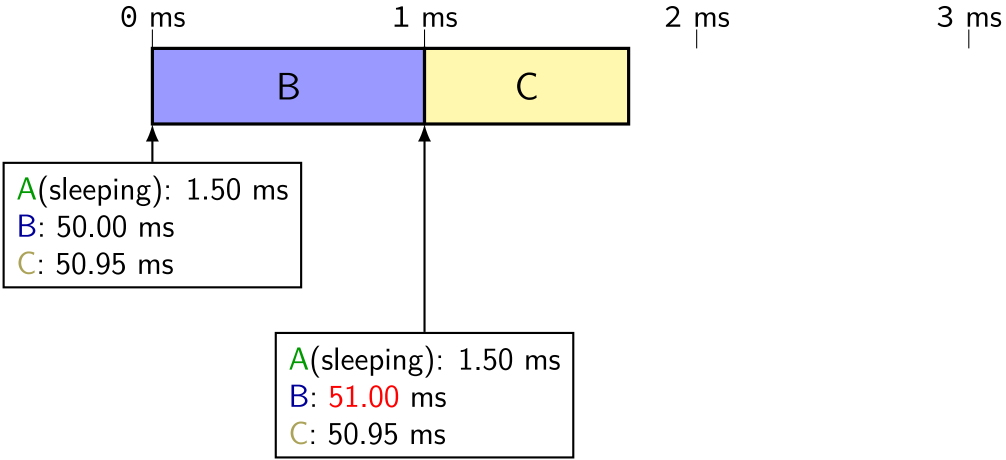
A doesn't use whole time...



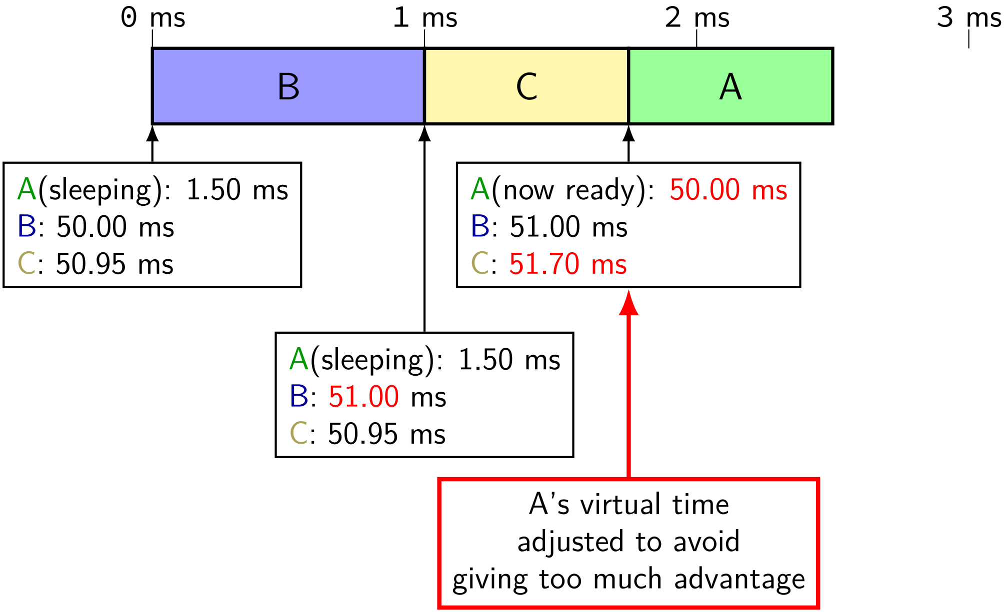
A's long sleep...

0 ms 1 ms 2 ms 3 ms

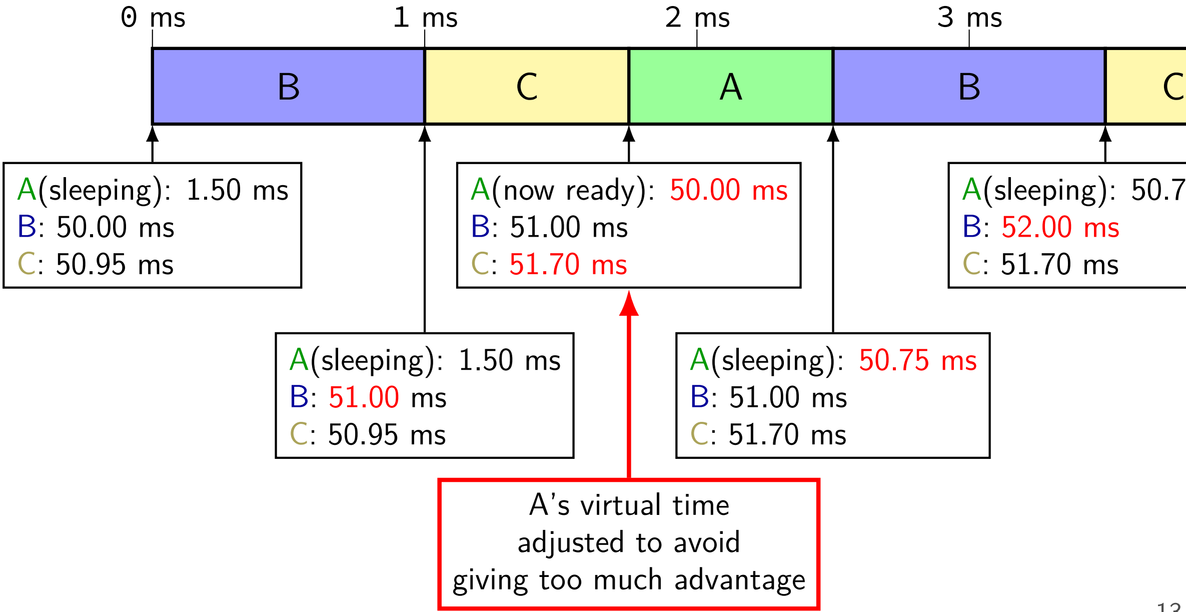
A's long sleep...



A's long sleep...



A's long sleep...



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

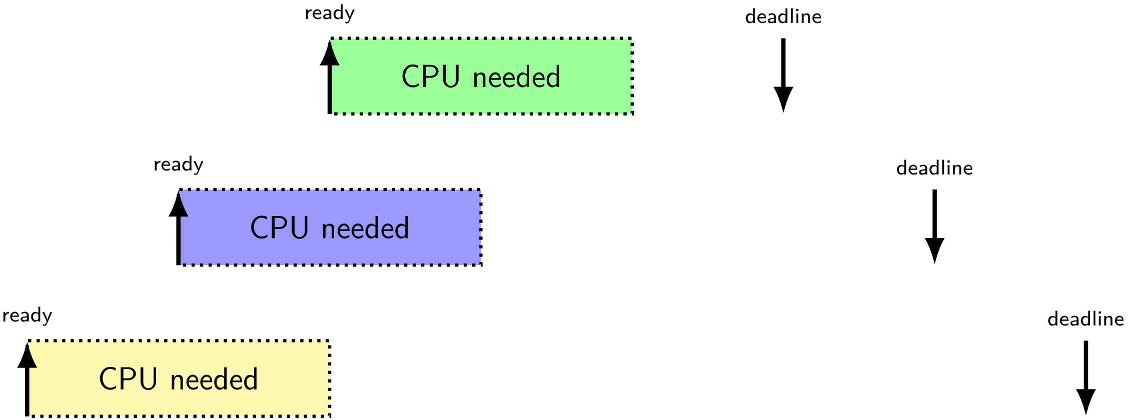
- best possible (by some metrics) given some work

alternate model: need gaurnetees

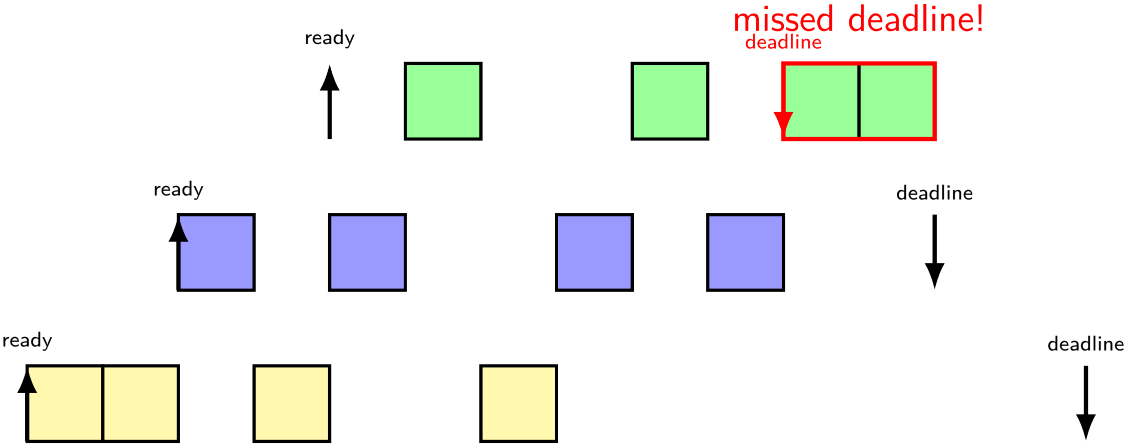
deadlines imposed by real-world

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

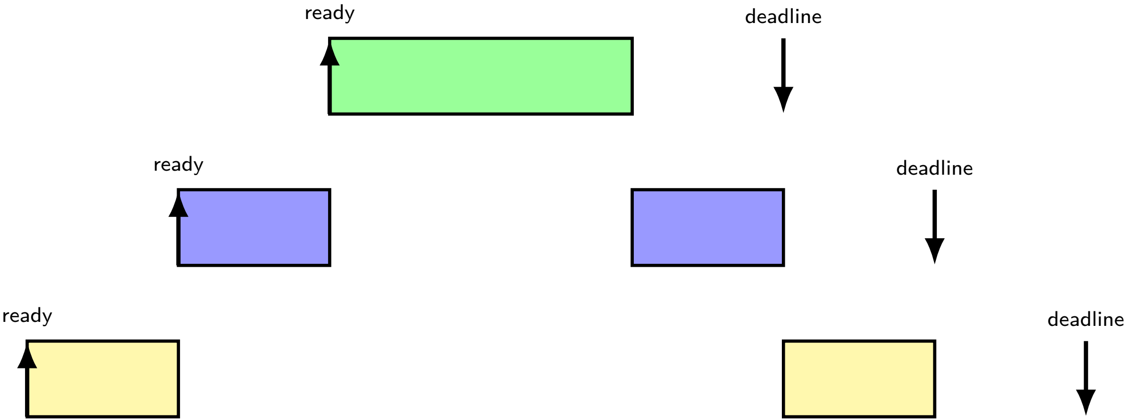
real time example: CPU + deadlines



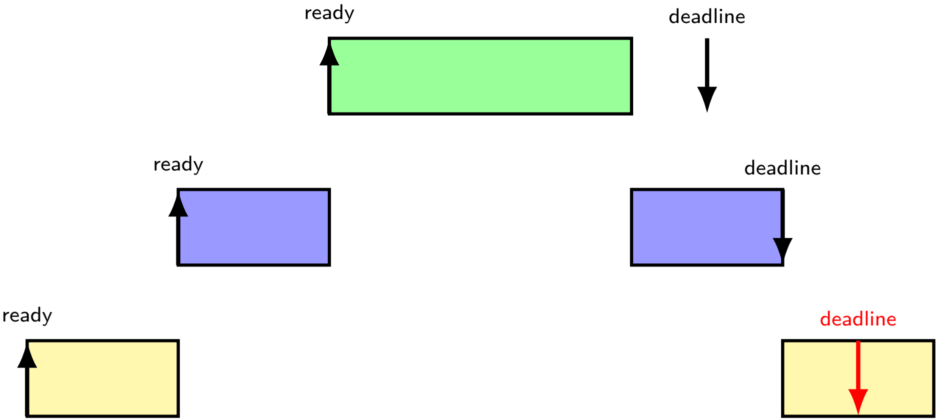
example with RR



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)
};
```

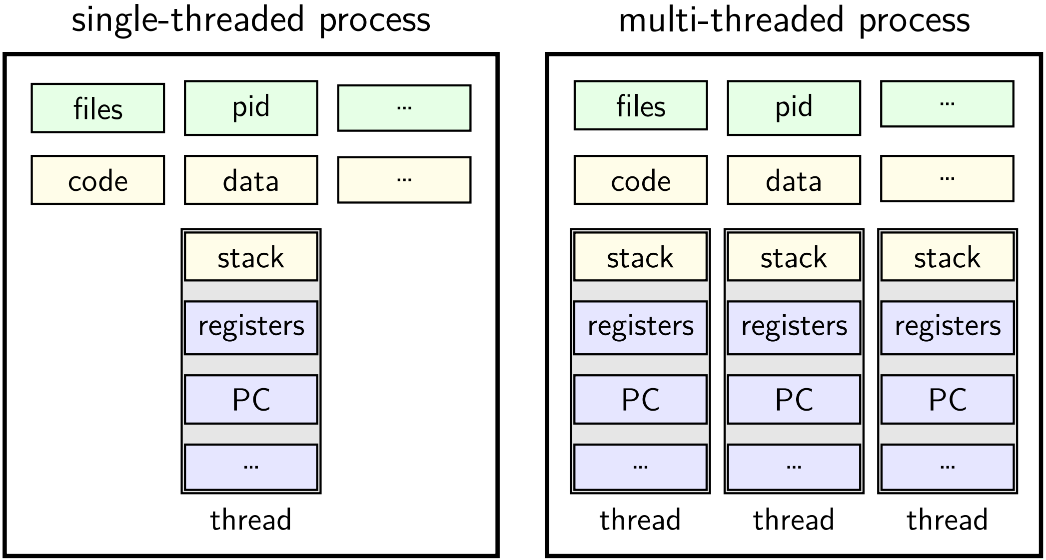
xv6 processes versus threads

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

- e.g. `clone(CLONE_FILES, ...)` — new process **sharing** open files
- e.g. `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

- e.g. `clone(CLONE_FILES, ...)` — new process **sharing** open files
- e.g. `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 *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

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

```
voi
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

```
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 */
}
```

thread attributes (extra settings) and function argument

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;
}
```

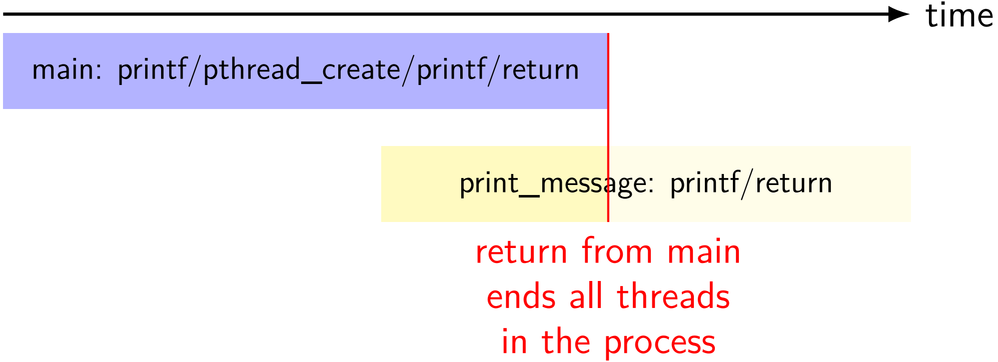
My machine: outputs `In the thread` about 4% of the time.

What happened?

a race

returning from main exits the entire process (all threads)

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



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

passing thread IDs (1)

```
DataType items[1000];
void *thread_function(void *argument) {
    int thread_id = (int) argument;
    int start = 500 * thread_id;
    int end = start + 500;
    for (int i = start; i < end; ++i) {
        DoSomethingWith(items[i]);
    }
    ...
}
void run_threads() {
    vector<pthread_t> threads(2);
    for (int i = 0; i < 2; ++i) {
        pthread_create(&threads[i], NULL,
            thread_function, (void*) i);
    }
}
```

passing thread IDs (1)

```
DataType items[1000];
void *thread_function(void *argument) {
    int thread_id = (int) argument;
    int start = 500 * thread_id;
    int end = start + 500;
    for (int i = start; i < end; ++i) {
        DoSomethingWith(items[i]);
    }
    ...
}
void run_threads() {
    vector<pthread_t> threads(2);
    for (int i = 0; i < 2; ++i) {
        pthread_create(&threads[i], NULL,
            thread_function, (void*) i);
    }
}
```

passing thread IDs (2)

```
DataType items[1000];
int num_threads;
void *thread_function(void *argument) {
    int thread_id = (int) argument;
    int start = thread_id * (1000 / num_threads);
    int end = start + (1000 / num_threads);
    if (thread_id == num_threads - 1) end = 1000;
    for (int i = start; i < end; ++i) {
        DoSomethingWith(items[i]);
    }
    ...
}
void run_threads() {
    vector<pthread_t> threads(num_threads);
    for (int i = 0; i < num_threads; ++i) {
        pthread_create(&threads[i], NULL,
            thread_function, (void*) i);
    }
    ...
```

passing thread IDs (2)

```
DataType items[1000];
int num_threads;
void *thread_function(void *argument) {
    int thread_id = (int) argument;
    int start = thread_id * (1000 / num_threads);
    int end = start + (1000 / num_threads);
    if (thread_id == num_threads − 1) end = 1000;
    for (int i = start; i < end; ++i) {
        DoSomethingWith(items[i]);
    }
    ...
}
void run_threads() {
    vector<pthread_t> threads(num_threads);
    for (int i = 0; i < num_threads; ++i) {
        pthread_create(&threads[i], NULL,
            thread_function, (void*) i);
    }
    ...
```

passing data structures

```
class ThreadInfo {
public:
    ...
};

void *thread_function(void *argument) {
    ThreadInfo *info = (ThreadInfo *) argument;
    ...
    delete info;
}

void run_threads(int N) {
    vector<pthread_t> threads(num_threads);
    for (int i = 0; i < num_threads; ++i) {
        pthread_create(&threads[i], NULL,
            thread_function, (void *) new ThreadInfo(...));
    }
    ...
}
```

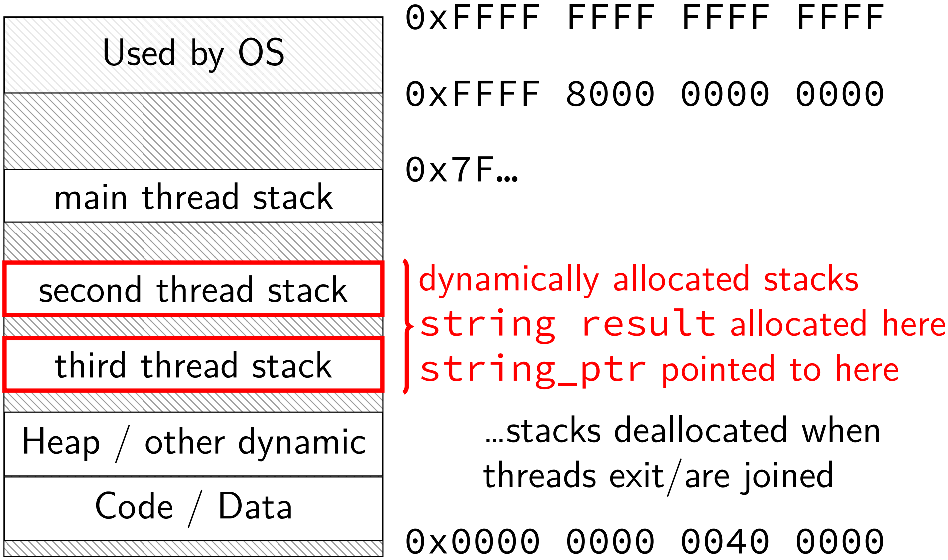
passing data structures

```
class ThreadInfo {
public:
    ...
};

void *thread_function(void *argument) {
    ThreadInfo *info = (ThreadInfo *) argument;
    ...
    delete info;
}

void run_threads(int N) {
    vector<pthread_t> threads(num_threads);
    for (int i = 0; i < num_threads; ++i) {
        pthread_create(&threads[i], NULL,
            thread_function, (void *) new ThreadInfo(...));
    }
    ...
}
```

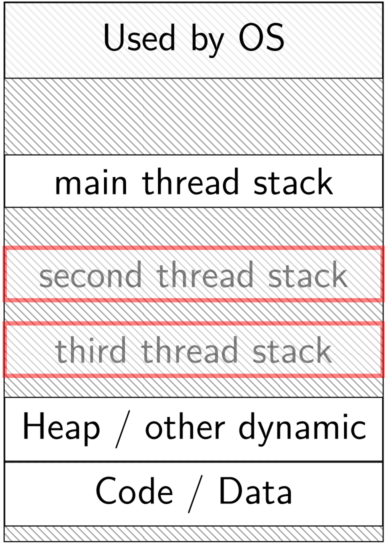
what's wrong with this?

```
/* omitted: headers, using statements */
void *create_string(void *ignored_argument) {
  string result;
  result = ComputeString();
  return &result;
}
int main() {
  pthread_t the_thread;
  pthread_create(&the_thread, NULL, get_string, NULL);
  string *string_ptr;
  pthread_join(the_thread, &string_ptr);
  cout << "string is " << *string_ptr;
}
```

program memory



program memory



0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

0x7F...

dynamically allocated stacks

`string result` allocated here

`string_ptr` pointed to here

...stacks deallocated when threads exit/are joined

0x0000 0000 0040 0000

thread resources

to create a thread, allocate:

new stack (how big???)

thread control block

pthreads: by default need to `join` thread to deallocate everything

thread kept around to allow collecting return value

pthread_detach

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_create(&show_progress_thread, NULL, show_progress, NULL)
    pthread_detach(show_progress_thread);
}
int main() {
    spawn_show_progress_thread();
    do_other_stuff();
    ...
}
```

starting threads detached

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_attr_t attrs;
    pthread_attr_init(&attrs);
    pthread_attr_setdetachstate(&attrs, PTHREAD_CREATE_DETACHED);
    pthread_create(&show_progress_thread, attrs, show_progress, NULL
    pthread_attr_destroy(&attrs);
}
```

setting stack sizes

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_attr_t attrs;
    pthread_attr_init(&attrs);
    pthread_attr_setstacksize(&attrs, 32 * 1024 /* bytes */);
    pthread_create(&show_progress_thread, NULL, show_progress,
}
```

a note on error checking

from pthread_create manpage:

```
ERRORS
       EAGAIN Insufficient  resources  to  create  another thread, or a system-imposed limit on the number of
              threads was encountered.  The latter case may occur in two ways: the RLIMIT_NPROC soft resource
              limit  (set  via  setrlimit(2)),  which  limits  the  number of process for a real user ID, was
              reached; or the kernel's system-wide limit on the number of threads,  /proc/sys/kernel/threads-
              max, was reached.

       EINVAL Invalid settings in attr.

       EPERM  No permission to set the scheduling policy and parameters specified in attr.
```

special constants for *return value*

same pattern for many other pthreads functions

will often omit error checking in slides for brevity

error checking pthread_create

```
int error = pthread_create(...);
if (error != 0) {
    /* print some error message */
}
```

the correctness problem

schedulers introduce non-determinism

- scheduler might run threads in any order
- scheduler can switch threads at any time

worse with threads on multiple cores

- cores not precisely synchronized (stalling for caches, etc., etc.)
- different cores happen in different order each time

makes reliable testing very difficult

solution: correctness by design

example application: ATM server

commands: withdraw, deposit

one correctness goal: don't lose money

ATM server

(pseudocode)

```
ServerLoop() {
    while (true) {
        ReceiveRequest(&operation, &accountNumber, &amount);
        if (operation == DEPOSIT) {
            Deposit(accountNumber, amount);
        } else ...
    }
}
Deposit(accountNumber, amount) {
    account = GetAccount(accountId);
    account->balance += amount;
    StoreAccount(account);
}
```

a threaded server?

```
Deposit(accountNumber, amount) {
    account = GetAccount(accountId);
    account->balance += amount;
    StoreAccount(account);
}
```

maybe Get/StoreAccount can be slow?

- read/write disk sometimes? contact another server sometimes?

maybe lots of requests to process?

- maybe real logic has more checks than Deposit()
- ...

all reasons to handle multiple requests at once

→ many threads all running the server loop

multiple threads

```
main() {
    for (int i = 0; i < NumberOfThreads; ++i) {
        pthread_create(&server_loop_threads[i], NULL,
                       ServerLoop, NULL);
    }
    ...
}

ServerLoop() {
    while (true) {
        ReceiveRequest(&operation, &accountNumber, &amount);
        if (operation == DEPOSIT) {
            Deposit(accountNumber, amount);
        } else ...
    }
}
```

a side note

why am I spending time justifying this?

multiple threads for something like this make things much trickier

we'll be learning why...

the lost write

`account->balance += amount;` (in two threads, same account)

Thread A	Thread B
`mov account->balance, %rax`	
`add amount, %rax`	
context switch	*context switch*
	`mov account->balance, %rax`
	`add amount, %rax`
context switch	*context switch*
`mov %rax, account->balance`	
context switch	*context switch*
	`mov %rax, account->balance`

the lost write

`account->balance += amount;` (in two threads, same account)

Thread A	Thread B
`mov account->balance, %rax`	
`add amount, %rax`	
— context switch —	— context switch —
	`mov account->balance, %rax`
	`add amount, %rax`
— context switch —	— context switch —
`mov %rax, account->balance`	
— context switch —	— context switch —
	`mov %rax, account->balance`

lost write to balance

"winner" of the race

the lost write

`account->balance += amount;` (in two threads, same account)

Thread A		Thread B
`mov account->balance, %rax` `add amount, %rax`		
	context switch	
		`mov account->balance, %rax` `add amount, %rax`
	context switch	
`mov %rax, account->balance`		
	context switch	
		`mov %rax, account->balance`

lost write to balance

"winner" of the race

lost track of thread A's money

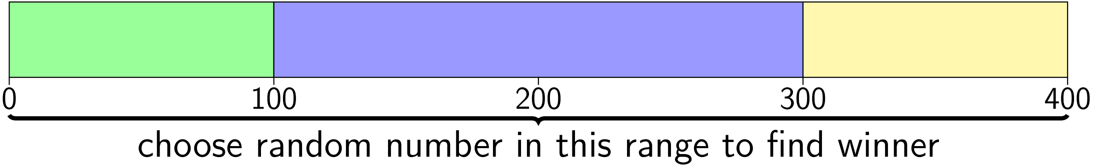
backup slides

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:



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

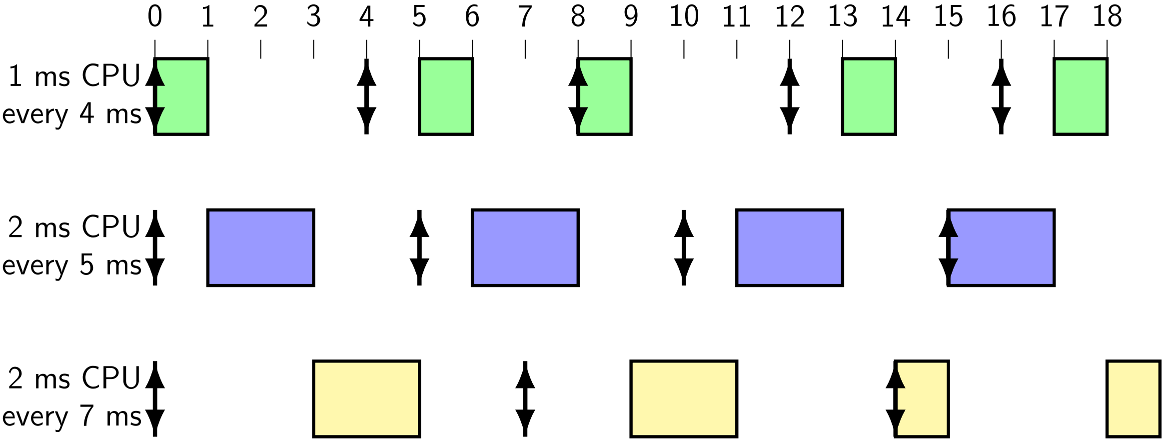
process B: 9 tickets, always runnable

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%

periodic tasks and deadlines



admission control

filter gaurentees: don't make promises you can't keep

theorem (Liu and Layland, 1973):

given periodic tasks (released *after each deadline*), deadlines D_i and computation times C_i, earliest deadline first will meet all deadlines if:
$$\sum_{i=1}^{n} (C_i/D_i) \leq 1$$

one idea: use this to accept/reject tasks

xv6 interrupt disabling: detail (3)

```
pushcli(void)
{
  int eflags;
  eflags = readeflags();
  cli();
  if (mycpu()->ncli == 0)
    mycpu()->intena = eflags & FL_IF;
  mycpu()->ncli += 1;
}

popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli - interruptible");
  if(--mycpu()->ncli < 0)
    panic("popcli");
  if(mycpu()->ncli == 0 && mycpu()->intena)
    sti();
}
```

xv6 interrupt disabling: detail (3)

```
pushcli(void)
{
  int eflags;
  eflags = readeflags();
  cli();
  if (mycpu()->ncli == 0)
    mycpu()->intena = eflags & FL_IF;
  mycpu()->ncli += 1;
}

popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli␣-␣interruptible");
  if(--mycpu()->ncli < 0)
    panic("popcli");
  if(mycpu()->ncli == 0 && mycpu()->intena)
    sti();
}
```

mycpu() — per-core information

xv6 interrupt disabling: detail (3)

```
pushcli(void)
{
  int eflags;
  eflags = readeflags();
  cli();
  if (mycpu()->ncli == 0)
    mycpu()->intena = eflags & FL_IF;
  mycpu()->ncli += 1;
}

popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli␣-␣interruptible");
  if(--mycpu()->ncli < 0)
    panic("popcli");
  if(mycpu()->ncli == 0 && mycpu()->intena)
    sti();
}
```

`intena` — were interrupts enabled before first pushcli()?

xv6 interrupt disabling: detail (3)

```
pushcli(void)
{
  int eflags;
  eflags = readeflags();
  cli();
  if (mycpu()->ncli == 0)
    mycpu()->intena = eflags & FL_IF;
  mycpu()->ncli += 1;
}

popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli␣-␣interruptible");
  if(--mycpu()->ncli < 0)
    panic("popcli");
  if(mycpu()->ncli == 0 && mycpu()->intena)
    sti();
}
```

ncli — # calls to pushcli - # calls to popcli

intended usage: each pushcli has matching popcli

xv6 interrupt disabling: detail (3)

```
pushcli(void)
{
  int eflags;
  eflags = readeflags();
  cli();
  if (mycpu()->ncli == 0)
    mycpu()->intena = eflags & FL_IF;
  mycpu()->ncli += 1;
}

popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli - interruptible");
  if(--mycpu()->ncli < 0)
    panic("popcli");
  if(mycpu()->ncli == 0 && mycpu()->intena)
    sti();
}
```

pushcli — always disable interrupts

xv6 interrupt disabling: detail (3)

```
pushcli(void)
{
  int eflags;
  eflags = readeflags();
  cli();
  if (mycpu()->ncli == 0)
    mycpu()->intena = eflags & FL_IF;
  mycpu()->ncli += 1;
}

popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli - interruptible");
  if(--mycpu()->ncli < 0)
    panic("popcli");
  if(mycpu()->ncli == 0 && mycpu()->intena)
    sti();
}
```

popcli — renable interrupts if last popcli
(and interrupts were enabled before)
(each pushcli had a matching popcli call)

Java synchronized primitive

```
Object MilkLock = new Object();

/* lock implicity acquired/released on
   entering/leaving this block */
synchronized (MilkLock) {
    if (no milk) {
        buy milk
    }
}
```

C++11 mutexes

```
#include <mutex>

std::mutex MilkLock;
{
    std::lock_guard nameDoesNotMatter(MilkLock);
    /* nameDoesNotMatter's constructor acquires lock */
    if (no milk) {
        buy milk
    }
    /* nameDoesNotMatter's destructor called automatically
       and releases lock
     */
}
```