Scheduling 1

Changelog

Changes made in this version not seen in first lecture:

- 10 September: RR varying quantum examples: fix calculation of response/wait time on Q=2
- 10 September: add priority scheduling and preemption slide
- 10 September: backup slides with pipe() exercises — see end of slide deck

Unix API summary

spawn and wait for program: `fork` (copy), then

- in child: setup, then `execv`, etc. (replace copy)
- in parent: `waitpid`

files: open, read and/or write, close

- one interface for regular files, pipes, network, devices, …

file descriptors are indices into per-process array

- index 0, 1, 2 = stdin, stdout, stderr
- `dup2` — assign one index to another
- `close` — deallocate index

redirection/pipelines

- open() or pipe() to create new file descriptors
- dup2 in child to assign file descriptor to index 0, 1

xv6: process table

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC]
} ptable;
```

fixed size array of all processes

lock to keep more than one thing from accessing it at once

- rule: don't change a process's state (RUNNING, etc.) without 'acquiring' lock

xv6: allocating a struct proc

```
acquire(&ptable.lock);

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  if(p->state == UNUSED)
    goto found;

release(&ptable.lock);
```

just search for PCB with “UNUSED” state

not found? fork fails

if found — allocate memory, etc.

xv6: creating the first process

struct proc with initial kernel stack
setup to return from swtch, then from exception

```
// Set up first user process.
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();

  initproc = p;
  ...
  inituvm(p->pgdir, _binary_initcode_start,
            (int)_binary_initcode_size);
  ...
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S
  ...
  p->state = RUNNABLE;
```

xv6: creating the first process

load into user memory
hard-coded "initial program"
calls execv() of `/init`

```
// Set up first user process.
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();

  initproc = p;
  ...
  inituvm(p->pgdir, _binary_initcode_start,
            (int)_binary_initcode_size);
  ...
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S
  ...
  p->state = RUNNABLE;
```

xv6: creating the first process

modify user registers
to start at address 0

```
// Set up first user process.
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();

  initproc = p;
  ...
  inituvm(p->pgdir, _binary_initcode_start,
            (int)_binary_initcode_size);
  ...
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S
  ...
  p->state = RUNNABLE;
```

xv6: creating the first process

set initial stack pointer

```
// Set up first user process.
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();

  initproc = p;
  ...
  inituvm(p->pgdir, _binary_initcode_start,
              (int)_binary_initcode_size);
  ...
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S
  ...
  p->state = RUNNABLE;
```

xv6: creating the first process

set process as runnable

```
// Set up first user process.
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();

  initproc = p;
  ...
  inituvm(p->pgdir, _binary_initcode_start,
            (int)_binary_initcode_size);
  ...
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S
  ...
  p->state = RUNNABLE;
```

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)
};
```

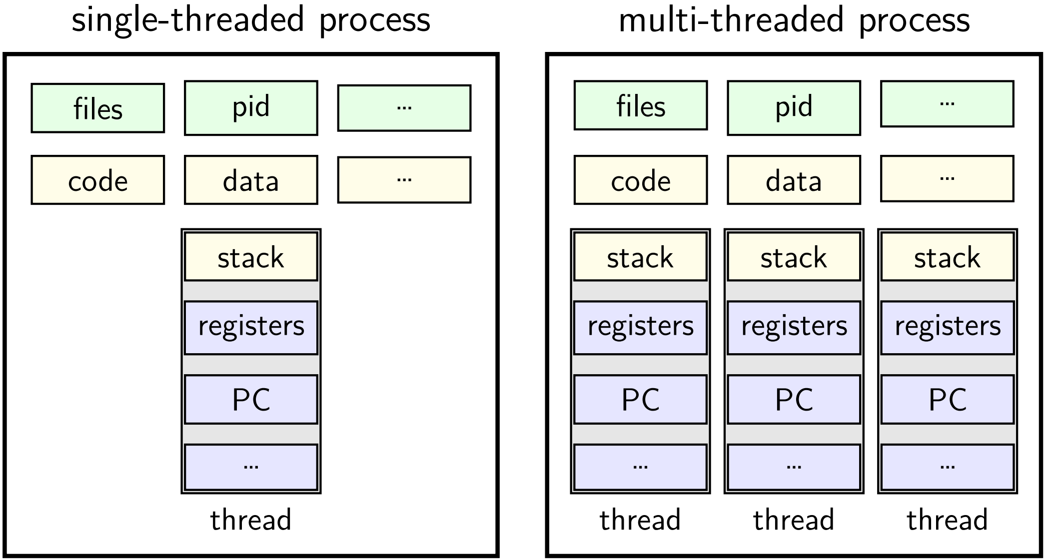
xv6 processes versus threads

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



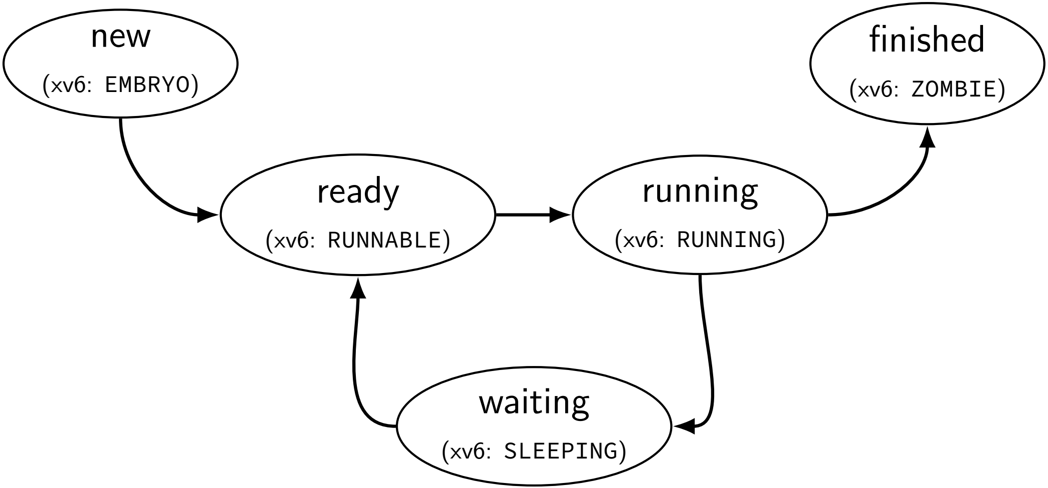
single-threaded process

files
pid
…
code
data
…
stack
registers
PC
…
thread

multi-threaded process

files
pid
…
code
data
…
stack
registers
PC
…
thread
stack
registers
PC
…
thread
stack
registers
PC
…
thread

thread states



new

(xv6: `EMBRYO`)

ready

(xv6: `RUNNABLE`)

running

(xv6: `RUNNING`)

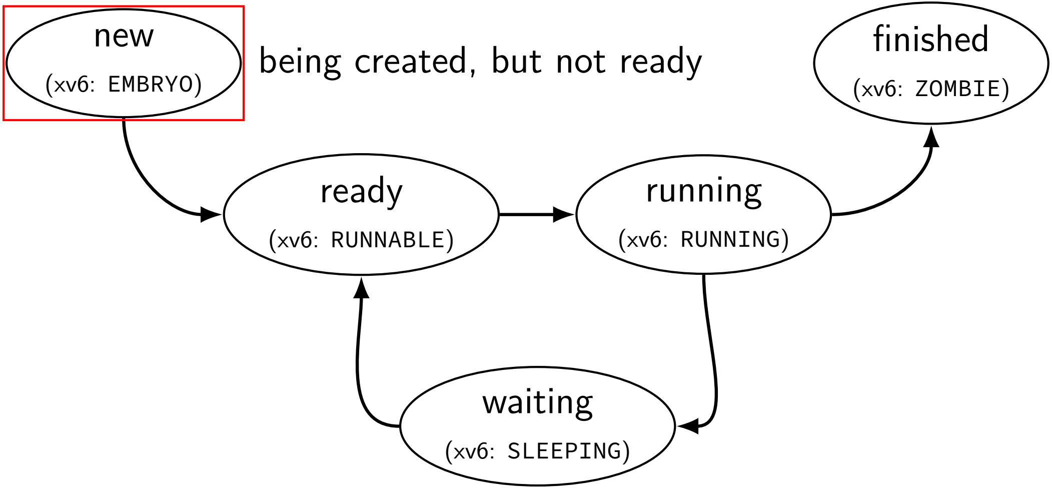
finished

(xv6: `ZOMBIE`)

waiting

(xv6: `SLEEPING`)

thread states



new

(xv6: `EMBRYO`)

being created, but not ready

finished

(xv6: `ZOMBIE`)

ready

(xv6: `RUNNABLE`)

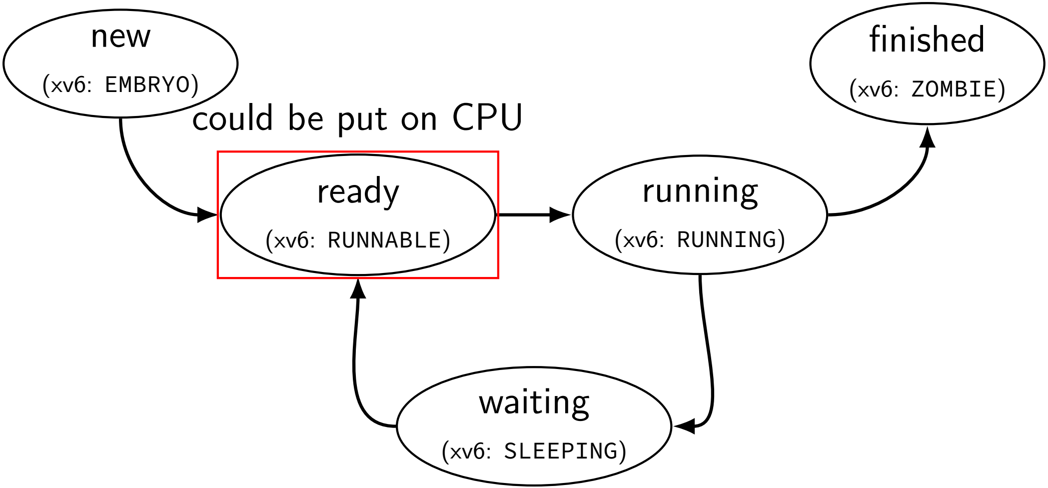
running

(xv6: `RUNNING`)

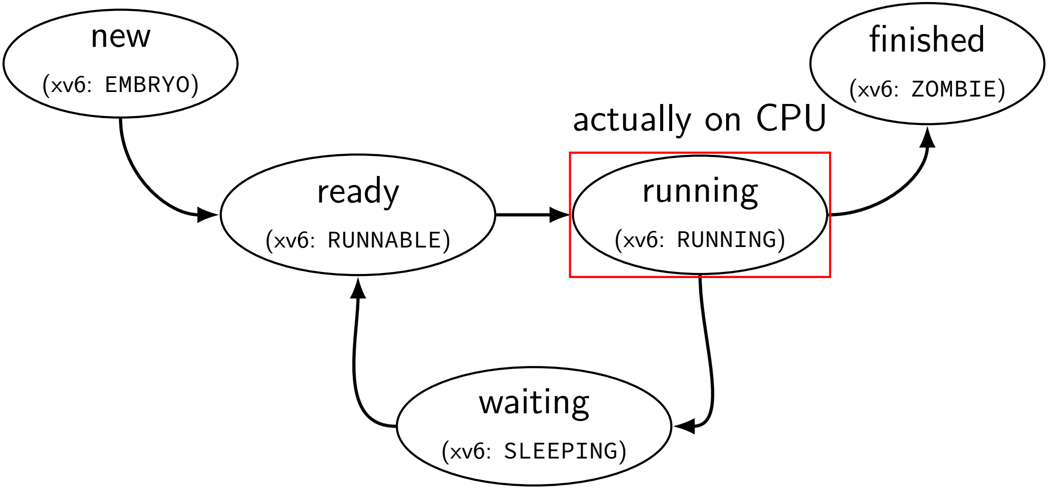
waiting

(xv6: `SLEEPING`)

thread states



thread states


new

(xv6: `EMBRYO`)

ready

(xv6: `RUNNABLE`)

actually on CPU

running

(xv6: `RUNNING`)

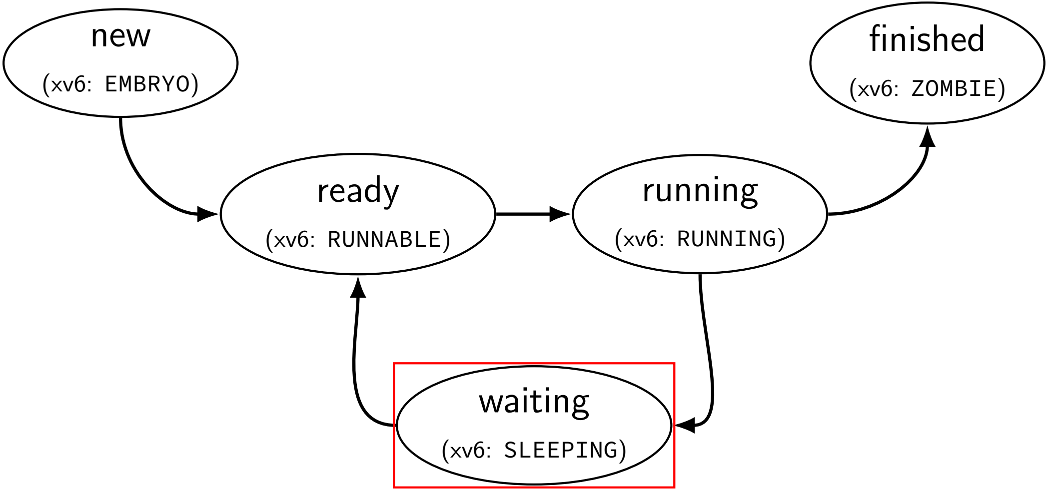
finished

(xv6: `ZOMBIE`)

waiting

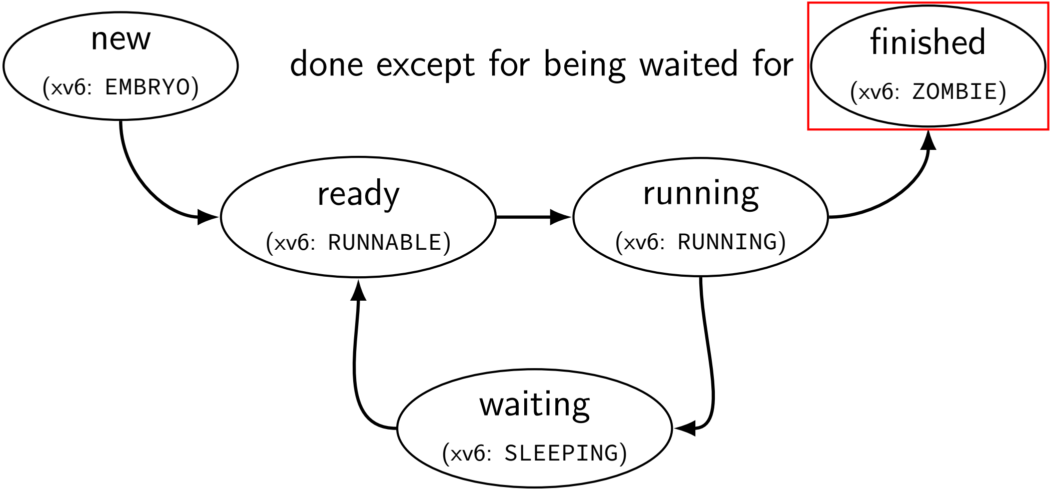
(xv6: `SLEEPING`)

thread states

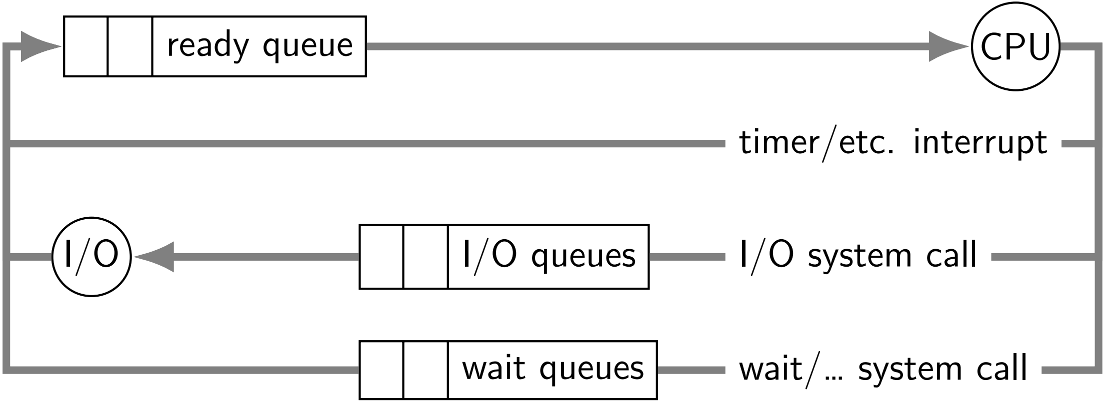


need external event to happen

thread states



alternative view: queues



ready queue

CPU

timer/etc. interrupt

I/O

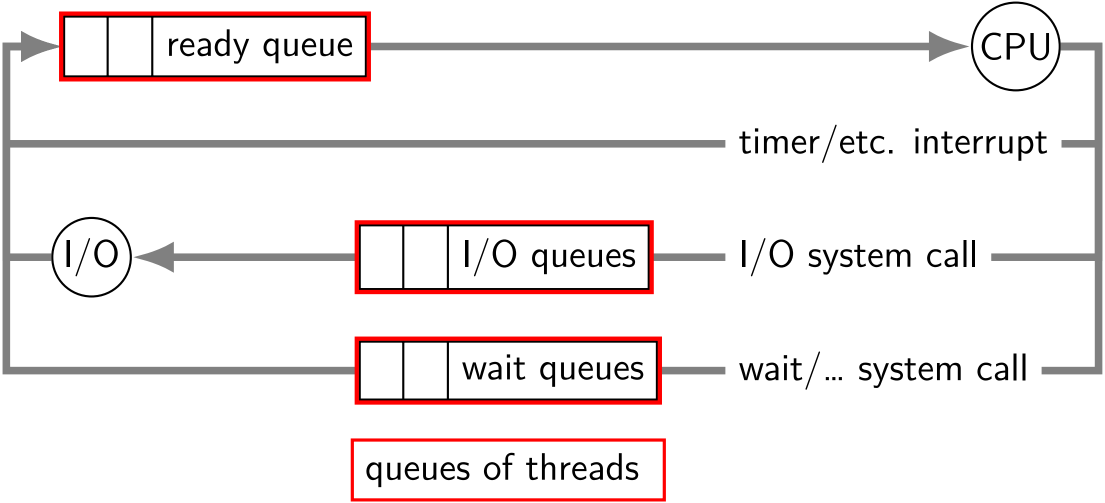
I/O queues

I/O system call

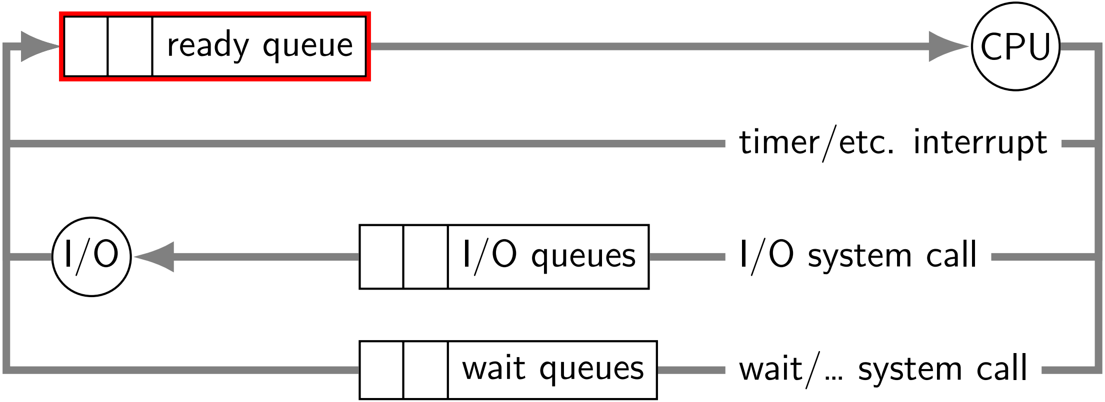
wait queues

wait/... system call

alternative view: queues



alternative view: queues



ready queue or run queue
list of running processes
question: what to take off queue first when CPU is free?

on queues in xv6

xv6 doesn't represent queues explicitly

- no queue class/struct

ready queue: process list ignoring non-RUNNABLE entries

I/O queues: process list where SLEEPING, `chan` = I/O device

real OSs: typically separate list of processes

- maybe sorted?

scheduling

scheduling = removing process/thread to remove from queue

mostly for the ready queue (pre-CPU)

- remove a process and start running it

example other scheduling problems

batch job scheduling

e.g. what to run on my supercomputer?

jobs that run for a long time (tens of seconds to days)

can't easily 'context switch' (save job to disk??)

I/O scheduling

what order to read/write things to/from network, hard disk, etc.

this lecture

main target: CPU scheduling

...on a system where programs do a lot of I/O

...and other programs use the CPU when they do

...with only a single CPU

many ideas port to other scheduling problems
- especially simpler/less specialized policies

scheduling policy

scheduling policy = what to remove from queue

the xv6 scheduler (1)

```
void scheduler(void)
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      ... /* switch to process */
    }
    release(&ptable.lock);
  }
}
```

the xv6 scheduler (1)

```
void scheduler(void)
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      ... /* switch to process */
    }
    release(&ptable.lock);
  }
}
```

infinite loop
every iteration: switch to a thread
thread will switch back to us

the xv6 scheduler (1)

enable interrupts (`sti` is the x86 instruction)
…but not acquiring the process table lock
disables interrupts

```
void scheduler(void)
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      ... /* switch to process */
    }
    release(&ptable.lock);
  }
}
```

the xv6 scheduler (1)

```
void scheduler(void)
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      ... /* switch to process */
    }
    release(&ptable.lock);
  }
}
```

make sure we're the only one accessing the list of processes

also make sure no one runs scheduler while we're switching to another process

(more on this idea later)

the xv6 scheduler (1)

iterate through all runnable processes in the order they're stored in a table

```
void scheduler(void)
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      ... /* switch to process */
    }
    release(&ptable.lock);
  }
}
```

the xv6 scheduler (1)

```
void scheduler(void)
  struct proc *p;
  struct cpu *c = mycp
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      ... /* switch to process */
    }
    release(&ptable.lock);
  }
}
```

switch to whatever runnable process we find
when it's done (e.g. timer interrupt)
it switches back, then next loop iteration happens

the xv6 scheduler: the actual switch

```
/* in scheduler(): */
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
```

the xv6 scheduler: the actual switch

prepare: change address space, change process state

```
/* in scheduler(
        // Switch
        // to release ptable.lock and then reacquire it
        // before jumping back to us.
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;

        swtch(&(c->scheduler), p->context);
        switchkvm();

        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;
```

the xv6 scheduler: the actual switch

```
/* in scheduler()
        // Switch to
        // to releas
        // before jumping back to us.
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;

        swtch(&(c->scheduler), p->context);
        switchkvm();

        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;
```

switch to kernel thread of process

that thread responsible for going back to user mode

the xv6 scheduler: the actual switch

```
/* in schedu
      // Swi
      // to
      // bef
      c->pro
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
```

after we've run the process until it's done, we end up here

…so, change address space back away from user process

the xv6 scheduler: the actual switch

```
/* in scheduler(): */
      // Switch to chosen pr
      // to release ptable.l
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
```

track what process is being run
so we can look it up in interrupt handler

the xv6 scheduler: on process start

```
void forkret() {
  /* scheduler switches to here after new process starts */
  ...
  release(&ptable.lock);
  ...
}
```

the xv6 scheduler: on process start

```
void forkret() {
  /* scheduler switches to here after new process starts */
  ...
  release(&ptable.lock);
  ...
}
```

scheduler switched with process table locked
need to unlock before running user code
(so other cores, interrupts can use table or run scheduler)

the xv6 scheduler: going from/to scheduler

```
/* function to invoke scheduler;
   used by the timer interrupt or yield() syscall */
void yield() {
  acquire(&ptable.lock);
  myproc()->state = RUNNABLE;
  sched();  // switches to scheduler thread
  release(&ptable.lock);
}
```

the xv6 scheduler: going from/to scheduler

```
/* function to invoke scheduler;
   used by the timer interrupt or yield() syscall */
void yield() {
  acquire(&ptable.lock);
  myproc()->state = RUNNABLE;
  sched();  // switches to scheduler thread
  release(&ptable.lock);
}
```

process table was locked
(to keep other cores/processes from using it)
unlock it before running user code
otherwise: timer interrupt won't work

the xv6 scheduler: going from/to scheduler

```
/* function to invoke scheduler;
   used by the timer interrupt or yield() syscall */
void yield() {
  acquire(&ptable.lock);
  myproc()->state = RUNNABLE;
  sched();  // switches to scheduler thread
  release(&ptable.lock);
}
```

yield: function to call scheduler
called by timer interrupt handler

the xv6 scheduler: going from/to scheduler

```
/* function to invoke scheduler;
   used by the timer interrupt or yield() syscall */
void yield() {
  acquire(&ptable.lock);
  myproc()->state = RUNNABLE;
  sched();  // switches to scheduler thread
  release(&ptable.lock);
}
```

make sure we're the only one accessing the process list
before changing our process's state
and before running scheduler loop

the xv6 scheduler: going from/to scheduler

```
/* function to invoke scheduler;
   used by the timer interrupt or yield() syscall */
void yield() {
  acquire(&ptable.lock);
  myproc()->state = RUNNABLE;
  sched();  // switches to scheduler thread
  release(&ptable.lock);
}
```

set us as RUNNABLE (was RUNNING)
then switch to infinite loop in scheduler

the xv6 scheduler: entering/leaving for sleep

```
void sleep(void *chan, struct spinlock *lk) {
  ...
    acquire(&ptable.lock);
  ...
  p->chan = chan;
  p->state = SLEEPING;

  sched();

  ...
    release(&ptable.lock);
  ...
```

the xv6 scheduler: entering/leaving for sleep

```
void sleep(void *chan, struct s
  ...
    acquire(&ptable.lock);
  ...
  p->chan = chan;
  p->state = SLEEPING;

  sched();

  ...
    release(&ptable.lock);
  ...
```

get exclusive access to process table before changing our state to sleeping and before running scheduler loop

the xv6 scheduler: entering/leaving for sleep

```
void sleep(void *chan, struct
  ...
    acquire(&ptable.lock);
  ...
  p->chan = chan;
  p->state = SLEEPING;

  sched();

  ...
    release(&ptable.lock);
  ...
```

set us as SLEEPING (was RUNNING)
use "chan" to remember why
(so others process can wake us up)

the xv6 scheduler: entering/leaving for sleep

```
void sleep(void *chan, str
  ...
    acquire(&ptable.lock);
  ...
  p->chan = chan;
  p->state = SLEEPING;

  sched();


  ...
    release(&ptable.lock);
  ...
```

…and switch to the scheduler infinite loop

the scheduling policy problem

what RUNNABLE program should we run?

xv6 answer: whatever's next in list

best answer?

- well, what do you care about?

some simplifying assumptions

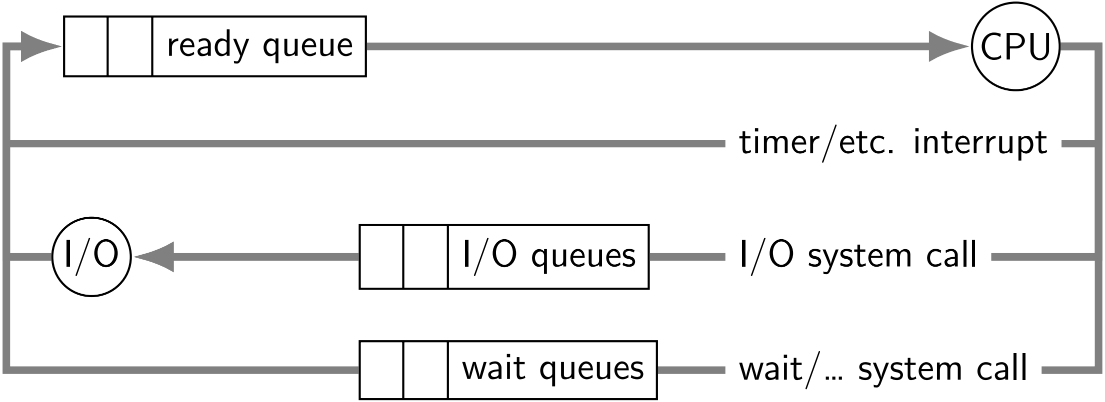
welcome to 1970:

one program per user

one thread per program

programs are independent

recall: scheduling queues



ready queue

CPU

timer/etc. interrupt

I/O

I/O queues

I/O system call

wait queues

wait/... system call

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)

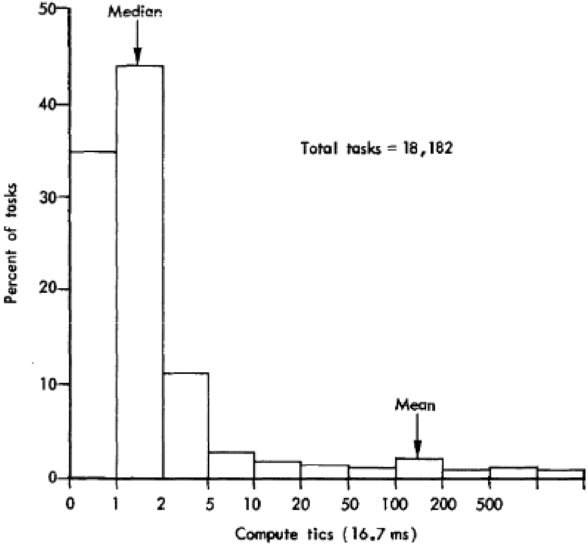
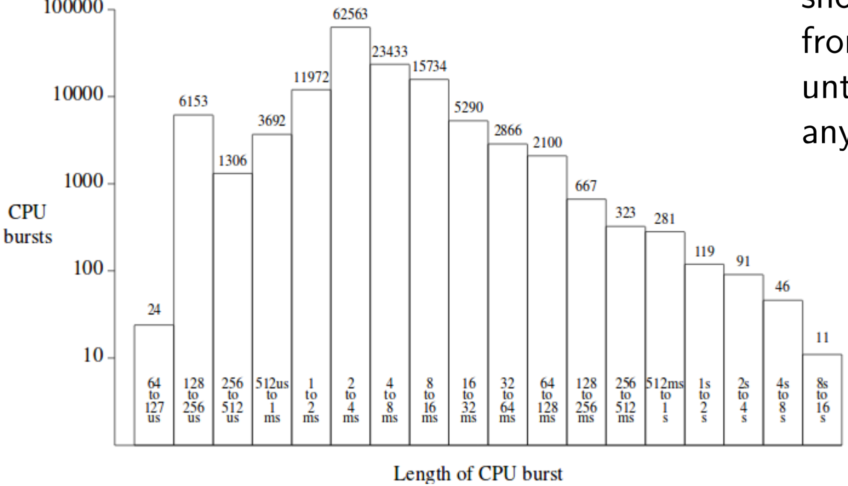


Figure 11—Compute time per task

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 (want *low*)

what user sees: from *keypress* to *character on screen*
(submission until job finsihed)

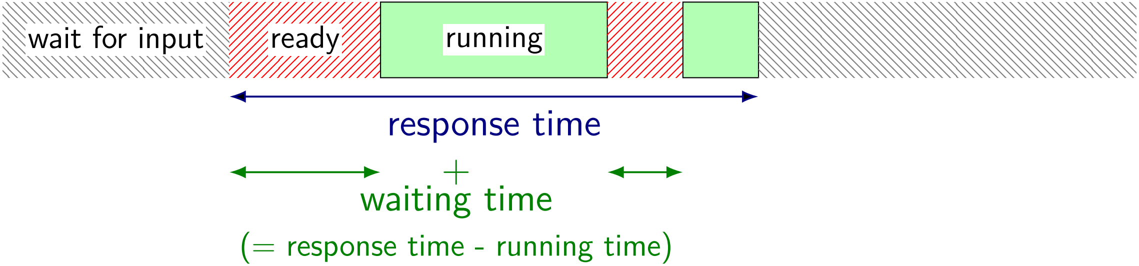
throughput (want *high*)

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

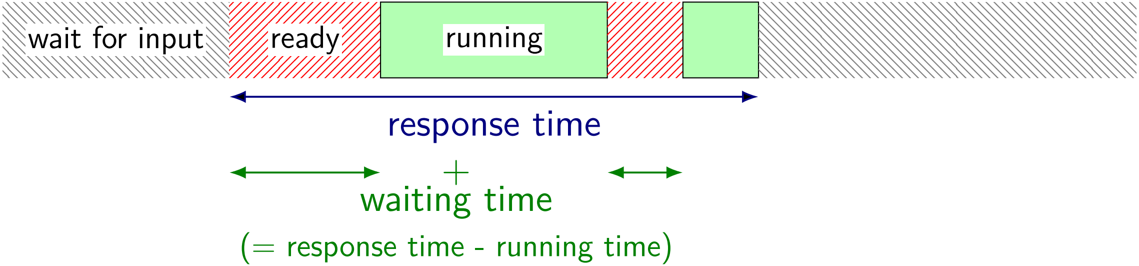
fairness

many definitions
all conflict with best average throughput/response time

response and wait time

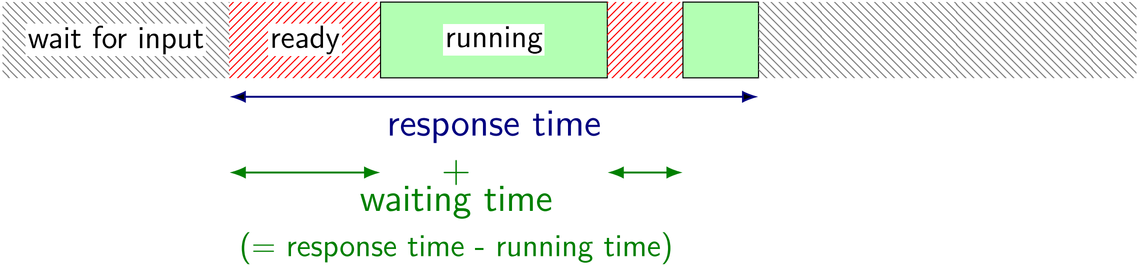


response and wait time



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

response and wait time



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

same as optimizing total/mean waiting time

response time and I/O

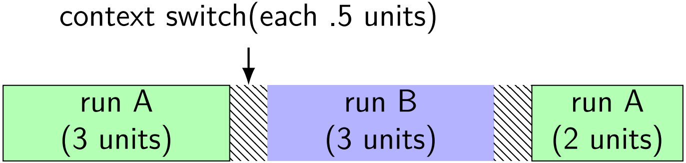
scheduling CPU bursts?

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

scheduling batch program on cluster?

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

throughput


context switch(each .5 units)

run A
(3 units)

run B
(3 units)

run A
(2 units)

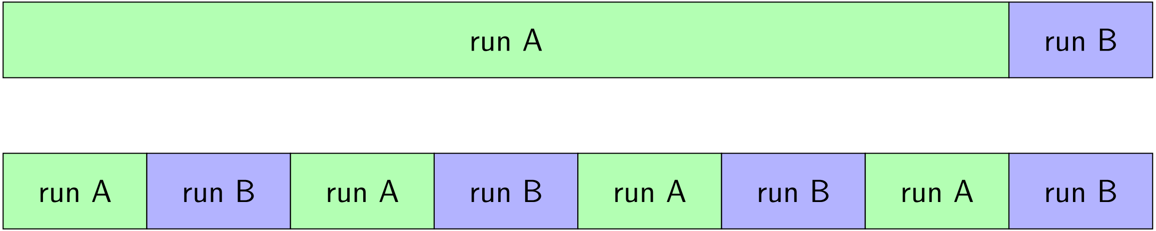
throughput: useful work done per unit time

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



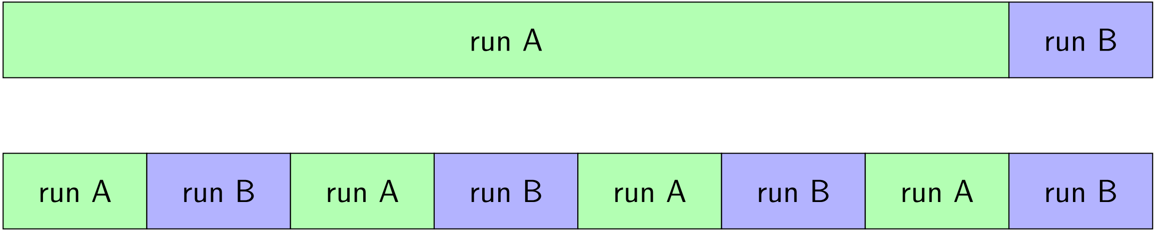
run A | run B

run A | run B | run A | run B | run A | run B | run A | run B

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

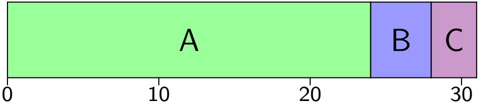
first-come, first-served (FCFS)

(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**



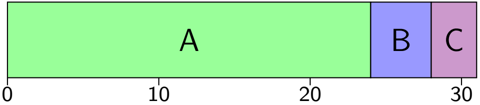
first-come, first-served (FCFS)

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

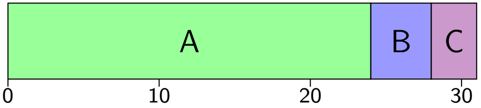
first-come, first-served (FCFS)

(AKA “first in, first out” (FIFO))

process	CPU time needed
A	24
B	4
C	3

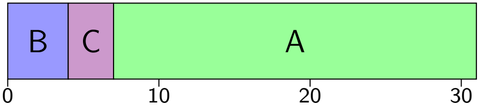
A ~ CPU-bound
B, C ~ I/O bound or interactive

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



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

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



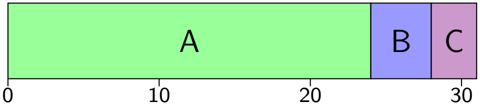
first-come, first-served (FCFS)

(AKA “first in, first out” (FIFO))

process	CPU time needed
A	24
B	4
C	3

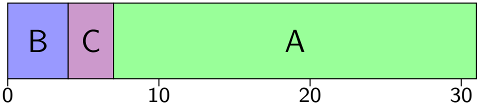
A $\sim$ CPU-bound
B, C $\sim$ I/O bound or interactive

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



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

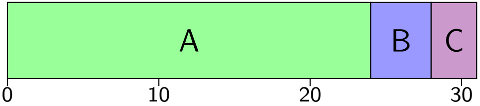
arrival order: **B**, **C**, **A**



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

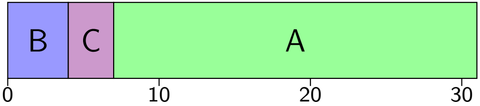
FCFS orders

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



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

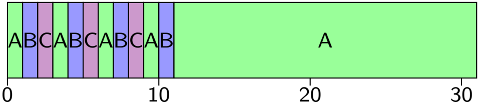
- like xv6's timer interrupt

requires good way of stopping programs whenever

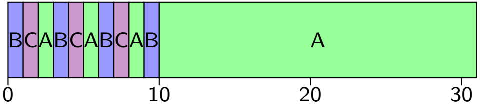
- like xv6's context switches

round robin (RR) (varying order)

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

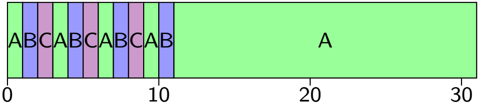


time quantum $= 1$,
order **B**, **C**, **A**



round robin (RR) (varying order)

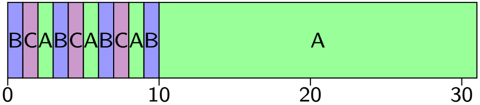
time quantum = 1,
order **A**, **B**, **C**



waiting times: (mean=6.7)
7 (**A**), 7 (**B**), 6 (**C**)

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

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

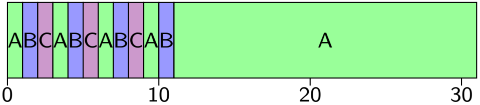


waiting times: (mean=6)
7 (**A**), 6 (**B**), 5 (**C**)

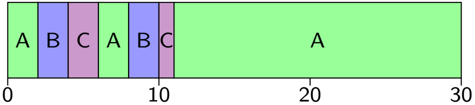
response times: (mean=16.3)
31 (**A**), 10 (**B**), 8 (**C**)

round robin (RR) (varying time quantum)

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

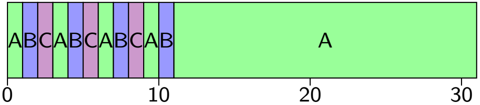


time quantum = 2,
order **A**, **B**, **C**



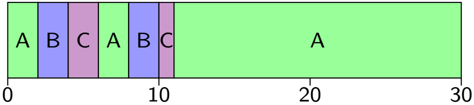
round robin (RR) (varying time quantum)

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



waiting times: (mean=6.7)
7 (**A**), 7 (**B**), 6 (**C**)
response 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**)
response 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)

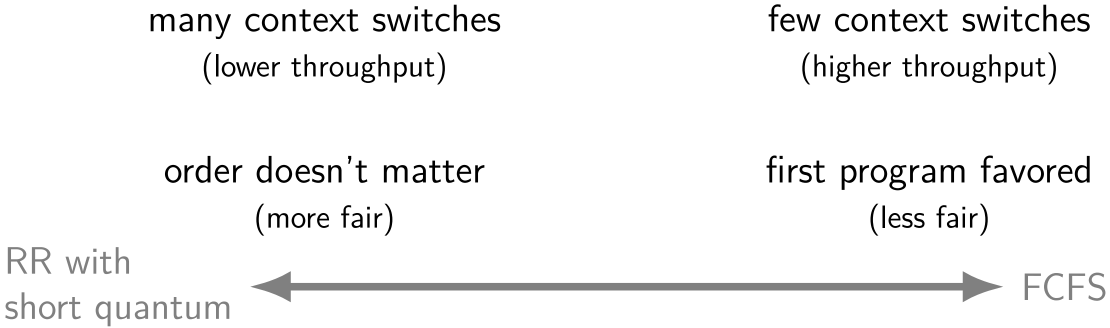
first program favored
(less fair)

RR with
short quantum

FCFS

smaller quantum: more fair, worse throughput

round robin and time quantums



many context switches
(lower throughput)

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

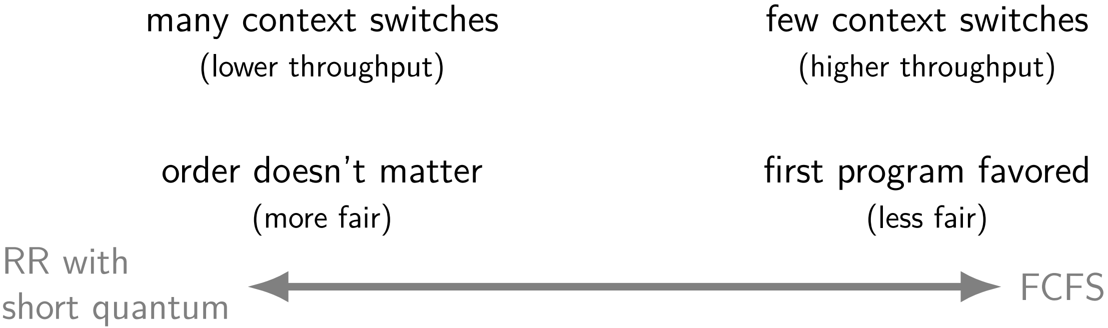
typical context switch: $\sim$ 0.01 ms to 0.1 ms
  - but tricky: lot of indirect cost (cache misses)
  - (above numbers try to include likely indirect costs)

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 response time?

exercise: round robin quantum

if there were no context switch overhead, *decreasing* the time quantum (for round robin) would cause average response time to ______________.

A. always decrease or stay the same

B. always increase of stay the same

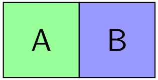
C. increase or decrease or stay the same

D. something else?

increase response time

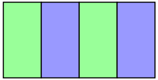
A: 1 unit CPU burst
B: 1 unit

$Q = 1$



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

$Q = 1/2$

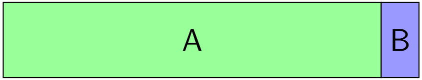


mean response time $=$
$(1.5+2) \div 2 = 1.75$

decrease response time

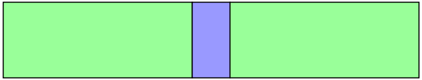
A: 10 unit CPU burst
B: 1 unit

Q = 10



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

Q = 5

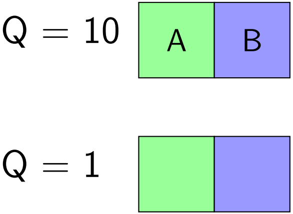


mean response time $=$
$(6 + 11) \div 2 = 8.5$

stay the same

A: 1 unit CPU burst

B: 1 unit



FCFS and order

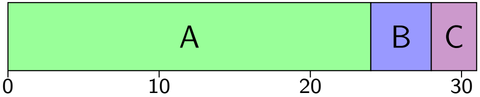
earlier we saw that with FCFS, arrival order mattered

big changes in response time

let's use that insight to see how to optimize response time

FCFS orders

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



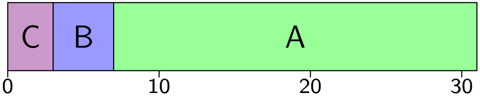
waiting times: (mean=17.3)

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

response times: (mean=27.7)

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

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



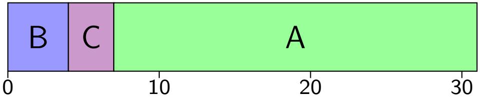
waiting times: (mean=3.3)

7 (**A**), 3 (**B**), 0 (**C**)

response 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**)

response times: (mean=14)

31 (**A**), 4 (**B**), 7 (**C**)

order and response time

best response time = run shortest CPU burst first

worst response time = run longest CPU burst first

intuition: “race to go to sleep”

diversion: some users are more equal

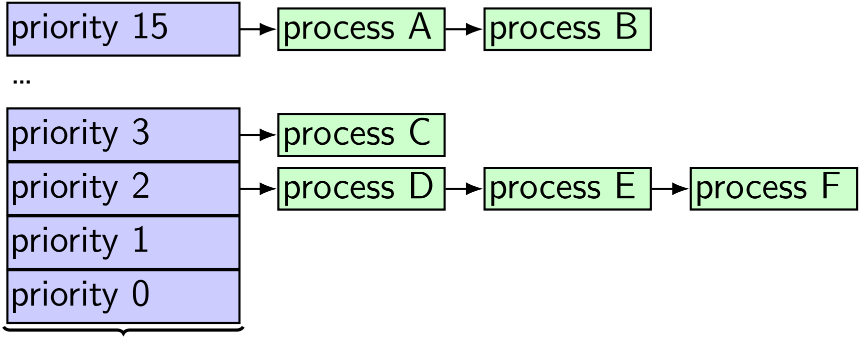
shells more important than big computation?

- i.e. programs with short CPU bursts

faculty more important than students?

scheduling algorithm: schedule shells/faculty programs first

priority scheduling



priority 15 → process A → process B

...

priority 3 → process C

priority 2 → process D → process E → process F

priority 1

priority 0

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:

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 response 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

- we'll talk about how to add preemption later (called SRTF)
- (the simplest possible idea doesn't quite work)

a practical problem

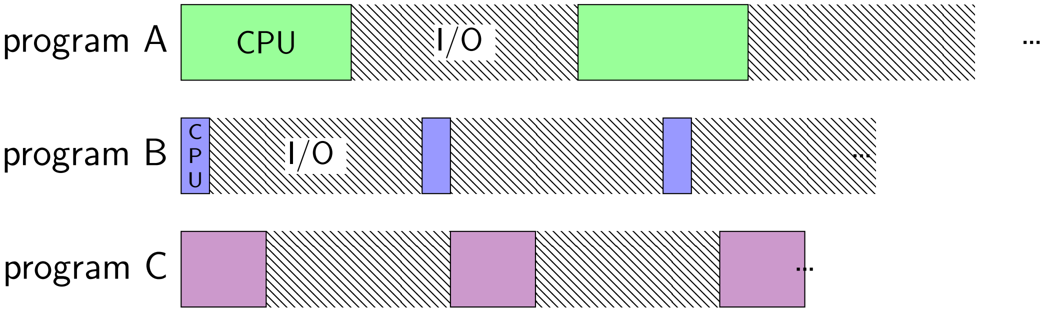
so we want to run the shortest CPU burst first

how do I tell which thread that is?

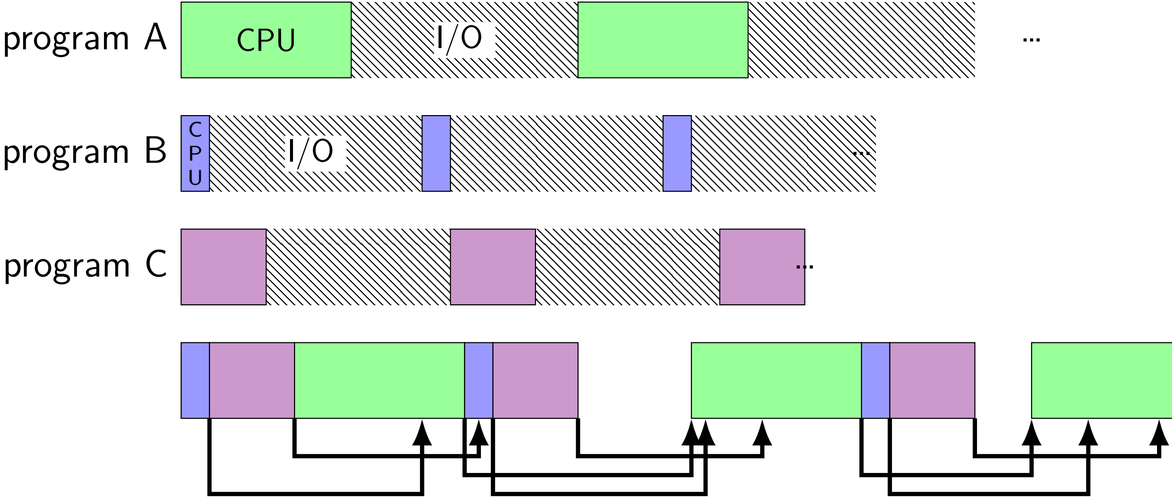
we’ll deal with this problem later

…kinda

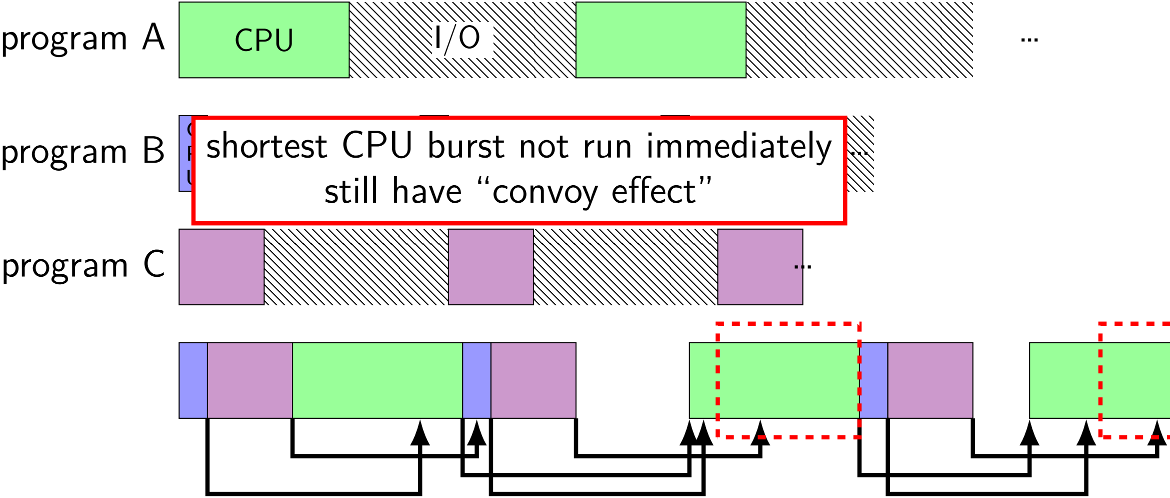
alternating I/O and CPU: SJF



alternating I/O and CPU: SJF



alternating I/O and CPU: SJF



shortest CPU burst not run immediately
still have “convoy effect”

backup slides

xv6: fork (1)

```
int
fork(void) {
  ...
  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    ... /* handle error */
  }
  np->sz = curproc->sz
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;
```

xv6: fork (2)

```
int
fork(void) {
  ...
  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);
```

exercise

```
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