

scheduling 1

Changelog

16 Feb 2021 (after lecture): write “runnable to not runnable” when defining turnaround time

last time

file descriptors as pointers to file

- pointers copied on fork

- close: set pointer to NULL

- dup2(X, Y): make Y point to same file as X

read/write system calls

- read/write up to amount requested

- read: wait for some data to be available

- write: wait for completion/failure

- returns amount written/read, or -1 on failure

- read = 0 means end-of-file

pipes and pipelines

- pipe library: pair of connected file descriptors

- OS managed buffer to store written, unread data

on the quiz generally

swtch questions were a lot less clear than I'd like

wasn't careful about eliminating issues

that would arise from other code using swtch() on first/second

should've used struct context pointer, not struct context

on the quiz and swtch

```
foo() {  
    ...  
    swtch(&x, y);  
    ...  
}
```

question: if bar () running, where are registers saved?

if registers saved by calling swtch: should be on foo()'s stack
also where compiler would save local vars that didn't fit in registers

if registers saved by exception interrupted . . . code
in trapframe, allocated on kernel stack
if foo() in kernel, the same stack foo() used

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

```
// 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;
}
```

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

xv6: creating the first process

```
// 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;
}
```

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

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 (accessible memory)

- 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

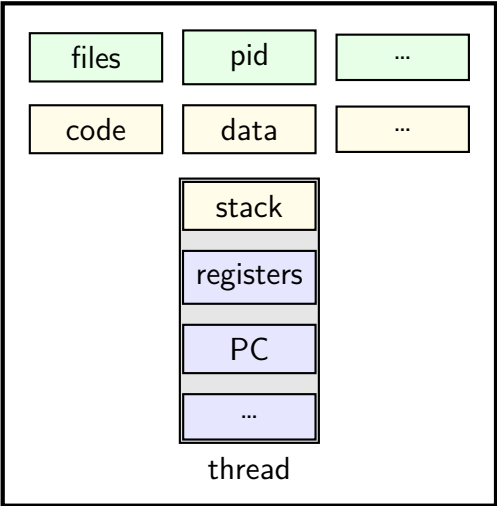
xv6: one thread per process

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

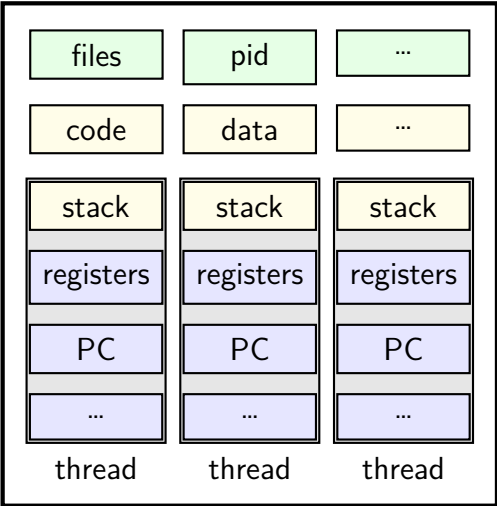
```
// Per-process state
struct proc {
    uint sz; // Size of process memory (bytes)
    pde_t* pgdir; // Page table
    char *kstack; // Bottom of kernel stack for this process
    enum procstate state; // Process state
    int pid; // Process ID
    struct proc *parent; // Parent process
    struct 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

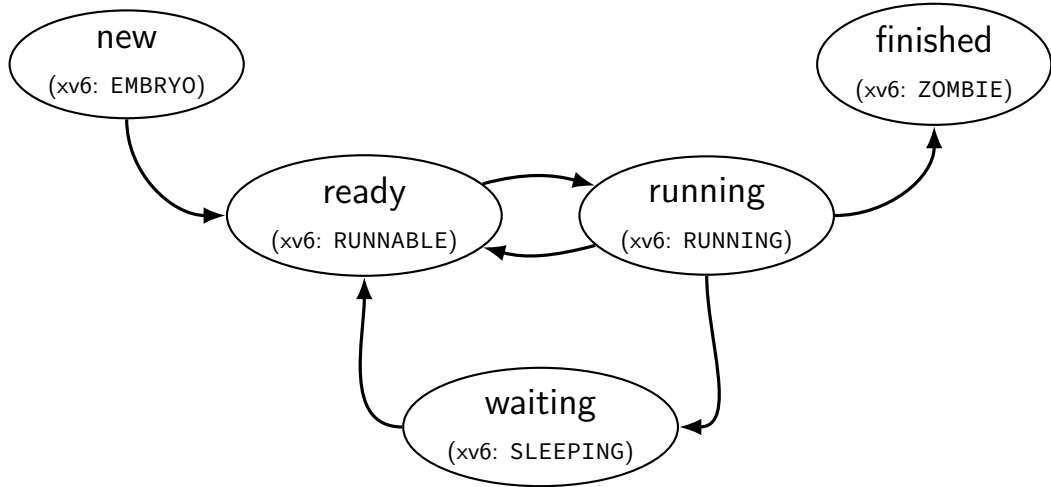
single-threaded process



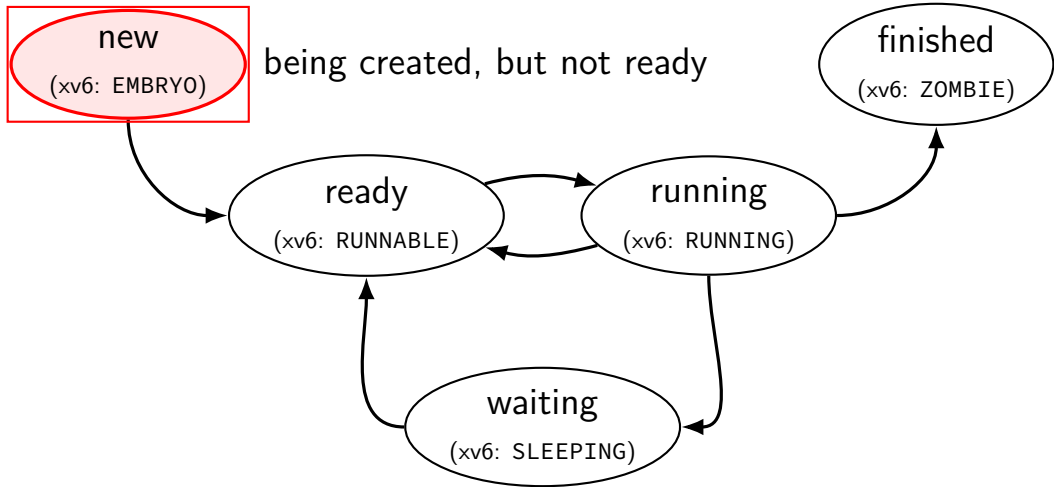
multi-threaded process



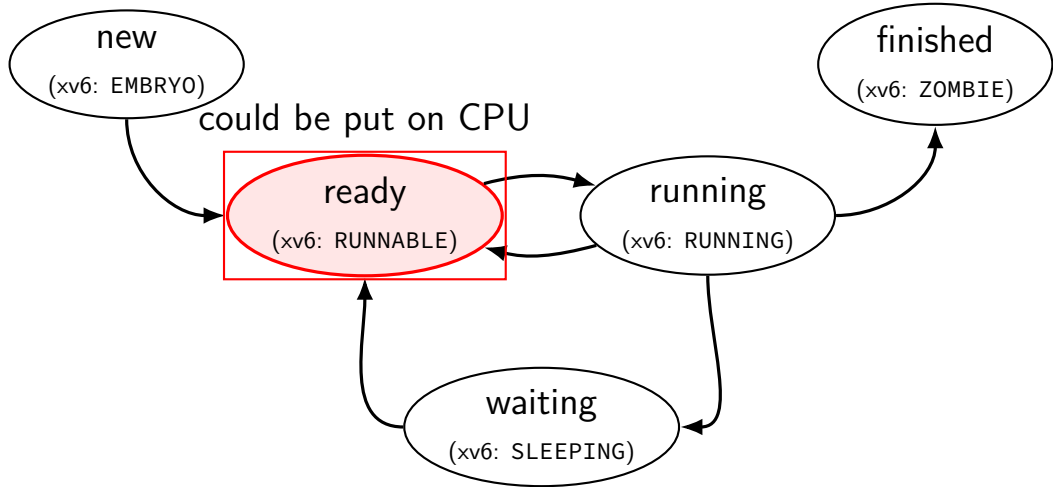
thread states



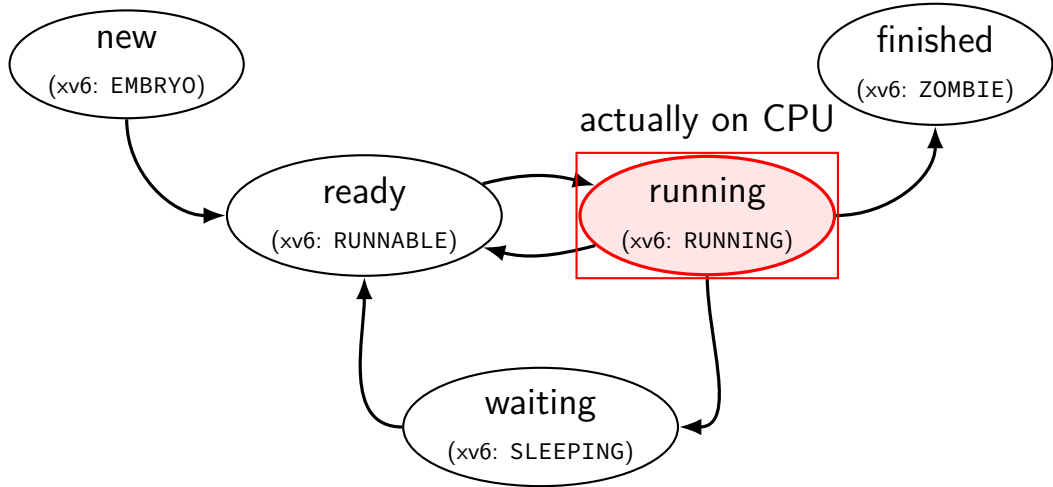
thread states



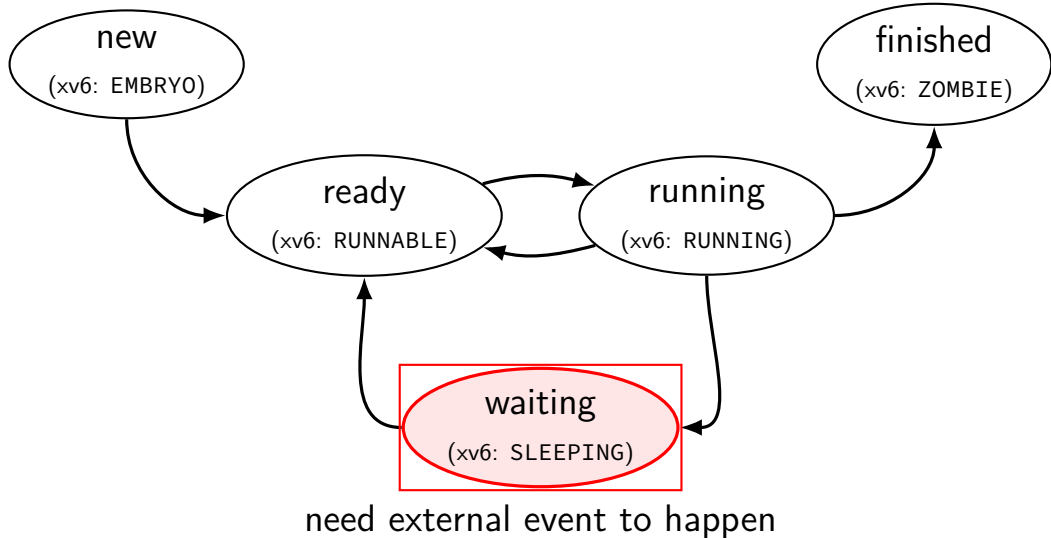
thread states



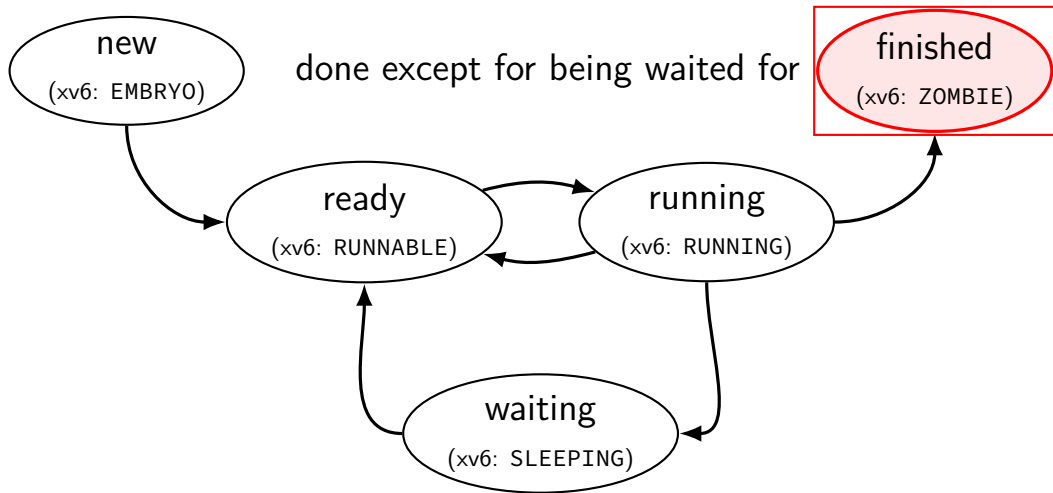
thread states



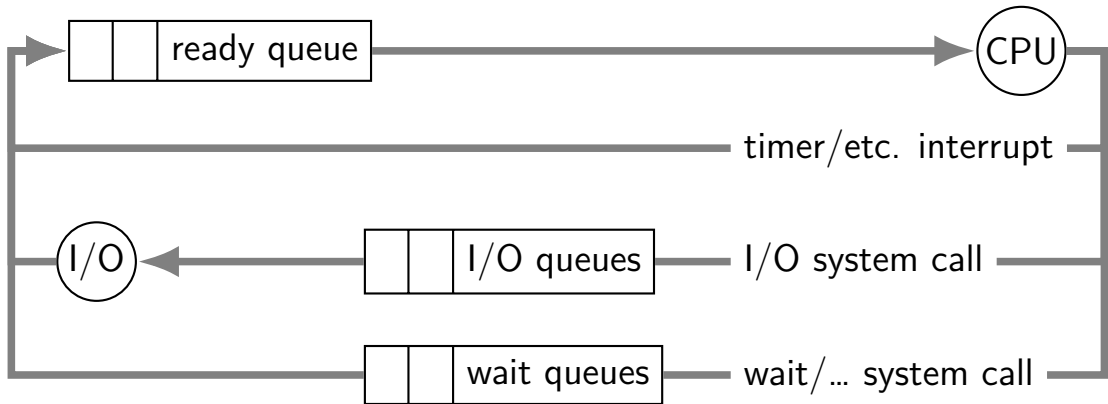
thread states



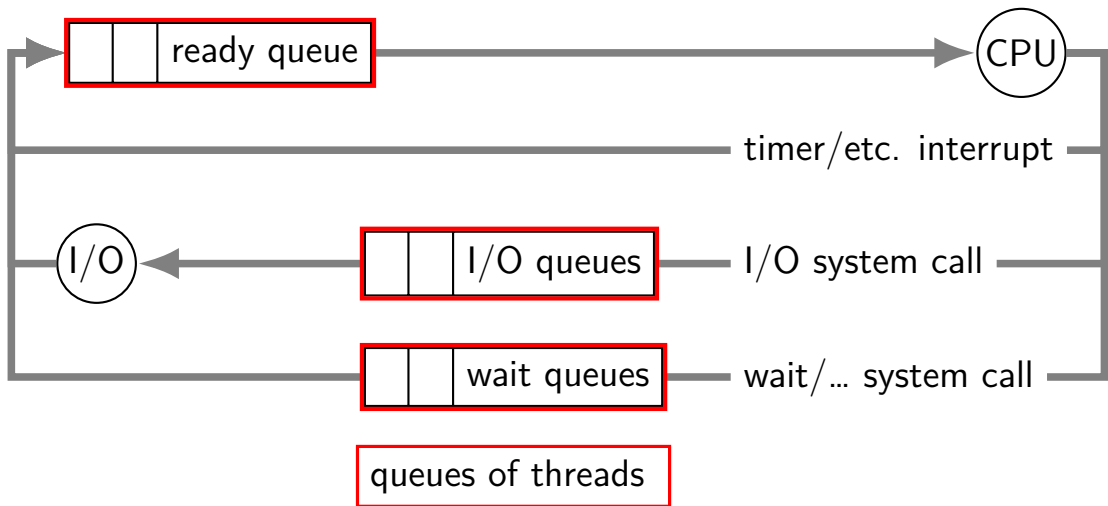
thread states



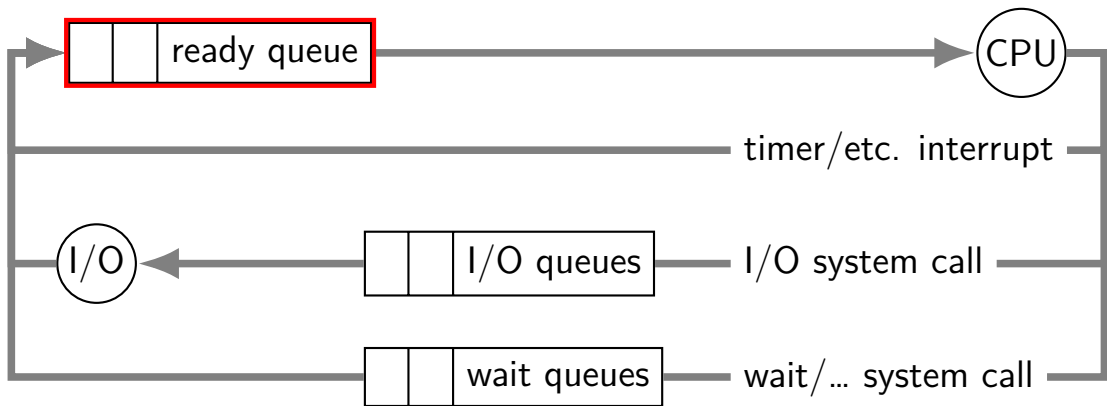
alternative view: queues



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

xv6 scheduler: outline

separate thread per core (with no associated process)

runs infinite loop:

- choose thread to switch to
- switch to that thread
- (and get switched back to)

program threads:

- do program stuff until the OS decides to stop
- then switch to current core's schedule thread
- (and get switched back to, repeat)

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;
            ... /* setup for process switch */
            swch(&(c->scheduler), p->context); /* ... */
            ... /* cleanup for process switch */
        }
        release(&ptable.lock);
    }
}
```


the xv6 scheduler (1)

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

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

```
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;  
        ... /* setup for process switch */  
        swch(&(c->scheduler), p->context); /* ... */  
        ... /* cleanup for process switch */
```

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

```
}
```

the xv6 scheduler (1)

```
void sched
struct p
struct c
c->proc

enable interrupts (sti is the x86 instruction)
makes sure keypresses, etc. will be handled
...(but acquiring the process table lock disables interrupts again)

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;
        ... /* setup for process switch */
        swch(&(c->scheduler), p->context); /* ... */
        ... /* cleanup for process switch */
    }
    release(&ptable.lock);
}
```

the xv6 scheduler (1)

```
void sched
struct p
struct d
c->proc

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;
        ... /* setup for process switch */
        swch(&(c->scheduler), p->context); /* ... */
        ... /* cleanup for process switch */
    }
    release(&ptable.lock);
}
```

make sure we're the only one accessing the list of processes
disables interrupts
e.g. don't want timer interrupt to switch while already switching

the xv6 scheduler (1)

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

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

```
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;  
        ... /* setup for process switch */  
        swch(&(c->scheduler), p->context); /* ... */  
        ... /* cleanup for process switch */  
    }  
    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;
            ... /* setup for process switch */
            swtch(&(c->scheduler), p->context); /* ... */
            ... /* cleanup for process switch */
        }
        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;
switchvm(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 chosen process
// to release ptable.
// before jumping back to it
c->proc = p;
switchvm(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: the actual switch

```
/* in scheduler(): */  
// Switch  
// to rele prepare: change address space, change process state  
// before jumping back to us.  
c->proc = p;  
switchvm(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 ju
```

```
    c->proc = p;
```

```
    switchvm(p);
```

```
    p->state = RUNNING;
```

switch to **kernel thread** of process

that thread responsible for going back to user mode

```
    switch(&(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 process c after we've run the process until it's done, we end up here  
// to restore the user process's state  
// before we switch to the next process  
c->proc = p; //...so, change address space back away from user process  
switch(&c->context, p->context);  
p->state = RUNNING;
```

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

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

switching to/from scheduler

- (1) acquire process table lock
 - prevent someone else from switching to scheduler at same time
 - ...causing confusion about what's running/runnable
 - (someone else = timer interrupt, another core, ...)
- (2) mark current process as not running
- (3) actually switch to scheduler thread
 - scheduler thread runs, possibly switches to other threads, etc.

- (4) scheduler thread switches back
 - invariant: process table lock held
 - invariant: current thread marked running
- (5) release process table lock

switching to/from scheduler

(1) acquire process table lock

prevent someone else from switching to scheduler at same time
...causing confusion about what's running/runnable
(someone else = timer interrupt, another core, ...)

(2) mark current process as not running

(3) actually switch to scheduler thread

scheduler thread runs, possibly switches to other threads, etc.

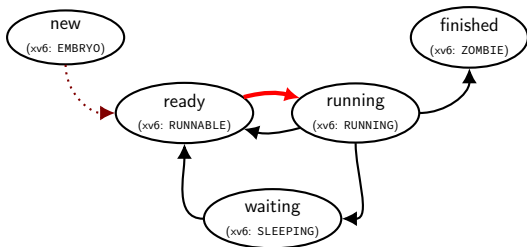
(4) scheduler thread switches back

invariant: process table lock held

invariant: current thread marked running

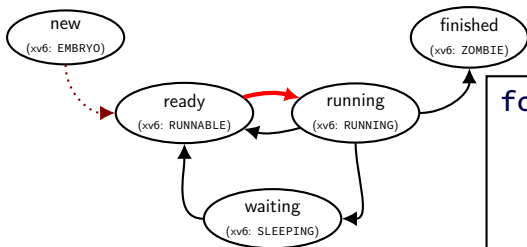
(5) release process table lock

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

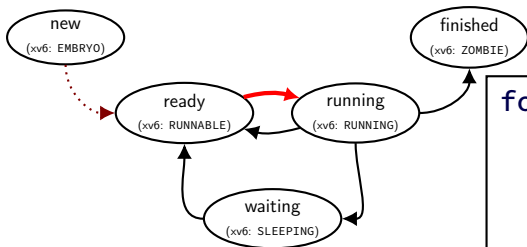


scheduler()

```
for (...) { // iterate over RUNNABLE
    ...
    p->state = RUNNING;
    swtch(&(c->scheduler), p->context);
    ...
}
```

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

the xv6 scheduler: on process start



scheduler()

```
for (...) { // iterate over RUNNABLE
    ...
    p->state = RUNNING;
    swtch(&(c->scheduler), p->context);
    ...
}
```

```
void forkret() {
    /* scheduler switches
    ...
    release(&ptable.lock);
    ...
}
```

scheduler switched with process table locked
need to unlock before running user code
(allow timer interrupts, etc.)

switching to/from scheduler

(1) acquire process table lock

prevent someone else from switching to scheduler at same time
...causing confusion about what's running/runnable
(someone else = timer interrupt, another core, ...)

(2) mark current process as not running

(3) actually switch to scheduler thread

scheduler thread runs, possibly switches to other threads, etc.

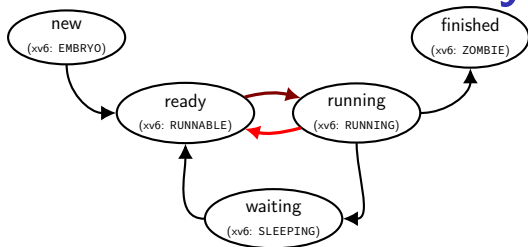
(4) scheduler thread switches back

invariant: process table lock held

invariant: current thread marked running

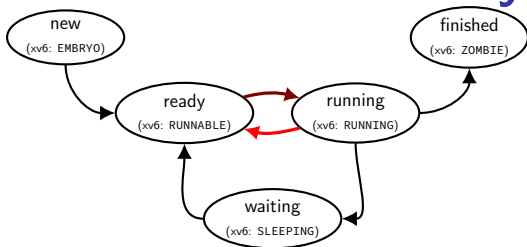
(5) release process table lock

the xv6 scheduler: yield (timer int.)



```
/* 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: yield (timer int.)

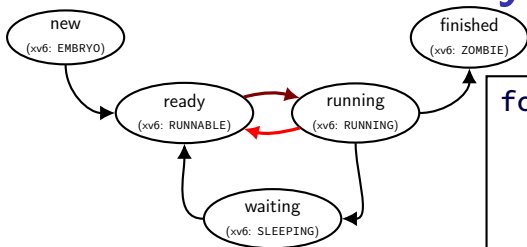


/ function to invoke scheduler;
used by the timer interrupt or y*

```
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: yield (timer int.)



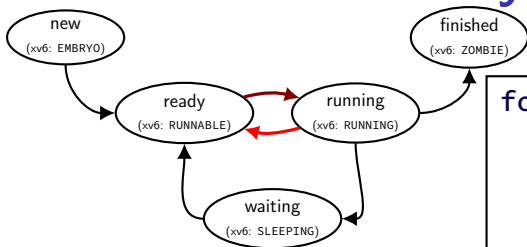
scheduler()

```
for (...) { // iterate over RUNNABLE
    ...
    p->state = RUNNING;
    swtch(&(c->scheduler), p->context);
    ...
}
```

```
/* 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: yield (timer int.)



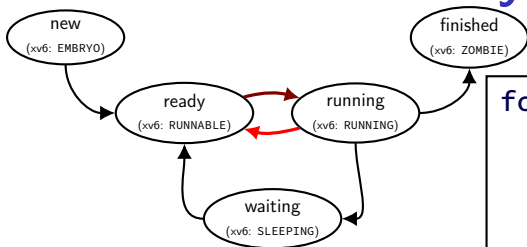
scheduler()

```
for (...) { // iterate over RUNNABLE
    ...
    p->state = RUNNING;
    swtch(&(c->scheduler), p->context);
    ...
}
```

```
/* function to
   used by the
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 / entering scheduler

the xv6 scheduler: yield (timer int.)



scheduler()

```
for (...) { // iterate over RUNNABLE
    ...
    p->state = RUNNING;
    swtch(&(c->scheduler), p->context);
    ...
}
```

/ function to invoke scheduler
used by the timer interrupt*

```
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

switching to/from scheduler

(1) acquire process table lock

prevent someone else from switching to scheduler at same time
...causing confusion about what's running/runnable
(someone else = timer interrupt, another core, ...)

(2) mark current process as not running

(3) actually switch to scheduler thread

scheduler thread runs, possibly switches to other threads, etc.

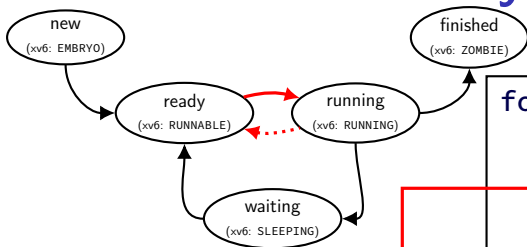
(4) scheduler thread switches back

invariant: process table lock held

invariant: current thread marked running

(5) release process table lock

the xv6 scheduler: yield (timer int.)

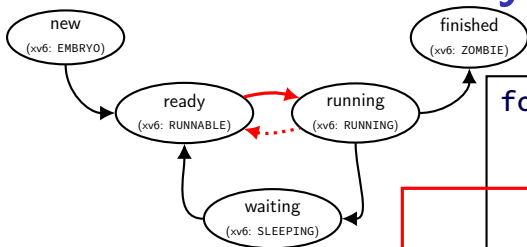


scheduler()

```
for (...) { // iterate over RUNNABLE
    ...
    p->state = RUNNING;
    swtch(&(c->scheduler), p->context);
    ...
}
```

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

the xv6 scheduler: yield (timer int.)

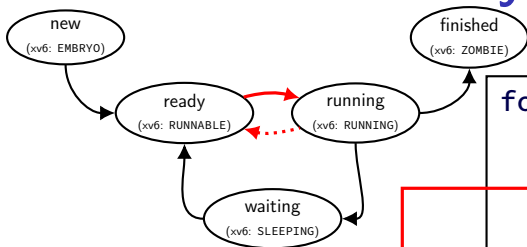


scheduler()

```
for (...) { // iterate over RUNNABLE
  ...
  p->state = RUNNING;
  swtch(&(c->scheduler), p->context);
  ...
}
```

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


the xv6 scheduler: yield (timer int.)

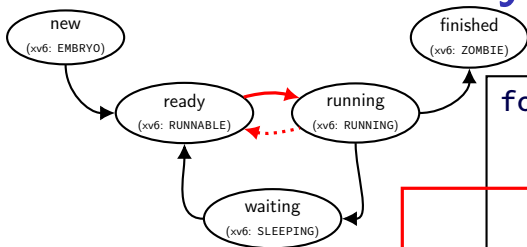


scheduler()

```
for (...) { // iterate over RUNNABLE
    ...
    p->state = RUNNING;
    swch(&(c->scheduler), p->context);
    ...
}
```

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

the xv6 scheduler: yield (timer int.)



scheduler()

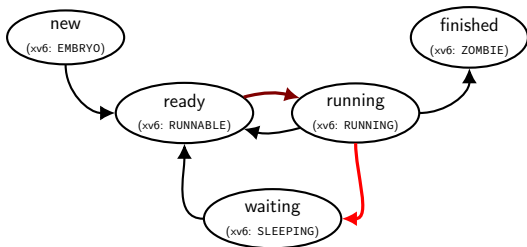
```
for (...) { // iterate over RUNNABLE
    ...
    p->state = RUNNING;
    swtch(&(c->scheduler), p->context);
    ...
}
```

/ function to invoke scheduler
used by the timer interrupt*

```
void yield() {
    acquire(&ptable.lock);
    myproc()->state = RUNNABLE;
    sched(); // switches to scheduler thread
    release(&ptable.lock);
}
```

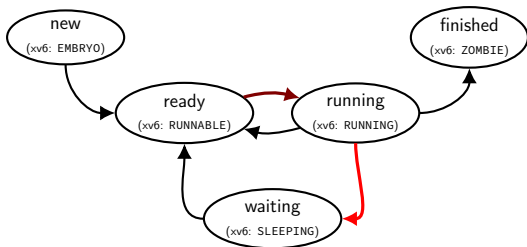
process table was 'locked'
unlock it before running user code
otherwise: timer interrupt/etc. won't work

the xv6 scheduler: entering/leaving for sleep



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

the xv6 scheduler: entering/leaving for sleep



```
void sleep(void *chan, ...) { ...
```

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

```
    ...
```

```
    p->chan = chan;
```

```
    p->state = SLEEPING;
```

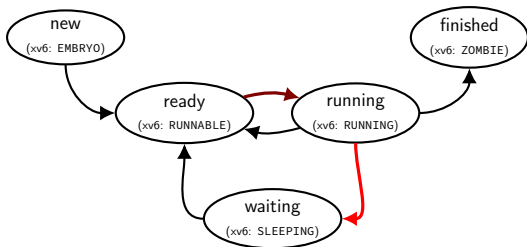
```
    sched();
```

```
    ...
```

```
    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



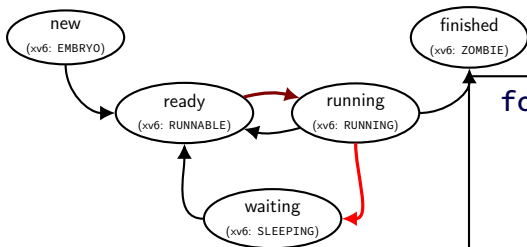
```
void sleep(void *chan, ...) { ...
    acquire(&ptable.lock);
    ...
    p->chan = chan;
    p->state = SLEEPING;

    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



scheduler()

```
for (...) { // iterate over RUNNABLE
  ...
  p->state = RUNNING;
  swtch(&(c->scheduler), p->context);
  ...
}
```

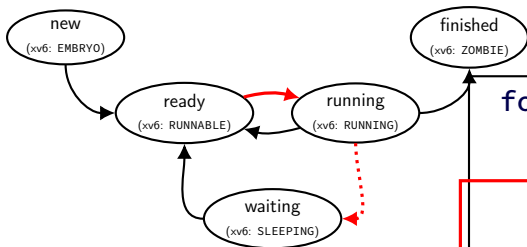
```
void sleep(void *chan, ...) { ...
  acquire(&ptable.lock);
  ...
  p->chan = chan;
  p->state = SLEEPING;
```

...and switch to the scheduler infinite loop

```
  sched();
```

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

the xv6 scheduler: entering/leaving for sleep



scheduler()

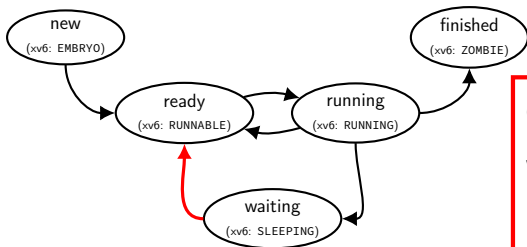
```
for (...) { // iterate over RUNNABLE
  ...
  p->state = RUNNING;
  switch(&(c->scheduler), p->context);
  ...
}
```

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

  sched();

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

the xv6 scheduler: SLEEPING to RUNNABLE



design choice:

wakeup just sets as runnable

actual switch always happens later

```
static void
```

```
wakeup1(void *chan)
```

```
{
```

```
    struct proc *p;
```

```
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
```

```
        if(p->state == SLEEPING && p->chan == chan)
```

```
            p->state = RUNNABLE;
```

```
}
```


xv6 scheduler odd choices

separate scheduler thread

pro: keep scheduler state (last process *p*) on the stack

con: slower — more thread switches

scan process list to find sleeping/waiting threads

alternative: separate list of waiting threads

(...definitely faster if lots of non-runnable threads)

process state tracking code tightly integrated with *policy*

alternative: utility function to manage process states, current process value, etc.

the scheduling policy problem

what RUNNABLE program should we run?

xv6 answer: whatever's next in list

best answer?

well, what should we care about?

some simplifying assumptions

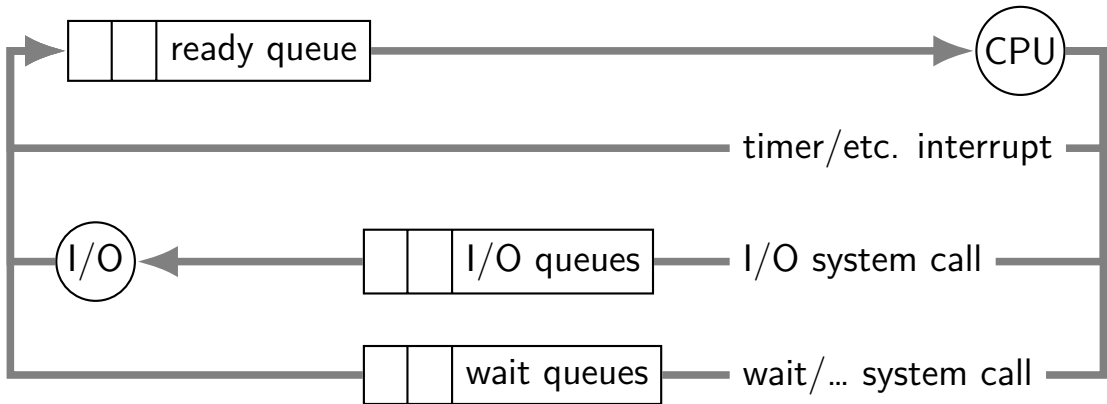
welcome to 1970:

one program per user

one thread per program

programs are independent

recall: scheduling queues



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)

shows compute time
from command entered
until next command prompt

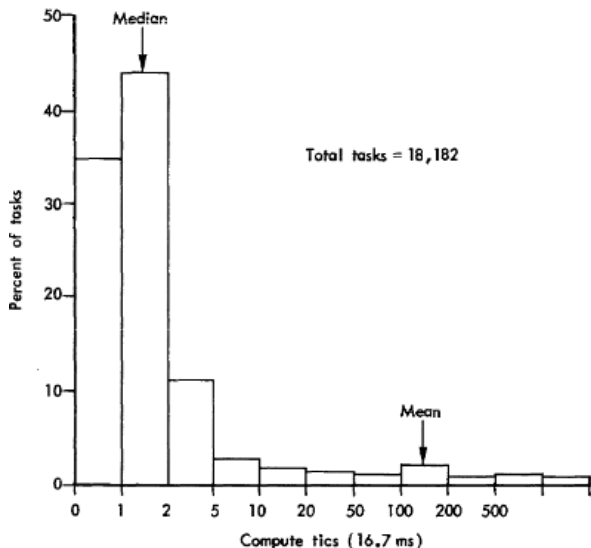
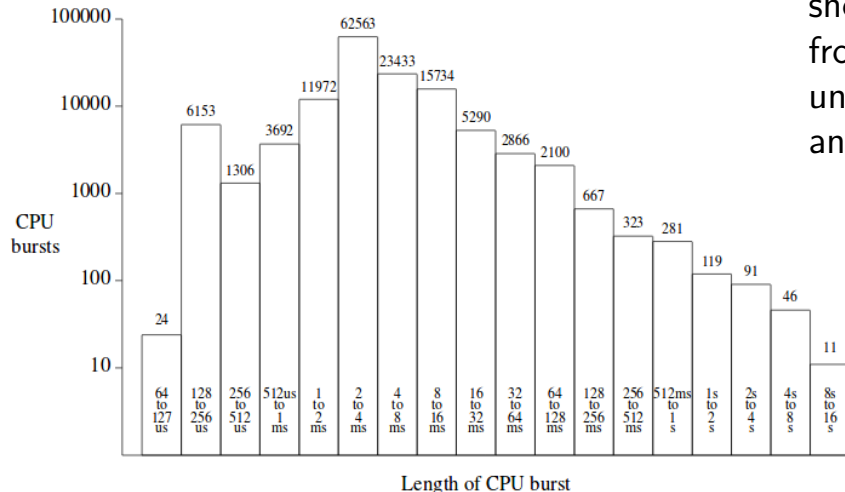


Figure 11—Compute time per task

CPU bursts and interactivity (one c. 1990 desktop)



shows CPU time from RUNNING until not RUNNABLE anymore

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

turnaround time (Arpaci-Dusseau) AKA **response time**
(Anderson-Dahlin)(want *low*)

(what Arpaci-Dusseau calls response time is related, but slightly different)

what user sees: from *keypress* to *character on screen*

(submission until job finished — runnable to not runnable)

throughput (want *high*)

total work per second

problem: overhead (e.g. from context switching)

fairness

many definitions

all **conflict** with best average throughput/turnaround time

turnaround time and I/O

scheduling CPU bursts? (what we'll mostly deal with)

turnaround time \approx time to start next I/O

turnaround time = time from runnable to not runnable again

important for fully utilizing I/O devices

closed loop: faster turnaround time \rightarrow program requests CPU sooner

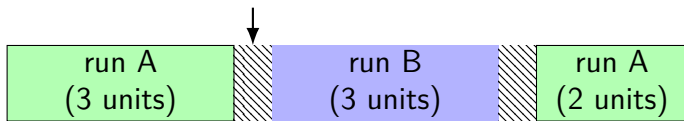
scheduling batch program on cluster?

turnaround time \approx how long does user wait

once program done with CPU, it's probably done

throughput

context switch
(each .5 units)



throughput: **useful work** done per unit time

$$\text{non-context switch CPU utilization} = \frac{3 + 3 + 2}{3 + .5 + 3 + .5 + 2} = 88\%$$

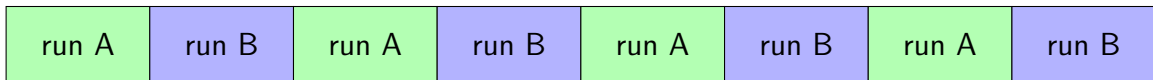
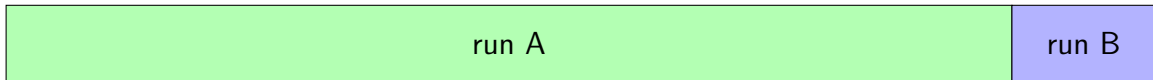
also other considerations:

- time lost due to cold caches

- time lost not starting I/O early as possible

- ...

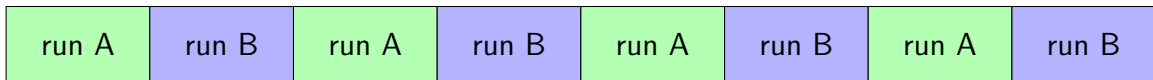
fairness



assumption: one program per user

two timelines above; which is fairer?

fairness

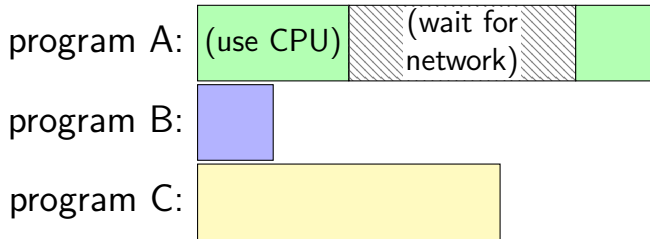


assumption: one program per user

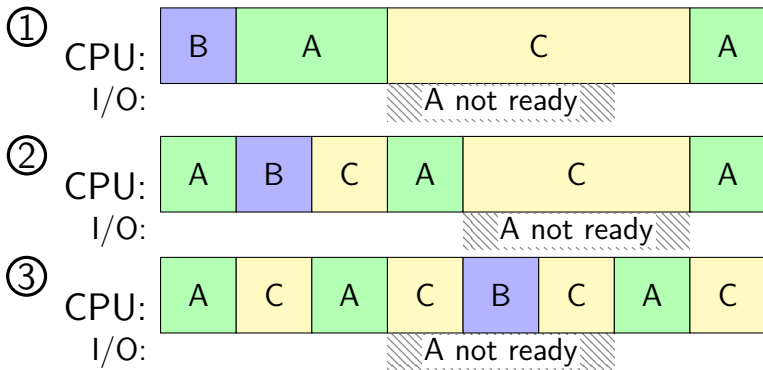
two timelines above; which is fairer?

easy to answer — but formal definition?

metrics example/exercise (1)

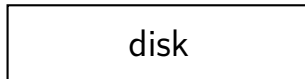
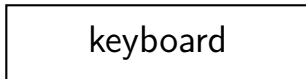
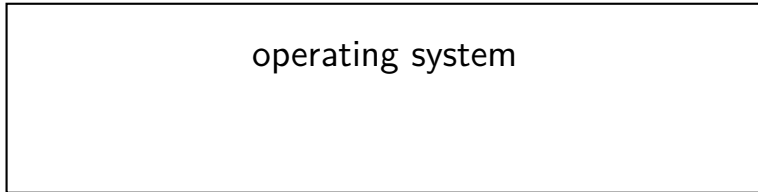
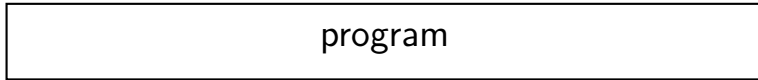


which schedule is better for:
throughput?
mean turnaround time?
fairness? responsiveness?

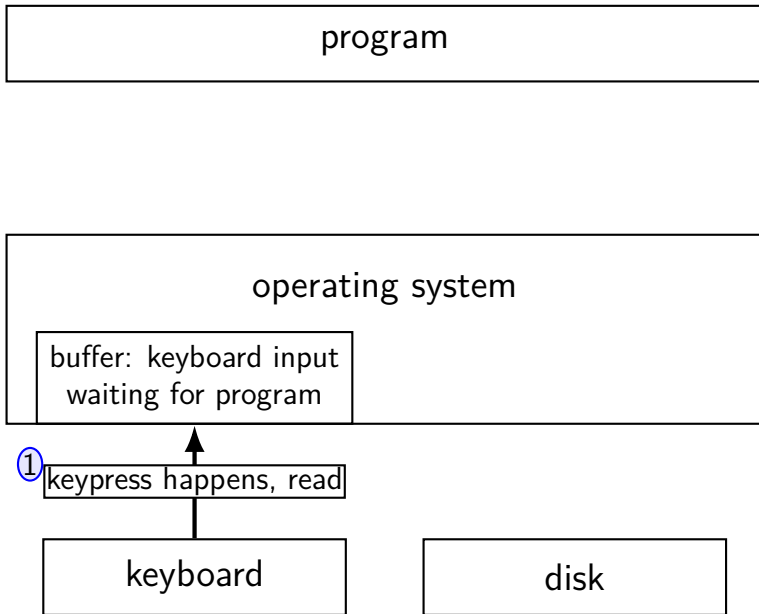


backup slides

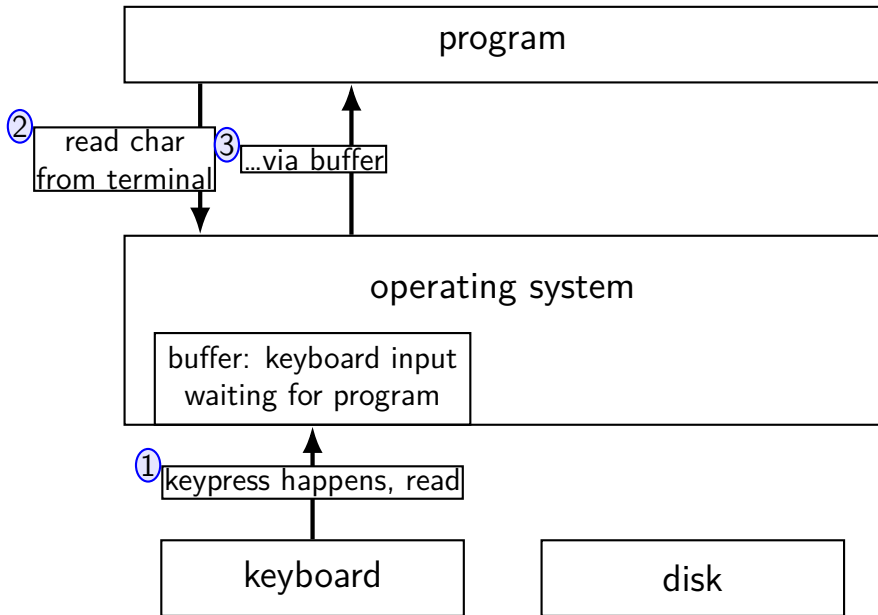
kernel buffering (reads)



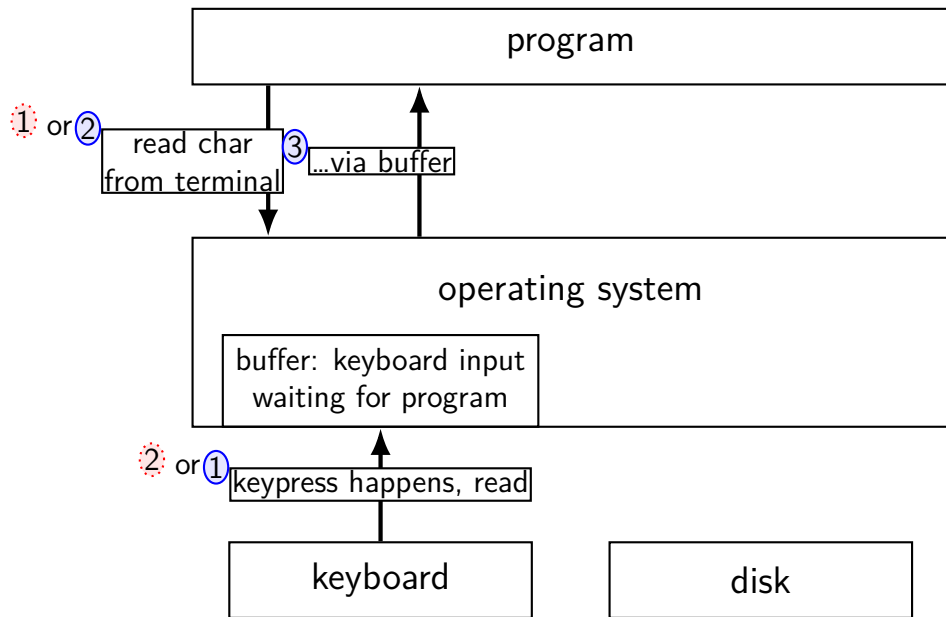
kernel buffering (reads)



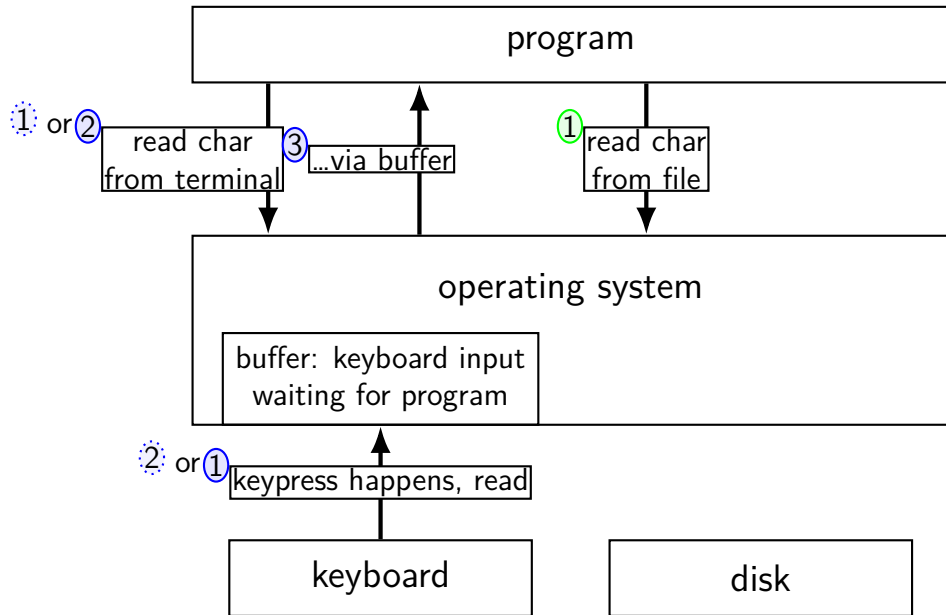
kernel buffering (reads)



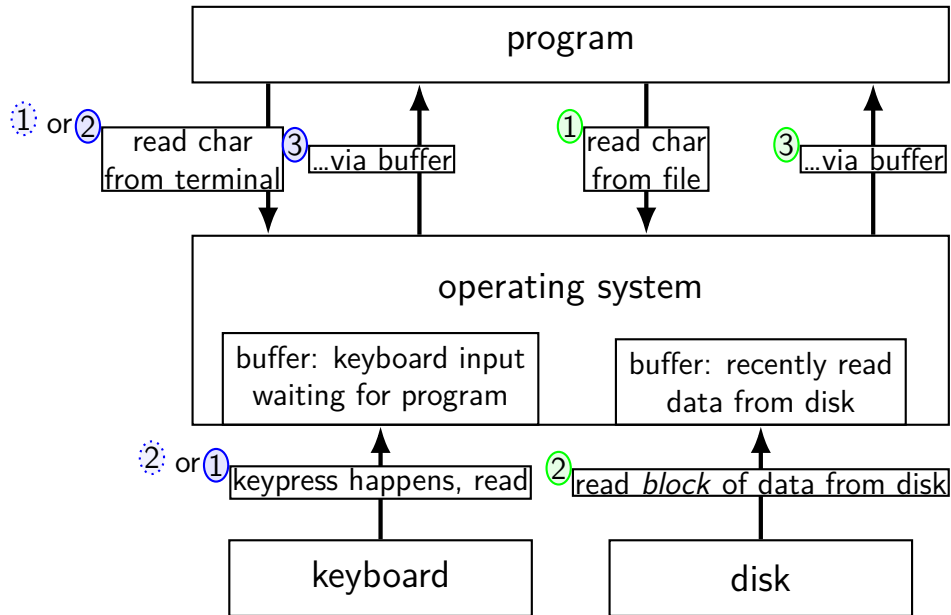
kernel buffering (reads)



kernel buffering (reads)



kernel buffering (reads)



kernel buffering (writes)

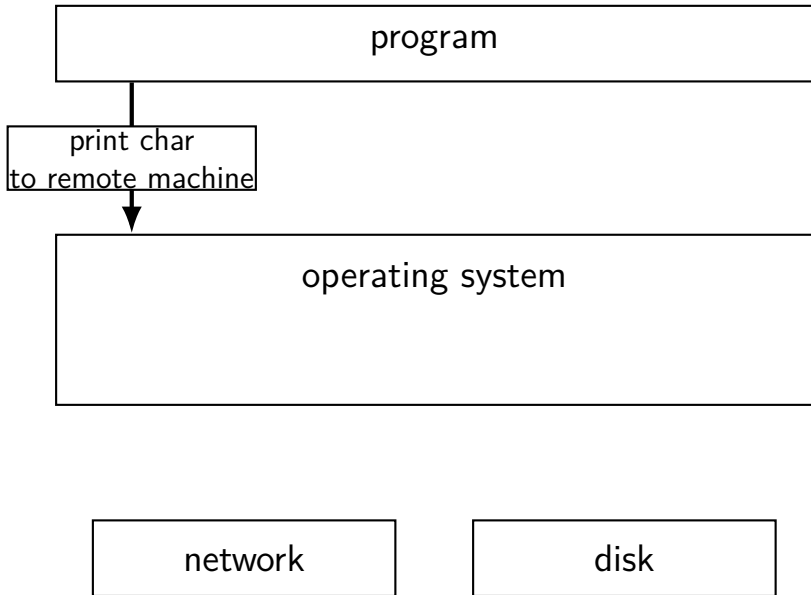
program

operating system

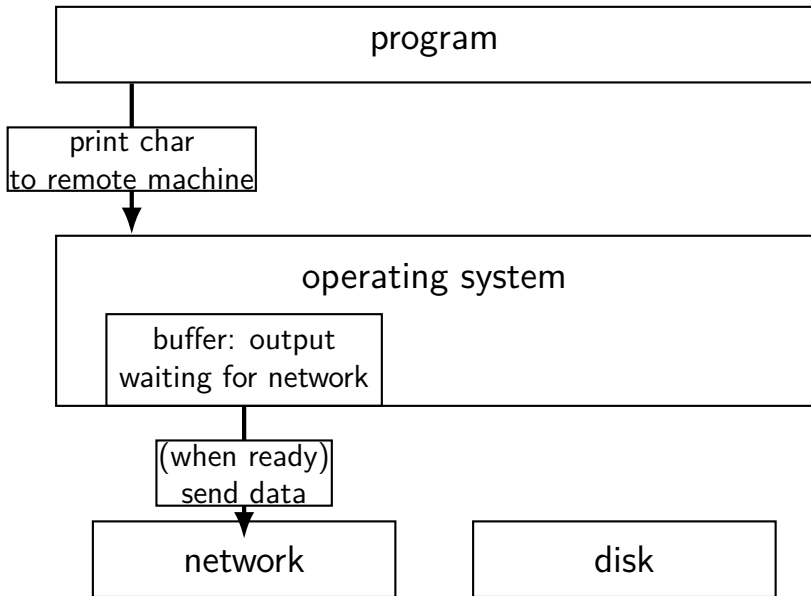
network

disk

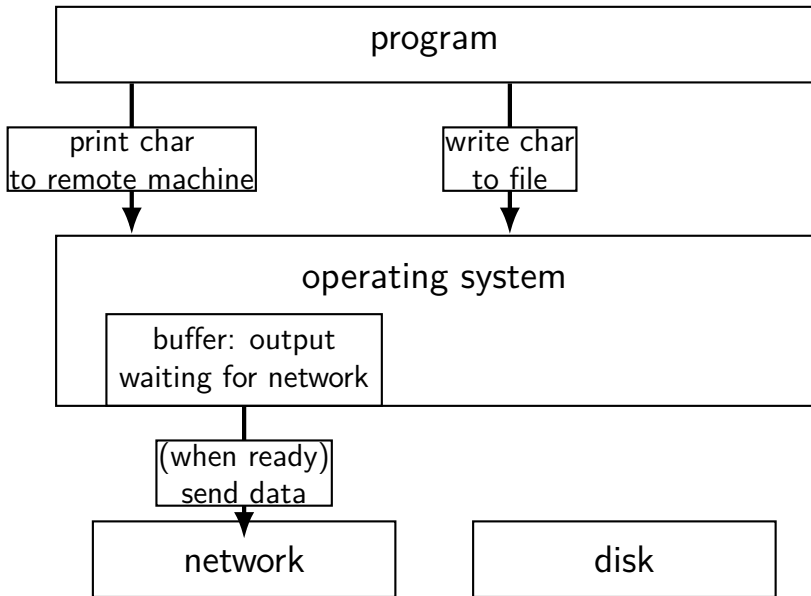
kernel buffering (writes)



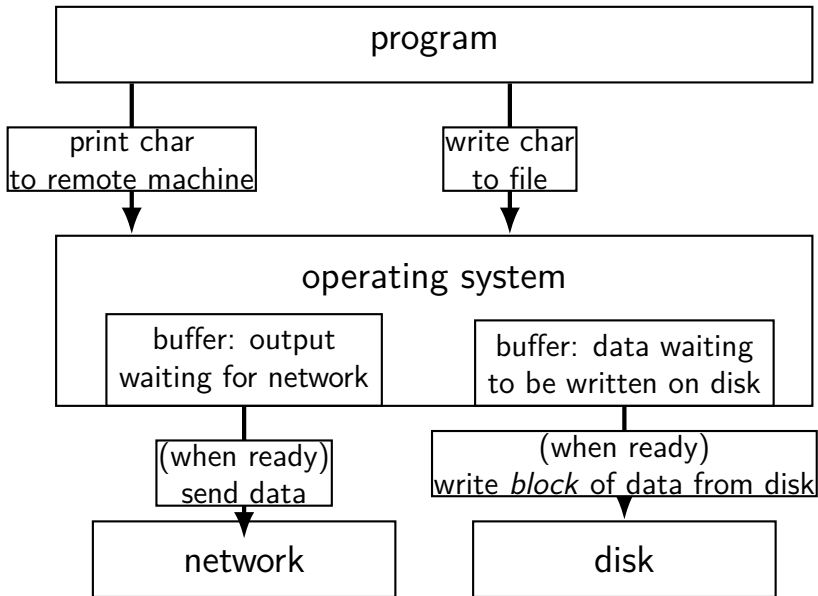
kernel buffering (writes)



kernel buffering (writes)



kernel buffering (writes)



read/write operations

read()/write(): move data into/out of buffer

possibly wait if buffer is empty (read)/full (write)

actual I/O operations — wait for device to be ready
trigger process to stop waiting if needed

mixing `stdio`/`iostream` and `raw read/write`

don't do it (unless you're very careful)

`cin/scanf` read some extra characters into a buffer?

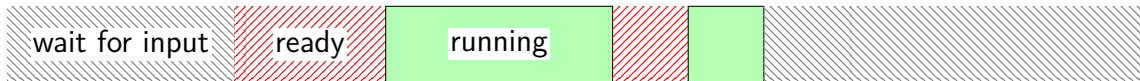
you call `read` — they disappear!

`cout/printf` has output waiting in a buffer?

you call `write` — out-of-order output!

(if you need to: some `stdio` calls specify that they clear out buffers)

turnaround and wait time

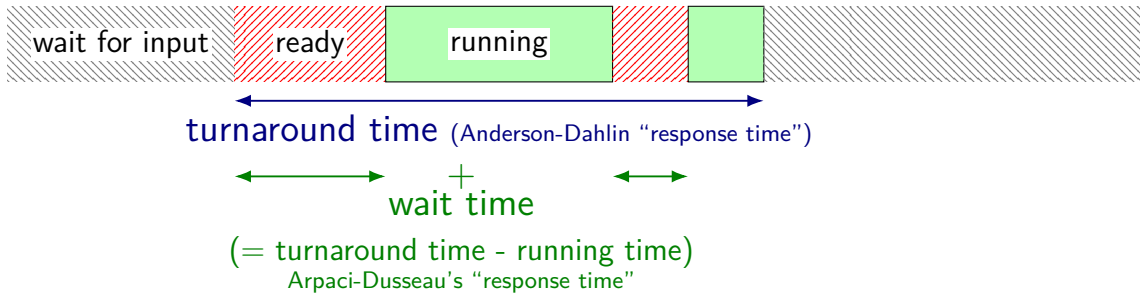


turnaround time (Anderson-Dahlin "response time")

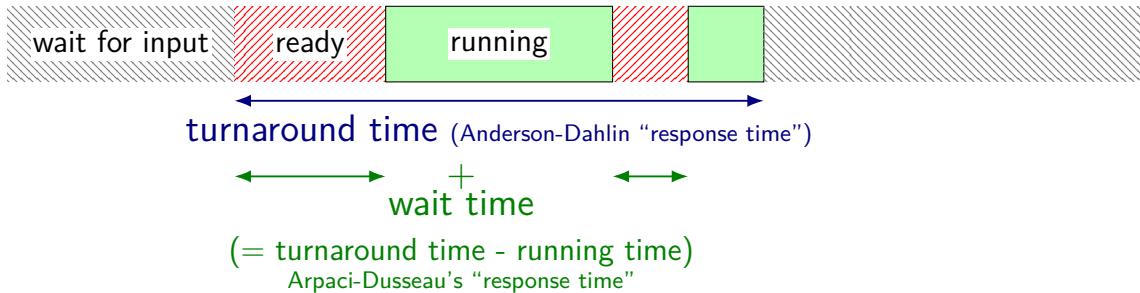
wait time

(= turnaround time - running time)
Arpaci-Dusseau's "response time"

turnaround and wait time

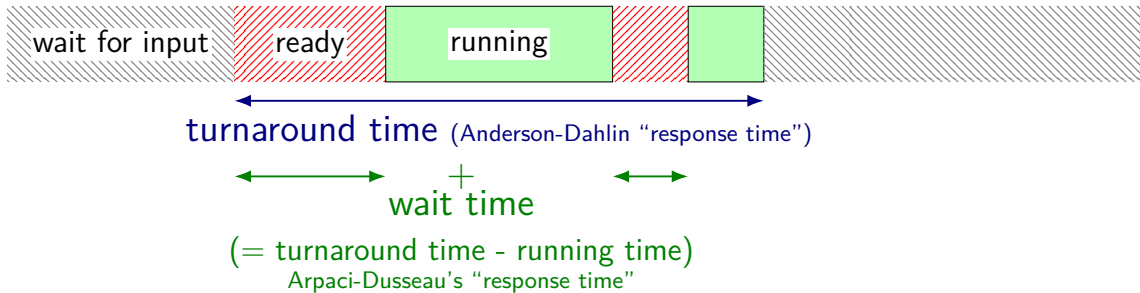


turnaround and wait time



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

turnaround and wait time



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

same as optimizing mean/total waiting time