last time

shell: redirection, pipelines

file descriptors point to open files

fork() copies pointers

dup2() – assign one pointer to another

close() sets pointer to NULL

automatic cleanup on no more refs

pipe() — pair of connect fds

- write X to one → read X from another

anonymous feedback / readings

I moved some things on the schedule without updating readings

caused some confusion — readings from scheduling (next topic) on earlier days

pipes

special kind of file: pipes

bytes go in one end, come out the other — once

created with `pipe()` library call

intended use: communicate between processes

- like implementing shell pipelines

pipe()

```
int pipe_fd[2];
if (pipe(pipe_fd) < 0)
    handle_error();
/* normal case: */
int read_fd = pipe_fd[0];
int write_fd = pipe_fd[1];
```

then from one process...

```
write(write_fd, ...);
```

and from another

```
read(read_fd, ...);
```

pipe() and blocking

BROKEN example:

```
int pipe_fd[2];
if (pipe(pipe_fd) < 0)
    handle_error();
int read_fd = pipe_fd[0];
int write_fd = pipe_fd[1];
write(write_fd, some_buffer, some_big_size);
read(read_fd, some_buffer, some_big_size);
```

This is likely to not terminate. What's the problem?

pipe example (1)

```
int pipe_fd[2];
if (pipe(pipe_fd) < 0)
    handle_error(); /* e.g. out of file descriptors */
int read_fd = pipe_fd[0];
int write_fd = pipe_fd[1];
child_pid = fork();
if (child_pid  == 0) {
    /* in child process, write to pipe */
    close(read_fd);
    write_to_pipe(write_fd); /* function not shown */
    exit(EXIT_SUCCESS);
} else if (child_pid > 0) {
    /* in parent process, read from pipe */
    close(write_fd);
    read_from_pipe(read_fd); /* function not shown */
    waitpid(child_pid, NULL, 0);
    close(read_fd);
} else { /* fork error */ }
```

pipe example (1)

'standard' pattern with fork()

```
int pipe_fd[2];
if (pipe(pipe_fd) < 0)
    handle_error(); /* e.g. out of file descriptors */
int read_fd = pipe_fd[0];
int write_fd = pipe_fd[1];
child_pid = fork();
if (child_pid  == 0) {
    /* in child process, write to pipe */
    close(read_fd);
    write_to_pipe(write_fd); /* function not shown */
    exit(EXIT_SUCCESS);
} else if (child_pid > 0) {
    /* in parent process, read from pipe */
    close(write_fd);
    read_from_pipe(read_fd); /* function not shown */
    waitpid(child_pid, NULL, 0);
    close(read_fd);
} else { /* fork error */ }
```

pipe example (1)

```
int pipe_fd[2];
if (pipe(pipe_fd) < 0)
    handle_error(); /* e.g. out of file d
int read_fd = pipe_fd[0];
int write_fd = pipe_fd[1];
child_pid = fork();
if (child_pid  == 0) {
    /* in child process, write to pipe */
    close(read_fd);
    write_to_pipe(write_fd); /* function not shown */
    exit(EXIT_SUCCESS);
} else if (child_pid > 0) {
    /* in parent process, read from pipe */
    close(write_fd);
    read_from_pipe(read_fd); /* function not shown */
    waitpid(child_pid, NULL, 0);
    close(read_fd);
} else { /* fork error */ }
```

read() will not indicate end-of-file if write fd is open (any copy of it)

pipe example (1)

```
int pipe_fd[2];
if (pipe(pipe_fd) < 0)
    handle_error(); /* e.g. out of fil
int read_fd = pipe_fd[0];
int write_fd = pipe_fd[1];
child_pid = fork();
if (child_pid  == 0) {
    /* in child process, write to pipe */
    close(read_fd);
    write_to_pipe(write_fd); /* function not shown */
    exit(EXIT_SUCCESS);
} else if (child_pid > 0) {
    /* in parent process, read from pipe */
    close(write_fd);
    read_from_pipe(read_fd); /* function not shown */
    waitpid(child_pid, NULL, 0);
    close(read_fd);
} else { /* fork error */ }
```

have habit of closing
to avoid ‘leaking’ file descriptors
you can run out

exercise

```c
pid_t p = fork();
int pipe_fds[2];
pipe(pipe_fds);
if (p == 0) { /* child */
  close(pipe_fds[0]);
  char c = 'A';
  write(pipe_fds[1], &c, 1);
  exit(0);
} 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 it has a (subtle) bug.

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

```c
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(0);
}
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 of these 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(0);
}
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 of these 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

empirical evidence

```
  8 0
374 01
210 012
 30 0123
 12 01234
  3 012345
  1 0123456
  2 01234567
  1 012345678
359 0123456789
```

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

read/write

```
ssize_t read(int fd, void *buffer, size_t count);
ssize_t write(int fd, void *buffer, size_t count);
```

read/write up to *count* bytes to/from *buffer*

returns number of bytes read/written or -1 on error

- ssize_t is a signed integer type
- error code in `errno`

read returning 0 means end-of-file (*not an error*)

- can read/write less than requested (end of file, broken I/O device, …)

read'ing a fixed amount

```
ssize_t offset = 0;
const ssize_t amount_to_read = 1024;
char result[amount_to_read];
do {
    /* cast to void * optional in C */
    ssize_t amount_read =
        read(STDIN_FILENO,
             (void *) (result + offset),
             amount_to_read - offset);
    if (amount_read < 0) {
        perror("read"); /* print error message */
        ... /* abort??? */
    } else {
        offset += amount_read;
    }
} while (offset != amount_to_read && amount_read != 0);
```

partial reads

on regular file: read reads what you request

but otherwise: usually gives you what's known to be available

- after waiting for something to be available

partial reads

on regular file: read reads what you request

but otherwise: usually gives you what's known to be available

after waiting for something to be available

reading from network — what's been received

reading from keyboard — what's been typed

write example (with error checking)

```
const char *ptr = "Hello, World!\n";
ssize_t remaining = 14;
while (remaining > 0) {
    /* cast to void * optional in C */
    ssize_t amount_written = write(STDOUT_FILENO,
                                   ptr,
                                   remaining);
    if (amount_written < 0) {
        perror("write"); /* print error message */
        ... /* abort??? */
    } else {
        remaining -= amount_written;
        ptr += amount_written;
    }
}
```

partial writes

usually only happen on error or interruption

- but can request "non-blocking"
- (interruption: via *signal*)

usually: write waits until it completes

- = until remaining part fits in buffer in kernel
- does not mean data was sent on network, shown to user yet, etc.

pipe: closing?

if all write ends of pipe are closed

- can get end-of-file (read() returning 0) on read end
- exit()ing closes them

→ close write end when not using

generally: limited number of file descriptors per process

→ good habit to close file descriptors not being used

(but probably didn't matter for read end of pipes in example)

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/check a process's state (RUNNING, etc.) without 'acquiring' lock
- rule: 'release' lock when done

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/check a process's state (RUNNING, etc.) without 'acquiring' lock
- rule: 'release' lock when done

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

```
// 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)
};
```

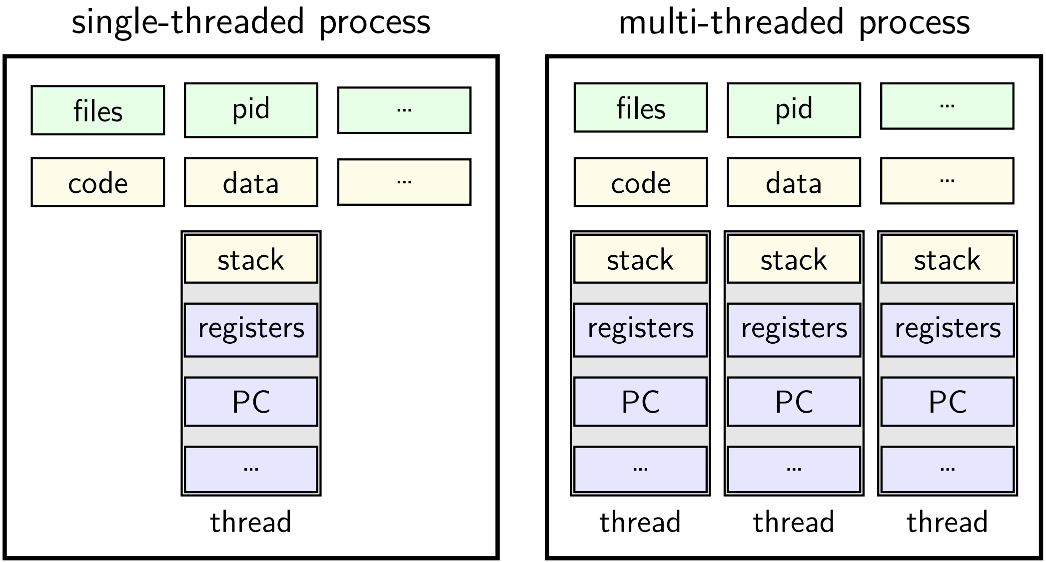
xv6 processes versus threads

xv6: one thread per process

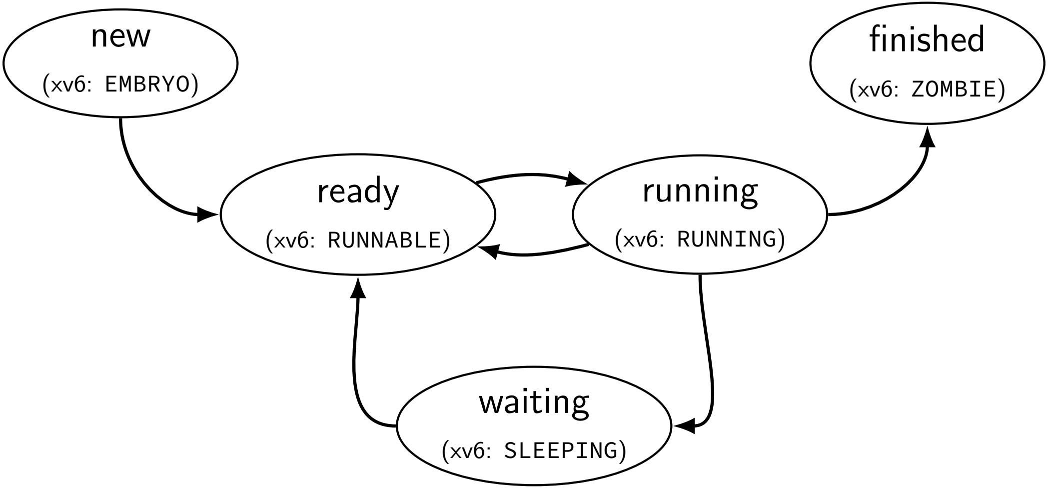
so part of the process control block
is really a *thread control block*

```
// 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)
};
```

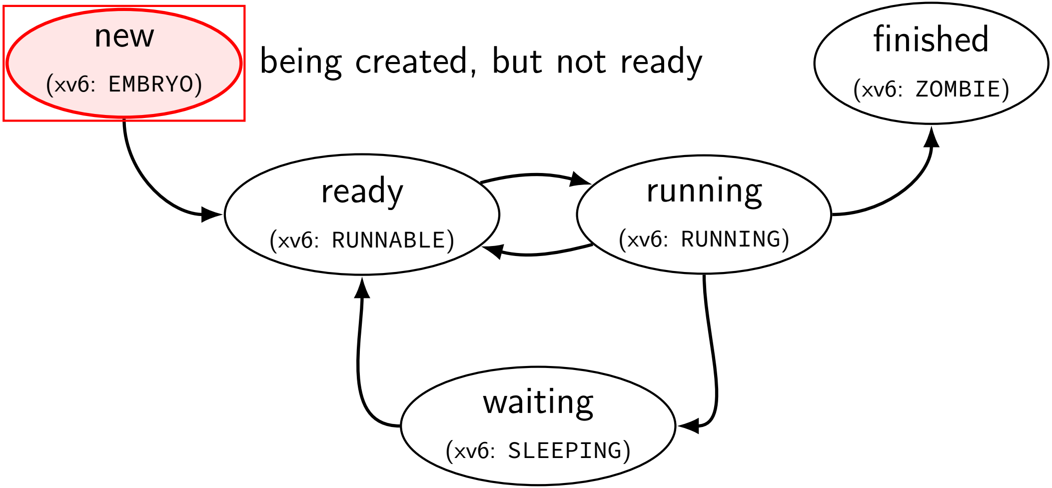
single and multithread processes



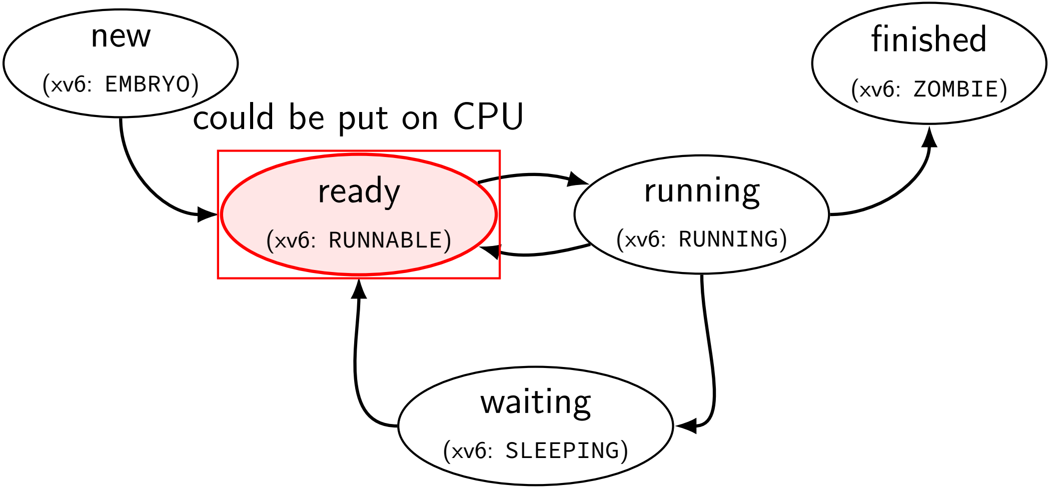
thread states



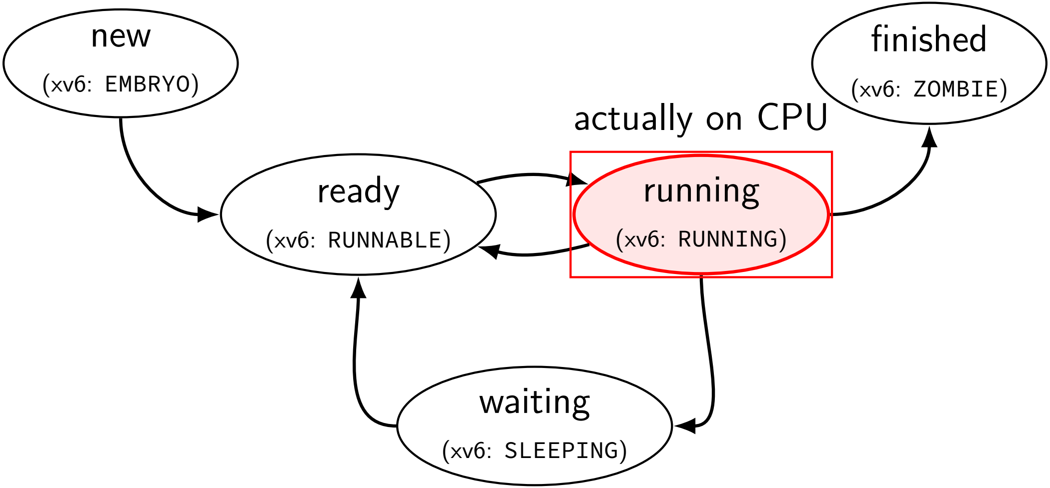
thread states



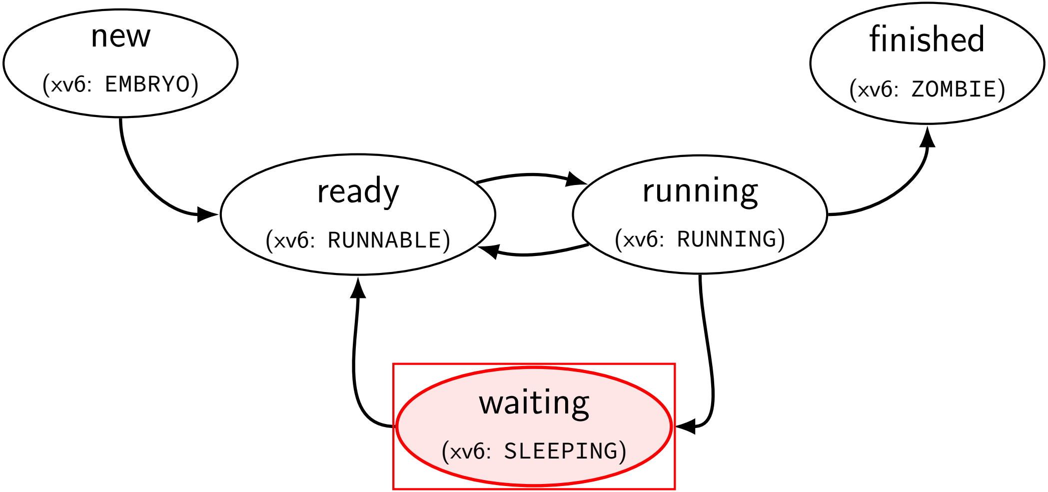
thread states



thread states

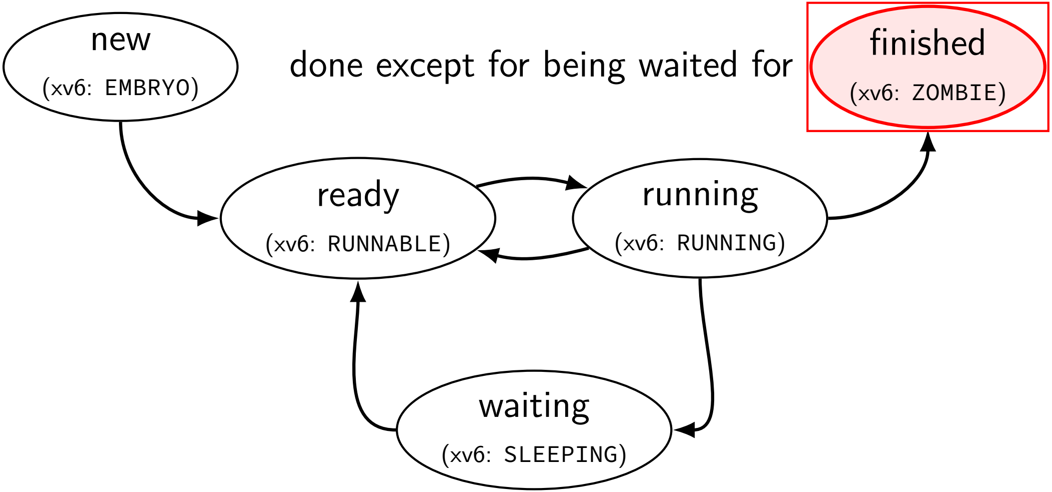


thread states

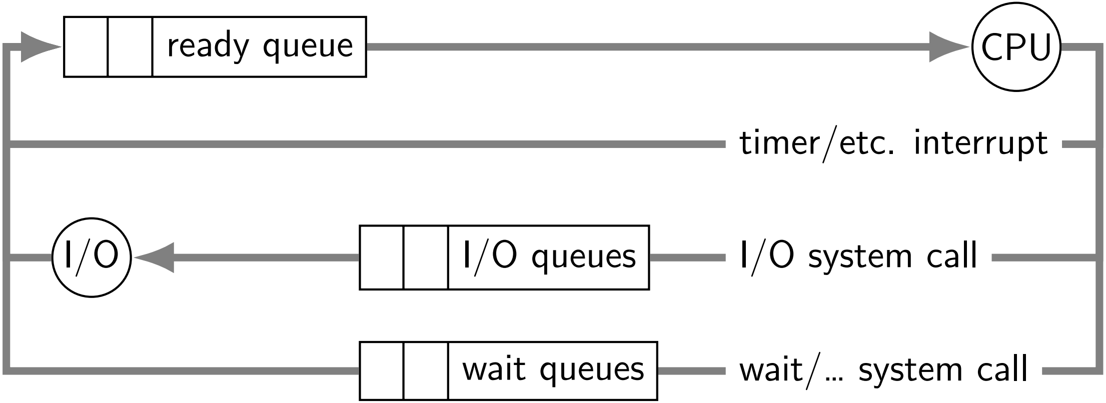


need external event to happen

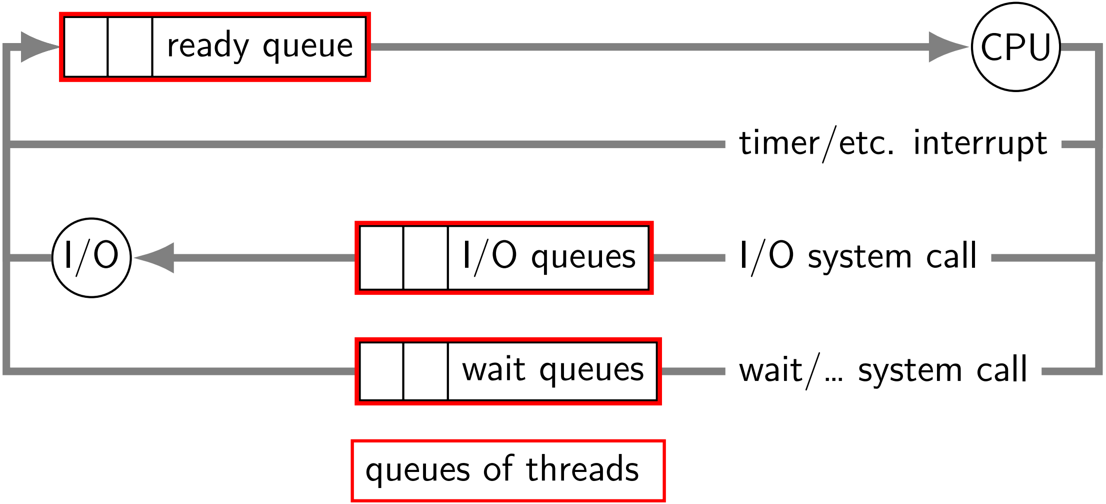
thread states



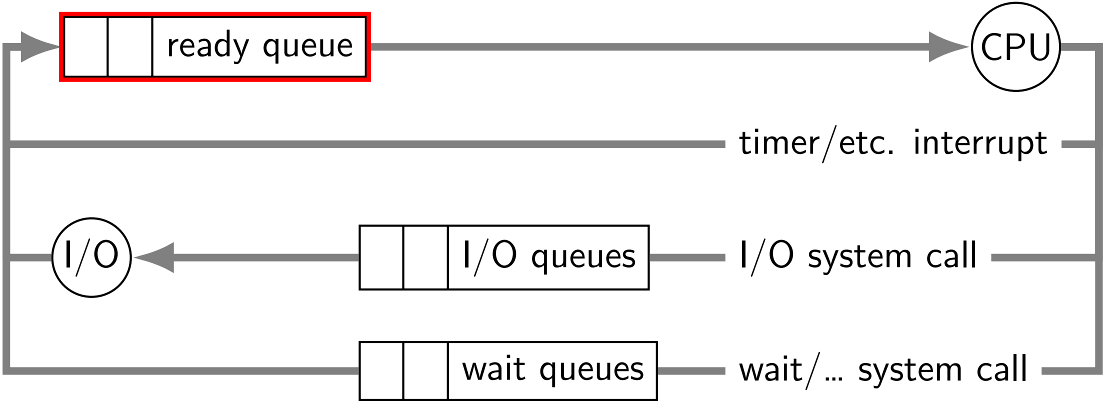
alternative view: queues



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

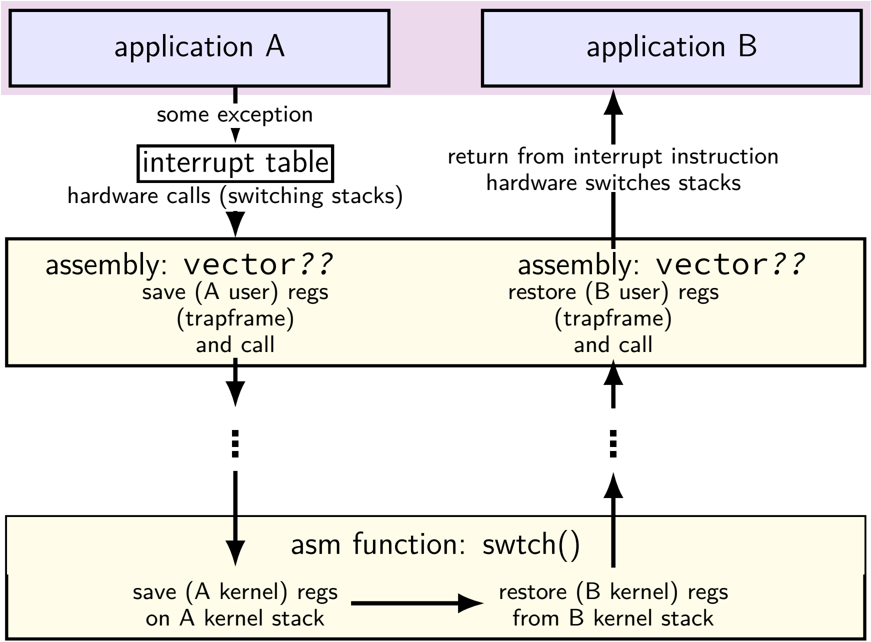
many ideas port to other scheduling problems

- especially simpler/less specialized policies

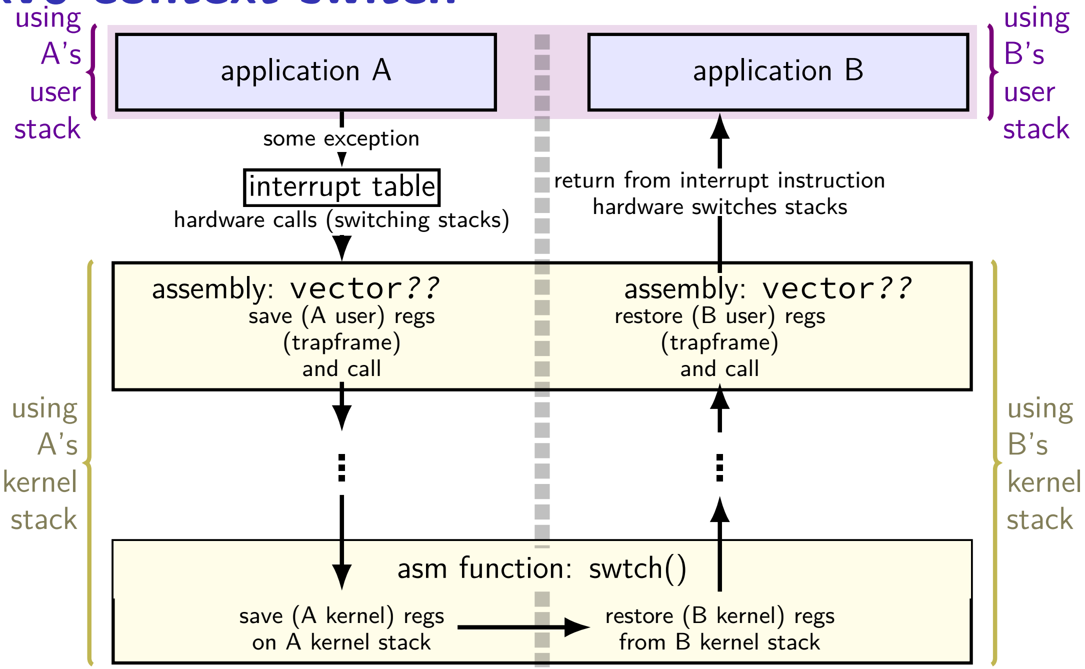
scheduling policy

scheduling policy = what to remove from queue

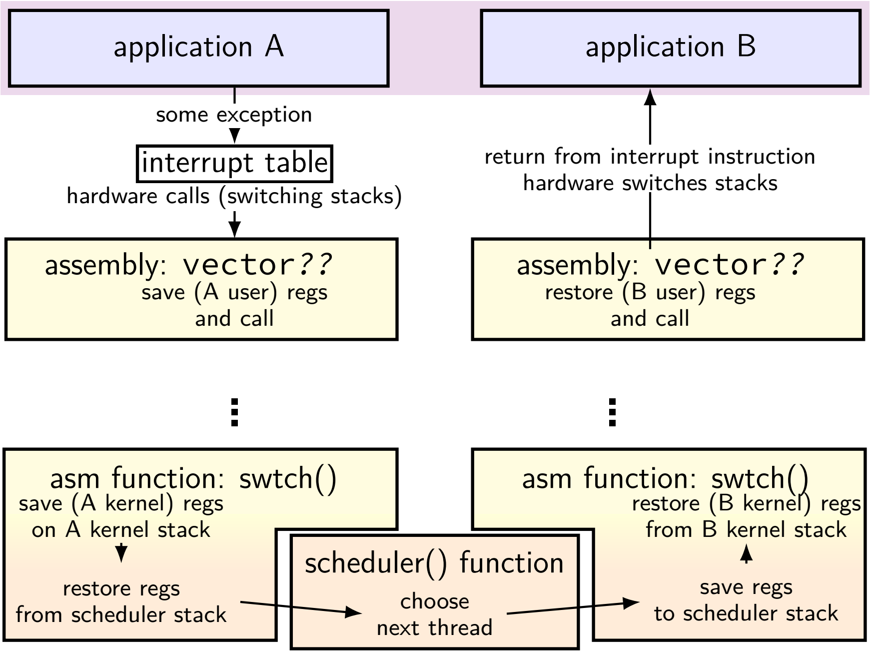
xv6 context switch



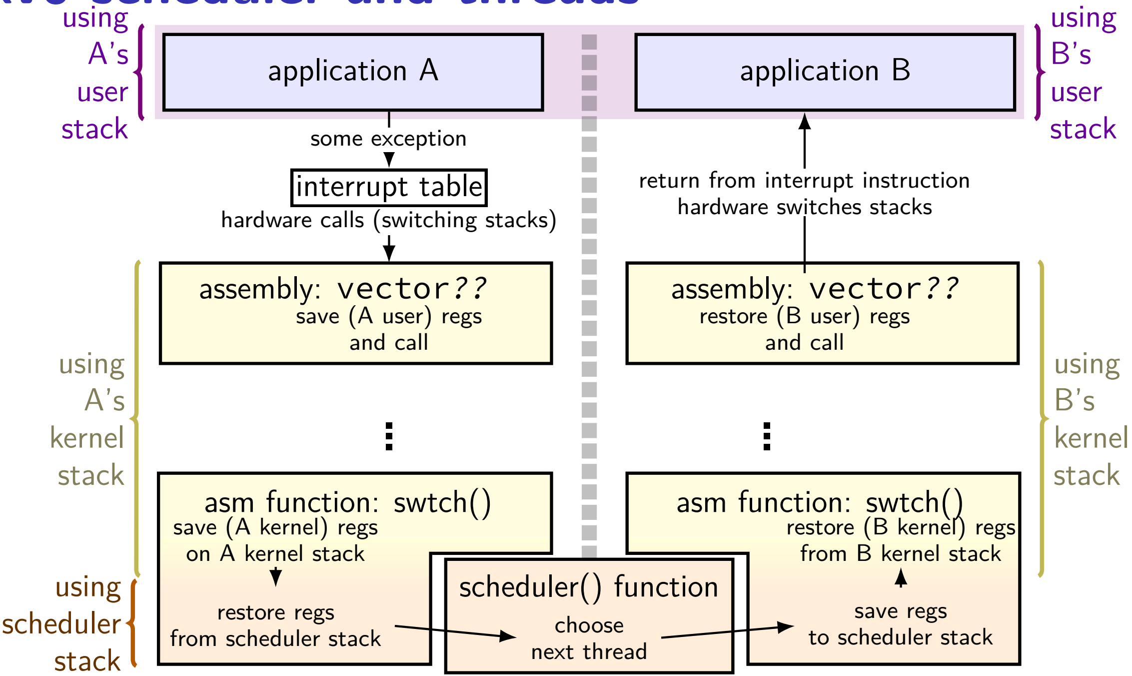
xv6 context switch



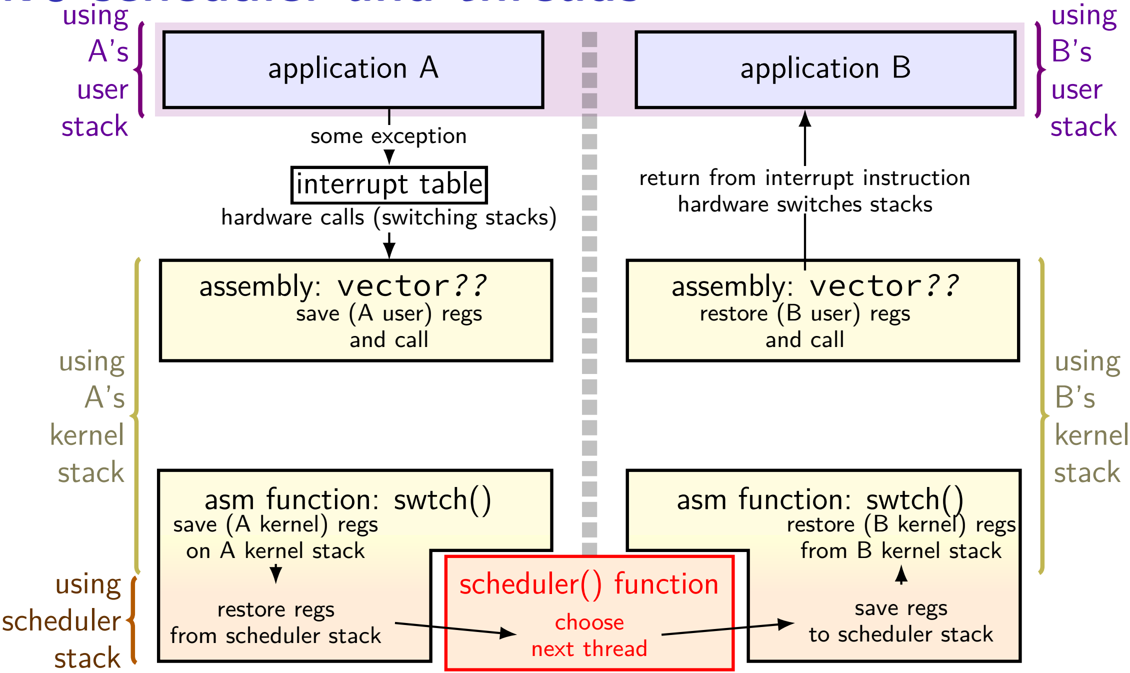
xv6 scheduler and threads



xv6 scheduler and threads



xv6 scheduler and threads



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);
  }
}
```

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);
  }
}
```

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

the xv6 scheduler (1)

```
void schedu
  struct p
  struct c
  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 */
      swtch(&(c->scheduler), p->context); /* ... */
      ... /* cleanup for process switch */
    }
    release(&ptable.lock);
  }
}
```

enable interrupts (`sti` is the x86 instruction)
makes sure keypresses, etc. will be handled

...(but acquiring the process table lock disables interrupts again)

the xv6 scheduler (1)

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

```
void sched
  struct p
  struct c
  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 */
      swtch(&(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);
  }
}
```

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

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;
      ... /* 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;
      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 chosen pr
      // to release ptable.l
      // before jumping back
      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: the actual switch

```
/* in scheduler(): */
      // Switch t
      // to relea
      // 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;
```

prepare: change address space, change process state

the xv6 scheduler: the actual switch

```
/* in scheduler(): */
      // Switch to
      // to releas
      // before ju
      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

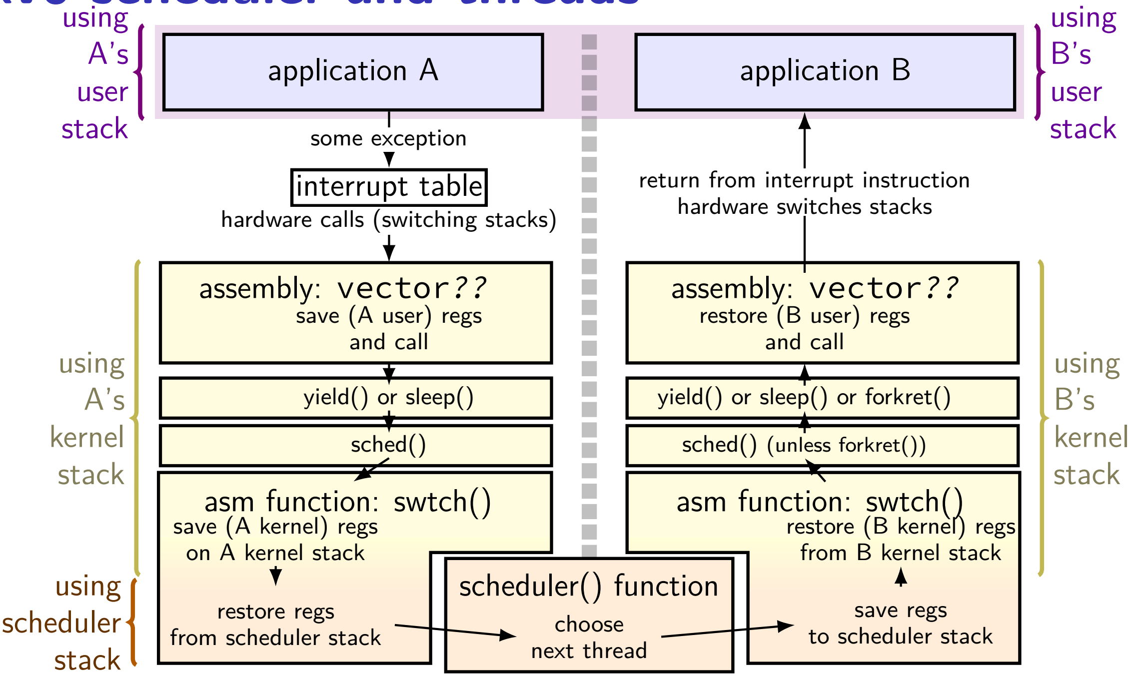
the xv6 scheduler: the actual switch

```
/* in scheduler(): */
      // Swi
      // to
      // bef
      c->pro
      switch
      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;
```

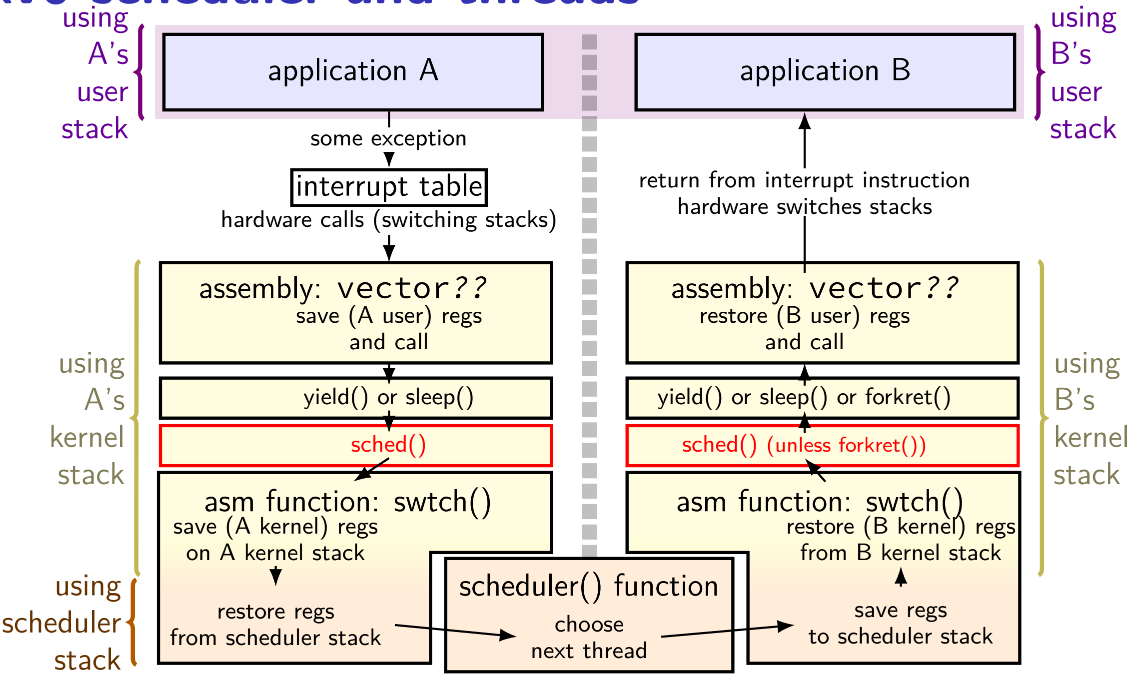
after we've run the process until it's done, we end up here

...so, change address space back away from user process

xv6 scheduler and threads



xv6 scheduler and threads



sched()

sched() — essentially just calls swtch()

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.

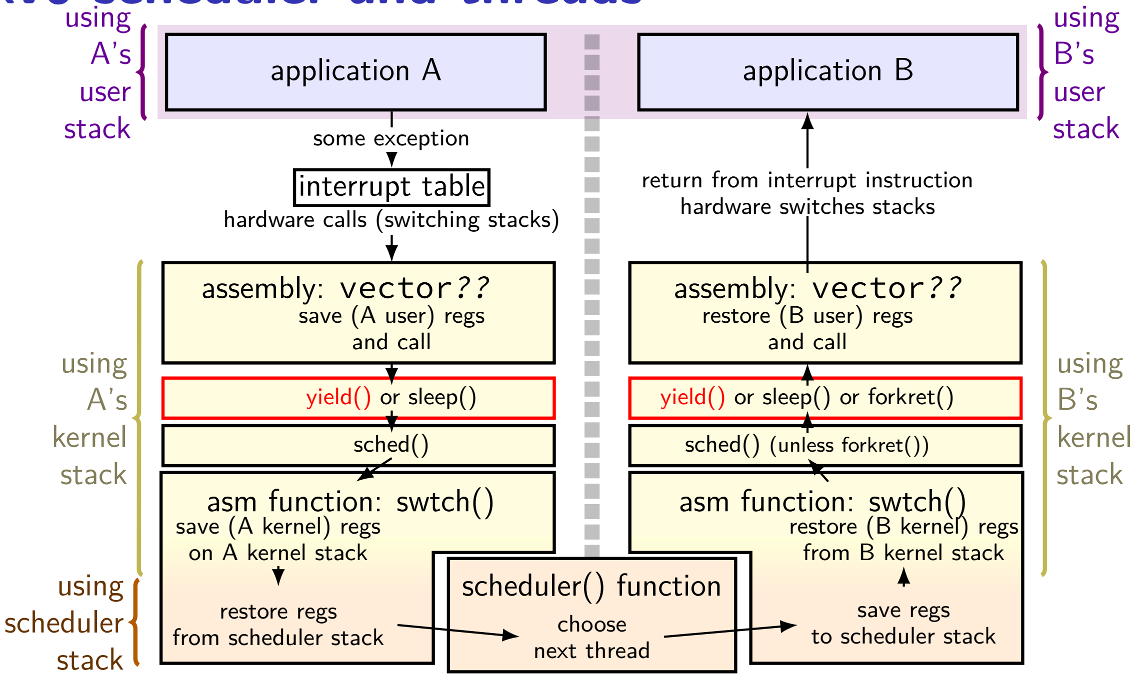
---

(4) scheduler thread switches back

invariant: process table lock held
invariant: current thread marked running

(5) release process table lock

xv6 scheduler and threads



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.

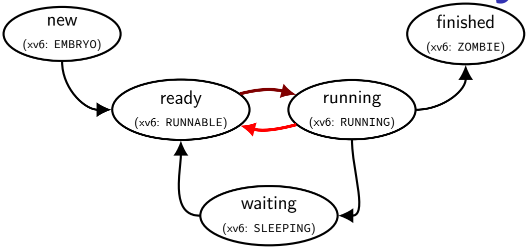
---

(4) scheduler thread switches back

invariant: process table lock held
invariant: current thread marked running

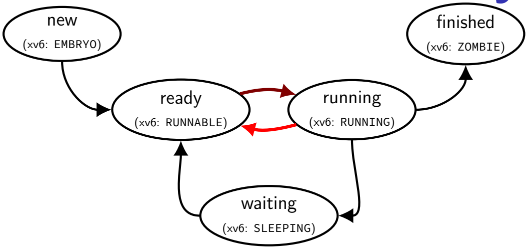
(5) release process table lock

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);
}
```

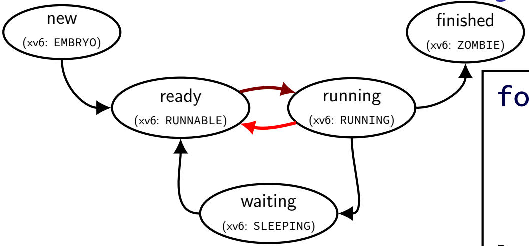
the xv6 scheduler: yield (timer int.)




```
/* 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.)

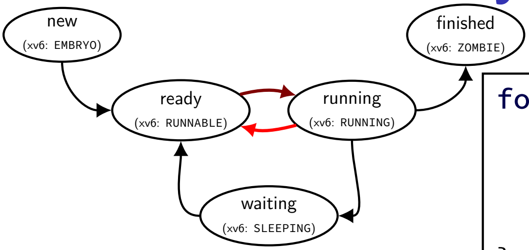


scheduler()

```
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.)



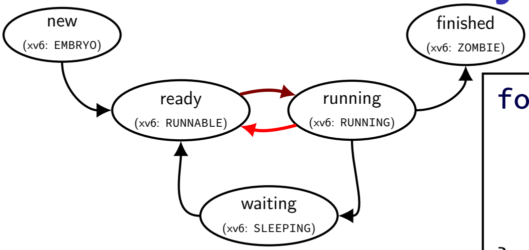
scheduler()

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

make sure we're the only one accessing the process list before changing our process's state / entering scheduler

```
/* function to
   used by the
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);
   ...
}
```

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

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

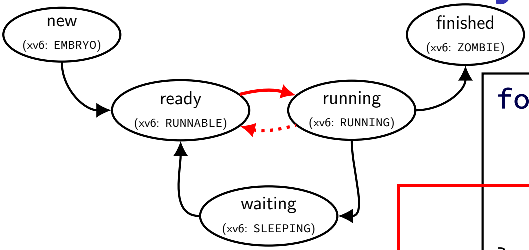
---

(4) scheduler thread switches back

invariant: process table lock held
invariant: current thread marked running

(5) release process table lock

the xv6 scheduler: yield (timer int.)

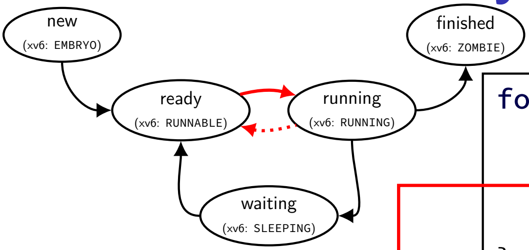


scheduler()

```
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.)

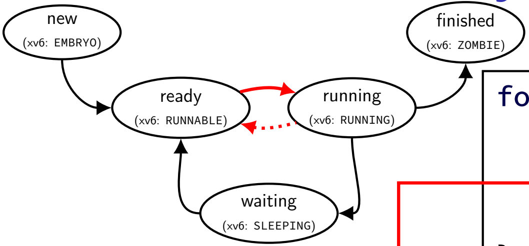


scheduler()

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

```
/* function to 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.)

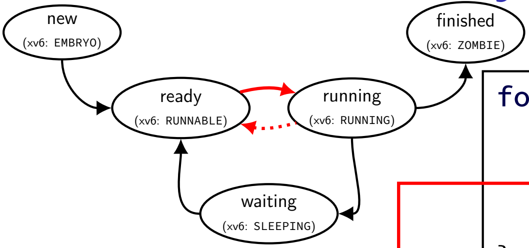


scheduler()

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

```
/* function to 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);
   ...
}
```

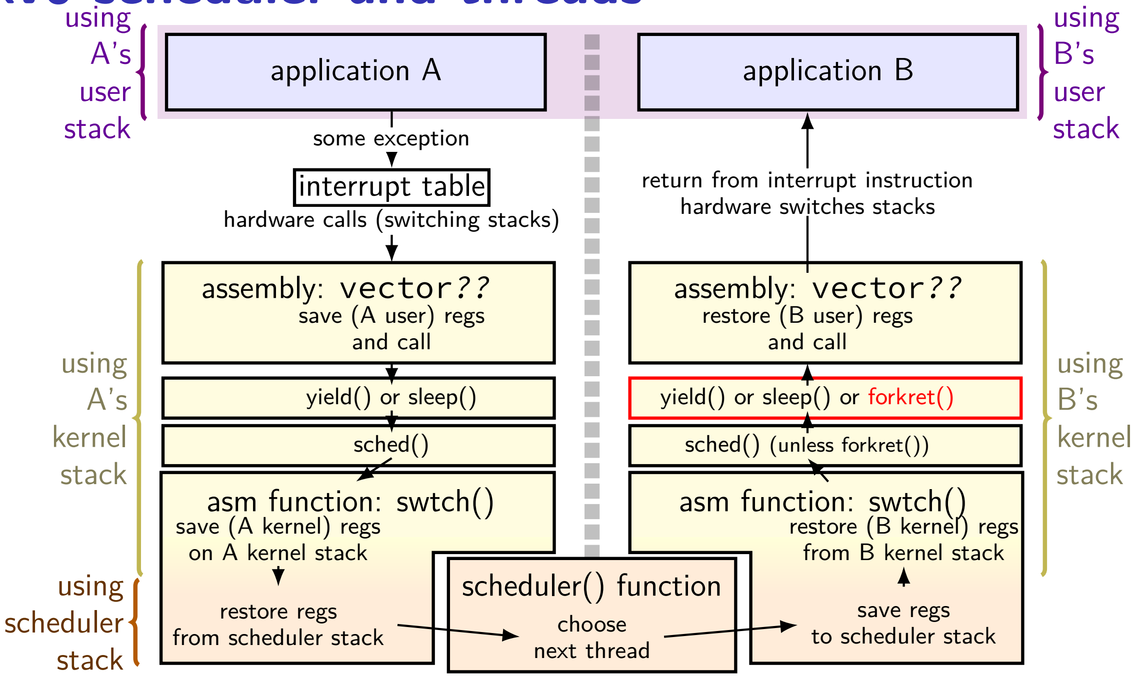
process table was 'locked'

unlock it before running user code

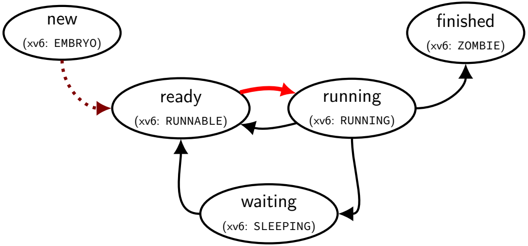
otherwise: timer interrupt/etc. won't work

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

xv6 scheduler and threads

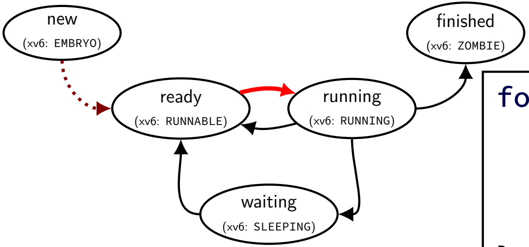


the xv6 scheduler: on process start




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

the xv6 scheduler: on process start

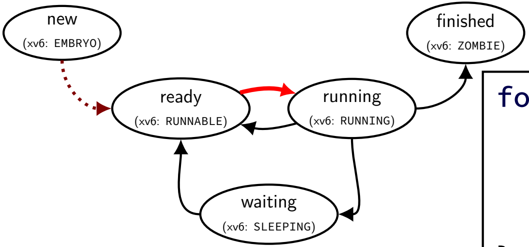


scheduler()

```
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



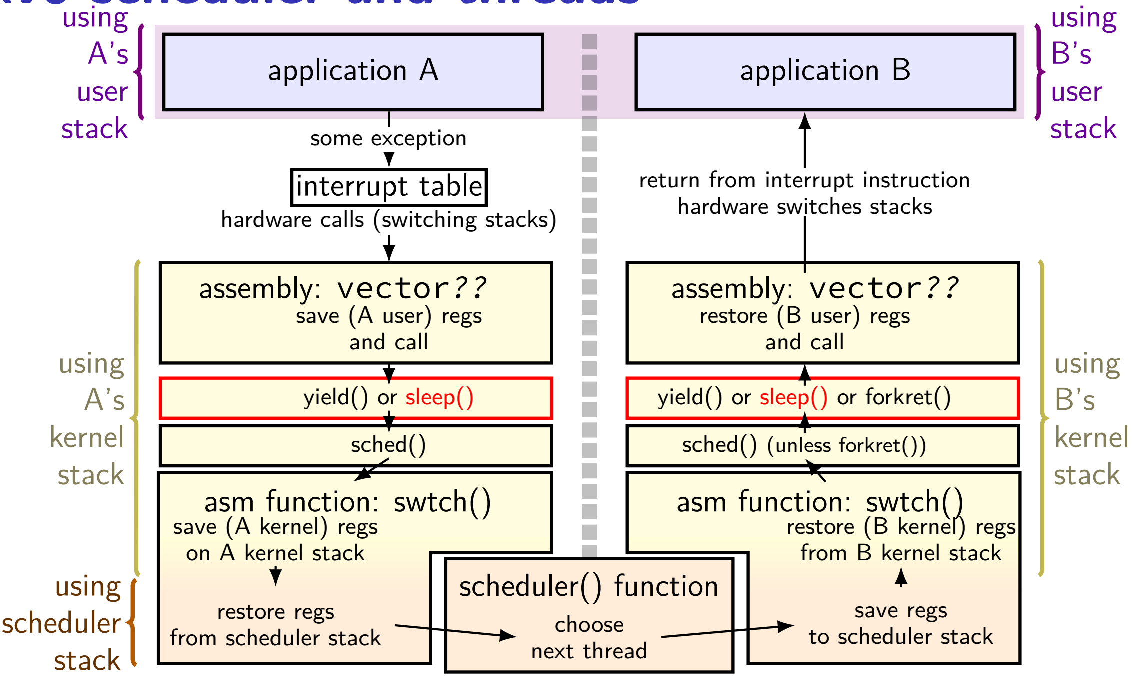
scheduler()

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

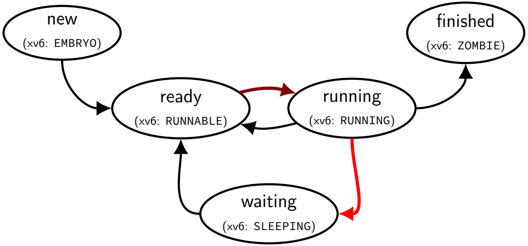
```
void forkret() {
   /* scheduler switches
   ...
   release(&ptable.lock);
   ...
}
```

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

xv6 scheduler and threads

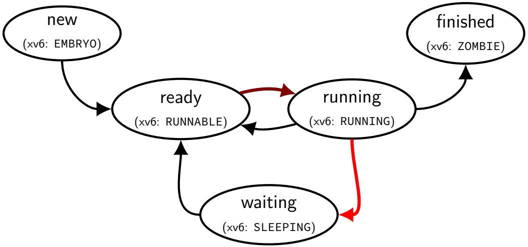


the xv6 scheduler: entering/leaving for sleep




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

  sched();

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

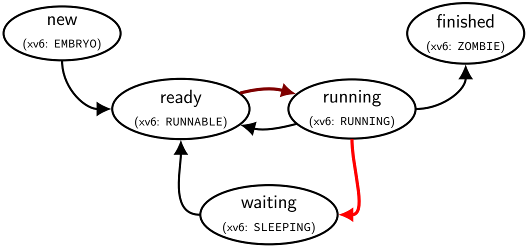
the xv6 scheduler: entering/leaving for sleep




```
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

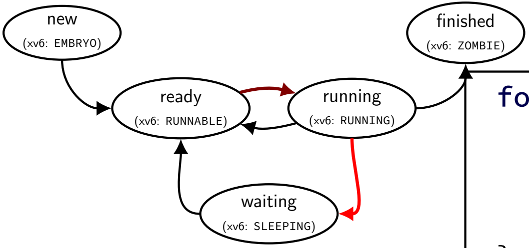
the xv6 scheduler: entering/leaving for sleep




```
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



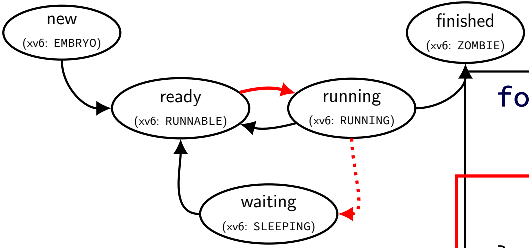
scheduler()

```
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;

    sched();

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

...and switch to the scheduler infinite loop

the xv6 scheduler: entering/leaving for sleep

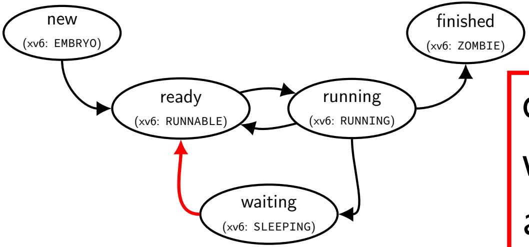


scheduler()

```
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;

  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

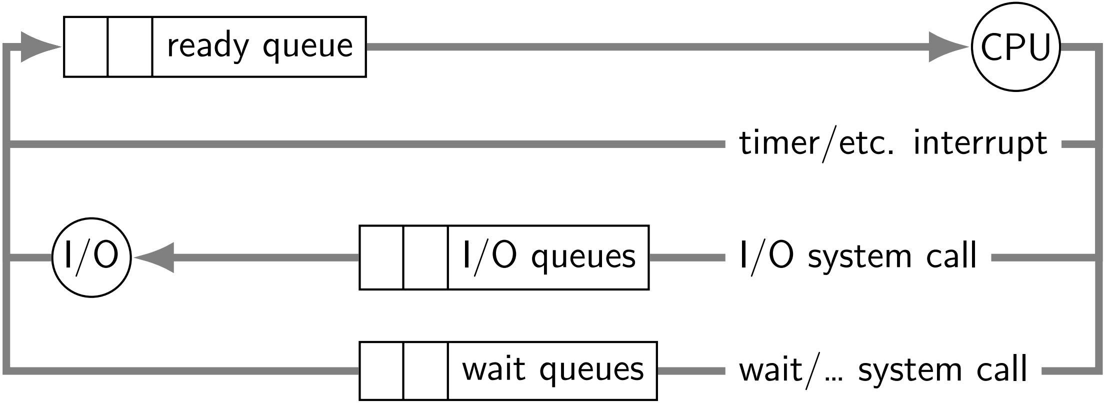
welcome to 1970:

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)

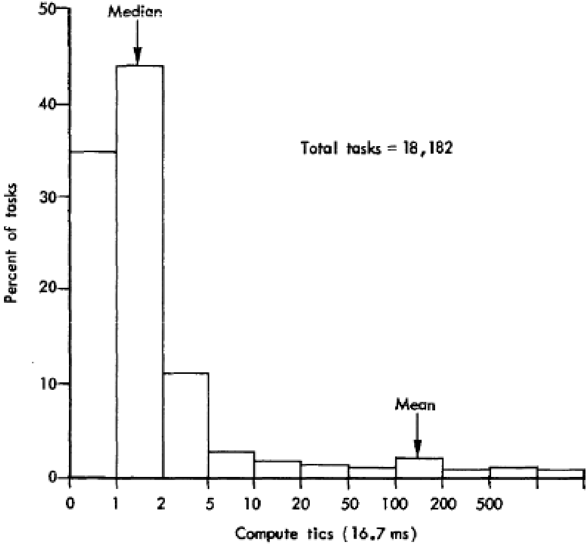
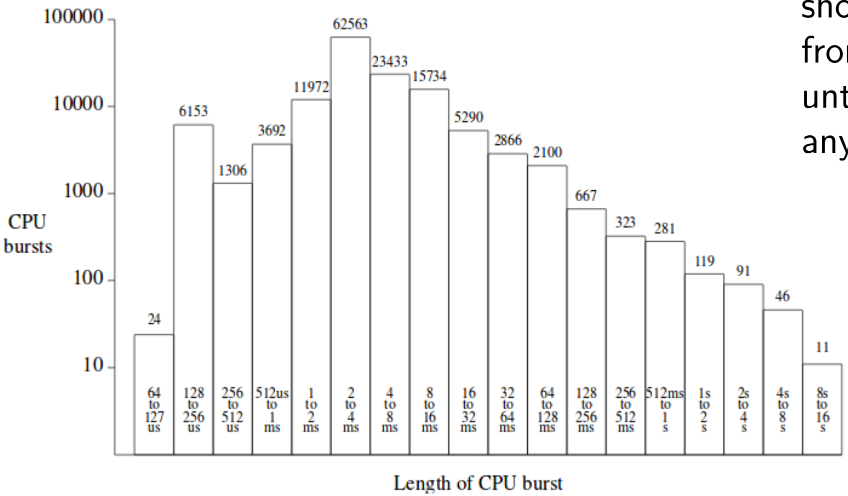


Figure 11—Compute time per task

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

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

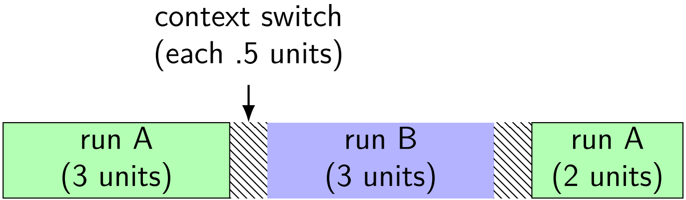
scheduling CPU bursts? (what we'll mostly deal with)

- turnaround time ≈ 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 → program requests CPU sooner

scheduling batch program on cluster?

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

throughput



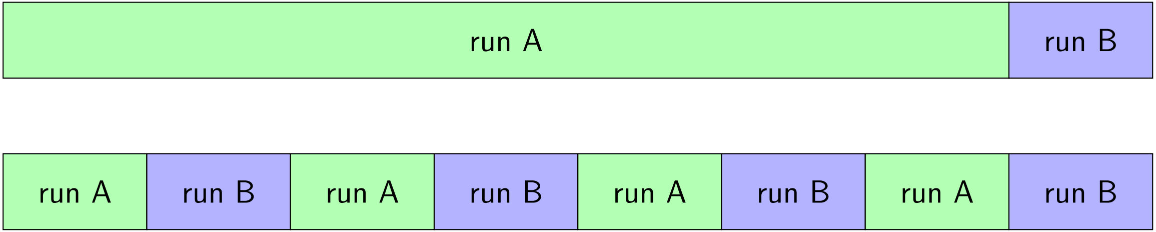
throughput: useful work done per unit time

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

also other considerations:

- time lost due to cold caches
- time lost not starting I/O early as possible
- ...

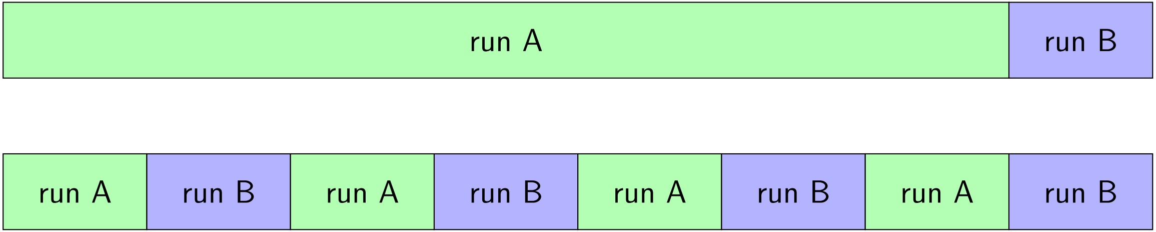
fairness



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?

backup slides