POSIX API (finish) / Scheduling intro

last time

shells: program for users to run other programs

files: open before use, read/write bytes, explicit close

file descriptor: index into per-process table

fork: copy table
- same index refers to same open file
- not deep copy — shared offsets, etc.

dup2: assign one entry to another

close: deallocate table entry

pipe: create pair of connected file descriptors

shell assignment corrections

make archive versus make submit

phrasing on outputting exit statuses

- output must be in order of pipeline
- don't care how you actually wait for commands (only that you do)

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

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

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

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

next topic: processes and scheduling

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

```
// 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*

```c
// 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)
};
```

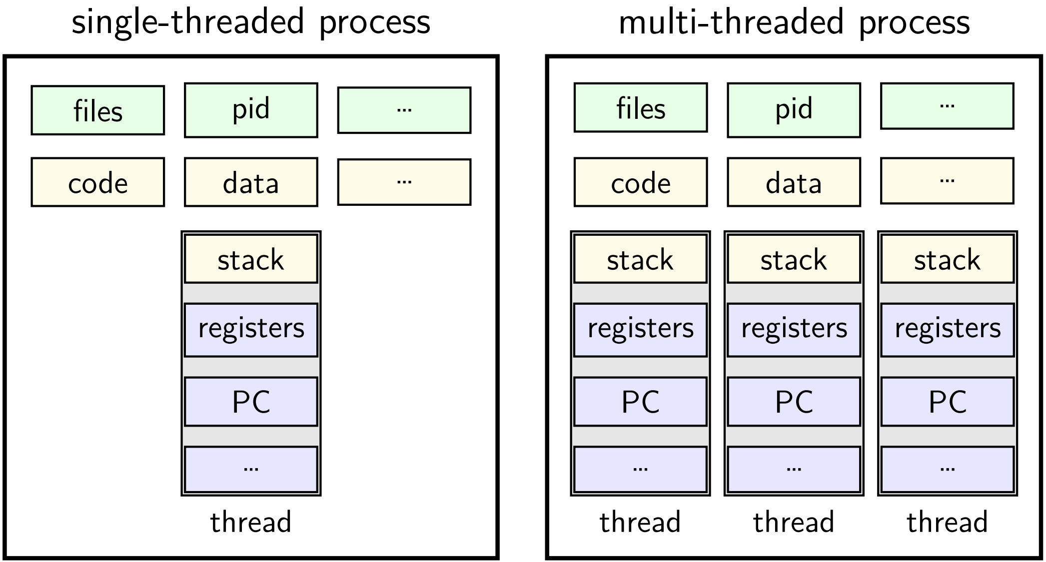
xv6 processes versus threads

xv6: one thread per process

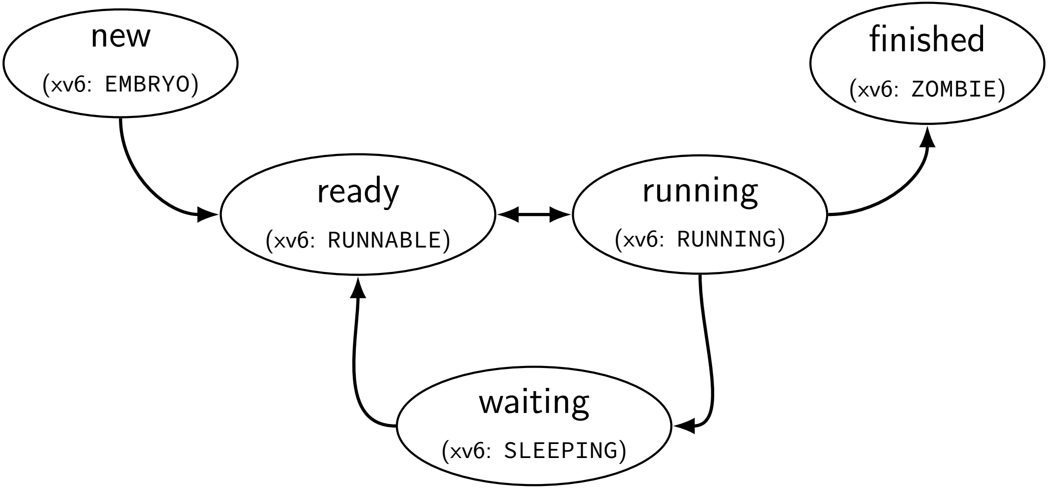
so part of the process control block
is really a *thread control block*

```
// 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)
};
```

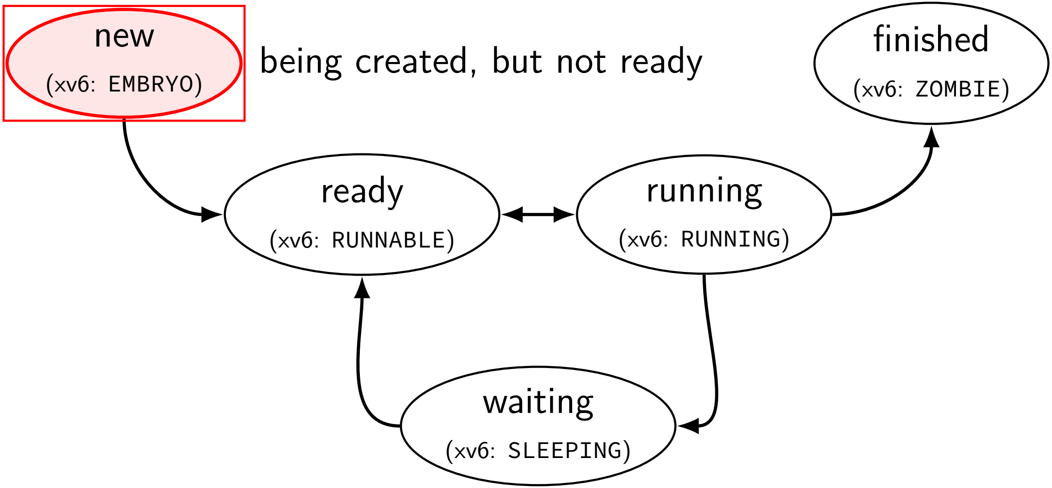
single and multithread processes



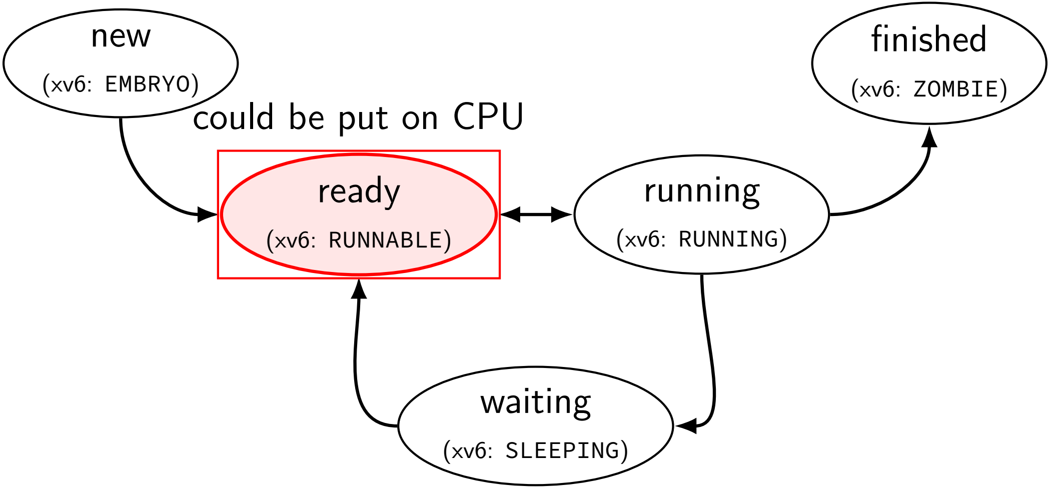
thread states



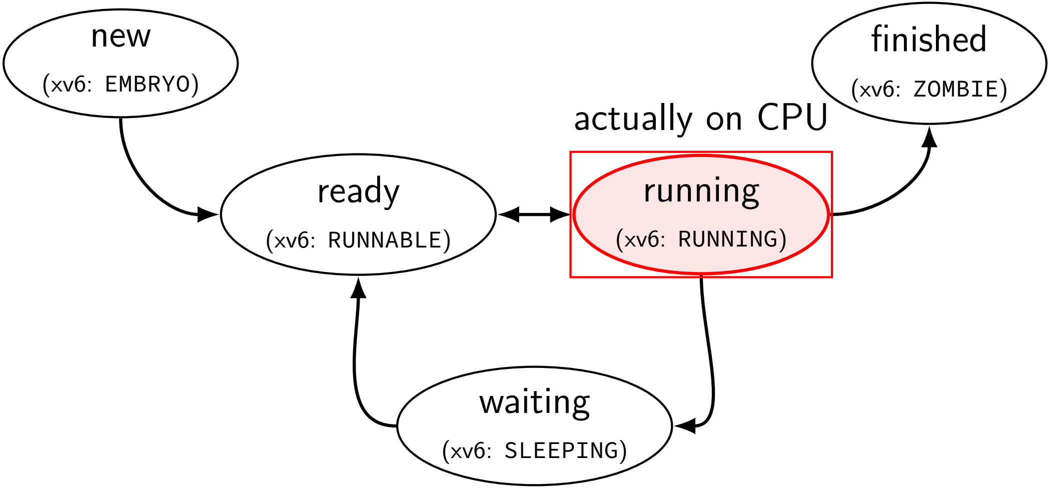
thread states



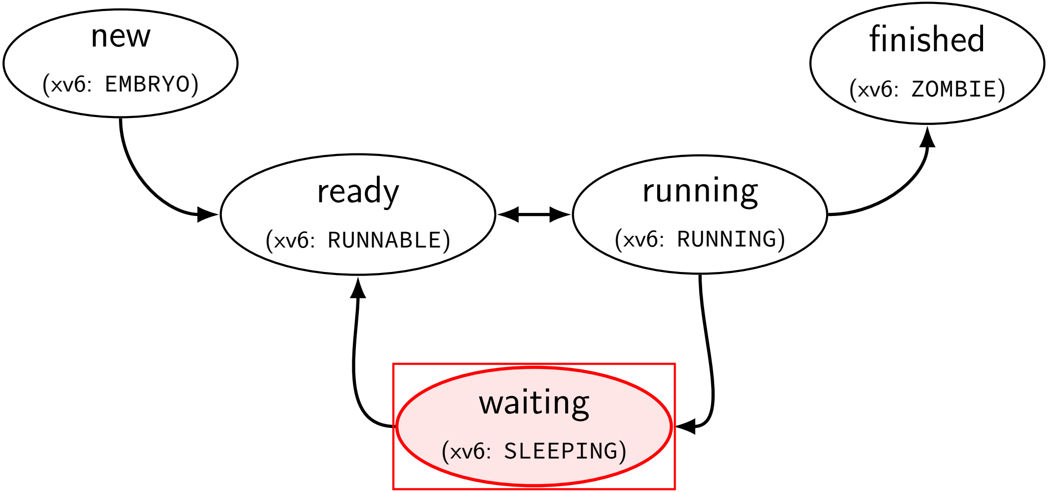
thread states



thread states

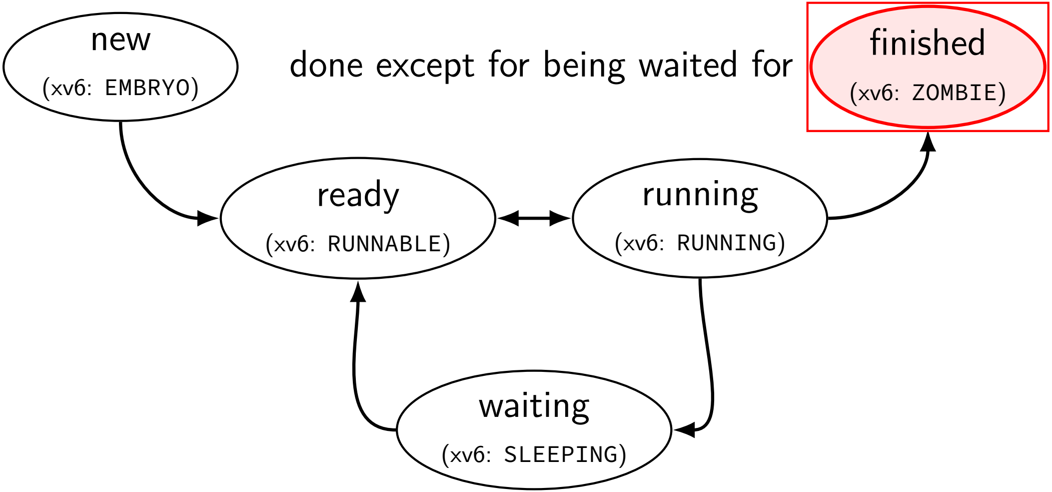


thread states

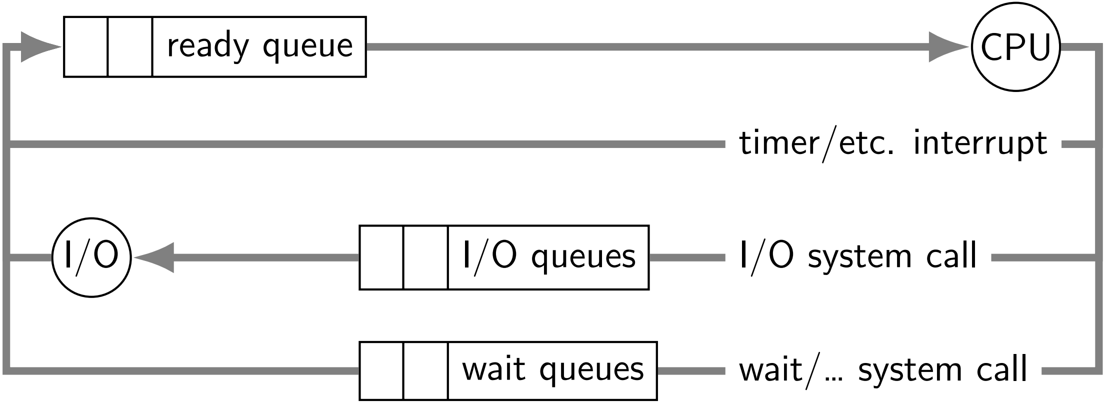


need external event to happen

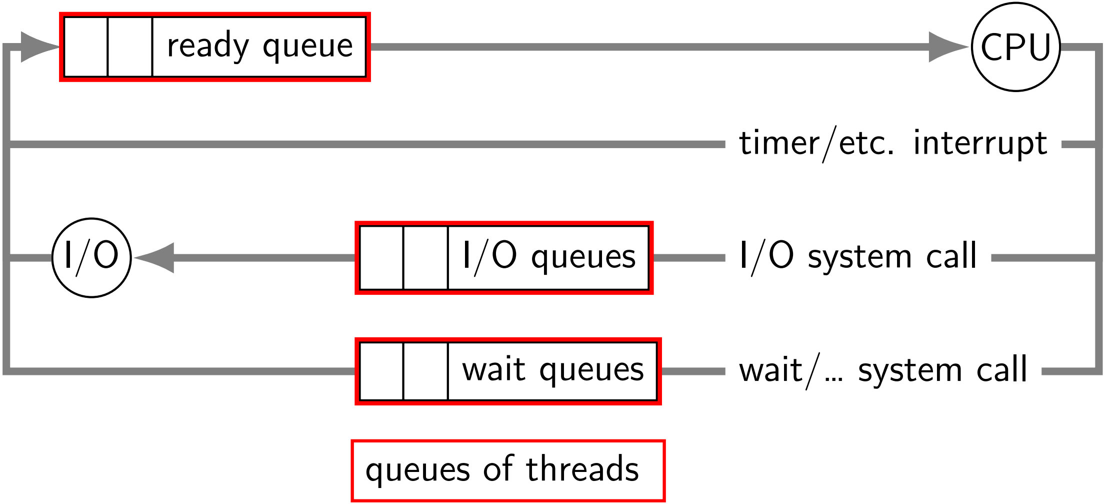
thread states



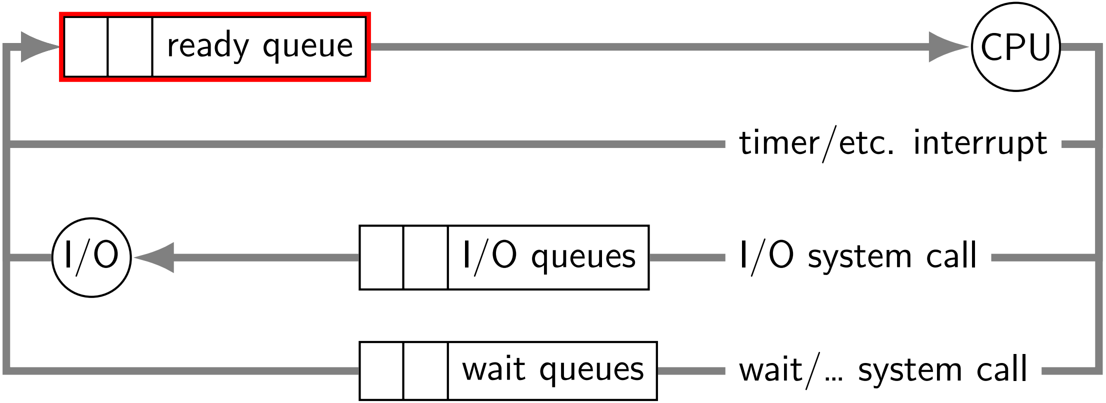
alternative view: queues



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

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)

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

  for(;;){
    // Enable interrupts o
    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);
  }
}
```

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)

```
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)

```
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);
  }
}
```

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;
      ... /* 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

```
/* 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

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()

```
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()

```
p->state = RUNNING;
swtch(&(c->scheduler), p->context);
```

```
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 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: going from/to scheduler

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: going from/to scheduler

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: going from/to scheduler

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);
}
```

the xv6 scheduler: going from/to scheduler

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: going from/to scheduler

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: going from/to scheduler

scheduler()

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

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

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

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

scheduler()

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

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);
  ...
```

scheduler()

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

the xv6 scheduler: SLEEPING to RUNNABLE

```
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 code choices

separate scheduler thread

- switch to scheduler, scheduler switches to next thread
- other OSes: call scheduler, switch directly to next thread
- pro: simpler code organization (keep scheduler state in local variables)
- con: slower — extra register saving and restoring

scan process list to find sleeping/waiting threads

- other OSes: separate lists of waiting/sleeping threads
- pro: simpler: no code to maintian queues of threads
- con: slower to find sleeping/waiting threads
- con: much, much slower if many waiting threads

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