CSCI 350 Project 3
Summer 2020

MFQ Scheduler

Kivilcim Cumbul

Objectives

1. Implement a basic non-preemptive MFQ (Multilevel Feedback Queue)
   - Changes in `proc.c, proc.h, p_stat.h, trap.c`
2. Add a new system call to get information about a running process
   - `getpinfo(pid)`
3. Design tests to demonstrate the correctness of your scheduler.
   - `test1.c,test2.c,test3.c`
4. Create timeline graphs

Before Starting…

Make sure you choose 1 CPU in your VM

And in your Makefile, replace CPUS := 2 with CPUS := 1

Part 1

In this project, you'll implement a simplified Multi-level Feedback Queue (MFQ) scheduler in xv6.

- proc.c, proc.h, pstat.h, trap.c

Chapter 5 of the xv6 manual.

The default and only scheduling policy in xv6 is round-robin, and we will change it!

Note: I will explain more high level this time because everyone can implement this differently

pstat.h

```
#ifndef _PSTAT_H_
#define _PSTAT_H_

/* NSCHEDSTATS is the number of sched_stat_t slots per process. The scheduler fills in the slots
as it schedules processes to record information about scheduling */

#define NSCHEDSTATS 1500

/*
 * responsible for recording the scheduling state per process at a particular tick
 * e.g. a process can have an array of sched_stat_t's, with each of them holding the info
 * of a scheduling round of the process
 */
struct sched_stat_t
{
  int start_tick; //the number of ticks when this process is scheduled
  int duration;   //number of ticks the process is running before it gives up the CPU
  int priority;   //the priority of the process when it's scheduled

  //you may add more fields for debugging purposes
};

#endif
```

Record scheduling stats
And use them again

It fills array for each process
and records info of its scheduling

DONT FORGET in proc.c — #include "pstat.h"

proc.h

Inside proc structure

```
int times[3]          // number of times each process was scheduled at each of 3
                      // priority queues

int ticks[3]          // number of ticks each process used the last time it was
                      // scheduled in each priority queue
                      // cannot be greater than the time-slice for each queue

uint wait_time;       // number of ticks each RUNNABLE process waited in the lowest
                      // priority queue
```

wait…

```
struct sched_stat_t
{
  int start_tick; //the number of ticks when this process is scheduled
  int duration;   //number of ticks the process is running before it gives up the CPU
  int priority;   //the priority of the process when it's scheduled

  //you may add more fields for debugging purposes
};

#endif
```

```
int times[3]          // number of times each process was scheduled at each of 3
                      // priority queues

int ticks[3]          // number of ticks each process used the last time it was
                      // scheduled in each priority queue
                      // cannot be greater than the time-slice for each queue

uint wait_time;       // number of ticks each RUNNABLE process waited in the lowest
                      // priority queue
```

We can put these either proc struct or stat struct

The only important thing is recording!

How do we call them?

In proc structure

```
proc->times[0]
proc->times[1]
proc->times[2]

proc->ticks0]
proc->ticks[1]
proc->ticks[2]

proc->wait_time
```

In sched_stat_t

```
proc->sched_stat_t[pid].start_tick
proc->sched_stat_t[pid].duration
proc->sched_stat_t[pid].priority
```

proc.h additional variables

You might define as much variables as you want into proc or sched_stats_t.
Some examples for proc:

```
int ticks;                  //number of timer ticks the process has run for
int total_ticks;           //total number of timer ticks the process has run for
struct sched_stat_t sched_stats[NSCHEDSTATS]; // schedule stats for each tick
int num_stats_used;        // count to the end of the array
```

Dont forget this!

Because array is predefined with the value we defined we have keep track where is the real end

Let's start scheduler! — Part 1

It is much easier to deal with fixed-sized arrays in xv6 than linked-lists. For simplicity, we recommend that you use arrays to represent each priority level (queue).

```
struct proc* q0[NPROC];
```
→ Level 1 — First → 1 tick

```
struct proc* q1[NPROC];
```
→ Level 2 → 2 ticks

```
struct proc* q2[NPROC];
```
→ Level 3 — Last → 8 ticks

proc.h

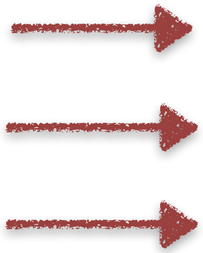
```
extern struct proc* q0[NPROC];

extern struct proc* q1[NPROC];

extern struct proc* q2[NPROC];
```

proc.c

```
struct proc* q0[NPROC];

struct proc* q1[NPROC];

struct proc* q2[NPROC];
```

Missing something?

```
struct proc* q0[NPROC];
struct proc* q1[NPROC];
struct proc* q2[NPROC];
```



Level 1 — First

Level 2

Level 3 — Last

COUNTERS!!

Can we use something else?

Sure!
Tail, Front, Capacity etc.

proc.h

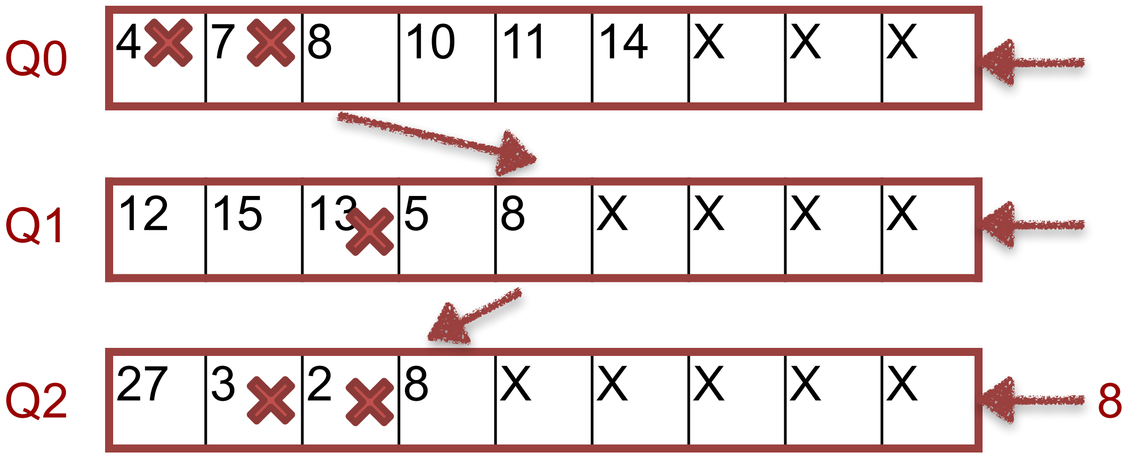
```
extern int c0;

extern int c1;

extern int c2;
```

proc.c

```
int c0=-1;

int c1=-1;

int c2=-1;
```

Visualization

Choose whoever is runnable (8 in this case)



1 - Numbered from 0 (highest) down to 2 (lowest).

2 - When ticks, highest priority RUNNABLE process is scheduled to run.

3 - If more than one process on the same level, then use round robin fashion.

4 - Q0 —> 1 timer tick. Q1 —> 2 timer ticks Q2 —> 8 timer ticks.

5- Can get out before CPU increase the tick —>CHEAT

6- When a new process arrives, it should start at priority 0 (highest priority). Place this process at the end of the highest priority queue

7 - At priorities 0, and 1, after a process consumes its time-slice it should be downgraded one priority level. Whenever a process is moved to a lower priority level, it should be placed at the end of the queue.

8 - If a process wakes up after voluntarily giving up the CPU it stays at the same priority level. Place this process at the end of its queue; it should not preempt a process with the same priority. —> CHEAT

9- Your scheduler should never preempt a lower priority process if a higher priority process is available to run.

In proc.c

In allocproc()

Allocate Process

Look in the process table for an UNUSED proc.
If found, change state to EMBRYO and initialize state required to run in the kernel.
Otherwise return 0.

This means we need to check our queues and if there is a unused etc. remove that process

```
After found -> Remove from queue:
if (p->pid > 0):
  for (i=0;(i<c0); i++)
    if p == q0[i]
      Delete q0[i]
      Shift all left

  for (i=0;(i<c1); i++)
    …
  for (i=0;(i<c2); i++)
    …
```

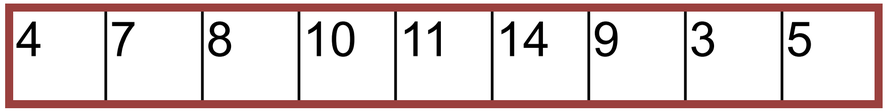
Do the same thing for q1 and q2

```
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

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

  release(&ptable.lock);
  return 0;

found:
```

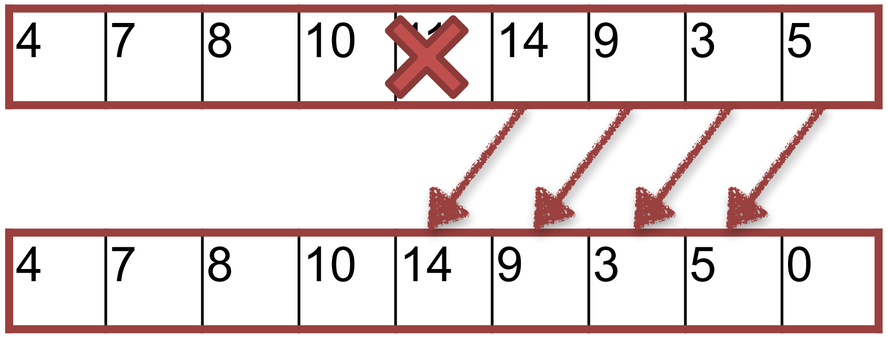
In proc.c

Shift all left?

Lets say we are looking for pid = 11



Not need to delete just shifting is enough



Last element?

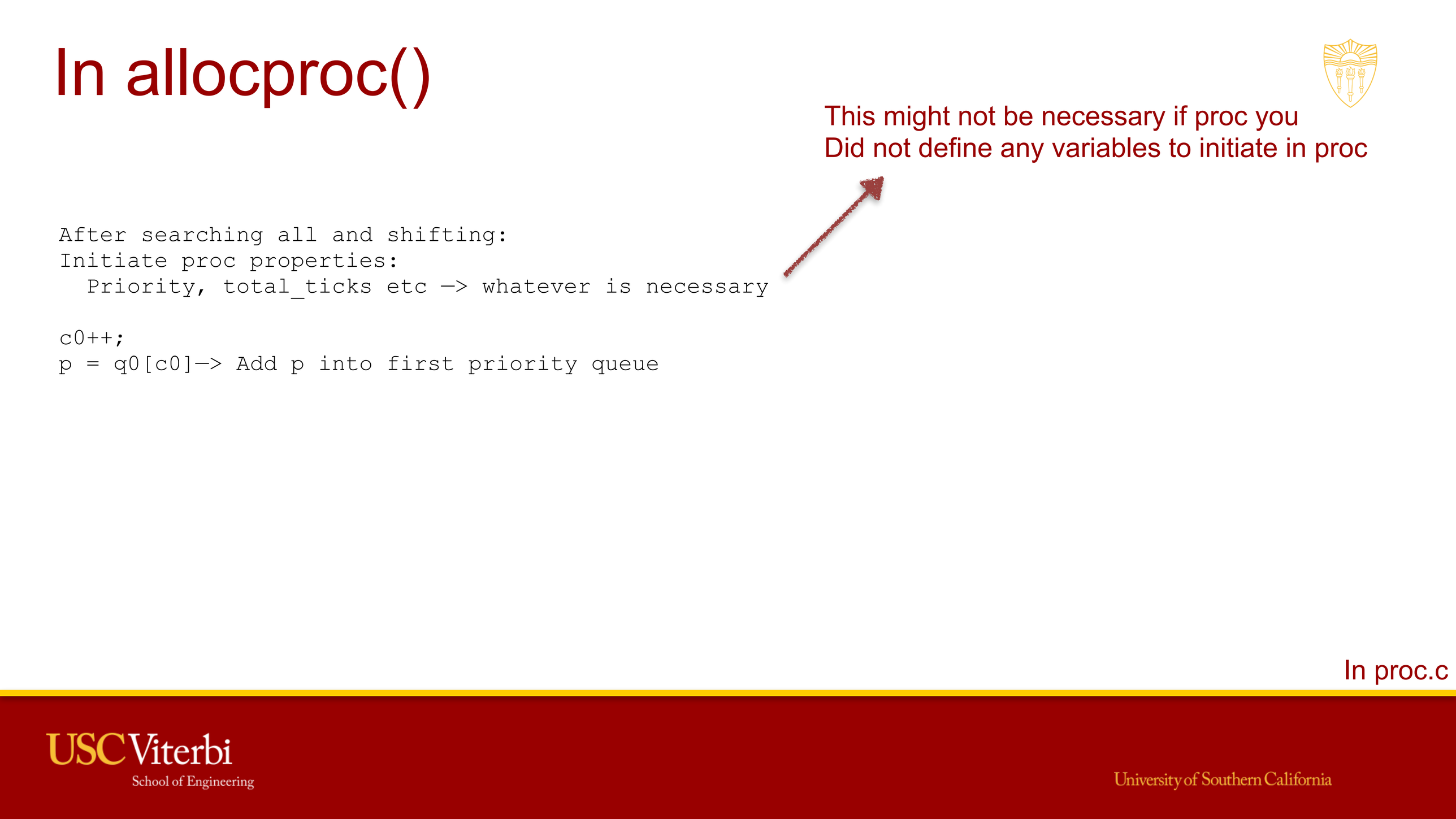
```
q0[c0]=0 ; (c0,c1,c2)
```

Also counter —> the tail of the array changes:

```
counter-- ; (c0,c1,c2)
```

In proc.c

In allocproc()



This might not be necessary if proc you
Did not define any variables to initiate in proc

```
After searching all and shifting:
Initiate proc properties:
  Priority, total_ticks etc –> whatever is necessary

c0++;
p = q0[c0]–> Add p into first priority queue
```

In proc.c

In userinit()

User initialize

Everything starts with userinit system call

```
Reset schedule stats because we
just start the first process here
```

Reset here

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

  p = allocproc();

  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S
```

In proc.c

From, xv6 manual pg 24

Code: creating the first process

Now the kernel runs within its own address space, we look at how the kernel creates user-level processes and ensures strong isolation between the kernel and user-level processes, and between processes themselves.

After `main` initializes several devices and subsystems, it creates the first process by calling `userinit` (1239). `Userinit`'s first action is to call `allocproc`. The job of `al-`

scheduler()

Userinit calls it at the beginning

Now that the first process's state is prepared, it is time to run it. After `main` calls `userinit`, `mpmain` calls `scheduler` to start running processes (1267). `Scheduler` (2708) looks for a process with `p->state` set to `RUNNABLE`, and there's only one: `initproc`. It sets the per-cpu variable `proc` to the process it found and calls `switchuvm` to tell the hardware to start using the target process's page table (1868). Changing page tables

From, xv6 manual pg 26

`scheduler` now sets `p->state` to `RUNNING` and calls `swtch` (2958) to perform a context switch to the target process's kernel thread. `swtch` saves the current registers and loads the saved registers of the target kernel thread (`proc->context`) into the x86 hardware registers, including the stack pointer and instruction pointer. The current context is not a process but rather a special per-cpu scheduler context, so `scheduler` tells `swtch` to save the current hardware registers in per-cpu storage (`cpu->scheduler`) rather than in any process's kernel thread context. We'll examine `swtch` in more detail in Chapter 5. The final `ret` instruction (2977) pops the target process's `%eip` from the stack, finishing the context switch. Now the processor is running on the kernel stack of process `p`.

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

  }
}
```

In proc.c

swtch()

Per-CPU process scheduler. Each CPU calls scheduler() after setting itself up.

Scheduler never returns. It loops, doing:

- choose a process to run
- swtch to start running that process
- eventually that process transfers control via swtch back to the scheduler.

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

  }
}
```

Process

```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  if(p == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");

  if(lk != &ptable.lock){  //DOC: sleeplock0
    acquire(&ptable.lock);  //DOC: sleeplock1
    release(lk);
  }
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;

  sched();

  // Give up the CPU for one scheduling round.
  void
  yield(void)
  {
    acquire(&ptable.lock);  //DOC: yieldlock
    myproc()->state = RUNNABLE;
    sched();
    release(&ptable.lock);
  }
```

exit() —> sched()

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

In proc.c

switchkvm/switchuvm

From, xv6 manual:

The **u** in `switchuvm` stands for User. The **k** in `switchkvm` stands for Kernel.

User mode can call system call

When changing protection levels from user to kernel mode, the kernel shouldn't use the stack of the user process, because it may not be valid. The user process may be malicious or contain an error that causes the user `%esp` to contain an address that is not part of the process's user memory. Xv6 programs the x86 hardware to perform a stack switch on a trap by setting up a task segment descriptor through which the hardware loads a stack segment selector and a new value for `%esp`. The function `switchuvm` (1873) stores the address of the top of the kernel stack of the user process into the task segment descriptor.

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

  }
}
```

Add process to cpu

Before running

After running

In proc.c

How to modify?

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

  }
}
```

Same

For loop finds next available running thread
But it does not work like that in MFQ

```
If q0 is not empty:
  Loop to find RUNNABLE process in the q0 array
      if q0[i] —>state != RUNNABLE
        Continue
  p = q0[i]   Found first runnable
  c->proc = p
  switchuvm(p)
  p->state = RUNNING

  WRITE STATS —> start_tick and priority   start_tick = ticks

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

  Duration = ticks - start
  p->num_stats_used++;
  p->times[0]++;
  p->ticks[0] = duration;
  Change total_ticks in proc if you have
```

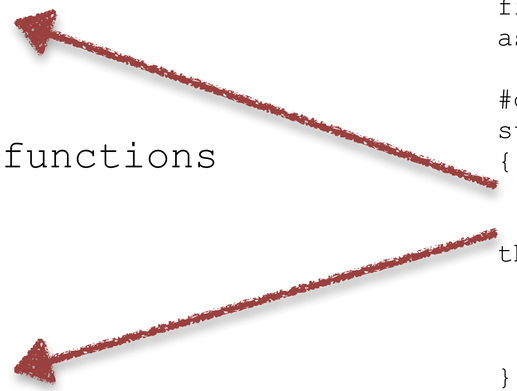
In proc.c

How to think about stats?

Arbitrary program in any language

How to time?

```
Start time = x

Do some work
Go to different functions
Come back

End time = y

Duration = y-x
```



```
#ifndef _PSTAT_H_
#define _PSTAT_H_

/* NSCHEDSTATS is the number of sched_stat_t slots per process. The scheduler
fills in the slots
as it schedules processes to record information about scheduling */

#define NSCHEDSTATS 1500
struct sched_stat_t
{
  int start_tick; //the number of ticks when this process is scheduled
  int duration;   //number of ticks the process is running before it gives up
the CPU
  int priority;   //the priority of the process when it's scheduled

  //you may add more fields for debugging purposes
};

#endif
```

In proc.c

tick, proc->tick, proc->tick[0]

tick comes from trap.c

proc->tick — extra : (increase in trap.c)
number of timer ticks the process has run for

proc->tick[] array:
number of ticks each process used the last time it was scheduled in each priority queue

proc->times[] array:
number of times each process was scheduled at each of 3

```
**********************
name= CPUintensive, pid= 5
wait time= 0
ticks= {1, 2, 8}
times= {1, 1, 3}
**********************
start=1, duration=1, priority=0
start=2, duration=0, priority=1
start=2, duration=0, priority=1
start=2, duration=0, priority=1
start=2, duration=0, priority=1
start=2, duration=0, priority=1
start=2, duration=0, priority=1
start=2, duration=2, priority=1
start=4, duration=8, priority=2
start=12, duration=8, priority=2
```

```
If q0 is not empty:
  Loop to find RUNNABLE process in the q0 array
      if q0[i] ->state != RUNNABLE
        Continue
  p = q0[i]   Found first runnable
  c->proc = p
  switchuvm(p)
  p->state = RUNNING

  WRITE STATS -> start_tick and priority   start_tick = ticks

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

  Duration = ticks - start
  p->num_stats_used++;
  p->times[0]++;
  p->ticks[0] = duration;
  Change total_ticks if you have
```

In proc.c

How to modify? cont'd

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

  }
}
```

Second half

```
If q0 is not empty:
  … Previous slide

  if(p->ticks >= 1)
    p->ticks = 0;
    copy proc to lower priority queue
    increase c1 count
    … other changes necessary
    q1[c1] = p; // put at the end of the q1
    q0[i] = 0;// delete proc from q0
    Shift all left from i  Dont forget counter-- and last element of array

  boost()
  c->proc = 0; Reset cpu
```

In proc.c

How to modify?

Q1

```
If q1 is not empty:
  Loop to find RUNNABLE process in the q1 array
      if q1[i] —>state != RUNNABLE
        Continue
  p = q1[i]
  c->proc = p
  switchuvm(p)
  p->state = RUNNING

  WRITE STATS —> start_tick and priority   start_tick = ticks

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

  Duration = ticks - start
  p->num_stats_used++;
  p->times[1]++;
  p->ticks[1] = duration;
  Change total_ticks in proc if you have
```

In proc.c

How to modify? cont'd

Q1

Second half

```
If q1 is not empty:
   … Previous slide

   if(p->ticks >= 2)
     p->ticks = 0;
     copy proc to lower priority queue
     increase c2 count
     … other changes necessary
     q2[c2] = p; // put at the end of the q2
     q1[i] = 0;// delete proc from q1
     Shift all left from i  Dont forget counter-- and last element of array

   boost()
   c->proc = 0; Reset cpu
```

In proc.c

How to modify?

Q2

```
If q2 is not empty:
  Loop to find RUNNABLE process in the q2 array
      if q2[i] —>state != RUNNABLE
        Continue
  p = q2[i]
  c->proc = p
  switchuvm(p)
  p->state = RUNNING

  WRITE STATS —> start_tick and priority   start_tick = ticks

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

  Duration = ticks - start
  p->num_stats_used++;
  p->times[2]++;
  p->ticks[2] = duration;
  Change total_ticks in proc if you have
```

In proc.c

How to modify? cont'd

Q2

Second half

```
If q1 is not empty:
   … Previous slide
   boost()
   if(p->ticks >=8)
      p->ticks = 0;
      increase c2 count
      move process to end of its own queue
      Shift left
      q2[c2] = p
   c->proc = 0;  Reset cpu
```

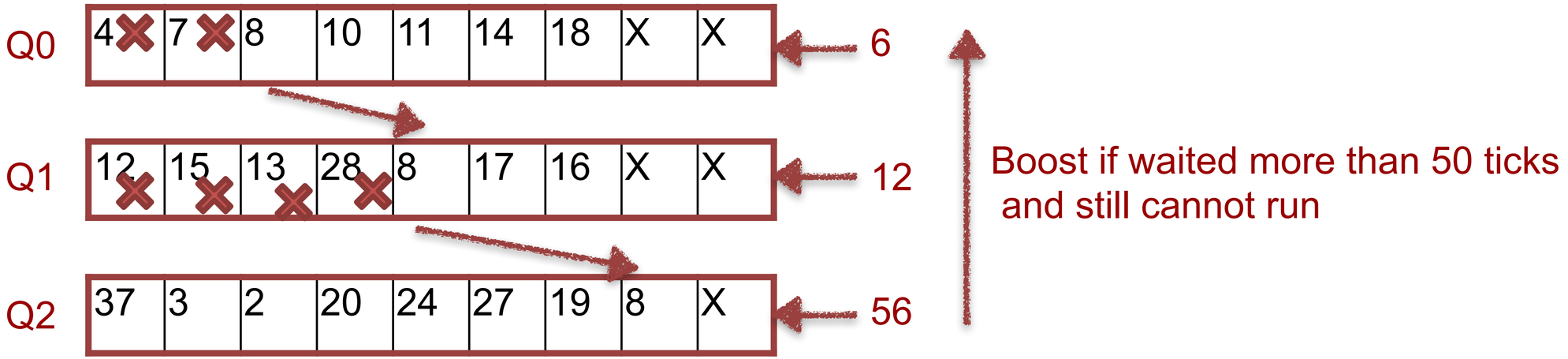
In proc.c

boost()

Task 1-10: You need to implement the priority boosting mechanism which will be used to increase a priority of a process that has not been scheduled in a while. The goal is to **avoid starvation**, which happens when a process never receives CPU time because higher-priority processes keep arriving.
After a **RUNNABLE process** has been waiting in the lowest priority queue for **50 ticks or more**, move the process to the **end of the highest priority queue**. This method of priority boosting is called **aging**.

Assume all became runnable after 8 gets down to Q2

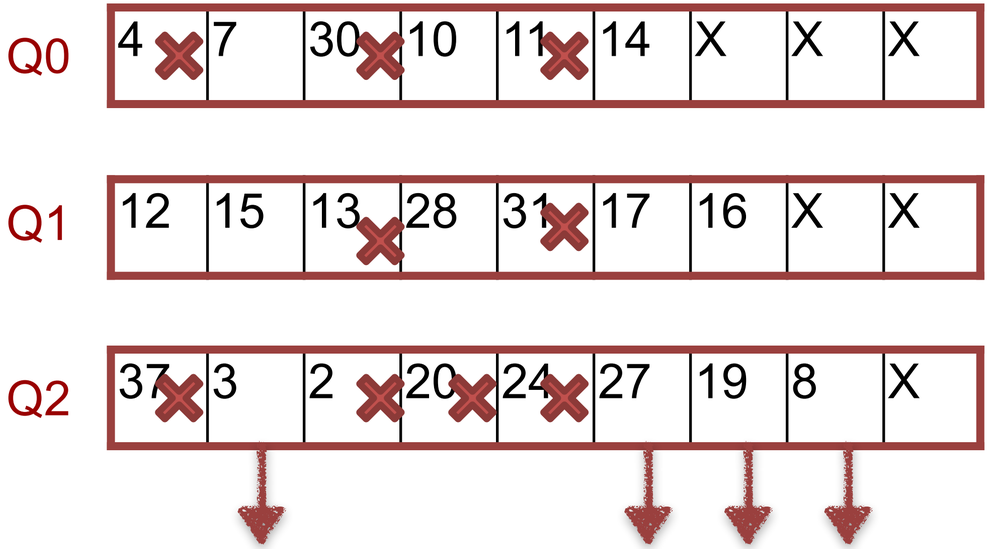
P8 normally should wait 56 ticks
But we boost after P20 finishes



In proc.c

boost()

```
boost (??):

  Increase wait time of all RUNNABLE processes
  in q2 according to last process' duration

  If any of the process' wait time passes 50
  boost it into the end of first priority queue

  p = p_to_boost
  Change priority of p
  c0++;
  q0[c0] = p
  Delete from q2 and shift left
```



When 7 works +1
When 10 works +1
When 12 works +2
When 15 works +2
When 3 works +8

In proc.c

getpinfo()

- Takes process id inside —> `getpinfo(int pid)`
- This is a system call so make sure to define it
- Define it in proc.h → `int getpinfo(int pid);`

→

```
syscall.h
syscall.c
user.h
usys.S
sysproc.c
```

```
**********************
name= CPUintensive, pid= 5
wait time= 0
ticks= {1, 2, 8}
times= {1, 1, 3}
**********************
start=1, duration=1, priority=0
start=2, duration=0, priority=1
start=2, duration=0, priority=1
start=2, duration=0, priority=1
start=2, duration=0, priority=1
start=2, duration=0, priority=1
start=2, duration=0, priority=1
start=2, duration=2, priority=1
start=4, duration=8, priority=2
start=12, duration=8, priority=2
```

We will not grade ticks and times because it can be interpreted and implemented differently

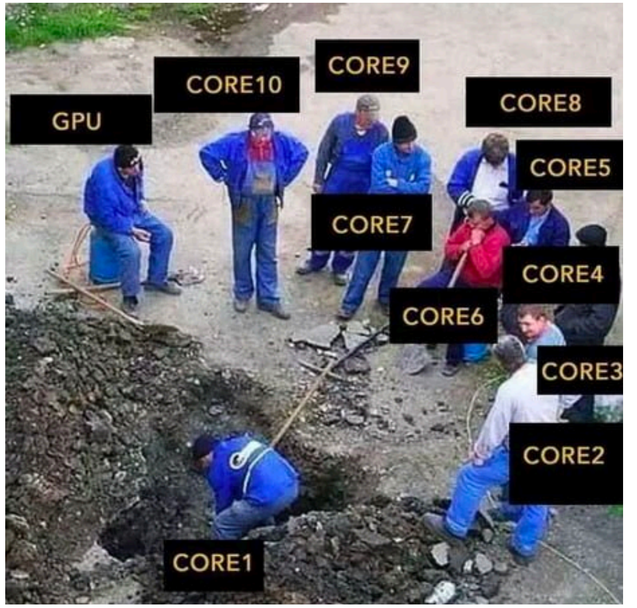
getpinfo()

```
getpinfo ():
  Get lock
  Get current process
  With for loop through all processes in ptable
    find pid
  If not found
    Release break
  Cprintf
  _____
  proc->name
  proc->wait_time
  proc->ticks[0],ticks[1],ticks[2]
  proc->times[0],times[1],times[2]
  _____
  // for each stat
  For each stat until num_of_stats
    Get/cprintf start_tick, duration and priority from each valid item in sched_stats
  Release lock
  Return 0
```

```
**********************
name= CPUintensive, pid= 5
wait time= 0
ticks= {1, 2, 8}
times= {1, 1, 3}
**********************
start=1, duration=1, priority=0
start=2, duration=0, priority=1
start=2, duration=0, priority=1
start=2, duration=0, priority=1
start=2, duration=0, priority=1
start=2, duration=0, priority=1
start=2, duration=0, priority=1
start=2, duration=2, priority=1
start=4, duration=8, priority=2
start=12, duration=8, priority=2
```

Test 1

```
Fork()

If (pid > 0)
   Do I/O instensive job
   wait();
   getpinfo(getpid());

Else:
    Do CPU intensive job
    exit()
```



Hints:

Do not need to use input in I/O intensive job
You can use output its input/output job!

Do not need to use recursion in CPU intensive job
Basic calculations are enough in for loop

Test 2

```
fork()
In parent:
   Fork multiple times – Like 10 times
   Do CPU or I/O intensive job
   getpinfo()

In child:
   Do CPU or I/O intensive job in each task
   getpinfo()
```

Hints:

Make sure your CPU job is taking long time else you cannot see boost!

Test 3

Hints:

Check slide 12 again

Graphing

Use excel or python - matplotlib
I might provide excel template later on Piazza
The link below is just an example it is not guaranteed that it works:
https://github.com/joshmin98/auto-graph-project3/

Any questions?