# CS 350
# PA4: MFQ
# Scheduler

## Bill Cheng

***http://merlot.usc.edu/william/usc/***

**Based on slides created by Kivilcim Cumbul**

# PA4

- Implement a basic non-preemptive MFQ (Multilevel Feedback Queue)
  - changes in "`proc.c`", "`proc.h`", "`trap.c`"
- Add a new system call to get information about a running process
  - `int getpinfo(pid)`
- Design tests to demonstrate the correctness of your scheduler
  - "`pa4-mixed.c`", "`pa4-aging.c`", "`pa4-cheat.c`"
- Create timeline graphs

# Part 1: Preparation

- Read Ch 5 of the xv6 book
- Download xv6 for PA4
  - open a *terminal* and type the following

```
cd ~/cs350
mkdir pa4
cd pa4
wget --user=USERNAME --password=PASSWORD \
  http://merlot.usc.edu/cs350-m25/programming/pa4/xv6-pa4-src.tar.gz
tar xvf xv6-pa4-src.tar.gz
cd xv6-pa4-src
```

  - make sure you choose 1 CPU in your VM

```
CPUS := 1
```

# Submission

- **Which files do you need to modify?**
  - open a terminal and type the following:

```
pwd
cd ~/cs350/pa4/xv6-pa4-src
make -n pa4-submit
```

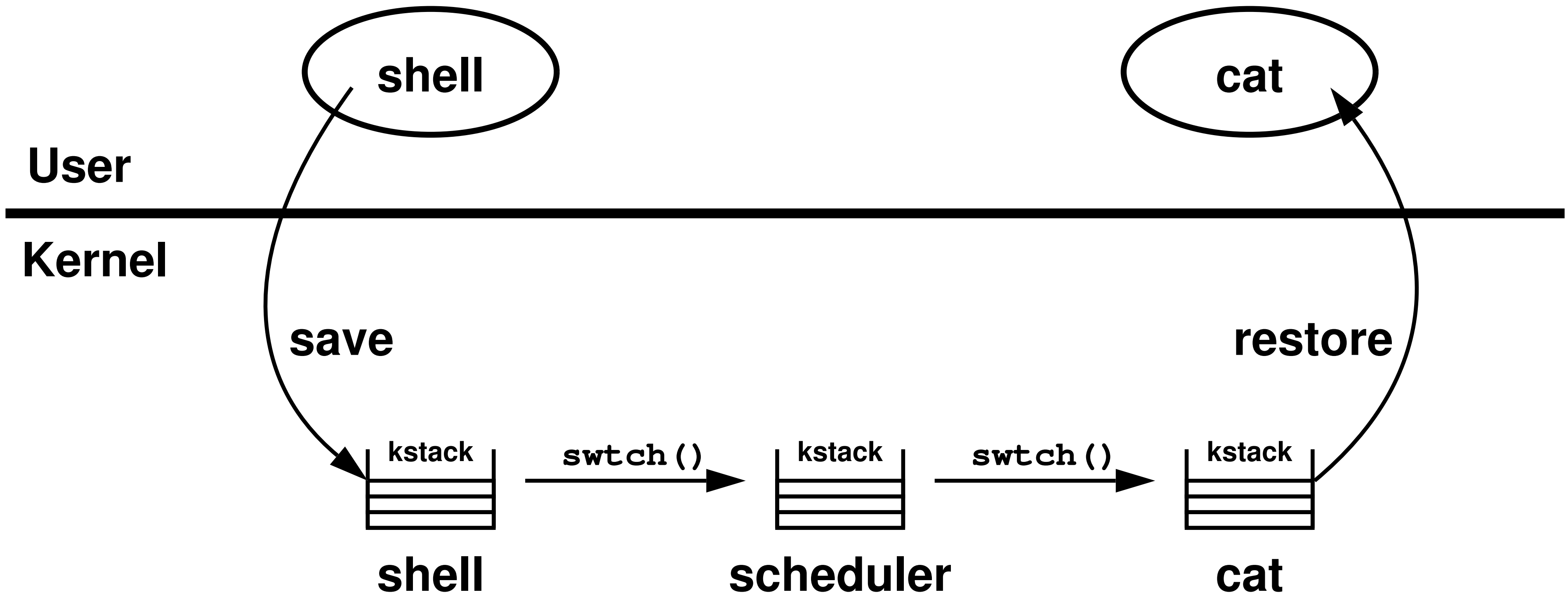
  - you should see:

```
tar cvzf pa4-submit.tar.gz \
        Makefile \
        pa4-README.txt \
        proc.c proc.h \
        trap.c \
        syscall.c syscall.h \
        sysproc.c \
        defs.h \
        pstat.h stat.h \
        user.h \
        usys.S \
        pa4-mixed.c pa4-mixed.out \
        pa4-mixed-parent.pdf pa4-mixed-child.pdf \
        pa4-aging.c pa4-aging.out pa4-aging.pdf \
        pa4-cheat.c pa4-cheat.out
```

# Part 2: MFQ Scheduler

- **In this project, you'll implement a simplified Multi-level Feedback Queue (MFQ) scheduler in xv6**
  - "`proc.c`", "`proc.h`", "`trap.c`"
- **Read Ch 5 of the xv6 book**
- **The default and only scheduling policy in xv6 is *round robin*, and we will change it**
- **In this project, you'll implement a simplified *Multi-level Feedback Queue (MFQ)* scheduler in xv6**

# XV6 Scheduler



# Default XV6 Scheduler: Round Robin

## Conceptually:

- a tick is 10ms
- we don't know how long a process needs to run

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| ticks = 1 | 4(2) | 7(4) | 8(7) | 10(1) | 11(3) | 14(6) | - | | |
| ticks = 2 | 7(4) | 8(7) | 10(1) | 11(3) | 14(6) | 4(1) | - | - | - |
| ticks = 3 | 8(7) | 10(1) | 11(3) | 14(6) | 4(1) | 7(3) | - | - | - |
| ticks = 4 | 10(1) | 11(3) | 14(6) | 4(1) | 7(3) | 8(6) | - | - | - |
| ticks = 5 | 11(3) | 14(6) | 4(1) | 7(3) | 8(6) | - | - | - | - |
| ticks = 6 | 14(6) | 4(1) | 7(3) | 8(6) | 11(2) | - | - | - | - |

# New XV6 Scheduler: MFQ

## Conceptually:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Q0 (high) | 4(1) | 7(1) | 8(7) | 10(1) | 11(1) | 14(1) | - | - | - |
| Q1 | 12(1) | 15(2) | 13(2) | 5(1) | - | - | - | - | - |
| Q2 (low) | 27(9) | 3(7) | 2(6) | - | - | - | - | - | - |

- if Q0 is not empty, serve Q0 round robin style
  - downgrade a process if it doesn’t give up the CPU volentarily
- if Q0 is empty and Q1 is not empty, serve Q1 round robin style (each process gets 2 ticks at a time)
  - downgrade a process if it doesn’t give up the CPU volentarily
- if Q0 and Q1 are empty and Q2 is not empty, serve Q2 round robin style (each process gets 8 ticks at a time)

# pstat.h

- don’t forget to:
  `#include "pstat.h"`
  in "`proc.c`"

```
#ifndef _PSTAT_H_
#define _PSTAT_H_
#define NTICKS 500
/* 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
```

# proc.h

## Inside struct proc

```
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

// schedule stats for each tick
struct sched_stat_t sched_stats[NSCHEDSTATS];
```

# How To Use Them?

- In `struct proc`

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

proc->ticks[0]
proc->ticks[1]
proc->ticks[2]

proc->wait_time
```

  - need to print these when printing stats

- In `struct sched_stat_t`

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

  - these are important stats of your process

# proc.h Additional Variables

- You may define as many variables as you want into `struct proc` or `struct sched_stats_t`
  - some examples for `struct proc`

```
int total_ticks;      // total number of timer ticks the process
                      // has run for
uint num_stats_used; // count to the end of the array
```

# Implementing MFQ

- For simplicity, we recommend that you use arrays to represent priority queues
  - it is much easier to deal with fixed-sized arrays in xv6 than dealing with linked-lists

```
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
```

  - need counters to represent queue sizes

- `proc.h`

```
extern struct proc* q0[NPROC];
extern struct proc* q1[NPROC];
extern struct proc* q2[NPROC];
extern int c0; // count
extern int c1; // count
extern int c2; // count
```

- `proc.c`

```
struct proc* q0[NPROC];
struct proc* q1[NPROC];
struct proc* q2[NPROC];
int c0 = -1;
int c1 = -1;
int c2 = -1;
```

- Can use something else like `tail`, `front`, `capacity`, etc.

# MFQ Scheduler

1) numbered from 0 (highest priority) down to 2 (lowest priority)
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) cheat: can get out before CPU increase the tick
6) when a new process arrives, it *should start at priority 0* (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* place it at the end of the highest priority queue; it should not preempt a process with the same priority
9) your scheduler should *never preempt a lower priority process* if a higher priority process is available to run

# Visualization

- tick 1: proc 4 needs 1 tick to finish

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Q0 | 4(1) √ | 7(1) | 8(7) | 10(1) | 11(1) | 14(1) | - | - | - |
| Q1 | 12(1) | 15(2) | 13(2) | 5(1) | - | - | - | - | - |
| Q2 | 27(9) | 3(7) | 2(6) | - | - | - | - | - | - |

# Visualization

- tick 2: proc 7 needs 1 tick to finish

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Q0 | 4(0) | 7(1) | 8(7) | 10(1) | 11(1) | 14(1) | - | - | - |
| Q1 | 12(1) | 15(2) | 13(2) | 5(1) | - | - | - | - | - |
| Q2 | 27(9) | 3(7) | 2(6) | - | - | - | - | - | - |

# Visualization

- tick 3: proc 8 needs 7 ticks to finish, use 1 tick

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Q0 | ~~4(0)~~ | ~~7(0)~~ | 8(7) | 10(1) | 11(1) | 14(1) | - | - | - |
| Q1 | 12(1) | 15(2) | 13(2) | 5(1) | - | - | - | - | - |
| Q2 | 27(9) | 3(7) | 2(6) | - | - | - | - | - | - |

# Visualization

- tick 4: proc 10 needs 1 tick to finish

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Q0 | 4(0) | 7(0) | 8(6) | 10(1) | 11(1) | 14(1) | - | - | - |
| Q1 | 12(1) | 15(2) | 13(2) | 5(1) | 8(6) | - | - | - | - |
| Q2 | 27(9) | 3(7) | 2(6) | - | - | - | - | - | - |

# Visualization

- tick 7: proc 12 needs 1 tick to finish

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Q0 | 4(0) | 7(0) | 8(6) | 10(0) | 11(0) | 14(0) | - | - | - |
| Q1 | 12(1) √ | 15(2) | 13(2) | 5(1) | 8(6) | - | - | - | - |
| Q2 | 27(9) | 3(7) | 2(6) | - | - | - | - | - | - |

# Visualization

- tick 8: proc 15 needs 2 ticks to finish

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Q0** | ~~4(0)~~ | ~~7(0)~~ | ~~8(6)~~ | ~~10(0)~~ | ~~11(0)~~ | ~~14(0)~~ | - | - | - |
| **Q1** | ~~12(0)~~ | 15(2) | 13(2) | 5(1) | 8(6) | - | - | - | - |
| **Q2** | 27(9) | 3(7) | 2(6) | - | - | - | - | - | - |

# Visualization

- tick 10: proc 13 needs 2 ticks to finish

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Q0 | ~~4(0)~~ | ~~7(0)~~ | ~~8(6)~~ | ~~10(0)~~ | ~~11(0)~~ | ~~14(0)~~ | - | - | - |
| Q1 | ~~12(0)~~ | ~~15(0)~~ | 13(2) √ | 5(1) | 8(6) | - | - | - | - |
| Q2 | 27(9) | 3(7) | 2(6) | - | - | - | - | - | - |

# Visualization

- tick 12: proc 5 needs 1 ticks to finish

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Q0 | ~~4(0)~~ | ~~7(0)~~ | ~~8(6)~~ | ~~10(0)~~ | ~~11(0)~~ | ~~14(0)~~ | - | - | - |
| Q1 | ~~12(0)~~ | ~~15(0)~~ | ~~13(0)~~ | 5(1) | 8(6) | - | - | - | - |
| Q2 | 27(9) | 3(7) | 2(6) | - | - | - | - | - | - |

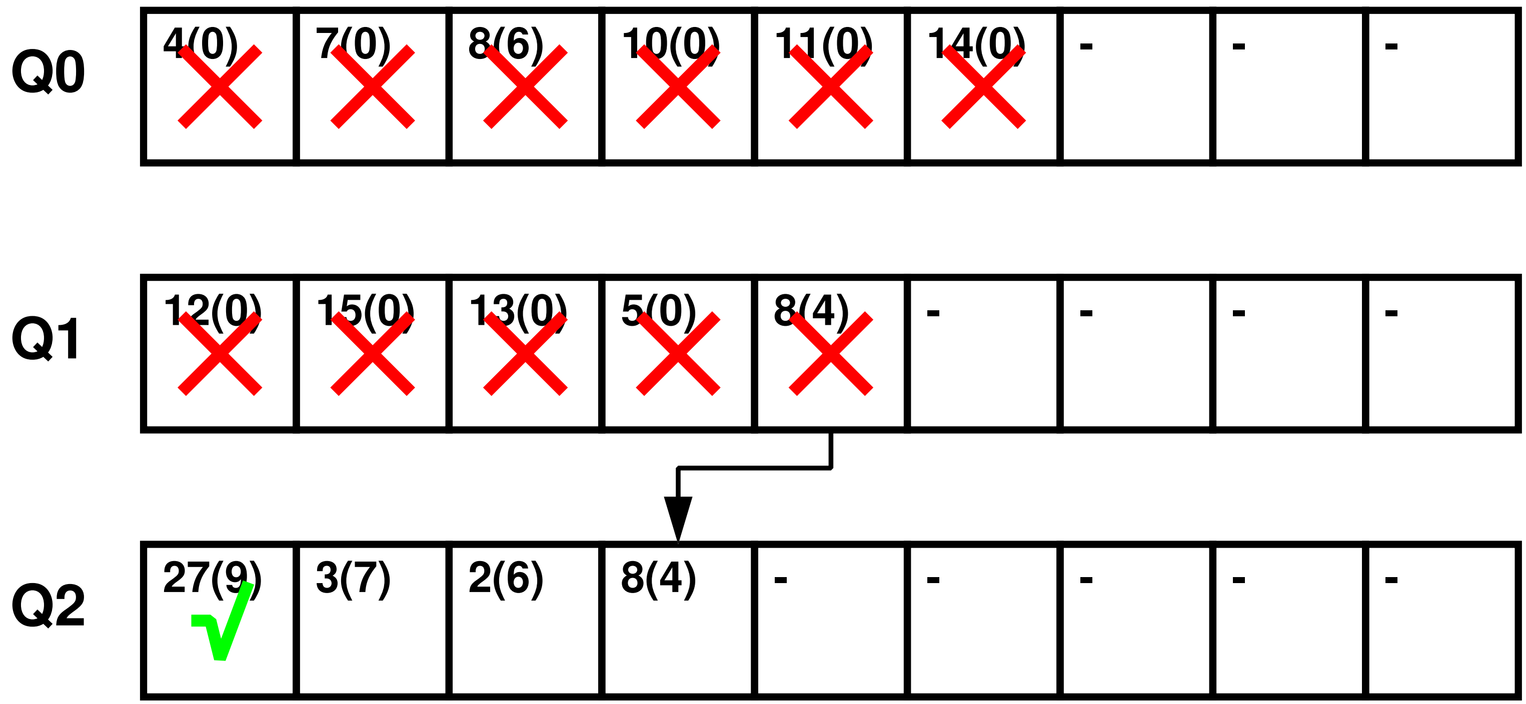
# Visualization

- tick 13: proc 8 needs 6 more ticks to finish

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Q0** | ~~4(0)~~ | ~~7(0)~~ | ~~8(6)~~ | ~~10(0)~~ | ~~11(0)~~ | ~~14(0)~~ | - | - | - |
| **Q1** | ~~12(0)~~ | ~~15(0)~~ | ~~13(0)~~ | ~~5(0)~~ | 8(6) √ | - | - | - | - |
| **Q2** | 27(9) | 3(7) | 2(6) | - | - | - | - | - | - |

# Visualization

- tick 15: proc 27 needs 9 more ticks to finish



# Visualization

- tick 23: proc 3 needs 7 ticks to finish

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Q0 | 4(0) | 7(0) | 8(6) | 10(0) | 11(0) | 14(0) | - | - | - |
| Q1 | 12(0) | 15(0) | 13(0) | 5(0) | 8(4) | - | - | - | - |
| Q2 | 27(1) | 3(7) | 2(6) | 8(4) | 27(1) | - | - | - | - |

# Visualization

- tick 23: proc 3 needs 7 ticks to finish

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Q0 | ~~4(0)~~ | ~~7(0)~~ | ~~8(6)~~ | ~~10(0)~~ | ~~11(0)~~ | ~~14(0)~~ | - | - | - |
| Q1 | ~~12(0)~~ | ~~15(0)~~ | ~~13(0)~~ | ~~5(0)~~ | ~~8(4)~~ | - | - | - | - |
| Q2 | ~~27(1)~~ | 3(7) | 2(6) | 8(4) | 27(1) | - | - | - | - |

- ***IMPORTANT:*** **since these are arrays, must *shuffle* the entire array to the left when removing a process from a queue**

# allocproc()

```
static struct proc*
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:
  // need remove p from queues if in there
  ...
}
```

- You may have a reference to `p` in one of the 3 queues
  - search for it and if found, remove it from the queue by shuffling left

# allocproc()

## Find p and shuffle left

```
if (p->pid >= 0){
  for (int i = 0; (i < c0); i++){
    if (p == q0[i]){
      // delete q0[i] by shifting array elements by
      // one position to the left starting at index i+1;
    }
  }
  for (int i = 0; i < c1; i++){
    // do the same thing for q1
  }
  for (int i = 0; i < c2; i++){
    // do the same thing for q1
  }
}
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4(1) | 7(1) | 8(7) | 10(1) | 11(1) | 14(1) | - | - | - |

# allocproc()

- Ex: look for pid 10 and remove it

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4(1) | 7(1) | 8(7) | 10(1) | 11(1) | 14(1) | - | - | - |

  - don’t just zero out the pointer

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4(1) | 7(1) | 8(7) | ~~10(1)~~ | 11(1) | 14(1) | - | - | - |

# allocproc()

- Ex: look for pid 10 and remove it

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4(1) | 7(1) | 8(7) | 10(1) | 11(1) | 14(1) | - | - | - |

  - don't just zero out the pointer

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4(1) | 7(1) | 8(7) | ~~10(1)~~ | 11(1) | 14(1) | - | - | - |

  - shuffle left

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4(1) | 7(1) | 8(7) | 11(1) | 14(1) | ~~-~~ | - | - | - |

    - don't forget to decrement the counter since the queue size has been reduced

# allocproc()

- **Need to initialize all the new PA4 fields**
- **At the end of `allocproc()`, need to add `p` to `q0` to the end of `q0`**

```
q0[c0++] = p;
```

# `userinit()`

- Everything starts with `userinit()`
  - on page 23 of the xv6 book, it says:

### Code: creating the first process

Now we'll look at how the kernel creates user-level processes and ensures that they are strongly isolated.

After `main` (1217) initializes several devices and subsystems, it creates the first process by calling `userinit` (2520). `Userinit`'s first action is to call `allocproc`. The job

  - need to reset/initialize scheduler stats at the beginning of `userinit()`

# scheduler()

- **main() calls userinit() then calls mpmain() which invokes the scheduler by calling scheduler()**
  - **on page 25 of the xv6 book, it says:**

### Code: Running the first process

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 (1257). Scheduler (2758) 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 (1879). Changing page tables

  - **on the bottom of page 25 of the xv6 book, it says:**

scheduler now sets p->state to RUNNING and calls swtch (3059) to perform a context switch to the target process's kernel thread. swtch first saves the current registers. 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. swtch then loads the saved registers of the target kernel thread (p->context) into the x86 hardware registers, including the stack pointer and instruction pointer. We'll examine swtch in more detail in Chapter 5. The final ret instruction (3078) 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.

  - **this is the *round robin* scheduler and you need to change it to use the *MFQ* scheduling policy**

# `scheduler()`

- The xv6 scheduler is a per-CPU process scheduler
  - each CPU calls `scheduler()` after setting itself up
  - scheduler never returns, it loops, doing:
    1) choose a process to run
    2) `swtch()` to run that process
    3) eventually that process transfers control by calling either `sleep()`, `yield()`, or `exit()`, which all calls `swtch()` to get back to the scheduler

- If a process calls `sleep()`, it would be considered that the process is *giving up the CPU volentarily*
  - need to make note of that since its crucial to MFS scheduling
  - calling `yield()` is not considered giving up the CPU volentarily because `yield()` is called in `trap()`

# switchuvm() & switchkvm()

- The `u` in `switchuvm()` stands for user, the `k` in `switchkvm()` stands for kernel
  - keep this part of code structure, need a different way to find `p`

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  ...
  for (;;) {
      ...
      // found a RUNNABLE process p, run it
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

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

      c->proc = 0;
      ...
  }
}
```

# How To Modify `scheduler()`

- In the round robin scheduler, the `scheduler()` loops through all processes

```
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE)
    continue;
```

  - need to look for a `RUNNABLE` process in `q0` instead
    - if no `RUNNABLE` process in `q0`, look in `q1`
    - if no `RUNNABLE` process in `q1`, look in `q2`

# How To Modify `scheduler()` - 1st Half

```
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 in q0
  p->sched_stats[?].start_tick = ticks;
  p->sched_stats[?].duration = 0;
  p->sched_stats[?].priority = 0;
  c->proc = p;
  switchuvm(p);
  p->state = RUNNING;

  swtch(&(c->scheduler), p->context);
  switchkvm();
  // p came back, update stats
  duration = ticks - p->sched_stats[?].start_tick;
  p->num_stats_used++;
  p->times[0]++; // number of times run in q0
  p->ticks[0] = duration; // q0 stat
  ... // update other things in your data structures
```

- the above is incomplete and depends on what you have

# How To Think About Stats

- How to time something in general?

```
start time = x
// do some work
// let others do their things
come back
end time = y
duration = y - x
```

```
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
```

# Stats

- `tick` cames from `proc.c`
- `proc->total_ticks`: number of timer ticks the process has run
- `proc->ticks[]`: number of ticks a process used the last time it was scheduled in each priority queue
- `proc->times[]`: number of times a process was scheduled at each priority queue

```
duration = ticks - start_tick;
p->num_stats_used++;
p->times[0]++;
p->ticks[0] = duration;
... // update other things in your data structures
```

```
PSTAT_START
*********
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=0, priority=1
start=4, duration=8, priority=2
start=12, duration=8, priority=2
PSTAT_END
```

# How To Modify `scheduler()` - 2nd Half

```
if q0 is not empty
  // continued from a few slides back

  if (p->ticks[0] >= 1)
    // move p to q1 (need to increment c1)
    // plus other necessary changes
    q1[c1] = p // put at end of q1
    q0[i] = 0 // delete p from q0
    // shift all left from i in q0
    ...

boost() // boost if necessary
c->proc = 0 // reset the CPU
```

- the above is incomplete and depends on what you have

# Q1

```
if q1 is not empty
  // loop to find RUNNABLE process in the q1 array
    if q1[i]->state != RUNNABLE)
      continue;

  p = q1[i]; // found first RUNNABLE in q1
  ...
  c->proc = p;
  switchuvm(p);
  p->state = RUNNING;

  swtch(&(c->scheduler), p->context);
  switchkvm();
  // p came back, update stats
  duration = ticks - start_tick;
  p->num_stats_used++;
  p->times[1]++; // number of times run in q0
  p->ticks[1] = duration; // q1 stat
  ... // update other things in your data structures
```

- the above is incomplete and depends on what you have

# Q1 - 2nd Half

```
if q1 is not empty
  // continued from previous slide

  if (p->ticks[1] >= 2)
    // move p to q2 (need to increment c2)
    // other necessary changes
    q2[c2] = p // put at end of q2
    q1[i] = 0 // delete p from q1
    // shift all left from i in q1
    ...

boost() // boost if necessary
c->proc = 0 // reset the CPU
```

- the above is incomplete and depends on what you have

# Q2

```
if q2 is not empty
  // loop to find RUNNABLE process in the q1 array
    if q2[i]->state != RUNNABLE)
      continue;

  p = q2[i]; // found first RUNNABLE in q1
  ...
  c->proc = p;
  switchuvm(p);
  p->state = RUNNING;

  swtch(&(c->scheduler), p->context);
  switchkvm();
  // p came back, update stats
  duration = ticks - start_tick;
  p->num_stats_used++;
  p->times[2]++; // number of times run in q0
  p->ticks[2] = duration; // q1 stat
  ... // update other things in your data structures
```

- the above is incomplete and depends on what you have

# Q2 - 2nd Half

```
if q2 is not empty
  // continued from previous slide

  if (p->ticks[2] >= 8)
    // increment c2
    // shift all left from i in q2
    q2[c2] = p // put at end of q2
    ...

boost() // boost if necessary
c->proc = 0 // reset the CPU
```

- the above is incomplete and depends on what you have

# boost()

- 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*

# boost()

- `wait_time`: number of ticks since the process last run
  - if any of the processes in `q2` has `wait_time` > 50, boost its priority and move it to the end of `q0`

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

  - please note that as far as grading goes, it's ***acceptable*** to boost a process in `q1` having `wait_time` > 50

# getpinfo()

- `int getpinfo(int pid)`
  - this is a system call, make sure to define it in `syscall.h`, `syscall.c`, `user.h`, `usys.S`, and `sysproc.c`
  - define it in `proc.h`

```
PSTAT_START
*********
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=0, priority=1
start=4, duration=8, priority=2
start=12, duration=8, priority=2
PSTAT_END
```

we will not grade these

these are the exact strings you must use because they are important delimeters

# getpinfo()

```
getpinfo(int pid):

  get lock
  get current process
  loop through all processes in ptable to find pid
  if not found
    release lock
    return -1
  cprintf:
    PSTAT_START
    *********
    proc->name
    proc->wait_time
    proc->ticks[0],ticks[1],ticks[2]
    proc->times[0],times[1],times[2]
    *********
  for each stat (until num_stats)
    if item valid in sched_stats
      cprintf: start_tick, duration, priority
  cprintf
    PSTAT_END
  release lock
  return 0
```

# Timer Interrupt

- The XV6 timer ticks every 10ms
  - timer interrupt code in `trap()`:

```
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

  - the above is the round robin scheduler
    - `ticks` (global variable) is increment in `trap()`

- What to do for the MFQ scheduler:

# Timer Interrupt

- For the MFQ scheduler
  - let `current_stat` be the last entry in the `sched_stats` of the running process
  - let `num_ticks` be the number of ticks experienced by the current process

```
num_ticks = ticks - current_stat.start_tick
```

  - call `yield()` if:
    - the priority of the running process is 0 and num_ticks ≥ 1
    - the priority of the running process is 1 and num_ticks ≥ 2
    - the priority of the running process is 2 and num_ticks ≥ 8

# Part 3: Write Test Programs

- Three tests
  - `pa4-mixed.c`: a mix of I/O intensive and CPU intensive processes
  - `pa4-aging.c`: demonstrate that you have implemented aging
  - `pa4-cheat.c`: demonstrate that a program can cheat to always run at highest priority

# pa4-mixed

```
pa4-mixed:

  pid = fork()
  if (pid > 0)
    io_intensive(num_ios);
    wait()
    getpinfo(getpid())
    exit()
  else
    cpu_intensive(num_millions);
    getpinfo(getpid())
    exit()
  end-if
```

- try different values of `num_ios` and `num_millions`
- total running time should be less than 100 ticks

# pa4-mixed

- What's an I/O intensive job?
  - sit in a tight loop calling `printf(99)`

```
static void io_intensive(int num_itr)
{
  for (int i = 0; i < num_itr; i++) {
    printf(99, "\n");
  }
}
```

  - play with `num_itr` to control the running time

- What's a CPU intensive job?
  - sit in a tight loop to increment an integer for millions of times
  - can use a `volatile long long` integer to slow things down

```
static void cpu_intensive(int num_millions)
{
  volatile unsigned long long k = 0;
  for (int i = 0; i < num_millions*1000000; i++) {
    k++;
  }
}
```

  - play with `num_millions` to control the running time

# pa4-aging

```
pa4-aging:

  pid = fork()
  if (pid > 0) then
    for (i = 0; i < 10; i++)
      if (fork() == 0) then
        cpu_intensive(num);
        getpinfo(getpid());
        exit();
      end-if
    end-for
    cpu_intensive(num);
    for (i = 0; i < 11; i++)
      wait()
    end-for
    getpinfo(getpid());
    exit()
  else
    cpu_intensive(num);
    getpinfo(getpid());
    exit()
  end-if
```

- try different values of `num`
- important to see that for at least one process, the priority would change from 2 to 0 for at least once
- total running time should be less than 500 ticks (remember that you can only keep track of up to 1500 stats)

# pa4-cheat

```
pa4-cheat:

  pid = fork()

  if (pid > 0)
    wait()
    exit()
  else
    do the following 30 times
      sleep(1)
    getpinfo(getpid())
    exit()
```

- important to see that the child process always run with `priority=0`
- child process must be scheduled to run for at least 30 times
- don't do anything else in the child (such as calling `printf()`)

- Since `sleep(1)` is considered giving up the CPU, the child should always run with highest priority (which can be seen from the `getpinfo()` printout
  - make sure your have modified `sleep()` to indicate that you have given up the CPU before `sleep()` calls `sched()`
- Your child process must call `sleep(1)` for at least 30 times

# Part 4: Graphing Results

- Use excel or python - matplotlib
- This is just an example it is not guaranteed that it works: https://github.com/joshmin98/auto-graph-project3/
  - it won't run on the "standard" system
  - you can create a transcript using the "`script`" program then process the "`transcript`" file
    - do "`man script`"

```
PSTAT_START
*********
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=4, duration=8, priority=2
start=12, duration=8, priority=2
PSTAT_END
```

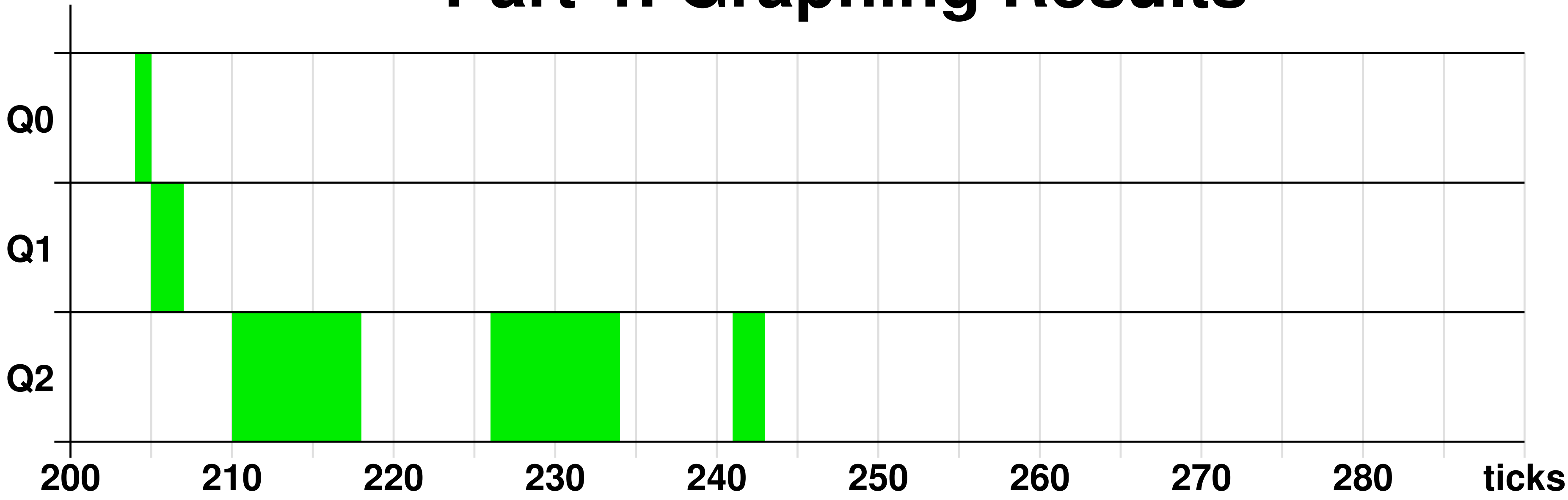
# Part 4: Graphing Results

## Ex: running `pa4-mixed`

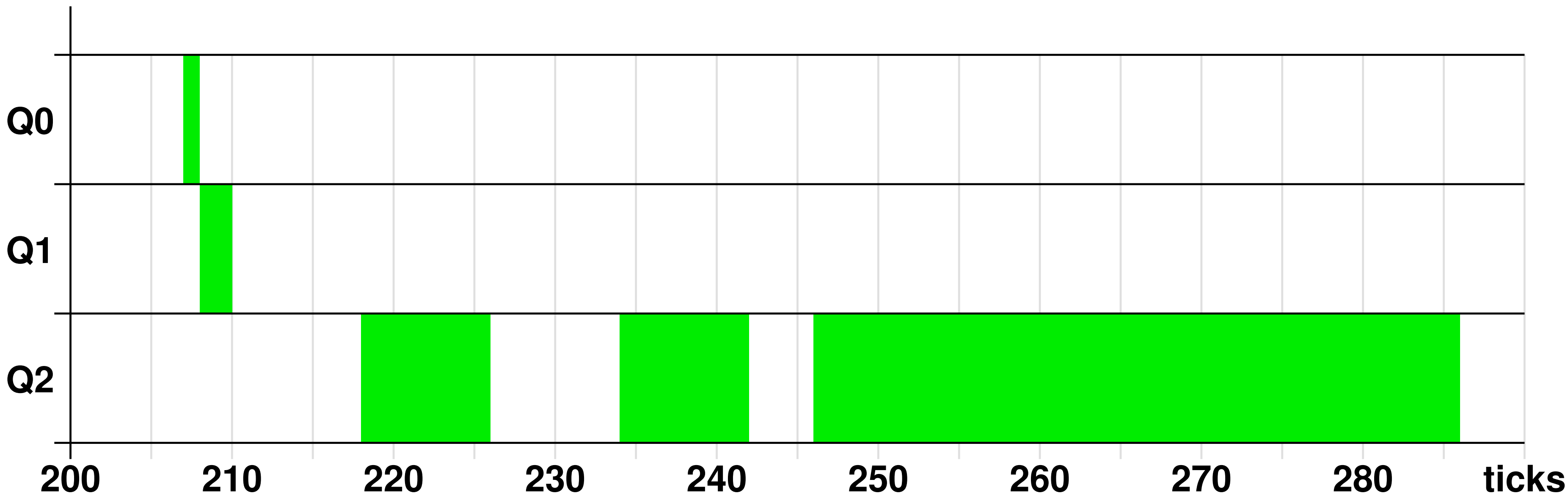
```
PSTAT_START
*********
name = pa4-mixed, pid = 3
wait time = 0
ticks = {1, 2, 4}
times = {7, 1, 3}
yield count = 4
willing yields count = 7
wait count = 0
*********
start=203, duration=0, priority=0
start=204, duration=0, priority=0
start=204, duration=0, priority=0
start=204, duration=0, priority=0
start=204, duration=0, priority=0
start=204, duration=0, priority=0
start=204, duration=1, priority=0
start=205, duration=2, priority=1
start=210, duration=8, priority=2
start=226, duration=8, priority=2
start=242, duration=4, priority=2
PSTAT_END
```

```
PSTAT_START
*********
name = pa4-mixed, pid = 4
wait time = 0
ticks = {1, 2, 8}
times = {1, 1, 7}
yield count = 9
willing yields count = 0
wait count = 0
*********
start=207, duration=1, priority=0
start=208, duration=2, priority=1
start=218, duration=8, priority=2
start=234, duration=8, priority=2
start=246, duration=8, priority=2
start=254, duration=8, priority=2
start=262, duration=8, priority=2
start=270, duration=8, priority=2
start=278, duration=8, priority=2
PSTAT_END
```

# Part 4: Graphing Results




Timeline graph for pa4-mixed (pid=3)




Timeline graph for pa4-mixed (pid=4)

# Part 4: Graphing Results

- Need to submit the following files:

```
pa4-mixed.c pa4-mixed.out
pa4-mixed-parent.pdf pa4-mixed-child.pdf

pa4-aging.c pa4-aging.out pa4-aging.pdf

pa4-cheat.c pa4-cheat.out
```

  - `pa4-*.out`: transcript of running the corresponding `pa4-*` program
  - `pa4-mixed-parent.pdf` and `pa4-mixed-child.pdf`: see previoius slide
  - `pa4-aging.pdf`: graph the timeline of *one of the processes* that's showing that it was boosted