Mini-Project 2: Ticket-based vs Priority-based Schedulers

Project Overview

This project involves implementing two different scheduling algorithms in xv6:

- **Part A**: Lottery (Ticket-based) Scheduler
- **Part B**: Priority-based Scheduler

Critical Understanding: xv6's Timer Tick Behavior

** IMPORTANT**: Before starting, understand how xv6's scheduler works:

Default Round-Robin Behavior

- xv6 has a **timer interrupt every ~10ms** (one "timer tick")
- On **every timer tick**, the running process yields to the scheduler
- The vanilla scheduler picks the **next RUNNABLE process** in the process table
- Pattern: A B C A B C... (each process runs for ~10ms)

Key Insight

The vanilla xv6 scheduler switches processes on EVERY timer tick - there's no explicit time slice tracking. For this project:

- **Lottery Scheduler**: Each timer tick = one lottery draw
- **Priority Scheduler**: Each timer tick = check priorities and schedule highest
- Both schedulers will naturally switch processes every ~10ms due to timer interrupts

---

Part A: Lottery Scheduler

Concept

The lottery scheduler assigns CPU time proportionally based on "tickets":

- Each process has a number of tickets (minimum 1)
- At each scheduling decision (every timer tick), hold a "lottery"
- Randomly pick a winning ticket number
- The process holding that ticket runs for the next tick

Example: If Process A has 10 tickets and Process B has 20 tickets:

- Total tickets = 30
- Process B should get ~66% of CPU time, Process A gets ~33%

Required System Calls

1. `int settickets(int number)`

Sets the number of tickets for the calling process.

- **Input**: Number of tickets (must be ≥ 1)
- **Returns**: 0 on success, -1 on failure
- **Default**: Each process starts with 1 ticket
- **Inheritance**: Child processes inherit parent's ticket count via `fork()`

2. `int getpinfo(struct pstat *)`

Retrieves information about all processes for testing.

- **Returns**: 0 on success, -1 on failure (e.g., NULL pointer)
- **Structure** (defined in `pstat.h`):

```
# ifndef _PSTAT_H_
# define _PSTAT_H_
# include "param.h"

struct pstat {
    int inuse[NPROC];   // whether this slot is in use (1 or 0)
    int tickets[NPROC]; // number of tickets this process has
    int pid[NPROC];     // PID of each process
    int ticks[NPROC];   // number of times process was scheduled
};
# endif // _PSTAT_H_
```

Implementation Details

Lottery Algorithm (executed every timer tick):

1. Count total tickets from all RUNNABLE processes
2. Generate random number: `winner = random() % total_tickets`
3. Iterate through processes, subtracting their tickets from winner
4. When winner < 0, that process is selected
5. Run the selected process until next timer tick

Random Number Generation

You'll need a pseudo-random number generator (PRNG) in the kernel. Simple option:

```
// Add to proc.c
static unsigned long rand_next = 1;

int
random(void)
{
    rand_next = rand_next * 1103515245 + 12345;
    return (unsigned int)(rand_next / 65536) % 32768;
}
```

Ticket Inheritance

In `fork()` (in `proc.c`), copy parent's tickets to child:

```
np->tickets = curproc->tickets;
```

Testing Strategy

Create a test program that:

1. Creates 5 child processes using `fork()`
2. Assigns different ticket counts: 10, 20, 30, 40, 50
3. Has each process do CPU-intensive work (loop)
4. Calls `getpinfo()` to collect scheduling statistics
5. Displays results showing ticket count vs actual CPU time

Expected Result: Processes with more tickets should accumulate more ticks (be scheduled more often).

Graph Requirements

Create a column graph showing:

- **X-axis**: Process ID
- **Y-axis (two columns per process)**:
    - Tickets assigned (one color)
    - Ticks received (another color)

The graph should show proportional relationship between tickets and ticks.

Part B: Priority-based Scheduler

Concept

Priority-based scheduler always runs the highest-priority process:

- **Priority range**: 0-200 (0 = highest priority, 200 = lowest)
- **Default priority**: 50
- **Tie-breaking**: If multiple processes have same priority, use round-robin

Example: Processes with priorities [30, 30, 30, 40, 50]:

- Run the three priority-30 processes in round-robin
- Then run priority-40 process
- Finally run priority-50 process

Required System Call

`int setpriority(int priority)`

Sets the priority of the calling process.

- **Input**: Priority value (0-200 inclusive)
- **Returns**: Old priority value
- **Validation**: Return -1 if priority out of range
- **Re-scheduling**: If new priority is lower (larger value), call `yield()` to reschedule immediately

Implementation Details

Scheduler Algorithm (executed every timer tick):

1. Scan all RUNNABLE processes
2. Find the highest priority (smallest number)
3. Among processes with highest priority, use round-robin:
    - Track last scheduled process at this priority
    - Start from next process in round-robin order
4. Run selected process until next timer tick

Round-Robin Within Priority Level

To avoid starvation among same-priority processes:

- Option 1: Track index of last scheduled process per priority
- Option 2: Iterate through process table starting from last scheduled position

Testing Strategy

Similar to Part A:

1. Create multiple processes with different priorities
2. Assign priorities: 10, 20, 30, 40, 50 (and others at default 50)
3. Have each process do CPU work
4. Use `getpinfo()` (with modified `pstat` structure) to collect data
5. Show that lower-numbered priorities get more CPU time

Graph Requirements

Show that processes with higher priority (lower numbers) receive more CPU time than lower priority processes.

Implementation Guide

Files to Modify

Both Part A and Part B require changes to these files:

Kernel Files:

- `kernel/proc.h` - Add fields to `struct proc`
- `kernel/proc.c` - Implement scheduler logic, modify `allocproc()`, `fork()`
- `kernel/sysproc.c` - Implement system call handlers
- `kernel/syscall.h` - Add system call numbers
- `kernel/syscall.c` - Register system calls
- `kernel/defs.h` - Add function declarations

User Files:

- `user/user.h` - Declare system calls for user programs
- `user/usys.S` - Add system call stubs
- `include/pstat.h` - Define `pstat` structure

Test Programs:

- `user/tickettest.c` (Part A) - Test lottery scheduler
- `user/prioritytest.c` (Part B) - Test priority scheduler
- `user/ps.c` - Display process information

Step-by-Step Implementation

Part A: Lottery Scheduler

Step 1: Modify `struct proc` in `kernel/proc.h`

```
struct proc {
    // ... existing fields ...
    int tickets;        // Number of lottery tickets
    int ticks;          // Number of times scheduled
};
```

Step 2: Initialize in `allocproc()` in `kernel/proc.c`

```
// In allocproc(), in the found: section
p->tickets = 1;  // Default: 1 ticket
p->ticks = 0;    // Not scheduled yet
```

Step 3: Implement lottery scheduler in `scheduler()` function

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

    for(;;){
        sti();
        acquire(&ptable.lock);

        // Calculate total tickets from RUNNABLE processes
        int total_tickets = 0;
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->state == RUNNABLE)
                total_tickets += p->tickets;
        }

        if(total_tickets > 0){
            // Hold lottery
            int winner = random() % total_tickets;

            // Find winning process
            for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
                if(p->state == RUNNABLE){
                    winner -= p->tickets;
                    if(winner < 0){
                        // This process wins!
```

```
                        c->proc = p;
                        switchuvm(p);
                        p->state = RUNNING;
                        p->ticks++;  // Increment schedule count

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

                        c->proc = 0;
                        break;
                    }
                }
            }
        }

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

Step 4: Implement `sys_settickets()` in `kernel/sysproc.c`

```
int
sys_settickets(void)
{
    int n;

    if(argint(0, &n) < 0)
        return -1;

    if(n < 1)
        return -1;

    acquire(&ptable.lock);
    myproc()->tickets = n;
    release(&ptable.lock);

    return 0;
}
```

Step 5: Implement `sys_getpinfo()` in `kernel/sysproc.c`

```
int
sys_getpinfo(void)
```

```
{
    struct pstat *ps;

    if(argptr(0, (void*)&ps, sizeof(*ps)) < 0)
        return -1;

    struct proc *p;
    int i = 0;

    acquire(&ptable.lock);

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++, i++){
        if(p->state == UNUSED){
            ps->inuse[i] = 0;
        } else {
            ps->inuse[i] = 1;
            ps->tickets[i] = p->tickets;
            ps->pid[i] = p->pid;
            ps->ticks[i] = p->ticks;
        }
    }

    release(&ptable.lock);
    return 0;
}
```

Step 6: Handle ticket inheritance in `fork()` in `kernel/proc.c`

```
// In fork(), after acquiring lock and before copying state
np->tickets = curproc->tickets;
```

Part B: Priority Scheduler

Step 1: Modify `struct proc` in `kernel/proc.h`

```
struct proc {
    // ... existing fields ...
    int priority;       // Process priority (0-200, lower is higher)
    int ticks;          // Number of times scheduled (for testing)
};
```

Step 2: Initialize in `allocproc()` in `kernel/proc.c`

```
// In allocproc()
p->priority = 50;  // Default priority
p->ticks = 0;
```

Step 3: Implement priority scheduler in `scheduler()` function

```
void
scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;
    static struct proc *last_sched = 0;  // For round-robin within priority

    for(;;){
        sti();
        acquire(&ptable.lock);

        // Find highest priority (minimum value)
        int highest_priority = 201;  // Higher than max valid priority
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->state == RUNNABLE && p->priority < highest_priority){
                highest_priority = p->priority;
            }
        }

        // Round-robin among processes with highest priority
        struct proc *start = last_sched ? last_sched + 1 : ptable.proc;
        struct proc *selected = 0;

        // Search from last_sched to end
        for(p = start; p < &ptable.proc[NPROC]; p++){
            if(p->state == RUNNABLE && p->priority == highest_priority){
                selected = p;
                break;
            }
        }

        // Wrap around if not found
        if(selected == 0){
            for(p = ptable.proc; p < start; p++){
                if(p->state == RUNNABLE && p->priority == highest_priority){
                    selected = p;
```

```
                    break;
                }
            }
        }

        if(selected != 0){
            p = selected;
            last_sched = p;

            c->proc = p;
            switchuvm(p);
            p->state = RUNNING;
            p->ticks++;

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

            c->proc = 0;
        }

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

Step 4: Implement `sys_setpriority()` in `kernel/sysproc.c`

```
int
sys_setpriority(void)
{
    int priority;

    if(argint(0, &priority) < 0)
        return -1;

    if(priority < 0 || priority > 200)
        return -1;

    struct proc *p = myproc();
    int old_priority;

    acquire(&ptable.lock);
    old_priority = p->priority;
    p->priority = priority;
```

```
    release(&ptable.lock);

    // If new priority is lower (larger value), yield
    if(priority > old_priority){
        yield();
    }

    return old_priority;
}
```

System Call Registration

Add to `kernel/syscall.h`:

```
// Part A
# define SYS_settickets 22
# define SYS_getpinfo   23

// Part B (if separate implementation)
# define SYS_setpriority 22
```

Add to `kernel/syscall.c`:

```
extern int sys_settickets(void);
extern int sys_getpinfo(void);
extern int sys_setpriority(void);

static int (*syscalls[])(void) = {
    // ... existing entries ...
    [SYS_settickets]  sys_settickets,
    [SYS_getpinfo]    sys_getpinfo,
    [SYS_setpriority] sys_setpriority,
};
```

Add to `user/user.h`:

```
int settickets(int);
int getpinfo(struct pstat*);
int setpriority(int);
```

Add to `user/usys.S`:

```
SYSCALL(settickets)
SYSCALL(getpinfo)
SYSCALL(setpriority)
```

Testing and Grading

Deliverables

1. **Modified xv6 source code** (as zip file)
2. **Report** (10%):
    - Explanation of modifications
    - Screenshots of code changes
    - Screenshots of test output
    - Graphs showing scheduler behavior
3. **Presentation** (20%)
4. **Code and Documentation** (70%)

Deadlines

- Submission: October 31, 11:59 PM
- Presentation: October 31, class time
- **No late presentations = automatic zero**

Testing Requirements

Both schedulers must demonstrate:

- Correct proportional CPU allocation
- Graph showing relationship between tickets/priority and CPU time
- Multiple test cases with different configurations
- Proper handling of edge cases (e.g., 1 ticket, priority 0, priority 200)

Common Pitfalls

1. **Not handling zero total tickets**: Check if any RUNNABLE processes exist
2. **Integer overflow in random()**: Use appropriate modulo operations
3. **Forgetting ticket inheritance**: Child must inherit parent's tickets in `fork()`
4. **Race conditions**: Always use `ptable.lock` when accessing process table
5. **Pointer validation**: Check user-space pointers with `argptr()` before use
6. **Timer tick confusion**: Remember scheduler runs on every ~10ms tick

7. **Round-robin within priority**: Must implement fair scheduling among same-priority processes

---

Additional Resources

- xv6 Book Chapter 5 (Scheduling)
- Support videos on Canvas → Resources
- Original xv6 source code: Canvas → Resources
- **Do NOT mix Part A and Part B code** - use separate xv6 copies