

Lab 2: Process Scheduling in xv6

1. Introduction

This lab implements two new scheduling algorithms in xv6: Lottery scheduling and Stride scheduling. The original xv6 uses a simple round-robin scheduler that selects processes in order. We replace it with proportional-share schedulers that allocate CPU time based on process priorities represented by tickets.

2. Design Overview

2.1 Lottery Scheduling

Lottery scheduling assigns tickets to each process. The more tickets a process has, the higher probability it gets selected. The scheduler randomly picks a winning ticket and selects the process whose ticket range contains the winning number.

2.2 Stride Scheduling

Stride scheduling uses a deterministic approach. Each process has a stride value calculated as `BIG_CONSTANT / tickets`. The scheduler always picks the process with the smallest pass value. After running, the process's pass value is incremented by its stride.

3. Implementation Details

3.1 Data Structure Modifications

File: `kernel/proc.h`

Added four fields to the `struct proc`:

```
int tickets;           // Lottery tickets
int stride;            // Stride value for stride scheduling
int pass;              // Stride Pass for stride scheduling
int sched_count;        // Number of times this process has been scheduled
```

- `tickets`: Number of lottery tickets (1-10000)
- `stride`: Calculated as `10000 / tickets`
- `pass`: Current pass value, starts at `stride`
- `sched_count`: Counter incremented each time the process is scheduled

3.2 Process Initialization

File: `kernel/proc.c` - `allocproc()`

When a new process is created, initialize scheduler fields:

```

p->tickets = 7; // Default 7 tickets
p->stride = 10000 / p->tickets;
p->pass = p->stride;
p->sched_count = 0;

```

File: `kernel/proc.c` - `fork()`

Child processes inherit parent's scheduler fields:

```

np->tickets = p->tickets;
np->stride = p->stride;
np->pass = p->pass;
np->sched_count = 0; // Child starts with 0 scheduling count

```

3.3 Lottery Scheduler Implementation

File: `kernel/proc.c` - `scheduler()`

The lottery scheduler works in three steps:

1. **Calculate total tickets:** Sum all tickets from RUNNABLE processes
2. **Generate winning ticket:** Use random number generator: `winning_ticket = rand() % total_tickets`
3. **Select process:** Find the process whose cumulative ticket range contains the winning ticket

Key implementation details:

- Uses LFSR (Linear Feedback Shift Register) for random number generation
- Two-pass algorithm: first pass calculates total, second pass selects process
- Maintains lock on chosen process until `swtch()` completes to avoid race conditions
- Double-checks process state before scheduling

```

// Random number generator (LFSR)
unsigned short lfsr = 0xACE1u;
unsigned short rand() {
    bit = ((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5)) & 1;
    return lfsr = (lfsr >> 1) | (bit << 15);
}

```

3.4 Stride Scheduler Implementation

File: `kernel/proc.c` - `scheduler()`

The stride scheduler:

1. Finds the RUNNABLE process with minimum `pass` value
2. Updates `pass` value: `pass += (BIG_CONSTANT / tickets)` after scheduling
3. Uses `BIG_CONSTANT = 10000` for stride calculation

Important implementation details:

- Initializes `min_pass = 0x7FFFFFFF` (maximum int) to ensure first process is found
- Must release previous chosen process's lock when updating `chosen` to avoid lock leak
- Maintains lock on chosen process until `swtch()` completes

Critical bug fix: When updating `chosen` to a new process, we must release the previous `chosen` process's lock:

```
if(p->state == RUNNABLE && p->pass < min_pass) {
    min_pass = p->pass;
    if(chosen != 0) {
        release(&chosen->lock); // Release previous chosen's lock
    }
    chosen = p;
    // Keep holding p->lock
}
```

3.5 System Calls

File: `kernel/syscall.h`

```
#define SYS_sched_statistics 25
#define SYS_sched_tickets 26
```

File: `kernel/sysproc.c`

`sys_sched_statistics()`: Prints scheduling statistics for all processes

- Format: `PID(name) : tickets: xxx, ticks: xxx`
- Traverses all processes and prints their tickets and scheduling count
- Uses process locks to protect data access

`sys_sched_tickets(int n)`: Sets calling process's ticket value

- Validates ticket value (1-10000)
- Returns 0 on success
- Silently ignores invalid values

File: `kernel/proc.c`

`print_sched_statistics()`: Kernel function that prints statistics

```

void print_sched_statistics(void) {
    struct proc *p;
    for(p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if(p->state != UNUSED) {
            printf("%d(%s): tickets: %d, ticks: %d\n",
                   p->pid, p->name, p->tickets, p->sched_count);
        }
        release(&p->lock);
    }
}

```

`exec_sched_tickets(int n)`: Kernel function that sets tickets

```

void exec_sched_tickets(int n) {
    struct proc *p = myproc();
    if(n < 1 || n > 10000) {
        return; // Invalid ticket value
    }
    acquire(&p->lock);
    p->tickets = n;
    release(&p->lock);
}

```

3.6 Scheduling Count Tracking

Every time a process is scheduled, `sched_count` is incremented. This happens in all three schedulers:

- Lottery scheduler: Increment before `swtch()`
- Stride scheduler: Increment before `swtch()`
- Round-robin scheduler: Increment before `swtch()`

This allows tracking how many times each process has been scheduled, regardless of which scheduler is used.

4. Key Technical Challenges

4.1 Race Condition Prevention

Problem: Between calculating total tickets and selecting a process, process states may change.

Solution:

- Hold lock on chosen process until `swtch()` completes
- Double-check process state before scheduling
- Release lock only after process is scheduled or state check fails

4.2 Lock Management

Problem: In Stride scheduler, when updating `chosen` to a new process, the previous `chosen` process's lock was not released, causing lock leak and deadlock.

Solution: Always release previous `chosen` process's lock before updating to a new process:

```
if(chosen != 0) {
    release(&chosen->lock);
}
chosen = p;
```

4.3 Stride Scheduler Initialization

Problem: Initializing `min_pass = 0` made the condition `p->pass < min_pass` always false since all passes are positive.

Solution: Initialize `min_pass = 0xFFFFFFFF` (maximum int value) to ensure first process is always found.

5. Files Modified

1. `kernel/proc.h`: Added scheduler fields to `struct proc`
2. `kernel/proc.c`:
 - o Modified `allocproc()`: Initialize scheduler fields
 - o Modified `fork()`: Copy scheduler fields from parent
 - o Modified `scheduler()`: Implemented Lottery and Stride schedulers
 - o Added `rand()`: LFSR random number generator
 - o Added `print_sched_statistics()`: Print scheduling statistics
 - o Added `exec_sched_tickets()`: Set process tickets
 - o Modified all schedulers: Increment `sched_count` on scheduling
3. `kernel/syscall.h`: Added system call numbers
4. `kernel/syscall.c`: Registered new system calls
5. `kernel/sysproc.c`: Implemented system call handlers
6. `kernel/defs.h`: Added function declarations
7. `user/user.h`: Added user-space function declarations
8. `user/usys.pl`: Generated system call stubs

6. Testing

The implementation was tested using `lab2_test` program:

- Creates multiple child processes with different ticket values
- Runs for specified time
- Prints scheduling statistics showing tickets and ticks for each process

- Verifies that scheduling proportions match ticket ratios

```
make clean  
make qemu LAB2=STRIDE
```

Then run `./lab2_test [SLEEP] [N_PROC] [TICKET1] [TICKET2] ...` to start the test program. Example output:

```
xv6 kernel is booting  
  
Stride scheduler  
init: starting sh  
$ ./lab2_test 100 3 30 20 10  
1(init): tickets: 7, ticks: 24  
2(sh): tickets: 7, ticks: 14  
3(lab2_test): tickets: 7, ticks: 21  
4(lab2_test): tickets: 30, ticks: 53  
5(lab2_test): tickets: 20, ticks: 36  
6(lab2_test): tickets: 10, ticks: 19  
$ QEMU 8.2.2 monitor - type 'help' for more information  
(qemu) quit
```

7. Conclusion

This lab successfully implements two proportional-share scheduling algorithms in xv6. The Lottery scheduler uses probabilistic selection based on tickets, while Stride scheduler provides deterministic proportional allocation. Both schedulers correctly track scheduling counts and support dynamic ticket assignment through system calls. The implementation handles race conditions and lock management carefully to ensure correctness.