xv6 Lab: Implementing Priority Scheduling

CSL301 - Operating System

Take Home Assignment -3

Objective: Implement Static Priority Scheduling in xv6

Your Task

You will modify the xv6 scheduler to use a static priority-based algorithm. This involves:

- Adding a priority field to the process control block (`struct proc`).
- Creating a new system call, `setpriority()`, that allows a user program to change its priority.
- Rewriting the kernel scheduler to always choose the highest-priority runnable process.

Why is this important?

This core OS concept lets the scheduler prioritize critical tasks, ensuring high-priority processes get more CPU time.

Files You Will Modify

Kernel Source

- `proc.h`
- `proc.c`
- `trap.c`
- `syscall.h`
- `syscall.c`
- `sysproc.c`

User-space Interface

- `user.h`
- `usys.S`

Testing & Building

- `prioritytest.c` (create this)
- `Makefile`

Task 1: Update `proc.h` — Add Priority Field

Your Task

Modify the core process structure to store the new information you want to track. Add a new integer field to `struct proc`.

```
// In proc.h, inside struct proc:
struct proc {
  // existing code
  _____  // New field for process priority
};
```

Hint

You are adding a new member to a C struct. Choose a descriptive name, and remember: a lower value means higher priority.

Task 2: Initialize Priority in `proc.c`

Your Task

A process's priority must be initialized when a new process structure is first allocated. The ideal place to set a default priority is within the `allocproc()` function.

```
1       // in proc.c, within allocproc()
2       if (p->state == UNUSED) {
3         p->state = EMBRYO;
4         p->pid = nextpid++;
5         ____________// Set a default priority
6         return p;
7       }
```

Task 2a: Implement Priority Scheduler in `proc.c`

Your Task

You will replace the existing `scheduler()` function with a new implementation that selects the highest-priority runnable process. The process with the lowest integer value for its priority will be chosen to run.

```
1  // in proc.c, replace the scheduler() function
2    void scheduler(void) {
3        struct proc* p;
4        struct cpu* c = mycpu();
5        c->proc = 0;
6
7        for (;;) {
8          // Enable interrupts on this processor.
9          sti();
10
11         struct proc* highest_priority_p = 0;
12         int highest_priority = 1000; // Use a high number to ensure any
               priority is lower.
13
14         // Loop over process table to find highest priority runnable process.
15         acquire(&ptable.lock);
```

Task 2b: Implement ....

```
1          // continue
2           for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
3             if (p->state == RUNNABLE) {
4               if (p->priority < highest_priority) {
5                 ____________________//update highest priority
6                 highest_priority_p = p;
7               }
8             }
9           }
10
11          if (highest_priority_p != 0) {
12            p = highest_priority_p;
13
14            // Switch to chosen process. It is the process's job
15            // to release ptable.lock and then reacquire it
16            // before jumping back to us.
17            c->proc = p;
18            switchuvm(p);
19            p->state = RUNNING;
20            swtch(&(c->scheduler), p->context);
21
22            // Process is done running for now.
23            // It should have changed its p->state before coming back.
24            c->proc = 0;
25          }
26          release(&ptable.lock);
27        }
28      }
```

Task 2c: Implement ‘setpriority‘ in `sysproc.c`

Your Task

You will create a new function, `sys_setpriority()`, that allows a user program to change its own priority. This function will take a single integer argument and update the current process's priority field.

```
1  // in sysproc.c
2  int sys_setpriority(void) {
3    int priority;
4    if (argint(0, &priority) < 0) {
5      return -1;
6    }
7    ----------------------//Assign the given priority value to the calling
         process's priority field
8    return 0;
9  }
```

Task 3: Update System Call Files

Your Task

Your final kernel modifications involve declaring the new system call and mapping it to its function.

```
// in syscall.h
// Add a new system call number for setpriority
#define SYS_setpriority 22

// in syscall.c
extern int sys_setpriority(void); // Declare the function

static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
// ... other syscalls ...
[SYS_setpriority] sys_setpriority, // Add to the syscalls array
};
```

Task 4: User-Space Interface

Your Task

Your final step is to make the new system call available to user programs. This involves adding its declaration to a header file and creating an assembly stub for the system call trap.

```
1  // in user.h
2  int setpriority(int); // Add this declaration
3
4  // in usys.S
5  #include "syscall.h"
6  #include "traps.h"
7  SYSCALL(setpriority)
```

Task 5: Create User-Space Test Program

Your Task

You will now create a user-space program to demonstrate priority scheduling. You'll also update the `Makefile` to compile and include your new program in the XV6 filesystem.

```
1  // in prioritytest.c
2  #include "types.h"
3  #include "stat.h"
4  #include "user.h"
5
6  int main(int argc, char* argv[]) {
7    int pid1, pid2;
8    printf(1, "Starting priority scheduling test...\n");
9
10   pid1 = fork();
11   if (pid1 == 0) {
12     setpriority(10); // High priority
13     printf(1, "Child 1 (pid %d) with high priority (10) started.\n",
           getpid());
14     for (int i = 0; i < 50000000; i++) {} // Busy loop
15     printf(1, "Child 1 finished.\n");
16     exit();
17   }
```

Task 5: Create User-Space .....

```
1   // in prioritytest.c
2     pid2 = fork();
3     if (pid2 == 0) {
4       setpriority(50); // Low priority
5       printf(1, "Child 2 (pid %d) with low priority (50) started.\n",
              getpid());
6       for (int i = 0; i < 50000000; i++) {} // Busy loop
7       printf(1, "Child 2 finished.\n");
8       exit();
9     }
10
11    wait();
12    wait();
13    printf(1, "Priority scheduling test complete.\n");
14    exit();
15  }
```

Task 6: Build and Run

Your Task

You will now compile the modified XV6 kernel and run it on QEMU. Then, from the XV6 shell, you'll execute your test program to verify the new scheduling behavior.

```
# In Makefile, add _prioritytest to the UPROGS list
UPROGS=\
...
_prioritytest\

# In your shell, from the xv6 root directory
make clean
make
make qemu
$ prioritytest
```

Submission Checklist

- Make sure your code compiles without warnings or errors.
- The `prioritytest` program should run and produce sensible output.
- Submit all the kernel and user-space codes that you have modified in a text file.
- Attach screenshots of your final output.

Good Luck!