COL331 Assignment 2 Report
Priority Scheduler

Utkarsh Singh, 2015ME10686

March 29, 2018

Objective

Continuing from Assignment 1, to modify the xv6 kernel to achieve the following functionalities:

1. Add Process Priority
   (a) Modify the sys_ps() system call to print the priority of each process.
   (b) Introduce a sys_setpriority() system call to set the priority of a process to a new value, given its pid.
2. Replace the Round-Robin scheduler with a simple Priority Scheduler.
3. Starvation
   (a) Introduce a sys_getpriority() system call to print the priority of a process, given its pid.
   (b) Handle starvation

Procedure

Each part is implemented in the order mentioned above.

Part 1(a)

The sys_ps() system call was implemented before as a part of Assignment 1. To introduce the notion of priority in xv6, firstly modify the struct definition of process in file *proc.h*. Modified **struct proc** is given below.

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
```

```
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int priority;                // Process priority
};
```

int priority has been included.

After this, we need to make sure that a new process is always given a default priority of 5. To do this, include the following line in functions userinit() and allocproc(), in file *proc.c*.

```
p->priority = 5;
```

In case of fork(), we set the priority of the child process to be the same as that of the parent process.
Finally, modify the function get_pid_name() which is used by the sys_ps() system call to print the process details. We now print the name, pid, priority and state of the process.

```
void
get_pid_name(void)
{
  struct proc *p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    int check = p->state;
    if(check == UNUSED || check == EMBRYO || check == ZOMBIE)
      continue;
    char *state_name;
    if(p->state == RUNNABLE)
      state_name = "RUNNABLE";
    else
    if(p->state == RUNNING)
      state_name = "RUNNING";
    else
      state_name = "SLEEPING";
    cprintf("pid:%d name:%s state:%s priority:%d\n",
                    p->pid, p->name, state_name, p->priority);
  }
}
```

This finishes Part 1(a).

Part 1(b)

For this, we have to declare and define the sys_setpriority system call. setpriority() will be the user function that will call the system call. This is done as follows.

1. Add the following in *syscall.c*.

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

2. Add the following in *syscall.h*.

   ```
   #define SYS_setpriority 25
   ```

3. Add the following to the array of functions in *syscall.c*.

   ```
   [SYS_setpriority]  sys_setpriority
   ```

4. Include sys_setpriority in the array of system call names in *syscall.c*.

5. Now let's define the sys_setpriority system call. These definitions will be given in *sysproc.c*. Add the following snippet to this file.

   ```
   int
       sys_setpriority(int id, int new_priority)
       {
        argint(0, &id);
    argint(1, &new_priority);
    set_priority(id, new_priority);
        return 0;
       }
   ```

6. Define the set_priority() function in *proc.c*. It modifies the priority with process íd́with value ńew_priority. Error message is printed for invalid id.

   ```
   void set_priority(int id, int new_priority)
       {
         if(new_priority < 1 || new_priority > 20)
       cprintf("ERROR: Value of priority can only lie in
               between 1 and 20\n");
     struct proc *p;
     int find = 0;
         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
         {
       int check = p->state;
       if(check == UNUSED || check == EMBRYO || check == ZOMBIE)
         continue;
       if(p->pid == id)
       {
         p->priority = new_priority;
         find = 1;
       }
     }
   ```

```
        if(find != 1)
          cprintf("ERROR: Cannot set priority because invalid pid\n");
      }
```

7. Add the following in *usys.S*.

```
SYSCALL(setpriority)
```

8. Finally add the following in *user.h*

```
int setpriority(int, int);
```

This completes Part 1(b).

Part 2

For this, we have to implement the priority based scheduler. We will modify the function scheduler() in *proc.c*. The modified function is shown below.

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

  for(;;){
    // Enable interrupts on this processor.
    sti();

    acquire(&ptable.lock);
    //First, find the maximum priority value
    int max_priority = -1;

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

    // Loop over process table looking for process to run.
    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.
      if(p->priority == max_priority)
      {
        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);

  }
}
```

What we do here, is that when the lock has been acquired for the ptable, we first find the value of the maximum priority existing in the ptable. After this, we will simply execute all the processes with the maximum priority in Round-Robin fashion.
This completes Part 2.

Part 3(a)

For Part 3, we use *proc_3.c* instead of *proc.c*. For convenience, copy the contents of *proc.c* into the new file.
For this, we have to declare and define the sys_getpriority system call. getpriority() will be the user function that will call the system call. This is done as follows.

1. Add the following in *syscall.c*.

   ```
   extern int sys_getpriority(void);
   ```

2. Add the following in *syscall.h*.

   ```
   #define SYS_getpriority 26
   ```

3. Add the following to the array of functions in *syscall.c*.

   ```
   [SYS_getpriority]   sys_getpriority
   ```

4. Include sys_getpriority in the array of system call names in *syscall.c*.

5. Now let's define the sys_getpriority system call. These definitions will be given in *sysproc.c*. Add the following snippet to this file.

```
int
    sys_getpriority(int id)
    {
      argint(0, &id);
      int priority = get_priority(id);
      return priority;
    }
```

6. Define the get_priority() function in *proc_3.c*. Error message is printed for invalid id.

```
int get_priority(int id)
    {
      struct proc *p;
  int ret_priority = -1;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    int check = p->state;
    if(check == UNUSED || check == EMBRYO || check == ZOMBIE)
      continue;
    if(p->pid == id)
    {
      ret_priority = p->priority;
    }
  }
  if(ret_priority == -1)
  {
    cprintf("ERROR: Cannot set priority because invalid pid\n");
    return -1;
  }
  else
  {
    return -1;
  }
}
```

7. Add the following in *usys.S*.

```
SYSCALL(get_priority)
```

8. Finally add the following in *user.h*

```
int get_priority(int);
```

This completes Part 3(a).

Part 3(b)

To handle starvation, we maintain a counter associated with each process. When a process is context switched in, we increase the value of the counter of that process by 1. Whenever the value of the counter associated with a process hits

50, we update the priority of every other process.
To start off, we define the counter. Modify the struct for ptable in file *proc_3.c*.

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
  int counter[NPROC];     //Counter defined
} ptable;
```

Wherever a new process is created, initialize counter value associated with that process' pid to 0.
Update the scheduler() function in *proc_3.c* as follows.

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

  for(;;){
    // Enable interrupts on this processor.
    sti();

    acquire(&ptable.lock);
    //First, find the maximum priority value
    int max_priority = -1;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if(p->state != RUNNABLE)
      {
        continue;
      }
      if(ptable.counter[p->pid] == 50)
      {
        struct proc *temp_p;
        for(temp_p = ptable.proc; temp_p < &ptable.proc[NPROC]; temp_p++)
        {
          if(temp_p->pid != p->pid)
          {
            if(temp_p->priority <20)
              temp_p->priority = temp_p->priority+1;
          }
        }
        ptable.counter[p->pid] = 0;
      }
    }

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

```
      {
        continue;
      }
      if(max_priority < p->priority)
      {
        max_priority = p->priority;
      }
    }

    // Loop over process table looking for process to run.
    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.
      if(p->priority == max_priority)
      {
        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.
        ptable.counter[p->pid]++;
        c->proc = 0;
      }
    }
    release(&ptable.lock);

  }
}
```

This completes Part 3(b), and also the assignment.