

Project 4: Priority-based Scheduling in Xv6

Due date: Midnight of Wednesday Mar 9, 2022

Project objective:

- Understand how a real operating system (Xv6) works.
- Implement Priority-based Round-robin Scheduling in Xv6.
- Practice implementing real Operating Systems in C and Makefiles.
- Practice code comprehension.

Project overview:

In this assignment, you will modify a real Operating System (Xv6) to implement **Priority-based Round-robin Scheduling**. This process will consist of:

- Adding priority to Xv6 processes.
- Modifying the PS system call from class to show priority.
- Writing your own NICE system call to control the priority of a process.
- Modifying the existing scheduling algorithm to incorporate priority.
- Write a simple multi-processing program to test your system call and scheduler.

Finally, you will write a **report** to showcase your work with text and screenshots.

Process Status System Call:

Start by implementing the *PS* system call as we did in class. Of course, the priority part is missing, but you will add that later in this project. Here are the instructions for adding the *PS* system call to your Xv6 Operating System:

1. Open the file “syscall.h” and add “#define SYS_ps 22” to the list of system calls.
2. Open the file “defs.h” and add “int ps(void);” under the proc.c comment.
3. Open the file “user.h” and add “int ps(void);”
4. Add the following code to the end of the file “proc.c”

```
int
ps()
{
    struct proc *p;

    //enable interrupts on processor
    sti();
```

```

//loop over process table
acquire(&ptable.lock);

cprintf("name \t pid \t state \t priority\n");

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

    if (p->state == SLEEPING)
        cprintf("%s \t %d \t SLEEPING \t priority \n", p->name, p->pid);
    else if (p->state == RUNNING)
        cprintf("%s \t %d \t RUNNING \t priority \n", p->name, p->pid);
    else if (p->state == RUNNABLE)
        cprintf("%s \t %d \t RUNNABLE \t priority \n", p->name, p->pid);

}
release(&ptable.lock);

return 0;
}

```

5. Open the file “sysproc.c” and add the following code to its end:

```

int
sys_ps(void)
{
    return ps();
}

```

6. Open “usys.s” and add “SYSCALL(ps)” to it.
7. Open “sys_call.c” and add the lines to the matching lists:

```

extern int sys_ps(void);
[SYS_ps]    sys_ps,

```

8. Create the file “ps.c” and add to it:

```

#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

```

```
int
main(void)
{
ps();
exit();
}
```

9. Open the Makefile and do the following:
 - a. Add “`_ps\`” under UPROGS in the Makefile
 - b. Add “`ps.c`” under EXTRAS in the Makefile
10. Compile everything by typing *make*
11. Run Xv6 by typing *make qemu-nox*
12. Test your new system call inside Xv6 by typing *ps*

Write the Multiprocessing file foo.c:

The goal here is to create a program that uses fork to create multiple CPU-bound processes. This will help us test our scheduler and system calls. The code for foo.c is provided, make sure you test it with the previously developed *PS* system call. Here are the steps to get the foo program to work under Xv6:

1. Create a new file in your Xv6 directory and name it “foo.c”
2. Open “foo.c” and write the following code in it:

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

int main(int argc, char *argv[]) {
    int pid;
    int k, n;
    int x, z;

    if(argc < 2)
        n = 1; //Default
    else
        n = atoi(argv[1]);
    if (n < 0 || n > 20)
        n = 2;

    printf(1, "creating %d processes!\n", n);
```

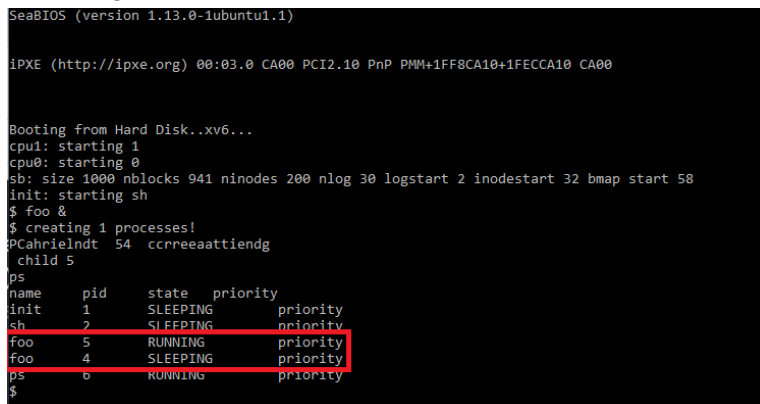
```

x = 0;
pid = 0;

for ( k = 0; k < n; k++ ) {
    pid = fork ();
    if ( pid < 0 ) {
        printf(1, "%d failed in fork!\n", getpid());
    } else if (pid > 0) {
        // parent
        printf(1, "Parent %d creating child %d\n",getpid(), pid);
        wait();
    }
    else{
        printf(1,"Child %d created\n",getpid());
        for(z = 0; z < 4000000000; z+=1)
            x = x + 3.14*89.64; //Useless calculation to consume CPU Time
        break;
    }
}
exit();
}

```

3. Open the Makefile and do the following:
 - a. Add “`_foo\`” under UPROGS in the Makefile
 - b. Add “`foo.c`” under EXTRAS in the Makefile
4. Compile everything by typing *make*
5. Run Xv6 by typing *make qemu-nox*
6. Type “`foo &`” to run the foo program and return to the command-line.
7. Type “`ps`” while foo is still running to see the two foo processes created. It should look something like this:



```

SeaBIOS (version 1.13.0-1ubuntu1.1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00

Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ foo &
$ creating 1 processes!
pCahrielndt 54 ccrreeaattindg
  child 5
ps
name  pid  state  priority
init   1   SLEEPING
sh     2   SLEEPING
foo    5   RUNNING
foo    4   SLEEPING
ps     6   RUNNING
$

```

8. Take a screenshot showing your *foo* programs running for your report.

Adding Priority to Xv6 Processes:

Now that we have the PS system call and a multiprocessing test program (foo.c) we can modify the process control block in Xv6 to include a priority between 0 and 20 for each process. Also, we want to modify the scheduler to schedule processes based on priority. To do that we must do the following:

1. Open “proc.h” and add an integer called *priority* to the proc struct.
2. Open “proc.c” and find the *allocproc* function which acts like a constructor for allocating processes.
3. Under the “found” section of the allocproc function, add priority and assign the value 10 (follow the same structure of the existing code where *p* is the process).
4. Open “exec.c” and locate the line: *curproc->tf->esp = sp;*
Under the line add the following code: *curproc->priority = 2;*
This gives the child process higher priority than its parent (lower number = higher priority).

Adding the Nice System call:

Now we are ready to add the NICE system call! As you remember, the NICE system call is responsible for changing the priority of a process while it is running using its process ID. Here are the steps to do that:

1. Open “syscall.h” and add “#define SYS_chpr 23” to the end of the list (chpr stands for change priority).
2. Open “defs.h” and add “int chpr(int pid, int priority);” under the proc.c comment.
3. Open “proc.c” and write a function at the end of the file according to the following pseudocode:

```
int
chpr(int pid, int priority)
{
    struct proc *p;

    acquire the lock to the ptable

    loop over all processes in the system
        if the process ID is equal to pid
            update its priority to priority
            break;

    release ptable lock

    return pid;
}
```

4. Open “sysproc.c” and add the following code to the end of the file:

```
int
sys_chpr (void)
{
    int pid, pr;
    if (argint(0, &pid) < 0)
        return -1;
    if (argint(1, &pr) < 0)
        return -1;

    return chpr(pid, pr);
}
```

5. Open “usys.s” and add “SYSCALL(chpr)” to the end of the list.
6. Open “syscall.c” and add the following lines to the matching part of the code:

```
extern int sy_chpr(void);
[SYS_chpr]      sys_chpr,
```

7. Create a new file called “nice.c” and the following code:

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

int main(int argc, char *argv[]) {
    int pid, priority;

    if(argc < 3){
        printf(1, "usage: nice pid priority\n");
        exit();
    }

    pid = atoi(argv[1]);
    priority = atoi(argv[2]);

    if (priority < 0 || priority > 20){
        printf(2, "invalid priority (0-20)!\n");
        exit();
    }
}
```

```
chpr(pid, priority);
exit();
}
```

8. Open the Makefile and do the following:
 - a. Add “`_nice\`” under UPROGS in the Makefile
 - b. Add “`nice.c`” under EXTRAS in the Makefile
9. Compile everything by typing `make`
10. Run Xv6 by typing `make qemu-nox`
11. Make sure `ps` and everything else still work
12. Take a screenshot showing how *nice* works for your report.

Updating the scheduler:

This is the main part of the project where you modify the existing scheduler to use priority. To update the scheduler, start by reading and understanding the existing scheduler code in the file “`proc.c`” then think about the changes needed to scheduler processes based on priority. Here are the steps needed:

1. Open “`proc.c`” and locate the *scheduler* function.
2. Inspect the scheduler code and understand every part of it, check the [Xv6 book](#) if needed.
3. Update the scheduler to pick the runnable process with the highest priority, maintaining the rest of the code.
4. Take a screenshot of your scheduler code and add it to your report.

This part does not involve modifying the Makefile or anything else. I am happy to help in this step if you run into any issues, but give it your best attempt first.

Testing:

Once ready, let’s update `ps` to show priority by going to “`proc.c`” and locating the `ps` function. Instead of printing the word priority, modify the code to print the priority of each process. After that, let’s compile and run Xv6 and do the following to test if priority scheduling and the *nice* system call are working:

1. Run `foo` as we did before.
2. Run `ps` to check the priority of `foo` processes.
3. Change the priority of one of the `foo` processes using the *nice* system call.
4. Run `ps` again to verify that the process with the highest priority runs more.
5. Take a screenshot of that for your report.

Report:

Organize your report as follows, maintaining a coherent document that includes screenshots and text to communicate the objective of your project:

1. Abstract: A brief summary of the project, in your own words. This should be no more than a few sentences. Give the reader context and identify the key purpose of the assignment.
2. Results: A section that goes over the system calls you implemented and the scheduling algorithm you modified. **Make sure to include screenshots of each part of this project.**
3. Discussion: A section that interprets and describes the significance of your findings focussing on the results.
4. Extensions: Describe any extensions you undertook, including text output, graphs, tables, or images demonstrating those extensions.
5. References/Acknowledgements.

Extensions:

You can be creative here and come up with your own extensions. I will suggest the following ideas:

- Implement lottery scheduling in Xv6.
- Implement new, useful system calls to your operating system.
- Add a username and a password to your Xv6 (no problem if password is stored in plain text).
- Implement Completely Fair Scheduler in Xv6. This involves implementing Red-Black Trees in C. 30 Points are possible with this extension!

Project submission:

- Add all your files to your ***project_4*** directory on Google Drive.
- Add your report to the same folder.
- Copy the rubric from Moodle to the project directory after you review it.