**Lab 2 Report**

Alp Can Aloglu, Najmeh Mohammadi Arani

SID : 862190749 / 862216499

E-mail: aalog001@ucr.edu / nmoha034@ucr.edu

- **All code changes you made on xv6 source code** (hint: `diff -r original_xv6 your_xv6`)

  Code diff is submitted through different submissions.

- **Detailed explanation about these changes**

  **defs.h**

```
122a123,124
> void            set_prior(int);
> int           get_prior(void);
```

  Added forward declarations for set_prior and get_prior functions.

  **exec.c**

```
101a102,107
>
>   //Time read for T_start
>   acquire(&tickslock);
>   curproc->T_start = ticks;
>   release(&tickslock);
>
```

  Acquired tickslock, saved the current tick to current processes T_start feature.

  **Makefile**

```
183a184,188
>         _test\
>         _test1\
>         _proc1\
>         _proc2\
>         _proc3\
220c225
```

```
< CPUS := 2
---
> CPUS := 1
```

Added the test functions to MakeFile under UPROGS. Changed CPU count to 1 as instructed.

**Proc.c**

```
90a91,94
>   // Lab2 prior value initialization with default 16
>   p->prior_val = 16;
>   p->burstCount = 0;
>   p->selected_To_Run = 0;
```

90a91,94:

In the allocproc function we initialize prior_val of current proc with default value of 16, burstCount with 0 and selected_To_Run with 0.

```
202a207,209
>   // Lab2
>   np->prior_val = curproc->prior_val;
```

202a207,209:

In fork function, children inherit the priority value of the parent process.

```
236a244,254
>   //Read ticks for T_finish
>   //Added for Lab 2
>   acquire(&tickslock);
>   curproc->T_finish = ticks;
>   release(&tickslock);
>   // Output for Turnaround
>   // Added for Lab 2
>   cprintf("Turnaround time for PID: %d is %d\n",curproc->pid,
```

```
(curproc->T_finish - curproc->T_start));
>   cprintf("Burst time time for PID: %d is %d\n",curproc->pid,
curproc->burstCount);
>   cprintf("Waiting time for PID: %d is %d\n",curproc->pid, ((curproc->T_finish -
curproc->T_start) - curproc->burstCount));
>
```

236a244,254:

In the exit function, we save the ticks to the current process' T_finish feature. Prints out the turnaround time by subtracting T_start from T_finish. Prints out burstCount, which is how many times the current process is scheduled. Prints out waiting time, which is Turnaround time we calculated minus burst count.

```
325c343
<   struct proc *p;
---
>   struct proc *p, *temp;
328d345
<
```

325c343:

Added proc pointer named temp to loop through ptable to find highest priority process in the ptable.

```
335,337c352,369
<     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
<       if(p->state != RUNNABLE)
<         continue;
---
>       temp = ptable.proc;
```

```
>      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
>         if (p->state != RUNNABLE) {
>            continue;
>         }
>         if(temp->selected_To_Run == p->selected_To_Run){
>            p->selected_To_Run = 1;
>            temp = p;
>            continue;
>         }
>         if (temp->prior_val > p->prior_val){
>            temp->prior_val = (temp->prior_val > 0) ? temp->prior_val - 1 : 0;
>            temp->selected_To_Run = 0;
>            temp = p;
>            temp->selected_To_Run = 1;
>            continue;
>         }
>         p->prior_val = (p->prior_val > 0) ? p->prior_val - 1 : 0;
338a371
>      }
```

335,337c352,369:

We initialize temp with the first proc in the ptable. We use selected_To_Run to keep track of which process is selected to run in the current cycle. Also, the fact that selected_To_Run is initialized with 0 at the beginning of all processes' allocation helps us determine the first runnable process.

First if statement after runnable check compares the temp's selected_To_Run feature to p's selected_To_Run. If they are the same it means we are at the beginning of the loop and all processes have selected_To_Run  set to 0. In the if statement we set selected_To_Run of the current process to 1, and assign it to temp. This ensures that we avoid this if after assigning the first runnable p to temp.

Second if statement after runnable check compares prior_vals of temp and p. If temp has higher prior_val (lower priority), we decrease temp's prior_val ( aging) , and set selected_To_Run of temp to 0. Then, we assign current p with lower prior_val to temp, and set its selected_To_Run to 1.

Lastly if we don't hit any of the if statements, that means the current process has higher prior_val( low priority) so for the aging process we decrease its prior_val.

```
342,352c375,385
<       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;
<     }
---
>         c->proc = temp;
>         switchuvm(temp);
>         temp->state = RUNNING;
>         temp->prior_val = (temp->prior_val < 31) ? temp->prior_val + 1 : 31; //
Decreases the priority of currently running process
>         temp->selected_To_Run = 0; // sets flag back to 0 for next loop for
priority check
>         temp->burstCount++; // Increments the burst count for each time a
process is scheduled
>         swtch(&(c->scheduler), temp->context);
>         switchkvm();
>         // Process is done running for now.
>         // It should have changed its p->state before coming back.
>         c->proc = 0;
```

342,352c375,385:

Because we are selecting the process with the lowest priority with temp proc pointer, we changed p to temps. Then we increase prior_val of the temp for aging. Increase burstCount to keep track of how many times it is scheduled. Lastly, we set selected_To_Run back to 0 to prepare for the next scheduler cycle.

```
534a568,596
>
> // sets priority for current process
> void
```

```
> set_prior(int priority) {
>     struct proc *curproc = myproc(); // gets current process
>     acquire(&ptable.lock);
>     if (priority > 31) {
>         curproc->prior_val = 31; // sets priority for current process
>     } else if (priority < 0) {
>         curproc->prior_val = 0; // sets priority for current process
>     }else{
>         curproc->prior_val = priority;
>     }
>     release(&ptable.lock);
>     yield(); // yields to scheduler
> }
>
> // gets priority from current process
> int
> get_prior(void)
> {
>     int priority;
>     struct proc *curproc = myproc(); // gets current process
>     acquire(&ptable.lock);
>     priority = curproc->prior_val; // sets priority for current process
>     release(&ptable.lock);
>     yield(); // yields to scheduler
>     return priority;
> }
```

534a568,596

Implementation of set_prior and get_prior functions.

In set_prior, we acquire the ptable.lock to stop scheduling. We check the given priority is in our range, then assign it to prior_val of current process according to our check. We release the ptable.lock. We call the yield function to continue scheduling.

In get_prior, similar to set_prior we acure ptable.lock. We read the priority of the current process to a local variable. Then we release ptable.lock. We call yield to continue scheduling. At the end we return the value of the current process' priority.

**proc.h**

```
51a52,56
>   int prior_val;             // Priority value for current process
>   int T_start;               // start time for current process
>   int T_finish;              // finish time for current process
>   int burstCount;            // counter for burst
>   int selected_To_Run;       // helper variable for scheduler
```

51a52,56:

We add the features prior_val, T_start, T_finish, burstCount and selected_To_Run to proc structure. Prior_val keeps track of the priority of the process. T_start and T_finish are used to calculate the turnaround time. burstCount is used to keep track of how many times this process is scheduled to calculate wait time. selected_To_Run is a helper flag to find the highest priority process.

**Syscall.c**

```
105a106,107
> extern int sys_set_prior(void); //lab 2
> extern int sys_get_prior(void); //lab 2
128a131,132
> [SYS_set_prior]   sys_set_prior, // lab 2
> [SYS_get_prior]   sys_get_prior, // lab 2
```

105a106,107 and 128a131,132:

Implementation of system calls for get_prior and set_prior. We add these system call tables.

**syscall.h**

```
22a23,24
> #define SYS_set_prior 22
> #define SYS_get_prior 23
```

22a23,24:

We define system call numbers for set_prior and get_prior for system call table.

**sysproc.c**

```
91a92,109
>
> // sets priority for current process
> int
> sys_set_prior(void)
> {
>     int priority;
>     if(argint(0, &priority) < 0)
>         return -1;
>     set_prior(priority);
>     return 0;
> }
>
> // gets priority from current process
> int
> sys_get_prior(void)
> {
>     return get_prior();
> }
```

91a92,109:

We get the input from user space and pass it to kernel space system call.

**user.h**

```
25a26,27
> void set_prior(int); //Set prior_val for current process
> int get_prior(void); //Gets prior_val from current process
```

25a26,27:

Forward declarations of system calls set_prior and get_prior.

**usys.S**

```
31a32,33
> SYSCALL(set_prior)
> SYSCALL(get_prior)
```

31a32,33:

Sets up the connection between user space system call and kernel space system call table.

- **Screenshots about how you run the related program(s) and results**

We tested the scheduler with two methods. First we tested it by calling fork . In the second method we used three different test files (proc1, proc2, proc3) and we ran them simultaneously.

First test implementation:

For the first test, we implemented two programs: test.c and test1.c, and the only difference between them is the length of the for loops to show different aspects of the scheduler. As you can see in the following code, first we set the priority of the main process to 0 (Highest priority), then we called fork three times to make three child processes. We gave a different priority to each of the children. The children have the priority of 30, 15, and 0 respectively and they run exactly the same code implemenration. Right after each call to the fork, we print the priority of the child process at start. We also print the priority of the children 5 times in between, and finally the priority of the child at the end. This way we can keep track of how the priority of children changes. At the end we used the if statement to enforce the parent processes to wait for their children to prevent zombies.

**test.c**

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

int main(int argc, char *argv[]){
   set_prior(0);
   int pid,pid1,pid2;
   printf(1,"%d's priority %d at start \n", getpid(), get_prior());
   pid = fork();
   if(pid == 0) {
       set_prior(30);
       printf(1,"child %d's priority %d at start \n",getpid(), get_prior());
       int i, j;
       for (i = 0; i < 10; i++) {
           asm("nop");
```

```
            for (j = 0; j < 1000000; j++) {
                asm("nop");
                //printf(1,"test1's priority %d \n", get_prior());
            }
            if(i%2 == 1)
                printf(1,"child %d's priority %d in between \n",getpid(),
get_prior());
        }
        printf(1,"child %d's priority %d at end \n",getpid(), get_prior());
        exit();
    }
    pid1 = fork();
    if(pid1 == 0) {
        set_prior(15);
        printf(1,"child %d's priority %d at start \n",getpid(), get_prior());
        int i, j;
        for (i = 0; i < 10; i++) {
            asm("nop");
            for (j = 0; j < 1000000; j++) {
                asm("nop");
                //printf(1,"test1's priority %d \n", get_prior());
            }
            if(i%2 == 1)
                printf(1,"child %d's priority %d in between \n",getpid(),
get_prior());
        }
        printf(1,"child %d's priority %d at end \n",getpid(), get_prior());
        exit();
    }
    pid2 = fork();
    if(pid2 == 0) {
        set_prior(0);
        printf(1,"child %d's priority %d at start \n",getpid(), get_prior());
        int i, j;
        for (i = 0; i < 10; i++) {
            asm("nop");
            for (j = 0; j < 1000000; j++) {
                asm("nop");
                //printf(1,"test1's priority %d \n", get_prior());
            }
            if(i%2 == 1)
                printf(1,"child %d's priority %d in between \n",getpid(),
get_prior());
        }
        printf(1,"child %d's priority %d at end \n",getpid(), get_prior());
        exit();
    }
    if(pid > 0){
        wait();
    }
    if( pid1 > 0){
        wait();
    }
    if(pid2 > 0){
        wait();
    }
```

```
    exit();
}
```

**test1.c**

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

int main(int argc, char *argv[]){
    set_prior(0);
    int pid,pid1,pid2;
    printf(1,"test1's priority %d at start \n", get_prior());
    pid = fork();
    if(pid == 0) {
        set_prior(30);
        printf(1,"child %d's priority %d at start \n",getpid(), get_prior());
        int i, j;
        for (i = 0; i < 5000; i++) {
            asm("nop");
            for (j = 0; j < 100000; j++) {
                asm("nop");
                //printf(1,"test1's priority %d \n", get_prior());
            }
            if(i%1000 == 0)
                printf(1,"child %d's priority %d in between \n",getpid(),
get_prior());
        }
        printf(1,"child %d's priority %d at end \n",getpid(), get_prior());
        exit();
    }
    pid1 = fork();
    if(pid1 == 0) {
        set_prior(15);
        printf(1,"child %d's priority %d at start \n",getpid(), get_prior());
        int i, j;
        for (i = 0; i < 5000; i++) {
            asm("nop");
            for (j = 0; j < 100000; j++) {
                asm("nop");
                //printf(1,"test1's priority %d \n", get_prior());
            }
            if(i%1000 == 0)
                printf(1,"child %d's priority %d in between \n",getpid(),
get_prior());
        }
        printf(1,"child %d's priority %d at end \n",getpid(), get_prior());
        exit();
    }
    pid2 = fork();
```

```
    if(pid2 == 0) {
        set_prior(0);
        printf(1,"child %d's priority %d at start \n",getpid(), get_prior());
        int i, j;
        for (i = 0; i < 5000; i++) {
            asm("nop");
            for (j = 0; j < 100000; j++) {
                asm("nop");
                //printf(1,"test1's priority %d \n", get_prior());
            }
            if(i%1000 == 0)
                printf(1,"child %d's priority %d in between \n",getpid(),
get_prior());
        }
        printf(1,"child %d's priority %d at end \n",getpid(), get_prior());
        exit();
    }
    if(pid > 0){
        wait();
    }
    if( pid1 > 0){
        wait();
    }
    if(pid2 > 0){
        wait();
    }
    exit();
}
```

Screenshots of first test's result:

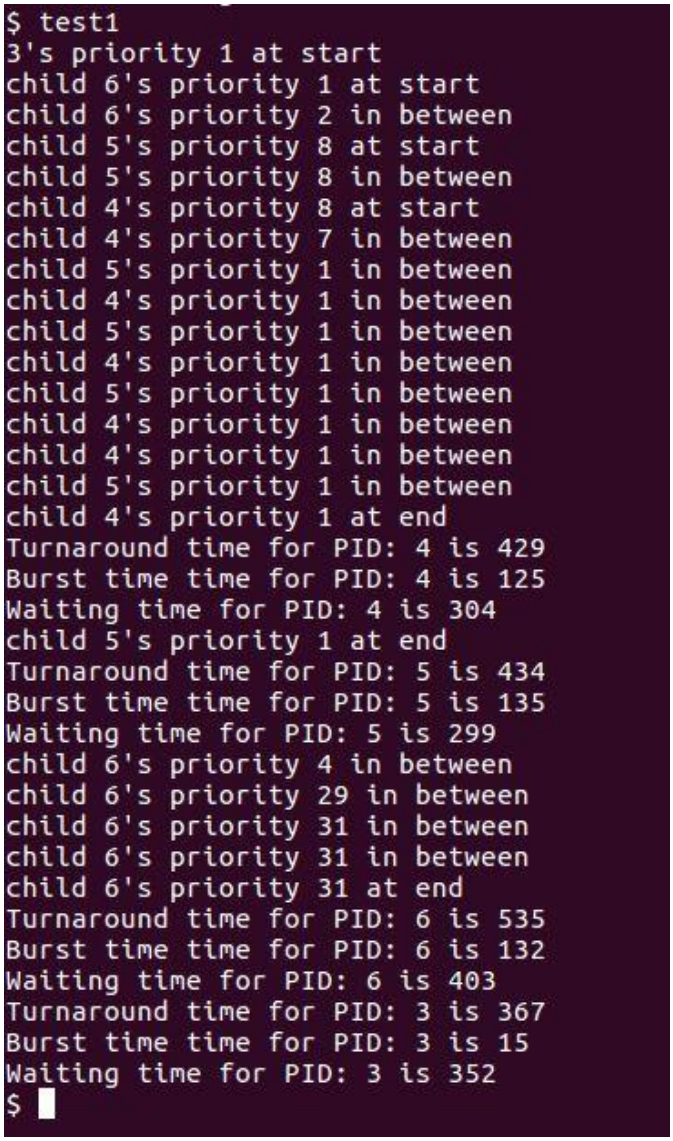
As you see in the following screenshots, the three children are identified by their pid. For test.c, the first child has a pid of 4, the second child has a pid of 5, and the third child has a pid of 6. As discussed before we gave the highest priority to the third child, and the lowest priority to the first child. Therefore as expected, the children 6 and 5 that have the higher priority would execute first and child 4 would have to wait to the point that the scheduler increases the priority of child 4 and decreases the priority of child 5 and 6, so the child 4 gets chosen to execute too.

As you can see In the test1.c screenshot, even though it had a longer nested for loop, there is no starvation, and all the processes had a chance to be chosen for execution by the scheduler.

test.c output :

```
$ test
3's priority 1 at start
child 6's priority 1 at start
child 6's priority 3 in between
child 6's priority 5 in between
child 6's priority 7 in between
chichild 6's priority 7 in between
ld 5's priority 8 at start
child 6's priority 7 in between
child 6's priority 7 at end
Turnaround time for PID: 6 is 170
child 5's priority 8 in between
Burst time time for PID: 6 is 13
Waiting time for PID: 6 is 157
child 5's priority 8 in between
child 5's priority 8 in between
child 4's priority 8 at start
child 5's priority 7 in between
child 5's priority 7 in between
child 5's priority 7 at end
Turnaround time for PID: 5 is 175
Burst time time for PID: 5 is 12
Waiting time for PID: 5 is 163
child 4's priority 8 in between
child 4's priority 6 in between
child 4's priority 8 in between
child 4's priority 10 in between
child 4's priority 12 in between
child 4's priority 13 at end
Turnaround time for PID: 4 is 179
Burst time time for PID: 4 is 15
Waiting time for PID: 4 is 164
Turnaround time for PID: 3 is 16
Burst time time for PID: 3 is 16
Waiting time for PID: 3 is 0
$
```

test1.c output:




Second test implementation:

In the second test, we created three .c files (Proc1.c, Proc2.c, Proc2.c). Inside each of them we set a different priority for the process and the rest of the code is exactly the same thing, which is a nested for loop.

**Proc1.c:**

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

```c
int main(int argc, char *argv[]) {
   set_prior(0); // N being the priority value you assign to this proc
   int i,j;
   for (i = 0; i < 46000; i++) {
       asm("nop");
       for (j = 0; j < 46000; j++) {
           asm("nop");
       }
   }
   exit();
}
```

**Proc2.c:**

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

int main(int argc, char *argv[]) {
   set_prior(20); // N being the priority value you assign to this proc
   int i,j;
   for (i = 0; i < 46000; i++) {
       asm("nop");
       for (j = 0; j < 46000; j++) {
           asm("nop");
       }
   }
   exit();
}
```

**Proc3.c:**

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

int main(int argc, char *argv[]) {
   set_prior(31); // N being the priority value you assign to this proc
   int i,j;
   for (i = 0; i < 46000; i++) {
       asm("nop");
       for (j = 0; j < 46000; j++) {
           asm("nop");
       }
   }
   exit();
}
```

Screenshots of second test's result:

To run the three programs together we used proc1&;proc3&;proc3 with other two different combinations of their sequences. The output is the result of executing the exit() at the end of

each program which prints the turnaround time, burst time, and the waiting time. You may refer to the code change part to see the implementation of exit() function inorder to print the times.

As you can see in the following outputs, even though when we run the programs in different sequences, their order of execution is based on their priority, which is controlled by the scheduler.

```
$ proc1&;proc2&;proc3
Turnaround time for PID: 5 is 2954
Burst time time for PID: 5 is 1
Waiting time for PID: 5 is 2953
Turnaround time for PID: 7 is 2955
Burst time time for PID: 7 is 1
Waiting time for PID: 7 is 2954
Turnaround time for PID: 6 is 946
Burst time time for PID: 6 is 482
Waiting time for PID: 6 is 463
zombie!
Turnaround time for PID: 4 is 987
Burst time time for PID: 4 is 492
Waiting time for PID: 4 is 495
$ Turnaround time for PID: 8 is 1431
Burst time time for PID: 8 is 480
Waiting time for PID: 8 is 951
zombie!

Turnaround time for PID: 9 is 3820
Burst time time for PID: 9 is 1
Waiting time for PID: 9 is 3819
$
```

```
$ proc3&;proc2&;proc1
Turnaround time for PID: 11 is 11477
Burst time time for PID: 11 is 1
Waiting time for PID: 11 is 11476
Turnaround time for PID: 13 is 11480
Burst time time for PID: 13 is 2
Waiting time for PID: 13 is 11478
Turnaround time for PID: 10 is 935
Burst time time for PID: 10 is 475
Waiting time for PID: 10 is 460
$ Turnaround time for PID: 12 is 963
Burst time time for PID: 12 is 471
Waiting time for PID: 12 is 492
zombie!
Turnaround time for PID: 14 is 1394
Burst time time for PID: 14 is 472
Waiting time for PID: 14 is 922
zombie!

Turnaround time for PID: 15 is 3087
Burst time time for PID: 15 is 2
Waiting time for PID: 15 is 3085
$
```

```
$ proc2&;proc1&;proc3
Turnaround time for PID: 24 is 4928
Burst time time for PID: 24 is 1
Waiting time for PID: 24 is 4927
Turnaround time for PID: 26 is 4930
Burst time time for PID: 26 is 2
Waiting time for PID: 26 is 4928
Turnaround time for PID: 25 is 956
Burst time time for PID: 25 is 474
Waiting time for PID: 25 is 482
Turnaround time for PID: 23 is 975
Burst time time for PID: 23 is 479
Waiting time for PID: 23 is 496
$ zombie!
Turnaround time for PID: 27 is 1429
Burst time time for PID: 27 is 487
Waiting time for PID: 27 is 942
zombie!

Turnaround time for PID: 28 is 3191
Burst time time for PID: 28 is 1
Waiting time for PID: 28 is 3190
$
```