

## OS Assignment-2

### Group Members:

Vani Pailla : U95251377

Asha Tummuru : U38611026

### Environment:

The system environment used for developing and testing this code is based on the xv6-public operating system, which typically runs on a simulated x86 architecture with 2 CPUs. This setup allows for basic multi-core processing features, enabling the use of different scheduling algorithms.

### For compilation and testing:

- Makefile configuration to compile and link the code, including uniq.c, find.c, and various test files (simple\_scheduler\_test.c, advance\_scheduler\_test.c).
- Testing was conducted with QEMU emulator commands, such as `make qemu-nox SCHEDULER=PRIORITY` and `make qemu-nox SCHEDULER=DEFAULT`, allowing us to switch between the custom priority-based scheduler and the default Round Robin scheduler.

Additional files, ex1.txt and ex2.txt, were used as test inputs for uniq and find commands to verify their functionality. Running `make clean` is advised before each test to ensure a clean build.

### Part 1: Adding uniq and find calls (30 pts)

1.1. The `uniq` command is designed to execute the task of identifying and displaying unique data from a specified input file. This fundamental functionality of Uniq involves identifying and printing non-duplicate data from the provided input source.

We have successfully implemented the flag-based `uniq` command with the following options:

- a) `-w`
- b) `-c`
- c) `-u`

Here are the details for each option:

- `uniq -w [N] <filename>`: The `-w` option compares the first `N` characters on each string.
- `uniq -c <filename>`: The `-c` option prints the count of the duplicate strings.
- `uniq -u`: This option prints the strings whose count is equal to 1.

Code:

```
int operation_w(int f, int cha, char opr)
{
    char current[SIZE];
    char x[20][50];
    int i = 0;
    for (int line_num = 0; getstr(f, current, SIZE) > 0; line_num++)
    {
        if (i == 0)
        {
            strcpy(x[i++], current);
            printf(1, "%s", current);
        }
        else
        {
            int is_unique = (opr == 'i') ? (strcmp(x[i - 1], current) != 0)
                                   : (strcmp_ignore_chars(x[i - 1], current, cha) != 0);

            (is_unique) ? (strcpy(x[i++], current), printf(1, "%s", current)) : 0;
        }

        memset(current, 0, SIZE);
    }

    return 0;
}
```

```
// Main uniq_c function
int uniq_c(int f, char opr)
{
    char x[20][50];
    int lineCount = read_lines(f, x); // Read all lines into x[] array

    if (lineCount >= 0)
    {
        for (int z = 0; z <= lineCount; )
        {
            int count = count_occurrences(x, z, lineCount); // Count identical lines
            print_result(opr, count, x[z]);                // Print the result based on the operation
            z += count;                                     // Move to the next distinct line
        }
    }

    return 0;
}
```

Outputs:

```
$ uniq -c ex1.txt
      3 I understand the Operating system.
      1 Thanks xv6.
      2 I love to work on OS.

$
$
$ uniq -u ex1.txt
Thanks xv6.

$
$ uniq -w 6 ex1.txt
I understand the Operating system.
Thanks xv6.
I love to work on OS.

$
$ cat ex1.txt | uniq
I understand the Operating system.
Thanks xv6.
I love to work on OS.

$
$ uniq ex1.txt
I understand the Operating system.
Thanks xv6.
I love to work on OS.

$
```

1.2. **Find:** In the context of file management, the 'find' command serves the purpose of locating files within a specified directory, providing information about their file type, and distinguishing between files and directories. Each file is assigned a unique inode value upon creation, ensuring its distinct identity throughout its existence. Notably, for this task, several flags are available for utilization.

Please remember the following commands:

- Use the command "printi" to display the inode number and file name.
- When using the command "inum," it will print the file name. If you provide a number with the flag, it will print the file names that have a higher inode value than the argument. Similarly, using "-<number>" will return the file names with a smaller inode value, and using "+<number>" will print the file names that have a higher

inode value than the argument. After reading the argument as the input, pass that argument as a parameter to the method called find.

In order to conduct testing of the code, we have generated the files ex1.txt and ex2.txt. Additionally, we have included the configuration for find.c, uniq.c, ex1.txt, and ex2.txt in the makefile.

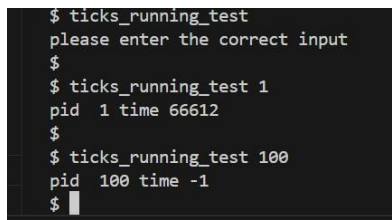
```
$ echo > b
$ mkdir a
$ echo > a/b
$ mkdir a/aa
$ echo > a/aa/b
$ find . -name b -printi
26 ./b
28 ./a/b
30 ./a/aa/b
$
$ find . -name ls
./ls
$
$ find . -name ls -type f
./ls
$
$ find . -name ls -printi
12 ./ls
$
```

```
$
$ find . -inum +3
./README
./cat
./echo
./forktest
./grep
./init
./kill
./ln
./ls
./mkdir
./rm
./sh
./stressfs
./usertests
./wc
./zombie
./uniq
./find
./ticks_running_
./simple_schedul
./advance_schedu
./console
./b
./a/b
./a/aa/b
./a/aa
./a
$
$
$ find . -inum -3
./ex1.txt
$
$ find . -inum 3
./ex2.txt
$
```

## 2. Part 2: Adding ticks\_running() (15 pts)

In the struct proc of the proc.h file, a global variable named start\_time has been instantiated. It is initialized in the allocproc method using the ticks variable. Additionally, a new system call titled ticks\_running has been implemented in proc.c. The ticks\_running method accepts a process ID (pid) as a parameter. This method traverses the process table (ptable) to identify the corresponding pid. Upon locating the matching pid, it calculates the execution time by subtracting the current ticks from the start\_time.

To test we have created a file called ticks\_running\_test.c



```
$ ticks_running_test
please enter the correct input
$
$ ticks_running_test 1
pid 1 time 66612
$
$ ticks_running_test 100
pid 100 time -1
$
```

## 3. Part 3: Implementing a Simple Scheduler (25 pts)

### Default scheduler (rounds robin):

In the xv6-public operating system, we utilize the Round Robin scheduler as the default. This scheduler assigns the next available position in the ready queue to the current process.

### Simple Scheduler:

We have selected the Shortest Job First (SJF) scheduling algorithm for a basic schedule from the two available schedulers - SJF and Round Robin (RR). SJF prioritizes a process with a smaller burst time for execution. For instance, if we have processes p1, p2, and p3 with burst times of 3, 6, and 7 respectively, according to the SJF algorithm, p1 will be executed first, followed by p2 and then p3. However, implementing SJF in real-time is unfeasible as we cannot predict the burst time of a process without its execution. Hence, we must employ a random function, and based on the random value assigned to each process, the process with the lowest value will be executed first.

I have defined a global variable named "sjf\_process\_length" in the "proc.h" file. The variable is initialized in the "allocproc" function. Within the scheduling method, I have implemented a logic to identify the process with the shortest burst time, which will be the first to be executed.

To test with Default Scheduler I have created a file called simple\_scheduler\_test.c.

Code:

```
for (;;)
{
    sti();
    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if (p->state != RUNNABLE)
            continue;
        if (p->pid > 2)
        {
            struct proc *p1;
            struct proc *shortest = p;
            for (p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++)
            {
                if (p1->state != RUNNABLE)
                    continue;
                shortest = (p1->sjf_process_length < shortest->sjf_process_length) ? p1 : shortest;
            }
            p = shortest;
        }
    }
}
```

```

// Switch to chosen process. It is the process's job
// to release ptable.lock and then reacquire it
// before jumping back to us.
c->proc = p;
switchvm(p);
p->state = RUNNING;
swtch(&(c->scheduler), p->context);
switchkm();
// Process is done running for now.
// It should have changed its p->state before coming back.
c->proc = 0;
}
release(&ptable.lock);
}
#else

```

To test this code I have created a file called `simple_scheduler_test`, in this I have created a 3 processes and making that 3 processes to sleep . and allowing the 3 processes to runnable at same time so that we can see those 3 processes will run based on shortest job first.

Output:

```

$ simple_scheduler_test
first process second started at 7087
first process first started at 7088
first process third started at 7088
first process lenght 58
second process lenght 9
third process lenght 84
process second is running
process second execution completed in 51
process first is running
process first execution completed in 52
process third is running
process third execution completed in 53
$

```

#### 4. Part 4: A (More) Advanced Scheduler (30 pts)

**Advance scheduler:**

1. **Priority Scheduling:** We have implemented a Priority scheduling system in our advanced scheduler. This approach selects processes based on their priority, which is determined by a priority variable added to the `proc.h` file. By default, processes created by the `fork` and `exec` methods are assigned a default priority.
2. **Scheduler Logic:** In the Scheduler method, we have developed a logic to select and schedule the process with the highest priority.
3. **set\_priority System Call:** We have introduced a system call called `set_priority`, which allows users to directly assign a priority to a process by providing the process ID and the desired priority as arguments.
4. **get\_priority System Call:** Additionally, we have implemented a system call called `get_priority` to retrieve the priority of a specified process by passing the process ID as an argument.
5. **Comparison with Default Scheduler:** To compare the advanced scheduler with the default scheduler, we have created a file named `advance_scheduler_test.c`. This file contains sample processes created using `fork` to test the functionality with different schedulers.
6. Make sure to run `make clean` before running the `qemu-nox` with flag.

**Note:** I have implemented the login in a way higher the value is the higher the priority. High priority will execute first.

**make qemu-nox SCHEDULER=PRIORITY**

**make qemu-nox SCHEDULER=DEFAULT**

Code:

```
for (;;)
{
    // cprintf("hello--\n");
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if (p->state != RUNNABLE)
            continue;
        struct proc *p1;
        struct proc *highpriority;
        highpriority = p;
        if (p->pid >= 2)
        {
            for (p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++)
            {
                if (p1->state != RUNNABLE)
                    continue;
                if (highpriority->priority < p1->priority || (p->priority == highpriority->priority && p->pid < highpriority->pid))
                {
                    highpriority = p1;
                }
            }
        }
        p = highpriority;
        // Switch to chosen process. It is the process's job
        // to release ptable.lock and then reacquire it
        // before jumping back to us.
        c->proc = p;
        switchvm(p);
        p->state = RUNNING;

        switch(&(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);
}
```

To test this code I have created a file called `advance_scheduler_test`, in this I have created a 3 processes and making that 3 processes to sleep . and allowing the 3 processes to runnable at same time so that we can see those 3 processes will run based on priority.

output:

```
$ advance_scheduler_test
first process A started at 2927
second process B started at 2928
third process C started at 2929
first process priority 3
second process priority 3
third process priority 5
third process C
third process C execution completed in 51
first process A
first process A execution completed in 52
second process B
second process B execution completed in 53
$
```