Operating System
CS 370
Spring 2019

Project 4.1
XV6 Scheduler

Due Date: 11:59 pm,4th March 2019
Must be submitted before deadline

Total Points: 100

Teaching Assistant:
Office Hours:
Graduate Assistant: Pradip Maharjan
Office Hours: MoWe 12 PM - 2 PM
Teaching Assistant: Luis Maya-Aranda
Office Hours: MoWe: 1 PM - 4 PM
TuTh: 10 AM - 12 PM

Course Web Page: http://osserver.cs.unlv.edu/moodle/

Objective: Become familiar with the **xv6** Teaching Operating System, and scheduling algorithms

Introduction:
In xv6, Round Robin(RR) scheduler is implemented by default. We have RR where time slices are assigned to each process in equal portions. The process is interrupted if it is not completed in given time slice and send back to the queue where it gets next time slice after all other processes in the queue gets their allocated time slice. The default time slice in xv6 is 100 ticks.

In this assignment, we will implement a scheduler benchmarking program in which we simulate processes with different CPU and IO bursts and collect data to track creation time, start time, run time, io time, wait time, end time, turnaround time and response time.

Resources:
The following resources provide more in-depth information regarding **xv6**. They include the **xv6** reference book, source code (pdf), and a tutorial on running and debugging.

1. **xv6** Reference Book: https://pdos.csail.mit.edu/6.828/2016/xv6/book-rev9.pdf
2. **xv6** Source Code PDF: https://pdos.csail.mit.edu/6.828/2016/xv6/xv6-rev9.pdf
3. Running and Debugging Tutorial: http://zoo.cs.yale.edu/classes/cs422/2010/lec/l2-hw

Project:
Complete the following activities.

- Review chapter 0, 1, and 3 of the **xv6** reference book.
- Implement the **wait2** system call.
- Implement the **updateIOtime** system call.
- Implement a scheduler benchmarking program.
- Modify the existing Round Robin scheduler to print the ready queue.

Submission
When complete, submit:

- Zip the entire **xv6** folder and submit (not qemu).
- Use "zip" to compress, not "tar" or others.
- A copy of the ***zipped xv6 folder*** via the class web page (assignment submission link) by 11:59 pm 4th March, 2019.

We will check for code similarities using MOSS. Please do not download code from the internet or distribute solutions anywhere including Github, Bitbucket and any other public code publishing platforms.

Submissions received after the due date/time will not be accepted.

1. Background:

The **xv6** operating system by default keeps track of very little information about a process. Often, operating systems keep track of certain information about processes for scheduling or benchmarking purposes. Some of the information that is kept includes the creation time of the process, the time the process was first scheduled, the run time of the process, the time the process ended, and the time spent by the process waiting for I/O. ***These fields should be added to the xv6 process structure located in proc.h***. Recall that **xv6** records time in clock ticks.

Every process in xv6 is stored in a Process Control Block (PCB) called the ptable, or process table. The ptable size is set at boot and never changed. Each entry in the ptable is a different process and unused entries are marked with the process state UNUSED. In xv6, each entry is a proc struct (from proc.h). For security purposes, all functions that can access the ptable are implemented in proc.c. No other file gets access to the ptable. If you need to access the ptable, you must write a function in proc.c. You may create a system call that calls a function in proc.c (similar to how you wrote a system call that calls cmostime() from lapic.c in a previous assignment).

2. Scheduler Benchmarking (benchmark.c):

To retrieve baseline benchmarking information on the already-existing **xv6** Round Robin scheduler, we will write a program named ***benchmark.c*** that forks a given number of child processes that will do some CPU and/or I/O operations. Information about each child process will be read from a file.

Note: **xv6** does not provide a built-in way to read integers from a file. Instead, characters should be read one at a time and stored in a buffer until whitespace is encountered. The contents of the buffer can then be converted into an integer. Recall the approach taken in the Assembly Language and Systems Programming class.

To benchmark the scheduler with simulated processes, we need to remove compiler optimization so that we can look into the simulation results accurately. We will simulate the CPU bursts by incrementing a counter a certain number of times and we will simulate IO bursts by printing a message to the screen a certain number of times. If we don't remove compiler optimization, then compiler will replace CPU burst for loop with optimized code. To remove compiler optimization, remove -O flags from CFLAGS variable in line 79 and 80 in **Makefile.**

Also add benchmark under UPROGS and EXTRA in **Makefile** like in previous assignment.

2.1 Input Specification(input.txt):

The input will be given in the form of a text file named ***input.txt***. The first line in the text file will contain a number, ***n***, that represents the number of processes to fork. The file will then be followed by ***n*** lines of the form where each number is a positive integer:

arrivalTime numCPUBursts CPUBurst IOBurst CPUBurst ...

Processes will be specified in order of their arrival time. The number of CPU bursts will be capped at five. Recall that the number of I/O bursts is the number of CPU bursts – 1. An I/O burst length will always be followed by a CPU burst length. CPU and I/O burst times will be between 1 and 20. A larger integer specifies a longer CPU or I/O operation while a smaller integer specifies a shorter CPU or I/O operation. For simplicity, you may assume that the input file contains no errors.

2.2 Creation Time and End Time:

Creation time is the time when the process entered the system. In **xv6**, processes are created in the ***allocproc()*** function located in ***proc.c***. The end time of a process is the time when the process finished executing. In **xv6**, processes are exited in the ***exit()*** function located in ***proc.c***. To access the timestamp, you can use ***ticks*** variable. (Hint: look how it is used in proc.c)

2.3 Start Time:

The start time of a process is the time when the process is scheduled for the first time. *This should only be set once per process*. Processes are scheduled in **xv6** in the ***scheduler()*** function located in ***proc.c***.

2.4 Run Time:

The run time of a process is the time the process spent in the CPU. In **xv6**, the software timer periodically sends interrupts so that **xv6** knows how much time has passed. Timer interrupts, like all other interrupts are handled in the ***trap.c*** file. Whenever a software timer interrupt occurs, the run time of the current running process should be incremented.

Note: Each interrupt has a associated number with it, which are defined and explained in traps.h file.
When trap function is called, it handles each trap based on the trap number it receives. For example, if tf->trapno = IRQ_TIMER, then the timer has called an interrupt which happens every 100 ticks. IRQ stands for interrupt request.

2.5 I/O Time:

As discussed in class, when a process is on the CPU it is in the RUNNING state. When a process needs I/O it is blocked and placed in the WAITING (here called SLEEPING) state. When the I/O request is finished, the process is placed back in the READY state (here called RUNNABLE). However, xv6 is a simple kernel, and does not bother with this overhead. Instead, if a running process calls for I/O, the user program makes the appropriate system call, and the currently running process gets re-used for the system call. If you look into proc.h, you will see a field called kstack (the kernel stack), a field called tf (the trapframe), and a field called context (the struct containing the registers of the currently running process). A system call performs a context switch by putting the arguments on the kstack, swapping the trapframe to the system call function and swapping the context registers. Then, it runs the system call, sets the kstack and tf back to NULL, and swaps the context back to the original value. However, the state of the process is unchanged; it is always RUNNING. Therefore, to keep track of the time spent doing I/O, we will just time how long it takes to perform printf. We call updateIOtime system call to keep track of io time.

2.5.1 Implementing IOTime System call:

To time how long it takes to perform printf, we need to modify the putc function in **printf.c** to time how long it takes to call write(). To do this, we call and save the uptime() before and after the write() call inside putc. Then, the difference is the time we spent in write(). Now we need to add this value to the current process's ioTime. To do this, we need to create a new system call with the following function prototype:

int IOTime(int time);

Create the system call IOTime() similar to a system call in previous assignments. You need to modify the following files:

1. user.h
2. usys.s
3. syscall.h
4. syscall.c

5. sysproc.c

In sysproc.c you will implement the system call by setting up the argument for *time* using argint, then calling updateIOTime in proc.c. You will create the updateIOTime function in proc.c and place its function prototype in defs.h so it can be called from sysproc.c in the function IOTime. The function prototype for updateIOTime is:

int updateIOTime(int time);

Inside updateIOTime() you will get the currently running process by calling myproc(), and then increment the process's ioTime field by *time*.

2.6 Wait2 System Call:

For the timing information of a process to be accessed from user space, a new system call needs to be implemented. Wait2 should take 5 arguments: the creation time, the start time, the end time, the run time , and the io time of a process. The wait time is the time spent by a process that is RUNNABLE waiting to be scheduled which we calculate in the user program by the equation below.

$$waittime = endtime - creationtime - runtime - I/O\ time$$

The function prototype of wait2 system call is:

*int wait2(int * creationTime, int* startTime, int* runTime, int* ioTime, int* endTime);*

A parent process calls wait2 to wait for a child process to terminate and to retrieve information about that child process. Wait2 should set up the arguments that were passed to it just like we did in the date assignment. Then, wait2 will call getProcessStats to wait on the process to exit and record the time information. The creation time, start time, run time, io time, and end time should be ***returned by reference*** to the parent process.

To implement wait2 system call, you need to modify following files like in previous assignment

1. user.h
2. usys.s
3. syscall.h
4. syscall.c
5. sysproc.c

The wait2 system call will call getProcessStats(*int * creationTime, int* startTime, int* runTime, int* ioTime, int* endTime)* to retrieve these time stamps. We need to implement getProcessStats in proc.c. The function prototype should be defined in defs.h as:

*int getProcessStats(int * creationTime, int* startTime, int* runTime, int* ioTime, int* endTime);*

Note that defs.h is a header file included in sysproc.c so we need to define the function here so that it is visible to a system call in sysproc.c.

The function getProcessStats behaves very similarly to the already-existing wait function in proc.c. Just like wait, getProcessStats should return the *pid* of the child process that terminated. However, getProcessStats will take 5 arguments, int* creationTime, int* startTime, int* runTime, int* ioTime and int* endTime and set these variables from the process's proc fields.

2.7 Parent Process:

The input file for the program will be given as a command line argument. The parent process should read the input file and store the information, perhaps in a 2D array where each row represents a process and entries in the row specify alternating CPU and I/O burst lengths.

Hint: The maximum number of processes **xv6** can handle is 64. Since the number of CPU bursts is capped at five, you can have up to nine columns to store the burst lengths (max CPU bursts (5) + max I/O bursts (4)). So, you may choose store the process information in a 2D array with 64 rows and at least 9 columns. However, you may find it easier to use an extra column and have the first column in each row hold the number of CPU bursts for that process.

The parent process is responsible for forking the child processes and waiting for them to finish. Processes are specified in the order of their arrival time. Before starting to fork children, the parent should get the start time in ticks by calling ***uptime()***. Before forking a child, the parent should check whether its arrival time is greater than the current time (retrieved via another call to ***uptime()***) minus the start time. If the arrival time is not greater, the parent should sleep until the arrival time is greater.

Information about each child's creation time, start time, end time, run time, and waiting time should be kept; hence, the wait2 system call should be used. The parent should also keep track of the turnaround time and response time of each process.

$$TAT_i = EndTime_i - CreationTime_i$$
$$Response_i = StartTime_i - CreationTime_i$$

After each child process exits, parent should print the creation time, start time, end time, run time, io time, waiting time, turnaround time, and response time of each process in the following format:

```
child[pid]: creationTime – [creationTime] startTime – [startTime] endTime – [endTime]
            runTime – [runTime] ioTime – [ioTime] waitTime – [waitTime]
            turnaround time – [tat] response time [response]
```

In addition, the parent should calculate and print the average turnaround time and average response time for all processes.

$$Average_{TAT} = \sum_{i=1}^{n} \quad TAT_i$$

$$Average_{Response} = \sum_{i=1}^{n} \quad Response_i$$

2.7.1 Parent Process Algorithm:

The algorithm for the parent process is as follows:

```
read input file
sTime = uptime()
for i = 0 to numProcs do
      currentTime = uptime()
      if arrivalTime[i] > (currentTime – sTime) then
            sleep until process is ready to arrive
      endif
      pid = fork()
      if pid == 0 then
            // child process algorithm
      endif
endfor
for i = 0 to numProcs do
```

```
      pids[i] = wait2(&creationTime, &startTime, &runTime,&endTime)
      store times for process
      calculate TAT and Response
endfor
for i = 0 to numProcs do
      print process timing information to the screen
endfor
print average TAT and average response to the screen
```

2.8 Child Processes:

Child processes will do some CPU calculations and possibly some I/O operations. The CPU operation will consist of incrementing a counter a given number of times. The I/O operation will consist of the process printing its *pid* to the screen a given number of times.

Recall that information is provided for each process about its CPU and I/O burst lengths. These lengths should be used to determine how many times to increment the counter or to print its *pid* to the screen. Because the range of burst lengths is [1, 20], if this value was simply used as the number of times to increment a counter, the operation would be over too quickly to demonstrate the effect of CPU bursts. As such, the numbers read in from the file for CPU burst lengths should be used as a multiplier for a constant CPU_ITERATIONS variable, which should be set equal to 10000000. The CPU operation thus consists of incrementing a counter cpu_burst_length * CPU_ITERATIONS times.

Each child should perform at least one CPU burst (since each process has at least one) followed by any remaining I/O and CPU bursts, alternating. The child, if it has any I/O bursts, should print its *pid* in the following format:

```
child [pid] prints for the [i] time
```

The child should not print anything during a CPU burst.

2.8.1 Child Process Algorithm:

Use the following algorithm sketch for the child processes. How you get the burst lengths for each process depends on how you set up your array(s) storing those lengths.

```
counter = 0
pid = getpid()
// perform cpu burst (always at least one)
for i = 0 to first_cpu_burst_length * CPU_ITERATIONS do
      counter++
endfor
for each remaining I/O and CPU burst pair do
      // next burst is an I/O burst
      for i = 0 to next_io_burst_length do
            print pid to screen
      endfor
      // next burst is a CPU burst
      for i = 0 to next_cpu_burst_length * CPU_ITERATIONS do
            counter++
endfor
endfor
exit()
```

Sample Input:

```
5
0 2 10 7 6
100 1 17
100 3 1 6 2 5 4
125 1 15
130 1 2
```

Sample Output:

```
$ benchmark input.txt
child 6 prints for the 1 time
child 6 prints for the 2 time
child 6 prints for the 3 time
child 6 prints for the 4 time
child 6 prints for the 5 time
child 6 prints for the 6 time
child 6 prints for the 1 time
child 6 prints for the 2 time
child 6 prints for the 3 time
child 6 prints for the 4 time
child 6 prints for the 5 time
child 4 prints for the 1 time
child 4 prints for the 2 time
child 4 prints for the 3 time
child 4 prints for the 4 time
child 4 prints for the 5 time
child 4 prints for the 6 time
child 4 prints for the 7 time
child 8: ctime - 608 - stime - 622 - etime - 1157
         iotime - 0 - rtime - 107 - wtime - 442
     turnaround time - 549 - response time - 14
child 6: ctime - 579 - stime - 587 - etime - 2424
         iotime - 50 - rtime - 433 - wtime - 1362
     turnaround time - 1845 - response time - 8
child 7: ctime - 600 - stime - 607 - etime - 3524
         iotime - 0 - rtime - 793 - wtime - 2131
     turnaround time - 2924 - response time - 7
child 4: ctime - 475 - stime - 480 - etime - 3600
         iotime - 34 - rtime - 937 - wtime - 2154
     turnaround time - 3125 - response time - 5
child 5: ctime - 575 - stime - 581 - etime - 3670
         iotime - 0 - rtime - 910 - wtime - 2185
     turnaround time - 3095 - response time - 6
Avg TAT: 2307
Avg Response: 8
```

Testing Your Program:

Once you have written your program, you will want to test it to verify that it works. In order to do this, however, you must get your input text file into the **xv6** file system. To do this, follow these steps:

1. In your Ubuntu environment, create the input text file and save it. Make sure the file is in the same directory as all of the **xv6** files.
2. Open up the Makefile for **xv6**
3. Look for ***fs.img*** (around line 177) and add your input test file to that line and the one immediately below it. The result should look like the following:

```
fs.img: mkfs README input.txt $(UPROGS)
  ./mkfs fs.img README input.txt $(UPROGS)
```

In order to understand the scheduler timing results better, you may find it helpful to limit **xv6** to one CPU. This is done with the command:

```
make CPUS=1 qemu
```

Viewing the Round Robin Scheduler

To visualize the Round Robin scheduler in xv6, you will make changes to the **scheduler()** function in proc.c. Familiarize yourself with the existing scheduler algorithm. Write a small amount of code to print the ready queue, or the RUNNABLE queue as it is called in xv6. You should create a for loop that loops from the current process **p** in the scheduler's for loop to the end of the process table, **ptable**. Print the pid of each process in the RUNNABLE state. Then, make another for loop from the beginning of the table to the current process **p** and print the pids of those RUNNABLE processes. Next, repeat this code after the process's state is changed to RUNNING to show an item being queued. Each queue should have the following format:

Head |--p1-p2-…-pK--| Tail. Running process: <pid/none>

To prevent an endless loop of printing, only print the RUNNABLE queue if the number of RUNNABLE processes is **greater than 1**. We can test this using a smaller input, since printing the queue slows down xv6 by a considerable amount. You may also decide to decrease the CPU_ITERATIONS variable for this part to shorten the runtime.

Sample Input:

```
3
0 1 1
100 1 1
100 1 1
```

Sample Output (Truncated):

```
Head |--6-4-5--| Tail. Running Process: None
Head |--4-5--| Tail. Running process: 6
Head |--4-5-6--| Tail. Running Process: None
Head |--5-6--| Tail. Running process: 4
Head |--5-6-4--| Tail. Running Process: None
Head |--6-4--| Tail. Running process: 5
Head |--6-4-5--| Tail. Running Process: None
Head |--4-5--| Tail. Running process: 6
Head |--4-5-6--| Tail. Running Process: None
Head |--5-6--| Tail. Running process: 4
Head |--5-6-4--| Tail. Running Process: None
Head |--6-4--| Tail. Running process: 5
Head |--6-4-5--| Tail. Running Process: None
Head |--4-5--| Tail. Running process: 6
Head |--4-5-6--| Tail. Running Process: None
Head |--5-6--| Tail. Running process: 4
Head |--5-6-4--| Tail. Running Process: None
Head |--6-4--| Tail. Running process: 5
Head |--6-4-5--| Tail. Running Process: None
Head |--4-5--| Tail. Running process: 6
Head |--4-5-6--| Tail. Running Process: None
Head |--5-6--| Tail. Running process: 4
Head |--5-6-4--| Tail. Running Process: None
Head |--6-4--| Tail. Running process: 5
Head |--6-4-5--| Tail. Running Process: None
Head |--4-5--| Tail. Running process: 6
Head |--4-5-6--| Tail. Running Process: None
Head |--5-6--| Tail. Running process: 4
Head |--5-6-4--| Tail. Running Process: None
Head |--6-4--| Tail. Running process: 5
Head |--6-4-5--| Tail. Running Process: None
Head |--4-5--| Tail. Running process: 6
Head |--4-5-6--| Tail. Running Process: None
Head |--5-6--| Tail. Running process: 4
Head |--5-6-4--| Tail. Running Process: None
Head |--6-4--| Tail. Running process: 5
Head |--6-4-5--| Tail. Running Process: None
```

```
Head |--6-5--| Tail. Running Process: None
Head |--5--| Tail. Running process: 6
Head |--5-6--| Tail. Running Process: None
Head |--6--| Tail. Running process: 5
Head |--6-5--| Tail. Running Process: None
Head |--5--| Tail. Running process: 6
Head |--5-6--| Tail. Running Process: None
Head |--6--| Tail. Running process: 5
Head |--6-5--| Tail. Running Process: None
Head |--5--| Tail. Running process: 6
Head |--5-6--| Tail. Running Process: None
Head |--6--| Tail. Running process: 5
Head |--6-5--| Tail. Running Process: None
Head |--5--| Tail. Running process: 6
Head |--5-6--| Tail. Running Process: None
Head |--6--| Tail. Running process: 5
Head |--6-5--| Tail. Running Process: None
Head |--5--| Tail. Running process: 6
Head |--3-5--| Tail. Running Process: None
Head |--5--| Tail. Running process: 3
Head |--5-3--| Tail. Running Process: None
Head |--3--| Tail. Running process: 5
Head |--3-5--| Tail. Running Process: None
Head |--5--| Tail. Running process: 3
Head |--5--| Tail. Running Process: None
Head |---| Tail. Running process: 5
child 4: ctime - 322 - stime - 325 - etime - 1231
         iotime - 0 - rtime - 326 - wtime - 583
     turnaround time - 909 - response time - 3
child 6: ctime - 447 - stime - 470 - etime - 1379
         iotime - 0 - rtime - 324 - wtime - 608
     turnaround time - 932 - response time - 23
child 5: ctime - 385 - stime - 449 - etime - 1386
         iotime - 0 - rtime - 338 - wtime - 663
     turnaround time - 1001 - response time - 64
Avg TAT: 947
Avg Response: 30
```

End Notes:

Benchmark.c

You are responsible to create a new file called **benchmark.c**, and write a program that will:

1. Accepts arguments from the command line interface. You are expected to do the necessary error checking with the corresponding error message.
2. Read input file
3. Parent process algorithm
4. Child process algorithm

List of files in xv6 that you need to work on in this assignment

1. Makefile
2. user.h
3. usys.s
4. syscall.h
5. syscall.c
6. sysproc.c
7. proc.h
8. proc.c
9. trap.c
10. defs.h