Amirkabir University of Technology

Computer engineering and it department
Operating Systems

Project 1

Authors:
Ghazal Sadeghian
M. Mahdi Naseri

Under the Supervision of:
Dr. Nastooh Taheri Javan

November 6, 2019

- Before you start your project, delete the .git folder in the XV6 repository and create a new repository, so the size of your final project will be reduced. Thus, you will be able to upload your project to the moodle. Note that your contribution to the project will be checked, which is available in git commits. In addition, your codes will be compared to each other using MOSS. As a result, we encourage you to do all parts of your project *YOURSELF*.

1 Sum of Even Digits

Write a program to create a parent process and child process. The parent process takes a number as an input and passes it to the child process. The child process should calculate the sum of even digits in the received number and concatenates the received number with *"The sum of even digits in the input number:"* string and calculated sum, then sends it back to the parent process. Finally, the parent process should print the final received string.

- This question should be implemented in C.
- You should not use available string functions for concatenating.
- You should use fork() and pipe() system calls in Linux for your implementation.

Sample Output:

```
1256664876886345212 The sum of even digits in the input number :68
```

2 Adding System Calls

Add the following system calls to xv6 operating system:

2.1 getChildren

This system call gets PID of a process and returns process ids of its children. When the process has more than one child, this system call should return a multi-digit number. And the order does not matter in this multi-digit number. For example, suppose your current process has two children and the first child and the second have PID 6 and 7, respectively. This system call should return 67 or 76 as an output. For adding this system call, first, you should add *getppid* system call. This system call returns PID of the parent process.

Write a user-level program named *getChildrenTest* for testing your implementation. First, create a number of children by using fork(). Then, for each process, you should print PID, PPID, and the output of the getChildren system call with the parent PID as an argument.

2.2 getCount

This system call gets a valid system call number (listed in the file "syscall.h") as an input and returns the number of times the referenced system call was invoked by the calling process.

Write a user-level program named *getCountTest* for testing your implementation, which gets the system call number as an input. For example, *"getCountTest 5"* command in the terminal should give the number of times the read system call has been invoked by the process.

Hint: You will likely need to update struct proc, located in proc.h. To re-initialize your data structure when a process terminates, look into the functions in proc.c.

3 Implementing CPU Scheduling Algorithms

In this part, you are assigned to implement two CPU scheduling algorithms in XV6. In other words, you should replace the present algorithm in XV6 with those which you will write.

To begin with, find out where the current scheduling algorithm of XV6 is located in order to have a better understanding of how this part of XV6 works. Examine the algorithm to locate where the context switches occur and try to match this algorithm with those that have been discussed in the class. Therefore, you will feel more convenient to write your own code. Additionally, consider the aim of this part, which is measuring the efficiency of some scheduling algorithms in the real world.

3.1 Modified XV6 Original Algorithm

The current algorithm in XV6 tries to do the context switch every CPU tick. You should add a parameter to param.h so as to have an arbitrary number of CPU ticks between each time that the algorithm wants to decide which process should be the next. This can be done by adding *#define QUANTUM 10* to param.h file and changing the algorithm's implementation a little bit. Moreover, you should change the number 10 to measure the difference in CPU scheduling, which will be discussed later.

3.2 Modified Priority Scheduling Algorithm

In this section, you should implement a preemptive algorithm that is based on processes' priorities. Therefore, you need to add a variable that is named *priority* to the PCB, which is located in proc.h file. The priority variable values are in [1, 5]; that means 1 for the highest priority and 5 for the lowest priority.

In this part, we do not want to consider just the priority itself to select the desired process. That is, you should add another variable to the proc.h file, which is named *calculatedPriority*. After each quantum, the algorithm updates the current calculatedPriority as calculatedPriority += priority just for the running process; then, the algorithm chooses a process whose calculatedPriority is highest, which means the smallest in size among all processes that are runnable. This can be the previous process or a new one, which requires a context switch.

Consequently, the calculatedPriority will increase gradually for those which have the highest priority since the algorithm wants those new processes to be the ones that will be selected. In addition, please pay attention to the values that the calculatedPriority will get in order to select an appropriate data type for it.

Furthermore, add a system call with this prototype to XV6: *int changePriority(int)*. This accepts a new priority value as int (1 to 5), and the output is 1 if the job has been done successfully, or -1 otherwise. Also, every new process should be assigned the lowest priority (5) and the minimum calculatedPriority among all the processes. That is because the algorithm wants to service new processes as soon as possible. If there is not any process in the table to select the minimum calculatedPriority, set the new process's calculatedPriority to 0.

3.3 Add a System Call to change the algorithm

You are required to add a new system call to XV6 that is responsible for changing the scheduling algorithm when the XV6 is running. The function's prototype is *int changePolicy(int)*. The input of this function could be 0 (XV6 original algorithm, the default one), 1 (Modified XV6 Original Algorithm), and 2 (Modified Priority Scheduling). Also, the output of this function is +1 if the task has been done successfully and -1 otherwise.

Moreover, consider setting initial values for priority and calculatedPriority just for the first time. We want XV6 to remember our previous priorities when it changes from 2 to 1 (or 0), and then to 2 again.

3.4 Add the ability to measure time

In this section, you should modify the *proc* structure in proc.h file. Add these variables in order to keep track of CBT, Turnaround time, and Waiting time:

creationTime, terminationTime, sleepingTime, readyTime, runningTime

When a process is created, assign initial values to these variables. Then update each process's attributes after each CPU clock tick.

Note that you should handle the time variables of processes that are in zombie states. In other words, when a process is in a zombie state, its turnaround time and waiting time should not be affected.

As all this information is only available in kernel space, you should add a system call to XV6 in order to have access to them from user space. This function has this prototype: *int waitForChild(struct* timeVariables)*. The structure *timeVariable*, which is its input, is something like this, which you should implement yourself:

Struct timeVariables{int creationTime; int terminationTime; int sleepingTime; int readyTime; int runningTime;};

Extend the original wait() function in XV6 to write waitForChild() function. Its output is PID of child's process, which has been terminated if the job has been done without any problem; otherwise, it is -1.

3.5 Test the algorithms' performances

3.5.1 Modified XV6 Original Algorithm's performance

Add a program that is named *OriginalSchedTest* to XV6. This program forks 10 times. Each of its children prints the following line 1000 times. The variable [i] is the ith time the process prints.

[PID]: [i]

The parent itself waits for all its children to exit (using waitForChild). Then you can use the values which are derived from this function to print the following parameters:

- Turnaround time, CBT, and Waiting time for all children
- Average Turnaround time, Average CBT, and Average Waiting time

Change the QUANTUM value to find an optimum value for it. Interpret changes of performance due to altering the QUANTUM value, TT, WT, and CBT in order to include them in your **report file**.

3.5.2 Modified Priority Scheduling Algorithm's performance

Add a program that is named *PrioritySchedTest* to XV6. This program forks 25 times. Set the priorities of processes [1-5] to 5, [6-10] to 4, [11-15] to 3, [16-20] to 2, and [25-30] to 1. Each of the parent's children prints the following line 500 times. The variable [child's number] is in [1-25], and the variable [i] is the ith time the process prints.

[child's number]: [i]

Like the previous part, the parent itself waits for all its children to exit (using waitForChild). Then you can use the values which are derived from this function to print the following parameters:

- Turnaround time, CBT, and Waiting time for all children
- Average Turnaround time, Average CBT, and Average Waiting time for each group with the same priority
- Average Turnaround time, Average CBT, and Average Waiting time

Discuss the sequence of prints, TT, WT, and CBT in your **report file**.

- Your report file's name should be *OS-[StNum]-P1-Report.pdf*. There is no need to report how you write this project. The only part which is mandatory to write in your report file is your interpretation of two last parts, which are discussing the performance of algorithms and understanding why the algorithms work like this.