Homework 1

CS 345 Operating Systems

Members: Lauren Dickman

Chapter 4 Questions:

1. **Run process-run.py with the following flags: -l 5:100,5:100. What should the CPU utilization be (e.g., the percent of time the CPU is in use?) Why do you know this? Use the -c and -p flags to see if you were right.**

Based on the book description of tracing, the CPU process for each line is incremented by 1 and that section is labeled as "time." So, I noticed that for each process (Process0 and Process1) there are 5 CPUs (not physical ones), also considered instructions, listed. I assumed that each process is using 50% of the CPU. Though considering that both processes add together it would be 10 CPUs in total. Thus, I thought that 100% of the CPU's time is in use which I was correct. It can be assumed that the CPU utilization will be 100% because there are no I/O processes needing to wait. At each time slot, the CPU is being used; meaning, at no point are both processes not using the CPU.

2. **Now run with these flags: ./process-run.py -l 4:100,1:0. These flags specify one process with 4 instructions (all to use the CPU), and one that simply issues an I/O and waits for it to be done. How long does it take to complete both processes? Use -c and -p to find out if you were right.**

Again, based on the book description of tracing, the CPU process for each line is incremented by 1 and that section is labeled as "time." So, I believe that it takes both processes 6 seconds to complete. I came up with 6 seconds because there were 4 lines for Process0 and 2 lines for Process1.

I was incorrect in the time it takes for the two processes to complete, as the results stated that it takes 11 seconds to complete both processes. The reason I was incorrect was because I did not take into consideration the amount of time waiting for the I/O to complete would take. Considering the extra waiting lines, I can see how the two processes would take 11 seconds, adding the 6 lines in the tracing of the processes.

3. **Switch the order of the processes: -l 1:0,4:100. What happens now? Does switching the order matter? Why? (As always, use -c and -p to see if you were right)**

In this case, Process0 initiates an I/O and becomes blocked waiting for it to complete. At this point Process0 is not using the CPU, but Process1 is. When I/O is completed, Processor0 begins using the CPU. Honestly, Process0 reminds me of the wait () function in which the parent process waits for the child process to be completed. In this case, yes, I believe that the order matters, because there are situations in which the parent process has to wait() for the child process to be completed. In this case, Process1 is the parent and Process0 is the child.

In comparison to the process -l 4:100,1:0, this one utilizes the system resources more efficiently because when waiting for the I/O, the other processes begins using the CPU. Additionally, I noticed when reversing the order, the -l 1:0,4:100 process takes a shorter time to complete than all the other processes, compared to -l 4:100,1:0. Running a process that just completed an I/O is a good idea, because it increases the utilization of system resources, and it takes less time to complete all the processes.

4. **We'll now explore some of the other flags. One important flag is -S, which determines how the system reacts when a process issues an I/O. With the flag set to SWITCH ON END, the system will NOT switch to another process while one is doing I/O, instead waiting until the process is completely finished. What happens when you run the following two processes (-l 1:0,4:100 -c -S SWITCH ON END), one doing I/O and the other doing CPU work?**

Based on the figure shown, when one process initializes the I/O a blocked state is created. In this blocked state, the I/O process no longer uses the CPU. However, the CPU does not begin running the other process. After the I/O process finishes, the other process switches context from ready to running which allows for the other process that does not initialize the I/O to execute instructions.

The CPU is utilized less because while one of the processes is waiting for the I/O to complete, the CPU is not being used. This causes the program to take longer to complete and not effectively use system resources.

5. **Now, run the same processes, but with the switching behavior set to switch to another process whenever one is WAITING for I/O (-l 1:0,4:100 -c -S SWITCH ON IO). What happens now? Use -c and -p to confirm that you are right.**

Based on the figure shown after running the code, the CPU is continually running programs regardless of waiting for the I/O of another program. As was in the previous case, when one process initializes the I/O, a blocked state is created. In this blocked state the CPU is no longer used by the process that initialized the I/O. However, during the block state of the process that uses the I/O, the CPU runs the other process. Overall, this situation demonstrates when a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.

In this situation it's the opposite, the CPU is utilized more because while one of the processes is waiting for the I/O to complete, the CPU goes running the other process. This causes the program to take less time to complete and effectively use system resources.

6. **One other important behavior is what to do when an I/O completes. With -I IO RUN LATER, when an I/O completes, the process that issued it is not necessarily run right away; rather, whatever was running at the time keeps running. What**

happens when you run this combination of processes? (Run ./process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH ON IO -I IO RUN LATER -c -p) Are system resources being effectively utilized?

When a process initiates an I/O it becomes blocked, waiting for it to complete. At that point the OS recognizes that processes that request an I/O are not using the CPU and starts running the other processes that have yet to be ran within the CPU. When processes that initialized the I/O are running, they either switch the context to ready or done depending on the instructions within that specific process. As a result, the CPU had to decide to run other processes that did not have an I/O request during the processes that did issue an I/O in the waiting phase; it improves resource utilization by keeping the CPU busy.

However, I noticed that was not the case throughout all the processes, especially in Process0. There were some cases where the CPU was not busy while the I/O was running in Process0. This results in system resources not being effectively utilized. In fact, according to the results, the CPU is utilized only by 60% in this behavior.

7. **Now run the same processes, but with -I IO RUN IMMEDIATE set, which immediately runs the process that issued the I/O. How does this behavior differ? Why might running a process that just completed an I/O again be a good idea?**

In these processes, the CPU remains busy constantly, even when processes were blocked by the I/O. When a process initiates an I/O it becomes blocked waiting for it to complete. At that point the OS recognizes that processes are is not using the CPU and starts running other processes. While processes that initialized the I/O are running, they either switch the context to ready or done depending on the instructions within that specific process. As a result, the CPU had to decide to run other processes while the process that issued an I/O is in the waiting phase; it improves resource utilization by keeping the CPU busy, by 100% according to the results.

Another aspect of the two behaviors is the time difference between the two. The terminal command that did not use I/O immediate took ten seconds longer, compared to the one that did. Running a process that just completed an I/O is a good idea, because it increases the utilization of system resources, and it takes less time to complete all the processes.

8. **Now run with some randomly generated processes: -s 1 -l 3:50,3:50 or -s 2 -l 3:50,3:50 or -s 3 -l 3:50,3:50. See if you can predict how the trace will turn out. What happens when you use the flag -I IO RUN IMMEDIATE vs. -I IO RUN LATER? What happens when you use -S SWITCH ON IO vs. -S SWITCH ON END?**

-s 1 -l 3:50,3:50

After running the process -s 1 -l 3:50,3:50 in the terminal, it appears that only Process0 began two I/Os and the other only used the CPU (Process1). I noticed that Process0 had a CPU instruction at the beginning. Based on that I can assume that Process0 is being run on the CPU

first, and when Process0 is waiting for the I/Os to be complete, the CPU focuses on running Process1. It's strange that process0 still did three instructions, but was split into one CPU and two I/O's.

The CPU does run Process1 when Process0 is waiting for the I/O to finish. However, Process1 completes faster than Process0 and results in the programs not utilizing the CPU as much due to Process0 waiting for the two I/Os to complete.

RUN IMMEDIATE vs. -I IO RUN LATER

When I ran the process in the terminal both behaviors resulted in the same answer and trace display. In both the run immediate and the run later, the CPU is constantly running. When one process initializes the I/O a blocked state is created. In this blocked state the CPU is no longer used by that process. During this, the CPU runs the other process.

S SWITCH ON IO vs. -S SWITCH ON END

The switch on to the I/O is faster and effectively uses the system resources, while the switch on end is slower and less efficiently uses the system resources. The reason that the switch on performs better than the switch end is because the CPU is continually running programs regardless of the I/O process. When one process initializes the I/O a blocked state is created. In this blocked state the CPU is no longer used by that process. In the switch on end, however, during the block state of the I/O process, the CPU does not run the other one.

_____ _____ _____ _____ _____

-s 2 -l 3:50,3:50

After running the process -s 2 -l 3:50,3:50 in the terminal, it appears that both processes (Process0 and Process1) use two I/Os and one CPU. However, there is a difference between the two processes; Process1 has a CPU instruction at the beginning while Process0 has the CPU instruction at the end. Based on that I can assume that Process1 is being run first on the CPU and while the CPU is not busy when Process1 is waiting for the I/Os to be complete, the CPU will focus attention on Process0.

RUN IMMEDIATE vs. -I IO RUN LATER

When running the process in the terminal both behaviors resulted in the same answer and trace display. In both the run immediate and the run later, the CPU is constantly running. When one process initializes the I/O a blocked state is created, which a process has performed some kind of operation that makes it not ready to run until some other event takes place. In this blocked state the CPU is no longer used by that process that initialized the I/O. During the block state of the process that uses the I/O, the CPU runs the other process.

S SWITCH ON IO vs. -S SWITCH ON END

The switch on to the I/O is faster and effectively uses the system resources. While the switch on end is slower and less efficiently uses the system resources. The reason that the switch on performs better than the switch end is because the CPU is continually running programs regardless of one process waiting for the one I/O to finish. When one process initializes the I/O a

blocked state is created, which a process has performed some kind of operation that makes it not ready to run until some other event takes place. In this blocked state the CPU is no longer used by that process that initialized the I/O. In the switch on end, however, during the block state of the process that uses the I/O, the CPU does not run the other process.

-s 3 -l 3:50,3:50

After running the process -s 3 -l 3:50,3:50 in the terminal, it appears that both processes (Process0 and Process1) use I/O. Process0 does one I/0 and Process1 does two I/O, however, I noticed that the CPU is called at the beginning and end of Process0, with this in mind I assume that the Process0 task is only waiting for the I/O to be completed. Though I assume that Process1 is not running until Process0 is done waiting for I/O to be completed.

It is true that the Process0 task is only to wait for the I/O to be completed and then it is finished using the CPU. However, even while Process0 is waiting for the I/O to be completed the other process (Process1) is running as well. In fact, Process1 is the process that takes the longest alone. I also noticed that as a result of one of the processes not using the CPU because it is finished with its task and the other process is not using the CPU because it's waiting for the I/O to finished only 50% of the CPU is utilized.

RUN IMMEDIATE vs. -I IO RUN LATER

When running the process in the terminal both behaviors resulted in the same answer and trace display. In both the run immediate and the run later, the CPU is constantly running. When one process initializes the I/O a blocked state is created, in which a process has performed some kind of operation that makes it not ready to run until some other event takes place. In this blocked state the CPU is no longer used by that process that initialized the I/O. During the block state of the process that uses the I/O, the CPU runs the other process.

S SWITCH ON IO vs. -S SWITCH ON END

The switch on to the I/O is faster and effectively uses the system resources. While the switch on end is slower and less efficiently uses the system resources. The reason that the switch on performs better than the switch end is because the CPU is continually running programs regardless of one process waiting for the one I/O to finish. When one process initializes the I/O a blocked state is created, which a process has performed some kind of operation that makes it not ready to run until some other event takes place. In this blocked state the CPU is no longer used by that process that initialized the I/O. In the switch on end, however, during the block state of the process that uses the I/O, the CPU does not run the other process.

Chapter 5 Questions:

1. **Write a program that calls fork(). Before calling fork(), have the main process access a variable (e.g., x) and set its value to something (e.g., 100). What value is the variable in the child process? What happens to the variable when both the child and parent change the value of x?**

The fork () creates a copy of the parent process, which is considered the child. However, the child and parent process have their own private address space exclusive of each other. In both processes, the child and the parent can't interfere in each other's address space, also known as their memory. Thus, the fork() creates two different outputs that in this case both the parent and child each maintain their copy of variables.

2. **Write a program that opens a file (with the open() system call) and then calls fork() to create a new process. Can both the child and parent access the file descriptor returned by open()? What happens when they are writing to the file concurrently, i.e., at the same time?**

Based on the code I've written both the parent and child write at the same time within the file then both writes are written within the file on the same line. Though I did try to read the two written nothing is written within the file. In the sense of trying to read the file, nothing is shown for both the parent and child in the terminal. However, in using the wait() call system within the parent process it results in the child being written first within the file and then followed by the parent.

3. **Write another program using fork(). The child process should print "hello"; the parent process should print "goodbye". You should try to ensure that the child process always prints first; can you do this without calling wait() in the parent?**

Instead of using wait(), to get the child to print before the parent is through using the open (). Where you create a file and have the parent and child be accessed at different times, in order to have child print first and parent process print second.

4. **Write a program that calls fork() and then calls some form of exec() to run the program /bin/ls. See if you can try all of the variants of exec(), including (on Linux) execl(), execle(), execlp(), execv(), execvp(), and execvpe(). Why do you think there are so many variants of the same basic call?**

In executing all these variants of exec(), they are produced the same outcome printing off all the file names within the hw1 file. The reason for the many variants of exec() was because the variants were added over time as new functionality was needed. Since you can't change the functionality of the old function names without breaking older programs that bind to the same library, so the different variants of the function are created with different parameters to handle different ways to execute. Overall, it depends on the users, and how they want to execute the files is what determines which variant to use.

5. **Now write a program that uses wait() to wait for the child process to finish in the parent. What does wait() return? What happens if you use wait() in the child?**

If the wait() was a success it returns the process ID of the terminated child, however, error-1 is returned. This occurs when the parent is waiting for the child. If I use wait() in the child process, then wait() returns –1. The reason –1 is returned is that at that point the child has already been terminated, so there is no wait for any process (child process) to exit.

6. **Write a slight modification of the previous program, this time using waitpid() instead of wait(). When would waitpid() be useful?**

The waitpid() is used when we want to wait for a specific child process rather than waiting for all child processes to exit. In this case of using waitpid, the parent is waiting for the specific child because the waitpid() returns the PID of the child process. Overall, the waitpid() allows us to specify more behaviors involving forks that have more than one child to deal with. In this case, waitpid() was not necessary because the fork() only had one child to deal with, if the fork included another child then waitpid() would be more useful. Since the wait() does not care which child should do complete its task first, the waitpid() would be more useful in situations when the parent is wanting a task done first by a specific child out of four children.

7. **Write a program that creates a child process, and then the child closes standard output (STDOUT FILENO). What happens if the child calls printf() to print some output after closing the descriptor?**

If the child prints some output after closing the descriptor, the terminal will not show what child has printed. While the parent process continues to print some output, if one is included, with or without using the wait() system call in the parent process. However, no error will occur when we close STDOUT file descriptor.

8. **Write a program that creates two children and connects the standard output of one to the standard input of the other, using the pipe() system call.**

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <fcntl.h>

#include <sys/wait.h>


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

   int mypipefd[2]; // ariel integer
```

```
int ret;
ret = pipe(mypipefd);
char buffer[20];
if (ret < 0 ){
    fprintf(stderr, "pipe failed\n");
    exit(1);
}
int rc = fork();
if (rc < 0){
    fprintf(stderr, "fork failed\n");
    exit(1);
} else if (rc == 0) {
    printf("Child-1 Process\n");
    write(mypipefd[1], "Hello there!", 12);
} else {
    printf("Parent-1 Process\n");
    int rc2 = fork();
    if (rc2 < 0){
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc2 == 0) {
        printf("Child-2 Process\n");
        read(mypipefd[0], buffer, 15);
        printf("To Child-2, from Child-1 buffer: %s\n", buffer);
    } else {
        wait(NULL);
        printf("Parent-2 Process\n");
    }
}
return 0;
```

}