Homework 2
CS 345 Operating Systems
Members: Christy Keinsley, Courtney Arrowood, and Lauren Dickman

Chapter 7 Simulation Questions:

Equations:

$$T_{response} = T_{firstrun} - T_{arrival}$$

$$T_{turnaround} = T_{completion} - T_{arrival}$$

1. **Compute the response time and turnaround time when running three jobs of length 200 with the SJF and FIFO schedulers.**

FIFO

Job 0: Time: 0	Turnaround: 200	Wait: 0
Job 1: Time: 200	Turnaround: 400	Wait: 200
Job 2: Time: 400	Turnaround: 600	Wait: 400

Average: Turnaround: (200+400+600) ÷ 3 = 400
Response Time: (0+200+400) ÷ 3 = 200

SJF

Job 0: Time: 0	Turnaround: 200	Wait: 0
Job 1: Time: 200	Turnaround: 400	Wait: 200
Job 2: Time: 400	Turnaround: 600	Wait: 400

Average: Turnaround:(200 + 400+600) ÷ 3 = 400
Response Time: (0+200+400) ÷ 3 = 200

2. **Now do the same but with jobs of different lengths: 100, 200, and 300.**

FIFO

Job 0: Time: 0	Turnaround: 100	Wait: 0
Job 1: Time: 100	Turnaround: 300	Wait: 100
Job 2: Time: 300	Turnaround: 600	Wait: 300

Average: Turnaround: (100+300+600) ÷ 3 = 333.3
Response Time: (0+100+300) ÷ 3 = 133.3

SJF

Job 0: Time: 0	Turnaround: 100	Wait: 0
Job 1: Time: 100	Turnaround: 300	Wait: 100

Job 2: Time: 300 Turnaround: 600 Wait: 300

Average: Turnaround:(100 + 300 + 600) ÷ 3 = 333.3

Response Time: (0 + 100 + 300) ÷ 3 = 133.3

3. **Now do the same, but also with the RR scheduler and a time-slice of 1 sec.**

RR

Job 0: Time: 0.00 Turnaround: 298 Wait: 198

Job 1: Time: 1.00 Turnaround: 499 Wait: 299

Job 2: Time: 2.00 Turnaround: 600 Wait: 300

Average: Turnaround:(298 + 499 +600) ÷ 3 = 465.67

Response Time: (0 + 1.00 + 2.00) ÷ 3 = 1.00

4. **For what types of workloads does SJF deliver the same turnaround times as FIFO?**

The work loads would have to be in ascending order. In the previous question, 1 and 2 questions, the SJF and the FIFO had the same turnaround times because the order is already set in ascending order. If the workload for the questions was set as for job 0 200, 100 for job 1, and 300 for job 2. Then turnaround times would be different because the SJF would do job 1 first while FIFO would do job 0 first. Job 0 has a larger work load, and would take longer to complete compared to the 100 workload for job 1 in the SJF delivery.

5. **For what types of workloads and quantum lengths does SJF deliver the same response times as RR?** Quantum length = length of time to run.

The workloads for SJF to produce the same response times as RR can be between the middle or highest runtime, but if the shortest is used the response time differs from the others.If above the highest runtime is used, only the highest possible runtime answer is given.

As a result of having the time slice of RR (-q) set as 100, the average of the response time and turnaround time was less than compared to the SJF with both deliveries having job 0 being 100 in length, job 1 being length of 200, and job 2 being length of 300. While setting the time slice of RR being greater to the longest job length that was set, it resulted in the RR and the SJF having the same averages in response and turnaround time.The results are the same, however, the RR delivery does not complete its function properly, where RR's time slicing is not being ran evenly among each process.

Evidence: When changing -q 1 to -q 200, in RR, we find the same results if we were to do 200,200,200 for SJF. Another example is when we change the workload for both RR & SJF to 100,200,300, and -q to 200 for the RR, we get an equivalent turnaround time of 333.33, and response time/wait of 133.33 for both the SJF and RR delivery.

6. **What happens to response time with SJF as job lengths increase? Can you use the simulator to demonstrate the trend?**

Findings: As the job lengths increase for SJF the response time grows based on inputs. As demonstrated below, if we increment by 200, multiplying by 2 and 3 respectively after the first SJF, the response time grows in multiples of 2 and 3. Overall, the response time increases as a result of increasing the job lengths.

SJF

Job 0: Time: 100	Turnaround: 100	Wait: 0
Job 1: Time: 200	Turnaround: 300	Wait: 100
Job 2: Time: 300	Turnaround: 600	Wait: 300

Average: Turnaround:(100 + 300 + 600) ÷ 3 = 333.33
Response Time: (0 + 100 + 300) ÷ 3 = 133.33

SJF

Job 0: Time: 200	Turnaround: 200	Wait: 0
Job 1: Time: 400	Turnaround: 400	Wait: 200
Job 2: Time: 600	Turnaround: 600	Wait: 600

Average: Turnaround:(200 + 400 + 600) ÷ 3 = 666.67
Response Time: (0 + 200 + 600) ÷ 3 = 266.67

SJF

Job 0: Time: 600	Turnaround: 600	Wait: 0
Job 1: Time: 1200	Turnaround: 1800	Wait: 600
Job 2: Time: 1800	Turnaround: 3600	Wait: 1800

Average: Turnaround:(600 + 1800 + 3600) ÷ 3 = 2000
Response Time: (0 + 600 + 1800) ÷ 3 = 800

7. **What happens to response time with RR as quantum lengths increase? Can you write an equation that gives the worst-case response time, given N jobs?**

Findings: As the quantum lengths increase for RR the response time grows based on inputs. As demonstrated below, if we begin with a quantum length of 1 and then increment by 5, the response time grows to equal to the quantum time of 1, 5 and 10. Overall, the response time increases as a result of increasing the quantum lengths.

Worst-Case Senior: Q(n-1).

RR -q1 -l 1000, 100, 10
Job 0: Time: 0.00 Turnaround: 1110 Wait: 110
Job 1: Time: 1.00 Turnaround: 210 Wait: 110
Job 2: Time: 2.00 Turnaround: 30 Wait: 20

Average: Turnaround:(1110 + 210 + 30) ÷ 3 = 450.00
Response Time: (0 + 1.00 + 2.00) ÷ 3 = 1.00

RR -q5 -l 1000, 100, 10
Job 0: Time: 0.00 Turnaround: 1110 Wait: 110
Job 1: Time: 5.00 Turnaround: 210 Wait: 110
Job 2: Time: 10.00 Turnaround: 30 Wait: 20

Average: Turnaround:(1110 + 210 + 30) ÷ 3 = 450.00
Response Time: (0 + 5.00 + 10.00) ÷ 3 = 5.00

RR -q10 -l 1000, 100, 10
Job 0: Time: 0.00 Turnaround: 1110 Wait: 110
Job 1: Time: 10.00 Turnaround: 210 Wait: 110
Job 2: Time: 20.00 Turnaround: 30 Wait: 20

Average: Turnaround:(1110 + 210 + 30) ÷ 3 = 450.00
Response Time: (0 + 5.00 + 10.00) ÷ 3 = 10.00

Chapter 13 Programing Questions:

2.) Now, run free, perhaps using some of the arguments that might be useful (e.g., -m, to display memory totals in megabytes). How much memory is in your system? How much is free? Do these numbers match your intuition?

Running free in terminal results in:

	Total	Used	Free	Shared	buff/cache	Available
Memory	15793152	3710060	9989792	207024	2093300	11570952
Swap	2097148	0	2097148			

We certainly were not expecting this difference in number, especially for the swap memory section, though not exactly not surprising considering how much is done on a computer. Memory that is available on the Lynn lab computers is 11570952. The amount of free memory is 9989792 and the amount of free swap memory is 2097148.

4. Now, while running your memory-user program, also (in a different terminal window, but on the same machine) run the free tool. How do the memory usage totals change when your program is running? How about when you kill the memory-user program? Do the numbers match your expectations? Try this for different amounts of memory usage. What happens when you use really large amounts of memory?

Memory-user program with free() within function at argument 2

	Total	Used	Free	Shared	buff/cache	Available
Memory	15793152	**3854244**	**9798960**	**215972**	**2139948**	**11416880**
Swap	2097148	0	2097148			

Memory-user program **without free** within function at argument 2

	Total	Used	Free	Shared	buff/cache	Available
Memory	15793152	**3864968**	**9788252**	**215972**	**2139932**	**11406156**
Swap	2097148	0	2097148			

Memory-user program with free() within function at argument 3

	Total	Used	Free	Shared	buff/cache	Available
Memory	15793152	**3904264**	**9750004**	**214420**	**2138884**	**11368364**
Swap	2097148	0	2097148			

Memory-user program **without free** within function at argument 3

	Total	Used	Free	Shared	buff/cache	Available
Memory	15793152	**3942776**	**9709972**	**216000**	**2140404**	**11328272**
Swap	2097148	0	2097148			

Overall, in using the program with or without the free method within the program the memory total and everything within Swap is the same throughout. There was not a very noticeable difference between the used, free, shared, buff, cache, and available sections. Though we did notice that there was a higher amount of usage in the program without the free function, compared to the lower numbers in usage . We were not surprised in our findings that the program having the free function within the program has a higher amount of free memory. In regards to killing the program the numbers are higher in terms of memory usage and free memory, which makes sense in killing the program it's making space within the operating system.

In using *kill* in the terminal to kill the program, it did terminate the program. Especially when using the getppid function, it literally closed the terminal itself. Overall, the kill did what we all expected it to do which was to terminate the program. Though in getting the PID using the getpid of the program, the program was not found by the kill because it already ran and finished.

7. Now run pmap on some of these processes, using various flags (like -X) to reveal many details about the process. What do you see? How many different entities make up a modern address space, as opposed to our simple conception of code/stack/heap?

Using -X and using it on the browser, Firefox, gives a full display of the extended format, including size, address, and etc. It has a lot of different stuff. X vs x, X is the extended version of x and gives you more detail about the extended format. When we used -p it showed the full pathway. There are so many different types of entities that make up modern address space, including Address, Perm, Offset, Device, Inode, Size, etc as opposed to the simple conception of the three tiers of code, stack, and heap.

8. Finally, let's run pmap on your memory-user program, with different amounts of used memory. What do you see here? Does the output from pmap match your expectations?

In comparison to the pmap of the firefox program, it was not at all surprising that the memory-user program was significantly shorter in the format (-X) than firefox. In this case, it is easier to see where the beginning of the "heap" and "stack" is made based on the mapping column of the format.

There is no difference in the formatting or mapping (such as using -X or -x) when the memory-user program is or is not using the free function within the program. We were surprised in how there is no difference in the program's total usage of memory when the free function was used compared with the program not using the free function. Though considering the program is small in the sense that memory leaks would not be a huge concern so it would make sense in that regard in the programs not having different results with and without using the free function.

Chapter 14 Programing Questions:

1. First, write a simple program called null.c that creates a pointer to an integer, sets it to NULL, and then tries to dereference it. Compile this into an executable called null. What happens when you run this program?

After running the program, it returned a *Segmentation Fault* in the terminal. This makes sense, because we did not free the allocated memory.

2. Next, compile this program with symbol information included (with the -g flag). Doing so let's put more information into the executable, enabling the debugger to access more useful information about variable names and the like. Run the program under the debugger by typing gdb null and then, once gdb is running, typing run. What does gdb show you?

When running gdb with our program (hw2-14-1.c aka null.c) it shows our program encountered a *Segmentation Fault* and lists the Memory Address in main(), where the problem occurred.

3. Finally, use the valgrind tool on this program. We'll use the memcheck tool that is a part of valgrind to analyze what happens. Run this by typing in the following: valgrind --leak-check=yes null. What happens when you run this? Can you interpret the output from the tool?

When running valgrind with our program (hw2-14-1.c aka null.c), it shows our program encountered a *Segmentation Fault* and provides a summary of the errors that are within the program. Specifically, valgrind states that there is an issue within their memory check, which is an error in memory. It also provides a solution to the errors within the program by stating that within this address it is not stack'd, malloc'd, or free'd. In our program, we would need to free our program to resolve this memory error.

4. Write a simple program that allocates memory using malloc() but forgets to free it before exiting. What happens when this program runs? Can you use gdb to find any problems with it? How about valgrind (again with the --leak-check=yes flag)?

In the previous program (hw2-14-1.c aka null.c), we used the calloc() instead of using malloc(). In both programs we did not free the programs which resulted in memory leaks. Even with programs using different allocating memory functions, in using gdb and valgrind the results are still the same which is that both programs have a *Segmentation Fault*. It again also provides solutions to errors, and we would need to free our program to resolve this memory error. The only noticeable difference between the two is that their address of main is different, when debugging using gdb, which is to be expected.

5. Write a program that creates an array of integers called data of size 100 using malloc; then, set data[100] to zero. What happens when you run this program? What happens when you run this program using valgrind? Is the program correct?

When we run this program, the output is 0, because we added a printf statement which prints the data in the terminal, in which we set the array data[100] to equal 0. The program ran with no error. Even using the gdb debugger tool gave no error statements within the terminal and allowed us to quit/exit the program normally.

When running the program using Valgrind (--leak-check=yes flag), however, the tool provided many statements of the program using "conditional jumps" which is giving our program 7 errors in total. Valgrind also gives us the option to use --track-origins=yes to tell us where (the address space) the uninitialised values came from, which is what the conditional jump depended on.

6. Create a program that allocates an array of integers (as above), frees them, and then tries to print the value of one of the elements of the array. Does the program run? What happens when you use valgrind on it?

In running the program it results in printing out zero regardless of what index we request from the array, simply putting in *data still results in 0 as the output. When running valgrind (--leak-check=yes flag) it stated that no leaks are possible because all heap blocks were freed.

Then we tried to have values within the allocated array with integers 0 to 99 (size of 100), which when we tried to get a specific index of 7 the program stated that it aborted the malloc(). So, to check to see if the malloc ran we added the printf lines into the program. We added one printf after initializing mallco() and a printf after the for loop of adding values in the array. When we added the print lines the program ran without aborting itself and there were no errors stated when using valgrind (--leak-check=yes flag) to debug.

7. Now pass a funny value to free (e.g., a pointer in the middle of the array you allocated above). What happens? Do you need tools to find this type of problem?

In our program we did free(&data[50]) as a way to pass a funny value to free. At first the program compiled, however, after we ran the program it resulted in an error of free being an invalid pointer. Then we used valgrind (--leak-check=yes flag) and it states that we were doing an invaild free within our program. Yes, we do need tools to find this kind of problem based on how both the terminal and valgrind state that the pointer within free is invalid.

8. Try out some of the other interfaces to memory allocation. For example, create a simple vector-like data structure and related routines that use realloc() to manage the vector. Use an array to store the vector's elements; when a user adds an entry to the vector, use realloc() to allocate more space for it. How well does such a vector perform? How does it compare to a linked list? Use valgrind to help you find bugs.

A vector performs well and is changeable and better than static. Vector is another name for a dynamic array, and with a dynamic array can perform an insertion and deletion from any position easily. Rather, with a linked list you must do extra steps, like finding the node for the specific item you want deleted or inserted by, which could be a whole process in of itself.