Operating Systems Lab

CPE 435-01

Lab 07: Scheduling

By: David Thornton

Lab Date: 22 February 2021

Lab Due: 1 March 2021

Demonstration Due: 1 March 2021

Introduction

The purpose of this lab is to give students an introduction to process scheduling algorithms.

Theory

Topic 1: Round Robin Scheduling

- Each process takes an equal amount of time to schedule.

Topic 2: Priority Based Scheduling

- Processes are executed in order of priority.

Topic 3: Preemptive vs Nonpreemptive Scheduling

- Preemptive scheduling means that if a process arrives with a higher priority than the current process, the CPU stops the current process and switches to the new one. Nonpreemptive scheduling means that if a new process arrives with a higher priority than the current process, the new process is put at the start of the queue and will be executed after the current process finishes.

Observations

All code performs mostly as expected. I did not do extensive testing. I also did not have time to implement a more robust sorting for processes. It appears that my wait time is not being calculated correctly.

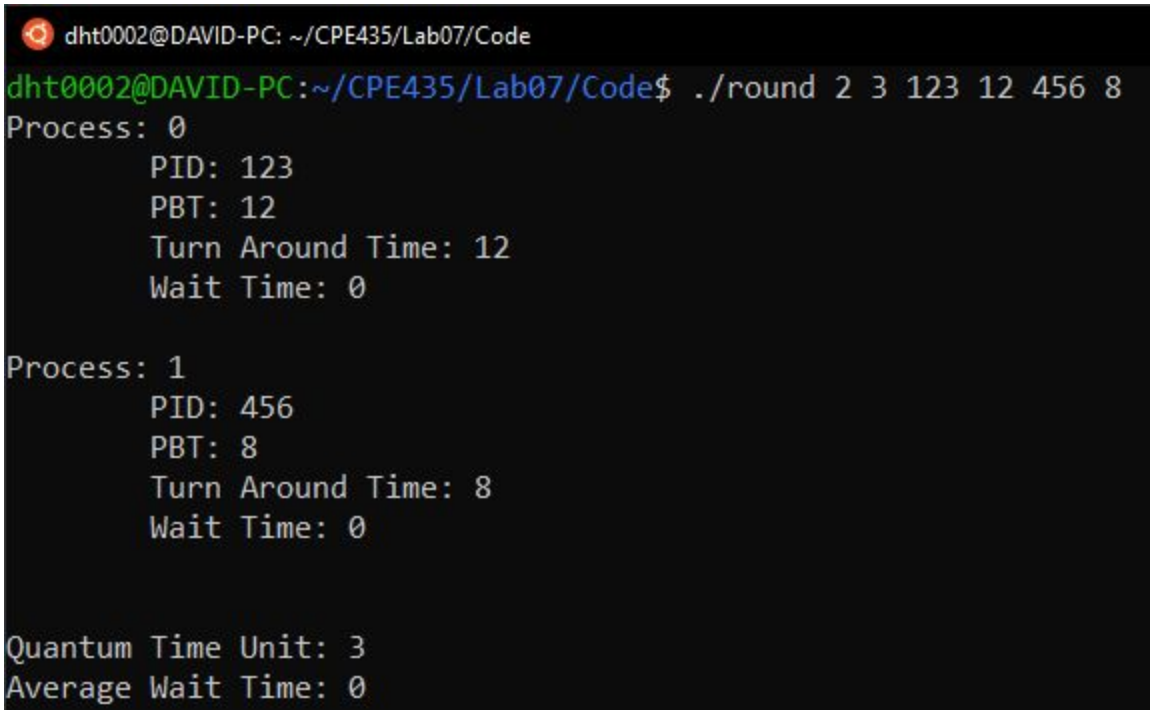

Figure 1: Normal execution of rrs.c

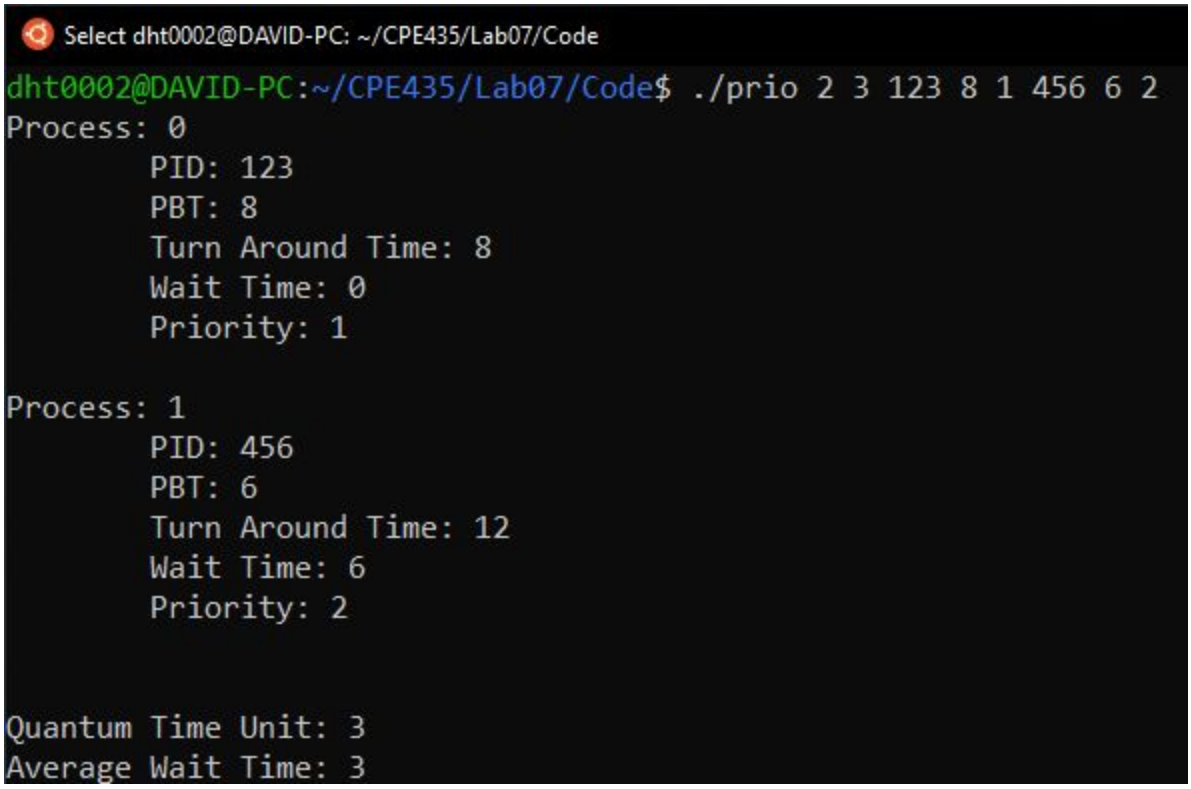

Figure 2: Normal execution of pbs.c

Conclusion

This lab was successful in expanding my knowledge of scheduling types. Demo link

Appendix

Appendix 1: rrs.c

```
// ****************************************
// Program Title: Lab 07
// Project File: rrs.c
// Name: David Thornton
// Course Section: CPE-435, SP 2021
// Due Date: 03/01/2021
// ****************************************
#include <stdbool.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

struct process
{
	int pid;
	int burst_time;
	int t_round; // turn around time
	int wait_time; // Wait Time = Turn around Time - Burst Time
};

int main(int argc, char* argv[])
{
	//Assuming ./rrs n qtu pid0 pbt0 pid1 pbt1 ...

	int n = atoi(argv[1]); // number of processes
	int qtu = atoi(argv[2]); // quantum time unit
	int pid_and_pbt[n]; // array for PID and PBT values
	int avgwait = 0;
	int count = 0;
	struct process *processes;

	for (int i = 3; i < argc; i++)
	{
		if(i % 2 == 1) // if argv 3, 5, 7, ...
			{
```

```
                        pid_and_pbt[i-3] = atoi(argv[i]);
                }
        else if(i % 2 == 0) // if argv 4, 6, 8, ...
                {
                        pid_and_pbt[i-3] = atoi(argv[i]);
                }
}

// Allocating memory for n number of processes
processes = malloc(sizeof(struct process) * n);

if (processes == NULL)
{
        printf("******** ERROR ********\n");
        printf("Memory allocation error\n");
        printf("Exiting now...\n");
        exit(1);
}

// Move values from pid_and_pbt[] to processes[]
for (int i = 0; i < n; i++)
{
        processes[i].pid = pid_and_pbt[i+count];
        processes[i].burst_time = pid_and_pbt[i+1+count];
        count++;
}

// Finding wait times
while (1)
{
        bool done = false;
        for (int i = 0; i < n; i++)
        {
                int t = 0;
                if (pid_and_pbt[2*i] > 0)
                {
                        done = true;
                        if(pid_and_pbt[2*i] > qtu)
                        {
                                t += qtu;
                                pid_and_pbt[2*i] -= qtu;
                        }
                        else
                        {
                                t += pid_and_pbt[i];
                                processes[i].wait_time = t - processes[i].burst_time;
                                pid_and_pbt[2*i] = 0;
```

```
                                        }
                                }
                        }
                        if (done == true)
                        {
                                break;
                        }
                }

                for (int i = 0; i < n; i++)
                {
                        avgwait += processes[i].wait_time;
                }
                avgwait = avgwait / n;

                // Finding turn around times
                for (int i = 0; i < n; i++)
                {
                        processes[i].t_round = processes[i].burst_time + processes[i].wait_time;
                }

                // Print results
                for (int i = 0; i < n; i++)
                {
                        printf("Process: %d\n", i);
                        printf("\tPID: %d\n", processes[i].pid);
                        printf("\tPBT: %d\n", processes[i].burst_time);
                        printf("\tTurn Around Time: %d\n", processes[i].t_round);
                        printf("\tWait Time: %d\n\n", processes[i].wait_time);
                }
                printf("\nQuantum Time Unit: %d\n", qtu);
                printf("Average Wait Time: %d\n", avgwait);
                void free(void*);
}
```

Appendix 2: pbs.c

```
// ***************************************
// Program Title: Lab 07
// Project File: pbs.c
// Name: David Thornton
// Course Section: CPE-435, SP 2021
// Due Date: 03/01/2021
// ***************************************
#include <stdbool.h>
#include <stdio.h>
```

```
#include <unistd.h>
#include <stdlib.h>

struct process
{
	int pid;
	int priority;
	int burst_time;
	int working_time;
	int t_round; // turn around time
	int wait_time; // Wait Time = Turn around Time - Burst Time
};

int main(int argc, char* argv[])
{
	//Assuming ./prio n qtu pid0 pbt0 priority0 pid1 pbt1 priority1 ...

	int n = atoi(argv[1]); // number of processes
	int qtu = atoi(argv[2]); // quantum time unit
	int pid_and_pbt[n];
	int avgwait = 0;
	int count = 0;
	int count2 = 0;
	struct process *processes;

	for (int i = 3; i < argc; i++)
	{
		pid_and_pbt[i-3] = atoi(argv[i]);
	}

	// Allocating memory for n number of processes
	processes = malloc(sizeof(struct process) * n);

	if (processes == NULL)
	{
		printf("******** ERROR ********\n");
		printf("Memory allocation error\n");
		printf("Exiting now...\n");
		exit(1);
	}

	// Move values from pid_and_pbt[] to processes[]
	for (int i = 0; i < n; i++)
	{
		processes[i].pid = pid_and_pbt[i+count];
		processes[i].burst_time = pid_and_pbt[i+count+1];
		processes[i].priority = pid_and_pbt[i+count+2];
```

```
                count += 2;
        }

        // sort processes by priority using qsort

        // Finding waiting times

        for (int i = 0; i < n; i++)
        {
                if (processes[i].priority == n - count2)
                {
                        processes[i].wait_time = processes[i+count2].burst_time +
processes[i+count2].wait_time;
                        count2++;
                        i = 0;
                }
        }

        // Finding average wait time
        for (int i = 0; i < n; i++)
        {
                avgwait += processes[i].wait_time;
        }
        avgwait = avgwait / n;

        // Finding turn around times
        for (int i = 0; i < n; i++)
        {
                processes[i].t_round = processes[i].burst_time + processes[i].wait_time;
        }

        // Print results
        for (int i = 0; i < n; i++)
        {
                printf("Process: %d\n", i);
                printf("\tPID: %d\n", processes[i].pid);
                printf("\tPBT: %d\n", processes[i].burst_time);
                printf("\tTurn Around Time: %d\n", processes[i].t_round);
                printf("\tWait Time: %d\n", processes[i].wait_time);
                printf("\tPriority: %d\n\n", processes[i].priority);
        }
        printf("\nQuantum Time Unit: %d\n", qtu);
        printf("Average Wait Time: %d\n", avgwait);
        void free(void*);
}
```