Assignment 3 Report

Real-time Operating System - 48450

Student Name: Corey Stidston

Student ID: 98119910

Table of Contents

1. Introduction

This assignment involves the development of two programs which showcase core concepts of Real Time Operating Systems.

Prog_1 simulates CPU Scheduling using the Shortest-Remaining-Time-First (SRTF) algorithm. Prog_1 also demonstrates the use of threads and the queue/FIFO data structure.

To simulate CPU Scheduling, the following data was hardcoded into data structures.

Process ID	Arrive time	Bust time
1	8	10
2	10	3
3	14	7
4	9	5
5	16	4
6	21	6
7	26	2

Prog_2 demonstrates CPU Deadlock Detection using the Bankers Algorithm. Prog_2 also demonstrates the use of signals, in particular, user defined signals i.e. SIGUSR1.

To demonstrate CPU Deadlock Detection, the following data was given in a text file. This data is extracted by Prog_2 and stored in data structures.

Process ID	*Allocation*	*Request*	*Available*
	A B C	*A B C*	*A B C*
P0	0 1 0	0 1 2	0 1 2
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 2	
P3	2 1 1	3 2 2	
P4	0 0 2	0 3 5	
P5	2 1 3	0 1 1	
P6	5 2 4	1 6 4	
P7	1 3 1	5 0 3	
P8	2 4 2	1 2 4	

2. Theory of Operation

1. CPU Scheduling

CPU Scheduling is a fundamental component of multi-programmed operating systems. Scheduling the CPU to switch between processes makes the computer more productive by minimizing the time the CPU has to wait.

There are several CPU Scheduling algorithms, all which have different advantages and disadvantages. The CPU Scheduling algorithm used in Prg_1 is the Shortest-Remaining-Time-First Algorithm which favours switching to a process which has the shortest remaining execution time.

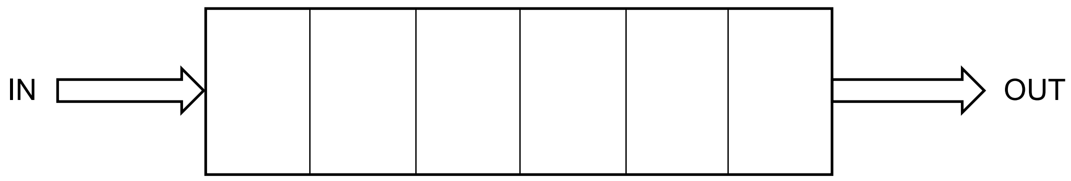
SRTF is an example of pre-emptive scheduling whereby CPU scheduling decisions can execute under all of the following circumstances: (Silberschatz 2012)

- When a process switches from the running to the waiting
- When a process switches from running to ready
- When a process switches from waiting to ready
- When a process terminates

Non-pre-emptive scheduling differs from pre-emptive scheduling in that a CPU allocated to a process will only be released when the process terminates or when it switches to waiting. Under pre-emptive scheduling, there is the chance of race conditions, whereby shared data is accessed by two or more processes causing undesired results.

2. Queue/FIFO

A queue otherwise known as FIFO, is a sequentially ordered data structure that uses the principle of 'first in, first out', whereby *'items are removed from the queue in the order in which they were added'* (Silberschatz 2012). Queues are most often used for ordering tasks and can be equated to a checkout line at a store or cars waiting at a traffic light.



Prg_1 uses an mkfifo to deliver data from thread 1 to thread 2. An mkfifo is a special file which acts as a queue/FIFO. It is similar to a pipe except that it is created differently and can exist beyond the process that creates it. (Kerrisk, M. 2017)

3. CPU Deadlock Detection

CPU Deadlocks occur when *'a waiting process is never able to change state because the resources it has requested are held by other waiting processes'* (Silberschatz 2012). In a multiprogramming environment, several processes will likely compete for the finite number of resources available on that machine. Thus, the occurrence of CPU deadlock is a realistic situation.

There are several methods that an operating system can use to prevent or deal with deadlocks, although operating systems typically do not provide deadlock prevention facilities (Silberschatz 2012).

The Banker's Algorithm was used in Prg_2 to detect CPU deadlock. When a process enters the system, it must declare the maximum number of instances of each resource type that it may need. Through this, the Bankers Algorithm can be used to determine whether the allocation of these resources will cause the system to be in an unsafe state, and therefore deadlock prone.

The following data structures are needed to implement the Banker's Algorithm:

- Available. A vector of length m (number of resources) to indicate the number of available resources for each type.
- Max. An n (number of processes) x m (number of resources) matrix to define the maximum resource demand of each process.
- Allocation. An n (number of processes) and m (number of resources) matrix to define the number of resources of each type currently allocated to a process.
- Need (aka Request). An n (number of processes) and m (number of resources) matrix indicating the remaining resource need of each process.

The Bankers Algorithm can be used to determine whether a system is in a safe state or not and whether a resource request can be safely granted. In Prog_2, we use the Bankers Algorithm to determine whether the system is safe or not based on a set of given resource data. The steps for determining the state of a system involve: (Silberschatz 2012).

1. Let Finish be a vector of n (number of processes) of type boolean, representing the completeness of each process.
2. Find an index i such that:
    a. Finish[i] == false (i.e. the process has not finished)
    b. For each resource, Need[i] is less than or equal to the resources that are currently available (represented by the Availability vector)

   If there is no such i. Go to step 4.

3. Available = Available + Allocation[i] (release he allocated resources for that process by making them available)
   Finish[i] = true;
   Go to step 2.
4. If Finish[i] == true for all i, the system is in a safe state.

4. Signals

'Signals are used in UNIX systems to notify a process that a particular event has occurred' (Silberschatz 2012). Signals follow the pattern of:

1. Signal generation
2. Signal is delivered to a process
3. Signal is handled

A signal can be received either synchronously or asynchronously. Synchronous signals are delivered to the same process where as asynchronously received signals are generated externally.

In Prg_2, a SIGUSR1 (User defined signal) is used to signal the end of file writing. SIGUSR1 and SIGUSR2 are signals that can be defined by the user. They can be used in any way but are most useful for simple inter-process communication. (GNU n.d.)

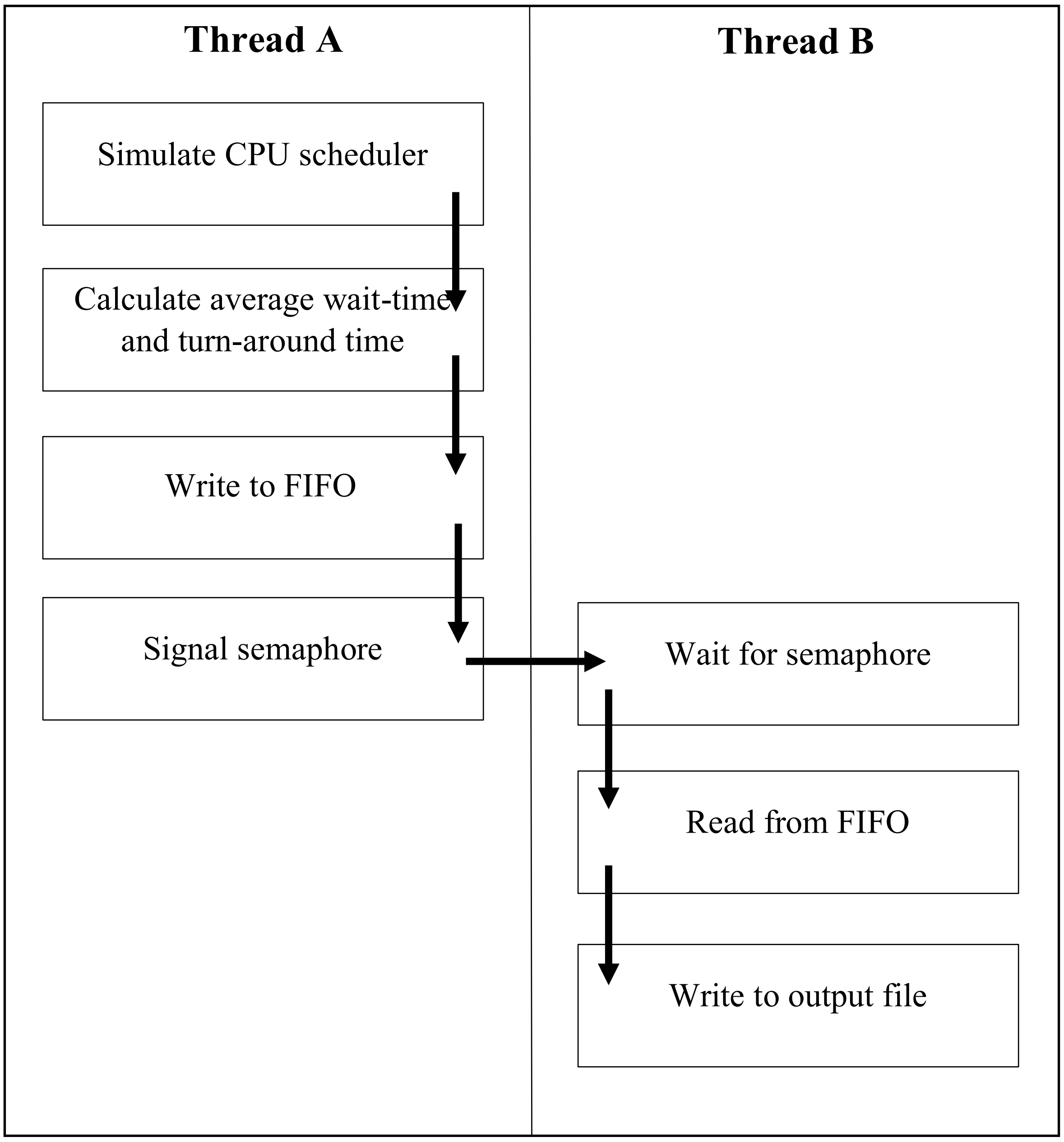
3. Program Operation

1. Prg_1

The following flow chart describes the program operation and flow of control of Prg_1.

As described below, Prg_1 operates with two threads, the purpose of Thread A is to Simulate the CPU Scheduler (using SRTF algorithm), calculate the average wait-time and turn-around time, write to FIFO and signal semaphore.

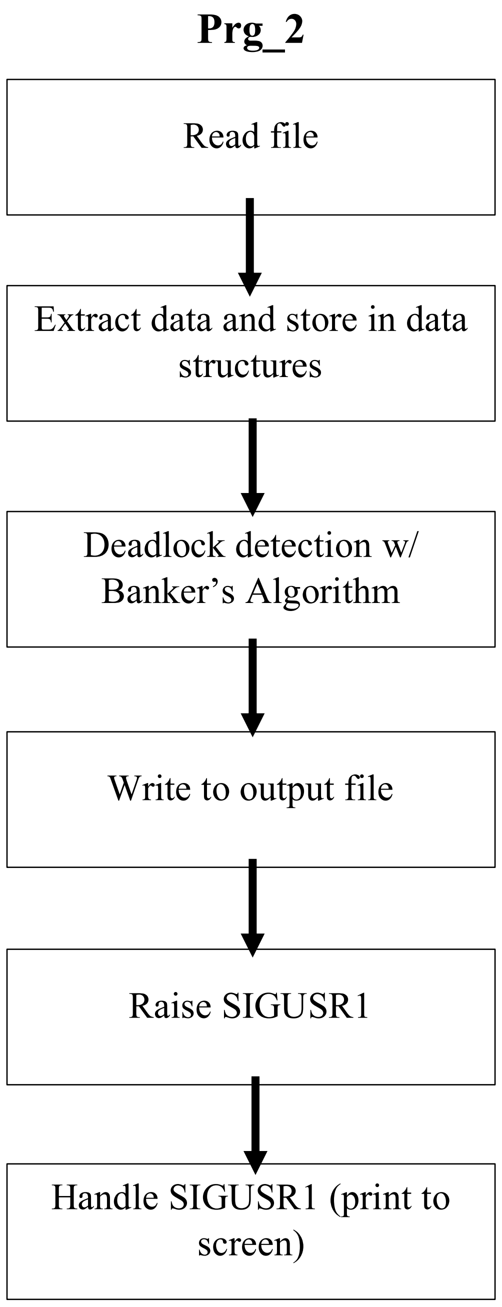
Thread B waits on a semaphore (which is signalled at the end of Thread A's execution. On signal, Thread B reads from the FIFO and writes the data to the output file.



2. Prg_2

The following flow chart describes the program operation and flow of control of Prg_2.

As described, Prg_2 has only one process and one thread of execution. The steps of the program include reading the input file, extracting and storing data, performing deadlock detection w/ Banker's Algorithm, write to output file and finally raise and handle SIGUSR1.

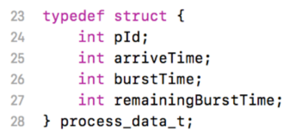

Prg_2
Read file
Extract data and store in data structures
Deadlock detection w/ Banker's Algorithm
Write to output file
Raise SIGUSR1
Handle SIGUSR1 (print to screen)

4. Implementation

The following section features screenshots of the source code and how it achieves the requirements of the assignment.

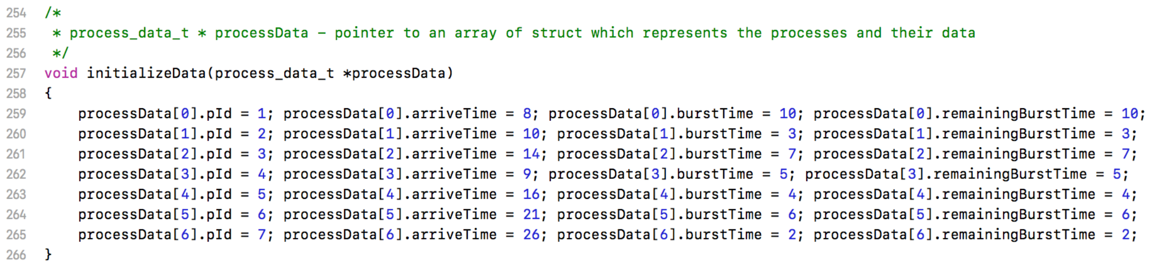
1. Prg_1

For storing data about the processes, I used a dynamic array (malloc) of a ‘process_data_t’ data structure which holds all the necessary values for simulating CPU scheduling and calculating the average wait-time and turn-around time. This can be seen below.



```c
typedef struct {
    int pId;
    int arriveTime;
    int burstTime;
    int remainingBurstTime;
} process_data_t;
```

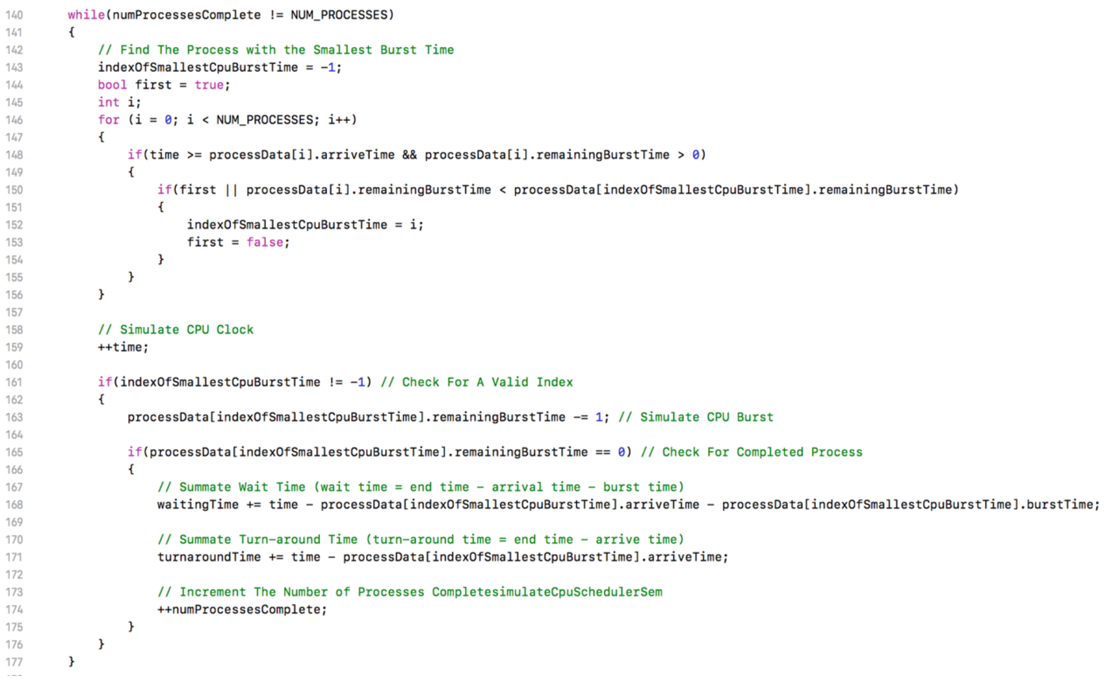
Data is placed into the structures statically in the function initializeData as seen below.



```c
/*
 * process_data_t * processData - pointer to an array of struct which represents the processes and their data
 */
void initializeData(process_data_t *processData)
{
    processData[0].pId = 1; processData[0].arriveTime = 8; processData[0].burstTime = 10; processData[0].remainingBurstTime = 10;
    processData[1].pId = 2; processData[1].arriveTime = 10; processData[1].burstTime = 3; processData[1].remainingBurstTime = 3;
    processData[2].pId = 3; processData[2].arriveTime = 14; processData[2].burstTime = 7; processData[2].remainingBurstTime = 7;
    processData[3].pId = 4; processData[3].arriveTime = 9; processData[3].burstTime = 5; processData[3].remainingBurstTime = 5;
    processData[4].pId = 5; processData[4].arriveTime = 16; processData[4].burstTime = 4; processData[4].remainingBurstTime = 4;
    processData[5].pId = 6; processData[5].arriveTime = 21; processData[5].burstTime = 6; processData[5].remainingBurstTime = 6;
    processData[6].pId = 7; processData[6].arriveTime = 26; processData[6].burstTime = 2; processData[6].remainingBurstTime = 2;
}
```

The bulk of the simulated CPU Scheduling takes place in the while loop inside the function simulateCpuScheduler, which operates on Thread A. As can be seen in the code snipper below, it follows a standard SRTF process of:

1. Finding the process with the smallest burst time
2. Performing execution of that process (CPU burst)
3. When the process is completed, marking it as completed
4. Repeating until all processes have completed



```c
    while(numProcessesComplete != NUM_PROCESSES)
    {
        // Find The Process with the Smallest Burst Time
        indexOfSmallestCpuBurstTime = -1;
        bool first = true;
        int i;
        for (i = 0; i < NUM_PROCESSES; i++)
        {
            if(time >= processData[i].arriveTime && processData[i].remainingBurstTime > 0)
            {
                if(first || processData[i].remainingBurstTime < processData[indexOfSmallestCpuBurstTime].remainingBurstTime)
                {
                    indexOfSmallestCpuBurstTime = i;
                    first = false;
                }
            }
        }

        // Simulate CPU Clock
        ++time;

        if(indexOfSmallestCpuBurstTime != -1) // Check For A Valid Index
        {
            processData[indexOfSmallestCpuBurstTime].remainingBurstTime -= 1; // Simulate CPU Burst

            if(processData[indexOfSmallestCpuBurstTime].remainingBurstTime == 0) // Check For Completed Process
            {
                // Summate Wait Time (wait time = end time - arrival time - burst time)
                waitingTime += time - processData[indexOfSmallestCpuBurstTime].arriveTime - processData[indexOfSmallestCpuBurstTime].burstTime;

                // Summate Turn-around Time (turn-around time = end time - arrive time)
                turnaroundTime += time - processData[indexOfSmallestCpuBurstTime].arriveTime;

                // Increment The Number of Processes CompletesimulateCpuSchedulerSem
                ++numProcessesComplete;
            }
        }
    }
```

For calculating the average waiting time and turn-around time, I summate waiting times and turn-around times once a process has completed (as seen on lines L:168, L171) and then after CPU scheduling simulation, I divide it by the number of processes. This makes the calculation of these values quick and easy, avoiding having to unnecessarily loop through data on completion of CPU scheduling purely to calculate the values.

Writing to the FIFO involves calling the function writeToFIFO twice for each calculation, pushing both the value and a related description.

```
    // Push Average Waiting Time To FIFO
    writeToFIFO(parameters->fifo, averageWaitingTimeDescription, averageWaitingTime);
    writeToFIFO(parameters->fifo, averageTurnaroundTimeDescription, averageTurnaroundTime);
```

The function concatenates the description and value into a char buffer and then writes this to the mkfifo.

```
void writeToFIFO(fifo_t * fifo, char * description, float value)
{
    char buffer[fifo->elementSizeInBytes];
    if(snprintf(buffer, sizeof(buffer), "%s = %f.\n", description, value) < 0)
    {
        perror("Error writing to buffer."); exit(1);
    }
    if(write(*fifo->fd, buffer, sizeof(buffer)) == -1)
    {
        perror("Failed to write to FIFO."); exit(1);
    }
    *fifo->numElements += 1;
}
```

As can be seen below in the snippet of the writeToFile function, Thread B, waits to be signalled by Thread A, and once signalled, writes the contents of the FIFO to the output file.

```
    sem_wait(&writeToFileSem);

    // Read From FIFO
    char buffer[parameters->fifo->elementSizeInBytes];
    int i;
    for(i = 0; i < *parameters->fifo->numElements; ++i)
    {
        if(read(*parameters->fifo->fd, buffer, sizeof(buffer)) == -1) // Read Into Buffer
        {
            perror("Failed To Read From FIFO."); exit(1);
        }
        fputs(buffer, writeFile); // Write To Output File
        if(ferror(writeFile))
        {
            printf("Error Writing to File.\n");
        }
    }
```

2. Prg_2

For storing the data used by Prg_2 to detect a CPU deadlock, I used the data structures described in the Banker's Algorithm on page 9. These are:

1. Availability vector of size NUM_RESOURCES (3)
2. Allocation Matrix of size MAX_NUM_PROCESSES (20) x NUM_RESOURCES (3)
3. Request Matrix of size MAX_NUM_PROCESSES (20) x NUM_RESOURCES (3)

As seen, I have specified the number of resources and therefore the program is not dynamic enough to respond to an input file containing more than or less than 3 resource types. However, the program can respond to an input file containing up to 20 processes (although this can be increased or decreased).

```c
#define MAX_NUM_PROCESSES 20

typedef enum
{
    RESOURCE_A, // 0
    RESOURCE_B, // 1
    RESOURCE_C, // 2
    NUM_RESOURCES // 3
} RESOURCE;

unsigned int availabilityVector[NUM_RESOURCES];
unsigned int allocationMatrix[MAX_NUM_PROCESSES][NUM_RESOURCES];
unsigned int requestMatrix[MAX_NUM_PROCESSES][NUM_RESOURCES];
unsigned int numProcesses;
```

Reading from the file is simple and involves discarding the first two lines and then slicing the strings based on spaces and tabs and passing this to the extractData.

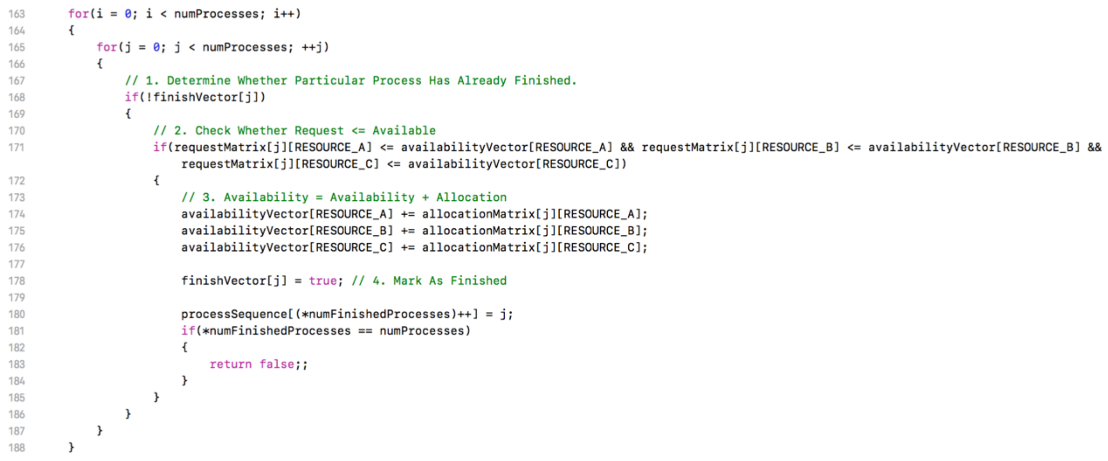
```c
    while(fgets(lineFromFile, BUFFER_SIZE, readFile) != NULL)
    {
        if(lineNo > 2) // Ignore Column Names
        {
            truncatedLine = strdup(lineFromFile); // Return Null Terminated Byte String

            int linePos = 0;
            int processId;
            while((lineSegment = strsep(&truncatedLine, delimiters)) != NULL ) // Slice String On Spaces and Tabs
            {
                if(strlen(lineSegment) >= 1) // Filter Out Unuseful Information
                {
                    extractData(linePos, &processId, lineSegment); // Extract and Store Data
                    ++linePos;
                }
            }
            ++numProcesses;
        }
        ++lineNo;
    }
```

Extract data converts the char values into integers and stores them into the appropriate data structure based on its position in the line.

```c
void extractData(unsigned int linePos, int * processId, char * lineSegment)
{
    /* Store Data Based On Line Position */
    switch(linePos)
    {
        case 0: {
                *processId = atoi(&lineSegment[1]);
                if(*processId > MAX_NUM_PROCESSES)
                {
                    printf("Error: this program does not accept more than %i number of processes.\n", MAX_NUM_PROCESSES); exit(1);
                }
            }
            break;
        case 1: allocationMatrix[*processId][RESOURCE_A] = atoi(lineSegment);
            break;
        case 2: allocationMatrix[*processId][RESOURCE_B] = atoi(lineSegment);
            break;
        case 3: allocationMatrix[*processId][RESOURCE_C] = atoi(lineSegment);
            break;
        case 4: requestMatrix[*processId][RESOURCE_A] = atoi(lineSegment);
            break;
```

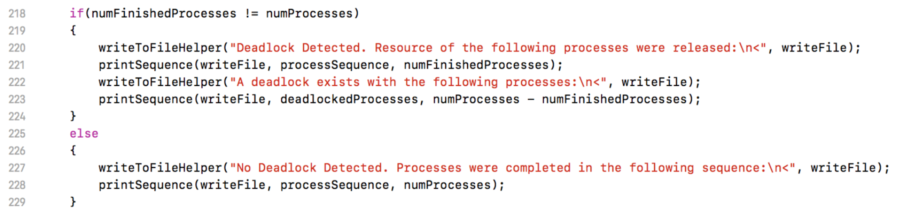
Deadlock detection is a simple conversion from the Banker's Algorithm:

1. Find a process whose request is less than or equal to the current availability
2. Release that process' allocated resources
3. Mark that process as finished
4. Repeat until all processes have finished or the nested loop ends and therefore a deadlock exists



```c
    for(i = 0; i < numProcesses; i++)
    {
        for(j = 0; j < numProcesses; ++j)
        {
            // 1. Determine Whether Particular Process Has Already Finished.
            if(!finishVector[j])
            {
                // 2. Check Whether Request <= Available
                if(requestMatrix[j][RESOURCE_A] <= availabilityVector[RESOURCE_A] && requestMatrix[j][RESOURCE_B] <= availabilityVector[RESOURCE_B] &&
                    requestMatrix[j][RESOURCE_C] <= availabilityVector[RESOURCE_C])
                {
                    // 3. Availability = Availability + Allocation
                    availabilityVector[RESOURCE_A] += allocationMatrix[j][RESOURCE_A];
                    availabilityVector[RESOURCE_B] += allocationMatrix[j][RESOURCE_B];
                    availabilityVector[RESOURCE_C] += allocationMatrix[j][RESOURCE_C];

                    finishVector[j] = true; // 4. Mark As Finished

                    processSequence[(*numFinishedProcesses)++] = j;
                    if(*numFinishedProcesses == numProcesses)
                    {
                        return false;;
                    }
                }
            }
        }
    }
```

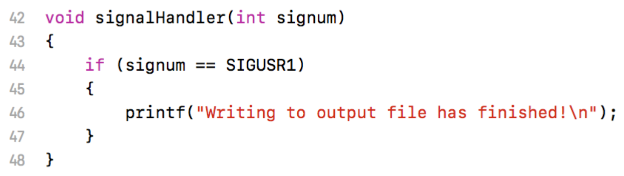
Writing to the output file involves using the helper methods I created: writeToFileHelper and printSequence, which are used to format the data and write it to the output file.



```c
    if(numFinishedProcesses != numProcesses)
    {
        writeToFileHelper("Deadlock Detected. Resource of the following processes were released:\n<", writeFile);
        printSequence(writeFile, processSequence, numFinishedProcesses);
        writeToFileHelper("A deadlock exists with the following processes:\n<", writeFile);
        printSequence(writeFile, deadlockedProcesses, numProcesses - numFinishedProcesses);
    }
    else
    {
        writeToFileHelper("No Deadlock Detected. Processes were completed in the following sequence:\n<", writeFile);
        printSequence(writeFile, processSequence, numProcesses);
    }
```

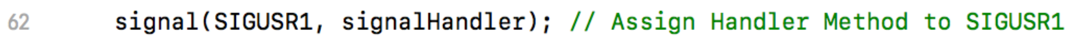
My use of signals inside Prog_2 involves:

1. Defining a signal handler



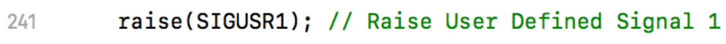
```c
void signalHandler(int signum)
{
    if (signum == SIGUSR1)
    {
        printf("Writing to output file has finished!\n");
    }
}
```

2. Assigning the handler to SIGUSR1



```c
    signal(SIGUSR1, signalHandler); // Assign Handler Method to SIGUSR1
```

3. Raising SIGUSR1

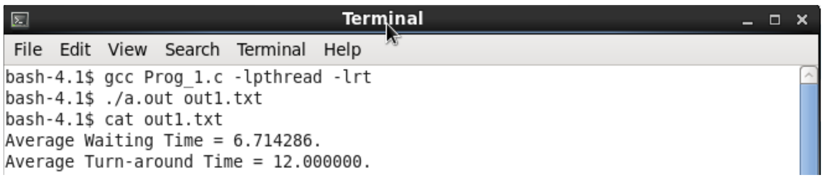


```c
    raise(SIGUSR1); // Raise User Defined Signal 1
```

5. Testing

Testing was performed on Red Hat Linux. The results of the tests can be seen below.

Execution of Prg_1 produces an average waiting time of 6.714286 and a turn-around time of 12. This is aligned with my hand calculation and therefore passes my test of Prg_1 code.



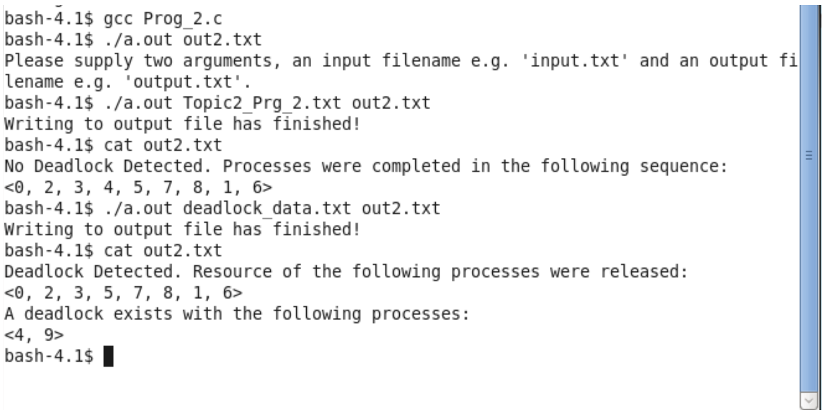
```
bash-4.1$ gcc Prog_1.c -lpthread -lrt
bash-4.1$ ./a.out out1.txt
bash-4.1$ cat out1.txt
Average Waiting Time = 6.714286.
Average Turn-around Time = 12.000000.
```

Execution of Prg_2 produces a result of no deadlock detected with a process sequence of:

<0, 2, 3, 4, 5, 7, 8, 1, 6>

This is aligned with my hand calculation and passes the test for determining a safe system.

To test whether the program could detect an unsafe system, I created the deadlock_data.txt which has large resource request values for process 4 and process 9 (thus creating a deadlock). As seen in the results below, the code passes the test for detecting an unsafe system.



```
bash-4.1$ gcc Prog_2.c
bash-4.1$ ./a.out out2.txt
Please supply two arguments, an input filename e.g. 'input.txt' and an output fi
lename e.g. 'output.txt'.
bash-4.1$ ./a.out Topic2_Prg_2.txt out2.txt
Writing to output file has finished!
bash-4.1$ cat out2.txt
No Deadlock Detected. Processes were completed in the following sequence:
<0, 2, 3, 4, 5, 7, 8, 1, 6>
bash-4.1$ ./a.out deadlock_data.txt out2.txt
Writing to output file has finished!
bash-4.1$ cat out2.txt
Deadlock Detected. Resource of the following processes were released:
<0, 2, 3, 5, 7, 8, 1, 6>
A deadlock exists with the following processes:
<4, 9>
bash-4.1$ 
```

6. Conclusion

In conclusion, assignment 3 has solidified my understanding of core RTOS concepts that have been taught thus far in the subject. I have a new found understanding of CPU scheduling, named pipes, CPU deadlock detection and signals. This will no doubt help me in my future development of low-level real time applications.

7. References

Silberschatz, P. B. Galvin & G. Gagne, 2012, Operating System Concepts, 9th edition, John Wiley & Sons, New York.

Kerrisk, M. 2017, Linux Programmer's Manual, man7.org, <http://man7.org/linux/man-pages/man3/mkfifo.3.html>

GNU n.d., Miscellaneous Signals, GNU.org, <https://www.gnu.org/software/libc/manual/html_node/Miscellaneous-Signals.html>