Introduction

Section 1

The key part of any computer system is the Operating System(OS). Through this report, a discussion of the concept of processes within Linux Operating Systems will transpire. However, all Linux OS's are built on the Linux Kernel and the Kernel is in charge of Process Management and the processes that will be discussed. Linux is a preemptive multitasking OS such that it applies some criteria for resource allocation. The topics will be divided into two main sections, Section one will illustrate Process Management, Process Creation, Process Scheduling, and Process Destruction. Furthermore, in Section Two this report investigates how Linux handles multitasking and will describe Race Condition, Mutex Locks, Critical Section Problem, Solutions to Critical Section Problem, and Deadlock & Starvation.

The Linux system was chosen for writing and demonstrating code on it. The main reason for using Linux is as follows.

1. It is considered the best Unix System for programming.
2. Huge amount of development tools.
3. Based on the C programming language.
4. Can create, compile and run C programs by Ubuntu's terminal.

The operating system of Linux permits multitasking which means the Linux Operating System can run many programs at the same time, for example, simultaneously surfing the Web and chatting via a chat application.
This report shall investigate how Linux manages and schedules these processes and how processes are created and destroyed, and also introduces several function's details.

As the foundation of creating any programs, the first step is always setting up the work environment of the system. It also includes the installation of all important build-essential packages (for example, "at" packages). Then the user can start to write a C program, using Ubuntu's graphical Text Editor, and save it in ".c" format, as C language files always have such file format. When the code is ready to compile, the user can perform it with the default gcc Compiler, which is integrated into Ubuntu's terminal. For example, "gcc sampleProgram.c -o executableProgram". By doing that you compile (gcc) your code (sampleProgram.c) by providing a name for the executable version of the file (executableProgram). After compiling code and creating an executable file, the user can run the program by typing its name in the command prompt ("./executableProgram").

The following figure(1,2,3) indicates the program code of section one.

```c
#include <stdio.h>                    // library for basic input output on C language
#include <stdlib.h>                   // for using "system" command
#include <stdbool.h>                  // for using bool variable, false/true declares
#include <string.h>                   // now we can add 2 strings
#include <unistd.h>                   // for cloning processes, getting PID and PPID
#include <sys/wait.h>                 // for scheduling tasks

void show() {
    system("ps -A");                  // list all running processes
}

void create() {
    pid_t pid;                        //pid_t - data type for any PID which we want to create
    pid = fork();                     //clone parent process to create child process

    printf("\nChild Process created! \n");
    printf("PID: %d", getpid());      // PID - child process identifier
    printf("\nPPID: %d\n", getppid());  // PPID - parent's PID

}

void killProcess() {
    char processPID[10];
    char getName[30];
    char kill[20];

    printf("Enter the PID to Kill: ");
    scanf("%s", processPID);          // gets PID

    strcat(getName, "ps -p ");        // ps -p
    strcat(getName, processPID);      // ps -p PID
    strcat(getName, " -o comm=");     // ps -p PID -o comm=       RETURNS NAME OF PROCESS BY GETTING ITS PID

    strcat(kill, "kill ");            // kill
    strcat(kill, processPID);         // kill PID
    system(kill);                     // kills process by PID

    printf("\n\nThis program was killed!");

    system(getName);                  // outputs name of killed process
}
```

Figure 1. Process program is written in c.

```c
void schedule() {
    char mins[5];
    char name[15];
    char command[60];

    printf("\n\n");
    printf("Insert name for text file:\n");
    printf("(should contain 1 solid word)\n");

    scanf("%s", name);

    strcat(command, "echo \"touch ");   // "echo "touch
    strcat(command, name);              // "echo "touch NAMEOFFILE
    strcat(command, " | date > ");      // "echo "touch NAMEOFFILE | date >
    strcat(command, name);              // "echo "touch NAMEOFFILE | date > NAMEOFFILE
    strcat(command, "\" | at now + ");  // "echo "touch NAMEOFFILE | date > NAMEOFFILE" | at now +

    printf("\n\n");
    printf("When you want to start it?");
    printf("\n\n");
    printf("Now:\t\t\tEnter \"0\"\n");
    printf("1 minute later:\t\tEnter \"1\"\n");
    printf("5 minutes later:\tEnter \"5\"\n");
    printf("15 minutes later:\tEnter \"15\"\n");

    scanf("%s", mins);

    strcat(command, mins);              // "echo "touch NAMEOFFILE | date > NAMEOFFILE" | at now + X
    strcat(command, " minutes");        // "echo "touch NAMEOFFILE | date > NAMEOFFILE" | at now + X minutes

    system(command);                    // system, schedule given task!

    printf("\n\nJob scheduled!\n");

}
```

Figure 2. Process program is written in c.

```c
int main() {
    bool menu = true;
    int option = 0;

    while (menu) {                  //Starting of User Interface for Main Menu

        printf("\n\n");
        printf("_____________________________________________\n");
        printf("|\t\t\t\t\t\t|\n");
        printf("|\tChoose Task Number\t\t\t|\n");
        printf("|\t\t\t\t\t\t|\n");
        printf("|\t1. Show All Processes\t\t\t|\n");
        printf("|\t2. Create a child process\t\t|\n");
        printf("|\t3. Schedule a process\t\t\t|\n");
        printf("|\t4. Kill a process\t\t\t|\n");
        printf("|\t\t\t\t\t\t|\n");
        printf("|\t! Enter other digit to exit program !\t|\n");
        printf("_____________________________________________\n\n");

        scanf("%d", &option);       //Choose option from menu

        switch (option)
        {
        default:                    // case which executes when "option" value is not 1 to 4
            menu = false;
            printf("Program Exiting...\n");
            break;

        case 1:
            show();
            break;

        case 2:
            create();
            break;

        case 3:
            schedule();
            break;

        case 4:
            killProcess();
            break;
        }
    }
```

Figure 3. Process program is written in c.

Process Management

Process Management is an essential part of any OS. The OS must manage the Central Processing Unit(CPU), memory space, file storage space, and input/output devices(I/O devices). A process refers to the series of steps or basic functions necessary for the normal execution of instructions. “*Management is a process because it performs a series of functions, like, planning, organizing, staffing, directing and controlling in a sequence*”, (ostoday.org, 2021). A process needs different assets such as CPU time, memory, files, and I/O devices, to complete its tasks. An operating system such as Linux is there to provide those assets. There are different types of processes that need to be managed, two types are

user processes and system processes. Most processes are user processes which is a process started by a user and run in the user space. A system process or a kernel process runs in kernel mode and only uses kernel space. Kernel processes execute in the more privileged kernel protection domain, a kernel process can access data that user processes cannot. (IBM.com, 2020) In Linux, each process is assigned a Process ID or PID. This is how the operating system identifies and keeps track of processes.

A process is a program in execution, when the process executes the process state changes. There are five process states: new, ready, running, waiting, and terminated. When the process state is new the process is being created. When a process is running the process instructions are being executed. When a process is waiting it is waiting for an event to occur such as an input/output request. When the process is in the ready state it is waiting to be assigned to a processor and if it is in the terminated state is killed. The OS is responsible for creating and deleting user processes and system processes, scheduling processes, stopping and starting processes, process synchronization, and process communication. System calls are used to provide the interface between a process and the operating system. For process management certain system calls are used such as fork() to create a new process, exec() to run a new program, wait() to make a process wait, exit() to terminate a process. Every process is represented in the operating system as Process Control Block (PCB). The PCB holds all the data the process needs to start and restart plus scheduling and memory management. Linux Operating systems have a scheduler that tracks how long the process holds the CPU and will periodically activate the scheduler. Ubuntu Linux supports four scheduling algorithms: Earliest Deadline First(EDF), First In First Out(FIFO), Round-Robin, and the default scheduler. The scheduler will take a process from the ready state and give it access to the CPU and remove it to the waiting state and tend to it through its life cycle.

If the subject is about process management the topic of threads needs to be addressed because multithreaded processes are made up of more than one thread and threads in a process may have different tasks but share the same memory with its process. There are three multithreading models Many-to-One, One-to-One, and Many-to-Many. Linux is an example of the One-to-One model, which maps each user thread to a kernel thread. When a process is managed so are its threads.

The fundamental principle of process management is separated into two operations that are usually combined but not always as the program submitted with this report does not use the exec() command. A new process is created with the fork() system call and a new program is run after the exec() system call; however, these are two separate functions. As in the submitted program, a new process is created with the fork() command without running a new program. The Linux model allows for simplicity by using fork() you are cloning a process and it is not necessary to tailor the new process to a new environment since it will be running in the existing environment and if changes need to be made to change can be implemented before running the new process. Now the process will have all the information needed by the OS to track the frame of reference of a single execution of a single program. This frame of reference can be broken down into three distinct properties: the process identity, environment, and context.

The process identity is made up of the following data: Process Identification(PID), Credentials, Personality, and Namespace. Most of these items can be altered if they need to be, however, the PID of a process cannot be changed and uniquely identifies that process until termination. The process environment is inherited from its parent process and is made up of two arrays: an argument array and an environment array. When a new process is invoked a new environment is set up, calling the exec() a process will provide the environment for the new program. The environment-variable mechanism of passing environment variables from one process to the next and the inheriting of variables from parent to child provide flexible ways of passing information from one process to another. Process identity and the process environment are both set up when the process is created and usually remain the same unless a process needs to change part of its identity or its environment. The process context is the state of a running process and is changing continually. Process context consists of the following items: scheduling context, accounting, file table, file-system context, signal-handler table, and Virtual memory context. The most important piece of the process context is the scheduling context has all the information needed by the scheduler to stop and restart processes. This includes the registers used by the process and process priority. System calls and interrupts use the processes kernel stack which is a separate area of kernel memory reserved for use by kernel-mode code. The accounting part keeps track of the resources used by each process and the total resources used by the process throughout its lifecycle. The file table is an array that keeps track of all open files of a process. The file-system context keeps track of requests to open files and contains the processes root directory, working directory, and namespace.

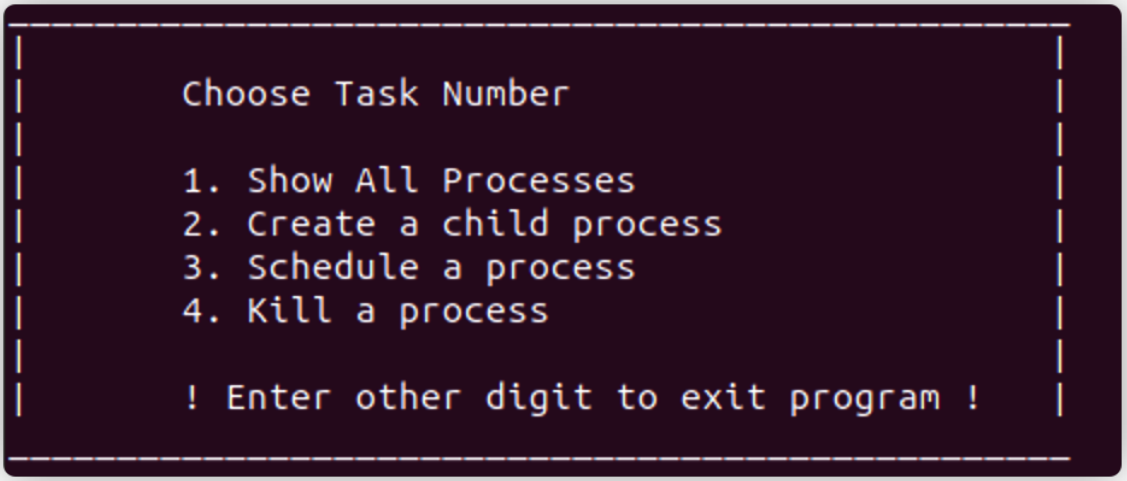


Figure 4 The interface of process management

```
while (menu) {                  //Starting of User Interface for Main Menu

    printf("\n\n");
    printf("_____________________________________________\n");
    printf("|\t\t\t\t\t\t|\n");
    printf("|\tChoose Task Number\t\t\t|\n");
    printf("|\t\t\t\t\t\t|\n");
    printf("|\t1. Show All Processes\t\t\t|\n");
    printf("|\t2. Create a child process\t\t|\n");
    printf("|\t3. Schedule a process\t\t\t|\n");
    printf("|\t4. Kill a process\t\t\t|\n");
    printf("|\t\t\t\t\t\t|\n");
    printf("|\t! Enter other digit to exit program !\t|\n");
    printf("_____________________________________________\n\n");
```

Figure 5. The interface's code part of process management

Figure 4 shows the menu interface as it appears when the program is run and figure 5 shows the menu interface in C code form. The menu box provides four options for the user to select and two tips to guide the user, all that contents in the interface achieved by the "printf", and switch function.

The main output function in the C programming language is the "printf" function that was used in the program submitted with this report to print out a formatted menu. To use printf() studio.h header file must be imported into the code of the program to do this one must use #include <studio.h> statement. To print an int type one can use printf with "%d" and that will reference an integer that is after the comma. In program 1 submitted with this report an example of printf() and %d is as follows: printf("PID: %d", getpid()). To reference a string variable one can use %s format specifier in C printf statement. To read input from a user the

scanf function is commonly used. An example from program 1 of scanf function and %s is as follows: scanf("%s", mins). (programiz.com, 2020)

Process Creation

To create a process in Linux you call the fork() system call creates a new process from an existing process called the parent process, at the period of execution course, the parent process will create new processes which are called child processes. Any child processes from the same parent are called siblings. In a Linux system, a system call, which is a procedure that provides communication between a process and the operating system, called fork is used to create a clone of the calling process; the clone is a child of the calling process which is the parent. This child process can itself create more processes forming a process hierarchy.

When a computer is turned on the boot “process” begins which finds the BIOS (Basic Input/Output System) and runs it. The BIOS is written on to Read-Only Memory(ROM) it checks the system and starts the kernel. In Linux, a process called init is in the boot image. It waits for someone to log in, the login process executes a shell to accept commands, these commands may start up more processes using fork() and exec(). This means that all processes in the whole system belong to a single tree, with init at the root. (Tanenbaum, 2015). Some of these processes are foreground processes meaning they get used by a user, others are background processes and are not associated with particular users. An example of a background process is a process that waits for email or a process that waits for a printing command. A background process is any process that doesn't interact with the user. Background processes run independently from other tasks and do not require us to do anything. Similar to how the body's breathing process works, we don’t think or do anything, it just happens. (study.com, 2021)
.
In the C program accompanying this report, the data type pid_t is used for the processes that are created. The function getppid() will return the Parent Process Identification(PID) and the function getpid() will return the current process PID which is the child process.

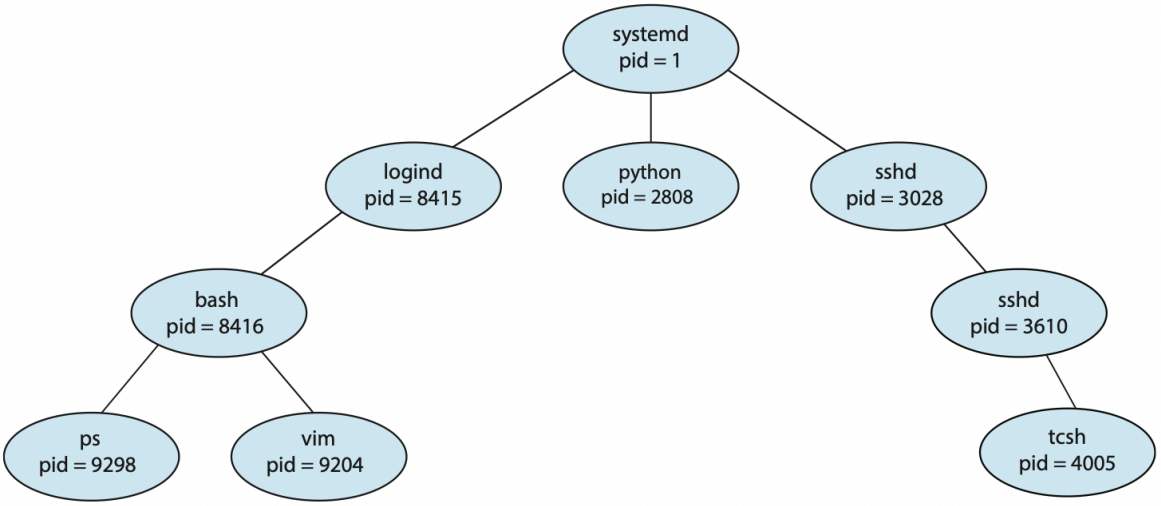


Figure 6. A treemap of Processes on a Linux system(*Galvin & Gagne Pub 2018, P116)*

Figure 6 describes a treemap for the typical Linux operating system, including each process's name and its PID.

There are four principal events that cause processes to be created.

1. System initialization.
2. Execution of a process creation system call by a running process.
3. A user requests to create a new process.
4. Initiation of a batch job.

System initialization:
"When an operating system is booted, typically numerous processes are created." Those processes can be defined as foreground processes and background processes, about foreground process take charge of performing work and interacting for the operator, and for the background process normally does not directly interact with the operator unless the application was designed as a background process for a particular purpose such as chat application, only active when the application received a new message or a face-time video call, for other time the chat application will in an inactive situation.

Execution of a process creation system call by a running process:
"Often a running process will issue system calls to create one or more new processes to help it do its job", for higher efficiency consideration, the process which contains an amount of data or steps can be divided into two or more sections to deal with.
For example, a multiprocessor allows each process to run on a different CPU to make processes faster.

A user requests to create a new process:
"In interactive systems, users can start a program by typing a command or (double)clicking on an icon. For example, in the Linux system open the terminal and type "code + target program name" then this program will be processed in an editor by using command order, or

the operator can select the target program and double click on it, also do the same job as using the command.

Initiation of a batch job:
"The last situation in which processes are created applies only to the batch systems found on large mainframes." for the inventory management system when users submit batch data to the system in a period of time. if the operating system has resources to deal with other processes, then the operating system will create a new process and arrange data into the process queue then run the next job from the input queue in it.

Technically, in all these cases, "*a new process is created by having an existing process execute a process creation system call".* That process can be an operator running the process with any input computer devices such as mouse click target program, keyboard type command. or the operating system creates the process to deal with batch data.

The new process created by an existing process executes a system call, *"the system call gives the operating system order to create a new process and indicates, directly or indirectly, which program to run in it"(ANDREW S. TANENBAUM HERBERT BOS,2015,p435).*

In user-space, a program calls fork, which results in a system call to the kernel function called sys_fork(). The function relationships are shown below graphically in Figure 7.

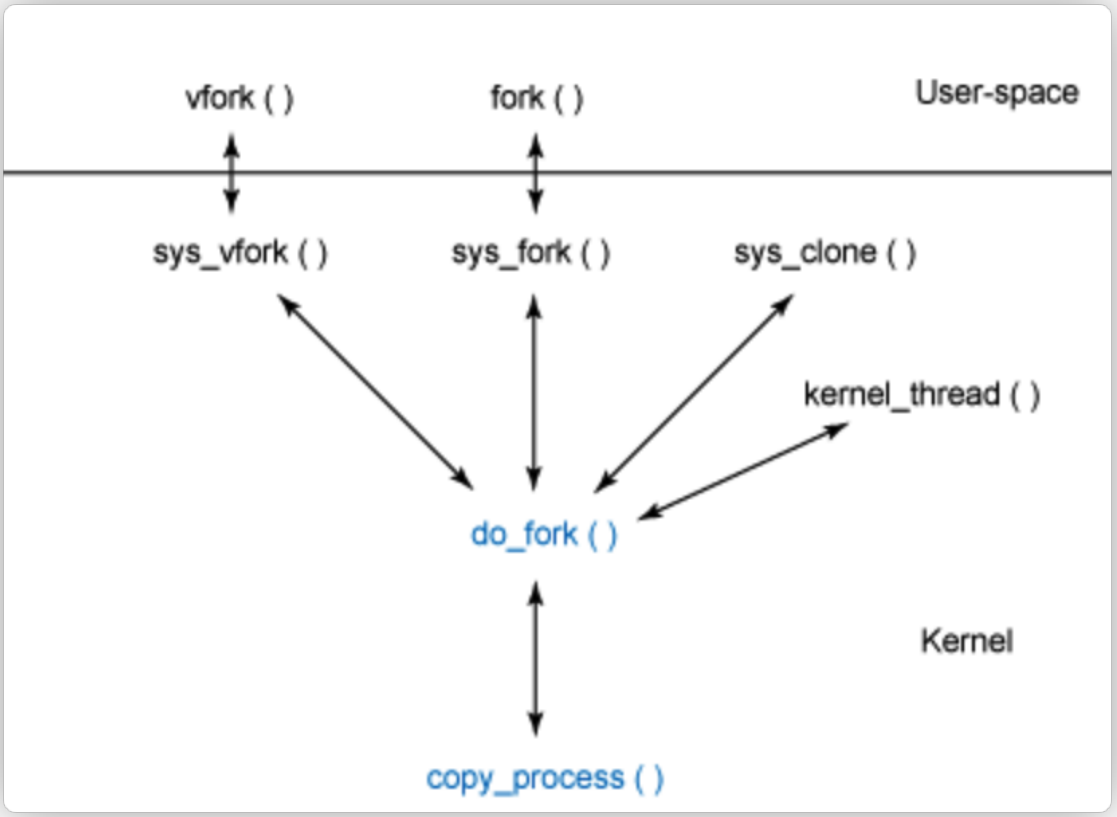


Figure 7. Function hierarchy for process creation

The do_fork function provides the basis for process creation and begins with a call to allocate a new PID. *“Next, do_fork checks to see whether the debugger is tracing the parent process. If it is, the CLONE_PTRACE flag is set in the clone_flags in preparation for forking. The do_fork function then continues with a call to copy_process, passing the flags, stack, registers, parent process, and newly allocated PID.”(developer.ibm.com, M. Jones,2008).* The copy_process function is where the new process is created as a copy of the parent. This function performs all actions except for starting the process.

The process creation menu interfaces, program code screenshots, and code explanation are shown in the below figures and contents.

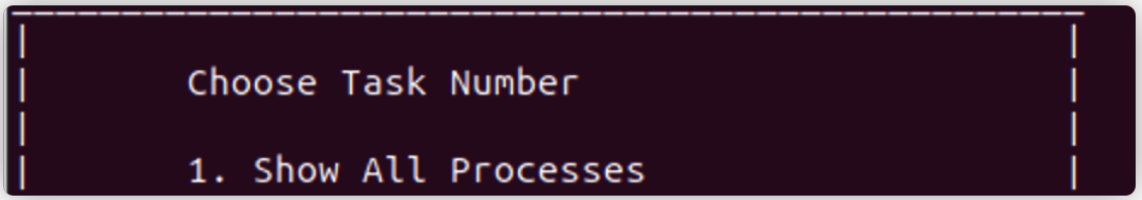


Figure 8. The interface of the first function(Show all running Processes)

```
void show() {
    system("ps -A");    // list all running processes
}
```

Figure 9. The interface of the first function(Show all running Processes-code part)

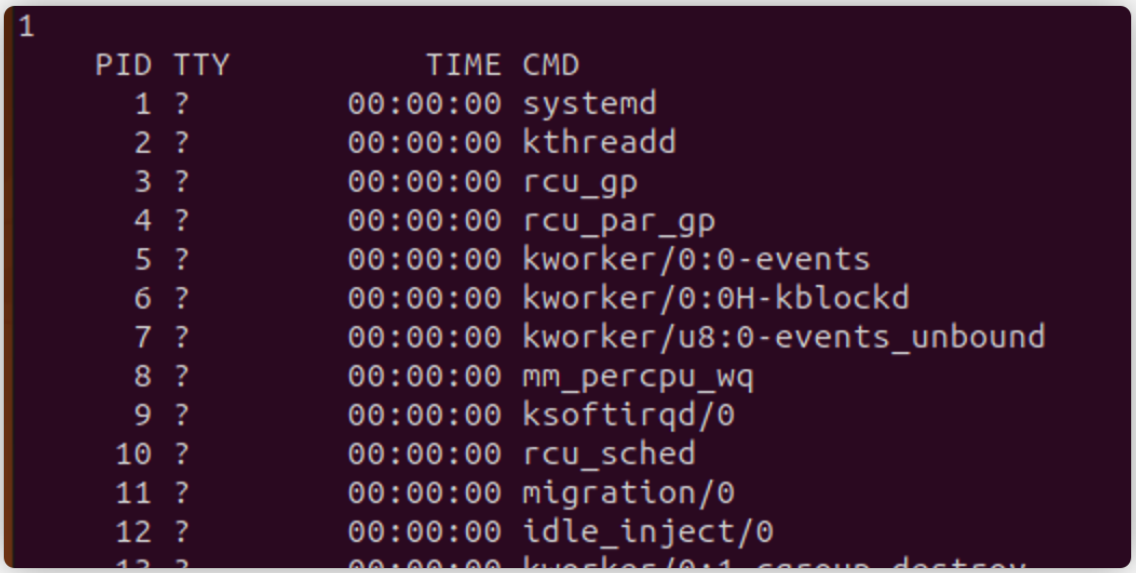


Figure 10. The results display a screenshot of the first function(Show all running Processes) The command of “ps -A” can check all of the currently running processes and display them on the screen.

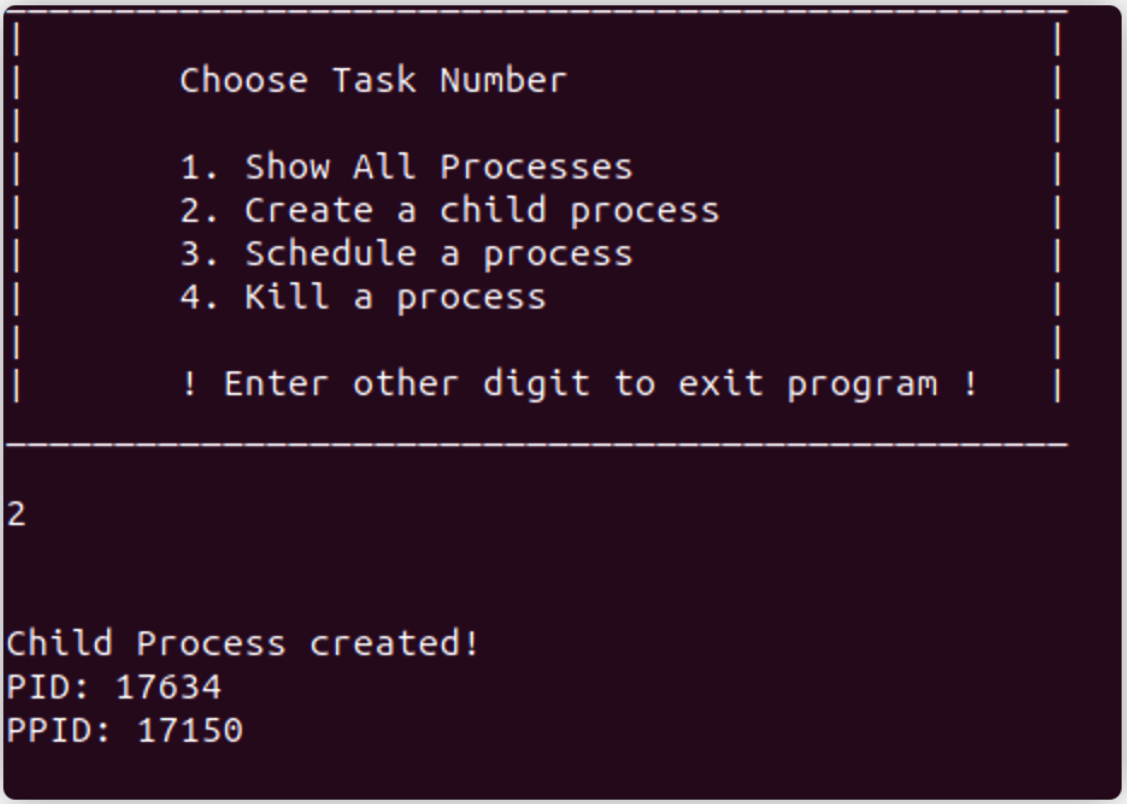


Figure 11. The interface of the second function(Create a child process)

```c
void create() {
    pid_t pid;              //pid_t - data type for any PID which we want to create
    pid = fork();               //clone parent process to create child process

    printf("\nChild Process created! \n");
    printf("PID: %d", getpid());        // PID - child process identifier
    printf("\nPPID: %d\n", getppid());  // PPID - parent's PID

}
```

Figure 12. The interface of the second function(Create a child process-code part)

In this part of Program #1 which was submitted with this report, it illustrated how a new Process Identifier (PID) was created and assigned to a process that was a clone of an existing process. After using the “fork()” method, the PID of the main process which is being cloned acquires the status of Parent Process Identifier (PPID), while the newly created PID becomes the child of it. Data type called “pid_t” was used to create and fork (clone) the main process. To demonstrate the results of actions we can use getpid() to check the child’s PID and getppid() to check PPID.

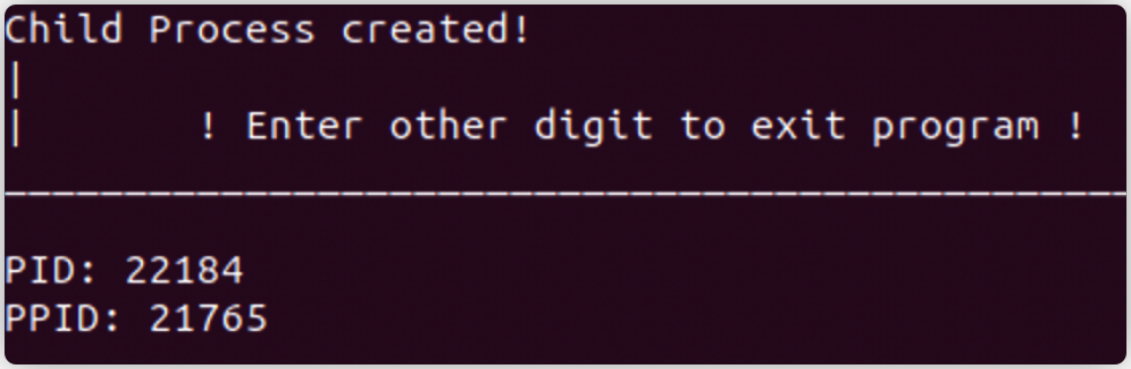


Figure 13. The result of the second function(create a child process)

As figure 13 displays from Program #1 after the create process function is successful the screen will show the child the process id and its parent's process id.

Process Scheduling

Linux is a preemptive multitasking operating system, which makes scheduling algorithms very important to maximize performance. The scheduling algorithms use some conditions to decide how long to allocate their resources to that task before moving on to another task. With a good scheduler, your computer would have a big difference in perceived performance and user satisfaction. In the early days of computing, input was in the form of card images on magnetic tape the scheduling was linear; it just ran what was on the tape. With the Advent of multiprogramming systems, the scheduling algorithms got more complex since there were more users waiting for service. The scheduler must also take into account the efficient use of the CPU because switching processes also called swapping can be time-consuming. During swapping a context switch is used, this stores and restores the state or context of a CPU in a Process Control Block or PCB so that a process execution can be resumed from the same point at another time. Context switching makes it possible for multiple processes to share a single CPU. After the current process state is stored in the PCB, the state of the next process to run is loaded from its own PCB and used to set the Program Counter, registers, State, Accounting information. Then the second process can start executing. However, context switching takes up valuable time and too many can lead to latency which would impact performance.

Most operating systems make a distinction between processes, lightweight processes, and threads. Linux uses the task structure to represent any execution context. Hence a single-threaded process will be represented with one task structure and a multithreaded process will have one task structure for each of the user-level threads. The kernel itself is multithreaded and has kernel-level threads which are not associated with any user process and are executing kernel code. Scheduling kernel tasks are an integral part of the scheduler, kernel tasks comprise both tasks that are requested by a running thread and tasks that execute internally for the kernel itself.

There are many different scheduling algorithms for example First Come First Serve (FCFS), Shortest Job First (SJF), Shortest Remaining Time First (SRTF), Round Robin (RR) scheduling and priority scheduling are a few. When processes are easily classified into different groups such as foreground processes and background processes they can be separated into different queues to be scheduled. A multi-level queue scheduling algorithm partitions the ready queue into several separate queues. These two types of processes have different requirements and so may have different scheduling necessities. These queues may have their own scheduling algorithm to suit their needs.

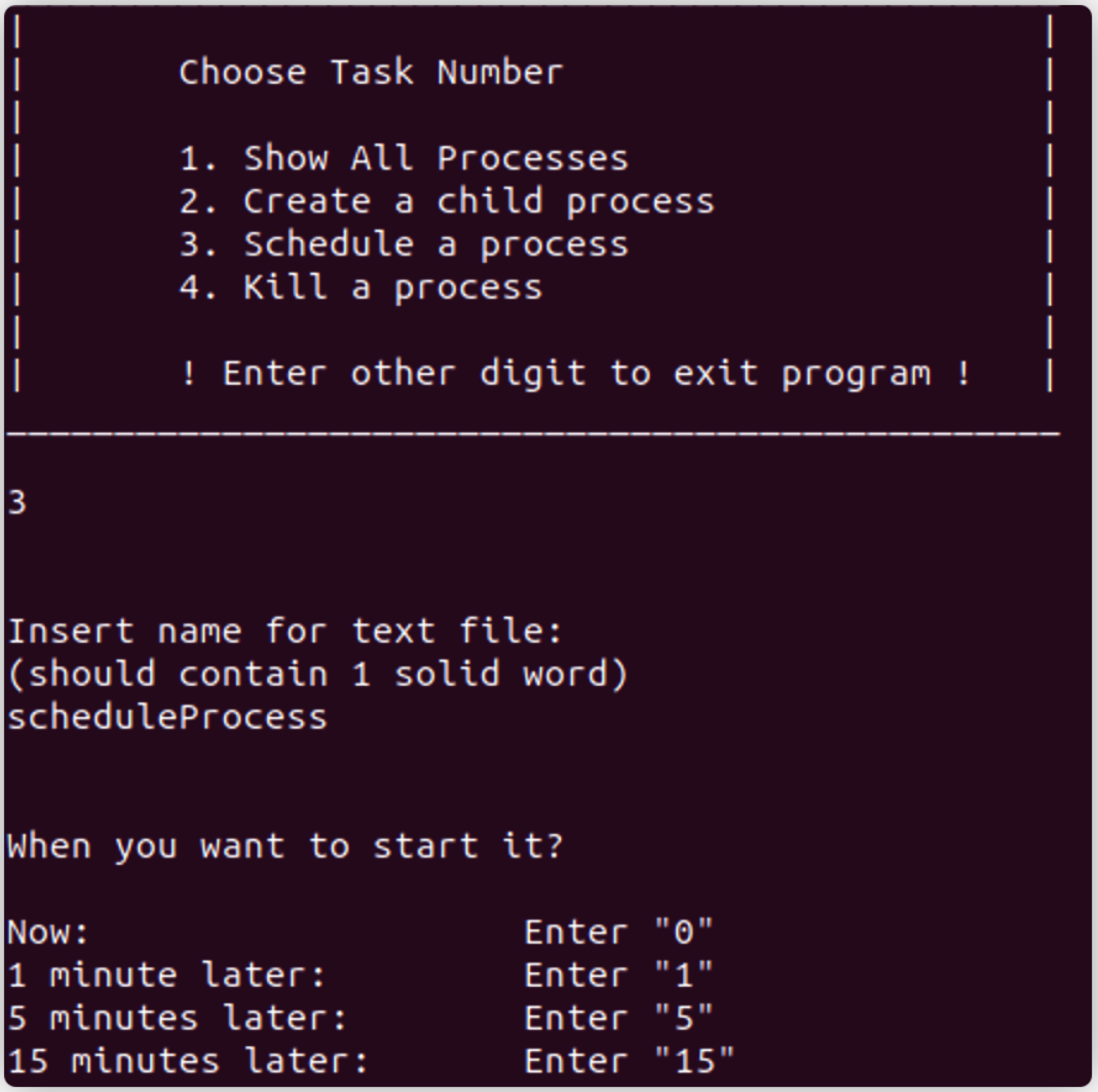


Figure 14. The interface of the third function(Schedule a process)

```c
void schedule() {
    char mins[5];
    char name[15];
    char command[60];

    printf("\n\n");
    printf("Insert name for text file:\n");
    printf("(should contain 1 solid word)\n");

    scanf("%s", name);

    strcat(command, "echo \"touch ");   // "echo "touch
    strcat(command, name);              // "echo "touch NAMEOFFILE
    strcat(command, " | date > ");      // "echo "touch NAMEOFFILE | date >
    strcat(command, name);              // "echo "touch NAMEOFFILE | date > NAMEOFFILE
    strcat(command, "\" | at now + ");  // "echo "touch NAMEOFFILE | date > NAMEOFFILE" | at now +

    printf("\n\n");
    printf("When you want to start it?");
    printf("\n\n");
    printf("Now:\t\t\tEnter \"0\"\n");
    printf("1 minute later:\t\tEnter \"1\"\n");
    printf("5 minutes later:\tEnter \"5\"\n");
    printf("15 minutes later:\tEnter \"15\"\n");

    scanf("%s", mins);

    strcat(command, mins);              // "echo "touch NAMEOFFILE | date > NAMEOFFILE" | at now + X
    strcat(command, " minutes");        // "echo "touch NAMEOFFILE | date > NAMEOFFILE" | at now + X minutes

    system(command);                    // system, schedule given task!

    printf("\n\nJob scheduled!\n");

}
```

Figure 15. The interface of the third function (Schedule a process-code part)

In C language which was used to create Programs #1 and #2, there is no such data type as string. Consequently, for storing information, command, or any other data one must use char arrays.

In this part of Program #2 the command-line utility “at” allows the user to schedule commands.Jobs created with “`at`” are being executed only once and can be given by the number of minutes after which the required task has to be performed or by the exact time on clock, like an alarm. By using a vertical bar, the user can perform many commands at once without writing them separately one by one. Combined with 'echo’ which performs commands written in quotes, “touch” and “at” it is possible to schedule 2 and more tasks at once. In Program #2 these commands are used to create a text file, if the directory has no text file with such name yet, insert date and time into it at a given time.

For working with “at” command the user has to install appropriate packages, as it is not installed by default. It can be done by typing the command “sudo apt-get install at”.

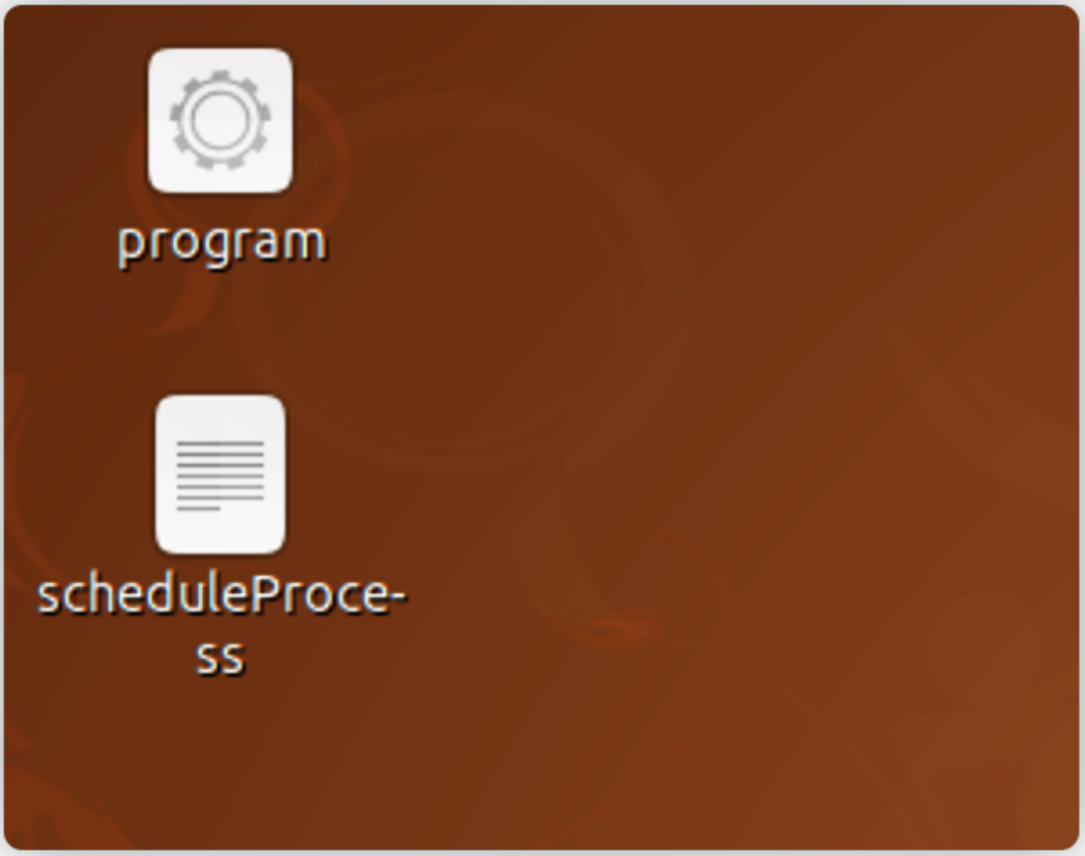


Figure 16. The result of schedule a process(locate on the desktop)

Figure 15 shows the code for the scheduling process for Program #1 below:
The system runs the program and asks the user to insert a name for the file, then offer four options for the user to select when the user enters ‘0’ means not wait to create a process immediately, enter ‘1’, ‘5’, ‘15’ means the Linux system will create a new process which called “scheduleProcess” after 1 minute, 5 minutes or 15 minutes on the desktop.

Process Destruction

During a process’s life cycle, it would have been created and would have run whatever job it was programmed to do. However, eventually, the process will terminate usually because of a few conditions. These conditions are normal exit(voluntary), error exit(voluntary), fatal error(involuntary), killed by another process(involuntary). Another reason a process may terminate is that there was an error either caused by the user or an error in the program such as a bug in the code. Finally, a process could be killed by another process given it has the necessary authorization.

When a compiler has compiled the program given to it, the compiler executes a system call to tell the operating system that it is finished.

The usual way for a process to terminate is to invoke the exit( ) library function, which releases the resources allocated by the C library, executes each function registered by the programmer, and ends up invoking a system call that evicts the process from the system. The exit( ) library function may be inserted by the programmer explicitly. Additionally, the C compiler always inserts an exit( ) function call right after the last statement of the main( ) function.

Alternatively, the kernel may force a whole thread group to die. This typically occurs when a process in the group has received a signal that it cannot handle or ignore or when an unrecoverable CPU exception has been raised in Kernel Mode while the kernel was running on behalf of the process.

When progress destruction occurs, the kernel must be notified so that it can release the resources owned by the process; this includes memory, open files, and any other odds and ends*(oreilly.com,2021*)

“process exit is driven, the process ends through a call to the kernel function do_exit”.

The following Figure 17 graphically indicates the “do_exit” process and the Function hierarchy for process destruction.

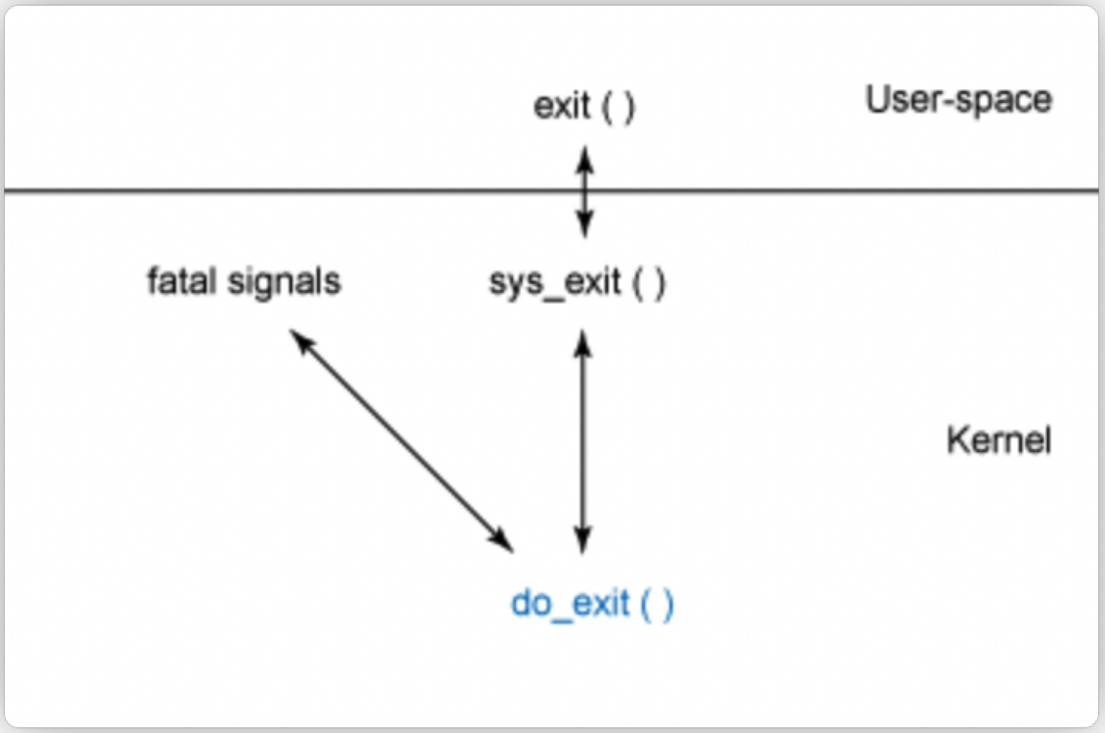


Figure 17. Function hierarchy for process destruction

“The purpose behind do_exit is to remove all references to the current process from the operating system (for all resources that are not shared)”(*developer.ibm.com, M. Jones,2008)*

The following Figures indicates the process destruction by another process

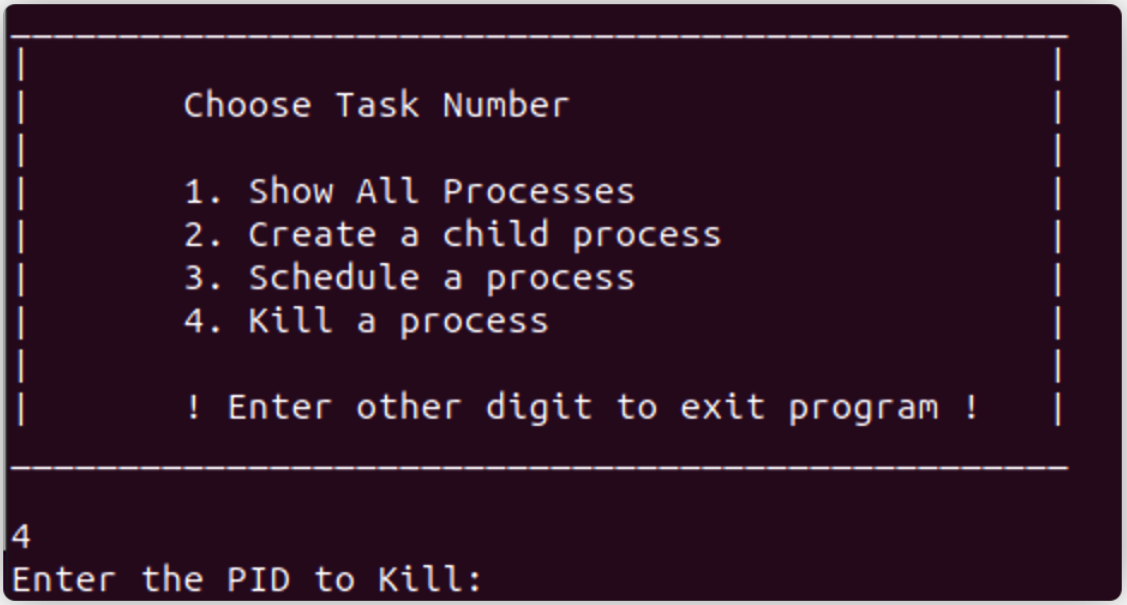


Figure 18. The interface of the fourth function(Kill a process)

```c
void killProcess() {
    char processPID[10];
    char getName[30];
    char kill[20];

    printf("Enter the PID to Kill: ");
    scanf("%s", processPID);        // gets PID

    strcat(getName, "ps -p ");      // ps -p
    strcat(getName, processPID);        // ps -p PID
    strcat(getName, " -o comm=");   // ps -p PID -o comm=       RETURNS NAME OF PROCESS BY GETTING ITS PID

    strcat(kill, "kill ");          // kill
    strcat(kill, processPID);       // kill PID
    system(kill);               // kills process by PID

    printf("\n\nThis program was killed!");

    system(getName);            // outputs name of killed process
}
```

Figure 19. The code part of the fourth function(Kill a process)

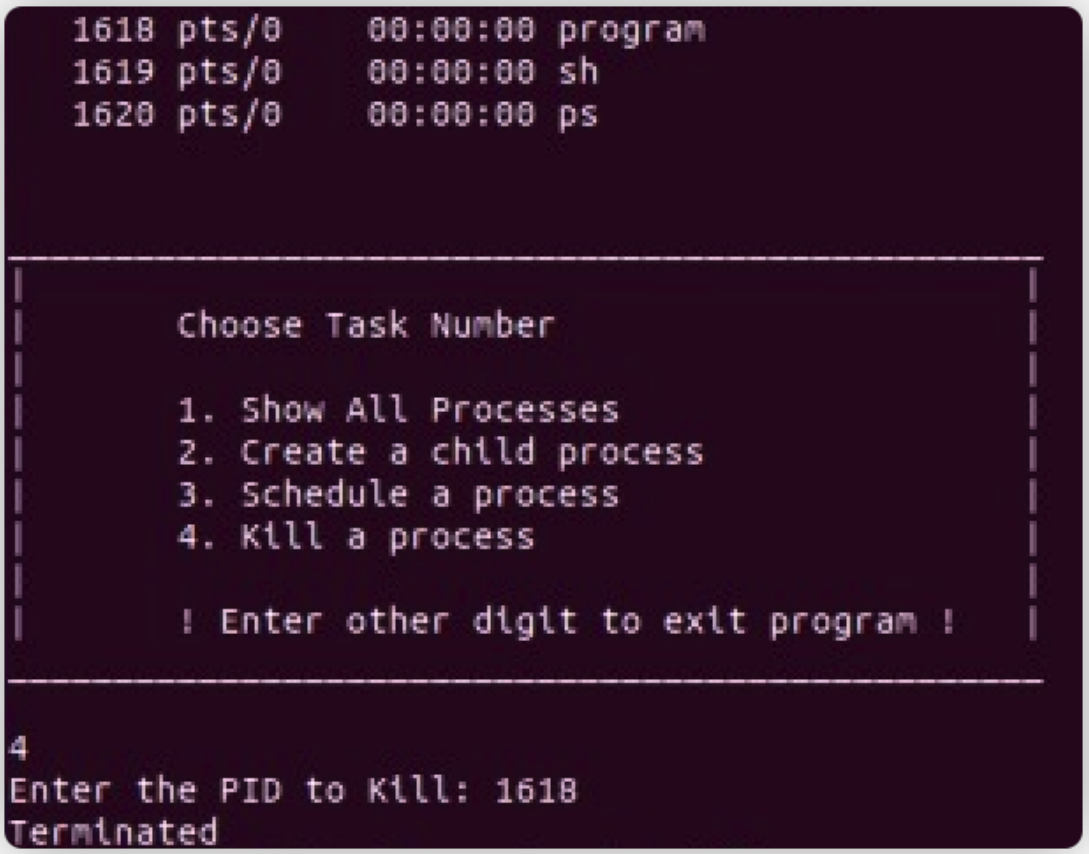


Figure 20. The result of the fourth function

The system needs the user to enter a PID to kill, the PID can be selected based on the result of the first function, the program receives the process ID and compares it with the database after finding the user assigned process id, the program will execute the kill command.

Code for Program #2

```c
#include <stdio.h>                  // library for basic input output on C language
#include <stdlib.h>                 // for using "system" command
#include <stdbool.h>                // for using bool variable, false/true declares
#include <string.h>                 // now we can add 2 strings
#include <unistd.h>                 // for cloning processes, getting PID and PPID
#include <sys/wait.h>               // for scheduling tasks
#include <pthread.h>                // for creating and playing with threads

int sharedInteger = 100;            // shared variable
int counterForDeadlock = 0;         // will be used in task 3


pthread_mutex_t MutexThread;        // creating mutex for locking threads || This thread is for demo of task 1, 2
pthread_mutex_t MutexForDeadlock1;  // will be used for Deadlock & Starvation
pthread_mutex_t MutexForDeadlock2;  // will be used for Deadlock & Starvation

/// Method Name: Multiply
/// Purpose: To multiply shared digit by 10 -> sleep -> save changed result
void* Multiply()
{
    int localValue;

    printf("\n\nThread #1 starts its work...\n");

    localValue = sharedInteger;
    printf("Thread #1 takes value of shared variable: %d\n", localValue);

    localValue *= 10;

    printf("Thread #1 updated local variable: %d\n", localValue);

    printf("Thread #1 goes to sleep...\n\n");
    sleep(0.5);                     //Sends to sleep for 0.5 s Thread 1, now Thread 2 will continue
    printf("Thread #1 resumes its work...\n");

    sharedInteger = localValue;
    printf("Thread #1 finished work and updated shared integer: %d\n\n", sharedInteger);
}
```

Figure 21. Code part of Program #2 which is written in “C” for section 2.

```c
/// Method Name: Divide
/// Purpose: To divide shared digit by 10 -> sleep -> save changed result
void* Divide()
{
    int localValue;

    printf("Thread #2 starts its work...\n");

    localValue = sharedInteger;
    printf("Thread #2 takes value of shared variable: %d\n", localValue);

    localValue /= 10;

    printf("Thread #2 updated local variable: %d\n", localValue);

    printf("Thread #2 goes to sleep...\n\n\n");
    sleep(0.5);                    //Sends to sleep for 0.5 s Thread 1, now Thread 2 will continue
    printf("Thread #2 resumes its work...\n");

    sharedInteger = localValue;
    printf("Thread #2 finished work and updated shared integer: %d\n\n", sharedInteger);
}

/// Method Name: MultiplyMutex
/// Purpose: To multiply shared digit by 10, but WITH THE USE OF MUTEX
void* MultiplyMutex()
{
    pthread_mutex_lock(&MutexThread); //lock this thread

    int localValue;

    printf("\n\nThread #1 starts its work...\n");

    localValue = sharedInteger;
    printf("Thread #1 takes value of shared variable: %d\n", localValue);

    localValue *= 10;

    printf("Thread #1 updated local variable: %d\n", localValue);

    printf("Thread #1 goes to sleep...\n\n");
    sleep(0.5);                     //Sends to sleep for 0.5 s Thread 1, now Thread 2 will continue
    printf("Thread #1 resumes its work...\n");
```

Figure 22. Program #2 code parts for section 2 are written in c.

```c
    sharedInteger = localValue;
    printf("Thread #1 finished work and updated shared integer: %d\n\n", sharedInteger);

    pthread_mutex_unlock(&MutexThread); //unlock this thread
}

/// Method Name: DivideMutex
/// Purpose: To divide shared digit by 10, but WITH THE USE OF MUTEX
void* DivideMutex()
{
    pthread_mutex_lock(&MutexThread);

    int localValue;

    printf("Thread #2 starts its work...\n");

    localValue = sharedInteger;
    printf("Thread #2 takes value of shared variable: %d\n", localValue);

    localValue /= 10;

    printf("Thread #2 updated local variable: %d\n", localValue);

    printf("Thread #2 goes to sleep...\n\n\n");
    sleep(0.5);  //Sends to sleep for 0.5 s Thread 1, now Thread 2 will continue
    printf("Thread #2 resumes its work...\n");

    sharedInteger = localValue;
    printf("Thread #2 finished work and updated shared integer: %d\n\n", sharedInteger);

    pthread_mutex_unlock(&MutexThread); //unlock this thread
}

/// Method Name: Race
/// Purpose: To demonstrate race condition | critical section Problem
/// How it works: Runs Multiply and Divide methods by order to demo wrong final result (100*10/10=100, but here you will get 10 OR 1000)
```

Figure 23. Program #2 code parts for section 2 is written in c.

```c
/// Method Name: Race
/// Purpose: To demonstrate race condition | critical section Problem
/// How it works: Runs Multiply and Divide methods by order to demo wrong final result (100*10/10=100, but here you will get 10 OR 1000)
void Race()
{
    pthread_t thread1, thread2; //pthread_t - data type to uniquely identify a thread

    pthread_create(&thread1, NULL, Multiply, NULL);  //  thread creation (thread | attribute | name of Method | argument).
                                                     //It creates thread and assigns to it chosen method
    pthread_create(&thread2, NULL, Divide, NULL);

    pthread_join(thread1, NULL); //wait for thread termination, then can be used by other processes
    pthread_join(thread2, NULL);

    printf("Final Value of shared integer is %d\n", sharedInteger); //prints the last updated value of shared variable
}

/// Method Name: Mutex
/// Purpose: To demonstrate Critical Section problem soluton with the use of MUTEX
/// How it works: Runs MultiplyMutex and DivideMutex methods by order to demo right final result (100*10/10=100)
void Mutex()
{
    pthread_t thread1, thread2; //pthread_t - data type to uniquely identify a thread

    pthread_mutex_init(&MutexForDeadlock1, NULL);
    pthread_mutex_init(&MutexForDeadlock2, NULL);

    pthread_create(&thread1, NULL, MultiplyMutex, NULL);  //  thread creation (thread | attribute | name of Method | argument).
                                                          //It creates thread and assigns to it chosen method
    pthread_create(&thread2, NULL, DivideMutex, NULL);

    pthread_join(thread1, NULL);                        //wait for thread termination, then can be used by other processes
    pthread_join(thread2, NULL);

    printf("Final Value of shared integer is %d\n", sharedInteger); //prints the last updated value of shared variable
}
```

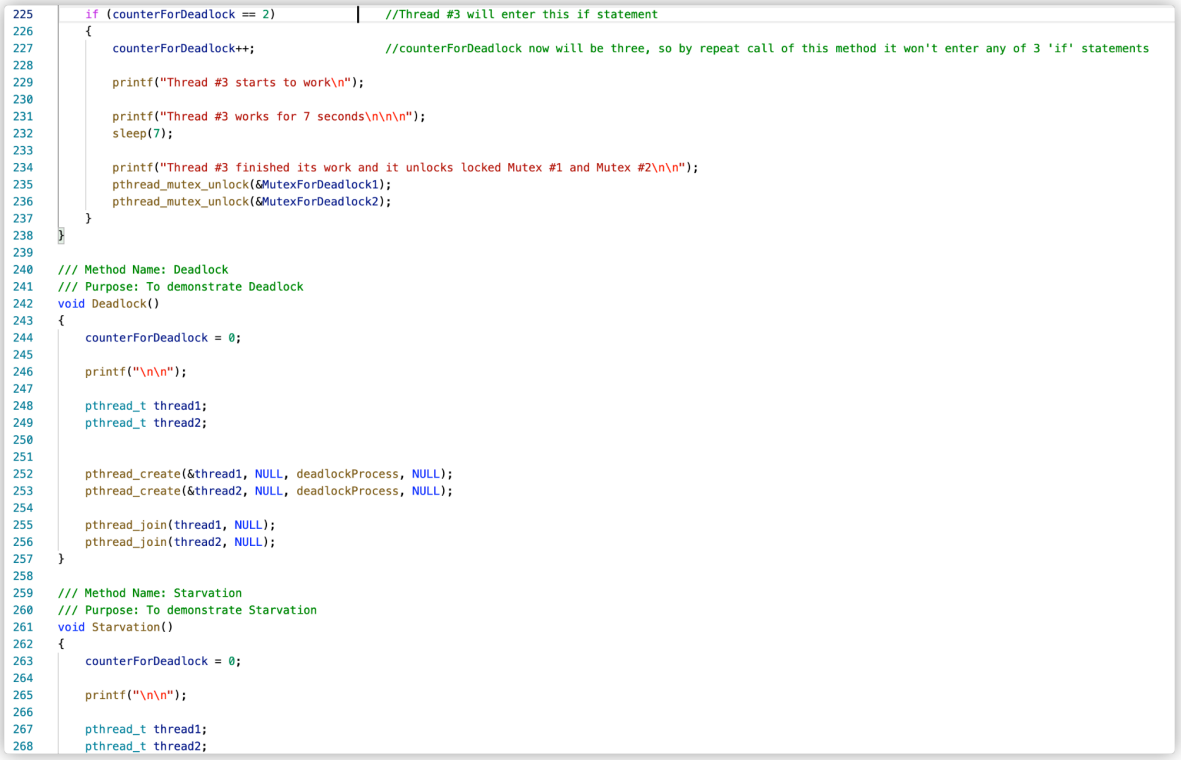
Figure 24. Program #2 code parts for section 2 are written in c.

```c
/// Method Name: deadlockProcess
/// Purpose: To demonstrate Deadlock example with the use of MUTEX
/// How it works: Thread #1 locks Mutex #1, Thread #2 locks Mutex #2. Then Thread #1 and #2 wait each other to lock each other's Mutex (#1 or #2)
void* deadlockProcess() {

    if (counterForDeadlock == 0)
    {
        printf("Thread #1 starts to work and locks Mutex #1\n");
        pthread_mutex_lock(&MutexForDeadlock1);

        counterForDeadlock++;                          //now next thread will go to another if statement (which is below)

        printf("Thread #1 goes to sleep with locked Mutex #1...\n\n\n");
        sleep(1);

        printf("Thread #1 resumes its work and waits for access to Mutex #2...\n");
        pthread_mutex_lock(&MutexForDeadlock2);
    }
    if (counterForDeadlock == 1)
    {
        printf("Thread #2 starts to work and locks Mutex #2\n");
        pthread_mutex_lock(&MutexForDeadlock2);

        printf("Thread #2 goes to sleep with locked Mutex #2...\n\n\n");
        sleep(1);

        printf("Thread #2 resumes its work and waits for access to Mutex #1...\n");
        pthread_mutex_lock(&MutexForDeadlock1);
    }
}
```

Figure 25. Program #2 code parts for section 2 are written in c.

```c
/// Method Name: starvationProcess
/// Purpose: To demonstrate Starvation example with the use of MUTEX
/// How it works: Same steps as in deadlockProcess method, but there is Thread #3, which will unlock Mutex #1 and #2 -> threads will finish their work.
void* starvationProcess() {

    if (counterForDeadlock == 0)                   //Thread #1 will enter this if statement
    {
        counterForDeadlock++;                      //next thread will go to another if statement (which is below)

        printf("Thread #1 starts to work and locks Mutex #1\n");
        pthread_mutex_lock(&MutexForDeadlock1);

        printf("Thread #1 goes to sleep with locked Mutex #1...\n\n\n");
        sleep(1);

        printf("Thread #1 resumes its work and waits for access to Mutex #2...\n\n\n");
        pthread_mutex_lock(&MutexForDeadlock2);

        printf("Thread #1 finally accessed to Mutex #2\n");
        printf("Thread #1 finished its work...\n\n\n");
        pthread_mutex_unlock(&MutexForDeadlock2);
    }
    if (counterForDeadlock == 1)                   //Thread #2 will enter this if statement
    {
        counterForDeadlock++;                      //next thread will go to another if statement (which is below)

        printf("Thread #2 starts to work and locks Mutex #2\n");
        pthread_mutex_lock(&MutexForDeadlock2);

        printf("Thread #2 goes to sleep with locked Mutex #2...\n\n\n");
        sleep(1);

        printf("Thread #2 resumes its work and waits for access to Mutex #1...\n\n");
        pthread_mutex_lock(&MutexForDeadlock1);


        printf("Thread #2 finally accessed to Mutex #1\n");
        printf("Thread #2 finished its work...\n\n\n");
        pthread_mutex_unlock(&MutexForDeadlock1);
    }
```

Figure 26. Program #2 code parts for section 2 are written in c.



```c
    if (counterForDeadlock == 2)                //Thread #3 will enter this if statement
    {
        counterForDeadlock++;                   //counterForDeadlock now will be three, so by repeat call of this method it won't enter any of 3 'if' statements

        printf("Thread #3 starts to work\n");

        printf("Thread #3 works for 7 seconds\n\n\n");
        sleep(7);

        printf("Thread #3 finished its work and it unlocks locked Mutex #1 and Mutex #2\n\n");
        pthread_mutex_unlock(&MutexForDeadlock1);
        pthread_mutex_unlock(&MutexForDeadlock2);
    }
}

/// Method Name: Deadlock
/// Purpose: To demonstrate Deadlock
void Deadlock()
{
    counterForDeadlock = 0;

    printf("\n\n");

    pthread_t thread1;
    pthread_t thread2;


    pthread_create(&thread1, NULL, deadlockProcess, NULL);
    pthread_create(&thread2, NULL, deadlockProcess, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
}

/// Method Name: Starvation
/// Purpose: To demonstrate Starvation
void Starvation()
{
    counterForDeadlock = 0;

    printf("\n\n");

    pthread_t thread1;
    pthread_t thread2;
```

Figure 27. Program #2 code parts for section 2 are written in c.

```
    pthread_t thread1;
    pthread_t thread2;
    pthread_t thread3;


    pthread_create(&thread1, NULL, starvationProcess, NULL);
    pthread_create(&thread2, NULL, starvationProcess, NULL);
    pthread_create(&thread3, NULL, starvationProcess, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);
}

/// Method Name: Main
/// Purpose: Main method with all Interactive Menu
int main()
{
    bool menu = true;
    int option = 0;

    while (menu) {                  //Starting of User Interface for Main Menu

        printf("\n\n");
        printf("_____________________________________________________\n");
        printf("|\t\t\t\t\t\t\t\t|\n");
        printf("|\tChoose Task Number\t\t\t\t|\n");
        printf("|\t\t\t\t\t\t\t\t|\n");
        printf("|\t1. Race Condition | Critical Section Problem\t|\n");
        printf("|\t2. Critical Section Solution | Mutex Lock\t|\n");
        printf("|\t3. Deadlock Example\t\t\t\t|\n");
        printf("|\t4. Starvation Example\t\t\t\t|\n");
        printf("|\t\t\t\t\t\t\t\t|\n");
        printf("|\t! Enter other digit to exit program !\t\t|\n");
        printf("_____________________________________________________\n\n");

        scanf("%d", &option);       //Choose option from menu

        switch (option)
        {
        default:                    // case which executes when "option" value is not 1 to 4
            menu = false;
            printf("Program Exiting...\n");
            break;
```

Figure 28. Program #2 code parts for section 2 are written in c.

```
288     while (menu) {                  //Starting of User Interface for Main Menu
289
290         printf("\n\n");
291         printf("_____________________________________________________________\n");
292         printf("|\t\t\t\t\t\t\t|\n");
293         printf("|\tChoose Task Number\t\t\t\t|\n");
294         printf("|\t\t\t\t\t\t\t|\n");
295         printf("|\t1. Race Condition | Critical Section Problem\t|\n");
296         printf("|\t2. Critical Section Solution | Mutex Lock\t|\n");
297         printf("|\t3. Deadlock Example\t\t\t\t|\n");
298         printf("|\t4. Starvation Example\t\t\t\t|\n");
299         printf("|\t\t\t\t\t\t\t|\n");
300         printf("|\t! Enter other digit to exit program !\t\t|\n");
301         printf("_____________________________________________________________\n\n");
302
303         scanf("%d", &option);       //Choose option from menu
304
305         switch (option)
306         {
307         default:                    // case which executes when "option" value is not 1 to 4
308             menu = false;
309             printf("Program Exiting...\n");
310             break;
311
312         case 1:
313             Race();
314             break;
315
316         case 2:
317             Mutex();
318             break;
319
320         case 3:
321             Deadlock();
322             break;
323
324         case 4:
325             Starvation();
326             break;
327         }
328     }
329 }
```

Figure 29. Program #2 code parts for section 2 are written in c.

Below Figure 30 shows the menu interface of function two.

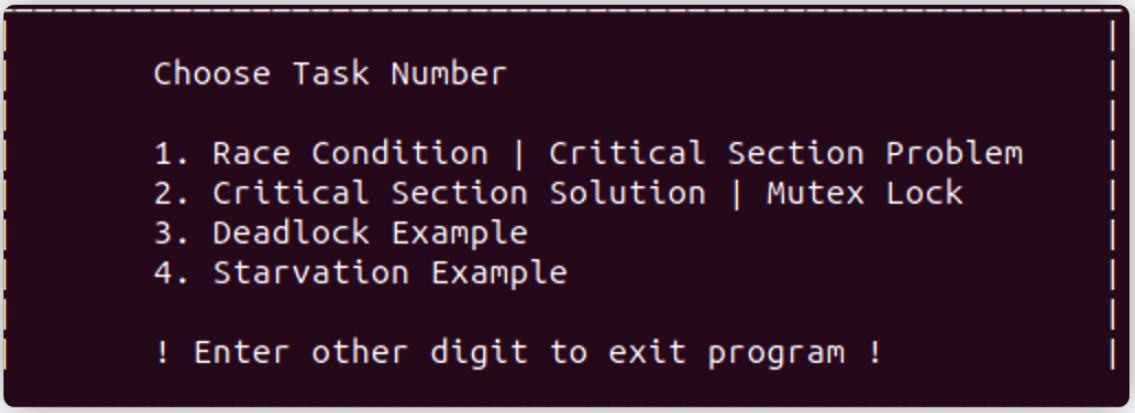


Figure 30 the interface of the Program #2 menu part for section 2.

For section two includes 4 functions, users can enter a relevant digit to select the function.

Section 2

Race Condition

Since Linux is a multitasking Operating system, many processes are running, reading, writing data, and working together. It is possible that they share some common storage. Due to multi-core CPU's and multithreading processes can execute concurrently or in parallel. With processes running at the same time there is a chance that they need access to the same data at the same time. If this happens the outcome of the execution depends on which order the processes get access this situation is defined as a race condition. When the two processes access the data, they both change the pointer address to the same spot and this causes one process to be blocked out. To stop this condition from happening we need to try to only let one process at a time have access to the same block of data. This is where mutual exclusion comes in, this is the way to make sure only one process has access to a shared variable or file at a time. When a process is accessing shared memory it is using its critical region or critical section, we need to have a way of not letting any two processes from accessing their critical region at the same time. Process synchronization is used to solve the critical-selection problem so processes may share data independently. Here each process must request permission to enter its critical section. If it is being used by another process the request is denied. If access to shared memory is not properly synchronized, race conditions can occur. If a programmer is trying to debug a program with race conditions it can cause a lot of grief and could confuse them to no end. Since the tests that are being run on the

program could pass ninety percent of the time but occasionally when the program's processes or threads need to access the same piece of memory at the same time causing a data race which could lead to memory corruption or undefined behavior. Race conditions could have affected the security of software as well if an attacker has access to shared resources could have another attacker use that resource simultaneously to have it malfunction and contribute to a denial of service attack or to change privileges. (Wikipedia.org, 2021). A real-world example similar to race conditions would be if two trains headed in opposite directions needed to use one track at a specific stretch because of a narrow pass. If both trains tried to gain access to the track at the same time it could cause a disaster. That is why train signaling is very important in the railroad industry. Similarly in computing, a signal can be sent saying a process is accessing the memory and the other process will have to wait. With computers increasing cores and threads software developers must be diligent in creating mutual exclusivity in processes that share memory.

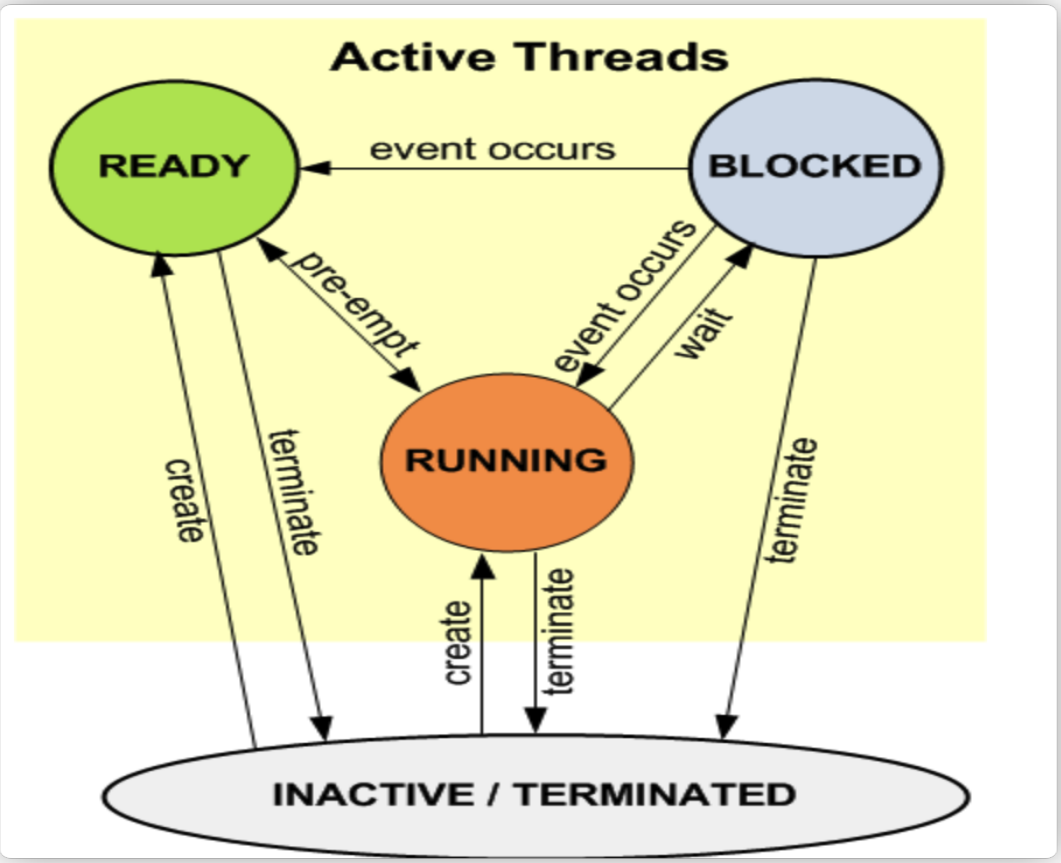


Figure 31. Thread states diagram

Threads can be divided into the following states:

1. Running: the thread which is currently running, one time only can run one thread.
2. Ready: the thread that is in a ready-to-run situation, once the running thread has terminated or is blocked, the next ready thread with the highest priority becomes the running thread.

3. Blocked: the thread that is in a blocked or delayed situation, waiting for an event to occur, or suspended is in the blocked state.
4. Terminated: threads are terminated with resources not yet released.
5. Inactive: the thread that is not created or has been terminated with all resources released.

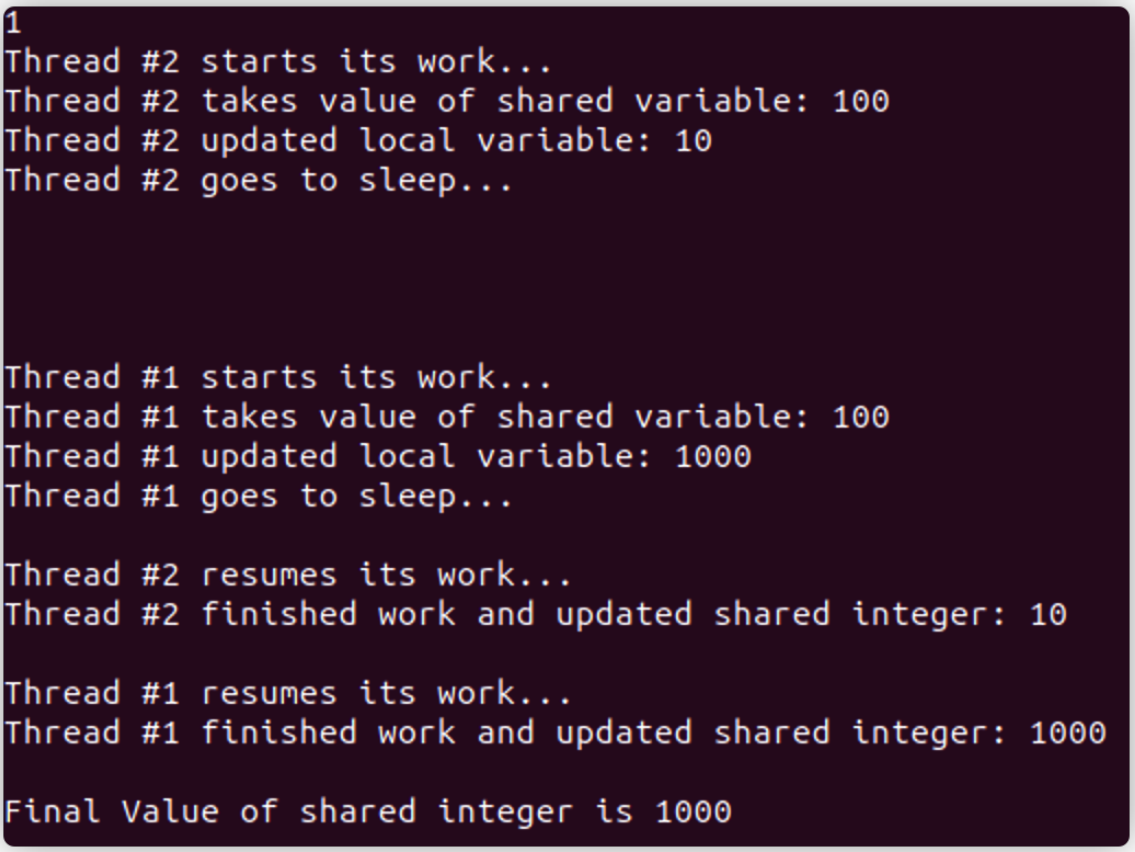


Figure 32. Race condition function 1 result

An example of race condition in Program #2 which accompanies this report is as follows: As the default shared integer is 100, thread 2 starts to work and the system call divide method, the shared integer = 100 will divide 10 and the result will be 10, however, the result will not immediately be stored in the shared memory, but go to sleep for 0.5s, as the following process development, when thread1 start to work then the system call multiply method 100*10 =1000, then thread 2 goes to sleep, and thread 1 will resume and update the shared integer as 1000 after thread 1 finishes the update, the thread starts to update the shared integer as 1000.

```c
void Race()
{
    pthread_t thread1, thread2; //pthread_t - data type to uniquely identify a thread

    pthread_create(&thread1, NULL, Multiply, NULL);  //  thread creation (thread | attribute | name of Method | argument).
                                                     //It creates thread and assigns to it chosen method
    pthread_create(&thread2, NULL, Divide, NULL);

    pthread_join(thread1, NULL); //wait for thread termination, then can be used by other processes
    pthread_join(thread2, NULL);

    printf("Final Value of shared integer is %d\n", sharedInteger); //prints the last updated value of shared variable
}
```

Figure 33. Race condition function relevant code part.

The pthread_create() function is used to create a new thread, with attributes specified by attr, within a process. Create two threads, name thread1, and thread2 by using the pthread_create() function. *(pubs.opengroup.org,1997)*

The pthread_join() function suspends execution of the calling thread until the target *thread* terminates, unless the target *thread* has already terminate, using pthread_join(thread1, null) and pthread_jonin(thread2, null) function to resume thread1 and thread 2 go back work. (pubs.opengroup.org, 1997)

```c
17  /// Method Name: Multiply
18  /// Purpose: To multiply shared digit by 10 -> sleep -> save changed result
19  void* Multiply()
20  {
21      int localValue;
22
23      printf("\n\nThread #1 starts its work...\n");
24
25      localValue = sharedInteger;
26      printf("Thread #1 takes value of shared variable: %d\n", localValue);
27
28      localValue *= 10;
29
30      printf("Thread #1 updated local variable: %d\n", localValue);
31
32      printf("Thread #1 goes to sleep...\n\n");
33      sleep(0.5);                     //Sends to sleep for 0.5 s Thread 1, now Thread 2 will continue
34      printf("Thread #1 resumes its work...\n");
35
36      sharedInteger = localValue;
37      printf("Thread #1 finished work and updated shared integer: %d\n\n", sharedInteger);
38  }
```

Figure 34. Race condition Multiply method code part

The multiply method, as the sharedInteger is a global value equal to 100, then localValue equals sharedInteger, thread #1 take localValue multiply 10 equal 1000, then thread #1 go to sleep for 0.5 seconds.

```c
void* Divide()
{
    int localValue;

    printf("Thread #2 starts its work...\n");

    localValue = sharedInteger;
    printf("Thread #2 takes value of shared variable: %d\n", localValue);

    localValue /= 10;

    printf("Thread #2 updated local variable: %d\n", localValue);

    printf("Thread #2 goes to sleep...\n\n\n");
    sleep(0.5);                    //Sends to sleep for 0.5 s Thread 1, now Thread 2 will continue
    printf("Thread #2 resumes its work...\n");

    sharedInteger = localValue;
    printf("Thread #2 finished work and updated shared integer: %d\n\n", sharedInteger);
}
```

Figure 35. Race condition Divide method code part

In the Divide method, thread #2 take the default sharedInteger and give it to localValue, then divide the localValue with 10, get 10 as a result, then go to sleep for 0.5 seconds.
Now, thread #2 resume to work and update localValue to the global parameter sharedInteger then unlock the shared memory, then thread #1 resume to work and also update the localValue to the global parameter sharedInteger.Int the last part, output the sharedInteger, because the sharedInteger be updated twice, the eventual output result will be 1000, which is the multiply method result, the divide method updated result will be covered by multiple method result.

Mutex Locks

Mutex locks are a tool used by software developers to prevent race conditions and to solve the critical region problem we will be discussing later in this report. There are hardware solutions but they are very complex and cannot be used by operating systems designers. The simplest high-level software tool is the mutex lock. Mutex is short for mutual exclusion, mutual exclusion is made possible by Mutex Locks by ensuring only one process or thread gets access to shared data. A mutex lock protects the critical regions and prevents race conditions A mutex is typically acquired and released around the code that accesses the shared data (usually a critical section). Only one process may have the mutex locked at any time. Any process trying to access a locked mutex will be prohibited until the process that is using the mutex will unlock the lock. Then the next process with the highest priority will have access to the lock and will be able to access their critical section and access shared data. This allows processes to sequence through their critical section in priority order. (Qnx.com, 2021). In the C program that accompanies this document pthread_mutex_lock() is the function we use to acquire the Mutex, this makes it impossible for any other process to run. After the process has finished completing its tasks, the pthread_mutex_unlock() function is used to unlock the Mutex and allow another process to acquire the Mutex and run its tasks.

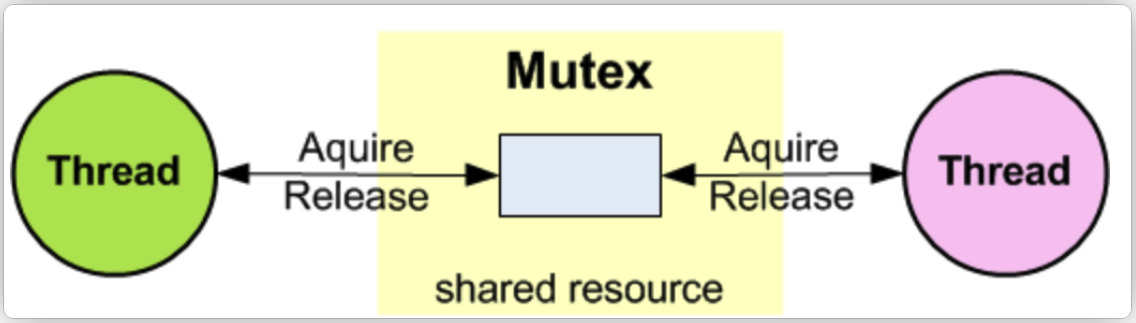


Figure 36. (*keli.com,2021*)

The advantage of a mutex is that it introduces thread ownership. When a thread acquires a mutex and becomes its owner, subsequent mutex acquired from that thread will succeed immediately without any latency. Thus, mutex acquisitions/releases can be nested.

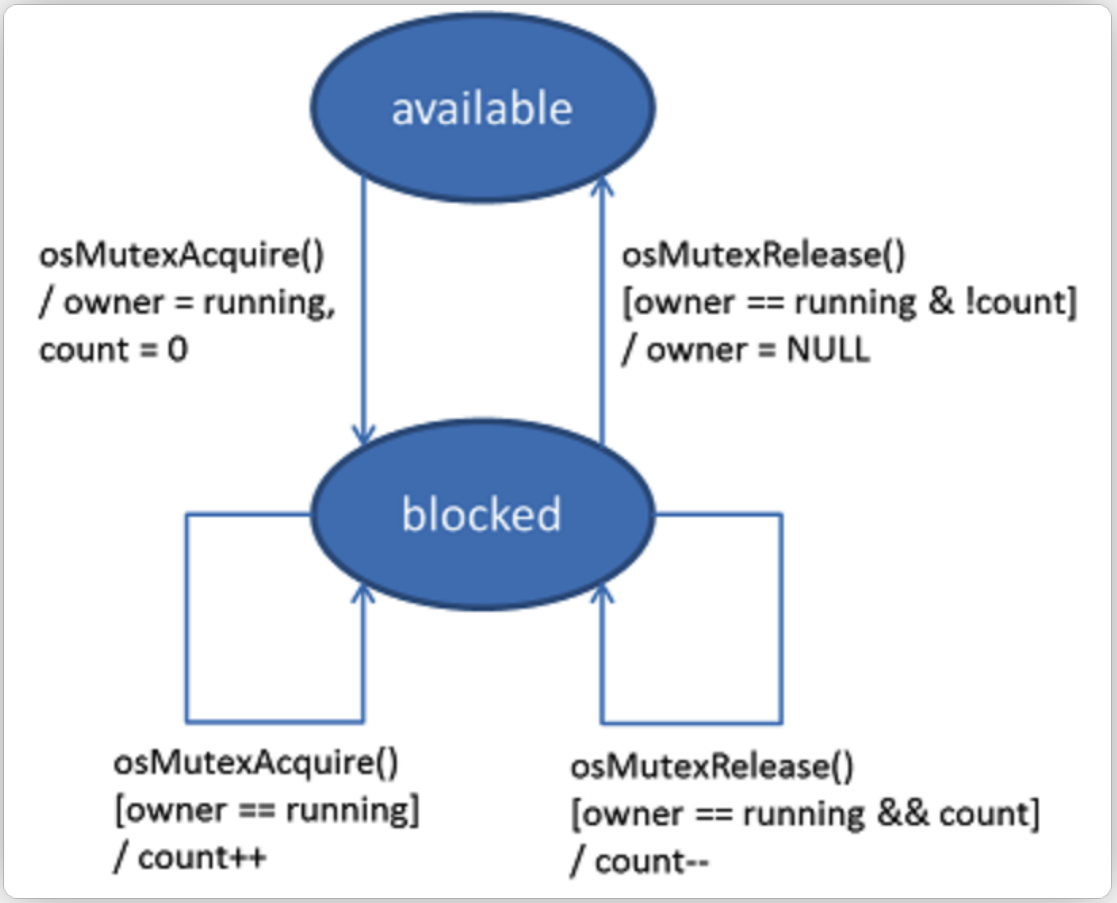


Figure 37. a mutex lock operating diagram. (*keli.com,2021)*

Critical Section Problem

As mentioned earlier race conditions give us the critical-section problem. Critical Section Problem is a piece of code inside a process that wants access to shared resources and that need not be executed while another process is in a corresponding section of code. This section of code is called the critical region or the critical section, and a fundamental part of the operating system working properly is that no two processes may be executing code simultaneously in their critical section. The section handling the request to enter the critical section is the entry section. The section after the critical section is the exit section and the code that is left is called the remainder section. This would give users a race condition and would cause an error. If the system was single-core and single-threaded there would be no critical section problem because only one process or thread would be running at one time but due to multicore and multi-threaded systems, it is a constant concern for OS developers.

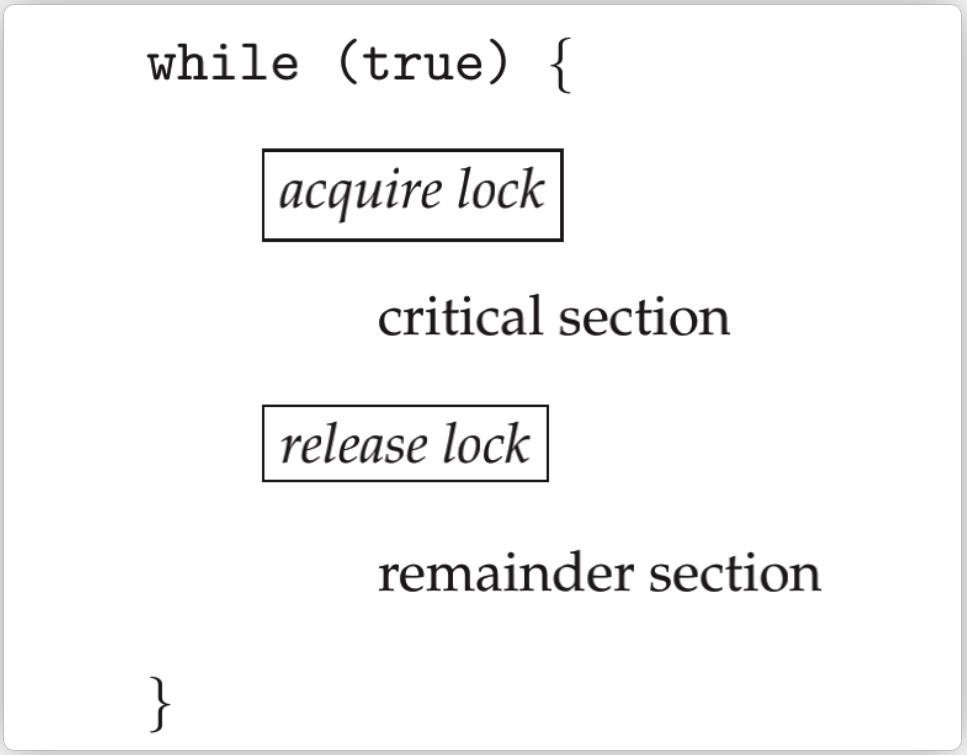


Figure 38. Using a mutex lock can solve the critical section problem.*(ANDREW S. TANENBAUM HERBERT BOS,2015,p271)*

The definition of acquire() is: when resources are not available the process waits.

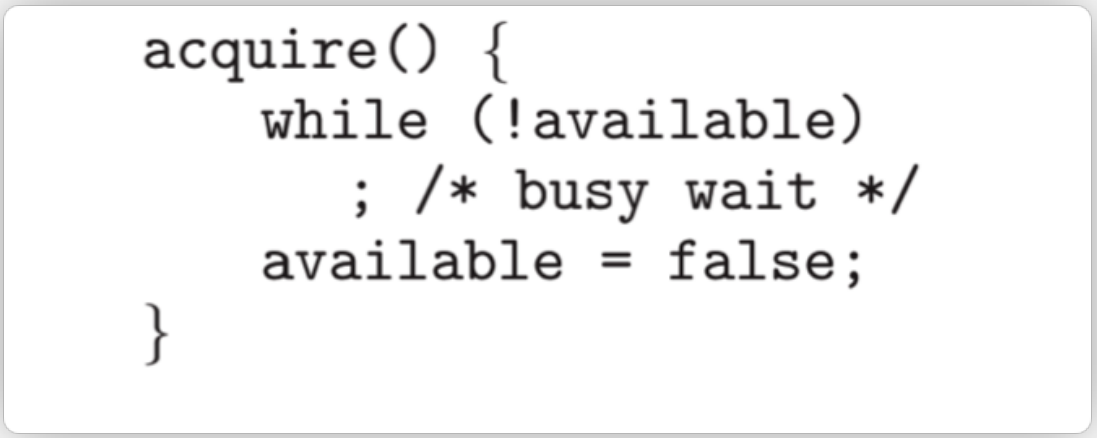


Figure 39. The algorithm of the process acquires the resource.*(Andrew S. Tanebaum Herbert Bos,2015,p271)*

The definition of release() is: when resources can be released then the resources will be available for other process use.

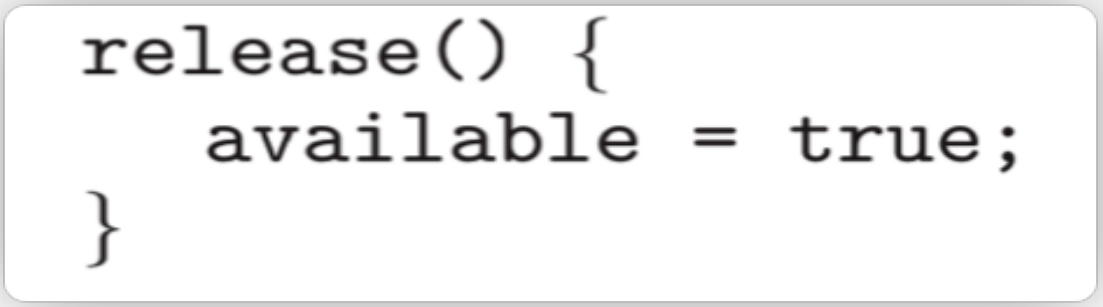


Figure 40. The algorithm of the process releases the resource.*(Andrew S. Tanebaum Herbert Bos,2015,p271).*

Lock contention

"Locks are either contented or uncontended". When a thread is in a locked situation then this situation can be called the contended situation, in another opposite situation, if a thread acquires to lock and successfully gets locked, then this situation can be named uncontended situation.

In addition, contended locks have high and low characteristics, for example, when a large number of threads acquire the lock or a few threads try to acquire the lock corresponding to the above two characteristics.

Solutions to Critical Section Problems

The critical section problem needs a solution to synchronize the different processes, synchronisation is the key in solving the critical section problem. The solution to the critical section problem must satisfy the following conditions: Bounded Waiting, Progress and Mutual Exclusion. As mentioned earlier mutual exclusion implies that only one process can be inside the critical section at any time. If any other processes require the critical section, they must wait until it is free. Bounded waiting means that each process must have a limited waiting time. It should not wait forever to access the critical section. Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free. As mentioned before mutual exclusion means that only one process can be inside the critical section at any time. If any other processes require the critical section, they must wait until it is free. (Tutorialpoint.com, 2021). We shall investigate ways of achieving mutual exclusion to solve critical-section problems. In operating system design two main ways to solve the critical section problem, building and implementing a preemptive kernel and building and implementing a non-preemptive kernel. A preemptive kernel lets a process be stopped to allow another process to run. The non-preemptive kernel does not allow an interrupt to occur on the process currently running, the process will run until completion. Since non-preemptive kernels allow a process to run for long periods of time they are not as responsive as OS with preemptive kernels. An example of an operating system with a non-preemptive kernel is Windows XP. As for designing kernels, it is much easier to design non-preemptive kernels rather than preemptive kernels. One solution is to disable interrupts, if we had a single-processor system we could have each process disable all interrupts just after entering its critical section and re-enable them just before leaving it. Peterson's solution is a classic software solution to the critical section problem. One problem with the Peterson solution is it is restricted to two processes that alternate execution between their critical regions and remainder sections. The algorithm uses two variables: a Boolean array with 2 indices and an integer name flag which is either a zero or a one. There is no state that can satisfy both turn =0 and turn =1 so there could be a state where both processes are in their critical region simultaneously. Since Peterson’s solution is restricted to two processes and modern computers are multicore and run multi-threaded processes the Peterson’s solution is not guaranteed to work on modern computer architectures. As discussed earlier in this document mutex locks are a solution to the critical section problem if you have a mutex And it is locked no other process can enter. This makes it impossible for any other process to access the lock and change any shared memory. Semaphores are another solution to the critical section problem which uses only two operations: wait() and signal(). These operations are executed atomically meaning when one operation is being changed no other process can simultaneously change it. The wait and signal operation are similar to street lights for wait() the process must wait and for signal() gives the process permission to execute. When a process wants to use a semaphore it calls the wait() function which decrements the count and when the process finishes using its shared memory it calls the signal function which increments the count. When the count is zero all resources are being used and all processes are blocked. When the process leaves and increments the count it is free for another process to call wait() decrement the count back to zero and block all other processes solving the critical section problem. The compare and swap algorithm is another attempt to solve the critical section problem. It is a hardware solution that Introduces atomic

variables as referenced previously so that these variables cannot be changed so that they may be used with semaphores to ensure mutual exclusion, ensure progress and stop bounded waiting.

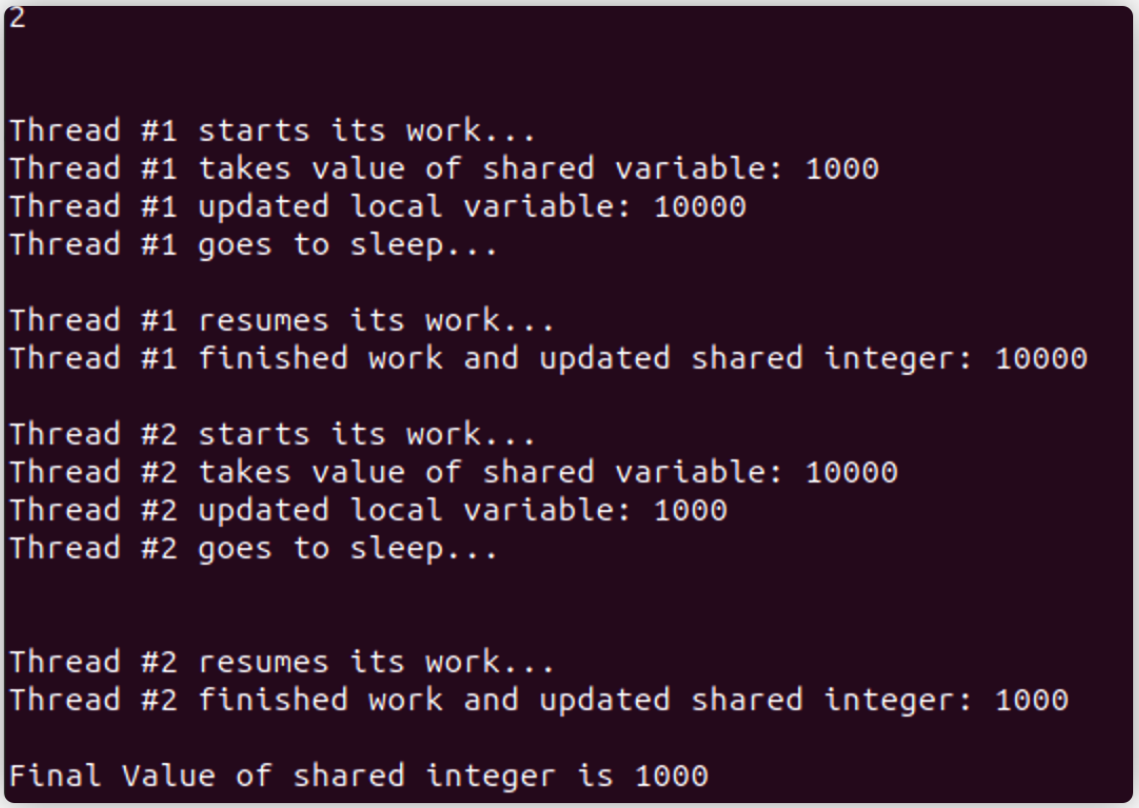


Figure 41. Mutex lock function result interface.

An example of a Mutex-lock in Program #2 which accompanies this report is as follows: Thread #1 starts to work in the terminal and the shared memory also be locked by mutex lock in the meantime, which means, only thread #1 finished work and updates the shared integer then thread #2 can start to work. Thread #1 calls the multiply method and takes the shared value with 1000 after multiplying 10 to get 10000 as result, then thread #1 goes to sleep for 0.5 seconds and then resumes work to update the result to the shared integer as 10000. Thread #2 join to work and call divide method, thread #2 taking the last updated shared integer with 10000 and dividing 10 to get 1000 as the last result, finally, thread #2 update the result to the shared integer with 1000, then the interface prints the shared value.

```
137  void Mutex()
138  {
139      pthread_t thread1, thread2; //pthread_t - data type to uniquely identify a thread
140
141      pthread_mutex_init(&MutexForDeadlock1, NULL);
142      pthread_mutex_init(&MutexForDeadlock2, NULL);
143
144      pthread_create(&thread1, NULL, MultiplyMutex, NULL);  //  thread creation (thread | attribute | name of Method | argument).
145                                                            //It creates thread and assigns to it chosen method
146      pthread_create(&thread2, NULL, DivideMutex, NULL);
147
148      pthread_join(thread1, NULL);                          //wait for thread termination, then can be used by other processes
149      pthread_join(thread2, NULL);
150
151      printf("Final Value of shared integer is %d\n", sharedInteger); //prints the last updated value of shared variable
152  }
```

Figure 42. Mutex lock function code from the program which accompanies this report

Compile the program by using gcc: To compile a multithreaded program using gcc, we need to link it with the pthreads library. The below is the command used to compile the program. ($ gcc filename.c -lpthread).*(geeksforgeeks.org,2019)*

Create two threads, thread1 and thread2 in the Mutex method by using “pthread_create(...) function”, the code “pthread_join(...)” function is the terminal for starting thread work in the mutex lock. The final code part of sharedInteger = localValue is for achieving update value function, then “printf(...,sharedInteger)” is for output the last updated value of a shared variable.

```
65   void* MultiplyMutex()
66   {
67       pthread_mutex_lock(&MutexThread); //lock this thread
68
69       int localValue;
70
71       printf("\n\nThread #1 starts its work...\n");
72
73       localValue = sharedInteger;
74       printf("Thread #1 takes value of shared variable: %d\n", localValue);
75
76       localValue *= 10;
77
78       printf("Thread #1 updated local variable: %d\n", localValue);
79
80       printf("Thread #1 goes to sleep...\n\n");
81       sleep(0.5);                        //Sends to sleep for 0.5 s Thread 1, now Thread 2 will continue
82       printf("Thread #1 resumes its work...\n");
83
84       sharedInteger = localValue;
85       printf("Thread #1 finished work and updated shared integer: %d\n\n", sharedInteger);
86
87       pthread_mutex_unlock(&MutexThread); //unlock this thread
88   }
```

Figure 43. Mutex lock thread multiply method code part.

The multiply mutex function is for thread achieving multiply method in the mutex lock, at the first line, the code of “pthread_mutex_lock(&MutexThread)” is for locking the shared memory purpose, when the shared memory is locked, there are will not be allowed other threads to join, which can avoid the deadlock occurring. After the thread finished the multiply method

operating procession and updated the operation result to the shared integer, then the code of “pthread_mutex_unlock(&lock)” will unlock the mutex lock allowing other threads to join.

```
90   /// Method Name: DivideMutex
91   /// Purpose: To divide shared digit by 10, but WITH THE USE OF MUTEX
92   void* DivideMutex()
93   {
94       pthread_mutex_lock(&MutexThread);
95
96       int localValue;
97
98       printf("Thread #2 starts its work...\n");
99
100      localValue = sharedInteger;
101      printf("Thread #2 takes value of shared variable: %d\n", localValue);
102
103      localValue /= 10;
104
105      printf("Thread #2 updated local variable: %d\n", localValue);
106
107      printf("Thread #2 goes to sleep...\n\n\n");
108      sleep(0.5);  //Sends to sleep for 0.5 s Thread 1, now Thread 2 will continue
109      printf("Thread #2 resumes its work...\n");
110
111      sharedInteger = localValue;
112      printf("Thread #2 finished work and updated shared integer: %d\n\n", sharedInteger);
113
114      pthread_mutex_unlock(&MutexThread); //unlock this thread
115  }
```

Figure 44. Mutex lock thread divide method code part.

After thread 1 finished the work and unlock the shared memory, thread 2 join into working and at firstly locked the shared memory by using code “pthread_mutex_lock(&MutexThread)”, then taking the thread 1 updated shared value do divide 10 operations, as similar as thread 1 operating steps, thread 2 also update the operating result to the shared integer, then unlock the shared memory.

Deadlocks & Starvation

The system of computers contains a variety of resources, and the CPU can process only one computer program at a time, *“Computer systems are full of resources that can be used only by one process at a time”.(*Andrews. Tanenbaum Herbert Bos,2015,*p435).* if two or more computer programs are using the same resource at the same time, the resource will block the computer programs from processing, which can lead to a computer system crash. *"At this point, both processes are blocked and will remain so forever. This situation is called a deadlock”.(Andrew S. Tanebaum Herbert Bos,2015,p435).*

Deadlocks can also occur in a variety of other situations, not only on the software resources but also on the hardware resources.

For example, in a database system, a program may have to lock several records it is using, to avoid race conditions. *"If process A locks record R1 and process Blocks record R2, and then each process tries to lock the other one's record, we also have a deadlock"(Andrew S. Tanebaum Herbert Bos,2015,p436).* Thus, deadlocks have a variety that can occur in different circumstances.

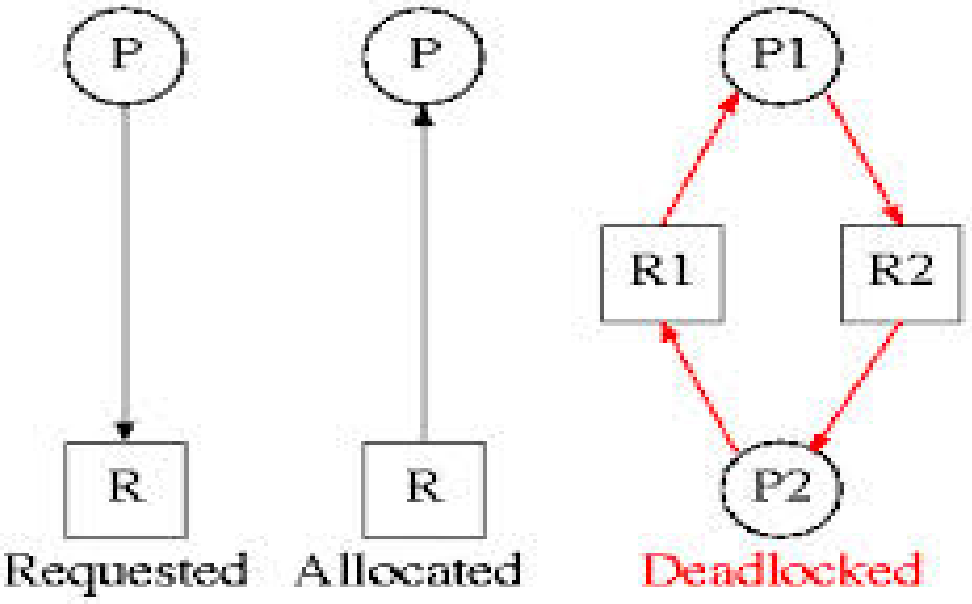
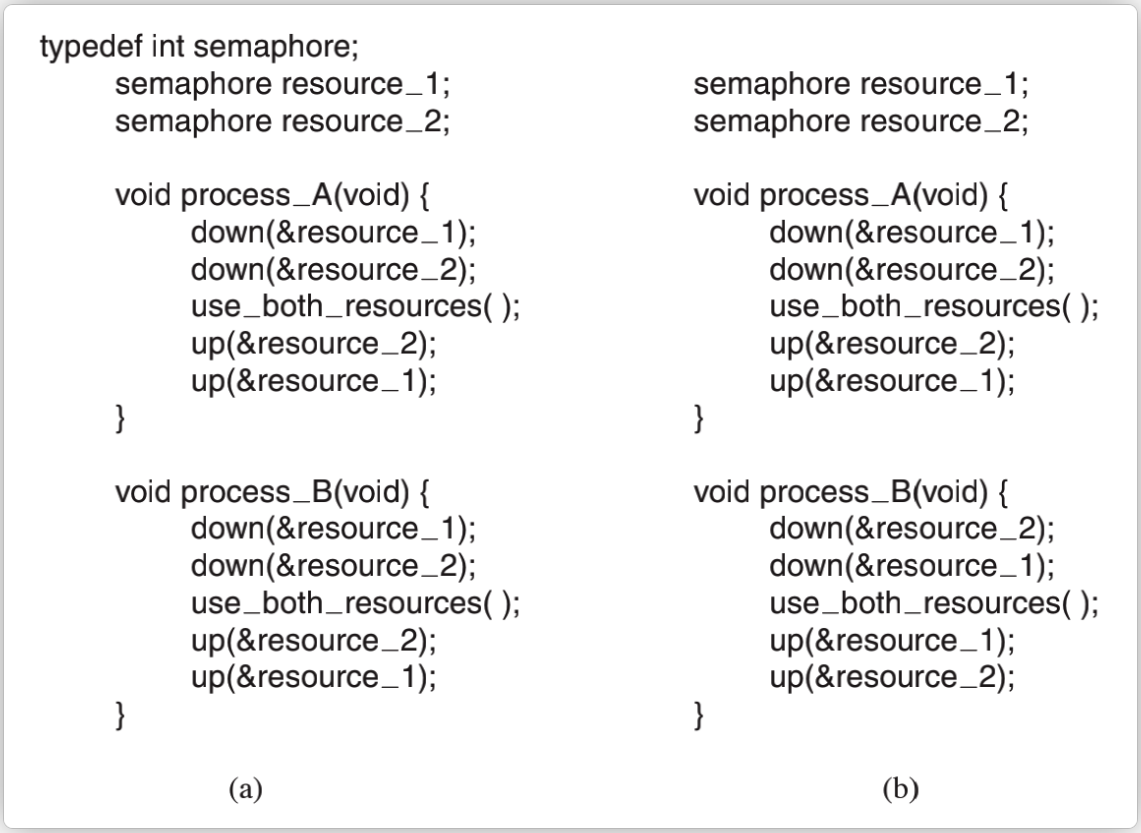


Figure 45. Deadlock diagram（*Image Courtesy: cs.nyu.edu, siber.cankaya.edu.tr*）

The following two program codes achieve the same function but are coded by using different coding styles, however, one of the programs running results will occur the deadlock, and another running normally without deadlock.



```
typedef int semaphore;
        semaphore resource_1;
        semaphore resource_2;

        void process_A(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }

        void process_B(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }
```

(a)

```
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
}

void process_B(void) {
        down(&resource_2);
        down(&resource_1);
        use_both_resources( );
        up(&resource_1);
        up(&resource_2);
}
```

(b)

Figure 46. (a) deadlock-free code. (b) code with potential deadlock.***(****ANDREW S. TANENBAUM HERBERT BOS,2015,p439****)****.*

"*Here we see how what appears to be a minor difference in coding style—which resource to acquire first—turns out to make the difference between the program working and the program failing in a hard-to-detect way. "*

The resource deadlock will happen in some particular situation, there are four conditions that can lead to resource deadlock occurring.

1. Mutual exclusion condition
2. Hold-and-wait condition
3. No-preemption condition
4. Circular wait condition

Mutual exclusion condition: " *Each resource is either currently assigned to exactly one processor is available." (Andrew S. Tanebaum Herbert Bos,2015,p440).* This means one process currently occupies one resource or there is one resource that can be used.

Hold-and-wait condition. *"Processes currently holding resources that were granted earlier can request new resources."(Andrew S. Tanebaum Herbert Bos,2015,p440).* which means one process can hold more than one resource, however, the other processes will have no resources to use and deadlock occurs.

No-preemption condition. *“Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.”* (Andrews. Tanenbaum Herbert Bos,2015,p440). This means the occupied resources have to be released by the holding process.
Circular wait condition. “*There must be a circular list of two or more processes, each of which is waiting for a resource held by the next member of the chain.”*(Andrews. Tanenbaum Herbert Bos,2015,p440)**.** This means, in the circular list each process is waiting for a resource from the following process to release the resource for use.

If the above four conditions happen on one resource at the same time, then a deadlock must occur.

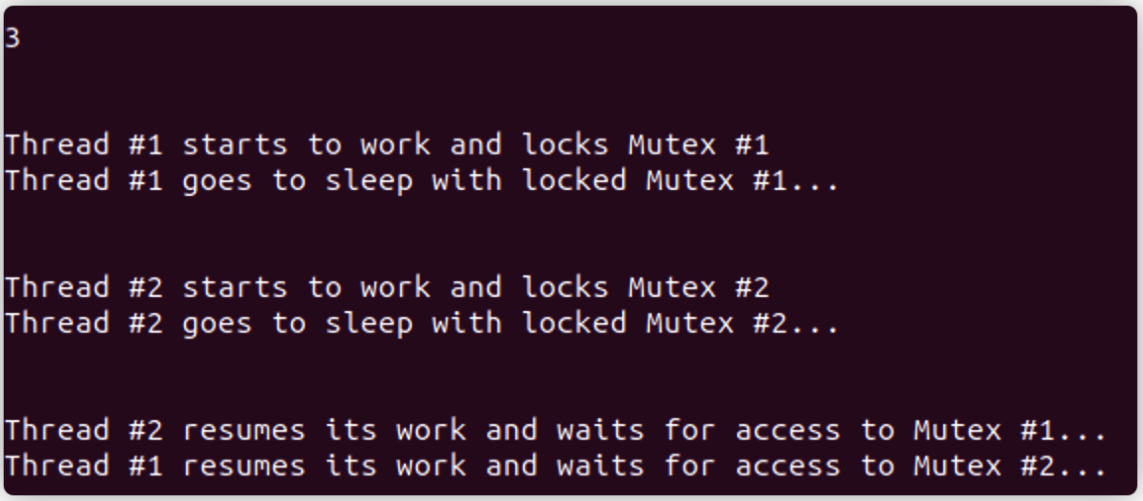
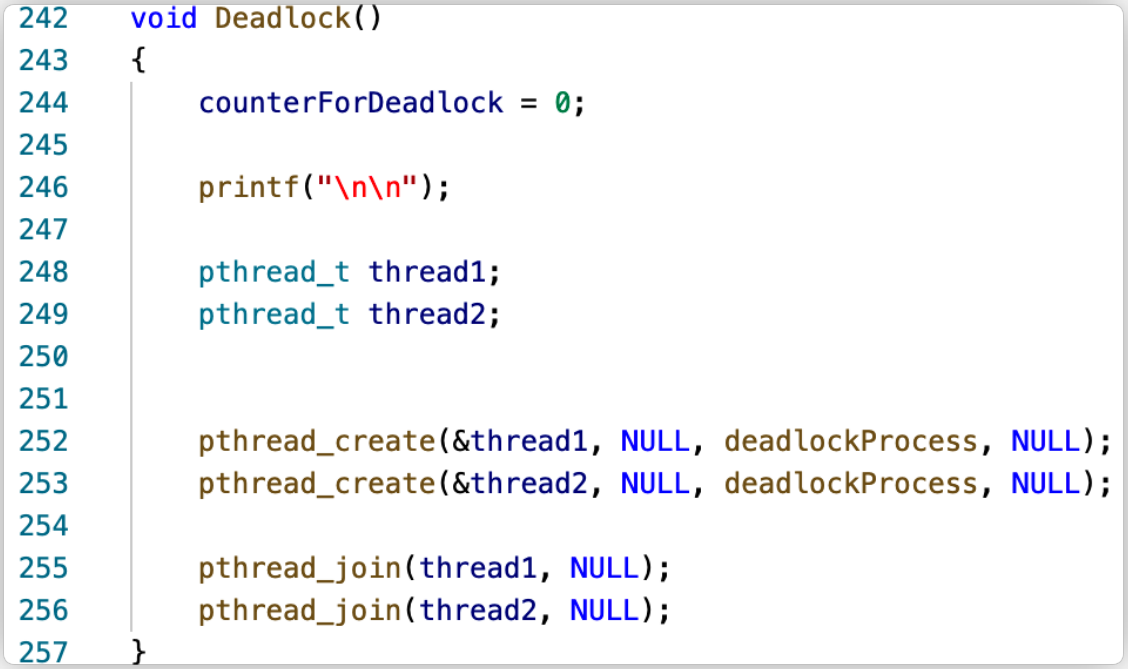


Figure 47. The Deadlock result from the program which accompanies this report

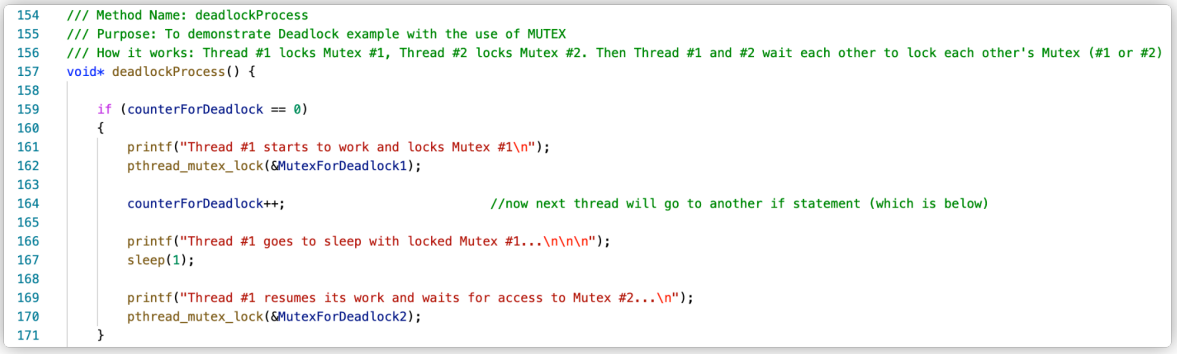
An example of deadlock from the program is included with the report:
Thread #1 starts to work and also locks shared memory as mutex #1 then goes to sleep, as similar as Thread #1, Thread #2 also after work, go to sleep and lock the shared memory. However, the deadlock will occur when Thread #1 & Thread#2 resume their work and try to access each other's shared memory.



```
242    void Deadlock()
243    {
244        counterForDeadlock = 0;
245
246        printf("\n\n");
247
248        pthread_t thread1;
249        pthread_t thread2;
250
251
252        pthread_create(&thread1, NULL, deadlockProcess, NULL);
253        pthread_create(&thread2, NULL, deadlockProcess, NULL);
254
255        pthread_join(thread1, NULL);
256        pthread_join(thread2, NULL);
257    }
```

Figure 48. The deadlock method code part

Set the default counter of deadlock is zero, then create two threads, thread 1 and thread 2, then the code “pthread_join(...)” is for achieving resume thread back to its work function.



```
154  /// Method Name: deadlockProcess
155  /// Purpose: To demonstrate Deadlock example with the use of MUTEX
156  /// How it works: Thread #1 locks Mutex #1, Thread #2 locks Mutex #2. Then Thread #1 and #2 wait each other to lock each other's Mutex (#1 or #2)
157  void* deadlockProcess() {
158
159      if (counterForDeadlock == 0)
160      {
161          printf("Thread #1 starts to work and locks Mutex #1\n");
162          pthread_mutex_lock(&MutexForDeadlock1);
163
164          counterForDeadlock++;                          //now next thread will go to another if statement (which is below)
165
166          printf("Thread #1 goes to sleep with locked Mutex #1...\n\n\n");
167          sleep(1);
168
169          printf("Thread #1 resumes its work and waits for access to Mutex #2...\n");
170          pthread_mutex_lock(&MutexForDeadlock2);
171      }
```

Figure 49. Thread #1 in the deadlock process method code part

The default global parameter counterForDeadlock is zero, match the if condition, thread #1start to work and lock shared memory as mutex #1, counterForDeadlock add one, then thread #1 sleep one second, thread #1 will resume its work until mutex lock #2 unlocked.

```
207     if (counterForDeadlock == 1)                    //Thread #2 will enter this if statement
208     {
209         counterForDeadlock++;                       //next thread will go to another if statement (which is below)
210
211         printf("Thread #2 starts to work and locks Mutex #2\n");
212         pthread_mutex_lock(&MutexForDeadlock2);
213
214         printf("Thread #2 goes to sleep with locked Mutex #2...\n\n\n");
215         sleep(1);
216
217         printf("Thread #2 resumes its work and waits for access to Mutex #1...\n\n");
218         pthread_mutex_lock(&MutexForDeadlock1);
219
220
221         printf("Thread #2 finally accessed to Mutex #1\n");
222         printf("Thread #2 finished its work...\n\n\n");
223         pthread_mutex_unlock(&MutexForDeadlock1);
224     }
```

Figure 50. Thread #2 in the deadlock process method code part

Figure 50 represents the deadlock process of the code that was submitted with this document and the description follows Based on the thread #1 running result, counterForDeadlock equal one satisfied the if condition, then the thread #2 start to work and lock the shared memory as mutex lock #2, then thread #2 go to sleep for one second. However, because both threads lock their current shared memory as mutex lock #1 and mutex lock #2 Then both threads keep waiting for each other to unlock the shared memory, and the system falls into the deadlock loop.

Starvation is an issue similar to a deadlock. In a heavily loaded operating system, requests for resources happen all the time. *"Some policy is needed to make a decision about who gets which resource when. This policy, although seemingly reasonable, may lead to some processes never getting service even though they are not deadlocked".* (Andrews. Tanenbaum Herbert Bos,2015,p463*).* For example, the processes with high priority will always get the resources more advanced than low priority processes, which means, it is impossible that low priority processes can get the resource.

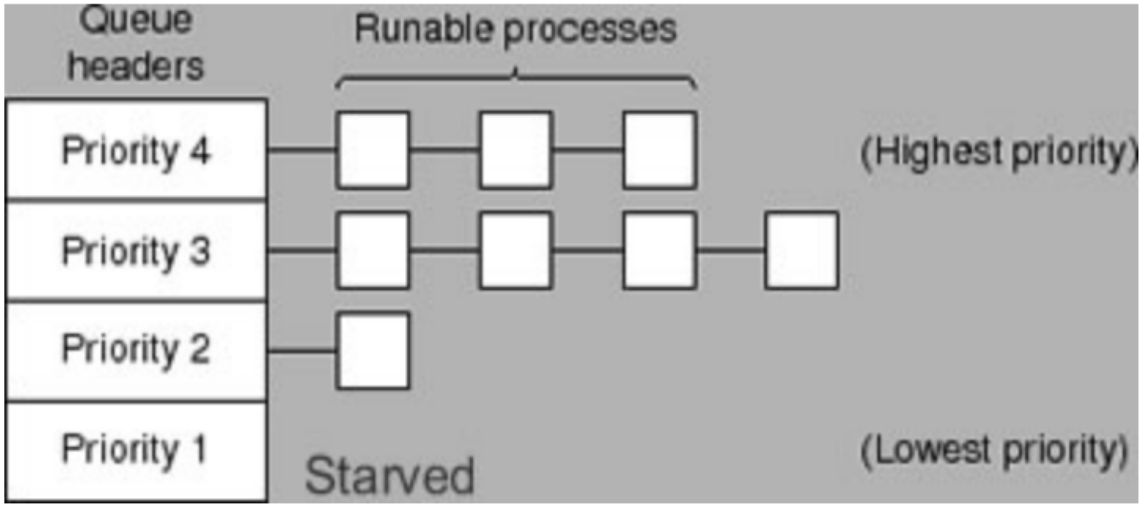


Figure 51. the starvation diagram with priority policy（*Image Courtesy: cs.nyu.edu, siber.cankaya.edu.tr*）

The starvation also can be avoided by using a first-come, first-served resource allocation policy, which means, all of the processes can be executed just by waiting in the queue's time differently, *"the process waiting for the longest gets served next. In due course of time, any given process will eventually become the oldest and thus get the needed resource."* *(*Andrews. Tanenbaum Herbert Bos,2015,*,p463).*

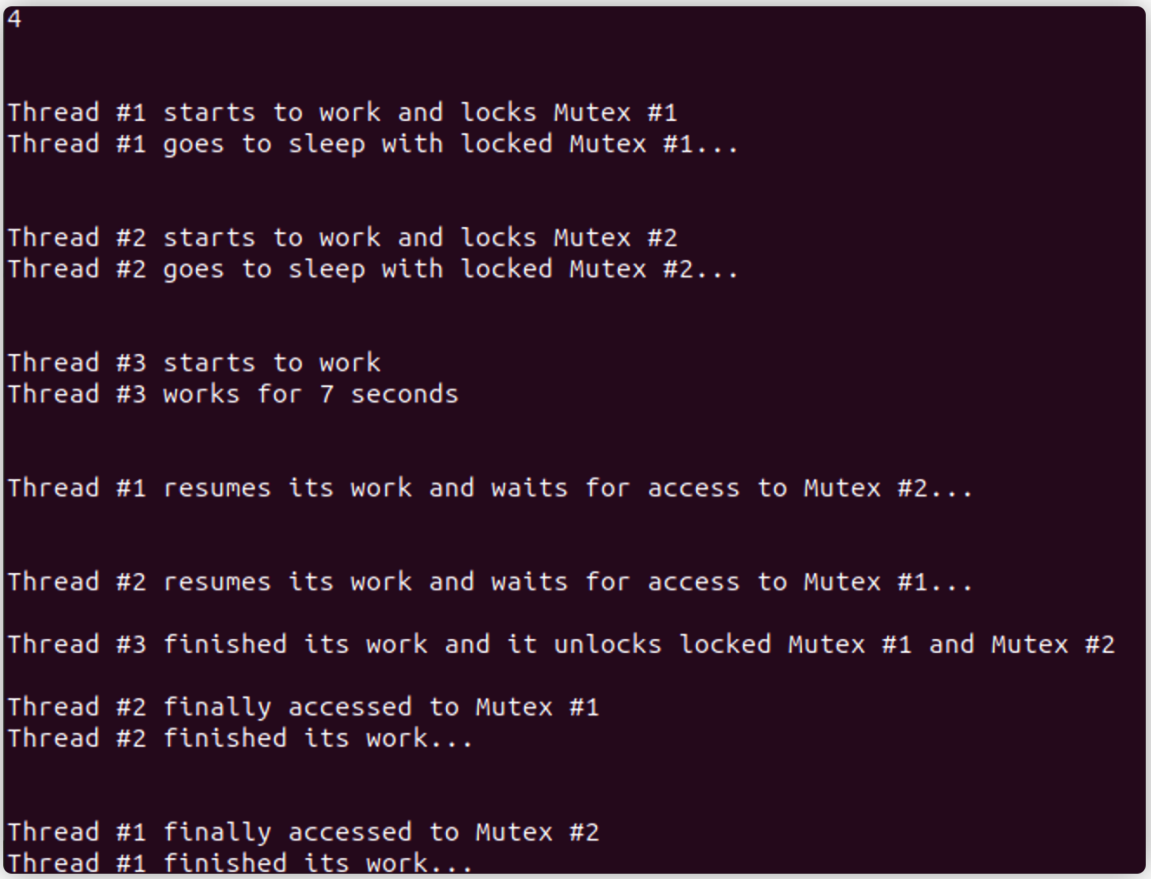


Figure 52. The result of starvation example

Thread #1 starts to works and locks the mutex #1 then goes to sleep. Similarly, thread #2 starts to work and lock the mutex #2, then goes to sleep. For now, the situation is the system falls into a deadlock because thread #1 and thread #2 both want to use each other's shared memory, but mutex #1 and mutex #2 were locked, Until thread 3 join into work and unlocked the mutex #1 and mutex #2, then thread #1 & thread #2 can resume works and accessed to mutex #2 and mutex #1. (unlike thread #1 and thread #2, thread #3 is only for unlocking mutex lock function)

```
261    void Starvation()
262    {
263        counterForDeadlock = 0;
264
265        printf("\n\n");
266
267        pthread_t thread1;
268        pthread_t thread2;
269        pthread_t thread3;
270
271
272        pthread_create(&thread1, NULL, starvationProcess, NULL);
273        pthread_create(&thread2, NULL, starvationProcess, NULL);
274        pthread_create(&thread3, NULL, starvationProcess, NULL);
275
276        pthread_join(thread1, NULL);
277        pthread_join(thread2, NULL);
278        pthread_join(thread3, NULL);
279    }
```

Figure 53. Starvation method code part

Set default counter of deadlock is zero, then create three threads by using code “ptherad_create(.....)”, for example, sending thread1 parameter into the starvation process method, then the code”pthread_join(...)” achieving resume thread back to its work function.

```
185 /// Method Name: starvationProcess
186 /// Purpose: To demonstrate Starvation example with the use of MUTEX
187 /// How it works: Same steps as in deadlockProcess method, but there is Thread #3, which will unlock Mutex #1 and #2 -> threads will finish their work.
188 void* starvationProcess() {
189
190     if (counterForDeadlock == 0)                  //Thread #1 will enter this if statement
191     {
192         counterForDeadlock++;                     //next thread will go to another if statement (which is below)
193
194         printf("Thread #1 starts to work and locks Mutex #1\n");
195         pthread_mutex_lock(&MutexForDeadlock1);
196
197         printf("Thread #1 goes to sleep with locked Mutex #1...\n\n\n");
198         sleep(1);
199
200         printf("Thread #1 resumes its work and waits for access to Mutex #2...\n\n\n");
201         pthread_mutex_lock(&MutexForDeadlock2);
202
203         printf("Thread #1 finally accessed to Mutex #2\n");
204         printf("Thread #1 finished its work...\n\n\n");
205         pthread_mutex_unlock(&MutexForDeadlock2);
206     }
```

Figure 54.The thread 1 code part of the starvation process method

As default counterForDeadlock parameter equal zero, satisfied if condition counterForDeadlock == 0, counterForDefaultLock add one, thread #1 start to work and lock the mutex #1, then sleep for one second.

```
if (counterForDeadlock == 1)                    //Thread #2 will enter this if statement
{
    counterForDeadlock++;                       //next thread will go to another if stateme

    printf("Thread #2 starts to work and locks Mutex #2\n");
    pthread_mutex_lock(&MutexForDeadlock2);

    printf("Thread #2 goes to sleep with locked Mutex #2...\n\n\n");
    sleep(1);

    printf("Thread #2 resumes its work and waits for access to Mutex #1...\n\n");
    pthread_mutex_lock(&MutexForDeadlock1);


    printf("Thread #2 finally accessed to Mutex #1\n");
    printf("Thread #2 finished its work...\n\n\n");
    pthread_mutex_unlock(&MutexForDeadlock1);
```

Figure 55. The thread 2 code part of the starvation process method

Based on the first part of the starvation process, the counterForDeadlock equal one now, which satisfied if condition counterForDeadlock == 1, then counterForDeadlock add one, thread #2 start to work and lock the mutex #2 then go to sleep for one second, the method stops now until mutex #1 unlocked.

```
225     if (counterForDeadlock == 2)                //Thread #3 will enter this if statement
226     {
227         counterForDeadlock++;                   //counterForDeadlock now will be three, so by repeat call of this method it won't enter any of 3 'if' statements
228
229         printf("Thread #3 starts to work\n");
230
231         printf("Thread #3 works for 7 seconds\n\n\n");
232         sleep(7);
233
234         printf("Thread #3 finished its work and it unlocks locked Mutex #1 and Mutex #2\n\n");
235         pthread_mutex_unlock(&MutexForDeadlock1);
236         pthread_mutex_unlock(&MutexForDeadlock2);
237     }
238 }
```

Figure 56. The thread 3 code part of the starvation process method

Based on the last two steps, conterForDeadlock equals two, then counterForDeadlock adds one, for now, the counterForDeadlock will equal three and not match any of ‘if’ conditions in the starvation process method, the reason is for avoiding repeating the process occurring error. After thread #3 sleep seven seconds, the mutex lock #1 and mutex lock #2 will be unlocked by code “pthread_mutex_unlock(...)”.

The difference between deadlock and starvation

The deadlock and starvation are similar to some degree, however, their relationship is deadlock implies starvation but starvation does not imply deadlock.

Arising conditions:

Unlike the deadlock, the arising conditions of starvation have the following circumstances:

1. *Uncontrolled management of resources*
2. *Process priorities being strictly enforces*
3. *Use of random selection*
4. *Scarcity of resources*

Prevention Techniques:

For deadlock:

1. *Infinite resources.*
2. *Waiting/sharing is not allowed.*
3. *Preempt the resources.*
4. *All requests were made at the start.*

For starvation:

1. *Independent manager for each resource.*
2. *No strict enforcement of priorities.*
3. *Avoidance of random selection.*
4. *Providing more resources.*

Progress:

For deadlock: "*No process can make progress"*, because the system falls into a deadlock loop and would not be able to deal with other progress until the deadlock loop is released.

For starvation: "*Apart from the victim process, other processes can progress or proceed".* Owning to the system still can deal with other processes, only the victim process can be influenced.

Ending A Deadlock:

For deadlock: "*Requires external intervention"*, which is the only way to end the deadlock loop.

For starvation: "*may or may not require external intervention",* because the system still can deal with other progress except the victim's progress and if there is not another high priority process that keeps on coming and executing then the victim process of starvation issue will be solved, so external intervention is not the must done condition.
(differencebetween.info,2021)

Therefore, deadlock and starvation are different from each other. Deadlock occurs when none of the processes in the set is able to continue processing due to occupancy of the required resources by some other process. On the other hand, starvation occurs when a lower priority process waits for an indefinite period of time to get the resource it requires.

Conclusion

Through this report a discussion of of the Linux Operating System took place. Beginning with how Linux managed processes and including a process lifecycle. Starting at the bootstrapping or boot process and how that creates more processes. Process scheduling was discussed and the different algorithms that schedule processes. Race Condition and the reason for race condition was written about. The critical section problem and the solutions to the critical section problem such as mutex locks and semaphores. Mutex locks where examined and explained. The reasons for deadlocks and starvation were investigated and solutions for deadlocks and starvation were examined.

Bibliography

References for Report:

Tanenbaum, A.S. and Bos, H., 2015. *Modern operating systems*. Pearson.

Silberchatz, A. Galvin P.B. Gagne g., 2018. Operating System Concepts. 10th Edition. John Wiley & Sons.

Available at: https://en.wikipedia.org/wiki/Race_condition
Accessed on: 24/11/2021

Available at: https://ostoday.org/linux/what-is-process-management-in-linux.html
Accessed on: 11/10/2021

Available at:
https://www.ibm.com/docs/en/aix/7.2?topic=environment-using-kernel-processes Accessed on 17/11/2021

Available at:
http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.sys_arch%2Ftopic%2Fkernel_Mutexes.html
Accessed on: 15/11/2021

Available at: https://www.tutorialspoint.com/race-condition-critical-section-and-semaphore
Accessed on: 15/11/2021

Available at:

https://www.keil.com/pack/doc/CMSIS/RTOS2/html/group__CMSIS__RTOS__MutexMgmt.html
Accessed on: 12/10/2021

Available at: http://www.differencebetween.info/difference-between-deadlock-and-starvation
Accessed on: 23/11/2021

Available at:
https://www.oreilly.com/library/view/understanding-the-linux/0596005652/ch03s05.html
Accessed on: 1/12/2021

Available at:
https://developer.ibm.com/tutorials/l-linux-process-management/
Accessed on: 1/12/2021

References for Programs 1 and 2:

Input-output in C:
Available at: https://www.programiz.com/c-programming/c-input-output
Accessed on: 05/11/2021

Strings in C:
Available at: https://www.programiz.com/c-programming/c-strings
Accessed on: 05/11/2021

At command in Linux:
Available at: https://linuxize.com/post/at-command-in-linux/
Accessed on:11/12/2021

Pthread_create:
Available at: https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_create.html
Accessed on: 11/11/2021

Pthread_join:
Available at: https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_join.html
Accessed on 12/11/2021

mutex, lock:
Available at: https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/
Accessed on 12/11/2021