

Institiúid Teicneolaíochta Cheatharlach



INSTITUTE *of*
TECHNOLOGY
CARLOW

At the Heart of South Leinster



redhat.®
L I N U X

Processes and Thread Synchronization

Students:	Theodora Tataru	Ana Griga
Students Number:	C00231441	C00231441
Supervisor:	Caroline Byrne	
Submission Date:	25 November 2019	

CONTENTS

Introduction.....	2
Processes	3
Process Management	5
Process Creation	7
Process Scheduling	8
Process Destruction.....	10
Thread Synchronisation	11
Race Condition	12
Mutex Locks	13
Critical Section problem.....	14
Solutions to Critical Section Problem	15
Deadlock & Starvation	16
Conclusion.....	18
Appendix 1 – Thread SYNCHRONIZATION without a variable lock	19
Appendix 2 – Thread SYNCHRONIZATION with a variable lock.....	20
Appendix 3 – Creating and Destroying a Process using Fork() and KILL()	21
Bibliography.....	22
Bibliography used in the writing report	22
Bibliography used in the code	23
Appendix 4 – Collaboration Form	24

No of words: 4135

This report contains detailed information about processes and threads in Linux Red Hat environment, requested by Caroline Byrne, as part of the continuous assignment for Operating Systems at Institute of Technology Carlow. This report includes:

- Processes in Linux Environment
- Threads, Mutex Locks, Threads Synchronization in Linux Environment
- Critical Section Problem
- Dead Lock and Starvation
- Collaboration Form
- Code Sample for Thread Synchronization **with** and **without** a Variable Lock

A computer is a device that can be instructed to carry out arithmetic and logical operations via programs. A computer is made of resources: hardware, software and data. The Operating System make sure that these resources are used as efficiently as possible and will provide the proper environment for the user to be able to use the software [1].

The Kernel, a very important component of the Operating System, is using the inter-process communication and system calls. The Kernel resides between the applications and the processing of data at the hardware level. When the Operating System is loaded into memory, in the boot process, the Kernel is the one loaded first and remains in memory until the system is shut down. The main tasks of the Kernel are to manage the disk, the tasks and the memory. Everything running in the low-level is managed by the Kernel.

The differences between the Operating System and the Kernel are as follows: Operating System is a graphical interface designed for the user, and the Kernel is a part of the Operating System that manages resources. The Kernel is a bridge between the software and the hardware of a system [24].

The notion of a process means a program in execution. A program is a dormant entity on the disk (hard disk, SSD, USB, etc.), but when active (active entity), the Operating System loads all its instructions into main memory [2].

“A program is a passive entity until it is launched, and a process can be thought of as a program in action.[23]”

Each process provides the resources needed to execute the program by creating a virtual space, executable code, a unique process identifier, environment variables, minimum and maximum working set sizes and will have at least one thread of execution but can create additional threads [3].

A CPU can execute only one process at a time, on a system with multiple CPUs there can exist more than one executing processes, but only one per CPU [2].

As a program executes, it will change its state as follows, see Fig 1:

- The process will be created, as the program is moved from the storage into the main memory: **NEW**
- Instructions from the active entity will be executed one by one sequentially: **RUNNING**
- The process may wait if needed that some events to occur: **WAITING**
- The process will wait in line, for its time to be executed: **READY**
- The process has finished the execution and will be destroyed: **TERMINATED** [4]

The Operating System plays an important role in the management of a process and is responsible of activities as follows: it schedules the amount of time a process will have the CPU for, creates and destroys a process, suspends processes, resumes processes, processes communication and processes synchronization.

A dormant program to become an active entity, also called a process, it needs to be loaded from the hard disk, to main memory and when this happens, the process will receive an absolute address. When the program begins to execute, the CPU will access the process data and instructions by accessing this absolute address. When the execution is finished, the memory can be released, and the next program can be loaded into memory and begin this cycle again.

In a multi-tasking system, as multiple processes may need each other, the computer needs to keep in the main memory few different processes, so there is a high need of process management. The Operating System will keep track of what parts of the memory are used, what processes are using it, decides which process, what data can be moved in or out of the memory and allocates/deallocates the memory as needed [4].

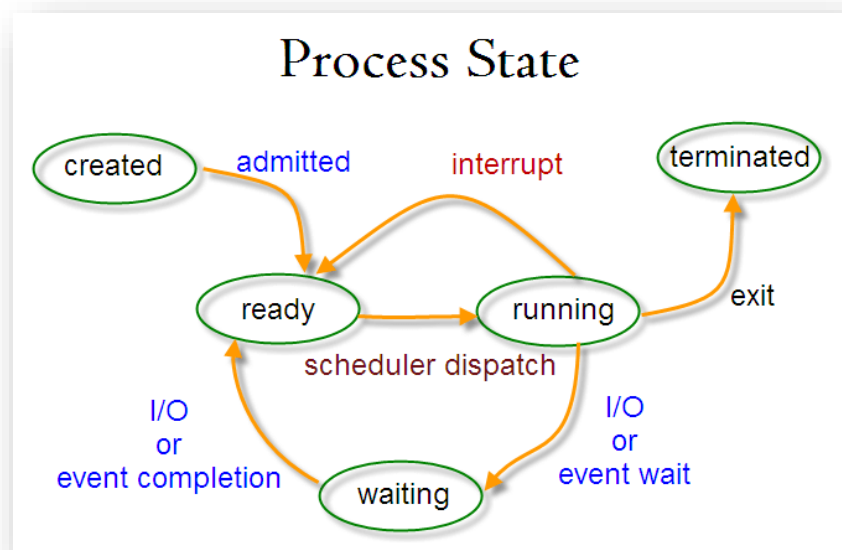


Fig 1. Process State. Source: www.tecmint.com, 2019

Linux is a multi-user and multi-processing system which means that more than one user can utilize the Operating System and multiple processes can run at the same time [1].

Linux keeps a record of all the processes in the system (See Fig 2), even if they are launched by the user or by the system itself. The record is a container of information about how the processes are running and what is happening in the system at that given point [22].

```

top - 16:42:44 up 6 min, 1 user, load average: 0.00, 0.15, 0.11
Tasks: 145 total, 1 running, 109 sleeping, 0 stopped, 0 zombie
%Cpu(s): 6.9 us, 2.4 sy, 0.0 ni, 90.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 4681696 total, 3318552 free, 576436 used, 786708 buff/cache
KiB Swap: 2097148 total, 2097148 free, 0 used, 3862488 avail Mem

  PID USER   PR  NI  VIRT  RES  SHR S  %CPU  %MEM    TIME+  COMMAND
 1152 theodora 20   0 1696176 162832 83884 S   9.6   3.5   0:25.70 gala
   652 root      20   0 497316 71868 43216 S   7.9   1.5   0:08.19 Xorg
 1853 theodora 20   0 691264 39476 28724 S   4.6   0.8   0:01.78 io.elementary.t
 2114 theodora 20   0 69696 3988 3368 R   0.7   0.1   0:00.32 top
    1 root      20   0 159612 8652 6528 S   0.0   0.2   0:02.50 systemd
    2 root      20   0      0      0      0 S   0.0   0.0   0:00.00 kthreadd
    4 root      0 -20      0      0      0 I   0.0   0.0   0:00.00 kworker/0:0H
    6 root      0 -20      0      0      0 I   0.0   0.0   0:00.00 mm_percpu_wq
    7 root      20   0      0      0      0 S   0.0   0.0   0:00.10 ksoftirqd/0
    8 root      20   0      0      0      0 I   0.0   0.0   0:00.27 rcu_sched
    9 root      20   0      0      0      0 I   0.0   0.0   0:00.00 rcu_bh
   10 root      rt   0      0      0      0 S   0.0   0.0   0:00.00 migration/0
   11 root      rt   0      0      0      0 S   0.0   0.0   0:00.00 watchdog/0
   12 root      20   0      0      0      0 S   0.0   0.0   0:00.00 cpuhp/0
   13 root      20   0      0      0      0 S   0.0   0.0   0:00.00 cpuhp/1
   14 root      rt   0      0      0      0 S   0.0   0.0   0:00.00 watchdog/1
   15 root      rt   0      0      0      0 S   0.0   0.0   0:00.00 migration/1
   16 root      20   0      0      0      0 S   0.0   0.0   0:00.06 ksoftirqd/1
   17 root      20   0      0      0      0 I   0.0   0.0   0:00.00 kworker/1:0
   18 root      0 -20      0      0      0 I   0.0   0.0   0:00.00 kworker/1:0H
   19 root      20   0      0      0      0 S   0.0   0.0   0:00.00 kdevtmpfs
   20 root      0 -20      0      0      0 I   0.0   0.0   0:00.00 netns
   21 root      20   0      0      0      0 S   0.0   0.0   0:00.00 rcu_tasks_kthre
   22 root      20   0      0      0      0 S   0.0   0.0   0:00.00 kauditd
   23 root      20   0      0      0      0 I   0.0   0.0   0:00.02 kworker/0:1
   24 root      20   0      0      0      0 S   0.0   0.0   0:00.00 khungtaskd
   25 root      20   0      0      0      0 S   0.0   0.0   0:00.00 oom_reaper
   26 root      0 -20      0      0      0 I   0.0   0.0   0:00.00 writeback
   27 root      20   0      0      0      0 S   0.0   0.0   0:00.00 kcompactd0
   28 root      25   5      0      0      0 S   0.0   0.0   0:00.00 ksm

```

Fig 2. TOP command. Source: Ana Griga's Linux VM. 2019

The Table of Processes can be accessed by using the **TOP** command in Linux Command Line. This command will display all the processes active in the system at that given point in time. The first column named Process Identification Number(**PID**) represent, the unique number of the process, the User (**USER**) column stands for the user that owns the process, **PR** column represents the priority of the task, the **NI** represents the Nice value of the task (negative represent a higher priority, and positive a lower priority.), the **VIRT** column stands for the Total Virtual Memory used by that process, the **SHR** column represents the amount of Shared Memory, the **%CPU** represents the CPU usage in percent, the **%MEM** shows the Memory usage of the process [22].

The processes can be categorized as parent and/or child. The parent class, also called **INIT** is the first program executed when the Linux system is booting up. This process manages all the other processes of the system. If you visualize the processes as a tree, INIT would be the root of that tree as it won't have any parent. The INIT process always has **ID 1**. All the following processes are also assigned with a process ID or PID [5].

The Kernel is in charge of allocating and de-allocating CPU time to each process. The **Dispatcher** is a part of the Kernel that controls the allocation of time of the CPU according to the process table. It performs the switching from one process to another as needed. The **Scheduler** is also part of the Kernel that adds new process and removes processes to the process table. The Scheduler policy is to maximize the response time, minimize turnaround time and minimize waiting time [2].

To create a new process, an existing process needs to duplicate itself, the copy is called a child process. The child will have the same environment, resource tables and other accounting services as the parent, the only difference would be the ID. The procedure described is called **forking**. [6]. The **Fork** system call returns two values, since the two processes are identical, the child and the parent are both examining the returned values, to discover which role they are going to play.

After forking, the address space of the child process is overwritten with the new process data, done through an **exec** call to the system. [9]. The fork and exec mechanism will switch the old command with a new one. This mechanism is creating all the RedHat processes.

Both processes, the parent and the child, are simultaneously processed, and the control is never returned to the parent process, unless there is an error in the execution [7].

Each process has a unique PID. As mentioned before, 1 is reserved for the INIT process, which is loaded into memory when the system is booted up. The process created after the INIT can have a value between 2 and 32,768. These values do not need to be assigned in sequential order. [5] See Fig 3.

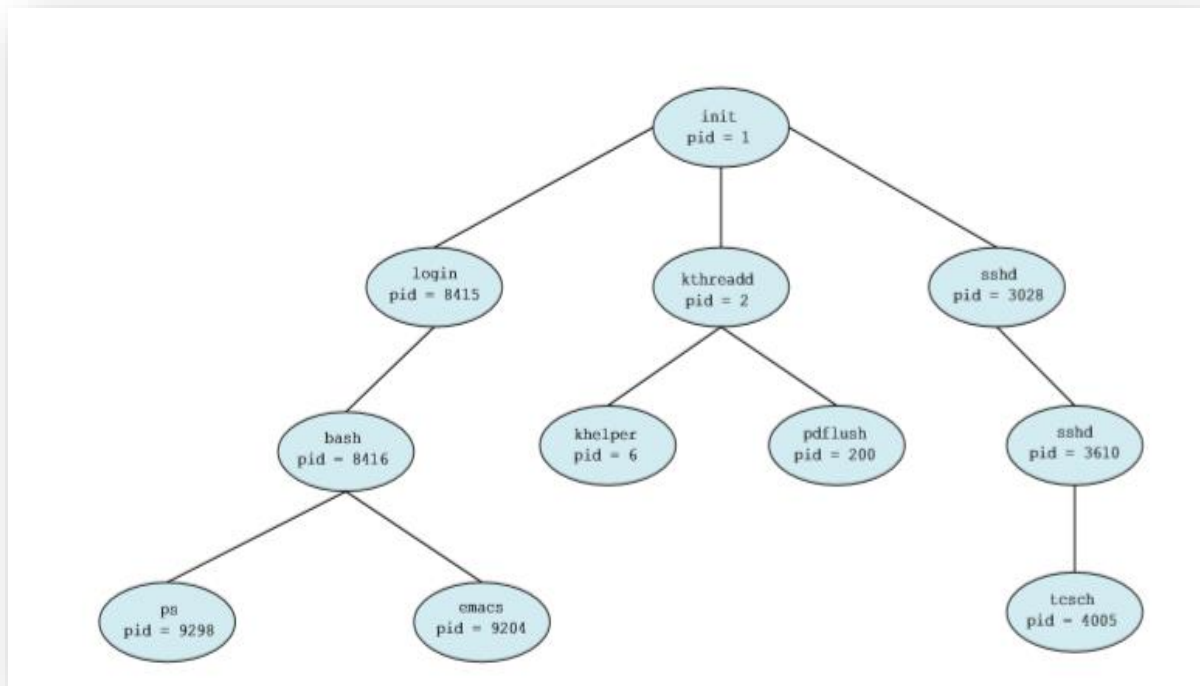


Fig 3. A tree of processes on a Linux system.

Source: Abraham Silberschatz, Peter Baer Galvin and Greg Gagne - *Operating Systems (9th Edition)* 2013

Only one process can have the “attention” of the CPU at once, but for the user, it creates an appearance that programs are running simultaneously. Everything is happening extremely fast. Therefore the user experience feels smooth and continuous [2].

On systems with multiple CPUs or multicore CPUs, multiple processes can execute at the same time. But each CPU or nucleus will run only one process at once [1].

As mentioned above, the program responsible for allocating time with the CPU for each waiting process is called **process scheduler**. The niceness of a process is a numeric value, important to the Kernel, to be able to know how a process should be treated in relation to other processes. Based on complex algorithms, the process scheduler decides which program is next to be executed by the CPU, and also decides when the execution should be ended, an action called **preemption** [8]. The scheduler manages the order and timing of processes to provide the best possible performance of the system by maintaining a fair balance of the time each process has with the CPU, based on their **static** and **dynamic** priorities [9].

When the scheduler in Linux runs, every process is examined, and its “rank” is computed. The process with the highest rank will run next. Looking at the scheduler delay in detail, the scheduler is using a timer that interrupts every 10 milliseconds [10].

Linux RedHat’s has three normal scheduling policies divided into two categories:

1. Realtime policies:

SCHED_FIFO – it is a static priority scheduler as it defines priority between 1 and 99. The scheduler will scan the SCHED_FIFO list and the highest priority will run.

SCHED_RR – is a more detailed version of FIFO where a thread is allowed to run only for a maximum time quantum.

2. Normal policies:

SCHED_OTHER – can be used at only static priority 0.

SCHED_BATCH – is a more detailed version of SCHED_OTHER assuming that each thread is more CPU consuming.

SCHED_IDLE – this policy is designed to run jobs with extremely low priority.

The way the scheduling policy acts in choosing which application thread will run is not an easy task. Usually, the real time policy should be used for important tasks and time critical tasks [11].

As mentioned above, processes can be preemptive and non-preemptive. Preemptive means that processes can be interrupted by a higher priority process that waits in the queue. In this case, the running process is switching from the running queue to the ready queue giving the right to run to the higher priority process. In this situation,

the preempted process needs to save its state at the time of interruption, so that when it is time to resume its time with the CPU, to know where to resume its activity. To be able to save the state in the moment of interruption, data will be saved in the registers and stack pointers. This switching, from an active state to a waiting state and vice versa costs processing time, because saving and reloading information into/from memory takes time. This is known as a context switch.

Non-Preemptive processes cannot be interrupted by any other processes while they are running. The ready processes from the queue need to wait until the running process is completed.

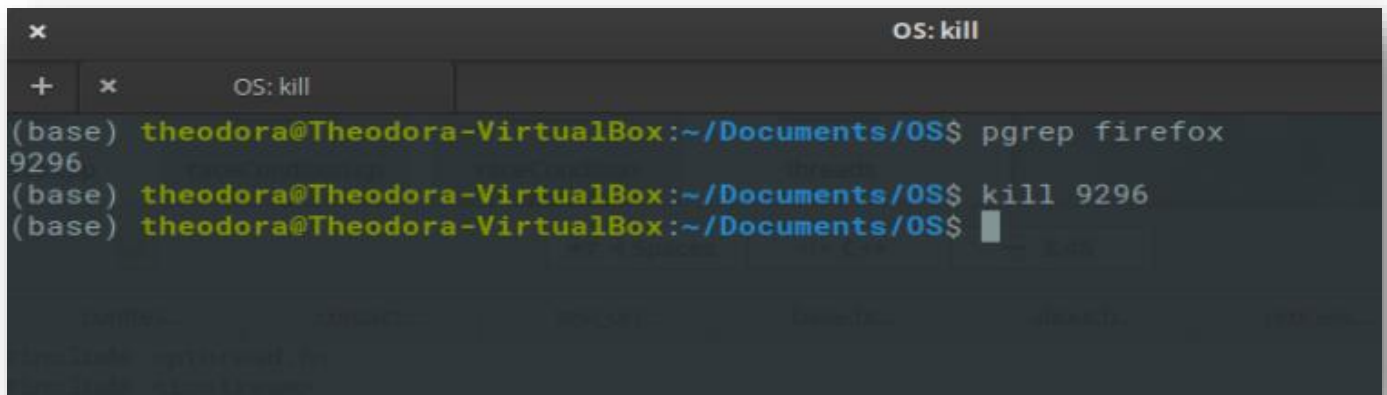
Both preemptive and non-preemptive processes are used and needed in Linux RedHat.

INIT, besides its role as the main process in the system, also has a very important role in the destruction of other processes.

When a process has finished its role, it calls a routine named **exit()** to notify that its job is done. **exit()** removes all references of the process: memory, open files and any other used resources that are not shared with other processes. It returns an exit code, an integer, that is explaining the reason for exiting – 0 indicating a successful termination. Before the process is allowed to disappear, the parent acknowledges the requirement of exiting of the child with a call to wait. The parent receives an indication if the child exits voluntarily or was killed and also obtains a brief of the child resources. If the parent no longer exists (if there is no **wait** call) the **INIT** process will adopt the orphan process and performs the **wait** call.

As mentioned above, processes communicate with each other using signals. In the case of destroying a process, the signal used is **SIGTERM**.

The **SIGTERM** is a generic signal used to terminate a program. This signal is a delicate and elegant way to terminate a process as the process is politely asked to terminate. This signal is produced by the *kill* command (See Fig 4), followed by the *PID* of the process. Deepening on the user that created the process, there might a need of sending the **SIGTERM** signal, logged in as root.



```

x                                     OS: kill
+  x   OS: kill
(base) theodora@Theodora-VirtualBox:~/Documents/OS$ pgrep firefox
9296
(base) theodora@Theodora-VirtualBox:~/Documents/OS$ kill 9296
(base) theodora@Theodora-VirtualBox:~/Documents/OS$

```

Fig 4. Using Kill Command. **Source:** Theodora Tataru's Linux VM. 2019

Sometimes, the **SIGTERM** might not work, but there exists a harsh way of terminating a process using **SIGKILL**. We can send this signal by, using the *kill - 9* command.

The **kill** command is used to terminate the process and can be used by the users that created the process or by the root [9]. See [Appendix 3](#)

Every process has and starts with a single **thread**, also called **primary thread**. The thread is the smallest unit of execution for which the processor allocates time, but because the thread is not a program, it cannot run on its own, it will run within a program. A computer can run in parallel with as many threads as there are CPU's on the computer [12].

A thread will have an ID, a program counter, a register set and a stack that are shared with the other threads of the same program. A program with only a thread in control can perform only one task at the time called **single thread**, while a program that runs multiple threads in parallel, can perform multiple tasks at the same time, called **multithreaded** process.

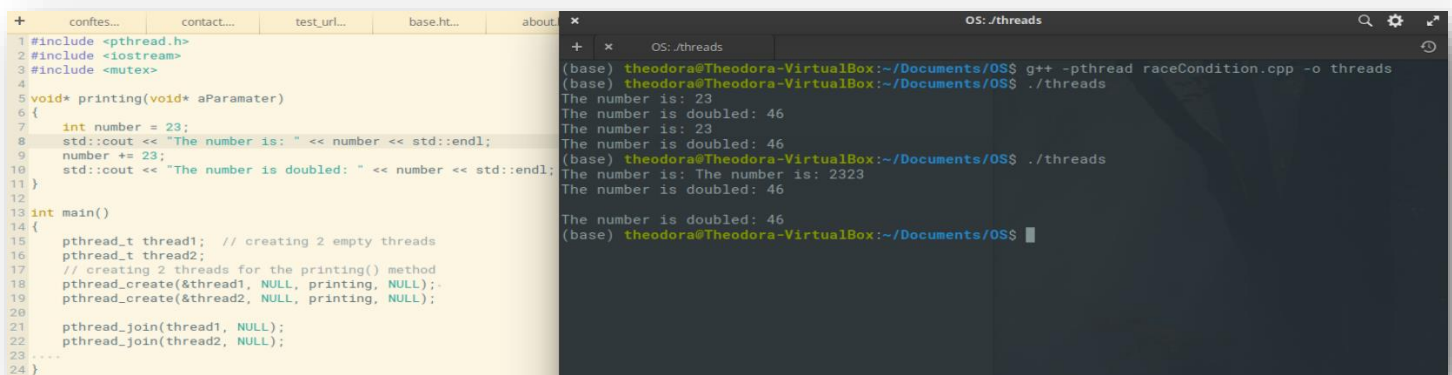
The benefit of multithreading is enormous: sharing memory and resources of the process they belong will save **time** and **resources**.

As mentioned earlier, the Linux system duplicates a process when there is a need for creating a child process by using `fork()` and `exec()`. Linux is using the term **task** for both processes and threads. When a thread is duplicated, by using `clone()`, a set of flags will determine how many resources will be shared between the parent and the child thread [4].

Sharing resources is extremely beneficial but there are multiple disadvantages as well: sharing the same resources could generate conflicts when multiple threads are trying to modify the same data(variables). That is the moment when **thread synchronization** starts: ensuring that multiple threads are not executing at the same program segment called the **critical section** which usually is modifying data, updating tables, writing to a file, etc. In that situation, if one thread is running this particular segment of the program, the others should wait until the active thread is finished its job. If the techniques applied for synchronization will fail, that would cause a **race condition**. [14]

In multi-tasking systems, like Linux, a processes life is complicated as thread A and thread B can simultaneously execute and could interface with each other [15].

Race Conditions happen when the program is relying on a particular sequence of events, but because of multiple threads running in parallel the expected result can be incorrect. This is happening when at least two threads are trying to access and manipulate shared resources. For example, if thread number 1 modifies a global variable, then thread number 2 reads and modify the same variable according to its instructions, then the first thread reads once more the same variable. In this case thread number 2 will experience an error, because the variable was changed unexpectedly. See Fig 6.



The image shows a code editor on the left and a terminal window on the right. The code editor contains a C++ program named `raceCondition.cpp` that demonstrates a race condition using `pthread`. The program has a global variable `number` initialized to 23. A function `printing` prints the value of `number` and then increments it by 23. In the `main` function, two threads are created, both calling the `printing` function. The terminal output shows the execution of the program, where the threads are interleaved, leading to an incorrect final value of 46 instead of the expected 69.

```

1 #include <pthread.h>
2 #include <iostream>
3 #include <mutex>
4
5 void* printing(void* aParamater)
6 {
7     int number = 23;
8     std::cout << "The number is: " << number << std::endl;
9     number += 23;
10    std::cout << "The number is doubled: " << number << std::endl;
11 }
12
13 int main()
14 {
15     pthread_t thread1; // creating 2 empty threads
16     pthread_t thread2;
17     // creating 2 threads for the printing() method
18     pthread_create(&thread1, NULL, printing, NULL);
19     pthread_create(&thread2, NULL, printing, NULL);
20
21     pthread_join(thread1, NULL);
22     pthread_join(thread2, NULL);
23
24 }

```

Terminal output:

```

(base) theodora@Theodora-VirtualBox:~/Documents/OS$ g++ -pthread raceCondition.cpp -o threads
(base) theodora@Theodora-VirtualBox:~/Documents/OS$ ./threads
The number is: 23
The number is doubled: 46
The number is: 23
The number is doubled: 46
(base) theodora@Theodora-VirtualBox:~/Documents/OS$ ./threads
The number is: The number is: 2323
The number is doubled: 46
The number is doubled: 46
(base) theodora@Theodora-VirtualBox:~/Documents/OS$

```

Fig 6. RACE CONDITION. Source: Theodora Tataru's Linux VM. 2019

The thread scheduling algorithm can swap between threads at any time, making the detection of a possible race condition hard to detect. The best way to prevent this is to make sure that only one thread at a time will access and modify a particular value. This action is called **synchronization** [4].

The result of an uncontrolled race condition depends on which thread gets there first and what other threads are running in parallel trying to read/write the same data. Usually the result is undesirable [15].

Each program has a segment in its body, called a **critical section**. In this section, usually global variables are changed, tables are updated, files are modified, etc. As mentioned earlier, only one thread should read or write at once to this delicate segment.

The simplest way of preventing the critical section from happening is with **mutual exclusion**, technical term: **mutex lock**. See Fig 7.

Mutex locks are used to prevent race conditions not to solve them. A process must access the lock before reading the critical section and will release the lock when it is finishing reading the critical section. The **acquire()** function is used to access and mark as unavailable the lock and then the **released()** function will free the lock. The mutex lock has a Boolean variable return that indicates if the lock is available or not. If the function **acquire()** succeeds accessing the lock then is considered unavailable.

If another process is trying to access the critical section in the program, it needs to wait until the lock is available.

The disadvantage of this mechanism is that sometimes there are threads waiting to execute. If a program is reading the critical section, the program waiting will continuously loop the **acquire()** function which is a problem because this **busy waiting** is wasting the CPU cycles when other processes could execute productively [4].

```

1  #include <pthread.h>
2  #include <iostream>
3  #include <mutex>
4
5  std::mutex mut;
6
7  void* printing(void* aParamater)
8  {
9      mut.lock();
10     std::cout << "Is this printing???" << std::endl;
11     mut.unlock();
12 }
13
14 int main()
15 {
16     pthread_t thread1; // creating 2 empty threads
17     pthread_t thread2;
18     // creating 2 threads for the printing() method
19     pthread_create(&thread1, NULL, printing, NULL);
20     pthread_create(&thread2, NULL, printing, NULL);
21
22     pthread_join(thread1, NULL);
23     pthread_join(thread2, NULL);
24
25 }
```

Fig 7. Mutex Lock. Source: Ana Griga - Linux VM. 2019

As described earlier, **process synchronization** allows multiple processes running in parallel to share system resources and to maintain data consistency. Each program will have instructions that will modify data in the **critical section**. It is called critical because if more than one process is running in this segment, the data consistency is lost. [18]. See Fig 8.

A simple solution for the critical section. Example:

```
acquireLock();
```

Process Critical Section

```
releaseLock();
```

Fig 8. Critical Section. Source: Ana Griga - Linux VM. 2019

The disadvantage of this mechanism is that it requires [4] **Busy waiting** or **busy looping** (a process which repeatedly calls to acquire() trying to enter the critical section of a program.)[17].

The critical section is almost never occupied. Therefore busy waiting occurs very rarely and for short periods of time but there is a possibility that this section could be very long, and the critical section could be minutes maybe hours occupied [4].

This critical section of a program needs rules and solutions to synchronize all these different processes running at the same time.

First, the mutual exclusion described above allows only one process to execute within the critical section at a time. The other processes need to wait until the critical section is unlocked.

Secondly, if the critical section is unlocked then **progress** needs to be made by allowing another process to access it.

Then last but not the least important **Bounded Waiting**, the OPERATING SYSTEM needs to make sure that each process has limited execution time so that the waiting processes will not wait endlessly [16].

A **deadlock** can occur in few situations, for example, when a process is waiting for an event that will never occur, when a closed circle of processes exists, and each process is waiting for a resource held by the other process. See Fig 9.

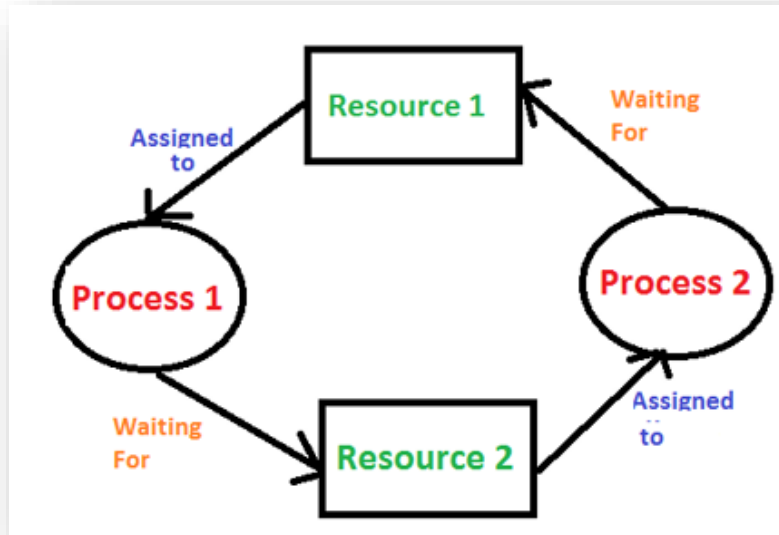


Fig 9. Deadlocks Source: www.geeksforgeeks.org 2019

Strategies for dealing with deadlocks will imply prevention or avoidance, detection which will imply recovery or ignoring the problem [2].

Even if **ignoring the deadlock** sounds like a bad idea, if deadlock is very rare and if the system does not have the ability to recover by itself, then it should be rebooted.

Mutex is one of the strategies used to **prevent** a deadlock. Also, resource holding can be prevented by forcing each process to request all the resources it needs to run in advance. Another way to avoid a deadlock occurring is to predict the possibility of it happening. This can be achieved by not allowing a condition to occur that may cause a deadlock by building checks into the code [2] (depending on the skill of the developer and the type of work being done by the system) [20].

Once a deadlock has been detected, the main goal is that the system should return to normal as quickly as possible. There are several algorithms used to detect a circular wait such as Banker's algorithm developed by Edsger Dijkstra. This method guarantees that the deadlock will not occur by denning one of the necessary conditions of deadlock.

Recovery will have at least one victim when it is removed from the deadlock. The algorithm should select the victim with the least negative effect and should also keep in mind the number of other jobs that would be affected if this process is selected as a victim [2].

Another major problem is **starvation**. A process can be ready, in the queue, ready to be executed when it is its turn to access resources. The scheduler might leave the process to wait indefinitely, if the Operating System is heavily loaded and the process has a low priority. Just a slight starvation can lead to an unresponsive system [2].

The starvation may be caused by bad programming, errors in scheduling, lack of resources and other errors.

Aging is a mechanism that increases the priority of a waiting process when that process is waiting for a long time. This procedure, even very useful, could as well take a very long time.

If a process would have a priority as low as 20, and by ageing, the priority will decrease with 1 every 15 seconds, the process turn to access resources would come only after 5 minutes.

Both deadlock and starvation are delaying the system. Deadlocks could lead to starvation and starvation could lead to deadlocks; both could be fatal for the running process. Programmers should take extra care and prevent these events from occurring.

This report shows the importance of the Processes and the concept of Thread Synchronization within the Linux Operating System.

It addresses the Process Management as an integral part of any Operating System and describes how processes are created and scheduled when multiple ones are running at a particular instance of time and destroyed.

The report explains the creation of a new process, called a child process using the `fork()` system call.

The process scheduler is also explained as being the Kernel's component that selects which process will run next. Deciding what process can run, the scheduler is responsible for the best utilization of the system giving the impression that multiple processes are executed simultaneously.

As described in this document the process destruction occurs when the process calls the `exit()` system call. A process is destroyed when it is ready to terminate or on return from the main subroutine of a program. When a process terminates, the Kernel releases the resources held by that process.

This assignment explains Thread Synchronization as the mechanism which ensures that two or more concurrent processes or threads do not execute simultaneously a particular program segment called critical section. Thread synchronization is used to avoid conflicts for accessing shared resources.

This document details the race condition which is a special condition that may occur inside a critical section. A critical section is a section of code that is executed by multiple threads and where the sequence of execution for the threads makes a difference in the result of the concurrent execution of the critical section.

When you have a multi-threaded application, the different threads sometimes share a common resource, such as a variable or similar. This shared source often cannot be accessed at the same time, so a construct is needed to ensure that only one thread is using that resource at a time.

This report also covers mutual exclusion which is a way of preventing the critical section problem and mutex locks which are used to prevent race conditions.

Both deadlock and starvation are mentioned in this document. These notions are related concepts in multiprocessing operating systems, which cause one or more threads or processes to stuck in waiting for the resources they need. Deadlock is a situation which arises when one or more processes request access to the same resource causing the process to freeze, whereas starvation is caused by deadlock which pushes the process off to an indefinite waiting mode state because processes are denied access to a resource held by a high priority process and need to wait forever.

```

// Theodora Tataru – C00231174
// Ana Griga – C00231441
// Thread synchronization without a lock()

#include <pthread.h>
#include <iostream>
#include <mutex>

void* printing(void* aParamater)
{
    std::cout << "This is the thread PID: " << pthread_self() << std::endl;
    int number = 23;
    std::cout << "The number is: " << number << std::endl;
    number *= 23;
    std::cout << "The number is doubled: " << number << std::endl;
}

int main()
{
    pthread_t thread1; // creating 2 empty threads
    pthread_t thread2;
    // creating 2 threads for the printing() method
    pthread_create(&thread1, NULL, printing, NULL);
    pthread_create(&thread2, NULL, printing, NULL);

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

} [25,26].

```

```

OS: ./raceCondition
+ guessingGame x OS: ./raceCondition
(base) theodora@Theodora-VirtualBox:~/Documents/OS$ ./raceCondition
This is the thread PID: This is the thread PID: 139823638443776139823630051072
The number is: 23
The number is doubled: 529

The number is: 23
The number is doubled: 529
(base) theodora@Theodora-VirtualBox:~/Documents/OS$ ./raceCondition
This is the thread PID: 139991577229056
The number is: 23
The number is doubled: 529
This is the thread PID: 139991585621760
The number is: 23
The number is doubled: 529
(base) theodora@Theodora-VirtualBox:~/Documents/OS$

```

Fig 10. Threads without a lock. Source: Theodora's Linux VM 2019

```

// Theodora Tataru – C00231174
// Ana Griga – C00231441
// Thread synchronization with a lock()

#include <pthread.h>
#include <iostream>
#include <mutex>
std::mutex mut;
void* printing(void* aParamater)
{
    mut.lock();
    std::cout << "This is the thread PID: " << pthread_self() << std::endl;
    int number = 23;
    std::cout << "The number is: " << number << std::endl;
    number *= 23;
    std::cout << "The number is doubled: " << number << std::endl;
    mut.unlock();
}
int main()
{
    pthread_t thread1; // creating 2 empty threads
    pthread_t thread2;
    // creating 2 threads for the printing() method
    pthread_create(&thread1, NULL, printing, NULL);
    pthread_create(&thread2, NULL, printing, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
} [25,26].

```

```

OS: ./myThreads
+ guessingGame x OS: ./myThreads
(base) theodora@Theodora-VirtualBox:~/Documents/OS$ g++ mutex.cpp -pthread -o myThreads
(base) theodora@Theodora-VirtualBox:~/Documents/OS$ ./myThreads
This is the thread PID: 139814615901952
The number is: 23
The number is doubled: 529
This is the thread PID: 139814607509248
The number is: 23
The number is doubled: 529
(base) theodora@Theodora-VirtualBox:~/Documents/OS$ ./myThreads
This is the thread PID: 140268514137856
The number is: 23
The number is doubled: 529
This is the thread PID: 140268505745152
The number is: 23
The number is doubled: 529
(base) theodora@Theodora-VirtualBox:~/Documents/OS$

```

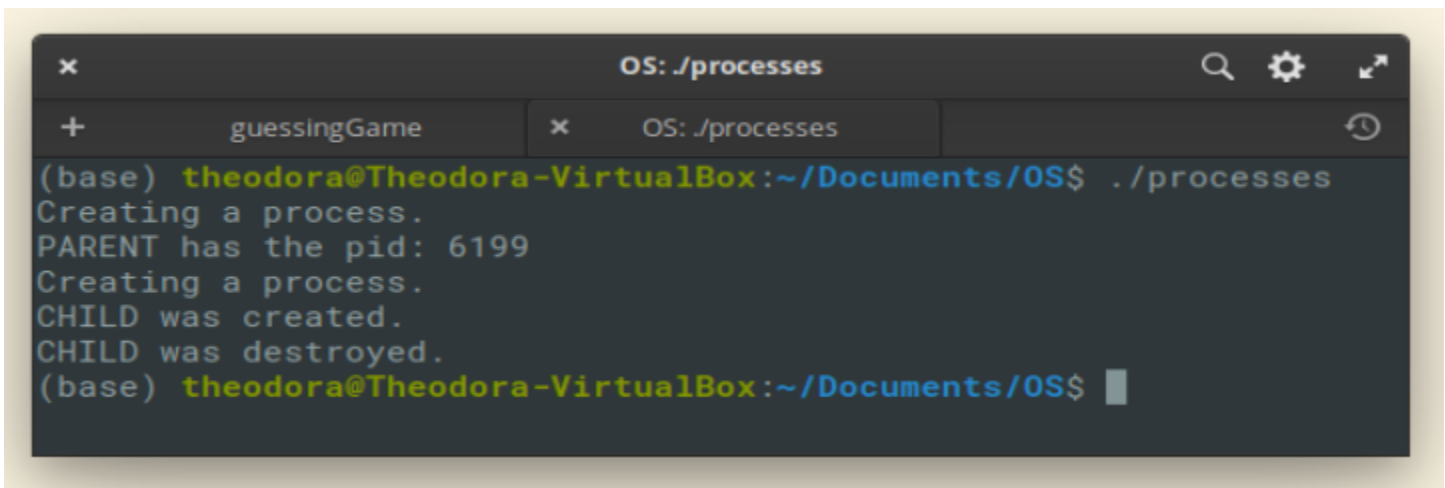
Fig 11. Threads with a lock. Source: Theodora's Linux VM 2019

```
// Theodora Tataru – C00231174
// Ana Griga – C00231441
// Creating/Destroying a Process
```

```
#include <unistd.h> //fork()
#include <iostream>
#include <signal.h> //sig and kill
#include <stdlib.h> //access sleep
```

```
int main()
{
    int pid = fork(); // child will have the PID of 0
    std::cout << "Creating a process." << std::endl; // this will be executed twice, as the child duplicates the
    parent.
    if(pid == 0)
    {
        std::cout << "CHILD was created." << std::endl;
    }

    if(pid > 0)
    {
        std::cout << "PARENT has the pid: " << pid << std::endl;
        sleep(1); // sleep for a second, so the child class is not destroyed so fast
        kill(pid,SIGKILL);
        std::cout << "CHILD was destroyed." << std::endl;
    }
} [25]
```



```
OS: ./processes
+ guessingGame x OS: ./processes
(base) theodora@Theodora-VirtualBox:~/Documents/OS$ ./processes
Creating a process.
PARENT has the pid: 6199
Creating a process.
CHILD was created.
CHILD was destroyed.
(base) theodora@Theodora-VirtualBox:~/Documents/OS$
```

Fig 12. Creating and Destroying Processes **Source:** Theodora's Linux VM 2019

BIBLIOGRAPHY USED IN THE WRITING REPORT

- [1] David Kelly, 2017-2018, *David Kelly Hardware Notes, First Year, Software Development*, IT Carlow.
- [2] Michael Gleeson, 2017-2018, *Lecture 8 – Introduction to processes, First Year, Software Development*, IT Carlow. pp(2-5)
- [3] Microsoft, 05/31/2018, “MicroOperating Systemoft”, Available: https://docs.micrOperating_Systemoft.com/en-us/windows/win32/procthread/about-processes-and-threads, [Accessed: 23.09.2019]
- [4] Abraham Silberschatz, Peter Baer Galvin and Greg Gagne - *Operating Systems (9th Edition)* 2013, *3rd Year OPERATING SYSTEM Slides*, IT Carlow.
- [5] Aaron Kili, 31 March 2017, ” All You Need To Know About Processes in Linux [Comprehensive Guide]”, Available: <https://www.tecmint.com/linux-process-management/> [Accessed: 23.09.2019]
- [6] Luis Colorado, 18 September 2014, “What is the point of the process fork creates being a copy of the parent?”, Available: <https://stackoverflow.com/questions/25876369/what-is-the-point-of-the-process-fork-creates-being-a-copy-of-the-parent>, [Accessed: 24.09.2019]
- [7] Unknown, “Difference between fork() and exec()”, Available: <https://www.geeksforgeeks.org/difference-fork-exec/>, Accessed: 24.09.2019]
- [8] Nikita Ishkov, 2015, “A complete guide to Linux Process scheduling”, University of Tampere, pp(2-4)
- [9] Evi Nemeth, Garth Snyder, Trent R. Hein, Ben Whaley, *Unix and Linux System Administration Handbook 4th edition*, 2010, pp(123-128)
- [10] Chandrima Sarkar, “Scheduler in Linux”, Available: https://www.cs.montana.edu/~chandrima.sarkar/AdvancedOPERATING_SYSTEM/CSCI560_Proj_main/index.html. [Accessed: 27.09.2019]
- [11] Red Hat, “CPU Scheduling”, Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-cpu-scheduler, [Accessed: 27.09.2019]
- [12] John O’Gorman, 2001, *Operating Systems with Linux*, Wiltshire
- [13] Gary Sims, 2016, *Processes and Threads*, https://www.youtube.com/watch?v=h_HwkHobfs0 [Accessed: 03.10.2019]
- [14] Kishlay Verma, “Mutex lock for Linux Thread Synchronization”, Available: <https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/>, [Accessed: 03.10.2019]
- [15] Jonathan Corbet and Alessandro Rubini, “Linux Device Drivers, Second Edition”, Available: <https://www.oreilly.com/library/view/linux-device-drivers/0596000081/ch03s07.html>, [Accessed: 04.09.2019]
- [16] Ricky Barnes, 10 October 2018, “Critical Section Problem”, Available: <https://www.tutorialspoint.com/critical-section-problem>, [Accessed: 04.10.2019]
- [17] Wikipedia,” Critical Section Problem”, Available: https://en.wikipedia.org/wiki/Busy_waiting, [Accessed: 04.10.2019]
- [18] Abhishek Ahlawat,” Process Synchronization”, 2019, Available: <https://www.studytonight.com/operating-system/process-synchronization>, [Accessed: 04.10.2019]
- [19] Unknown, “Introduction of Deadlock in Operating System”, Available: <https://www.geeksforgeeks.org/introduction-of-deadlock-in-operating-system/> [Accessed: 05.10.2019]
- [20] Manuel Schulte, 23 November 2014, “Why does deadlock not occur in Unix?”, Available: <https://www.quora.com/Why-does-deadlock-not-occur-in-Unix>, [Accessed: 05.10.2019]
- [21] Saloni Gupta, ”Starvation and Aging in Operating Systems”, Available: <https://www.geeksforgeeks.org/starvation-and-aging-in-operating-systems/>, [Accessed: 05.10.2019]
- [22] Unknown, “top command in Linux with Examples”, Available: <https://www.geeksforgeeks.org/top-command-in-linux-with-examples/>, [Accessed: 05.10.2019]
- [23] Unknown, 21 May 2004, “Processes: A Brief Introduction”, 2004 - 2006. The Linux Information Project, Available: <http://www.linfo.org/process.html>, [Accessed: 07.09.2019]
- [24] Unknown, “Kernel”, Available: <https://www.techopedia.com/definition/3277/kernel>, [Accessed: 13.10.2019]

BIBLIOGRAPHY USED IN THE CODE

[25] Authors: Tataru Theodora, Ana Griga with the co-op: Martin O’Sullivan and Robert Kamenicky

[26] Kishlay Verma, “Mutex lock for Linux Thread Synchronization”, Available: <https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/>, [Accessed: 15.10.2019]

Collaboration Form

Student Name: Theodora Tataru Ana Griga
Student No: C00231174 C00231441

The following is a total list of all students I collaborated with whilst conducting research for this assignment:

Student Name: Theodora Tataru
Student Number: C00231174
% of work: 50%

Student Name: Ana Griga
Student Number: C00231441
% of work: 50%

The following is a printout of the code I submitted for this assignment. Each line or block of code includes the following;

- 1. A line number**
- 2. A comment**
- 3. The authors (*students*) name**