SECR2043 - OPERATING SYSTEMS (Lab Exercise)

Written by: Dr. Farkhana Muchtar

LAB 4: LINUX PROCESS MONITORING AND MANAGEMENT

Introduction of Process Management

1 Overview

The goal of this lab exercise is to familiarize students with the concepts and techniques of Linux process management. By offering hands-on experience, students will gain a deeper understanding of how operating systems manage and control processes.

This lab exercise is organized into two parts. The first part (**Section 2 & 3**), concentrates on teaching students to utilize built-in Linux tools and commands for process monitoring and management. To streamline this portion of the exercise, a dummy background process will be created (**Section 2**) to serve as an example, which will be examined and simulated using Linux commands for process monitoring and management.

In the other hand, the second part (**Section 4**) of this lab exercise, provides students with hands-on experience using fork and other system calls for process monitoring and management across various programs. By showcasing practical examples in both C and Python programming languages, students will gain a solid understanding of how to apply fork and other system calls for effective process monitoring and management.

2 Creating a Dummy Process for Process Monitoring and Management Exercises

Purpose: We are creating this dummy program to have multiple instances of two child processes running in the background. It is intended to provide a suitable scenario for practicing process monitoring and management commands, such as `ps`, `top`, `htop`, and `killall` in **Section 3**.

Step 1: Write dummy process programs.

The dummy process consists of three C source files: `parent.c`, `child1.c`, and `child2.c`. `parent.c` is responsible for creating two child processes that will run `child1` and `child2` programs. These child processes will run in the background, and each will create additional child processes.

1. Write a main program called `parent.c` using Nano editor:

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

```
void handle_signal(int sig) {
    printf("Main process (ID: %d) received signal: %d. Terminating...\n", getpid(),
sig);
    exit(0);
}

void create_child_processes(const char *child_name, int count) {
    for (int i = 0; i < count; ++i) {
        pid_t pid = fork();

        if (pid == 0) {
            execlp(child_name, child_name, NULL);
            perror("execlp");
            exit(1);
        }
    }
}

int main() {
    signal(SIGTERM, handle_signal);
    signal(SIGINT, handle_signal);

    pid_t main_pid = getpid();
    printf("Main process ID: %d\n", main_pid);

    pid_t pid = fork();

    if (pid == 0) {
        // First child process
        create_child_processes("./child1", 2);
        sleep(1800); // Sleep for 30 minutes
        exit(0);
    } else {
        pid = fork();

        if (pid == 0) {
            // Second child process
            create_child_processes("./child2", 3);
            sleep(1800); // Sleep for 30 minutes
            exit(0);
        }
    }

    // Make the main process run in the background
    setsid();
    sleep(1800); // Sleep for 30 minutes

    return 0;
}
```

2. Write the first child program, named `child1.c` using Nano editor.

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

void handle_signal(int sig) {
    printf("Child1 process (ID: %d) received signal: %d. Terminating...\n", getpid(),
sig);
    exit(0);
}

int main() {
    signal(SIGTERM, handle_signal);
    signal(SIGINT, handle_signal);

    printf("I am child1 with process ID: %d and parent process ID: %d\n", getpid(),
getppid());
    sleep(1800); // Sleep for 30 minutes
    return 0;
}
```

3. Write the second child program, named `child2.c` using Nano editor.

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

void handle_signal(int sig) {
    printf("Child2 process (ID: %d) received signal: %d. Terminating...\n", getpid(),
sig);
    exit(0);
}

int main() {
    signal(SIGTERM, handle_signal);
    signal(SIGINT, handle_signal);

    printf("I am child2 with process ID: %d and parent process ID: %d\n", getpid(),
getppid());
    sleep(1800); // Sleep for 30 minutes
    return 0;
}
```

Step 2: Compile dummy process codes using GCC.

```
gcc parent.c -o parent
gcc child1.c -o child1
gcc child2.c -o child2
```

Step 3: To run the dummy process in the background, execute the compiled `parent` binary followed by an ampersand (&):

```
./parent &
```

and press `Enter` key, two times.

The ampersand (&) tells the shell to run the process in the background. This will start the parent process, which creates two child processes running `child1` and `child2` programs. These child processes will run in the background for 30 minutes and print messages periodically.

You will use this dummy process to practice Linux process monitoring and management commands such as `ps`, `top`, `htop`, `pstree`, `kill`, `pkill`, and `killall` in **Section 3**.

Remember, when running process monitoring and management commands, you may need to look for "`parent`", "`child1`" and "`child2`" processes in the output to find the processes created by the dummy process.

3 Understanding Linux Processes, Process Monitoring, Process Management, and Process Termination

In this section, we will explore Linux processes, process management, and process termination using tools like `top`, `htop`, `ps`, `kill`, `killall`, and `pkill`. We will also provide more detailed usage of these commands, discuss available options, and demonstrate how to find and manage specific processes. This tutorial will help you gain a deeper understanding of Linux processes and their management.

3.1 Exercise 1: Using top to Monitor Processes.

Purpose: This tutorial aims to provide an extensive guide on using the `top` command in Linux for process monitoring, using the dummy process created in a previous tutorial. The `top` command is widely used for displaying dynamic real-time information about running processes.

Prerequisites: Before starting this tutorial, make sure you have the dummy process running in the background.

Step 1: Basic usage of top command
To start using the `top` command, simply type `top` in the terminal:

```
top
```

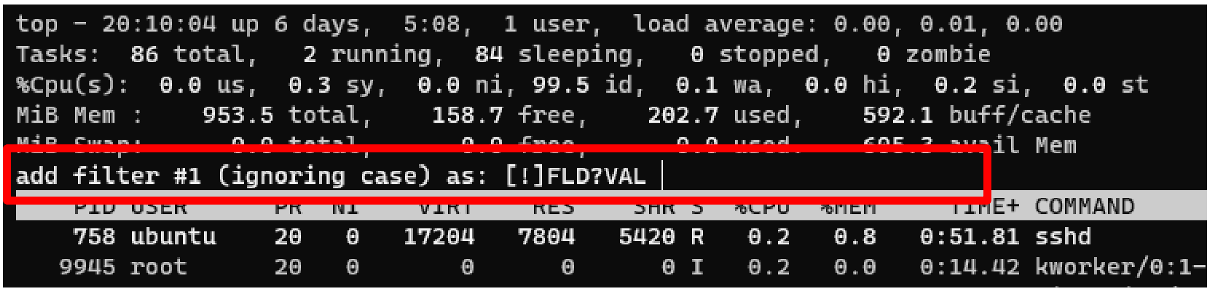
This will display a live, real-time view of the processes running on your system, including their process ID (PID), user, CPU usage, memory usage, and more.

Step 2: Filtering processes by name

To filter the list of processes displayed by top, press `o` followed by the filter expression. In our case, we will filter by the process name, either "`child1`" or "`child2`":

```
Press o key
```

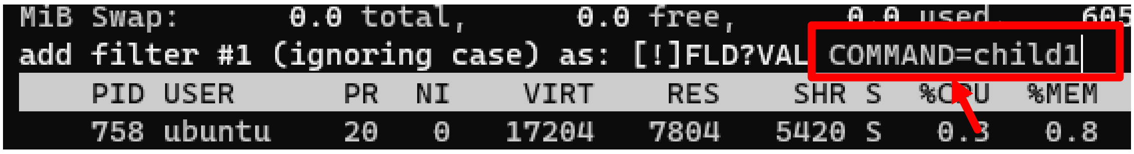
Filter expression will appear like this:



```
top - 20:10:04 up 6 days,  5:08,  1 user,  load average: 0.00, 0.01, 0.00
Tasks:  86 total,   2 running,  84 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.3 sy,  0.0 ni, 99.5 id,  0.1 wa,  0.0 hi,  0.2 si,  0.0 st
MiB Mem :    953.5 total,    158.7 free,    202.7 used,    592.1 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.    605.3 avail Mem
add filter #1 (ignoring case) as: [!]FLD?VAL
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  758 ubuntu    20   0   17204   7804   5420 R   0.2   0.8   0:51.81 sshd
 9945 root      20   0       0      0      0 I   0.2   0.0   0:14.42 kworker/0:1-
```

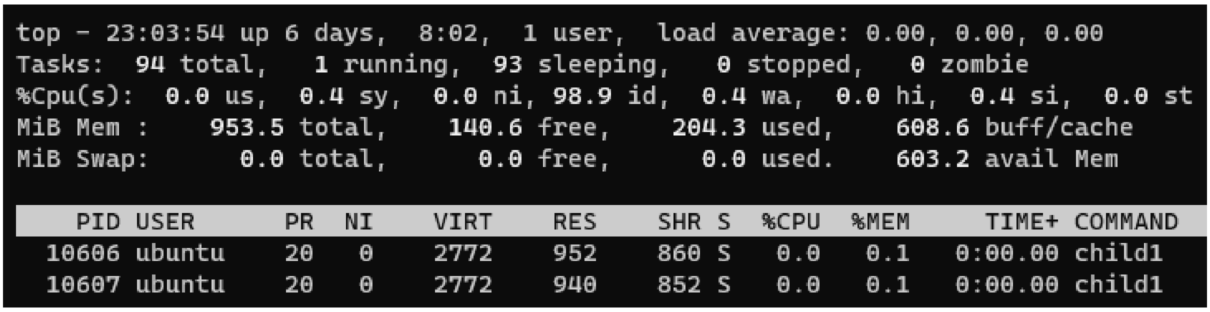
Inside filter expression prompt, insert this expression:

```
COMMAND=child1
```



```
MiB Swap:      0.0 total,      0.0 free,      0.0 used.    605
add filter #1 (ignoring case) as: [!]FLD?VAL COMMAND=child1
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM
  758 ubuntu    20   0   17204   7804   5420 S   0.3   0.8
```

Press `Enter` to apply the filter.



```
top - 23:03:54 up 6 days,  8:02,  1 user,  load average: 0.00, 0.00, 0.00
Tasks:  94 total,   1 running,  93 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.4 sy,  0.0 ni, 98.9 id,  0.4 wa,  0.0 hi,  0.4 si,  0.0 st
MiB Mem :    953.5 total,    140.6 free,    204.3 used,    608.6 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.    603.2 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
10606 ubuntu    20   0    2772    952    860 S   0.0   0.1   0:00.00 child1
10607 ubuntu    20   0    2772    940    852 S   0.0   0.1   0:00.00 child1
```

To clear the filter, press = key, followed by `Enter`.

Step 3: Sorting processes by a specific column

By default, `top` sorts processes by their CPU usage. To change the sorting column, press the < or > keys. For example, to sort processes by memory usage, navigate to the %MEM column using the < or > keys.

Step 4: Displaying and hiding columns

To customize the columns displayed by `top`, press `f` to open the Fields Management screen. Use the arrow keys to navigate through the list of columns, and press `d` or `s` to toggle the display of a specific column. Press `Enter` to apply the changes.

Step 5: Changing the update interval

By default, `top` updates the process information every 3 seconds. To change the update interval, press `d` followed by the desired number of seconds, for example 0.5 seconds, and then press Enter.

Step 6: Sending signals to processes

Before proceeding with this step, redo **Step 2** of the tutorial (without pressing = key), to display only the `child1` process in the list of processes.

`top` allows you to send signals to processes directly from its interface. To do so, press `k` followed by the process ID (PID) of the process you want to send a signal to. By default, `top` will send the SIGTERM (15) signal, which asks the process to terminate gracefully. To send a different signal, type the desired signal number after entering the PID. Press `Enter` to send the signal.

Conclusion: In this tutorial, you learned how to use the `top` command for process monitoring with the dummy process. You can now use these techniques to monitor and manage processes in a Linux environment.

3.2 Exercise 2: Using htop for Process Monitoring

Purpose: This tutorial aims to provide an extensive guide on using the `htop` command in Linux for process monitoring, using the dummy process created in a previous tutorial. The `htop` command is an interactive process viewer that offers advanced features and a user-friendly interface compared to `top`.

Prerequisites:
Before starting this tutorial, make sure you have the dummy process running in the background.

```
killall parent child1 child2
./parent &
```

If `htop` is not installed on your system, you can install it using your package manager. (e.g., `sudo apt install htop` for Ubuntu/Debian).

Step 1: Basic usage of `htop` command

To start using the `htop` command, simply type `htop` in the terminal:

```
htop
```

This will display a live, real-time view of the processes running on your system, including their process ID (PID), user, CPU usage, memory usage, and more.

Step 2: Filtering processes by name

To filter the list of processes displayed by `htop`, press `F4` to open the filter prompt. In our case, we will filter by the process name, either "`child1`" or "`child2`":

```
[child1]
```

Press `Enter` to apply the filter. To clear the filter, press `F4` and delete the filter expression.

Step 3: Sorting processes by a specific column

By default, `htop` sorts processes by their CPU usage. To change the sorting column, click on the desired column header with your mouse or use the `F6` key to open the sort options, then use the arrow keys to navigate and press `Enter` to apply.

Step 4: Displaying and hiding columns

To customize the columns displayed by `htop`, press `F2` to open the Setup screen. Navigate to the "Columns" tab using the arrow keys. Select the columns you want to display or hide by navigating to them and pressing the Spacebar. Press `F10` to save the changes and exit the Setup screen.

Step 5: Sending signals to processes

`htop` allows you to send signals to processes directly from its interface. To do so, use the arrow keys to navigate to the desired process, and press `F9` to open the signal selection screen. By default, htop will send the `SIGTERM(15)` signal, which asks the process to terminate gracefully. To send a different signal, navigate to the desired signal and press `Enter`.

Step 6: Exiting `htop`

To exit `htop`, press `F10` or the `Q` key.

Conclusion: In this tutorial, you learned how to use the `htop` command for process monitoring with the dummy process. You can now use these techniques to monitor and manage processes in a Linux environment more efficiently and effectively.

3.3 Exercise 3: Using ps Command for Process Monitoring

Purpose: This tutorial aims to provide an extensive guide on using the `ps` command in Linux for process monitoring, using the dummy process created in the previous tutorial. The `ps` command is widely used for obtaining information about the currently running processes.

Prerequisites: Before starting this tutorial, make sure you have the dummy process running in the background.

```
killall parent child1 child2
./parent &
```

Step 1: Basic usage of `ps` command

The `ps` command without any options displays information about the current shell's processes. To view all running processes, use the `-e` option:

```
ps -e
```

Step 2: Filtering processes by name

To filter the list of processes by their name, you can use `grep`. In our case, we will search for our `"child1"` and `"child2"` processes:

```
ps -e | grep "child1"
ps -e | grep "child2"
```

Step 3: Displaying additional process information

To display more information about the processes, you can use the `–f` option for full-format listing:

```
ps -ef | grep "child1"
ps -ef | grep "child2"
```

The output will include additional information such as the process owner, parent process ID (PPID), start time, and more.

Step 4: Customizing output columns

To customize the output columns, use the -o option followed by a comma-separated list of column names. For example, to display the process ID (PID), user, CPU usage, memory usage, and command, use the following command:

```
ps -eo pid,user,pcpu,pmem,cmd | grep "child1"
ps -eo pid,user,pcpu,pmem,cmd | grep "child2"
```

Step 5: Sorting output by a specific column

To sort the output by a specific column, use the `--sort` option followed by the column name. For example, to sort the output by CPU usage:

```
ps -eo pid,user,pcpu,pmem,cmd --sort=pcpu | grep "child1"
ps -eo pid,user,pcpu,pmem,cmd --sort=pcpu | grep "child2"
```

Step 6: Displaying process hierarchy

To display the process hierarchy, use the `--forest` option:

```
ps -ef --forest | grep "child1"
ps -ef --forest | grep "child2"
ps -ef --forest | grep -E "parent|child"
```

This will show the parent and child processes in a tree-like structure, making it easy to visualize the process relationships.

Conclusion: In this tutorial, you learned how to use the `ps` command for process monitoring with the dummy process. You can now use these techniques to monitor and manage processes in a Linux environment.

3.4 Using pstree Command for Process Monitoring

Purpose: This tutorial aims to provide an extensive guide on using the `pstree` command in Linux for process monitoring, using the dummy process created in a **Section 2**. The `pstree` command is used to display the process tree of a system in a hierarchical format.

Prerequisites: Before starting this tutorial, make sure you have the dummy process running in the background.

```
killall parent child1 child2
./parent &
```

Step 1: Basic usage of `pstree` command

To start using the `pstree` command, simply type `pstree` in the terminal:

```
pstree
```

This will display the process tree of your system, with each process represented as a node in the tree. Processes are organized hierarchically, with child processes branching from their parent processes.

Step 2: Displaying PIDs with `pstree`

By default, `pstree` only shows the process names. To display the process IDs (PIDs) as well, use the -p option:

```
pstree -p
```

This will show the PIDs of each process in parentheses alongside their names.

Step 3: Filtering the process tree by a specific process

To display only the process tree for a specific process and its descendants, you can use the `-s` option followed by the PID of the process. In our case, we will filter by the PID of the parent process created in the dummy process tutorial:

```
pstree -p -s <parent_pid>
```

Replace <parent_pid> with the actual PID of the parent process. This will show only the `parent` process and its child processes, "`child1`" and "`child2`".

Step 4: Displaying the process tree in ASCII format

By default, `pstree` uses special characters to draw the tree structure. To display the tree using ASCII characters, use the `-A` option:

```
pstree -A
```

This can be useful if you are using a terminal that does not support special characters.

Step 5: Showing the number of threads for each process

To display the number of threads for each process in the tree, use the `-c` option:

```
pstree -c
```

This will show the number of threads in curly braces {} next to each process.

Conclusion: In this tutorial, you learned how to use the `pstree` command for process monitoring with the dummy process. You can now use these techniques to monitor and manage processes in a Linux environment more effectively.

3.5 Using pgrep Command for Process Monitoring

Purpose: This tutorial aims to provide an extensive guide on using the `pgrep` command in Linux for process monitoring, using the dummy process created in a previous tutorial. The `pgrep` command is used to search for processes based on their names and other attributes.

Prerequisites: Before starting this tutorial, make sure you have the dummy process running in the background.

```
killall parent child1 child2
./parent &
```

Step 1: Basic usage of `pgrep` command

To use the `pgrep` command, simply type `pgrep` followed by the name of the process you are looking for:

```
pgrep child1
```

This will return the PIDs of all the "`child1`" processes running on your system.

Step 2: Searching for multiple processes

To search for multiple processes at once, you can use regular expressions. For example, to search for both "`child1`" and "`child2`" processes, use the following command:

```
pgrep 'child[12]'
```

This will return the PIDs of all "`child1`" and "`child2`" processes running on your system.

Step 3: Displaying process names with PIDs

By default, `pgrep` only shows the PIDs of the matched processes. To display the process names as well, use the `-l` option:

```
pgrep -l 'child[12]'
```

This will show the PIDs alongside the process names for each matched process.

Step 4: Searching for processes by user

To search for processes by the user running them, use the `-u` option followed by the user name or user ID. For example, to search for "`child1`" processes run by the current user, use the following command:

```
pgrep -u $(whoami) child1
```

This will return the PIDs of all "`child1`" processes running under the current user.

Step 5: Displaying the full command line of the matched processes

To display the full command line of the matched processes, use the `-a` option:

```
pgrep -a 'child[12]'
```

This will show the PIDs alongside the full command line for each matched process.

Conclusion: In this tutorial, you learned how to use the `pgrep` command for process monitoring with the dummy process. The `pgrep` command is a powerful tool for searching and filtering processes based on their names and other attributes. You can now use these techniques to monitor and manage processes in a Linux environment more effectively.

3.6 Adjusting Process Priorities with nice and renice Commands

Purpose: This tutorial aims to provide an extensive guide on using the `nice` and `renice` commands in Linux for adjusting the priorities of processes, using the dummy process created in a previous tutorial.

Prerequisites: Before starting this tutorial, make sure you have the dummy process running in the background.

```
killall parent child1 child2
./parent &
```

Step 1: Understanding the nice value and process priorities

In Linux, each process has a "nice" value, which ranges from -20 (highest priority) to 19 (lowest priority). The default "nice" value for a new process is 0. The `nice` command is used to set the "nice" value of a process when starting it, while the `renice` command is used to change the "nice" value of an already running process.

Step 2: Starting a process with a specific nice value

To start a process with a specific "nice" value, use the nice command followed by the `-n` option and the desired "nice" value. For example, to start a new "`child1`" process with a "nice" value of 10, use the following command:

```
nice -n 10 ./child1 &
```

This will start the "child1" process with a lower priority.

Step 3: Viewing the nice value of a process

To view the "nice" value of a process, use the ps command with the `-o` option to customize the output format. For example, to display the PID, "nice" value, and command name for the "`child1`" processes, use the following command:

```
ps -o pid,nice,comm -C child1
```

Step 4: Changing the nice value of a running process

To change the "nice" value of a running process, use the `renice` command followed by the desired "nice" value and the PID of the process. For example, to change the "nice" value of a "`child1`" process with the PID 12345 to 5, use the following command:

```
sudo renice 5 12345
```

Note that you need to use sudo because only the root user can increase the priority (lower the "nice" value) of a process.

Step 5: Verifying the nice value change

After changing the "nice" value, use the `ps` command again to verify the change:

```
ps -o pid,nice,comm -C child1
```

Conclusion: In this tutorial, you learned how to use the `nice` and `renice` commands to adjust process priorities with the dummy process. Adjusting process priorities can be useful for managing system resources and ensuring that important processes receive enough CPU time. You can now use these techniques to monitor and manage process priorities in a Linux environment more effectively.

3.7 Monitoring Process File and Network Usage with lsof Command

Purpose: This tutorial aims to provide an extensive guide on using the `lsof` command in Linux for monitoring file and network usage by processes, using the dummy process created in a previous tutorial.

Prerequisites: Before starting this tutorial, make sure you have the dummy process running in the background.

Step 1: Understanding the `lsof` command

`lsof` stands for "list open files" and is a powerful command-line tool that provides information about files opened by processes. In Unix and Linux, everything is treated as a file, including network sockets and devices. Thus, `lsof` can also provide information about network connections and devices used by processes.

Step 2: Listing open files for a specific process

To list the open files for a specific process, use the `lsof` command followed by the `-p` option and the PID of the process. Use `ps -ef | grep <process_name>` to get the PID of the process.

For example, to list the open files for a "`child1`" process with the PID 12345, use the following command:

```
lsof -p 12345
```

This will display a list of open files, including regular files, directories, devices, and network connections, used by the process.

Step 3: Listing open files for processes with a specific command name

To list open files for all processes with a specific command name, use the `lsof` command followed by the `-c` option and the command name.

For example, to list open files for all "`child1`" processes, use the following command:

```
lsof -c child1
```

This will display open files for all instances of the "`child1`" process.

Step 4: Listing open files for a specific user

To list open files for all processes owned by a specific user, use the `lsof` command followed by the `-u` option and the username. For example, to list open files for all processes owned by the user "`ubuntu`", use the following command:

```
lsof -u ubuntu
```

Step 5: Filtering the `lsof` output

`lsof` provides various options to filter its output based on criteria such as file type, file descriptor, and protocol. For example, to list only network connections for the "`child1`" processes, use the following command:

```
lsof -c child1 -i
```

To list only open TCP connections for the "`child1`" processes, use the following command:

```
lsof -c child1 -i tcp
```

Conclusion: In this tutorial, you learned how to use the `lsof` command to monitor file and network usage by the dummy process. The `lsof` command is a powerful tool for troubleshooting and investigating process activities in a Linux environment. By understanding the capabilities of `lsof`, you can more effectively monitor and manage processes in your system.

3.8 Tracing System Calls and Signals with strace Command

Purpose: This tutorial aims to provide an extensive guide on using the `strace` command in Linux for tracing system calls and signals made by the dummy process created in a previous tutorial.

Prerequisites: Before starting this tutorial, make sure you have the dummy process running in the background.

```
killall parent child1 child2
./parent &
```

Step 1: Understanding the `strace` command

`strace` is a powerful command-line tool used for tracing system calls and signals made by a process. It can be used for debugging, troubleshooting, and understanding the inner workings of a process. By examining the system calls made by a process, you can gain insights into its behavior and resource usage.

Step 2: Attaching `strace` to a running process

To attach `strace` to a running process, use the `strace` command followed by the `-p` option and the PID of the process.

For example, to attach `strace` to a "`child1`" process with the PID 12345, use the following command:

```
strace -p 12345
```

This will display a real-time trace of system calls made by the process, along with their arguments and return values.

Step 3: Tracing a new process with `strace`

To trace a new process with `strace`, simply prepend the `strace` command to the command used to start the process. For example, to trace a new "child1" process, use the following command:

```
strace ./child1
```

This will display a trace of system calls made by the new process from the beginning of its execution.

Step 4: Limiting the output of `strace`

`strace` provides various options to limit its output and display only the information you're interested in. For example, to display only file-related system calls for a "`child1`" process with the PID 12345, use the following command:

```
strace -p 12345 -e trace=file
```

To display only network-related system calls for the same process, use the following command:

```
strace -p 12345 -e trace=network
```

Step 5: Saving `strace` output to a file

To save the output of `strace` to a file, use the `-o` option followed by the output file name. For example, to save the output of tracing a "`child1`" process with the PID 12345 to a file named "`trace.txt`", use the following command:

```
strace -p 12345 -o trace.txt
```

Conclusion: In this tutorial, you learned how to use the `strace` command to trace system calls and signals made by the dummy process. The `strace` command is a valuable tool for understanding the behavior of processes, identifying bottlenecks, and troubleshooting issues in a Linux environment. By mastering the use of `strace`, you can gain deeper insights into your system's processes and improve your ability to monitor and manage them effectively.

3.9 Monitoring System Performance with vmstat Command

Purpose: This tutorial aims to provide an extensive guide on using the `vmstat` command in Linux for monitoring system performance, including memory, processes, and CPU usage, while the dummy process created in a previous tutorial is running in the background.

Prerequisites: Before starting this tutorial, make sure you have the dummy process running in the background.

```
killall parent child1 child2
./parent &
```

Step 1: Understanding the `vmstat` command

`vmstat` (virtual memory statistics) is a command-line tool in Linux that displays information about system processes, memory, paging, block I/O, traps, and CPU activity. It provides an overview of how the system resources are being utilized, which can help you identify performance bottlenecks and optimize resource usage.

Step 2: Running the `vmstat` command

To run the `vmstat` command, simply enter `vmstat` in the terminal, followed by an optional interval (in seconds) for updating the display:

```
vmstat 5
```

This command will display the `vmstat` output every 5 seconds. If no interval is provided, `vmstat` will display the average values since the last reboot.

Step 3: Interpreting the `vmstat` output

The `vmstat` output is organized into several columns, each representing a different aspect of system performance. Some of the key columns are:

- **procs:** Displays the number of processes in different states, such as running (r), waiting (w), or blocked (b).
- **memory:** Shows information about memory usage, such as total (swpd), free (free), buffer (buff), and cache (cache) memory.
- **swap:** Provides information about swap usage, such as swap-ins (si) and swap-outs (so).
- **io:** Displays information about input/output operations, such as blocks received (bi) and blocks sent (bo).
- **system:** Shows the number of interrupts (in) and context switches (cs) per second.
- **cpu:** Provides information about CPU usage, such as user (us), system (sy), idle (id), waiting (wa), and steal (st) time.

Step 4: Analyzing the impact of the dummy process

While the dummy process is running in the background, you can use `vmstat` to monitor how it affects system performance. Pay attention to the `procs`, `memory`, and `cpu` columns, as they can help you identify any resource bottlenecks caused by the dummy process. For example, an increase in the number of running processes (r) or a decrease in free memory (free) could indicate that the dummy process is consuming a significant amount of system resources.

Step 5: Monitoring system performance over time

To monitor system performance over a longer period, you can redirect the `vmstat` output to a file by using the following command:

```
vmstat 5 > vmstat_output.txt &
```

This command will save the `vmstat` output every 5 seconds to a file named "`vmstat_output.txt`". To stop the command, use the kill command followed by the process ID of the `vmstat` command.

Conclusion: In this tutorial, you learned how to use the `vmstat` command to monitor system performance while the dummy process is running in the background. By using `vmstat` effectively, you can gain insights into your system's resource usage, identify performance bottlenecks, and optimize your system for better performance.

3.10 Terminating Processes with kill, pkill, and killall Commands

Purpose: This tutorial aims to provide an extensive guide on using the `kill`, `pkill`, and `killall` commands in Linux to terminate processes, with a focus on the dummy process created in a previous tutorial.

Prerequisites: Before starting this tutorial, make sure you have the dummy process running in the background.

```
killall parent child1 child2
./parent &
```

Step 1: Understanding process termination commands

- **kill:** This command sends a signal to a process, usually to terminate it. The most common signal is SIGTERM (15), which asks the process to terminate gracefully, allowing it to perform cleanup operations before exiting. If a process does not respond to SIGTERM, you can use the SIGKILL (9) signal, which forcefully kills the process.
- **pkill:** This command sends a signal to processes based on their names or other attributes, such as the process owner. Like kill, the default signal is SIGTERM.
- **killall:** This command sends a signal to all processes with the specified name, effectively terminating all instances of that process. The default signal is also SIGTERM.

Step 2: Using the `kill` command

To use the `kill` command, first determine the process ID (PID) of the dummy process by using the ps command:

```
ps -ef | grep 'child1\|child2'
```

This command will display the PIDs of both `child1` and `child2` processes. To terminate a process using the kill command, enter the following:

```
kill -15 <PID>
```

Replace <PID> with the actual PID of the process you want to terminate. For example:

```
kill -15 4381
```

If the process does not respond to the SIGTERM signal, use the SIGKILL signal:

```
kill -9 <PID>
```

For example:

```
kill -9 4382
```

Step 3: Using the `pkill` command

The `pkill` command allows you to terminate processes by their names or other attributes. To terminate the `child1` process, enter the following command:

```
pkill -15 child1
```

To terminate the `child2` process, use the following command:

```
pkill -15 child2
```

If the processes do not respond to the SIGTERM signal, use the SIGKILL signal:

```
pkill -9 child1
pkill -9 child2
```

Step 4: Using the `killall` command

The `killall` command terminates all instances of the specified processes. To terminate all instances of the child1 and child2 processes, enter the following commands:

```
killall -15 child1
killall -15 child2
```

If the processes do not respond to the SIGTERM signal, use the SIGKILL signal:

```
killall -9 child1
killall -9 child2
```

Conclusion: In this tutorial, you learned how to use the `kill`, `pkill`, and `killall` commands to terminate processes, focusing on the dummy process running in the background. These commands provide powerful and flexible options for managing and terminating processes in a Linux environment.

Written by: Dr. Farkhana Muchtar

4 Linux Process Management using Fork and Other System Calls

Objective:

The objective of these lab exercises is to familiarize students with Linux process management using fork and other system call functions. Students will learn the basics of process creation, communication, and control in a Linux environment.

Explanation:

This lab exercise will cover the following concepts:

1. Process creation using the fork system call
   a. Understanding the fork system call and its behaviour.
   b. Creating child processes and distinguishing between parent and child processes.

2. Parent and child process communication
   a. Using regular pipes and named pipes (FIFOs) for inter-process communication (IPC).
   b. Sharing data between processes using shared memory.
   c. Communicating using file descriptors and signals.

3. Process control using wait, exec, and other system call functions
   a. Ensuring the parent process waits for the child process to complete before terminating.
   b. Replacing the current process image with a new process image.
   c. Synchronizing parent and child processes using semaphores.
   d. Managing resource limits in Linux systems

Students will learn:

1. The fundamentals of process management in Linux
2. How to create and manage processes using fork and other system call functions
3. How to perform inter-process communication (IPC) between parent and child processes
4. How to control and terminate processes in a Linux environment

By completing these exercises, students will gain a comprehensive understanding of Linux processes, process management, and various techniques for inter-process communication and synchronization. They will be equipped with the skills to create complex programs involving parent and child processes in both C and Python programming languages.

Each exercise will include both C and Python code examples, along with an explanation of the concepts being covered. It is advisable to implement all the exercises inside `/home/<username>/workspace` directory.

4.1 Exercise 1: Basic Fork Usage

Objective: Understand the fork system call and learn how to create child processes using fork.

Explanation: In this exercise, students will learn how to create new processes using the fork system call in both C and Python. They will understand the difference between parent and child processes and how to identify them in their programs.

4.1.1 Basic Fork Usage in C Code

Step 1: Write a new C code, named `basic_fork_usage.c` using Nano editor.

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

int main() {
    pid_t pid;

    pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        return 1;
    } else if (pid == 0) {
        printf("Child process with PID: %d\n", getpid());
    } else {
        printf("Parent process with PID: %d, child PID: %d\n", getpid(), pid);
    }

    return 0;
}
```

Step 2: Compile the code using GCC.

```
gcc basic_fork_usage.c -o basic_fork_usage
```

Step 3: Run the compiled program.

```
./basic_fork_usage
```

4.1.2 Basic Fork Usage in Python Code

Step 1: Write a new Python code, named `basic_fork_usage.py` using Nano editor.

```
import os

pid = os.fork()

if pid < 0:
    raise Exception("Fork failed")
elif pid == 0:
    print("Child process with PID:", os.getpid())
else:
    print("Parent process with PID:", os.getpid(), "child PID:", pid)
```

Step 2: Run Python code using Python interpreter.

```
python basic_fork_usage.py
```

4.2 Exercise 2: Fork and Wait

Objective: Learn how to use the `fork` and `wait` system calls to manage the execution order of parent and child processes in a simple scenario.

Explanation: In this exercise, students will use the `fork` and `wait` system calls to create a single child process and ensure that the parent process waits for the child process to complete. The child process will perform a simple task and exit, while the parent process waits for it to finish before resuming its own tasks. This exercise will help beginners understand the importance of process synchronization and how to manage the execution order of parent and child processes.

4.2.1 Fork and Wait in C Code

Step 1: Write a new C code, named `fork_and_wait.c` using Nano editor.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid;

    pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        return 1;
    } else if (pid == 0) {
        // Child process
        printf("Child process %d is running\n", getpid());
        sleep(2);
        printf("Child process %d is exiting\n", getpid());
        return 0;
    } else {
        // Parent process
        printf("Parent process %d is waiting for child process %d\n", getpid(), pid);
        wait(NULL);
        printf("Parent process %d resumes execution\n", getpid());
    }

    return 0;
}
```

Step 2: Compile the code using GCC.

```
gcc fork_and_wait.c -o fork_and_wait
```

Step 3: Run the compiled program.

```
./fork_and_wait
```

4.2.2 Fork and Wait in Python Code

Step 1: Write a new Python code, named `fork_and_wait.py` using Nano editor.

```
import os
import time

def main():
    pid = os.fork()

    if pid < 0:
        raise Exception("Fork failed")
    elif pid == 0:
        # Child process
        print(f"Child process {os.getpid()} is running")
        time.sleep(2)
        print(f"Child process {os.getpid()} is exiting")
    else:
        # Parent process
        print(f"Parent process {os.getpid()} is waiting for child process {pid}")
        os.waitpid(pid, 0)
        print(f"Parent process {os.getpid()} resumes execution")

if __name__ == "__main__":
    main()
```

Step 2: Run Python code using Python interpreter.

```
python fork_and_wait.py
```

4.3 Exercise 3: Fork and Exec

Objective: Understand the `exec` family of functions and learn how to launch external programs and commands using `fork` and `exec`.

Explanation: In this exercise, students will learn how to use `fork` and `wait` to create child processes and manage their execution order, as well as how to use `fork` and `exec` to create child processes that run external programs with command -line arguments.

These exercises will help students to gain hands-on experience in managing process execution and running external programs with command-line arguments using the `fork`, `wait`, and `exec` system calls in both C and Python programming languages.

Written by: Dr. Farkhana Muchtar

4.3.1 Fork and Exec in C Code

Step 1: Write a new C code, named `fork_and_exec.c` using Nano editor.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid;

    pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        return 1;
    } else if (pid == 0) {
        // Child process
        char *args[] = {"ls", "-l", "-a", NULL};
        execvp("ls", args);
    } else {
        // Parent process
        wait(NULL);
    }

    return 0;
}
```

Step 2: Compile the code using GCC.

```
gcc fork_and_exec.c -o fork_and_exec
```

Step 3: Run a compiled program

```
./fork_and_exec
```

4.3.2 Fork and Exec in Python Code

Step 1: Create a new Python code, named fork_and_exec.py with Nano editor.

```
import os
import sys

pid = os.fork()

if pid < 0:
    raise Exception("Fork failed")
elif pid == 0:
    os.execl("/bin/ls", "ls", "-l")
    sys.stderr.write("Exec failed\n")
    sys.exit(1)
```

```
else:
    os.waitpid(pid, 0)
    print("Child process completed")
```

Step 2: Run Python code using Python interpreter.

```
python fork_and_exec.py
```

4.4 Exercise 4: Inter-Process Communication (IPC) using Pipes

Objective: Learn how to use regular pipes (also known as unnamed or anonymous pipes) for inter-process communication between parent and child processes created with fork.

Explanation: In this exercise, students will learn how to perform IPC using pipes in both C and Python. They will create a pipe for communication between the parent and child processes and exchange messages between them. Students will learn how to create regular pipes (unnamed or anonymous pipes) for IPC and use them to transfer data between parent and child processes. They will understand how to read and write data to the pipe and how to synchronize the processes using the pipe.

What is Inter-Process Communication (IPC)?
Inter-process communication (IPC) refers to the various methods and mechanisms used for exchanging data and messages between processes running on a Linux system. IPC allows processes to cooperate, share resources, and synchronize their activities. Examples of IPC methods include pipes, named pipes (FIFOs), shared memory, and message queues.

What is Pipe?
In the Linux process context, a pipe is a mechanism that allows data to be transferred between two processes through a unidirectional communication channel. A pipe has a read end and a write end, enabling one process to write data to the pipe while another process reads the data.

What is Regular Pipe?
In Linux, pipes are commonly categorized into two types: unnamed pipes (also called "regular pipes" or "anonymous pipes") and named pipes (also known as "FIFOs").

Unnamed pipes, or regular pipes, are the pipes created using the `pipe()` system call in C or the `os.pipe()` function in Python. They provide a unidirectional communication channel between processes, usually between a parent and child process, and exist only for the lifetime of those processes. Unnamed pipes do not have a presence in the filesystem and are typically used for short-lived, closely related process communication.

4.4.1 IPC with Regular Pipes in C Code

Step 1: Create a new C code, named `fork_regular_pipes_ipc.c` with Nano editor.

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
```

```
int main() {
    int pipe_fd[2];
    pid_t pid;
    char message[] = "Hello from parent!";
    char buffer[20];

    if (pipe(pipe_fd) < 0) {
        perror("Pipe creation failed");
        return 1;
    }

    pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        return 1;
    } else if (pid == 0) {
        close(pipe_fd[1]);
        read(pipe_fd[0], buffer, sizeof(buffer));
        printf("Child received: %s\n", buffer);
        close(pipe_fd[0]);
    } else {
        close(pipe_fd[0]);
        write(pipe_fd[1], message, strlen(message) + 1);
        close(pipe_fd[1]);
        wait(NULL);
    }

    return 0;
}
```

Step 2: Compile the C code using GCC.

```
gcc fork_regular_pipes_ipc.c -o fork_regular_pipes_ipc
```

Step 3: Run a compiled program.

```
./fork_regular_pipes_ipc
```

4.4.2 IPC with Regular Pipes in Python Code

Step 1: Create a new Python code, named `fork_regular_pipes_ipc.py` with Nano editor.

```
import os

pipe_fd = os.pipe()
pid = os.fork()
message = b"Hello from parent!"

if pid < 0:
    raise Exception("Fork failed")
```

```
elif pid == 0:
    os.close(pipe_fd[1])
    buffer = os.read(pipe_fd[0], 20)
    print("Child received:", buffer.decode())
    os.close(pipe_fd[0])
else:
    os.close(pipe_fd[0])
    os.write(pipe_fd[1], message)
    os.close
```

Step 2: Run a Python code with Python interpreter

```
python fork_regular_pipes_ipc.py
```

4.5 Exercise 5: IPC using Named Pipes (FIFOs)

Objective: Learn how to use named pipes (FIFOs) for inter-process communication (IPC) between parent and child processes.

Explanation: Students will learn how to create named pipes (FIFOs) in the filesystem and use them to send data between parent and child processes.

What is Named Pipes (FIFOs)?
Named pipes, also known as FIFOs (First In, First Out), are a special type of file that exists in the file system and facilitates bidirectional communication between processes. Unlike regular pipes, named pipes have a persistent presence in the file system, allowing unrelated processes to communicate with each other.

4.5.1 IPC using Named Pipes (FIFOs) in C Code

Step 1: Create a new C code, named `ipc_named_pipes.c` with Nano editor.

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>

int main() {
    int fd;
    char *fifo_path = "/tmp/my_fifo";
    char message[] = "Hello from sender!";
    char buffer[20];

    mkfifo(fifo_path, 0666);
    fd = open(fifo_path, O_RDWR);

    if (fd < 0) {
        perror("Opening FIFO failed");
        return 1;
    }

    write(fd, message, sizeof(message));
    read(fd, buffer, sizeof(buffer));

    printf("Received: %s\n", buffer);

    close(fd);
    unlink(fifo_path);

    return 0;
}
```

Step 2: Compile the code using GCC.

```
      gcc ipc_named_pipes.c -o ipc_named_pipes
```

Step 3: Run a compiled code: `./ipc_named_pipes`

4.5.2 IPC using Named Pipes (FIFOs) in Python Code

Step 1: Create a new Python code, named `ipc_named_pipes.py` with Nano editor.

```
import os

fifo_path = "/tmp/my_fifo"
message = b"Hello from sender!"

os.mkfifo(fifo_path, 0o666)
fd = os.open(fifo_path, os.O_RDWR)

if fd < 0:
    raise Exception("Opening FIFO failed")

os.write(fd, message)
buffer = os.read(fd, 20)

print("Received:", buffer.decode())

os.close(fd)
os.unlink(fifo_path)
```

Step 2: Run a Python code using Python interpreter

```
      python ipc_named_pipes.py
```

4.6 Exercise 6: IPC using Shared Memory

Objective: Learn how to share data between parent and child processes using shared memory.

Explanation: Students will learn how to use shared memory segments for inter-process communication (IPC) between parent and child processes, allowing them to exchange data directly without using pipes or other IPC mechanisms.

`ipc_shared_memory` program will send a message using shared memory. Another program called `ipc_read_shared_memory` program will read that message through shared memory.

Written by: Dr. Farkhana Muchtar

What is Shared Memory?
Shared memory is a method of inter-process communication (IPC) that allows multiple processes to access and share a common memory region. It is a fast and efficient way to exchange data between processes, as it eliminates the need for data to be copied between process address spaces.

4.6.1 IPC using Shared Memory in C Code

Step 1: Create a new C code, named `ipc_shared_memory.c` with Nano editor.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>

#define SHM_SIZE 1024

int main() {
    int shmid;
    key_t key;
    char *shm, *s;

    key = ftok("shmfile", 65);
    shmid = shmget(key, SHM_SIZE, 0666 | IPC_CREAT);
    shm = shmat(shmid, NULL, 0);

    memcpy(shm, "Hello from shared memory!", 25);

    sleep(20); // sleep 20 seconds before the message is terminated

    shmdt(shm);
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

Step 2: Compile the code using GCC.

```
gcc ipc_shared_memory.c -o ipc_shared_memory
```

Step 3: Write another C code, named `ipc_read_shared_memory.c` to read the message in the shared memory

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
```

```
#include <unistd.h>

#define SHM_SIZE 1024

int main() {
    int shmid;
    key_t key;
    char *shm, *s;

    key = ftok("shmfile", 65);
    shmid = shmget(key, SHM_SIZE, 0666 | IPC_CREAT);
    shm = shmat(shmid, NULL, 0);

    // Read the message from the shared memory segment
    printf("Message from shared memory: %s\n", shm);

    shmdt(shm);
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

Step 4: Compile the code using GCC

```
gcc ipc_read_shared_memory.c -o ipc_read_shared_memory
```

Step 5: Run `ipc_shared_memory` at the background using ampersand (&), to send the message using shared memory. Run `ipc_read_shared_memory` to read the message in the shared memory.

```
./ipc_shared_memory &
./ipc_read_shared_memory
```

4.6.2 IPC using Shared Memory in Python Code

Step 1: Install `sysv_ipc` library using pip

```
pip install sysv_ipc
```

Step 2: Create empty file name shmfile

```
touch shmfile
```

Unlike C code version, `sysv_ipc` library for Python is stricter compared to the standard C library implementation. For that reason, we have to create empty file `shmfile` first, before using it in Python.

At the same time, we also ignoring warning message when using ftok function in Python due to stricter nature of sysv_ipc library in Python.

Step 3: Create a new Python code, named `ipc_shared_memory.py` with Nano editor.

```
import sysv_ipc
import os
import time
import warnings  # Import the warnings module

# Ignore warnings from sysv_ipc library
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    key = sysv_ipc.ftok("shmfile", 65)

shm = sysv_ipc.SharedMemory(key, sysv_ipc.IPC_CREAT, 0o666, 1024)

shm.write("Hello from shared memory!")

time.sleep(20)  # Wait for 20 seconds before the terminated
shm.detach()
shm.remove()
```

Step 3: Write another Python code, named `ipc_read_shared_memory.py` to read a message in shared memory.

```
import sysv_ipc
import warnings

SHM_SIZE = 1024

# Ignore warnings from sysv_ipc library
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    key = sysv_ipc.ftok("shmfile", 65)

# Attach to the shared memory segment
shm = sysv_ipc.SharedMemory(key)

# Read the message from the shared memory segment
message = shm.read().decode('utf-8').rstrip('\0')
print(f"Message from shared memory: {message}")

# Detach from the shared memory segment
shm.detach()

# Remove the shared memory segment
shm.remove()
```

Step 4: Run `ipc_shared_memory.py` using Python interpreter at the background using ampersand (&) to send the message using shared memory. `Run` `ipc_read_shared_memory.py` using Python interpreter to read those message from shared memory.

```
python ipc_shared_memory.py &
python ipc_read_shared_memory.py
```

SECR2043 - OPERATING SYSTEMS (Lab Exercise)

Written by: Dr. Farkhana Muchtar

4.7 Exercise 7: Process Synchronization using Semaphores

Objective: Learn how to use semaphores for process synchronization and coordination between parent and child processes.

Explanation: Students will learn how to use semaphores to synchronize and coordinate the execution of parent and child processes, ensuring that they access shared resources in a controlled manner.

What is Process Synchronization?
Process synchronization in Linux process management refers to the coordination and management of the execution order and timing of multiple processes running concurrently in the operating system. The main goal of process synchronization is to ensure that processes execute in a way that avoids conflicts, maintains data consistency, and allows for efficient resource sharing.

What is Semaphore?
A semaphore is a synchronization primitive used to manage access to shared resources and coordinate the execution of concurrent processes or threads in a multi-processing or multi-threading environment. Semaphores are typically used to solve synchronization problems, such as ensuring that only a limited number of processes can access a shared resource simultaneously, or enforcing the order of execution between different processes or threads.

In essence, a semaphore is an integer value that can be manipulated using two atomic operations, wait (sometimes called P or down) and signal (sometimes called V or up). The wait operation decrements the semaphore value if it is greater than zero; otherwise, it blocks the calling process or thread until the semaphore value becomes positive. The signal operation increments the semaphore value and, if there are any blocked processes or threads waiting on the semaphore, wakes up one of them.

4.7.1 Process Synchronization using Semaphores in C Code

Step 1: Create a new C code, named `process_sync_semaphores.c` with Nano editor.

```
#include <fcntl.h>
#include <semaphore.h>
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    sem_t *semaphore;
    pid_t pid;
    char *sem_name = "/my_semaphore";

    semaphore = sem_open(sem_name, O_CREAT | O_EXCL, 0666, 0);

    if (semaphore == SEM_FAILED) {
        perror("Creating semaphore failed");
        return 1;
    }

    pid = fork();
    if (pid < 0) {
        perror("Fork failed");
        return 1;
    } else if (pid == 0) {
        printf("Child waiting for semaphore\n");
        sem_wait(semaphore);
        printf("Child acquired semaphore\n");
        // Critical section starts
        sleep(5);
        // Critical section ends
        sem_post(semaphore);
    } else {
        sleep(1);
        printf("Parent releasing semaphore\n");
        sem_post(semaphore);
        wait(NULL);
    }

    sem_close(semaphore);
    sem_unlink(sem_name);

    return 0;
}
```

Step 2: Compile the C code with GCC.

```
gcc process_sync_semaphores.c -o process_sync_semaphores
```

Step 3: Run a compiled program.

```
./process_sync_semaphores
```

4.7.2 Process Synchronization using Semaphores in Python Code

Step 1: Install `posix_ipc` library using pip

```
pip install posix_ipc
```

Step 2: Create a new Python code, named `process_sync_semaphores.py` with Nano editor.

```
import posix_ipc
import os
import time

sem_name = "/my_semaphore"

try:
    semaphore = posix_ipc.Semaphore(sem_name, posix_ipc.O_CREAT |
posix_ipc.O_EXCL, 0o666, 0)
except posix_ipc.ExistentialError:
    raise Exception("Creating semaphore failed")
```

```
pid = os.fork()

if pid < 0:
    raise Exception("Fork failed")
elif pid == 0:
    print("Child waiting for semaphore")
    semaphore.acquire()
    print("Child acquired semaphore")
    # Critical section starts
    time.sleep(5)
    # Critical section ends
    semaphore.release()
    semaphore.close()  # Close the semaphore in the child process
else:
    time.sleep(1)
    print("Parent releasing semaphore")
    semaphore.release()
    os.waitpid(pid, 0)
    semaphore.close()  # Close the semaphore in the parent process

    # Unlink the semaphore after closing it
    posix_ipc.unlink_semaphore(sem_name)
```

Step 2: Run a Python code with Python interpreter

```
python process_sync_semaphores.py
```

4.8 Exercise 8: File Descriptor Sharing

Objective: Understand file descriptor sharing between parent and child processes and learn how to use this feature in a program.

Explanation: Students will learn how file descriptors are shared between parent and child processes after a fork and how to use this feature to enable communication between processes or to manipulate shared files.

What is File Descriptor?
A file descriptor is a non-negative integer that serves as an abstract handle or an identifier for accessing files, directories, sockets, or other I/O resources. File descriptors are used by the operating system to manage and keep track of open files and I/O resources for each running process.

When a process opens a file or creates a new I/O resource, the operating system assigns a unique file descriptor to that resource. The process then uses this file descriptor to perform read, write, or other I/O operations on the resource. The operating system manages a file descriptor table for each process, which maps the file descriptors to the corresponding I/O resources.

4.8.1 File Descriptor Sharing in C Code

Step 1: Create a new empty text file, named `tempfile.txt`.

```
touch tempfile.txt
```

Step 2: Create a new C code, named `fd_sharing.c` with Nano editor.

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    int fd;
    pid_t pid;
    char message[] = "Hello from parent!";
    char buffer[20];
    fd = open("tempfile.txt", O_CREAT | O_RDWR, 0666);


    if (fd < 0) {
        perror("Opening file failed");
        return 1;
    }
    pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        return 1;
    } else if (pid == 0) {
        lseek(fd, 0, SEEK_SET);
        read(fd, buffer, sizeof(buffer));
        printf("Child received: %s\n", buffer);
    } else {
        write(fd, message, sizeof(message));
        wait(NULL);
    }

    close(fd);
    unlink("tempfile.txt");

    return 0;
}
```

Step 2: Compile the C code with GCC.

```
gcc fd_sharing.c -o fd_sharing
```

Step 3: Run a compiled program.

```
./fd_sharing
```

4.8.2 File Descriptor Sharing in Python Code

Step 1: Create a new empty text file, named `tempfile.txt`.

```
touch tempfile.txt
```

Step 2: Create a new Python code, named `fd_sharing.py` with Nano editor.

```
import os

filename = "tempfile.txt"
message = b"Hello from parent!"

fd = os.open(filename, os.O_CREAT | os.O_RDWR, 0o666)

if fd < 0:
    raise Exception("Opening file failed")

pid = os.fork()

if pid < 0:
    raise Exception("Fork failed")
elif pid == 0:
    os.lseek(fd, 0, os.SEEK_SET)
    buffer = os.read(fd, 20)
    print("Child received:", buffer.decode())
    os._exit(0)  # Exit child process gracefully
else:
    os.write(fd, message)
    os.waitpid(pid, 0)

os.close(fd)

# Unlink the file only in the parent process
if pid > 0:
    os.unlink(filename)
```

Step 2: Run Python code with Python interpreter

```
python fd_sharing.py
```

4.9 Exercise 9: Resource Limits

Objective: Learn how to use the `getrlimit` and `setrlimit` system calls to manage resource limits for processes and understand their impact on the fork system call.

Explanation: Students will learn how to use the `getrlimit` and `setrlimit` system calls to query and set resource limits for processes, and understand how these limits can impact the behavior of the fork system call and the creation of child processes.

SECR2043 - OPERATING SYSTEMS (Lab Exercise)

Written by: Dr. Farkhana Muchtar

What is Resource Limit?
Resource limits, also known as process resource limits or ulimits, are constraints imposed by the operating system on the amount of various system resources that a process can consume. These limits are intended to prevent a single process from monopolizing system resources, which could lead to system instability, degraded performance, or even crashes.
Resource limits are typically set on a per-user or per-process basis and can be configured by the system administrator. Some of the common resource limits include CPU time, file size, data size, stack size, core file size, open files and number of processes.

4.9.1 Resource Limits in C Code

Step 1: Create a new C code, named `resource_limits.c` with Nano editor.

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/resource.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    struct rlimit rl;
    pid_t pid;

    getrlimit(RLIMIT_NPROC, &rl);
    printf("Current max processes: %ld\n", rl.rlim_cur);

    rl.rlim_cur = 10;
    setrlimit(RLIMIT_NPROC, &rl);

    pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        return 1;
    } else if (pid == 0) {
        printf("Child process created\n");
    } else {
        wait(NULL);
    }

    return 0;
}
```

Step 2: Compile the C code using GCC.

```
gcc resource_limits.c -o resource_limits
```

Step 3: Run a compiled program

```
./resource_limits
```

4.9.2 Resource Limits in Python Code

Step 1: Write a new Python code, named `resource_limits.py` with Nano editor.

```
import os
import resource

rl = resource.getrlimit(resource.RLIMIT_NPROC)
print("Current max processes:", rl[0])

resource.setrlimit(resource.RLIMIT_NPROC, (10, rl[1]))

pid = os.fork()

if pid < 0:
    raise Exception("Fork failed")
elif pid == 0:
    print("Child process created")
else:
    os.waitpid(pid, 0)
```

Step 2: Run a Python code with Python interptreter.

```
python resource_limit.py
```

4.10 Exercise 10: Signal Handling

Objective: Learn how to handle signals in parent and child processes created with fork and understand the impact of signal handling on process management.

Explanation: Students will learn how to set up signal handlers for parent and child processes and understand how signal handling can be used to manage and control the execution of processes created with fork. They will understand how the two processes can react to different signals and communicate using signals in both C and Python programming languages.

What is Signal Handling?
Signal handling is a mechanism in Unix-like operating systems, including Linux, that allows processes to receive and respond to asynchronous notifications called signals. Signals are software interrupts sent by the operating system, by other processes, or even by the process itself, to notify a process of certain events or conditions. Signal handling involves defining how a process should react to specific signals it receives.

Signals can be used to:

1. Terminate a process (e.g., SIGTERM, SIGINT).
2. Suspend a process and later resume its execution (e.g., SIGSTOP, SIGCONT).
3. Notify a process of an error (e.g., SIGSEGV, SIGFPE).
4. Communicate between processes (e.g., SIGUSR1, SIGUSR2).
5. Request a process to perform specific actions (e.g., SIGHUP for configuration reload).

By implementing proper signal handling, a process can respond to various events and conditions in a controlled and predictable manner, improving the overall robustness and stability of the software.

4.10.1 Signal Handling in C Code

Step 1: Create a new C code, named `signal_handling.c` with Nano editor.

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

void signal_handler(int signum) {
    if (signum == SIGUSR1) {
        printf("Received SIGUSR1\n");
    }
}

int main() {
    pid_t pid;

    if (signal(SIGUSR1, signal_handler) == SIG_ERR) {
        perror("Signal failed");
        return 1;
    }

    pid = fork();


    if (pid < 0) {
        perror("Fork failed");
        return 1;
    } else if (pid == 0) {
        printf("Child process\n");
        pause();
    } else {
        printf("Parent process\n");
        sleep(1);
        kill(pid, SIGUSR1);
        wait(NULL);
    }

    return 0;
}
```

Step 2: Compile the code using GCC.

```
gcc signal_handling.c -o signal_handling
```

Step 3: Run a compiled program.

```
./signal_handling
```

4.10.2 Signal Handling in Python Code

Step 1: Create a new Python code, named `signal_handling.py` with Nano editor.

```
import os
import signal
import time

def signal_handler(signum, frame):
    if signum == signal.SIGUSR1:
        print("Received SIGUSR1")

signal.signal(signal.SIGUSR1, signal_handler)

pid = os.fork()

if pid < 0:
    raise Exception("Fork failed")
elif pid == 0:
    print("Child process")
    signal.pause()
else:
    print("Parent process")
    time.sleep(1)
    os.kill(pid, signal.SIGUSR1)
    os.waitpid(pid, 0)
```

Step 2: Run a Python code using Python interpreter.

```
python signal_handling.py
```