Lecture_4_Processes_and_Multi_threading_programming

February 6, 2025

1 Introduction

Today, we'll dive into the core principles of processes and multi-threading programming in C:

- Understanding how processes are created and managed in C.
- Exploring the concepts of process synchronization and inter-process communication (IPC).
- Grasping the principles of multi-threading, thread creation, and thread management.
- Understanding thread synchronization techniques to avoid race conditions and deadlocks.
- How processes and threads enable efficient parallel execution and resource sharing.

Understanding processes and multi-threading in C is crucial because they allow programs to perform multiple tasks concurrently, manage system resources efficiently, and handle complex computations in parallel.

---

2 Processes

If files are the most fundamental abstraction in a Unix system, processes are the runner up. Processes are object code in execution: active, running programs. But they're more than just object code — processes consist of data, resources, state, and a virtualized computer.

Processes begin life as executable object code, which is machine-runnable code in an executable format that the kernel understands. The format most common in Linux is called **Executable and Linkable Format (ELF)**. The executable format contains metadata, and multiple *sections* of code and data. Sections are linear chunks of the object code that load into linear chunks of memory. All bytes in a section are treated the same, given the same permissions, and generally used for similar purposes.

The most important and common sections are the **text section**, the **data section**, and the **bss section**.

The **text section** contains executable code and read-only data, such as constant variables, and is typically marked read-only and executable.

The **data section** contains initialized data, such as C variables with defined values, and is typically marked readable and writable.

The **bss section** contains uninitialized global data. Because the C standard dictates default values for global C variables that are essentially all zeros, there is no need to store the zeros in the object code on disk. Instead, the object code can simply list the uninitialized variables in the bss section,

and the kernel can map the zero page (a page of all zeros) over the section when it is loaded into memory. The bss section was conceived solely as an optimization for this purpose. The name is a historic relic; it stands for block started by symbol.

Other common sections in ELF executables are the **absolute section** (which contains nonrelocatable symbols) and the **undefined section** (a catchall).

A process is also associated with various system resources, which are arbitrated and managed by the kernel. Processes typically request and manipulate resources only through system calls. Resources include timers, pending signals, open files, network connections, hardware, and IPC mechanisms. A process's resources, along with data and statistics related to the process, are stored inside the kernel in the process's **process descriptor**.

A process is a virtualization abstraction. The Linux kernel, supporting both preemptive multitasking and virtual memory, provides every process both a virtualized processor and a virtualized view of memory. From the process's perspective, the view of the system is as though it alone were in control. That is, even though a given process may be scheduled alongside many other processes, it runs as though it has sole control of the system. The kernel seamlessly and transparently preempts and reschedules processes, sharing the system's processors among all running processes. Processes never know the difference. Similarly, each process is afforded a single linear address space, as if it alone were in control of all of the memory in the system. Through virtual memory and paging, the kernel allows many processes to coexist on the system, each operating in a different address space. The kernel manages this virtualization through hardware support provided by modern processors, allowing the operating system to concurrently manage the state of multiple independent processes.

3 Threads

Each process consists of one or more **threads of execution** (usually simplified to threads). A thread is the unit of activity within a process. In other words, a thread is the abstraction responsible for executing code and maintaining the process's running state.

Most processes consist of only a **single thread**; they are called single-threaded. Processes that contain multiple threads are said to be **multi-threaded**. Traditionally, Unix programs have been single-threaded, owing to Unix's historic simplicity, fast process creation times, and robust IPC mechanisms, all of which mitigate the desire for threads.

A thread consists of a stack (which stores its local variables, just as the process stack does on nonthreaded systems), processor state, and a current location in the object code (usually stored in the processor's **instruction pointer**). The majority of the remaining parts of a process are shared among all threads, most notably the process address space. In this manner, threads share the virtual memory abstraction while maintaining the virtualized processor abstraction.

Internally, the Linux kernel implements a unique view of threads: they are simply normal processes that happen to share some resources. In user space, Linux implements threads in accordance with POSIX 1003.1c (known as Pthreads). The name of the current Linux thread implementation, which is part of `glibc`, is the Native POSIX Threading Library (NPTL).

4 Process hierarchy

Each process is identified by a unique positive integer called the `process ID` (`pid`). The `pid` of the first process is `1`, and each subsequent process receives a new, unique `pid`.

In Linux, processes form a strict hierarchy, known as the **process tree**. The process tree is rooted at the first process, known as the **init process**, which is typically the **init** program. New processes are created via the `fork()` system call. This system call creates a duplicate of the calling process. The original process is called the **parent**; the new process is called the **child**. Every process except the first has a parent. If a parent process terminates before its child, the kernel will **reparent** the child to the init process.

When a process terminates, it is not immediately removed from the system. Instead, the kernel keeps parts of the process resident in memory to allow the process's parent to inquire about its status upon terminating. This inquiry is known as **waiting on** the terminated process. Once the parent process has waited on its terminated child, the child is fully destroyed. A process that has terminated, but has not yet been waited upon, is called a **zombie**. The init process routinely waits on all of its children, ensuring that reparented processes do not remain zombies forever.

4.1 Example 1: Example: Creating a child process

Processes are independent execution units with their own memory space. You can create a new process using the `fork()` system call. The `fork( )` function creates a child process which is exactly identical to the parent process except for the return value of `fork( )`:

- Negative on failure.
- `0` for the child process.
- Positive integer (child PID) for the parent process.

[8]:
```
%%writefile child-process.c

#include <stdlib.h>        /* needed to define exit() */
#include <unistd.h>        /* needed for fork() and getpid() */
#include <stdio.h>        /* needed for printf() */

int main() {
        int pid;        /* process ID */

        switch (pid = fork()) {
        case 0:                /* a fork returns 0 to the child */
                printf("Hello from the child process! (PID: %d)\n", getpid());
                break;

        default:        /* a fork returns a pid to the parent */
                printf("Hello from the parent process! (PID: %d, Child PID:␣
↪%d)\n", getpid(), pid);
                break;

        case -1:        /* something went wrong */
```

```
                perror("fork");
                exit(1);
        }
        exit(0);
}
```

```
Overwriting child-process.c
```

[9]:
```
%%script bash
gcc child-process.c -o child-process
./child-process
```

```
Hello from the parent process! (PID: 4814, Child PID: 4815)
Hello from the child process! (PID: 4815)
```

4.2 Example 2: Child and parent processes

Let us consider the following program that uses the child to compute partial sums (`sum`) and parent to compute the partial products (`pdt`) of an array of integers.

[18]:
```
%%writefile child-process2.c

#include <stdlib.h>        /* needed to define exit() */
#include <unistd.h>        /* needed for fork() and getpid() */
#include <stdio.h>        /* needed for printf() */

int main() {
  int A[] = {1, 2, 3, 4, 5, 6};
  int sum = 0, pdt = 1, pid;

        switch (pid = fork()) {
        case 0:                /* a fork returns 0 to the child */
                for (int i = 0; i < 6; i++) sum += A[i];
    printf("This is child process computed sum %d \n", sum);
                break;

        default:        /* a fork returns a pid to the parent */
    for (int i = 0; i < 6; i++) pdt *= A[i];
                printf("The parent process completed the product %d \n", pdt);
                break;

        case -1:        /* something went wrong */
                perror("problem creating a process");
                exit(1);
        }
        exit(0);
}
```

```
Overwriting child-process2.c
```

[19]:
```
%%script bash
gcc child-process2.c -o child-process2
./child-process2
```

```
The parent process completed the product 720
This is child process computed sum 21
```

4.3 Example 3: Execute any shell command with up to 2 arguments

[20]:
```
%%writefile shell.c

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

#define MAX_ARGS 3 // Command + 2 arguments

int main() {
    char input[100];
    char *args[MAX_ARGS + 1]; // Array to hold command and arguments (+1 for␣
 ↪NULL)
    pid_t pid;

    while (1) {
        // Prompt the user for input
        printf("myshell> ");
        if (fgets(input, sizeof(input), stdin) == NULL) {
            break; // Exit on EOF (Ctrl+D)
        }

        // Remove the newline character from the input
        input[strcspn(input, "\n")] = '\0';

        // Tokenize the input into command and arguments
        int i = 0;
        args[i] = strtok(input, " ");
        while (args[i] != NULL && i < MAX_ARGS) {
            i++;
            args[i] = strtok(NULL, " ");
        }
        args[i] = NULL; // Last element must be NULL for execvp

        // Check if the user entered a command
        if (args[0] == NULL) {
            continue; // Empty input, prompt again
```

```
        }

        // Create a child process to execute the command
        pid = fork();
        if (pid < 0) {
            perror("fork"); // Fork failed
            exit(EXIT_FAILURE);
        } else if (pid == 0) {
            // Child process: Execute the command
            execvp(args[0], args);
            // If execvp returns, there was an error
            perror("execvp");
            exit(EXIT_FAILURE);
        } else {
            // Parent process: Wait for the child to finish
            wait(NULL);
        }
    }

    return 0;
}
```

```
Writing shell.c
```

[22]:
```
%%script bash
gcc shell.c -o shell
```

[24]:
```
!./shell
```

```
myshell> ls -l
total 68
-rwxr-xr-x 1 root root 16136 Feb  6 08:59 child-process
-rwxr-xr-x 1 root root 16096 Feb  6 09:23 child-process2
-rw-r--r-- 1 root root   688 Feb  6 09:23 child-process2.c
-rw-r--r-- 1 root root   569 Feb  6 08:59 child-process.c
drwxr-xr-x 1 root root  4096 Feb  4 14:22 sample_data
-rwxr-xr-x 1 root root 16392 Feb  6 09:31 shell
-rw-r--r-- 1 root root  1554 Feb  6 09:31 shell.c
myshell> gcc --version
gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

myshell> echo Hello World from Linux!
Hello World
myshell> ls
child-process  child-process2  child-process2.c  child-process.c  sample_data
```

```
shell  shell.c
myshell> cat shell.c

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

#define MAX_ARGS 3 // Command + 2 arguments

int main() {
    char input[100];
    char *args[MAX_ARGS + 1]; // Array to hold command and arguments (+1 for
NULL)
    pid_t pid;

    while (1) {
        // Prompt the user for input
        printf("myshell> ");
        if (fgets(input, sizeof(input), stdin) == NULL) {
            break; // Exit on EOF (Ctrl+D)
        }

        // Remove the newline character from the input
        input[strcspn(input, "\n")] = '\0';

        // Tokenize the input into command and arguments
        int i = 0;
        args[i] = strtok(input, " ");
        while (args[i] != NULL && i < MAX_ARGS) {
            i++;
            args[i] = strtok(NULL, " ");
        }
        args[i] = NULL; // Last element must be NULL for execvp

        // Check if the user entered a command
        if (args[0] == NULL) {
            continue; // Empty input, prompt again
        }

        // Create a child process to execute the command
        pid = fork();
        if (pid < 0) {
            perror("fork"); // Fork failed
            exit(EXIT_FAILURE);
        } else if (pid == 0) {
            // Child process: Execute the command
```

```
            execvp(args[0], args);
            // If execvp returns, there was an error
            perror("execvp");
            exit(EXIT_FAILURE);
        } else {
            // Parent process: Wait for the child to finish
            wait(NULL);
        }
    }

    return 0;
}
myshell> ^C
```

The program runs in a loop, allowing the user to enter multiple commands until they exit. The program reads input from the user using `fgets()` and tokenizes into the command and its arguments using `strtok()`. The `fork()` system call creates a child process. The child process uses `execvp()` to replace its memory space with the specified command. If `execvp()` fails, an error message is printed. The parent process waits for the child process to finish using `wait()`.

The program supports commands with up to 2 arguments. If you need to support more arguments, increase the value of `MAX_ARGS`. It does not handle complex shell features like piping (`|`), redirection (`>`, `<`), or background execution (`&`). For such features, you would need to extend the program.

5 Threading

Threading is the creation and management of multiple units of execution within a single process. Threading is a significant source of programming error, through the introduction of data races and deadlocks. The topic of threading can — and indeed does — fill whole books. We will cover the basics of the multithreading programming in C: Why use threads? What design patterns help us conceptualize and build threading applications? And, finally, what are data races and how can we prevent them?

5.1 Where threads can be used?

- Threads are lightweight variants of the processes that share the memory space. You can create threads using the `pthread` library.
  - **More efficient usage of the available computational resources**
    - When a process waits for resources (e.g., reads from a periphery), it is blocked, and control is passed to another process
    - Thread also waits, but another thread within the same process can utilize the dedicated time for the process execution
    - Having multi-core processors, we can speedup the computation using more cores simultaneously by **parallel algorithms**
  - **Handling asynchronous events**
    - During blocked i/o operation, the processor can be utilized for other computational

    - One thread can be dedicated for the i/o operations, e.g., per communication channel, another threads for computations
  - **Context switching**
    - The cost of switching from one thread to a different thread within the same process is significantly cheaper than process-to-process context switching.
  - **Memory savings**
    - Threads provide an efficient way to share memory yet utilize multiple units of execution. In this manner they are an alternative to multiple processes.

5.2 Threads vs Processes

Feature	Processes	Threads
Memory Space	Separate memory space	Shared memory space
Creation Overhead	Higher (due to memory duplication)	Lower (shares memory with parent)
Communication	Requires IPC (e.g., pipes, shared memory, message queues)	Direct access to shared memory
Synchronization	Not needed (independent execution)	Required (e.g., mutex, semaphore, condition variables)
Execution Speed	Slower (context switching between processes is expensive)	Faster (lightweight, less context switching overhead)
Resource Sharing	No sharing (each process has its own resources)	Shares code, data, and file descriptors within the process
Failure Impact	One process crash doesn't affect others	Thread crash can affect the whole process
Example API	`fork()`, `exec()` in Unix/Linux	`pthread_create()` in POSIX threads

6 Example 4: Creating threads

[25]:
```
%%writefile threads.c

#include <stdio.h>
#include <pthread.h>

// Function to be executed by the thread
void* thread_function(void* arg) {
    int thread_id = *(int*)arg;
    printf("Hello from thread %d!\n", thread_id);
    pthread_exit(NULL); // Exit the thread
}

int main() {
    pthread_t threads[3]; // Array to hold thread IDs
    int thread_args[3];   // Array to hold thread arguments

    // Create 3 threads
    for (int i = 0; i < 3; i++) {
        thread_args[i] = i;
        int result = pthread_create(&threads[i], NULL, thread_function,␣
 ↪&thread_args[i]);
        if (result != 0) {
            fprintf(stderr, "Error creating thread %d\n", i);
            return 1;
        }
    }

    // Wait for all threads to finish
```

```
    for (int i = 0; i < 3; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("All threads have finished execution.\n");
    return 0;
}
```

```
Writing threads.c
```

[26]:
```
%%script bash
gcc threads.c -o threads
```

[27]:
```
!./threads
```

```
Hello from thread 0!
Hello from thread 1!
Hello from thread 2!
All threads have finished execution.
```

`pthread_create()` creates a new thread. The thread executes the function passed as the third argument (`thread_function` in this case).

`pthread_join()` waits for the thread to finish execution.

Threads share the same memory space, so they can access global variables and heap memory.

7 Example 5: Synchronization in multi-threading

When multiple threads access shared resources, synchronization is required to avoid race conditions.

[28]:
```
%%writefile synchronization.c

#include <stdio.h>
#include <pthread.h>

int shared_counter = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* thread_function(void* arg) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&mutex); // Lock the mutex
        shared_counter++;           // Critical section
        pthread_mutex_unlock(&mutex); // Unlock the mutex
    }
    pthread_exit(NULL);
}
```

```
int main() {
    pthread_t threads[2];

    // Create 2 threads
    for (int i = 0; i < 2; i++) {
        pthread_create(&threads[i], NULL, thread_function, NULL);
    }

    // Wait for threads to finish
    for (int i = 0; i < 2; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Final value of shared_counter: %d\n", shared_counter);
    return 0;
}
```

```
Writing synchronization.c
```

[29]:
```
%%script bash
gcc synchronization.c -o synchronization
```

[30]:
```
!./synchronization
```

```
Final value of shared_counter: 200000
```

A mutex is used to ensure that only one thread can access the `shared_counter` at a time. Without synchronization, the final value of `shared_counter` might be incorrect due to race conditions.

8 Example 6: Multi-threading without synchronization

[31]:
```
%%writefile without-synchronization.c

#include <stdio.h>
#include <pthread.h>

int shared_counter = 0;

void* thread_function(void* arg) {
    for (int i = 0; i < 100000; i++) {
        shared_counter++; // Critical section (no synchronization)
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[2];
```

```
    // Create 2 threads
    for (int i = 0; i < 2; i++) {
        pthread_create(&threads[i], NULL, thread_function, NULL);
    }

    // Wait for threads to finish
    for (int i = 0; i < 2; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Final value of shared_counter: %d\n", shared_counter);
    return 0;
}
```

```
Writing without-synchronization.c
```

[37]:
```
%%script bash
gcc without-synchronization.c -o without-synchronization
./without-synchronization
```

```
Final value of shared_counter: 136942
```

Since each thread increments shared_counter `100 000` times, the expected final value is `200 000`. However, due to the race condition, the final value will likely be less than `200 000`.

8.1 Race Condition

The operation `shared_counter++` is not atomic. It involves three steps:

1. Read the current value of `shared_counter`.
2. Increment the value
3. Write the new value back to `shared_counter`.

Without synchronization, both threads can simultaneously read the same value, increment it, and write back the same result, causing lost updates.