Processes and Threads

Cerecedo Gallegos Jesús ,Gallegos Ruiz Diana Abigail, Hernandez Torres Marco Antonio, Montaño Mata America Itzel, Ramírez Hernández Ana Daniela, Romero García César Eduardo

***Abstract*—This article provides an in-depth exploration of processes and threads tailored for students navigating the Distributed Systems landscape. It uncovers the intricacies of processes, the functional units of execution, delving into their structures, states, and the critical role they play in managing resources. Threads, described as code sequences within processes, are introduced, shedding light on their significance for concurrent tasks.**

The narrative extends to process scheduling, distinguishing between non-preemptive and preemptive strategies, and outlining common scheduling algorithms.In the context of Distributed Systems, the creation and termination of processes take center stage, illustrating the essential steps involved in these fundamental operations.

The article culminates in an examination of the practical significance of signals, offering a nuanced understanding of their role in the graceful termination and synchronization of processes. Tailored for students, this article serves as both an educational resource and a practical guide, setting the stage for a comprehensive understanding of processes and threads in the realm of Distributed Systems.

I. Introduction

In the realm of computing, a process is a term used to describe a program or command currently executing within a computer system. Processes serve as the means by which a computer manages and executes multiple tasks and applications concurrently, enabling effective user interaction. A key distinction from a program lies in the fact that a program is essentially a set of instructions required to perform a specific task.
[6]

Some critical aspects to consider about processes include:

- Processes exist in parent-child hierarchies.
- The system assigns a Process Identification Number (PID) to each process upon initiation.
- When a process runs on a system, it utilizes available resources such as memory and CPU time.

Regarding threads, it can be described as a sequence of code executing within the context of a process, as threads cannot run independently; they require supervision from a process.

In general terms, a process is an executing program, encompassing the program counter value, registers, and variables. It is also included in the program's memory space.

A. Some types of processes

Zombie Process A zombie process is a terminated process that no longer runs but continues to be recognized in the process table (meaning it still has a PID). System space is no longer allocated to such a process. Zombie processes persist in the process table until the parent process dies or the system shuts down and restarts.

Orphan Process
An orphan process is one that has lost its parent process, typically responsible for cleaning up the process resources. In Unix/Linux, when a parent process terminates, its child processes become orphaned and are adopted by the "init" process, which becomes the new parent.

B. Process Scheduling

Non-preemptive Scheduling In non-preemptive scheduling, once a process gets its turn for execution, it cannot be suspended. In other words, CPU usage cannot be taken away until the process willingly relinquishes it. However, this scheme has drawbacks, as processes with infinite loops can indefinitely postpone others.

Preemptive Scheduling Preemptive scheduling allows the operating system to take away CPU usage from a running process. It involves a clock generating periodic interrupts, during which the scheduler decides whether the current process continues or yields to another. [3]

Various scheduling algorithms can be applied in both approaches:

- Round Robin (preemptive)
- Priority-based (preemptive)
- Shortest Remaining Time First (preemptive)
- Shortest Job Next (non-preemptive)
- First-Come-First-Served (non-preemptive)

Each algorithm uses different criteria to determine the order of CPU assignment to processes.

II. Process States

In an operating system, a process can transition through several states in its lifetime: - Ready: The process is prepared for execution but has not yet been allocated CPU time.

- *Running:* The process is actively executed by the CPU.
- *Blocked:* When a process awaits the completion of an I/O operation, it is in the blocked state.
- *Terminated:* The process reaches this state upon completion and prepares for removal from the system.

A. State Transitions

Processes can change from one state to another due to various events:

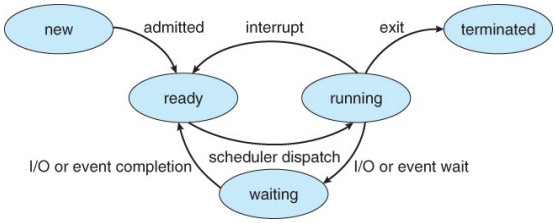

Fig. 1: Diagram of process state Operating, Taken from Systems Course, University of Illinois at Chicago, [Online]. Available: https://www.cs.uic.edu/ jbell/CourseNotes/OperatingSystems/3_Processes.html

1) *Ready to Running:* A process in the ready state gains CPU access, moving to the running state when the process scheduler allocates CPU time.
2) *Running to Blocked:* If a process needs to perform an I/O operation, it transitions to the blocked state, waiting for the operation's completion.
3) *Blocked to Ready:* After completing the I/O operation, the process returns to the ready state, awaiting its turn for execution.
4) *Running to Terminated:* When a process finishes its task, it moves to the terminated state.

B. Examples of State Transitions:

To illustrate state transitions, consider these examples: - Example 1 - Ready to Running: A text editing process awaits its turn for execution. When the user launches the application, the process moves to the running state. - Example 2 - Running to Blocked: The same text editing process performs a file-saving operation, entering the blocked state during I/O. - Example 3 - Blocked to Ready: After completing the save operation, the process returns to the ready state, ready to resume execution. - Example 4 - Running to Terminated: When the user closes the text editing application, the process reaches termination, moving to the terminated state.

III. Process Creation

creation of processes in an operating system is an intrinsic and critical process enabling the efficient execution of multiple tasks concurrently. The fundamental steps involved in this process are detailed below. [9]

1. Resource Reservation:
Initiating a new process involves the allocation of essential resources by the operating system. This includes memory space assignment, Process Identification Number (PID) allocation, CPU register configuration, and other vital resource assignments.

2. Parent Process Copy:
In numerous operating systems, the new process originates as an exact duplicate of the parent process. This replication encompasses the code, data, and other resources inherent to the parent process.

3. Resource Allocation:
Subsequently, the new process may receive additional resources or inherit resources from the parent process, depending on the specific implementation of the operating system.

4. Execution Start:
The new process is queued for execution, and at some point, the operating system allocates CPU time for its effective execution.

The creation of processes is a cardinal principle enabling the concurrent execution of diverse tasks, each representing an independent instance of a running program with its memory space and associated resources.

A. Example of Process Creation in C

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

int main() {
  // Fork a new process
  pid_t child_pid = fork();

  if (child_pid == -1) {
    // Forking failed
    perror("Forking failed");
    exit(1);
  } else if (child_pid == 0) {
    // This code runs in the child process
    printf("Child process: Hello, I am the child!\
n");
    // Terminate the child process
    exit(0);
  } else {
    // This code runs in the parent process
    printf("Parent process: I created a child with
 PID %d.\n", child_pid);
  }

  // This code runs in both the parent and child
  processes
  printf("This message is printed by both the
  parent and child processes.\n");

  return 0;
}
```

IV. Process Termination and Signals

Once processes are created, it is often necessary to terminate them to release resources used during their lifetime. In concurrent programming, a common practice is for a parent process to wait for a child process to finish before continuing its execution. To synchronize these processes, the `exit` and `wait` calls are used, where the `fork()` call creates the child process, and the child uses the `exit` function to return control to the parent process [7].

In the context of C programming, a return made from the main function has a similar effect to using `exit`. In other words, it is equivalent to the classic `return 0;` statement, where the value 0 conventionally indicates that the process has terminated successfully. However, other values may indicate errors or specific termination conditions, varying

according to developers [6].

It's important to note that various forms of termination exist, many of which make use of system signals. For example, pressing `CTRL+C` sends a `SIGINT` signal, allowing safe termination, and the process can handle this signal. Additionally, SIGKILL is used for forced termination of uncontrollable processes consuming many resources, although this must be handled with caution, depending on the use case.

Speaking in detail about signals, these are communication and control mechanisms used to notify events and modify the behavior of processes and applications at runtime. Some common signals include `SIGINT` for interruption, `SIGTERM` for graceful termination, `SIGKILL` for forced termination, `SIGHUP` for signaling a process to reload its configuration or restart, and `SIGSEGV` for a process attempting to access memory without proper permissions.

These signals are crucial in the control and management of processes in operating systems, allowing proper handling of termination and interaction between processes.

V. Threads

Within a program, a Thread is a separate execution path. It is a lightweight process that the operating system can schedule and run concurrently with other threads. The operating system creates and manages threads, and they share the same memory and resources as the program that created them. This enables multiple threads to collaborate and work efficiently within a single program.

A thread is a single sequence stream within a process. Threads are also called lightweight processes as they possess some of the properties of processes. Each thread belongs to exactly one process. In an operating system that supports multithreading, the process can consist of many threads. But threads can be effective only if CPU is more than 1 otherwise two threads have to context switch for that single CPU.

A. Difference Between Process and Thread

The primary difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces. Threads are not independent of one another like processes are, and as a result, threads share with other threads their code section, data section, and OS resources (like open files and signals). But, like a process, a thread has its own program counter (PC), register set, and stack space.

Components of Threads: These are the basic components of the Operating System

- Stack Space
- Register Set
- Program Counter

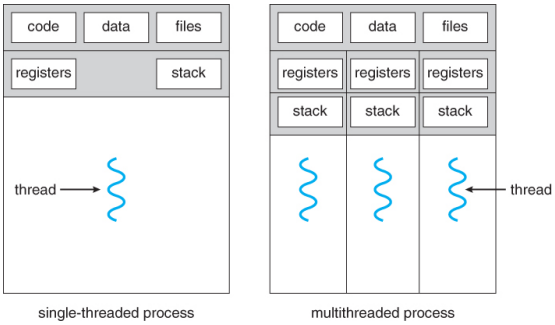

Fig. 2: Processes and Threads, Taken from DEV Community, [Online].Available:https://dev.to/suvhotta/processes-and-threads-part-1-pm2

B. Advantages of Thread

1) *Responsiveness:* If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.
2) *Faster context switch:* Context switch time between threads is lower compared to the process context switch. Process context switching requires more overhead from the CPU.
3) *Effective utilization of multiprocessor system:* If we have multiple threads in a single process, then we can schedule multiple threads on multiple processors. This will make process execution faster.
4) *Resource sharing:* Resources like code, data, and files can be shared among all threads within a process. Note: Stacks and registers can't be shared among the threads. Each thread has its own stack and registers.
5) *Communication:* Communication between multiple threads is easier, as the threads share a common address space. while in the process we have to follow some specific communication techniques for communication between the two processes.
6) *Enhanced throughput of the system:* If a process is divided into multiple threads, and each thread function is considered as one job, then the number of jobs completed per unit of time is increased, thus increasing the throughput of the system.

References

[1] (2023) Procesos. IBM. [Online]. Available: https://www.ibm.com/docs/es/aix/7.3?topic=processes-
[2] Modelo de procesos. [Online]. Available: https://www.ecured.cu/Modelo_de_procesos
[3] A. Nakayama and J. Solano. Guía práctica de estudio 12: Hilos. [Online]. Available: http://profesores.fi-b.unam.mx/annkym/LAB/poo_p12.pdf
[4] U. de la República Uruguay. Sistemas operativos. curso 2014. procesos. [Online]. Available: https://www.fing.edu.uy/inco/cursos/sistoper/recursosTeoricos/5-SO-Teo-Procesos.pdf
[5] M. Kerrisk, *The Linux Programming Interface*. No Starch Press, 2010.
[6] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. John Wiley & Sons, 2009.
[7] W. R. Stevens and S. A. Rago, *Advanced Programming in the UNIX Environment*. Addison-Wesley, 2013.
[8] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*. Pearson, 2014.
[9] F. Marquez, *Unix - Programación Avanzada*. Alfaomega Grupo, 2005.
[10] U. of Illinois at Chicago. Operative systems. proccess. [Online]. Available: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/3_Processes.html