AEC1 – Operating Systems

ISMAEL HERNÁNDEZ CLEMENTE

Instructions

OPERATING SYSTEMS

The topic of the debate will focus on aspects seen in the first unit of the operating systems subject. The debate will focus on three key blocks:

1st The appearance of operating systems

2nd Structure of a SSOO

3. Functionalities of operating systems

Block 1: The emergence of operating systems and basic concepts

As explained in the unit, it is necessary to first view the following video that briefly explains the CTSS (Compatible Time Sharing) project System) : LINK: https://www.youtube.com/watch?v=Q07PhW5sCEk&t=278s

Students are asked to reflect on the following questions and justify their answers:

- Explain in a summarized way and in your own words what the CTSS consists of
- Identify a couple of commercial Operating Systems (even if they are no longer in use) and identify what type they are, any notable characteristics (positive or negative) and, if they are no longer used, the reason for their disappearance.

Block 2: Structure of an SSOO

Students are asked to search for information about the POSIX or WIN32 standards, select some of the functionalities they describe and relate it to its implementation in a commercial operating system.

Block 3: Functionalities of operating systems

Share with your colleagues which of the high-level functionalities of an operating system you consider essential and why

The objective of the debate is to share with classmates the initial and basic knowledge acquired about operating systems. It is desirable that, in addition to students contributing their ideas, they take advantage of their contribution to comment on the contributions of their classmates and thus have the essence of a debate.

Block 1: Emergence of Operating Systems

As my colleague mentioned, the CTSS, developed by MIT, was one of the first systems to implement the concept of time sharing, allowing several users to simultaneously access a single central computer. This meant that the user had the impression of working on their own machine thanks to multitasking, which maximized the efficient use of the computer's resources.

For me it is important to highlight that the CTSS, despite its limitations and looking to the future, changed the way in which resources were used, allowing multiple users to interact with the system at the same time and I believe that this opened the door for many developments in computing, today present in servers and distributed systems.

On the other hand, the Unix-based Mac OS remains a very relevant operating system. The colleague highlights its integration between hardware and software, which has indeed been, in my opinion, one of the reasons for Apple's success worldwide. Regarding the limitations mentioned, especially the exclusivity for Apple hardware, we can reflect on how this strategy has worked in favor of the company, focusing on a niche market willing to pay more for this optimization, although it limits access to more users; in the end, Apple buyers prefer the exclusivity offered by the brand rather than compatibility and other relevant aspects.

As a second operating system in my case I will use the example of the defunct BeOS, developed by Be Inc. in the 90s, it was an operating system designed specifically for processing digital media, such as video and audio. Its architecture was optimized for multimedia applications, allowing excellent multitasking performance and fast access to disks and devices. However, BeOS never managed to gain enough commercial support and was overshadowed by other more popular operating systems such as Windows and Mac OS with which it tried to compete. Its demise was largely due to the lack of hardware support as companies only had eyes for the two big tech giants (Apple and Microsoft) and their dominance of established systems.

Block 2: Structure of an Operating System

The colleague mentions the POSIX standard and its implementation in Linux through Pthreads , which leads me, since I recently carried out a project on it, to link it with one of the main problems that this functionality solves, Dijkstra's philosophers' problem, a classic synchronization problem in concurrent computing that illustrates the difficulties inherent in managing shared resources between multiple processes.

This problem arises in a scenario where several philosophers sit around a table and alternate between thinking and eating. In order to eat, each philosopher needs two forks, which are limited resources, prompting the need to ensure mutual exclusion and avoid race conditions. In this context, proper synchronization is crucial to avoid problems such as deadlock (where philosophers sit indefinitely waiting for forks) and circular waiting (where each philosopher holds one fork and waits for the other, blocking the others). The tools provided by the POSIX standards, and in particular thread management via Pthreads , offer effective solutions to these synchronization challenges.

POSIX and its solution to the Philosophers' Problem

1. Thread Synchronization

POSIX provides several synchronization mechanisms that are essential for solving problems such as the diner philosophers' problem. These mechanisms include:

- **Mutexes** : Mutexes (mutual exclusion) allow only one thread to access a shared resource at a given time. In the case of Philosophers, this means that only one Philosopher can use a fork at a time. By implementing mutexes for each fork, it can be ensured that one Philosopher does not attempt to use a fork that is already in use by another.
- **Condition variables** : These allow threads to be blocked and woken up based on certain conditions. For example, a philosopher might wait on a condition variable until both forks are available before proceeding to eat.

This careful synchronization avoids race conditions and allows philosophers to behave in a coordinated manner.

2. Communication between Threads

Communication between threads is essential in concurrent environments. POSIX facilitates this communication through the use of shared variables and data structures. In the context of the philosophers' problem:

- Philosophers can indirectly "communicate" about the state of forks by using shared variables that indicate whether a fork is available or not. This effective communication allows philosophers to coordinate the use of forks without conflict, which in turn helps prevent deadlock.
- Implementing priority logic or signals can further improve this communication, allowing one philosopher to wake up another when a fork is released, thus optimizing access to resources.

3. Execution Control

Execution control features in POSIX, such as suspending and resuming threads, are useful for implementing solutions to the philosopher's problem. For example:

- A philosopher can suspend his execution until both forks are available, thus avoiding the situation where a philosopher starts eating without having access to the two necessary forks. By suspending his thread, the philosopher relinquishes control to the system, allowing other philosophers to access the resources.
- Upon resuming execution, the philosopher can eat without conflict, allowing for more efficient use of shared resources and reducing the likelihood of circular waiting conditions.

Implementation in Commercial Operating Systems

On modern operating systems like **Linux , the** Pthreads implementation is highly efficient and well documented. This provides developers with a solid framework to address synchronization problems like the one the philosophers pose:

Pthreads is designed to be efficient in its handling of threads, allowing multiple threads to run effectively on systems with multiple CPU cores.

Extensive documentation and community support make it easy for developers to understand threading concepts and apply best practices when implementing solutions.

Pthreads ' portability allows solutions developed on Linux to be easily adapted to other POSIX-compliant systems. This is especially advantageous for cross-platform software development, as the same principles of synchronization and thread management can be applied in different environments without significant modifications.

While I strongly agree with the solid explanation Jesus has given about WIN32, I would also like to mention that the WIN32 API is not only limited to basic file management operations, but also includes functions for file synchronization, such as FlushFileBuffers , which ensures that all data in memory is written to disk, thus improving data integrity.

I also add that it allows temporary file management, making it easy to create files that are automatically deleted when the application is closed. This is useful for operations that require temporary storage and helps keep the system clean of unnecessary files. On the other hand, support for access paths in WIN32 is a strong point.

Finally, the extensive documentation and resources available for the WIN32 API allow developers to quickly find examples and guides, contributing to a more efficient and effective implementation of these functions in their applications.

Block 3: Functionalities of Operating Systems

For me, thread management is a key feature in modern operating systems, which allows for the concurrent execution of multiple threads within a single process. A thread is the smallest unit of processing that can be scheduled and executed by an operating system.

Each thread in a process shares the same address space and resources, such as memory, but has its own stack and registers, allowing threads to communicate and work together efficiently.

- Thread management allows multiple threads to run at the same time, making it possible for applications to perform multiple tasks simultaneously, such as downloading data, processing information, and updating the user interface, without blocking the overall operation of the application.
- Threads within a process share resources such as memory and open files, which facilitates communication and data exchange between them. This is more efficient than inter-process communication (IPC), which requires more complex mechanisms.
- Threads are lighter compared to processes. Creating and destroying threads is generally faster than doing so with processes, and context switching between threads is less expensive in terms of time and resources.

One of the problems I have encountered most often in large C projects is the management of so-called zombie processes. These occur when a child process finishes its execution, but its parent does not collect its status. This can happen, for example, if the parent process does not call the wait ( ) function to obtain the child's exit code. Although the zombie process does not consume significant

resources, it does occupy an entry in the system's process table, which can lead to resource exhaustion if not handled properly.

In addition to these, I can think of several other problems that thread management solves, such as:

- Race conditions: These occur when two or more threads attempt to access and modify shared resources simultaneously. Without proper synchronization, this can lead to unpredictable results and application errors.
- Deadlocks : These occur when two or more threads are blocked, each waiting for the other to release a resource. This can lead to threads sitting in an indefinite waiting state, affecting the system's ability to perform tasks.
- Resource leaks: If a thread does not properly release the resources it has acquired, this can lead to a gradual depletion of resources, which will affect system performance and the ability to create new threads or processes.