Paul Kummer

Dr. Hanku Lee

Operating Systems CSIS 430-01

09/16/2021

Summarizations of Chapter 2

2.1 Processes

With the use of any computer, processes are used to do any work on the computer. A process is simply a program that can run on the computer. More specifically, a process, for a very brief period, will have complete use of the CPU to accomplish its task. To users, it may appear that there are several processes running at the same time, known as pseudo-parallelism or multiprogramming, but the computer only rapidly switches between processes for the CPU. The exception is for multiprocessor systems that have multiple cores, but for simplicity, the summarization will assume one CPU unless specified otherwise.

2.1.1 The Process Model

To create the illusion of many processes running in parallel, multiprogramming is used. When this is done, each process has its own virtual CPU. It will have its own program counter, registers, and variables. Therefore, each process will run on one CPU, but it has its own information loaded when the scheduler gives it access to the CPU.

When a process runs, it is executing code from a program. With multiprogramming, the code that is being read from a program can be interrupted so another program can run. The same thing will happen with this new program as the scheduler gives control to another process. This will continue to happen until there are no processes requiring the CPU, which is never. The key difference between a program and a process is that a process is the execution of the program and could be at any line of code within the program. The program is all the lines of code that when completely executed will accomplish a task.

There are some processes that are time critical and must be processed in real-time. The execution of videos with audio on a separate channel is one of these instances. If the audio starts playing at the wrong frame of the video, the sound will be out of sync with the video, and it will not be enjoyable to watch. To overcome this obstacle, the process must have idle loops that execute for specific durations of time to ensure that the audio and the video play together.

2.1.2 Process Creation

Processes are created for several reasons. Four main reasons for a process to be created are system initialization, a running process calling a process-creation system, a user requesting a process creation, and initiations of batch jobs.

As computer starts up, processes are automatically started and some run in the background. The processes that startup and are hidden from the user are called daemons, and they typically are email clients, antivirus software, or software updaters. These daemons are needed for programs to stay current and for other software to run correctly. For instance, an email client may startup a process that just listens on a port to detect new mail and when it does detect new mail, it alerts the user.

A running process can create other running processes too. In UNIX this is accomplished through a method called fork. Having a process create other processes can be useful if a larger process exists that needs to transform some data and retrieve data at the same time. Then one process can work on processing the data while the other is retrieving the data. If this is done, and multiple CPUs exist, the time for a process to finish can be greatly improved.

The process creation that most users are aware of is a process creation through direct interaction from the user. Icons on a desktop of a GUI operating system do exactly this when a user clicks on them. The user clicks the icon, and a process is created that begins executing the program. Many of these processes can be interacted with by the user, and it is typically what people think of when they use a computer.

Finally, processes can be created for batch jobs, which are typically found on large mainframes. There are many processes that need to occur when inventory management is done, and the daily sales need to be computed. To accomplish this, the computer will have a queue of processes that need to occur and create them whenever resources are available.

Whenever a process is created, whether its with fork or CreateProcess, it has its own address space. This is true for the parent and child processes. Even though the child process is identical to the parent process, it has its own address space. It is possible for processes to share non-writable memory, and it is called copy-on-write. If any changes need to be made to the memory, it is copied from the shared memory to its private address space. Therefore, no changes from one process can affect another processes address space.

2.1.3 Process Termination

Processes usually terminate when they have finished performing their task, which is a normal exit and is voluntary. They can also finish voluntarily through an error exit, occurring at runtime. For example, if a user attempts to open a word document that doesn’t exist in the specified directory, the process will terminate and typically state the error that occurred.

There are two other involuntary ways for a program to terminate through a fatal error or another process killing a process. A fatal error can occur if the process begins execution, but for some reason the process attempts to do something that isn’t allowed, such as dividing by zero or accessing a memory address out of bounds. A process can also be terminated, killed, by another process with higher privileges. For instance, say a process is hanging and using too many system resources, a user may start the task manager in Windows OS and force terminate a process.

2.1.4 Process Hierarchies

Processes can create other processes. This is true for the newly created processes too, allowing a process to have many or no children. When this occurs, there becomes a process tree. Each process has its own address space, but they are still all part of the same group. Groups will receive all the same signals of what they are associated with. An example is a keyboard sending out signals to a group of processes associated with the keyboard. In the Windows OS, there is an exception to hierarchies. Windows has a special token, handle, that allows a parent to control its children, but this token can be passed to other processes, allowing disinheriting.

2.1.5 Process States

A process can be in one of three different states. These states are ready, running, or blocked. When a process is ready, it is waiting for the scheduler to allocate the CPU to it so it can begin processing but is currently doing nothing. If the process is running, it has the CPU allocated to it and is carrying out its instructions. The last state a program can be in is blocked. In this state, it is waiting for some external event to happen so it can run. An example of an event that causes a process to be blocked could be the process needing some input from the user before it can process any more instructions. Once the proper event happens for a blocked process, like the users entering input, the process can move to the ready state.

2.1.6 Implementation of Processes

Since many processes can run at the same time, the operating system must keep track of all the processes. To do this, a process table is made that contains the state of all the processes currently running. The information about a process in the table consists of its program counter, the stack pointer, memory allocation, status of open files, accounting and scheduling information, and anything else relating to the process. All this information is important for the process starting from wherever it left off if is being restarted. If it is a new process, it will be at its starting point.

The fact that processes have all their information stored in the process table allows for the use of interrupts. When a process is running, a hardware interrupt can be called from the interrupt vector to request access to system resources. With this interrupt, the currently running process will save all its updated information into the process table so that when it can be run again. After the interrupt is finished, the scheduler will load the next ready process to be run.

2.1.7 Modeling Multiprogramming

Without multiple processes being able to run in pseudo-parallel, the CPU would be sitting idle. If the CPU is idle for too long, it is an inefficient use of its abilities. Multiprogramming overcomes this inefficiency by allowing another process to use the CPU if one process is blocked, awaiting an event to occur. Therefore, the more programs that can be loaded into memory, the more efficient the CPU can become because it will have more ready processes to select from. However, the law of diminishing returns applies to memory. At some point the percentage of improvement will begin decreasing as more memory is added, decreasing the return on investment.

2.2 Threads

Threads are almost identical to processes, with one notable exception. One process can consist of many threads. This means that multiple threads can use a shared address space, which processes cannot.

2.2.1 Thread Usage

Some of the same reasons threads exist are the same reasons multiple processes exists. Multiple threads within one process allows parts of the process to continue while one thread might be waiting for another thread to finish some instructions. Therefore, instead of a process completely stalling out, the process can continue running. The speed gains are only noticed if some of the threads rely on the I/O systems. Also, because the threads share an address space, they all have access to the same information, which processes cannot do with other processes. Additionally, threads can be created and destroyed very quickly, allowing faster speeds of execution than a process.

Suppose a programmer is writing a program that is thousands of lines long. Now the programmer decides to change a variable name. Without the use of threads, after the programmer changes that variable name, he/she will have to wait while the editor changes all the variables to the new name. Furthermore, the editor will also have to check for conflicting names against reserved keywords after the change too. The user would then have to wait quite a while until they could resume writing code. However, threads do exist, and the programmer can replace that variable name with a new one and continue writing new code. This is because one thread is going through the document loaded in the address space changing all the occurrences of that variable while another thread is handling the input from the keyboard to write new lines of code into a buffer. This is not possible with processes because there is no shared address space, so one process can't work on the same document that another process is working on.

2.2.2 The Classical Thread Model

When thinking about the concept of threads, its best to think of them as a bunch of independent instructions that all require access to the same resources as files or data in memory. All threads must execute within a process, which is where the shared resources reside. This is like how processes share physical memory and access to printers and monitors. Since threads and processes are so similar, threads are sometimes called lightweight processes. Also, modern technology often supports multithreading, which allows threads to switch rapidly just like processes.

2.2.4 Implementing Threads in User Space

Threads can be created in two different spaces. They can be in the user space or the kernel space.

In the user space, the operating system is not aware of the use of threads. This is convenient if the OS doesn't support threads. When threads are done in this method, there must be a library loaded that adds the support for threads. The first threaded programs to exist were done this way. Every process must keep track of its own threads in a thread table when the process is in the user space.

Threads in user space are inherently faster than threads in the kernel. They do not have to flush memory or trap the kernel when one thread is switched for another. This prevents wasting system resources and avoids system calls. However, threads do have a downside. They can cause page faults when an instruction isn't in memory, causing the process to be blocked while the instruction is fetched. Also, threads can maintain control of the CPU in a process indefinitely, not letting other threads run. There are some applications of threads where the benefits of using the thread are completely negated because thread blocking is seldomly used and the application is mostly bound to the CPU.

2.2.5 Implementing Threads in the Kernel

Threads ran in the kernel space have information about all threads stored in a thread table managed by the kernel. Only the kernel will make or destroy threads. It is very similar to how the kernel maintains a process table for all ongoing processes. Furthermore, when a thread makes a call that could block, it must be made as a system call. The use of system calls makes kernel threads much slower than user threads.

Since making and destroying kernel threads is so expensive, kernel threads may be recycled. Instead of destroying an unused thread, it is made nonrunnable so that when another thread is needed, the new values can be put into an old thread that is made runnable. By doing this, some of the overhead of making and destroying threads is reduced.

2.2.9 Making Single-Threaded Code Multithreaded

Turning a single threaded program into a multithreaded program might be desirable to gain improved performance. However, performing this task can bring some problems to light. One major problem is with global variables. If a thread writes to a global variable that then gets overwritten by a different thread, the global variable may be an incorrect value for other threads. There are ways to create private global variables confined to a thread, but it isn't as straightforward and traditional global variables. Other similar problems arise because libraries not designed for multithreading can send signals that are not specifically directed to a thread. Therefore, many threads may try to do something they were not intended, or the wrong thread does something designated for a different thread. Threads sharing an address space can be difficult to manage when a program wasn't designed with these concepts in mind.

Features of Processes and Threads		
	Processes	Threads
Memory Space	Independent	Shared with a process
Blocking	A blocked process does not stop other processes from running	A blocked process will stop all threads of a process from running. This can happen if a page fault occurs.
Global Variables	No conflicts will occur with other processes since they have different memory spaces.	Conflicts can occur with other threads because they share a memory space, allowing a thread to change a global variable that should not be changed.
Switching	Slows down the performance of a program because it requires system calls. Does allow the use of a clock interrupt to make a process give up the CPU.	Does not require system calls and can quickly switch to a different thread. However, it does not use a clock to force a thread to switch.
Creating/Termination	Requires system calls and is costly.	Does not require system calls and can be performed quickly.

Differences of Threads v. Processes		
	Processes	Threads
Independent address Space	Yes	No
When blocked, stops the process	Yes	No
Contains its own variables	Yes	No
Traps the kernel on switch	Yes	No
Supports custom scheduler algorithms	No	Yes
Supports clock interrupts	Yes	No

Works Cited

Tanenbaum, A., &amp; Boschung, H. T. (2015). Modern operating systems (4th ed.). Pearson.