Paul Kummer

Dr. Hanku Lee

Operating Systems CSIS 430-01

10/08/2021

Summarization of Chapter 2: Processes & Threads

2.1 Processes

2.1.1 The Process Model

A process is similar to a program, with the main difference being that a process is loaded into memory and is being executed while a program is lines of code stored on permanent storage. Processes can be run sequentially or in a pseudo parallel, which is called multiprogramming. With multiprogramming, control to processes will be switched back and forth rapidly, making it seem like they are executing together. Each process will have its own address space, which can be saved when another process needs to execute.

2.1.2 Process Creation

Processes can be created by system initialization, a system call from a process, the user requesting a process creation, and the initiation of a batch job. When a computer starts, many processes are started just to initialize the operating system. Some of these processes will run in the background and are referred to as daemons. They may be responsible for detecting incoming emails. A process may also start other processes. Processes are most notably started by the user when they execute a specified program. Then the program is loaded into memory and a process is started to execute the code. There are also situations where consecutive processes are performed to accomplish some larger goal like an end-of-the-day report of a retailer.

2.1.3 Process Termination

A process can be terminated in four different ways. The first and most desirable way is by the program finishing its execution and voluntarily closing the process. Processes can also terminate when there is an error detected in the code. For example, if an attempt to open a file that doesn't exist happens, the program may state that the file is not found and end the process. Another form of termination caused by an error is by bad programming where an exception isn't handled properly. This could be a memory address being accessed that has no value or indexing an array index past its range. The program will do what it is told, but the data will be bad and force the process to terminate. Lastly, a process can be terminated by another process. For instance, if a process is supposed to connect to a socket, but is not getting a response, the process may infinitely loop trying to connect. Another process may see that the process is stuck and terminate the hung process if it has adequate permissions.

2.1.4 Process Hierarchies

A process hierarchy is a tree structure where every process can be traced back to a root process. Therefore, every process will have one and only one parent. The method of creating a

process tree is by processes using system calls to create a new process. This can be achieved in Unix by using fork(). In the Windows OS, process hierarchies do not exist because a parent can disinherit its children and pass a token/handle to other processes.

2.1.5 Process States

In the OS there is a scheduler that picks processes to have access to the CPU. This allows each process to have a chance at accomplishing their tasks without waiting long periods of time. For the scheduler to pick a process to run, the process state is looked at. Each process can be in one of three states of running, read, or blocked. When a process is ready, it indicates to the scheduler that it would like access to the CPU, and it is an eligible candidate to run. If a process is picked to run and is currently using the CPU, its status will be running. When an interrupt signal is sent, the currently running process's status will be changed to change blocked. When this occurs, the address space of the process is saved so it can be loaded back into memory when it is called again by the scheduler.

2.1.6 Implementation of Processes

The management of processes requires the use of a table to keep track of all the processes. That table is called a process table or process control blocks. It will maintain the current state of all the processes, the process's stack pointer, the process's program counter, and any other information associated with a process.

The storage of process' information in a process table is important for processes to be able to change states. An interrupt vector gets signals from an interrupt service procedure that, when called, causes the current process to have its current values saved to the process table. Then a new process will be loaded and begin executing. With the combination of the process table and interrupts, processes can be switched to and from CPU control in the exact state of when they last had control.

2.1.7 Modeling Multiprogramming

If a computer ran two processes concurrently, the CPU may be idle for a very long time. For instance, if the first process requires a user input, but the user doesn't enter anything. The current process will be waiting for input. Then the CPU will only be utilized when the second process is running. To increase the CPU performance, probabilistic statistics can be used to determine the best number of processes to run concurrently to maximize the use of the CPU. With the addition of every process, there is slightly more overhead, but less chance the CPU will be idle. Specifically, the more processes that can be loaded into memory is dependent on how much memory is available. So, the overhead can be directly tied to a machine's physical characteristics. At some point, adding more memory will not increase the performance much, and may be a waste of resources.

2.2 Threads

2.2.1 Thread Usage

Threads are like processes in the aspect that each thread and process both have set instructions that they will carry out when they have access to the CPU. There are some significant differences though. One major difference is that processes are completely independent from each other. That is, one process cannot access data from another process.

They have different address spaces with different program counters and stack counters. In contrast, a thread is part of the same address space of possibly many other threads. This allows threads to share variables among each other.

There are a few advantages to threads over processes. The first was already mentioned, and it is the shared address space. The second is the ability of threads to be quickly created and destroyed. They do not require special system calls to make or destroy them, allowing threads to change quickly. Another important advantage is noticed by users when a process involves I/O operations. If a process is waiting for input without threads, the process will do nothing until it gets input. However, with threads in a process, one thread could be stalled waiting for input while another thread is busy accomplishing a task. This can greatly increase CPU utilization. With true multi-processor systems, each thread can be assigned a CPU and even greater CPU utilization can be achieved.

2.2.2 The Classical Thread Model

Traditional threads have two separate ways to be viewed. One as a collection of related resources that are grouped together. The other way is thread execution, or just thread. A thread can be viewed as a block of code scheduled for execution.

When a process has multiple threads, the programming supports multithreading. With classical programming processes and multiple threads per process. Each process will have the threads take turns executing. This is very similar to how a scheduler will have the processes take turns using the CPU. Also, much like a process having an address space separate from other processes, threads also have some private data only available to that thread. The date includes a program counter, registers, stack, and state.

Typically, when a process is created it will have a single thread. This thread then can create more threads. Threads optionally can have a hierarchical structure. They can destroy themselves, make more threads, wait for other threads, or voluntarily give up CPU time.

2.2.3 POSIX Threads

Without standardization, it would be difficult to implement threads across many platforms. Therefore, the IEEE created the standard 1003.1c. In that standard, Pthreads, a package with many calls that can be done on threads. Most UNIX systems support the IEEE Pthreads package.

2.2.4 Implementing Threads in User Space

When threads are run in user space, the kernel doesn’t know of their existence. The kernel only sees processes and assumes they are single threaded. All management of threads must then be done within the process. This allows threads to be used in an operating system that doesn’t have any support for threads. Also, threads have information stored about them in a thread table that resides in a process. This thread table will behave similar to a process table storing information about processes, only it will be threads instead.

Threads in user space have an advantage of not trapping the kernel. This allows them to be faster. However, since the threads in user space to not use the kernel to make system calls, there may be circumstances where they need to make a system call. One way of doing this is wrapping a system call in another procedure, called a jacket or wrapper. Another disadvantage

is that if a thread causes a page fault, the process will be blocked, making all threads stop until the hardware returns what it was requested to fetch.

2.2.5 Implementing Threads in Kernel Space

Threads in kernel space operate like threads in user space. They both use the same address space as the process they are running in. However, when threads are run in kernel space. The cost of them to switch, be created, and be destroyed is much greater because they require system calls that are inherently slow. Also, the thread table is managed by the kernel too, much like a process table. This is good and bad. The threads can be switched and run when a page fault occurs, but, once again, the kernel must manage the threads and is slow. To make kernel level threads more efficient, threads structure may be left intact upon destruction and reassigned to a new thread instead of starting from scratch.

2.2.6 Hybrid Implementations

There is also a hybrid implementation of threads that utilizes both kernel and user level threads. This allows the flexibility of choosing what kind of thread is best for a given circumstance. This implementation may have one kernel level thread that spawns into multiple other threads in the user space. The kernel is unaware of the user level threads and only manages the kernel threads. This is like the movie Inception where there are dreams inside of dreams and people are unaware that they are in a dream instead of reality.

2.2.7 Scheduler Activations

Scheduler activations is a hybrid implementation strategy to get the best out of user threads and kernel threads. If possible, the threads will remain in user space because of their rapid speeds at changing threads and other thread related management. However, when an interrupt occurs that stalls user level threads, the kernel will begin to manage threads so that not all threads are stopped while the interrupt is occurring, like a page fault. By doing this, threads will be able to run even when an interrupt happens by allowing the kernel to select a new thread to run.

2.2.8 Pop-Up Threads

With traditional threads on distributed systems a thread or process waits for incoming messages to receive and process them as they arrive. However, this can be achieved in another way with the use of pop-up threads. A pop-up thread is a thread that is created whenever a request is received, and the new thread is given all the information of the original request. This allows many threads to be processed simultaneously without having to wait for prior threads to finish before their processing begins.

2.2.9 Making Single-Threaded Code Multithreaded

Code that was originally designed to be run on a single thread can be converted to multiple threads. However, there are a few drawbacks. One problem arises because threads can all access variables within their address space. Some of the variables should not be treated as a global variable. Therefore, if the code isn't converted correctly, some procedures may be able to manipulate data they shouldn't. The problem can be avoided by making some variable contained within a procedure. There can also be problems with buffers being overwritten, interrupts, and threads stacks overflowing. All these issues must be addressed and can be quite complicated.

2.3 Interprocess Communication

2.3.1 Race Conditions

With interprocess communication, there can be occurrences where two separate processes try to write to the same memory address. This could happen in an unlikely event and cause one of the processes data to be overwritten by a different process. When two or more processes are both trying to write to a same address at the same time, a race condition occurs, and the information that the processes are working with could become corrupted.

2.3.2 Critical Regions

To prevent race conditions, areas of some procedures need to be identified as a critical region. A critical region contains variables or data that only one process should read or write at a time. When a critical region is identified processes need to be provided mutual exclusion to the critical region.

2.3.3 Mutual Exclusion with Busy Waiting

Once the critical regions are noted, process must be given mutual exclusion to critical regions as they read or write the data in that area.

There are a few ways to grant mutual exclusion. For single processor devices, interrupts can be disabled, preventing any other processes from running until the running process finishes. This method isn't very desirable because multi-processor devices are common and if the interrupts are not reenabled, system failure will be the result.

Another choice for mutual exclusion is with lock variables. This will hopefully stop processes from entering a critical region if a variable is set to say that another process is in the region. This does have a fatal flaw of the race condition though.

Similar to the lock variable strategy, the strict alternation strategy can avoid a race condition in accessing the critical region. With this strategy, each process will take turns entering the critical region and then give the other process control when it leaves. Busy waiting will be done by the process not in the critical region, which means it will continuously test for a condition that gives it access into the region. This is a bad solution because it assumes that the processes will need to go back and forth in the critical region. One process could be theoretically blocked forever from entering the critical region if the other process no longer needs to enter the region.

To combat some of the problems or race conditions and permanent blocks to a critical region, Peterson's solution can be used to grant a process access to the critical region. It uses a function that a process must call with its process id to request access to the critical region. If no other processes make a request at the same time the process will be granted permission otherwise the function called last will be the one granted access.

An approach to making region mutually exclusive involves hardware and locking a memory word. This word is used as a variable that cannot be changed until it is unlocked. So whatever process sets the signal to locked will be given access to the critical region while the other process wanting to use the critical region will have to be busy waiting for the signal to change.

2.3.4 Sleep and Wakeup

The previous solutions to mutual exclusion all involved processes using busy waiting that wastes CPU's time. Instead, processes may be blocked so they don't have to do any processing while they await access to the critical region.

A solution is to use a buffer and producers and consumers. The producers will keep producing while the buffer is zero and the consumer will keep consuming when the buffer isn't zero. If the buffer goes to zero, the consumer will be put to sleep. If the buffer is full the producer will be put to sleep. However, there is also a race conditions that can cause this system to fail. One fix is to use a walkup waiting bit, which serves as a buffer to prevent signals from being lost and both processes sleeping indefinitely.

2.3.5 Semaphores

Semaphores are variable types that are used to save the number of wakeups pending or saved. Before a process can be put to sleep an atomic action is done that checks if there are any wakeups that need to be done and modifies the semaphore if needed. This will prevent a race condition. The semaphore is accessed and changed with down or up, allowing processes to sleep or wakeup. To make the Semaphore atomic, block system calls must be done.

2.3.6 Mutexes

A mutex is a variable that is a shared resource and can be locked or unlocked. Once a variable is locked, all other threads trying to access the variable are blocked until the current process in the critical region unlocks the mutex. To avoid busy waiting a thread can call yield when it test the lock and fails.

Linux has a feature called a futex that is a fast user space mutex. It primarily operates in user space user space unless it absolutely needs to run in the kernel space. When a thread has to wait, the kernel puts the thread in a queue instead of the thread spin locking.

To make programming easier, a library called pthread can be used that provides threads with many options. a pthread can make a mutex, destroy a mutex, acquire a lock or block, acquire a lock or fail, or release a lock. They can use conditions to determine when a critical region can an cannot be entered. When condition variables and mutexes are used together with pthreads, a truly mutual exclusive region can be setup that doesn't waste CPU time and allows threads to continue working.

2.3.7 Montiors

Monitors are another way of solving the mutual exclusion problem. With monitors, code is grouped together into one common block, called the monitor The monitor has an interger and a condition saved to the monitor object. Nothing within that object can be modified when something else is accessing it. Once a thread is done with, he monitor it will send a signal changing the condition of the monitor. Monitors rely on the compiler to setup mutual exclusion, so no sleep or wakeup calls are needed to be done directly. The nature of a monitors prevents race conditions because other threads are automatically blocked when it is in use. In java the synchronization keyword can be used in front of a class to guarantee only one thread is using a monitor at a time.

2.3.8 Message Passing

Message passing utilizes system calls for interprocess communication. One process will send a message to another process and the other process will receive a communication. Much like network communications, a sender will wait for acknowledge signal to send more information. There are also numbers assigned with each message, therefore duplicate messages can be ignored if a message is resent.

The producer and consumer problem can use message passing and a mailbox. Messages will be sent to the mailbox and a consumer will read the messages from the mailbox. If the sender and receiver are synchronized, they can use a rendezvous instead of the mailbox to send a message directly to the receiver.

2.3.9 Barriers

Barriers are used to block processes until all other required processes have reached a specified point. This is important if there are phases to a program that require some processes to have settled on some final values. Once a desired number of processes finish their tasks the blocked processes will be unblocked and allowed to continue.

2.3.10 Avoiding Locks: Read-Copy-Update

If possible, it is ideal to not use locks. However, this is not possible in many circumstances. In some situations, this can be done. The use of Read-Copy-Update allows data to be read without locks. However, readers will be allotted a grace period to prevent indefinite reads in the read-side critical section.

2.4 Scheduling

2.4.1 Introduction to Scheduling

Schedulers are an important part of an operating system. They decide when a process gets to run. Old computers had limited resources, and they had to be used wisely. Now computers are a lot faster, and the delays are not as noticeable when there are inefficiencies. However, choosing which process should run when and for how long is still important. Now scheduling may be done to prolong a devices battery life instead of processing data as fast as possible, maximizing use of the CPU.

Processes can typically take two different forms in terms of scheduling. They can be either CPU-bound or I/O bound. The CPU-bound processes are primarily computing data the whole time they are running. the I/O bound processes are the exact opposite. They spend most of their time awaiting some input.

Scheduling can take two forms of preemptive or nonpreemptive. Nonpremptive scheduling allows a process to “run until it blocks or voluntarily releases the CPU” (Tanenbaum 153). The preemptive algorithm has a fixed amount of time a process can run before it is blocked. Furthermore, scheduling algorithms can be categorized as batch, interactive, or real time. Based on the category, the scheduler will have different goals of how to pick the next process to run. Some commonalities between all the categories are fairness, policy enforcement, and balance. Batch Systems will require throughput, turnaround time, and CPU utilization. Interactive systems will require response time and proportionality. Real-time systems will want to meet deadlines and have predictability.

2.4.2 Scheduling in Batch Systems

The simplest form of scheduling is with first-come first-served. The first process to request the CPU will get it and continuing using it until it is finished.

Another nonpreemptive strategy is shortest job first. The process that has a calculated shortest running time will run until it is finished before longer processes. All jobs runtime should be known ahead of time for optimal performance.

A preemptive option is shortest remaining time next. This algorithm will pick the process with the shortest remaining time to run next.

2.4.3 Scheduling in Interactive Systems

With interactive systems, there are seven different scheduling options Round Robin, Priority Scheduling, Multiple Queues, Shortest Process Next, Guaranteed Scheduling, Lottery Scheduling, and Fair-Share Scheduling.

1. **Round Robin:** Every user gets a set quantum to run before the next process
2. **Priority Scheduling:** Processes with more importance are run before lower importance processes.
3. **Multiple Queues:** Processes move to different queues as they use up their quanta and each queue has a different quantum amount.
4. **Shortest Process Next:** An estimate of processes running time is made with a calculation called aging to determine what process has the shortest run time, and it is selected.
5. **Guaranteed Scheduling:** Based on the number of users, each user is guaranteed 1/n amount of the processor's time.
6. **Lottery Scheduling:** Users are assigned tickets that give them a chance to be run. More tickets increase the odds of being chosen.
7. **Fair-Share Scheduling:** Users get a set amount of CPU time no matter how many processes each user has running.

2.4.4 Scheduling in Real-Time Systems

Scheduling in real-time can have to different meanings. There are hard real times and soft real times. Hard real time has absolute deadlines that must be met by a certain time. Soft real time has deadlines that can be missed with little consequence. Real time processes can also be periodic or aperiodic.

2.4.5 Policy Versus Mechanism

The scheduling mechanism strictly picks what processes it thinks are the best processes to be run at specific times. A scheduling policy helps the mechanism pick the best process by changing process's children's scheduling priority.

2.4.6 Thread Scheduling

Thread scheduling is very similar to process scheduling. However, because threads can exist in user space, the kernel doesn't know about the threads and only cares about the process. Therefore, all thread scheduling will take place in the user level and the kernel will interrupt the process to switch to a different set of threads. With Kernel level threads, if a processes threads yield, it's possible that threads from another process are ran before the first process used its quantum.

2.5 Classical IPC Problems

2.5.1 The Dining Philosophers Problem

The Dining Philosophers Problem demonstrates how if all processes request certain resources at the same time, a program can fail. It shows that there needs to be mutual exclusion to certain parts of a program so that only some process can use resources at a time. Therefore, in the problem, only two philosophers that are one philosopher apart should be granted access to forks to eat their slippery spaghetti. If this doesn't happen the philosophers could starve to death because they will never possess two forks at one time.

2.5.2 The Readers and Writers Problem

There are many manifestations of the Dining Philosophers Problem that occur in everyday life. One common occurrence is airline reservations. People must be able to read correct data about available flights and seats and people must also be able to assign themselves to planes and seats at the same time. Therefore, multiple processes that read should be able to access data, however, if a write needs to occur, no reads can take place. Priority must be given to a writer in some way so that a stead stream of readers cannot prevent a write indefinitely. Some users before the write may have inaccurate data, but you can't make everyone happy.

Works Cited

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