OPERATING SYSTEMS ORGANIZATION
MSIT 5170

University of Massachusetts Lowell
Department of Computer Science
Spring Semester 2020

Week #4 Lesson

1. The material in this lesson follows the reading assignment for ch 2.1 through the chapter 2.2 from the Tanenbaum text
2. Please read the text before you read through this lesson.
3. The material in chapter 2 will focus on process and thread details, with emphasis on process life cycles and thread states.
4. We continue to look at system level tools, and I encourage students to experiment with these tools. We will also build some simple applications in the Linux environment to explore some process and thread elements.

Processes and Threads:

Last week I introduced the construct of an operating system process, identifying it as the primary unit of organization within an operating system. We will now make a detailed examination of a process. Remember, a process is a living, dynamic entity that encapsulates a computation. Processes don't actually execute, but they gather the resources necessary for their resident program to be executed by one or more process threads. All threads belong to some process in the system, and it's always a thread that is dispatched to a CPU (central processing unit). Threads execute their machine instructions (the instructions forming part of their process address space), allowing the program within the process to make computational progress. A program is simply a passive collection of code and data, and to execute a program typically requires a process to be created, the program to be loaded into the process, and a thread component of the process to be dispatched to a CPU to begin executing the programs code.

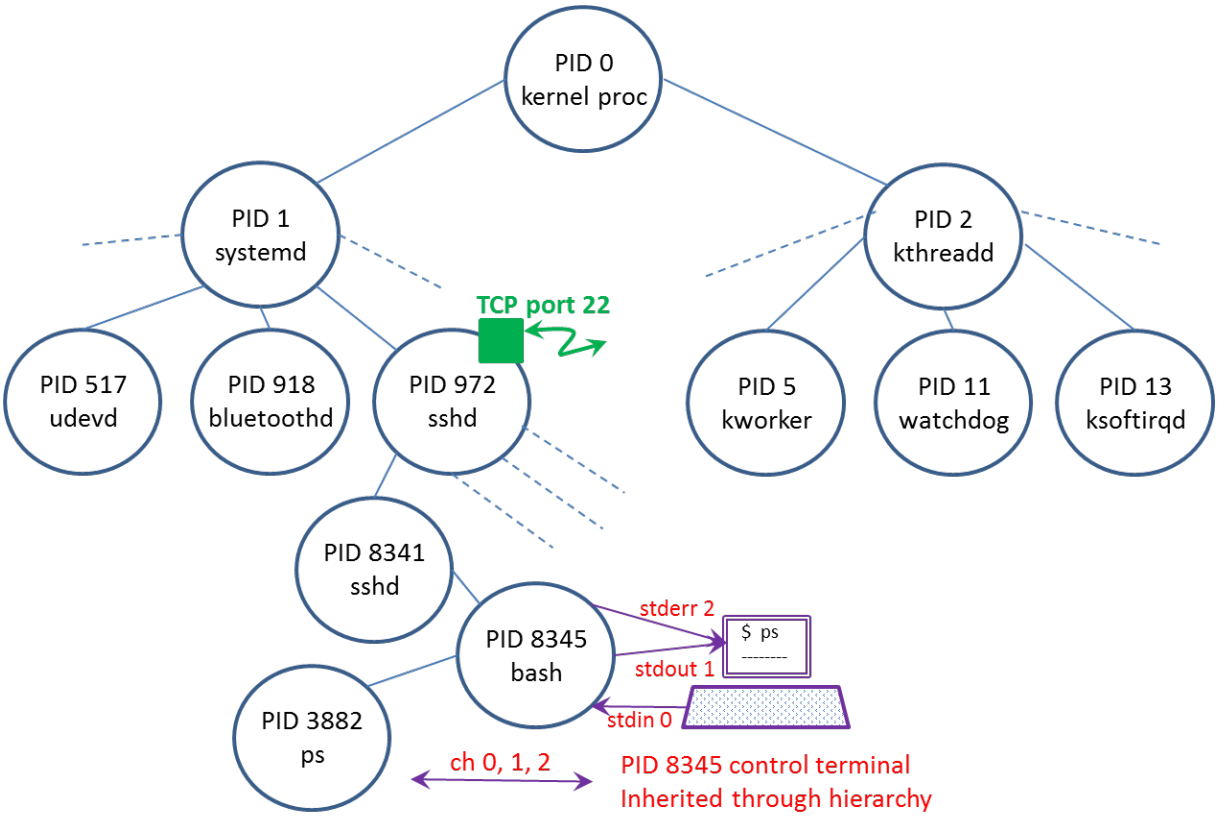
I will demonstrate much of our process specifics around the process model used in the Linux operating system environment, but you should understand that Windows processes are much more like Linux processes than they are different, and all of the important ideas that we look at through Linux glasses, apply in virtually the same way in the Windows world.

Operating System Startup:

The action of bringing an operating system to life from a cold start is typically called booting the operating system. Boot time activities vary from one system to another, but in Linux (and most UNIX based systems) the activity involves the only time during the systems life that code will run outside of the process construct. There is only monolithic boot code to execute while the system is configuring the environment to accommodate the operating system to come. At the end of the boot sequence, the monolithic boot code transforms itself into the first system process (in Linux this is a kernel process with no user scope), and this first process then creates

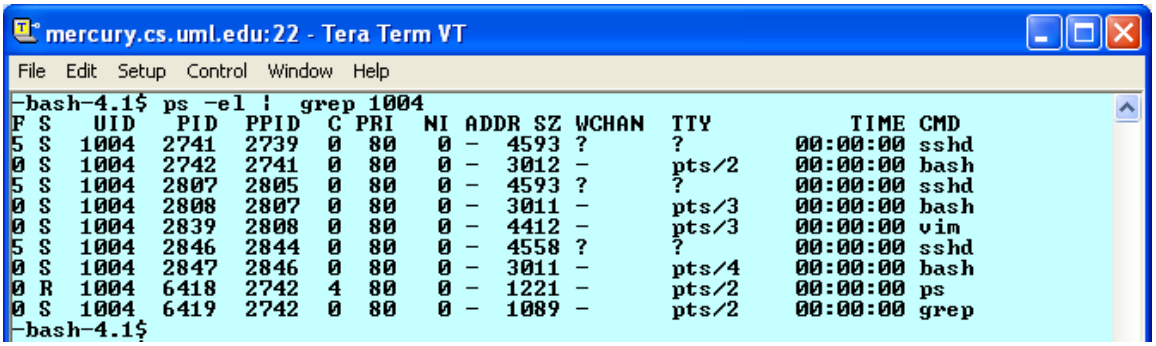
the first 2 user visible processes, the first with PID 1, running a program called systemd (init in older versions of Linux/UNIX) , and the second with PID 2 , running a program called kthreadd. These two initial user processes are immediate descendants of the boot time kernel process PID 0, and each of these processes will have a role in bringing the rest of the system to life. Once PID 0 is created, all other process creation will be done by some existing thread that, like all threads, belongs to some existing process. The main role of PIDs 1 and 2 is to create several more processes that have a part in maintaining the system and providing various services. These processes are often called daemons, based on the fact that they are started by the initial boot time user processes, and not by some user that has logged into the system.

Daemon processes are not really different from any other process, but their credentials often provide them with more privileges than processes that have been started from user login sessions. Many of these daemons have a UID (user ID) of 0, meaning that they are running with the credentials of the root user (superuser), and therefore have virtually unlimited access to all system components. The concept of a process with root credentials is similar to the concept of a process with administrator credentials in a Windows system. In the Linux system, all process have a parent child relationship, forming a tree with the kernel process PID 0 at the root. So every process is related in some way to every other process in the system, since they all have the same grand ancestor. Windows does not visibly maintain this sort of process tree, but it does keep track of all child-parent relationships in the kernel process data structures.



The diagram above shows a collection of processes that includes some of the early processes created at boot time from an actual Linux system. Most of the processes are running with root credentials and serve various daemon functions. I have expanded one of these daemons shown with PID 972 and running the sshd (secure shell daemon) program. A thread in this

process monitors TCP port 22 (the inbound port for ssh connections), and whenever a connection request shows up (from some outside client that is trying to log into the system over the network), PID 972 creates a child process to handle that request. Like its parent, the child (here shown as PID 8341), is running with root credentials, and has the ability to look into all system data in order to authenticate the requesting connector (validate the user name and password). While PID 8341 retains its root credentials, it spawns its own child to handle this user, changing the credentials of this child (here PID 8345) to those of the authenticated user, thereby limiting access through this process to only what the user credentials allow. PID 8341 checks this user's account data, and finds that the user wants to have a process created for it at login time that is set to run the bash shell program, so PID 8341 creates a child (here PID 8345) and loads that child process with the bash program, providing the user with a command line environment in a dumb terminal emulation. PID 8341 created the pseudo control terminal before creating the child that would eventually become the bash process.. The actual user, in this case, may be running an ssh program like putty or Tera Term from a Windows system. These are freeware programs for Windows that provide a dumb terminal emulation screen, and can be used to connect to a Linux host as described above. Here is a screen shot of my Tera Term session interacting with a process on a Linux host running the bash shell. I've asked the bash shell to create a child process to run the ps program (you can see the ps process (PID 6814) itself listed among the various processes that I currently own on this system. In this example, I have asked the bash process to actually create 2 children, the ps child and the grep child. The ps child process will generate a lot of data (there are currently 250+ processes running on this host), but that output is sent to the grep process for filtering. I've asked grep to only report lines that have my user ID on them (my UID is 1004). You can see the results, including the sshd login process shown in the diagram above (there are actually 3 sshd processes here because I've used ssh to log into this system from 3 different terminal emulation windows).

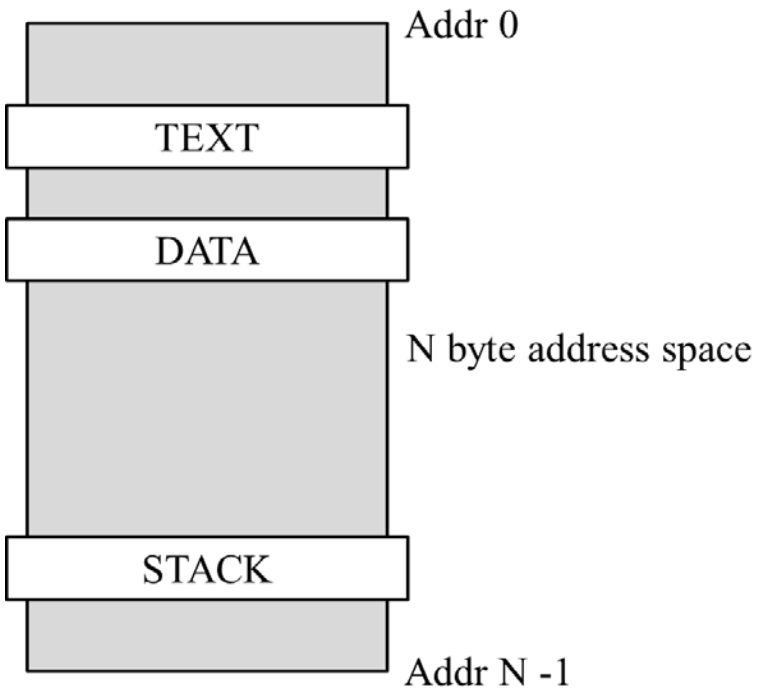


The output presented by ps includes my user ID (UID), the PID of each process shown, the PPID (parent PID) of each process, and the name of the program that each process is running in the far right column. Other fields provide additional per-process information that we will look at later. First, we want to revisit the elements that comprise a process, and examine more details.

Process Attributes:

On Linux platforms we can expect a process to be comprised of a number at attributes, some of which we have already seen, and some of which we will discuss in more detail below.

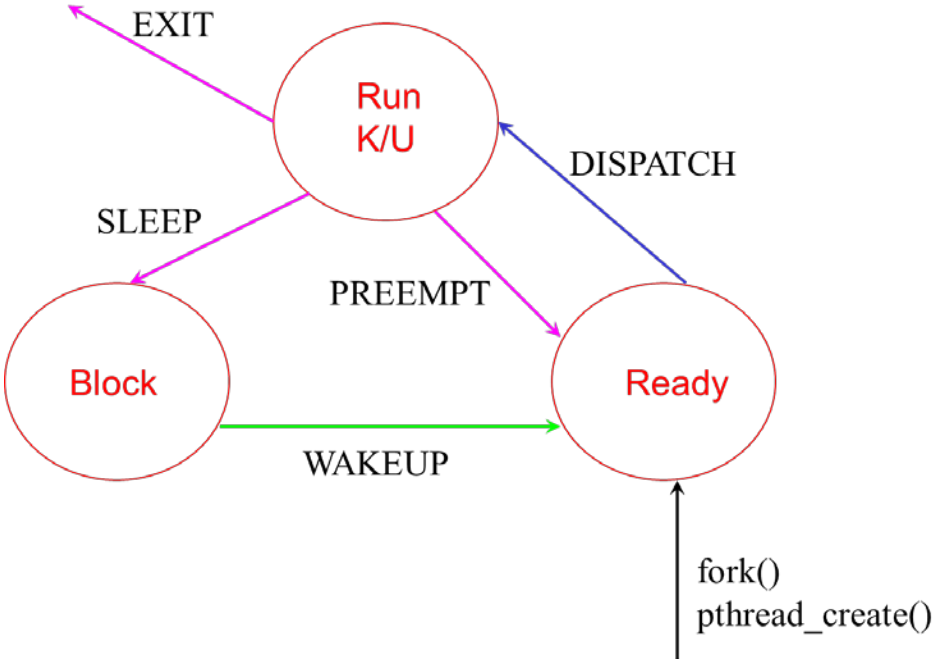
- A process ID (PID)
- A parent process ID (PPID)
    - Each process, with the exception of the initial kernel process PID 0, must have a parent PID. If a process should find that its parent has terminated before it has, then the process will be adopted by PID 1. Any process will either have a PPID of its original parent, or that of PID 1.
- An Address Space
    - Each process in a Linux system is given a private address space that consists of some collection of contiguous virtual linear addresses that the process may populate with memory objects. The size of this space is a function of the underlying hardware that the operating system is running on. A Linux system running in 32 bit mode on an x86 platform will normally provide each process in the system with a 3 GB address space.
    - A process populates its address space with memory objects, including:
        - A Text object that includes the program code for the program running in the process
        - A Data object that contains global variables and dynamic heap allocation space needed by the process
        - A Stack object that manages the local variables and activation records of a thread of the program (must have one stack per thread).
    - The memory objects described above are not the only memory objects that may be found in a process address space, but a process cannot exist unless at least these three are found in the address space.



- Credentials

- A process has identity credentials, which serve to mediate the access that the execution elements (threads) of the process have to various resources. These include the UID and GID (group ID) of the process, and are inherited from the creating parent process. Generally, these credentials are set in the user shell process during login, and all subsequent processes created in the hierarchy from this shell process down will share common credentials via inheritance.

- One or more Threads
  - The execution elements of a process are commonly known as threads. Threads are the schedulable entities within a process, and only a thread can make computations progress. Threads always have a specific state at any given instant, and can move from one state to another over time. The state transition diagram below depicts the states of thread in a Linux system:



  - When a thread is in the Run state, it is executing on a CPU. From the Run state it may choose to finish (EXIT), wait for some event like a read from disk to complete (SLEEP), or move back to the Ready state, yielding the CPU to some more worthy thread (PREEMPTION).
    - If a thread leaves the Run state, then another thread must be moved to the Run state via the DISPATCH arc immediately.
    - There must be a thread in the Run state on each CPU at all times for the system to function properly, so during the system boot activity, the PID 0 kernel process creates one thread per CPU called an IDLE thread. An IDLE thread typically has the lowest priority of all threads in the system, so is only dispatched to a CPU when there is no other thread in the system ready to run.
  - If a thread is in the Block state, we assume that some event will eventually occur that will allow it to be moved to the Ready state.
    - The type of event that we anticipate is something like an interrupt to a CPU from a controller that has just completed some task (ie., a disk

controller that just completed transferring data from a disk device into memory).
        - When the interrupt is raised against a CPU, that processor will save the state of the thread it is running (remember, a CPU must be running some thread at all times), and the running thread will be forced to enter the kernel (if it's not already there) and execute the appropriate exception routine to handle the event that just occurred.
        - The handler code will, among other things, determine if any thread was in the Block state waiting for this event, and if so, will transition that thread from the Block state back to the Ready state, following the WAKEUP arc.
        - When the thread has finished running the specific exception routine, it will go back to whatever it was doing before the event occurred. If it was running in user mode before the event it will return to user mode after the event is handled, and if it happened to be running in kernel mode when the event occurred, it will return to whatever it was doing in kernel mode before the event occurred.
    - You should see from this description, that the operating system code is run by arbitrary threads in one of two ways:
        - A thread specifically makes a system call, in which case the thread voluntarily moves from running in user mode to running in kernel more to fulfill the system call. This is known as in-context kernel entry.
        - A thread is snatched into the kernel to run an exception handler because that thread happened to be running on the CPU when the exception was presented to that CPU. This is known as out-of-context kernel entry.
    - You should also understand that since each CPU has an interval hardware timer that posts an interrupt to the CPU after some few milliseconds, no thread remains running in user mode for long, before it finds itself in the kernel running the exception handler for the interval timer. These interval clock interrupts provide a mechanism to ensure that no thread can monopolize a system CPU for any length of time.
    - A thread can only EXIT the system from the Run state, which assures the system that any thread that wants to finish what it's doing will have to run its way out of the system, cleaning up its resources as it leaves. In other words, threads must commit suicide, they cannot be killed by some other thread from a distance.
        - If we want to get a thread out of the system, we must first get it to the Run state, and then convince it to commit suicide.
        - This is accomplished by sending a software exception to a target process in the form of a SIGNAL (Linux has 32 different types of signals). The signal forces some thread in the target process to not only kill itself, but to kill the entire process and all of its threads.
        - When the last thread in a process moves through the exit code, it brings the entire process down with it. A process cannot exist without at least one living thread.
- Other process attributes include:
    - A working directory (shared by all threads).
    - A table of open channels (shared by all threads), connecting the process to files, and various inter-process communication features (IPC mechanisms like pipes, sockets, shared memory, etc.).
    - A table describing SIGNAL behavior (shared by all threads), with an entry for each of the 32 signals that can be used.

- A default set of scheduling parameters used by threads created within this process; unless they are overridden during the thread create activity.
    - A process defines a default scheduling policy, priority, CPU affinity, etc., used as thread attributes during the creation of threads.

Threads:

As we've seen above, threads comprise the executable elements of a process. A process must have at least one thread to exist, but a single process may have thousands of threads if needed. If an application logically has multiple concurrent activities that it must support, it may be deployed as a collection of singly threaded processes, or as a single multiply threaded process.

In either model, we will have threads running on CPUs to deploy the application, but the cost of switching a CPU from one thread to another is very dependent upon the threads involved in the context switching activity. The cost of a context switch (in terms of execution time and effort) depends on the address space manipulation that accompanies the context switch.

- When a thread from some process PID x leaves the Run state (yields the CPU to another thread), and the inbound thread (the thread being dispatched onto the CPU) lives in another process such as PID y, we say that we have a heavy weight context switch. Because each process has its own address space, when we switch between threads from different processes we have to discard the cached address space components of the outbound thread's process, and re-establish a new cached address space on the CPU for the inbound thread's process address space. This activity has a very negative effect on system performance.
- If we switch between threads that live in the same process, however, we can retain the cached elements of the process address space, and avoid the performance impact of a heavy weight context switch. This type of switching is known as a light weight context switch, and supports much greater system performance.

As the available hardware becomes more concurrent by way of multiple cores, embedded memory controllers and very high speed peripheral interconnects, applications built around multithreaded technology are increasingly in demand. Highly concurrent applications, however, bring with them the need to manage complex synchronization requirements. This need for synchronization has always been a core issue for operating systems developers, but it is now becoming a growing concern for application developers working in the world of multithreaded software deployment. We will take a look at these synchronization issues next week, as we continue to work our way through chapter 2.

Week #4 Summary:

Operating systems are organized around the process construct. A process is the unit of management for most contemporary operating systems, and is viewed as a resource container for some computation.

Processes have many attributes, but key among them is a private address space, a collection of credentials, a executable program, and one or more threads to execute that program on a CPU. During a system boot, the monolithic boot code brings the system up to a point where the first process can be created, and the operating system is established. All subsequent processes

which may come into existence during the life of the system are created by a thread from some exiting process.

We consider a running system as a collection of processes, each of which has one or more threads that may, from time to time, be dispatched onto a CPU. A critical requirement of such a system is that each CPU in the system must have a thread running on it at all times. During system boot, the initial kernel process addresses this issue by creating an IDLE thread for each of the CPUs in the system. IDLE threads have a very low priority, and so will only be dispatched to a CPU if there are no other threads in the system that want to run.
The need for a thread per CPU at all times is a critical part of how an operating system runs its code and manipulates its data. The operating system is just a passive collection of code and data that must be shared by the threads on a platform, and it's these individual threads that actually run the operating system code when it needs to be run. A thread that needs system services (like access to network packets that are arriving over the internet), can voluntarily place itself into the operating system address space and run the OS code necessary to drive the support required by making system calls. The code in the operating system that responds to system calls is referred to as base level code. There are other code routines in the kernel that must be run asynchronously when certain events occur in the system, however, and this code is executed by whatever thread is running on a given CPU at the time that the CPU in question receives an interrupt or some other form of exception. Since we can never know when an exception will posted against a CPU, it is critical for each CPU to be running some thread that can be immediately snatched into the kernel and coerced to run the exception code (thus the absolute need for the IDLE threads).

Because user threads are frequently executing in the kernel (as a result of a system call or exception), the system has to be carefully designed and deployed to avoid race conditions and data collisions among these potentially many concurrent threads. Using our school building analogy again, you're safe from collision with anyone if you're the only one running around your classroom (private address space), as is the case for a singly threaded process. If you leave your room to move into the corridor, however (as the result of making a system call or fielding an exception on your CPU), you must be more careful about your movements, because there are other active entities (threads) out there (in the kernel's address space) that you could run into. This need for thread synchronization will be our topic next week.