Industrial Workstation Overview (Part 5: Processes)

\*Overview of processes

--Consists of an address space and the set of data structures within the Kernel

--The address space is a set of memory pages that the kernel has marked for the process’s use.

--The kernel has an internal data structure record that stores information about each process

--The process’s address space map

--The current status of the process (sleeping, stopping, runnable…)

--The execution priority of the process

--Information about the resources the process has used

--Information about the files and network ports the process has opened

--The process’s signal mask (a record of which signals are blocked)

--The owner of the process

--An execution thread, usually known as simply a thread is the result of a fork in execution within a process. A thread inherits many of the attributes of the process that contains it.

--Concurrent execution is simulated by the kernel on old style uniprocessor systems but on multicore and multi-CPU architectures, the threads can run simultaneously on different cores.

--PID: process ID number is a unique ID number. ID namespaces added recently kind of change this with concurrency

--PPID: parent PIDs: To create a new process, the PPID of the new process must first clone itself (fork) and then overwrite it with an exec call

\*UID and EUID: real and effective user ID

--A process’s UID is the user identification number of the person who created it or it is a copy of the UID value of the parent host.

--The EUID is the “effective” user ID, an extra UID used to determine what resources and files a process has permission to access at any given moment. UID and EUID as usually the same except for the case when a program uses a setuid

--Why have bot a UID and EUID? It is useful to maintain a distinction between identity and permission and because a setuid program may not wish to operate with expanded permissions all the time.

--Most systems also keep track of a “saved UID” which is a copy of the process’s EUID at the point at which the process first begins to execute. Unless the process takes steps to destroy this saved UID, it remains available for use as the real or effective UID. A conservatively written setuid program can therefore renounce its special privileges for the majority of its execution accessing them only when needed.

--GID and EGID are the same structure as UID and EUID

\*Niceness

--A process’s scheduling priority determines how much CPU time it receives. The kernel uses a dynamic algorithm to compute priorities, allowing for the amount of CPU time that a process has recently consumed and the length of time it has been waiting to run.

------------Part 5.2 the Life Cycle of a Process-----------

\*Process’s Life Cycle

--Fork is used to create a copy of the original process which is largely identical to the parent. The new process has a distinct PID and its own accounting information. Fork returns two different values. Zero from the child perspective and the PID of the newly created child from the parents perspective.

--When a process completes, it calls a routine named \_exit to notify the kernel that it is ready to die. Before a process can be allowed to disappear completely, the kernel requires that its death be acknowledged by the process’s parent, which the parent does with a call to wait.

--If the child outlives the parent process, init takes the process and orphans it where they can then die.

\*Signals

--Signals are process-level interrupt requests. About thirty different kinds are defined and are used in the following ways.

--Sent among processes as a means of communication

--Sent by the terminal driver to kill, interrupt, or suspend processes: Key board such as ctr c or ctr z

--Sent by the administrator to achieve solutions

--Sent by kernel when a process commits an infraction such as division by zero

--Sent by the kernel to notify a process of an “interesting” condition such as the death of a child or the availability of data on an I/O channel

--One of two things happen when a signal is received

--the handler is called with information about the context in which the signal was delivered.

--Default action is taken on behalf of the process.

--Specifying a handler routine for a signal within a program is referred to as catching the signal. When the handler completes, execution restarts from the point at which the signal was received.

--To prevent signals from arriving, programs can request that they be either ignored or blocked. A signal that is ignored is simply discarded and has no effect on the process. A blocked signal is queued for delivery, but the kernel doesn’t require the process to act on it until the signal has been explicitly unblocked.

--List of common signals is on page 125 table 5.1

--Signals KILL and STOP cannot be caught, blocked, or ignored. The KILL signal destroys the receiving process, and STOP suspends it execution until a CONT signal is received. CONT may be caught or ignored but not blocked.

--TSTP is a “soft” version of STOP that might be best described as a request to stop.

--Terminal emulators send a WINCH signal when their configuration parameters change.

--Summary of important signals that sound the same

--KILL is unblockable and terminates a process at the kernel level. A process can never actually receive this signal

--INT is sent by the terminal driver when you type ctr c. It is a request to terminate the current operation.

--TERM is a request to terminate execution completely. It is expected that the receiving process will clean up its state and exit.

--HUP has two common ideas

--It is thought of as a reset request by daemons.

--If a daemon is capable of rereading its configuration file and adjusting to changes without restarting, a HUP can generally be used for this.

-----------Part 5.4 KILL: Send Signals-------------

\*Kill Command

--Kill is most commonly used to terminate a process. Kill can send any signal, but by default it sends a TERM. Kill can be used by normal users on their own processes or by root on any process. Syntax = kill [-signal] pid where signal is the number or symbolic name

--A kill without a signal number does not guarantee the process will die, because the TERM signal can be caught, blocked, or ignored.

--The command kill -9 pid guarantees that the process will die because signal 9, KILL cannot be caught, blocked, or ignored. Use only if a “polite” kill will not work

--The killall command performs wildly different functions on different systems. Linux killall kills process by name EX. Sudo killall httpd. On UNIX, killall kills init and shuts down the machine.

--pgrep and pkill commands search for process by name and display or signal them respectively.

---------Part 5.5 Process States---------

\*Process States

--Execution States

--Runnable = The process can be executed

--Sleeping = The process is waiting for some resource

--Zombie= The process is trying to die

--Stopped= The process is suspended and not allowed to execute

----------Part 5.6 Nice and Renice Scheduling Priority-------

\*Nice and Renice

--The “niceness” of a process is a numeric hint to the kernel about how the process should be treated in relation to other processes contending for the CPU. A high nice value means a low priority for CPU scheduling. Many different ranges exist with the common being -20 to +19.

--A newly created process inherits its nice value from its parents. The owner of a process can increase its nice value but not decrease it. Root can set any niceness

--I/O performance has not kept up with CPU speed and is the major bottle neck on most systems. High nice processes can still bog down memory or I/O

--The nice command is used at process creation and the renice command is used later with a value given as an argument.

--Nice is not built into bash so the entire path name should be provided.

--The most common niced program is that of ntpd as it is the clock synchronization and is usually used in critical mission.

------------Part 5.7 Monitor Processes---------

\*The ps Command

--The ps command is the system admins main tool for monitoring processes. It shows the PID, UID, and control terminal of processes. It also gives information about how much memory a process is using, how much CPU time it has consumed, and its current status.

--A good command to run is ps aux as ‘a’ gives the option for all processes, x means to show even processes that don’t have a control terminal, and u selected the user oriented output

\*Dynamic Monitoring with top, prstat, and topas

\*The /proc filesystem

--top and ps read their process status information from the /proc directory, a pseudo-filesystem in which the kernel exposes a variety of information about he systems state.

\*strace is used to see every system call a process is making

--With other attachments, it is also to view the information of forked processes under it.