OS & Syscall Basics 4: Processes, IPC, Signals

1. Operating Systems
2. System calls
3. Filesystems and Memory
4. **Processes, IPC, Signals**
   - **processes**
   - **process attribute syscalls**
   - **process management**
   - **multi-process programs**
   - **process synchronization syscalls**
   - **threads**
   - **interprocess communication**
   - **signals**
   - **signal syscalls**

Processes

One of the key concepts in Linux/UNIX OS's is that of a **process**: a running program.

Processes are identified by a positive integer ID, known as the **process identifier** (or **process ID** or just **PID**).

Each process has a number of **attributes**, including:

- **parent PID** (PPID)
- **current working directory** (CWD)
- **controlling terminal**
- **real user ID** (RUID) and **group ID** (RGID)
- **effective user ID** (EUID) and **group ID** (EGID)
- scheduling priority ("nice" value)
- **file descriptor table** (open file links and info)

Process Attribute Syscalls

There are syscalls to get/set process many process attributes:

chdir – change **current working directory** (CWD)

getcwd – get CWD

getpid, getppid – get PID, PPID

setreuid, setregid – set real and/or effective UID/GID

getresuid, getresgid – get real, effective, and saved UID/GID

getrlimit, setrlimit – get, set resource limits

umask – change file creation **mask**

nice – change **scheduling priority**

OS Process Management

All processes except for the first process must be created by another process (the "**parent process**").

Linux maintains information about the current set of processes in its **process table**.

The process table is kept in memory (RAM) at all times.

It is a *doubly-linked list* of **task_struct**'s (i.e., task structures).

Each task structure contains the information that might be needed while a process is not running and might be needed to prepare the process to run on a CPU.

Which process will run next on a CPU/core and for how long is determined by the kernel **scheduler**.

Process Management Syscalls

There are syscalls for creating and terminating processes:

_exit – terminate process immediately

fork – create a subprocess (initially a "copy" of parent)

vfork – variant of `fork`

clone – Linux-specific variant of `fork`

execve – execute new program in process

execv,...,execl,... – `execve` wrapper variants

waitpid – wait for subprocess to terminate and collect results

wait – simplified version of `waitpid`

waitid,wait4 – variants of `waitpid`

Multi-Process Programs

A **multi-process program** is a program that uses *multiple processes* to carry out the program's functionality.

Multi-process programs can be created using just `fork`, `execve`/etc., `waitpid`/`wait`, and `_exit`/`exit`.

However, one of the challenges of writing multi-process programs is that it is the *kernel scheduler* that decides when each process will be executed and for how long.

This can affect the *order* that statements in the different processes end up being executed.

Various **synchronization mechanisms** may be required to ensure a particular execution order for certain statements.

Process Synchronization Syscalls

There syscalls for using several different process syncrhonization mechanisms:

waitpid,wait – suspend process until subprocess terminates

pause – suspend process until receive signal (see below)

sigsuspend – suspend process until receive signal (see below)

sem_open,sem_wait,sem_post – POSIX semaphores

semget,semctl – System V semaphores

pipe – create pipe (support sychronization via I/O operations)

Threads

Processes are (mostly) separate from one another: separate address spaces, FD tables, signal masks, etc.

Threads are OS constructs that allow multiple *"threads of execution"* within a single process' address space.

Unlike processes, threads share the same variables and code (but have separate stacks and program counters).

Linux supports the **POSIX Threads** (Pthreads) thread model.

Pthreads are kernel-level entities, so thread scheduling is done by the kernel process scheduler just as with processes.

Thread Syscalls

There are a large number of thread-related *wrapper* "system calls" such as: `pthread_create()`, `pthread_mutex_init()`, etc.

However, these calls that do not map directly to kernel syscalls.

There are only a few *kernel syscalls*, such as:

clone – create a new process/thread

futex – fast locking (mutex)

Interprocess Communication

Interprocess communication (IPC) refers to mechanisms for one process to send information to another (without using files are intermediates).

A number of IPC mechanisms are supported, including:

pipes – unidirectional, related processes only

FIFOs – unidirectional, filesystem entry

sockets – intra and inter-machine, bidirectional

message queues – message-oriented rather than stream-oriented

shared memory – memory pages shared among processes

Interprocess Communication Syscalls

Some syscalls for IPC:

pipe – create a pipe

socket – create a socket

bind – associate an address with a socket

connect – connect to a socket

socketcall – kernel syscall for all socket-related calls

mq_send,mq_receive – POSIX message queues

msgsend,msgget – System V IPC message queues

ipc – kernel syscal for System V IPC

mmap – map file or device into memory

Signals

Signals are a mechanism for informing processes that some event has occurred.

Because they are delivered **asynchronously**, they are effectively a type of **software interrupt**.

Signals can be used for very simple *IPC*.

They are also used for *process synchronization*.

Signal Syscalls

There are a number of signal-related syscalls, including:

kill – send signal to a process

pause – suspend process until receive signal

alarm – set an alarm clock for delivery of a signal

signal – implement ANSI C signal handling

sigaction – POSIX signal handling

sigsuspend – suspend process until receive signal

sigpending – examine pending signals

sigreturn – return from signal handler and cleanup stack frame

sigprocmask – change blocked signals