OS & Syscall Basics 1: Operating Systems

1. **Operating Systems**
   - **os overview**
   - **layered design**
   - **kernel vs. user mode**
2. System Calls
3. Filesystems and Memory
4. Processes, IPC, Signals

©Norman Carver

Operating Systems

Requiring or even allowing users and application software direct access to a computer's hardware raises many problems:

- modern computer hardware typically requires long, complex sequences of instructions to accomplish the kinds of tasks computer users and applications want accomplished (e.g., storing a file on a harddrive)
- different types of hardware can require different instructions to accomplish basically the same task (e.g., reading a file from a harddrive vs. from a CDROM)
- direct access to the hardware would lead to reliability and security issues
- multiuser systems could have issues with resource contention among users

©Norman Carver

Operating Systems (contd.)

These issues make it impractical and inappropriate for users and application software to interact directly with computer hardware.

All but the most primitive modern computer systems have a key *layer of software* that sits between user application software and the hardware that makes up the computer system.

This software is called the **operating system** (OS).

One key function of an OS is providing an *abstract computer* that is much easier to use than the actual (hardware) computer.

This abstract computer will also provide a *uniform interface* across different hardware (e.g., Linux on PC vs. on mainframe).

©Norman Carver

Operating Systems (contd.)

So the first major function of an OS is providing a simplified and uniform interface to a computer's hardware.

The *second major function* of an OS is to *manage a computer's hardware resources* so programs can run *securely and efficiently*.

This function should be invisible to users if the OS is doing a good job.

When we talk about the hardware resources of a computer we are talking about: CPU(s), memory, hard drive(s), CDROM drives, network cards, printers, etc.

©Norman Carver

Operating Systems (contd.)

An OS provides a large number of services to users/programs:

- filesystem(s) for storing data
- security policies to limit access to the computer and its files
- allocating and sharing memory to allow multiple programs to run simultaneously
- scheduling CPU usage by programs to allow fair and efficient execution of multiple programs
- preventing programs from reading/writing in each others' memory
- controling access to devices like hard drives, and scheduling access to be efficient
- implementing network protocols for network communication

Operating Systems (contd.)

The core functionality of an OS is referred to as the **kernel**.

Modern OSs are large, complex pieces of software, on the order of 5 million lines of code!

On so called **monolithic kernel** OSs like Linux, the kernel is basically the entire OS.

On **microkernel** OSs, the kernel will not be the entire OS; instead some parts of the OS will not run in kernel mode (see below).

"Linux" is an OS kernel along with a large set of **device drivers**.

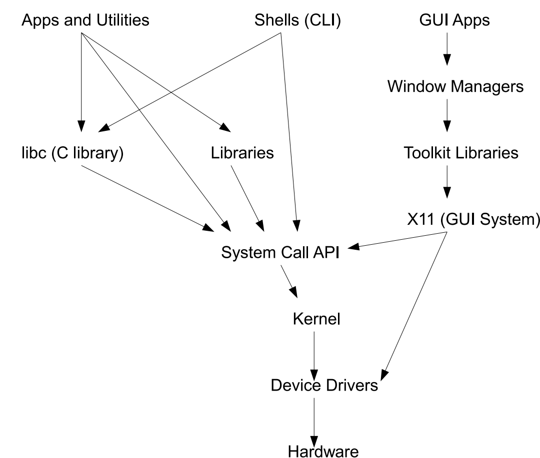
Layered Design of OS's

What most people think of as the "operating system" is actually the OS plus more:

- **device drivers**: interface between kernel and hardware
- a **kernel**: the true/basic OS
- **libraries**: higher level programming interfaces
- **shells**: **command line** interface to OS
- **GUIs/window managers**: graphical interface to OS
- **application software**: allows users to get things done

These components are arranged in *layers* between the hardware and users of the computer system.

Layered Design of OS's (contd.)



Linux Distributions

A Linux **distribution** is a packaging of:

- Linux kernel(s) + device drivers
- shells (bash, csh, etc.)
- GUI (X11, Xfree, X.org)
- window managers (KDE, Gnome, etc.)
- boot managers (LILO, GRUB)
- application software
- installation and maintenance tools

Because much of this additional software comes from the GNU project, some people refer to Linux as "GNU/Linux."

Kernel Mode vs. User Mode

Virtually all modern processors have multiple **modes**.

The instructions that can be carried out by a process running in a certain mode may be limited.

At the very least, processors will have two modes:

- **kernel mode** (also called supervisor, privileged, or master): the *unrestricted mode*, where any processor instruction is allowed to be executed
- **user mode**: a *restricted mode*, where instructions to access I/O devices may not be allowed, access to certain areas of memory may not be allowed, and so forth

Some CPUs support more than two modes, resulting in what are known as **protection rings** or **privilege rings**.

Kernel Mode vs. User Mode (contd.)

An OS *kernel* needs to have unlimited access to devices and memory, so it will need to run in *kernel mode*.

Applications cannot be allowed unlimited access, so they will need to run in a user mode.

The term **kernel space** refers to code requiring the processor be in kernel mode, while **user space** refers to code that can be run in a user mode.

A consequence of the use of modes is that every time the OS must do something for an application, the processor mode must be *switched*, and this involves some *time cost*.

OS & Syscall Basics 2: System Calls

1. Operating Systems
2. **System Calls**
   - **system calls API**
   - **syscall wrapper functions**
   - **documentation**
3. Filesystems and Memory
4. Processes, IPC, Signals

©Norman Carver

System Calls API

Since the operating system controls access to all of a computer's resources, users and application programs must make requests to the OS to use of any of these resources.

The *interface* (API) between programs and the OS consists of a set of functions known as **system calls**.

Each system call can also be considered as an **entry point** into the functionality of the OS kernel.

The system calls of an OS define the set of services that the OS can provide to programs.

Linux has slightly over 300 defined system calls.

©Norman Carver

System Calls API (contd.)

OS system calls typically provide the following services:

- **process/thread** creation and control
- memory allocation/deallocation
- file creation, reading, writing, etc
- directory creation and manipulation
- **interprocess communication**, including network communication
- process synchronization and file locking
- **signal** handling (software interrupts)
- device I/O

©Norman Carver

Syscall Wrapper Functions

Exactly how a system call is invoked will depend on the processor architecture, as it involves switching from user to kernel mode.

Often it requires that the **system call code** (number) be placed in a register, argument addresses be set on the (user) stack, and a TRAP or similar **interrupt** type instruction be executed, after which the kernel code **dispatches** to the correct code based on the syscall code.

Luckily, the ugly elements of this process are hidden from user programs because an interface to the syscalls using **C wrapper functions** is part of the C standard library (libc).

In fact, this is how the calls are defined in the POSIX/SUS standards (i.e., as C functions).

©Norman Carver

Syscall Wrapper Functions (contd.)

For example, the `open` system call *wrapper function* is defined as:

```
int open(const char *pathname, int flags)
int open(const char *pathname, int flags, mode_t mode)
```

A C program will call one of these functions, and the *library code* that implements the function will do the required setup and use the correct architecture-specific method to initiate the appropriate kernel system call.

The Linux kernel `open` system call is syscall 5, named `sys_open` in the kernel code.

Syscall Wrapper Functions (contd.)

It is defined in the file `fs/open.c`:

```
asmlinkage long sys_open(const char __user *filename,
                         int flags, int mode)
{
        long ret;

        if (force_o_largefile())
                flags |= O_LARGEFILE;

        ret = do_sys_open(AT_FDCWD, filename, flags, mode);
        /* avoid REGPARM breakage on x86: */
        prevent_tail_call(ret);
        return ret;
}
```


Syscall Wrapper Functions (contd.)

The wrapper "system calls" in libc do not always have a one-to-one mapping to kernel syscalls.

There may be no equivalent kernel syscall, or multiple wrapper syscalls may invoke the same kernel syscall.

For example, there is no kernel syscall that directly implements the SUS calls `opendir()` and `seekdir()`.

Since *directories* are simply *specially formatted files*, most of the directory syscall functions can be implemented using file syscalls like `open` and `lseek`.

On the other hand, there is a kernel syscall for wrapper `readdir()`.

Syscall Wrapper Functions (contd.)

Another example is the wrapper syscalls `wait()` and `waitpid()`, which both invoke the Linux kernel syscall `sys_waitpid`.

While `wait()` and `waitpid()` are both required by SUS, Linux is free to implement them using kernel syscalls in any way it chooses, and `wait()` is basically a simplified version of `waitpid()`.

Perhaps the most extreme example of there not being direct mappings from wrapper calls to kernel syscalls is the kernel syscall `socketcall`, which implements all of the socket-related wrapper calls: `socket()`, `bind()`, `accept()`, `connect()`, etc.

Documentation

Further documentation on Linx system calls can be found by:

- `man 2 syscalls`
- `man 2 syscall` (call to invoke a system call by number)
- `more /usr/src/linux/arch/x86/kernel/syscall_table_32.S` (or similar)
- `nm -D /lib/libc-2.7.so` (or similar)

©Norman Carver

OS & Syscall Basics 3: Filesystems and Memory

1. Operating Systems
2. System calls
3. **Filesystems and Memory**
   - **filesystems**
   - **filesystem syscalls**
   - **file locking**
   - **memory management**
   - **program address space**
   - **memory management syscalls**
4. Processes, IPC, Signals

Filesystems

The method for storing files on harddrives and other devices are supplied by the OS, and are one of the most obvious aspects of the OS for users.

The format used to store files is referred to as a **filesystem**.

Linux supports a large set of filesystems, including "native" filesystems, various UNIX filesystems, and Windows filesystems.

The current standard native Linux filesystems are **ext3** and **ext4**.

Both are **journaling filesystems**, meaning they write a **journal** of disk changes so can quickly recover from power failures, etc.

Linux can also access Windows **FAT** and **NTFS** filesystems.

Filesystem Syscalls

There are a fairly large number of filesystem-related syscalls; here are some of the more important ones, grouped by basic purpose:

Syscalls for using files in programs:

open – open a file (return **file descriptor**)
close – close file descriptor (and so file)
read – read from file descriptor
write – write to file descriptor
lseek – reposition read/write offset for file descriptor
pread – read from file descriptor at a given offset
pwrite – write to file descriptor at a given offset
ftruncate – change file descriptor (file) length (size)

Filesystem Syscalls (contd.)

Syscalls for file and filesystem management:

creat – create a file
truncate – change file length (size)
link – create a new link/name for a file
symlink – create a symlink for a file
unlink – remove a link/name and possibly delete file
rename – rename a file, possibly moving it
chmod – change file permissions
lchown – change file ownership
stat – retrieve file/inode information
utime – set file/inode mtime, atime
mount, umount – mount, unmount a filesystem

Filesystem Syscalls (contd.)

Syscalls for directory management:

mkdir – create a new directory

rmdir – delete a (empty) directory

readdir – read one directory entry

In addition to the above kernel syscalls, there are a number of directory-related library wrapper "system calls" including:

opendir – open a directory (for use with `readdir`)

closedir – close a directory

seekdir – set position in directory

File Locking

File locking refers to preventing multiple processes from accessing and/or modifying a file at the same time.

Some system calls for synchronization and locking:

flock – apply/remove an advisory lock on open file

lockf – apply/remove an advisory lock on open file

fcntl – open file locking (plus more)

futex – fast mutex (locking)

(There are various issues with the file locking syscalls that can make their use unreliable.)

OS Memory Management

Another key function of an OS is **memory management**.

Some key elements of OS memory management are:

- allowing multiple programs to run "concurrently"
- allowing programs with larger address spaces than the amount of RAM to run
- preventing programs from interfering with each other
- supporting the use of **shared libraries** (DLLs)

OS Memory Management (contd.)

As with most moden OSs, Linux supports a **virtual memory** model and **paging** mechanism:

- each process has its own **address space**, which can be as large as is allowed by the architecture's address word size (e.g., 32 bits allows 0 through $2^{32} - 1$ or 4GB)
- object code addresses are **virtual addresses** that must be mapped to physical memory addresses at runtime
- the address space is broken up into equal size contiguous chunks called **pages**
- individual pages can be mapped into physical memory **page frames** as needed and in any order
- pages may also be **swapped/paged out** to a harddrive

OS Memory Management (contd.)

- the entire address space of a process need not be in physical memory at once to execute
- if a virtual address whose page is not in memory is accessed, a **page fault** is generated, causing the OS to **swap/page in** the needed page
- the process' **page table** keeps track of whether each page is in memory, its page frame or disk location, and other info
- a page may be accessible to multiple processes (for **shared libraries** and **memory-mapped files**)
- **page replacement algorithms** determine what pages should be in memory and which swapped out to avoid page faults as much as possible.

©Norman Carver

Process Address Spaces

The address space of any Linux process consists of several different **segments**:

text – machine code/instructions, generally read-only

data – *initialized* global and `static` variables

bss (or **BSS**) – *uninitialized* global/`static` variables (automatically zeroed/`NULL`'d at startup)

heap – dynamically allocated memory

memory mapping – shared libraries and shared memory IPC

thread stacks – stacks for addition Pthreads of process

stack – the program/execution stack (userspace)

These segments are placed in the address space in the order given: text segment at lowest addresses and stack segment at highest addresses.

©Norman Carver

Process Address Spaces (contd.)

The stack *bottom* is at the *top* of the (user's) virtual address range and *grows down* in address as stack frames are pushed on.

The heap sits at the *top* of the data segment and *grows up* in address if it must be expanded (e.g., by C runtime).

The top of the heap (end of program plus variables/data) is called the **program break**.

The virtual address range between the break and the stack top is used for **dynamic libraries**, **shared memory**, and **thread stacks**.

The text, data, and bss segments come directly from the contents of the **executable file** being run.

©Norman Carver

Memory Management Syscalls

System calls for allocating program memory were considered too architecture specific to be standardized in POSIX/SUS.

When writing portable code, the standard C language memory management functions are to be used: `malloc()`, `free()`, etc.

Of course, these functions may need to make kernel syscall(s) to *expand heap memory* (by expanding the data segment).

The Linux kernel syscall for this is named **brk** because the program break is adjusted.

Though not formally standardized, this syscall name is common among UNIXes (along with **sbrk**).

©Norman Carver

Memory Management Syscalls (contd.)

There are a few other memory management-related kernel syscalls, many dealing with mapping files into a process' address space:

mmap – map file or device into memory

munmap – unmap file or device into memory

msync – synchronize a file with a memory map

mremap – re-map a virtual memory address

remap_file_pages – create a non-linear file mapping

mprotect – set protection on address space pages

mlock, mlockall – lock part or all of process' virtual address space into RAM

munlock, munlockall – unlock process' virtual address space

©Norman Carver

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`

©Norman Carver

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.

©Norman Carver

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)

©Norman Carver

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.

©Norman Carver

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

©Norman Carver