DM510: Operating System Structure

Lars Rohwedder

Disclaimer

These slides contain (modified) content and media from the official Operating System Concepts slides: https://www.os-book.com/OS10/slide-dir/index.html

Today's lecture

- Chapter 2 of course book
- Introduction to first programming project

Operating System Services

Main services

The following services are provided to user programs or directly to the user via system programs and **user interface** (command-line, graphical user interface, touch-screen,...)

Program execution: load program into memory and run, end execution

I/O operations: to file or device

File-system manipulation: read, write, create, delete, search, etc. files and directories

Communication: between processes or between computers of a network, mostly via **shared memory** or **message passing**

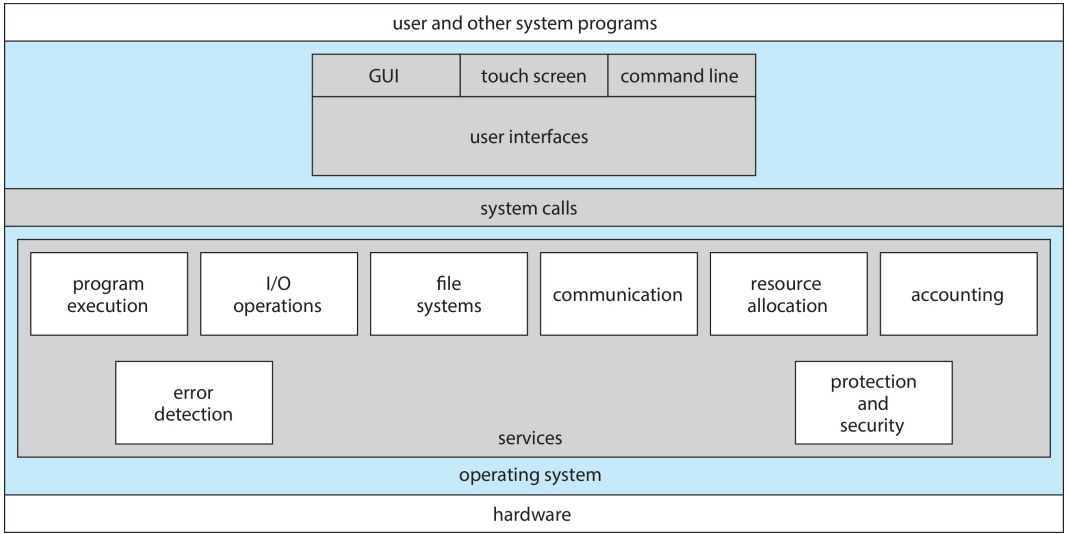
Error detection: appropriate action to recover from errors, debugging facilities

Resource allocation: of CPU, main memory, file storage, I/O devices, etc.

Logging: statistics of resource usage, errors, etc.

Protection and security: protect processes/users from others or system from outside

Placement of services within operating system



Services are accessed by user/system programs via **system calls**

System Calls

Issuing system calls

System call in Linux

```
int code = syscall(__NR_hellokernel, 42);
```

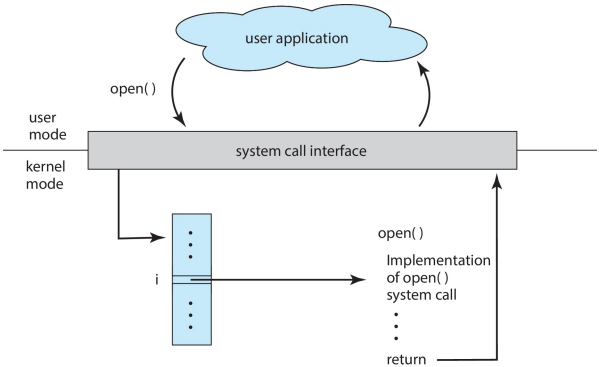
- Results in trap/software interrupt, i.e., kernel takes over

Issuing system calls

System call in Linux

```
int code = syscall(__NR_hellokernel, 42);
```

- Results in trap/software interrupt, i.e., kernel takes over
- Kernel then executes specific functions depending on type of system call, in this example `__NR_hellokernel`

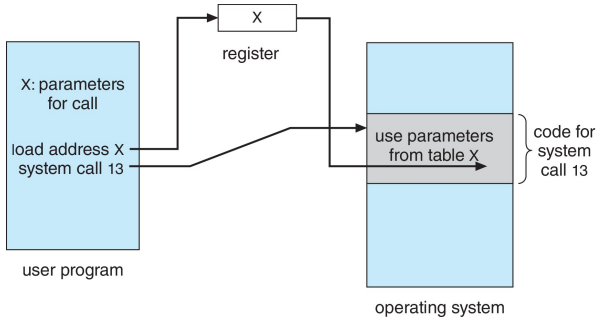


Issuing system calls

System call in Linux

```
int code = syscall(__NR_hellokernel, 42);
```

- Results in trap/software interrupt, i.e., kernel takes over
- Kernel then executes specific functions depending on type of system call, in this example `__NR_hellokernel`
- Parameters need to be passed with system call. In Linux we pass pointers or integers via registers



Passing pointer to table of large parameter data

System calls in practice

- Using system calls directly is inconvenient and error-prone: need to know calling conventions.
- Instead of issuing system calls directly, standard libraries/APIs are used, whose implementation makes the actual system calls
- informally, the API functions are also referred to as "system calls"

Read from file in C (Unix)

```
char buf[512];
ssize_t num = read(file, buf, 512);
```

APIs in Windows and Unix

	Windows	Unix
Process control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File management	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device management	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
	⋮	

System Programs

Role of system programs

Typical users access system services not via system calls or APIs, but via **system programs**. They can be simple wrappers of system calls, but also significantly more complex programs.

Important system programs and services

File management

Modify and navigate files and directories

- Functions: create, delete, copy, rename, list
- Most operating systems provide graphical file browser
- Also integrated in command lines (in Unix: cd, ls, mkdir, rm, ...)

Important system programs and services

File management

Modify and navigate files and directories

- Functions: create, delete, copy, rename, list
- Most operating systems provide graphical file browser
- Also integrated in command lines (in Unix: cd, ls, mkdir, rm, ...)

Status information

Sometimes stored in files, sometimes accessed via programs (graphical or command line)

- Device information: disk space, CPUs
- Date, time
- Performance, logs, debug information
- system configuration (**registry** in Windows; `/etc/` directory in Linux)

Important system programs and services (cont.)

File modification

- Text editor
- Search within file or filter contents
- Transformation (e.g. search and replace)

Important system programs and services (cont.)

File modification

- Text editor
- Search within file or filter contents
- Transformation (e.g. search and replace)

Background services

- Often active from system boot to shutdown
- Example: network daemon
- Known as **services, subsystems, daemons**

Important system programs and services (cont.)

File modification

- Text editor
- Search within file or filter contents
- Transformation (e.g. search and replace)

Programming language support

- Compiler
- Interpreter
- Debugger

Background services

- Often active from system boot to shutdown
- Example: network daemon
- Known as **services, subsystems, daemons**

Important system programs and services (cont.)

File modification

- Text editor
- Search within file or filter contents
- Transformation (e.g. search and replace)

Programming language support

- Compiler
- Interpreter
- Debugger

Background services

- Often active from system boot to shutdown
- Example: network daemon
- Known as **services, subsystems, daemons**

Communication

Virtual connections between

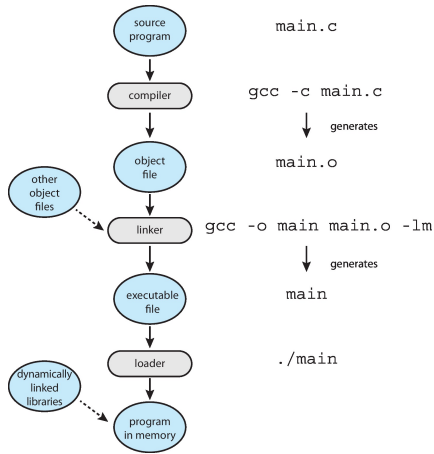
- processes
- users
- computer systems

Details on specific system services

Executing programs

Dynamic linking and loading

- Copy executable (instructions and data) to main memory
- Copy **dynamically linked libraries** into memory if necessary: program code shared by different programs (Windows: .dll, Linux: .so)
- Link final library addresses (e.g. function pointers) into executable's code
- Set program counter register to first instruction of executable's code



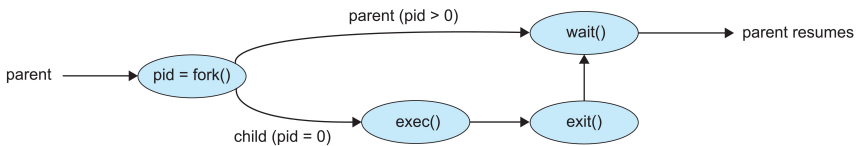
Executing programs

Dynamic linking and loading

Example: command line

How does a command line in Linux execute a program, e.g. `./helloworld`?

- Need to create new process using `fork()` system call
- The new process then calls linker and loader using `exec()`



Executing programs

Dynamic linking and loading

Example: command line

Binary (in-)compatibility

Usually executables only compatible with target operating system due to: file formats, calling conventions, system calls

- Some languages (Java, Python,...) only need compatible VM/intepreter
- Some "similar" operating systems (or versions) carefully maintain compatibility
- **Application Binary Interface (ABI)** is architecture equivalent of API, defines how different components of binary code can interface for a given operating system on a given architecture, CPU, etc.

Debugging

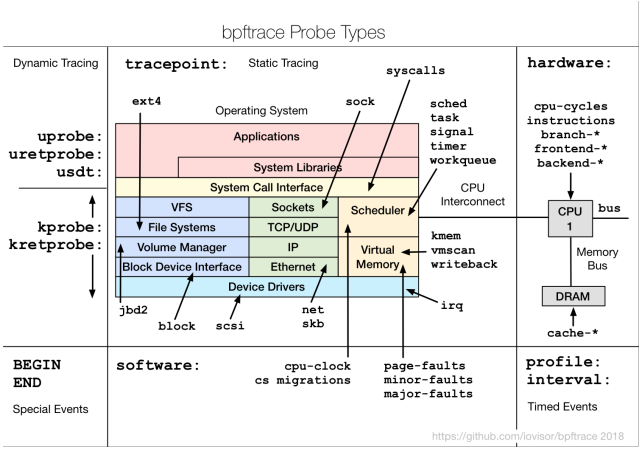
Kernighan's Law

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

- Finding and fixing **bugs**: kernel generates log files for errors and **core dump** for memory capture of crashed user program (similarly **crash dump** for crashed system/kernel)
- **performance tuning**: periodically sample instruction pointer for statistics (**profiling**), record data at specific events (**tracing**)

Bpftrace

bpftrace is a tool that makes it possible to execute user-specified scripts at various kernel events (via **eBPF**) while ensuring system's stability and performance.



Demo

See lecture/course website

Operating System Implementation

Development

- In low-level language C or C++, earlier assembly

Development

- In low-level language C or C++, earlier assembly

Linus Torwalds on C++ (2007)

> When I first looked at Git source code two things struck me as odd:
> 1. Pure C as opposed to C++. No idea why. Please don't talk
> about portability, it's BS.

YOU are full of bullshit.

C++ is a horrible language. It's made more horrible by the fact that a lot of substandard programmers use it, to the point where it's much much easier to generate total and utter crap with it. Quite frankly, even if the choice of C were to do *nothing* but keep the C++ programmers out, that in itself would be a huge reason to use C.

<rant continues...>

Development

- In low-level language C or C++, earlier assembly
- Some components (especially system programs) may be in higher programming languages

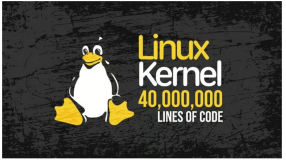
Development

- In low-level language C or C++, earlier assembly
- Some components (especially system programs) may be in higher programming languages
- Developing an operating system is a highly challenging task for **Software Engineering** due to the requirements on security, performance, and the huge size of the project

Linux Kernel Surpasses 40 Million Lines

The Linux kernel has rapidly grown, reaching an impressive milestone, surpassing 40 million lines of code.

By Bobby Borisov - On January 31, 2025



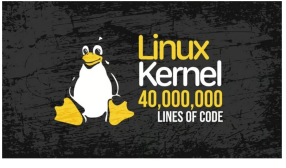
Development

- In low-level language C or C++, earlier assembly
- Some components (especially system programs) may be in higher programming languages
- Developing an operating system is a highly challenging task for **Software Engineering** due to the requirements on security, performance, and the huge size of the project
- Careful architectural decisions are necessary. Common to all modern operating systems is a **modular** structure, where different modules encapsulate different functionality. Sometimes modules can be even loaded dynamically while the system is up, e.g. device drivers in Linux

Linux Kernel Surpasses 40 Million Lines

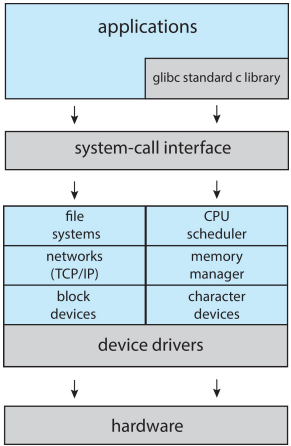
The Linux kernel has rapidly grown, reaching an impressive milestone, surpassing 40 million lines of code.

By Bobby Borisov - On January 31, 2025



Monolithic architecture

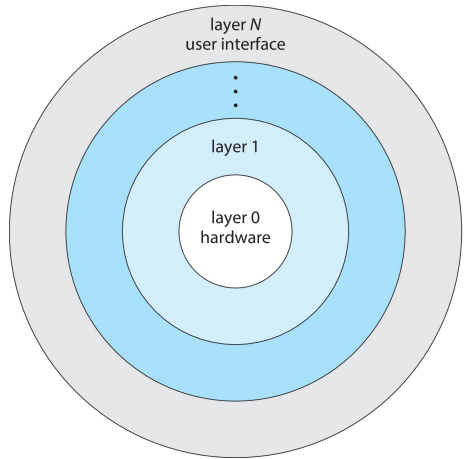
- Linux and Windows are mostly **monolithic** architectures: extremely large codebase, many different parts depending on each other
- Leads to more challenging debugging and unit tests
- Generally better performance (lower overhead) compared to other architectures



Linux

Layered architecture

- Each layer adds more abstraction
- Each layer uses only the functions and services of lower layers
- Layers can be tested separately to follow specification: easier testing and debugging
- Disadvantages: specifying layers and arrangement is challenging, high function call overhead
- Appears to some extend in many operating systems, but usually only to a very limited amount

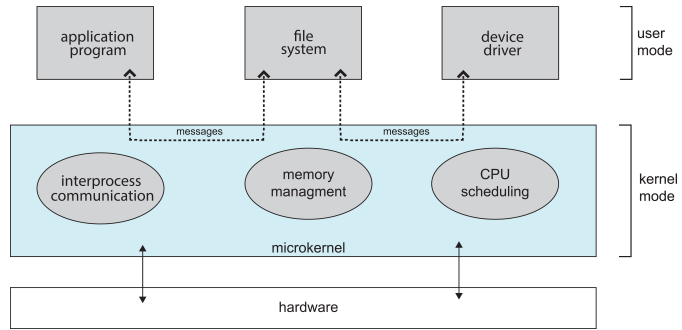


Microkernel architecture

- kernel very small; much of the operating system's functionality moved to user mode
- User modules communicate via message passing
- Example: **Mach kernel**, used e.g. in MacOS

Pro/Con

+ More extendable, portable, secure, easier to debug
- Overhead due to message passing and switches between user and kernel mode



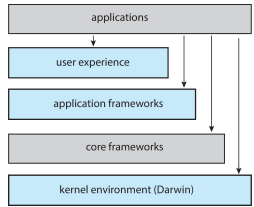
More examples

Android

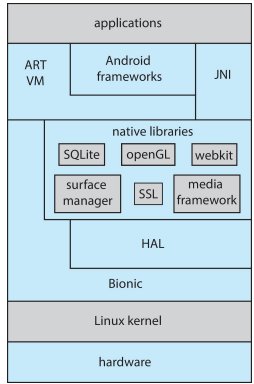
- Uses Linux kernel, very open
- Java for apps (with non-standard API). For energy reasons: ahead-of-time (AOT) instead of just-in-time (JIT) compilation

iOS

- Based on MacOS
- very closed and restricted compared to Android (and also MacOS)
- Objective-C for apps (compiles to machine code)



MacOS and iOS



Android

Booting a system

Firmware and bootloader

- The first thing that is executed when a computer is powered on is the **firmware**, usually stored in read-only memory (ROM). This firmware initializes hardware and starts the bootloader. Most common firmware is UEFI (previously BIOS)

Firmware and bootloader

- The first thing that is executed when a computer is powered on is the **firmware**, usually stored in read-only memory (ROM). This firmware initializes hardware and starts the bootloader. Most common firmware is UEFI (previously BIOS)
- The **bootloader** understands non-volatile storage (SSDs, HDDs, CDs, USB sticks) and looks for specific boot partition that contain an operating system and all information to start it. Either it waits for user selection or it automatically selects one of the boot partitions and then starts the actual operating system. Common bootloaders are GRUB and Windows Boot Manager