Operating System Structures

DM510 Operating Systems

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

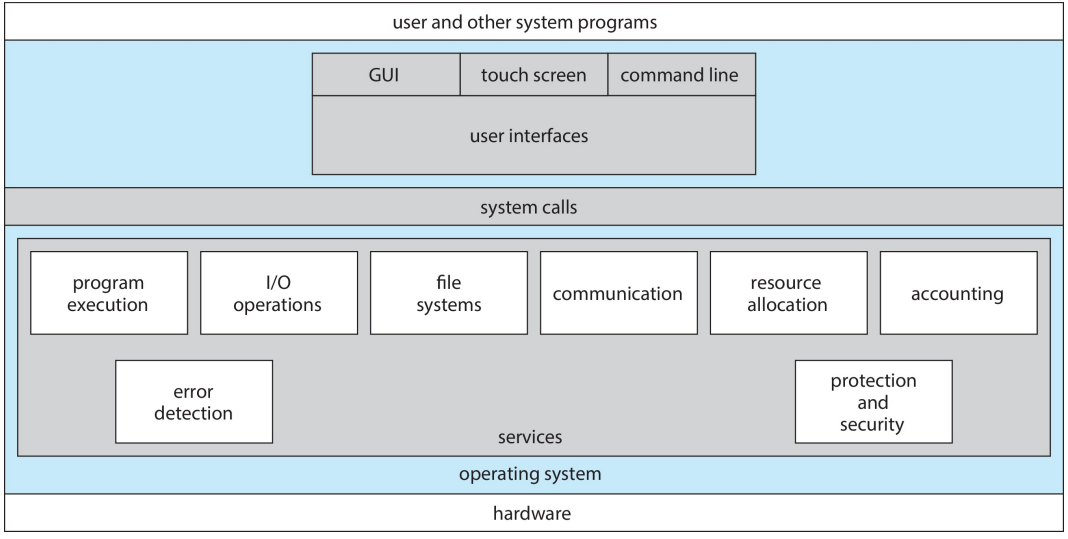
- Chapter 2: Systems structure (system services, system calls, system programs, architecture)
- Raspberry Pi delivery

Operating system services

An operating system typically provides the following services:

- **User interface:** command-line, graphical user interface, touch-screen,...
- **Program execution:** load program into memory and run, terminate process
- **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:** recovering from errors, debugging facilities
- **Resource allocation:** 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

Services within an operating system



System Calls

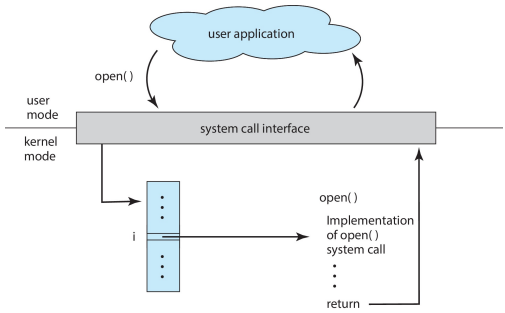
Issuing system calls

System calls are the interface from user mode programs to the kernel

System call in Linux

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

- Triggers a trap (software interrupt)
- Kernel (interrupt handler) calls function implementing system call functionality



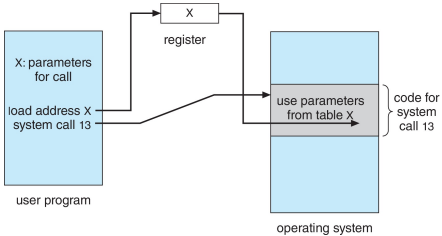
Issuing system calls

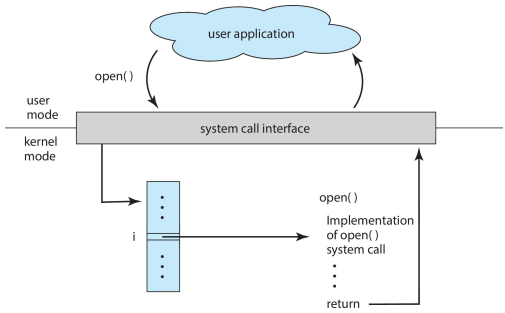
System calls are the interface from user mode programs to the kernel

System call in Linux

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

- Triggers a trap (software interrupt)
- Kernel (interrupt handler) calls function implementing system call functionality
- Parameters are passed via pointers or integers in registers. Since kernel uses different address space (see lectures on virtual memory), dereferencing pointers requires conversion





Standard APIs

- Using system calls directly is inconvenient and error-prone: need to know calling conventions
- Instead of issuing system calls directly, programmers use **Application Programming Interfaces (APIs)** provided via shared libraries, which are wrappers of the actual system calls
- We refer to these wrappers also as “system calls”

Read from file in C (Unix)

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

APIs in Windows and Unix

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

System Programs

Role of system programs

- Operating systems expose services to the user using **system programs** (sometimes called system utilities, system services) that run in user mode, for example, as command line tools or programs with graphical interface.
- They can be simple wrappers of system calls, but also significantly more complex programs.

Important system programs

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

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, ...)

File modification

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

Important system programs

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, ...)

File modification

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

Communication

Virtual connections between

- processes
- users
- computer systems

Important system programs

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, ...)

File modification

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

Communication

Virtual connections between

- processes
- users
- computer systems

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

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, ...)

File modification

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

Communication

Virtual connections between

- processes
- users
- computer systems

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)

Programming language support

- Compiler
- Interpreter
- Debugger

Important system programs

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, ...)

File modification

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

Communication

Virtual connections between

- processes
- users
- computer systems

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)

Programming language support

- Compiler
- Interpreter
- Debugger

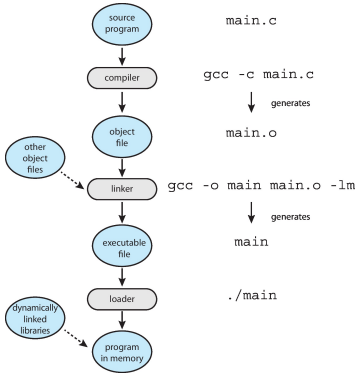
Background services

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

Executing programs

Dynamic linking and loading

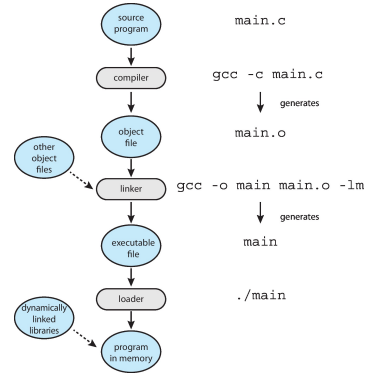
- Copy executable (instructions and data) to main memory
- Copy **dynamically linked libraries** into memory if necessary: program code shared by different programs (`.dll` or `.so`)
- Link final library addresses (e.g. function pointers) into executable's code
- Set program counter register to beginning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



Executing programs

Dynamic linking and loading

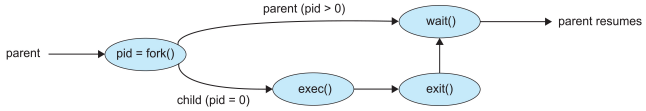
- Copy executable (instructions and data) to main memory
- Copy **dynamically linked libraries** into memory if necessary: program code shared by different programs (`.dll` or `.so`)
- Link final library addresses (e.g. function pointers) into executable's code
- Set program counter register to beginning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



Example: command line

How does a command line in Linux execute a program?

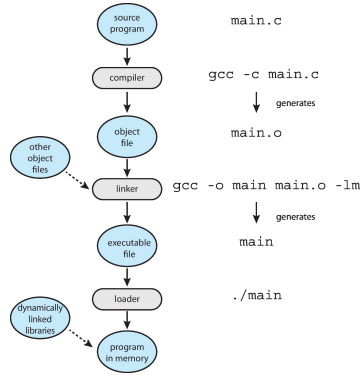
- `fork()` system call creates copy of current process
- The new process then calls loader using `exec()`



Executing programs

Dynamic linking and loading

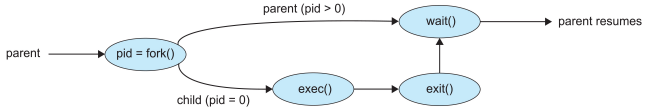
- Copy executable (instructions and data) to main memory
- Copy **dynamically linked libraries** into memory if necessary: program code shared by different programs (`.dll` or `.so`)
- Link final library addresses (e.g. function pointers) into executable's code
- Set program counter register to beginning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



Example: command line

How does a command line in Linux execute a program?

- `fork()` system call creates copy of current process
- The new process then calls loader using `exec()`



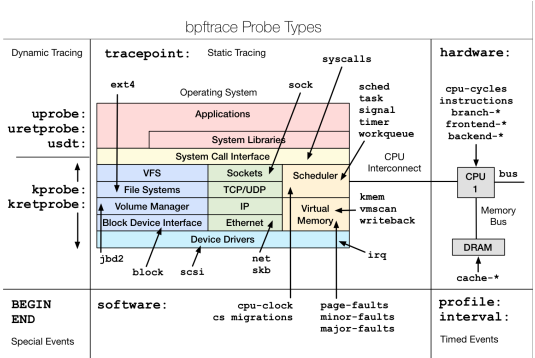
Linking/loading implementations and standard APIs differ → binary incompatibility between operating systems

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 tracing tool for Unix that makes it possible to execute user-specified scripts at various kernel events (via **eBPF**) while ensuring system's stability and performance.
- Demo in (future) lecture and course website

Operating System Implementation

Development

- In low-level language C, C++, assembly (very recently: Rust)

Development

- In low-level language C, C++, assembly (very recently: Rust)

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, C++, assembly (very recently: Rust)
- Some components (especially system programs) may be in higher programming languages

Development

- In low-level language C, C++, assembly (very recently: Rust)
- 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

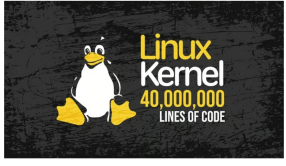
Development

- In low-level language C, C++, assembly (very recently: Rust)
- 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 kernel modules encapsulate different functionality. Sometimes kernel modules can be even loaded dynamically while the system is up, e.g. device drivers in Linux

Linux Kernel Surpasses 40 Million Lines

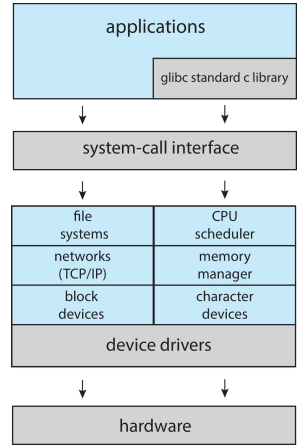
The Linux kernel has rapidly grown, reaching an impressive milestone, surpassing 40 million lines of code.

By Bobby Borisov - On January 31, 2025



Monolithic architecture

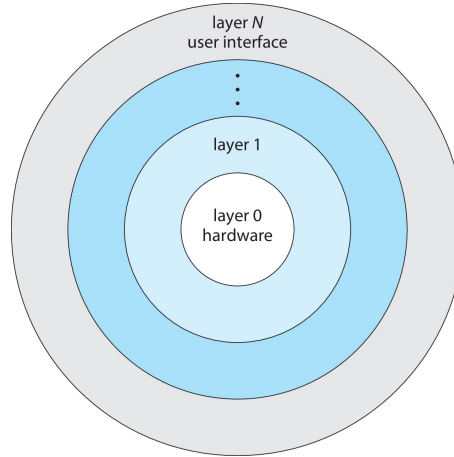
- Linux and Windows are mostly **monolithic** architectures: large kernel, complicated dependencies between modules
- Leads to more challenging debugging and unit tests
- Generally better performance (lower overhead) compared to other architectures



Linux

Layered architecture

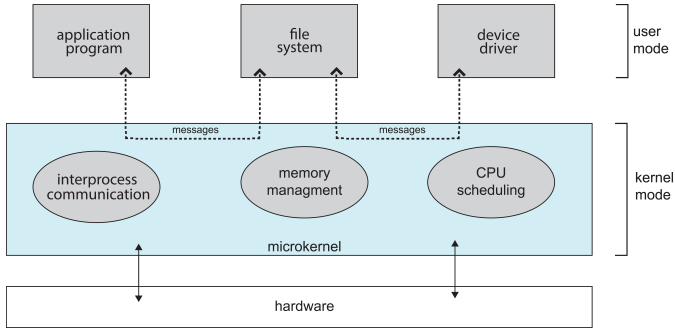
- Each layer adds more abstraction
- Each layer uses only the functions and services of lower layers
- Layers can be tested separately to follow specification: easier testing and debugging
- Disadvantages: specifying layers and arrangement is challenging, high function call overhead
- Appears to some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

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



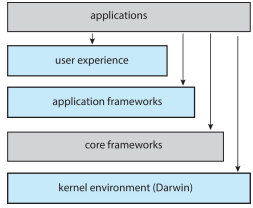
More examples

Android

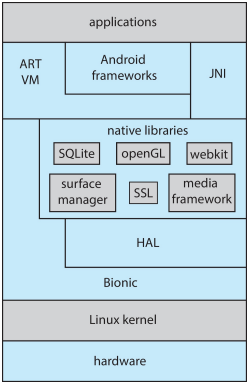
- Uses Linux kernel, very open
- Java for apps (with non-standard API). For energy reasons: ahead-of-time (AOT) instead of just-in-time (JIT) compilation

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