Introduction to OS

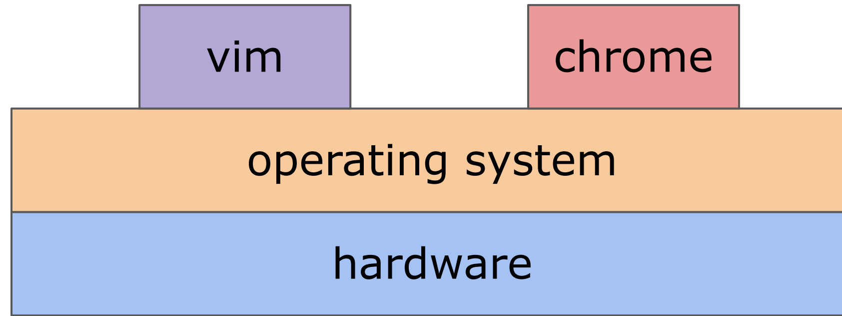
W4118 Operating Systems I

columbia-os.github.io

Credits to Jae, Jason, and David Mazières

What is an operating system?

Software between the applications and the hardware



Makes the hardware useful to the programmer. How?

What does an operating system do?

Provides abstractions for applications – [EASY?]

- Manages and hides details of hardware
- Accesses hardware through low-level interfaces unavailable to applications

Provides protection between applications – [HARD?]

- Prevents one process/user from messing with another

Is it an OS?

MacOS?

Android?

JVM?

JavaOS?

GPU/SSD/NIC on-device software?

What type of OS do we need?

Personal workstation

Data center

Mobile phone

Smart {thermostat,oven,fridge,etc.}

Airplane/Spaceship Computer

Why study OS?

OS is a mature field

- A handful of OS dominate the market
- High barrier for entry for a new OS

with many interesting open questions

- Security without sacrificing performance
- Scalability – fast networks, high core counts, low service times, big data…

and achieving high performance and efficiency is an OS issue

- e.g., high performance servers, apps that do not drain the battery, etc.

and a recent demand for new OS

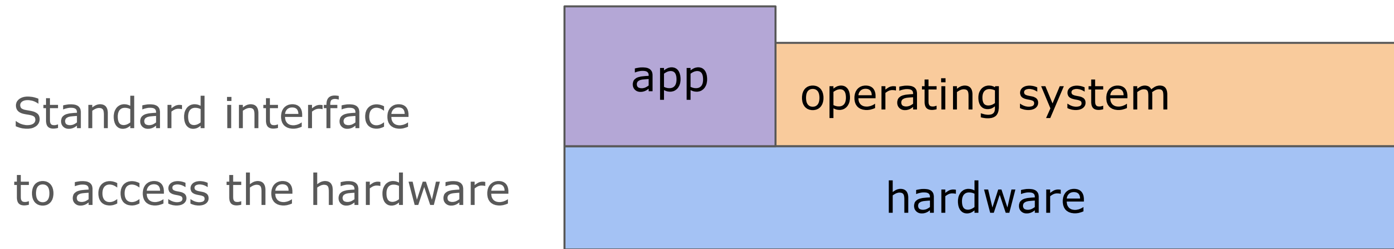
- smart devices/IoT, data center computing, new technologies (CXL, accelerators)

Early operating systems

Just a library of standard services **[no protection]**

Standard interface

to access the hardware



Simplifying assumption:

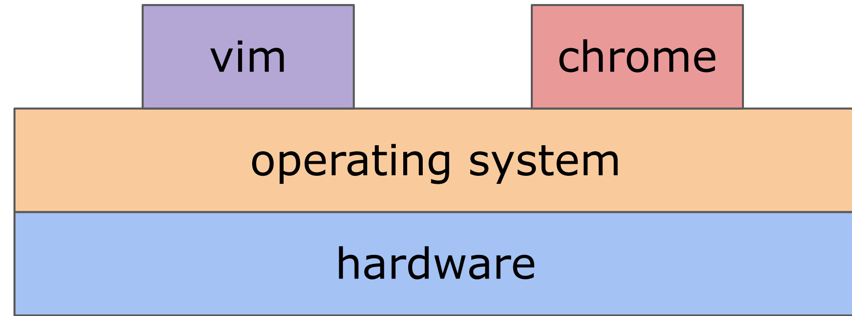
- One program/user at a time
- No bad/malicious program/user (often bad assumption)

Problem: Poor utilization

- of hardware – e.g., CPU idle while waiting for a disk access
- of human user – e.g., waiting for a program to finish execution

Idea: Multitasking

Run more than one process at a time

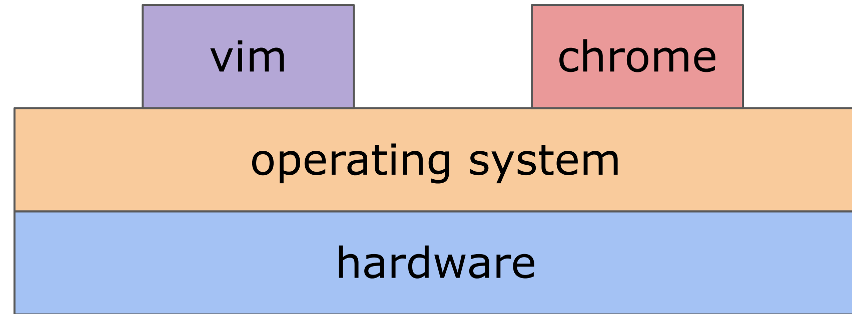


When one process blocks (waiting for disk, network, user) run another

Problem: What can a bad process do?

Idea: Multitasking

Run more than one process at a time

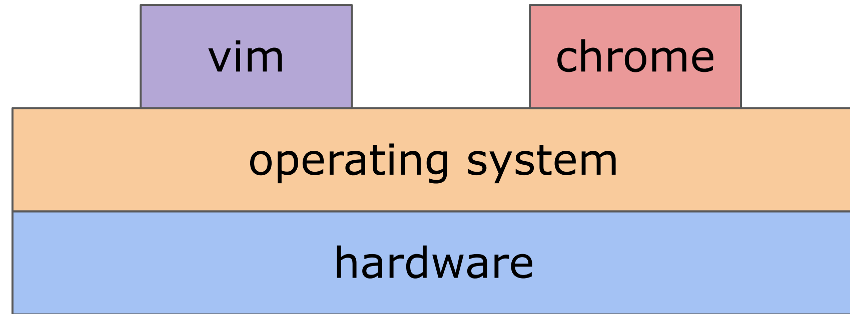


When one process blocks (waiting for disk, network, user) run another

Problem: What can a bad process do?

- Go into an infinite loop and hold the CPU forever
- Access/change another process' memory and files

Idea: Multitasking



Run more than one process at a time

When one process blocks (waiting for disk, network, user) run another

Problem: What can a bad process do?
- Go into an infinite loop and hold the CPU forever
- Access/change another process' memory

Solution: Protection
- **Preemption** – reclaim the CPU from a running process
- **Access Control** – control who can access which memory region

Protection: Isolating bad programs

Hardware support in the CPU

- applications run in unprivileged “user” CPU mode
- OS runs in privileged “kernel” CPU mode
- only privileged mode can make protection-related changes

Preemption

- forcefully reclaim resources from a bad-behaving process

Example: CPU Preemption

Kernel sets timer interrupt to vector back to the kernel

- Regains control whenever interval timer fires
- Gives CPU to another process if someone else needs it
- Note: must be in supervisor mode to set interrupt entry points
- No way for user code to hijack interrupt handler

Kernel programs timer to fire every 10msec

- must be in supervisor mode to write the appropriate timer registers
- user code cannot reprogram the timer

Result: Cannot monopolize the CPU with an infinite loop

- Worst case get 1/N of the CPU time with N CPU-hungry processes

Code Example: CPU Preemption

Protection is not enough

How can *you* monopolize the CPU?

Protection is not enough

How can *you* monopolize the CPU?

Use multiple processes

Code Example

For many years you could grind the OS to a halt with:

- `int main() { while(1) fork(); }`
- Keeps creating more processes until you overload the system

Protection is not enough

How can *you* monopolize the CPU?

Use multiple processes

Code Example

For many years you could grind the OS to a halt with:

- `int main() { while(1) fork(); }`
- Keeps creating more processes until you overload the system

Solution:

- Limit number of processes per user
- Enforce CPU quota per user rather than per process
- Kick out misbehaving users and applications

Protection: Isolating bad programs

Hardware support in the CPU

- applications run in unprivileged “user” CPU mode
- OS runs in privileged “kernel” CPU mode
- only privileged mode can make protection-related changes

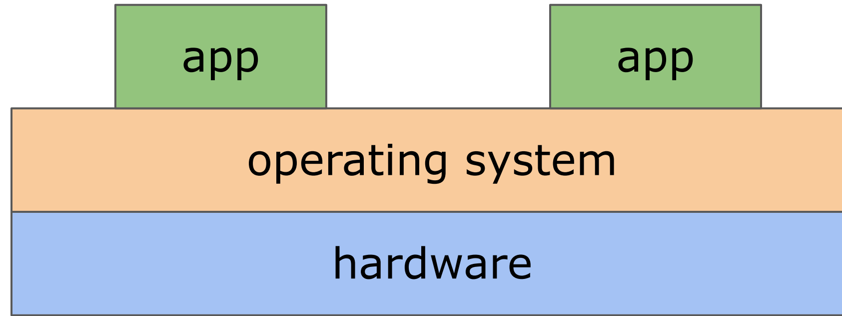
Preemption

- forcefully reclaim resources from a bad-behaving process

Interposition

- place OS between application and “important stuff”
- track everything a process is allowed to use
- for every access, check if legal

Typical OS Structure



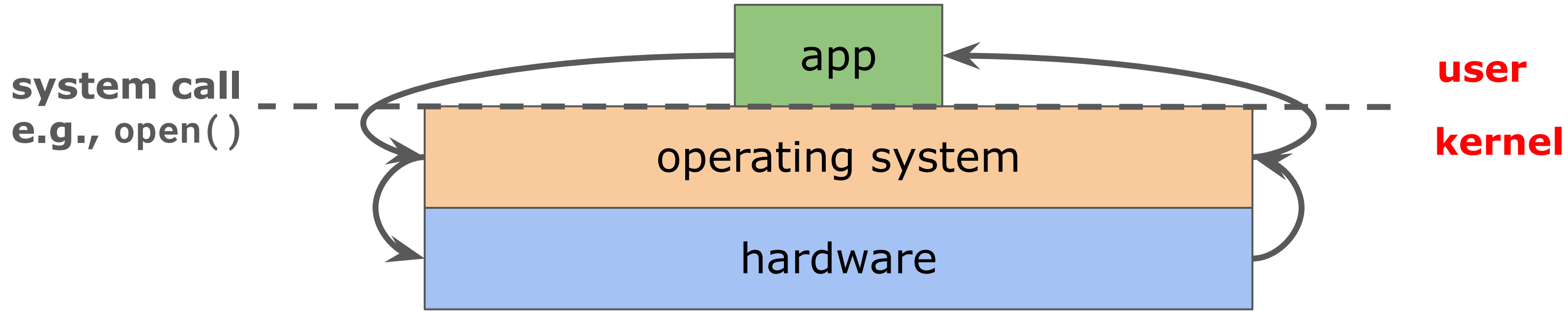
Applications run as processes in unprivileged user-level mode

The OS runs in kernel/privileged mode:

- Manages processes
- Interposes between processes and the hardware

How do applications "talk" with the kernel?

How to interpose: System Calls



Applications invoke the kernel through ***system calls:***

- Special hardware instruction gives control to the kernel, calling one of hundreds system call handlers (functions)

System Calls

Goal: Ask the kernel to do “stuff” that cannot be done in user mode

The kernel provides a well-defined system call interface:

- Applications store arguments in specific registers and *trap* in the kernel
- Kernel performs the operation and returns the result

System calls are a low-level interface/API

- Libraries/languages build higher-level abstractions, e.g., `read()` vs. `scanf()`

Example: POSIX/UNIX interface:

- `read()`,`write()`,`open()`,`close()`,`fork()`

Example: File I/O System Calls (1/2)

Application open files (or devices) by name:

- I/O happens through the open files

```
int open(char *path, int flags, /*int mode*/...);
```

- flags: O_RDONLY, O_WRONLY, O_RDWR
- O_CREAT: create the file if non-existent
- O_EXCL: (w. O_CREAT) create if the file exists already
- O_TRUNC: truncate the file
- O_APPEND: start writing from the end of the file
- mode: final argument with O_CREAT

Returns file descriptor used for all file I/O

What if something goes wrong?

Error Returns

What if `open` fails? Returns -1 (invalid fd)

Most system calls return -1 on failure

- more information about the error stored in the global `errno` variable
- is this a good design choice?

`#include <sys/errno.h>` for possible values

- 2=ENOENT "no such file or directory"
- 13=EACCESS "permission denied"

`perror()` prints a human readable message based on the `errno`

- perror("initfile");
  → `"initfile: No such file or directory"`

Example: File I/O System Calls (2/2)

```
int read(int fd, void *buf, int nbytes);
```
- Returns number of bytes read
- Returns 0 bytes at end of file, or -1 on error

```
int write(int fd, const void *buf, int nbytes);
```
- Returns number of bytes written, -1 on error

```
off_t lseek(int fd, off_t pos, int whence);
```
- whence: 0-start, 1-current, 2-end
- Returns previous file

```
int close(int fd);
```

File descriptors

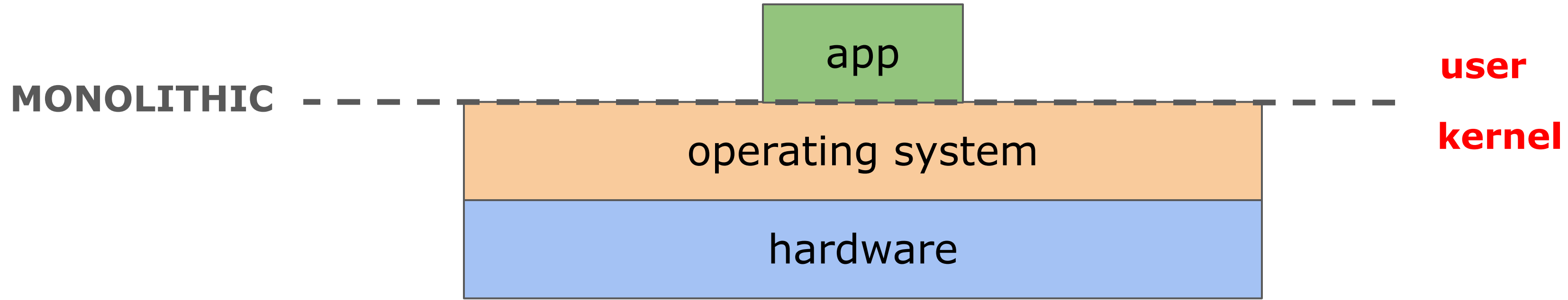
- **File descriptors are inherited by the children processes**
    - When one process spawns another, same fds by default

- **Descriptors 0, 1, 2 have a special meaning**
    - “0”: standard input (stdin)
    - “1”: standard output (stdout)
    - “2”: standard error (stderr)

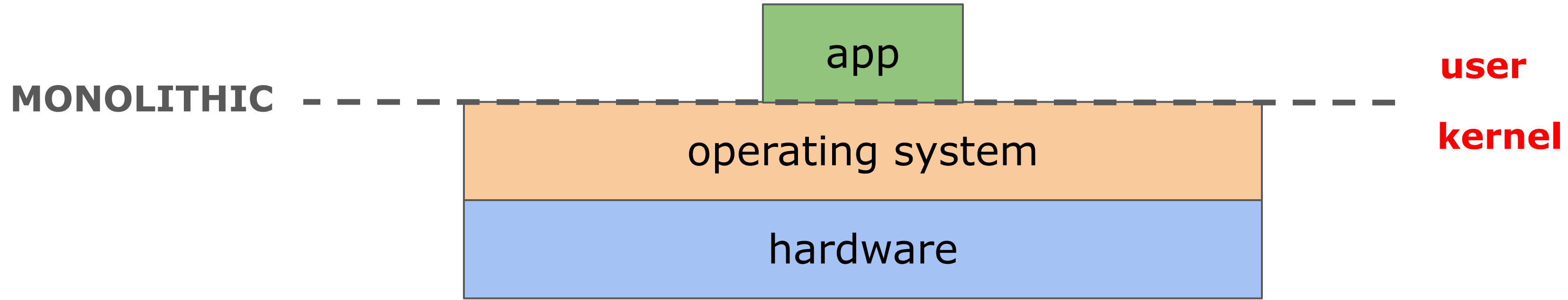
Code Example: Print a file

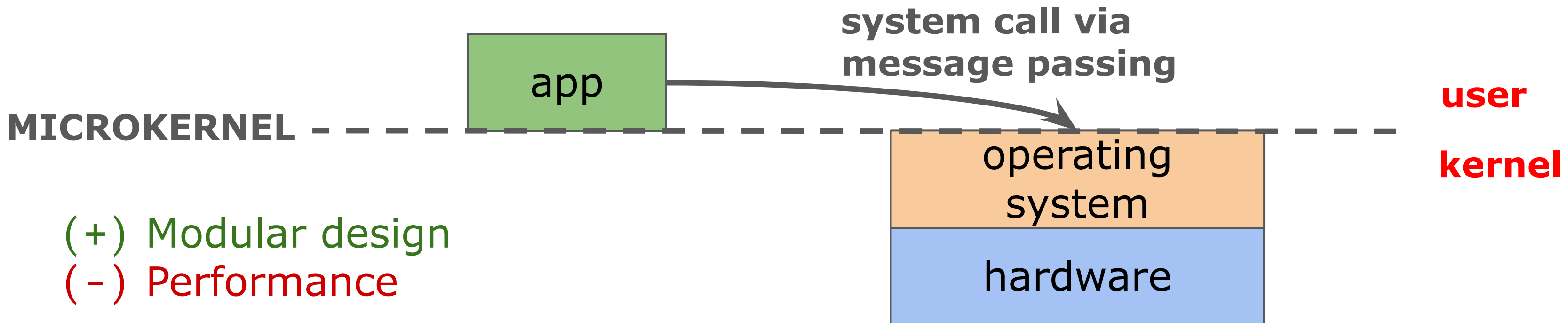
- **Print the contents of a file**
- **Can see system calls using *strace***

What are other possible OS structures?



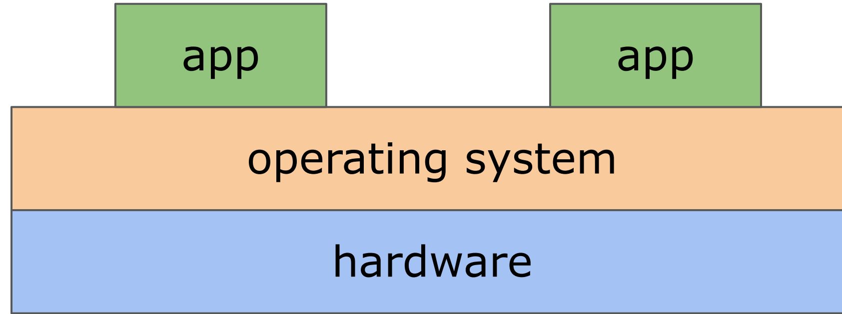
What are other possible OS structures?





(+) Modular design
(-) Performance

What about memory?



How can we protect the memory of one process from another?

Solution: Memory Virtualization

Key idea: If you can't name it, you can't touch it

Definitions:

- *Address space*: all memory locations a program can name (linear)
- *Virtual address*: address in process' address space
- *Physical address*: address in physical memory
- *Translation*: map physical to virtual address

Translation done on every load, store, and instruction fetch

- Done in hardware in modern CPUs for speed

The OS sets up the mapping and ensures one process' translation do not include another process' memory

Memory Virtualization Advanced Features

Kernel-only virtual addresses

- The kernel memory is part of all address spaces
- Apps cannot touch kernel memory

Read-only virtual addresses are useful

- Sharing of code pages between applications, e.g., libraries
- Inter-process communication, memory-mapped files, …

Execute-disabled VA

- Makes code injection attacks harder

Code Example: Memory Virtualization

Putting it all together: System Contexts

1. **User-level** – CPU in user mode running an application
2. **Kernel process context** – Running kernel code on behalf of an application
   - system call, exception handling (page fault, numeric exception, etc.)
3. **Kernel code not associated with a process:**
   - timer interrupt
   - device interrupt
   - “softirq”, “tasklet” (Linux-specific)
4. **Context switch code** – Change which process is running
   - Requires changing the current address space
5. **Idle** – Do (almost) nothing

Policies: Resource Allocation

Multitasking enabled higher resource utilization

Example:

- Process downloads a big file over the internet, e.g., a movie
- You play a game while downloading the file
- CPU utilization higher than if just downloading

Is multitasking always better?

Resource Allocation and Performance

Multitasking enabled higher resource utilization

Example:
- Process downloads a big file over the internet, e.g., a movie
- You play a game while downloading the file
- CPU utilization higher than if just downloading

Is multitasking always better? Depends on the cost of switching

Example: Disk much much slower than memory
- 1 GiB memory in the machine
- 2 processes run, needing 1 GiB each
- When switching between processes need to load and store data to/from disk
- Faster to run one at a time rather than context switching

Conclusion: Scheduling is a hard problem

A Little Bit of History

Predecessor of UNIX

Multics ("MULTiplexed Information and Computing Service")

- Developed in MIT, targeting time-sharing computers
- Supported modularity (i.e., plugging, unplugging components) and (some) protection (virtual memory)

GE 645 running at MIT in 1967

The arrival of UNIX

Developed in Bell Labs in 1969-1971

Initially written in assembly but rewritten in C in 1973

Unix philosophy:
- provide a set of simple tools, each of which performs a limited, well-defined function
- simple API, e.g., everything is a file

Why was it successful?

PDP-11

The arrival of UNIX

Developed in Bell Labs in 1969-1971

Initially written in assembly but rewritten in C in 1973

Unix philosophy:

- provide a set of simple tools, each of which performs a limited, well-defined function
- simple API, e.g., everything is a file

Why was it successful?

1. Open source software – everyone can use it
2. Written in C – portable

PDP-11

The evolution of UNIX → Linux

1977 - Berkeley Software Distributions or **BSD**
- macOS is based on its descendants

1980-90s – UNIX wars
- Different companies (ATT, DEC, Sun, IBM) commercialize UNIX
- Led to closed-source implementations of UNIX

1991 – First version of Linux released
- Fully open-source
- Enforces the descendants to be open-source (“GPL” license)