OS Fundamentals

CSC 420 - Operating Systems - Lyon College Spring 2024

Marcus Birkenkrahe

January 24, 2024

README

Image: where is the operating system ...

A summary of material for an overview of OS fundamentals based mostly on the textbook by Silberschatz et al. (2018), chapter 1.

Fundamentals of Operating Systems

What you will learn:

- What OS do and what they don't do
- Computer system components
- The OS kernel and interrupts
- System startup and daemons
- Multiprogramming and timesharing

- Virtualization
- Distributed systems
- Kernel data structures
- Computing environments
- FOSS OS

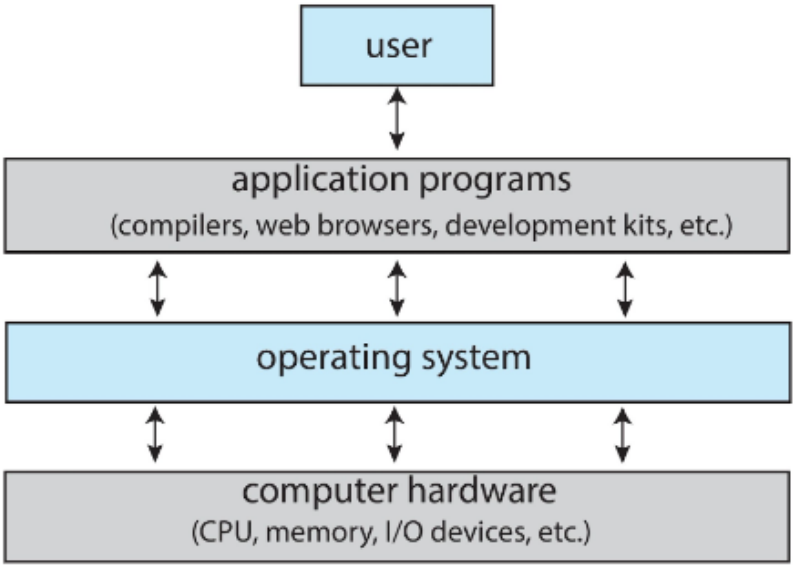
See also:

1. Fundamentals of Operating Sytems in 77 videos (Davis, 2018).
2. Slides for chapter 1: Introduction of Silberschatz et al. (2018)
3. Operating systems foundations with Linux on the Raspberry Pi (2019)

Introduction

An operating system is software that manages a computer's hardware.
Before 1960s: every system action had to be programmed explicitly.
Early 1960s: allowed programmers to focus on application development.
OS are complex managers of all of the computer's resources

Computer system components



Example flow:

1. User presses letter 'A' on keyboard
2. Editor (app) requests CPU time/memory to display 'A'
3. OS schedules CPU and allocates memory (RAM)
4. CPU works away in concert with RAM

What issues could any of these actors have?

1. What if another process is already running - priority?
2. What if the input isn't "clear" (like pressing two keys at once)?
3. What if there is more than one CPU?
4. How much memory is needed?
5. Is the file system involved in this at all?

What does the user want?

- Ease of use
- Good performance
- Doesn't care about resource utilization
- Depends on scenario: workstation, controller, PC, phone, mainframe
- Most devices are resource poor (CPU/memory can be stretched)

What does the OS want?

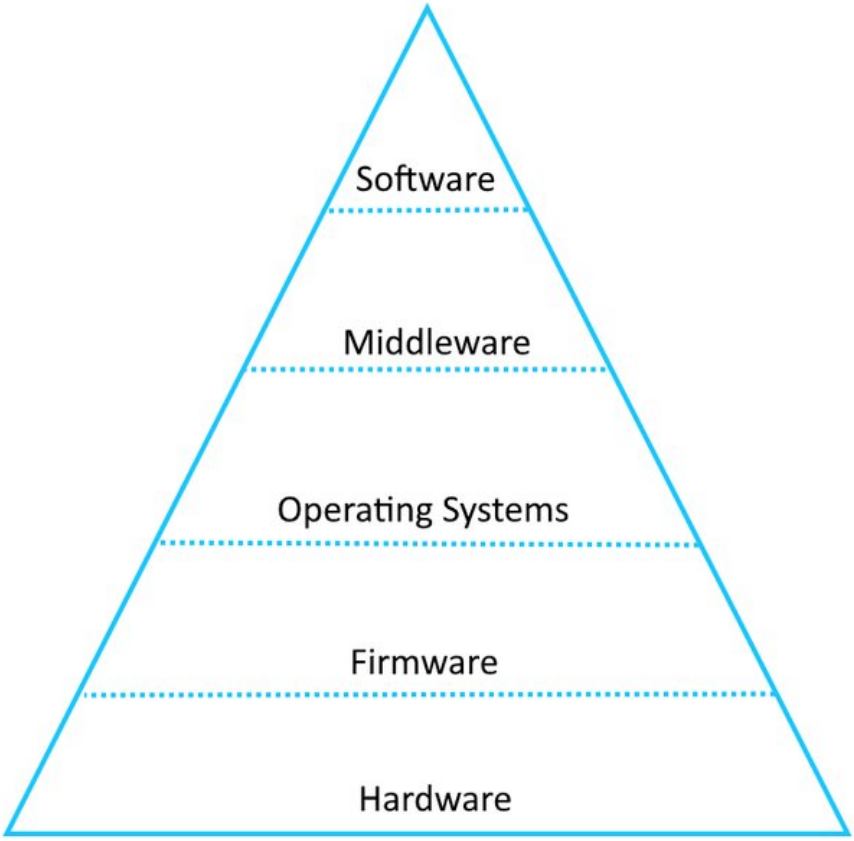
- Manage all resources
- Have total access to all resources
- Decide between conflicting requests
- Lives a fast-paced life compared to the user
- Gained in functionality and in size (Linux: 30M lines)

What's the kernel?

The kernel ('Kern' is German for 'nucleus') is the one program running at all times on the computer. Everything else is either a system program or an application program.

Today's OS include "middleware" (additional service frameworks), like databases, multimedia, and graphics. For example `React.js` is a popular framework to develop web applications with JavaScript and TypeScript.

Old-style OS like DOS (Disk Operating Systems) did only a fraction of today's OS work load - operating the disk drives. You had to configure the interface for every application yourself.



Alternative approaches

- Exokernels (applications get more control)
- Unikernel (single app, e.g. cloud, in a virtual machine)
- Microkernels (reduce kernel to absolute essentials)
- Distributed operating systems (spread out over machines)
- Persistent operating systems (all main memory is persistent)
- Agent-based models (autnomous agents structure jobs)

A minimal pseudo kernel in Python

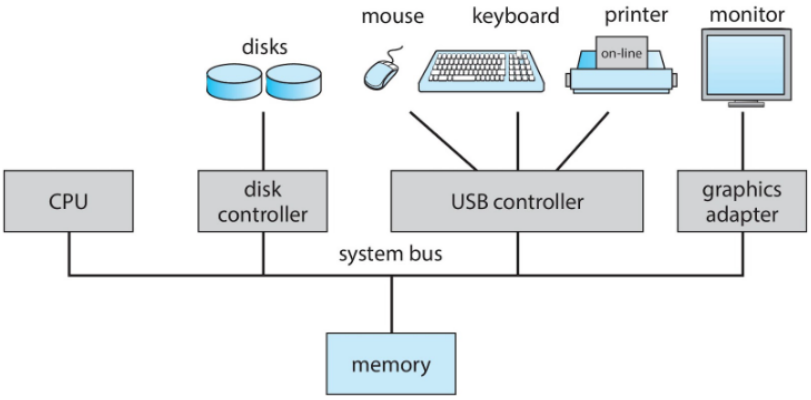
To run, tangle and start `kernel.py` script in the shell, or execute here and switch to `*Python*` buffer to enter kernel operations.

```
def handle_command(command):
    # This function handles the commands input by the user
    if command == "exit":
        # If the command is 'exit', stop the kernel loop
        return False
    elif command == "hello":
        # If the command is 'hello', print a greeting message
        print("Hello from the pseudo-kernel!")
    else:
        # Handle any unknown commands
        print(f"Unknown command: {command}")
    return True

def minimal_kernel():
    # This function represents the main loop of the pseudo-kernel
    running = True
    while running:
        # Get command input from the user
        command = input("kernel> ")
        # Process the command and decide whether to continue running
        running = handle_command(command.strip())

if __name__ == "__main__":
    # Entry point of the program
    minimal_kernel()
```

Computer system organization



A memory-centric computer model:

1. System memory (RAM)
2. System software and hardware (CPU etc.)
3. Application software and hardware (peripherals, I/O devices)

[This is the same approach that is used in the ARM book.]

An Input/Output operation, dissected

1. Device driver loads registers in the device controller.
2. Device controller decides which action to take, like 'read character' from keyboard.
3. Controller transfers data from device to its local buffer.
4. When transfer is complete, device driver is informed via an **interrupt** signal from the controller.

Process, interrupted

- Interrupts are used to handle asynchronous events.
- Device controllers and hardware faults raise interrupts.
- To handle the workload, the OS uses an event-vector table.

Event vector tables

There are two interrupt request lines:

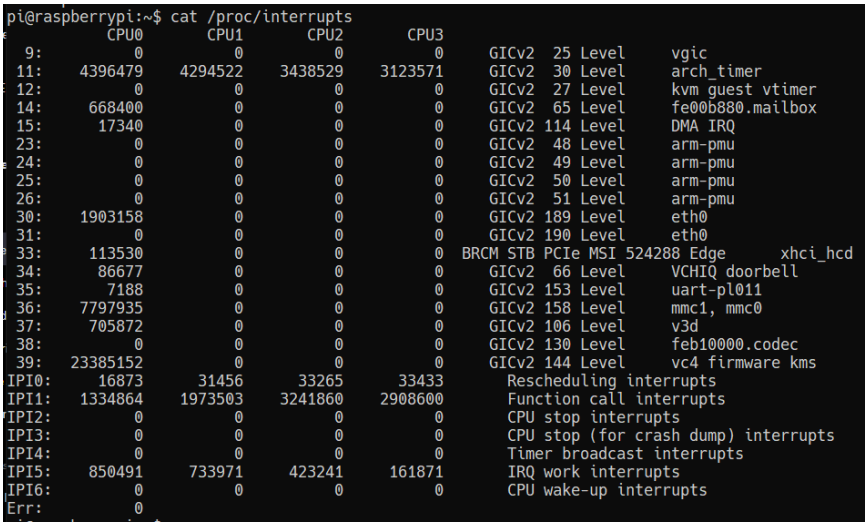
1. Non-maskable interrupts for unrecoverable memory errors.
2. Maskable interrupts can be switched off temporarily.

Efficient because there are tens of thousands of processes most of whom don't need to be interrupted.

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Figure 1: Intel processor event-vector table

Interrupt log and priorities



```
pi@raspberrypi:~$ cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
  9:          0          0          0          0     GICv2  25 Level     vgic
 11:    4396479    4294522    3438529    3123571     GICv2  30 Level     arch_timer
 12:          0          0          0          0     GICv2  27 Level     kvm guest vtimer
 14:     668400          0          0          0     GICv2  65 Level     fe00b880.mailbox
 15:      17340          0          0          0     GICv2 114 Level     DMA IRQ
 23:          0          0          0          0     GICv2  48 Level     arm-pmu
 24:          0          0          0          0     GICv2  49 Level     arm-pmu
 25:          0          0          0          0     GICv2  50 Level     arm-pmu
 26:          0          0          0          0     GICv2  51 Level     arm-pmu
 30:    1903158          0          0          0     GICv2 189 Level     eth0
 31:          0          0          0          0     GICv2 190 Level     eth0
 33:     113530          0          0          0  BRCM STB PCIe MSI 524288 Edge      xhci_hcd
 34:      86677          0          0          0     GICv2  66 Level     VCHIQ doorbell
 35:       7188          0          0          0     GICv2 153 Level     uart-pl011
 36:    7797935          0          0          0     GICv2 158 Level     mmc1, mmc0
 37:     705872          0          0          0     GICv2 106 Level     v3d
 38:          0          0          0          0     GICv2 130 Level     feb10000.codec
 39:   23385152          0          0          0     GICv2 144 Level     vc4 firmware kms
IPI0:     16873      31456      33265      33433       Rescheduling interrupts
IPI1:   1334864    1973503    3241860    2908600       Function call interrupts
IPI2:         0          0          0          0       CPU stop interrupts
IPI3:         0          0          0          0       CPU stop (for crash dump) interrupts
IPI4:         0          0          0          0       Timer broadcast interrupts
IPI5:    850491     733971     423241     161871       IRQ work interrupts
IPI6:         0          0          0          0       CPU wake-up interrupts
Err:          0
```

Figure 2: interrupt log on Pi 4 (uptime: 4 days)

The file `/proc/interrupts` on Linux systems provides information about the interrupts, but it doesn't directly show interrupt priorities.

In standard Linux systems, hardware interrupts are handled without a fixed priority scheme.

Explanation of the figure:

- IRQ number: This is a unique identifier for each interrupt. For example, 11 or 30.
- CPU columns (CPU0, CPU1, etc.): These columns show the number of times each CPU has serviced the corresponding interrupt. For instance, interrupt 11 has been serviced 4396479 times by CPU0, 4294522 times by CPU1, and so on. This also gives an insight into the load balancing of interrupt handling between different CPUs.
- Interrupt Type (GICv2, BRCM STB PCIe MSI, etc.): This shows the type of interrupt controller that is handling the interrupt. GICv2 refers to the Generic Interrupt Controller version 2, which is commonly used in ARM processors.
- Trigger Type (Level, Edge): This tells you whether the interrupt is level-triggered or edge-triggered. Level-triggered interrupts mean the

interrupt line remains asserted until the interrupt is cleared. Edge-triggered means the interrupt is generated on a change of state (from low to high or high to low).

- Interrupt Descriptor: This is a human-readable description of what the interrupt is used for. For example, arch_timer is the system timer, eth0 is the first Ethernet interface, and xhci_hcd is a USB 3.0 host controller driver.
- The numbers in the CPU columns: These are the counts of how many times the interrupt has been serviced by the respective CPU since the last reboot.

System startup

_ The CPU needs something in memory to work with.

- The first program loaded at boot is the bootstrap program.
- It is stored in the ROM (Read Only Memory) or EEPROM (Electrically Erasable Programmable ROM) - non-volatile firmware memory.
- The bootstrap program initializes the system and loads the kernel.
- It starts system daemons - service programs outside of the kernel.
- The configuration data to boot are the BIOS (Basic Input Output System) data. (Needed e.g. when you want to boot from USB).

The Linux system daemon `systemd`

- Once the kernel is loaded it can start serving system & users.

- Its first daemon is `systemd`, which starts many other daemons.
- The kernel now waits for interrupts to call it to service.

Multiprogramming (batch system)

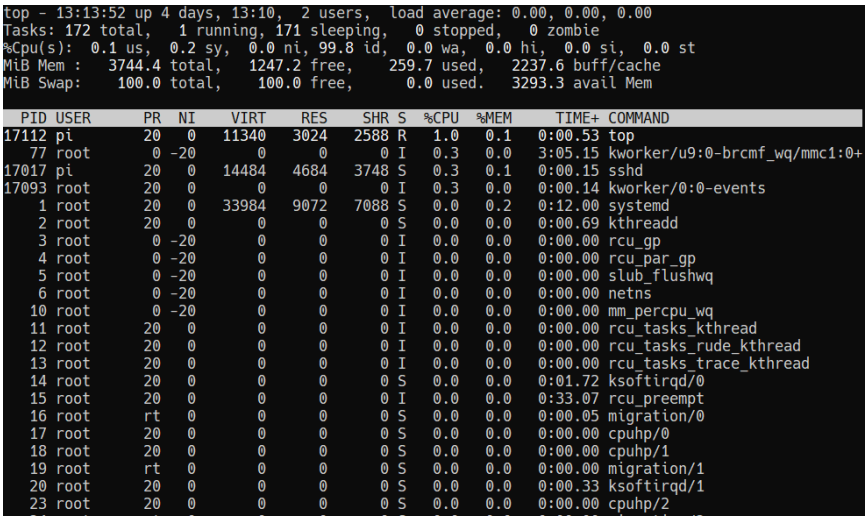
- Jobs (code + data) are organized to keep the CPU always busy.
- One job is selected and run via job scheduling.
- Jobs may have to wait (e.g. for I/O) and the OS switches to another.

In the old days, without batch operations, you'd have to wait until one job is finished before beginning another.

In R, the `R CMD BATCH` is an example for a scripting language in batch mode:

```
$ echo "str(mtcars)" > batchTest.R
$ R CMD BATCH batchTest.R
$ ls batch*
$ cat batchTest.Rout
```

`top` processes occupying the CPU



How and what the CPU is busy with you can see with the `top` Linux program that refreshes every 5 secs or so.

The screenshot also shows the `sshd` that supervises the `ssh` (secure shell) program used by me to connect to the Pi from my Windows box.

Timesharing (multitasking)

- CPU switches jobs so fast that users get the illusion of interactive computing
- Timesharing gives rise to the different OS management tasks:
  1. Memory management (e.g. assign and use local variables)
  2. Process management (e.g. using the CPU)
  3. Scheduling management (e.g. switching processes)
  4. Device management (e.g. find and use printer)
  5. File management (e.g. find and use files)

We will not dive into these theoretically (much), but practically with the help of the shell and the shell scripting language `bash`.

Sources

- Silberschatz et al, Operating System Concepts (10e), 2018, Wiley.
- Shotts, The Unix Command Line (2e), 2019, NoStarch.
- Will, Operating System Basics, 2014, YouTube.
- Davis, Fundamentals of Operating Systems, 2018, YouTube.
- Vanderbauwhede/Singer, Operating systems foundations with Linux on the Raspberry Pi, 2019, ARM education.