Lecture: Introduction

Anton Burtsev
Let's take a brief look at how computers work.
CPU

- 1 CPU
  - 4 cores
  - 2 logical (HT) threads each
Memory
Memory abstraction

| WRITE($addr$, $value$) $\rightarrow$ $\emptyset$ |
| Store $value$ in the storage cell identified by $addr$. |
| READ($addr$) $\rightarrow$ $value$ |
| Return the $value$ argument to the most recent WRITE call referencing $addr$. |
What does CPU do internally?
CPU execution loop

- CPU repeatedly reads instructions from memory
- Executes them
- Example

ADD EDX, EAX, EBX

// EDX = EAX + EBX
Simple observation

• Hardware executes instructions one by one
What is an operating system?
Task #1: Run your code on a piece of hardware

- Read CPU manual
- A tiny boot layer
  - Initialize CPU
  - Jump to the entry point of your program
    - main()
- This can be the beginning of your OS!
Task #2: Print something on the screen

- On the screen or serial line

```c
printf() {
    ...  
    asm("mov [<magic constant>], char");
    ...
}
```
I/O Devices

Memory Bus

PCI Bus

"South Bridge"

PCH

SATA

USB

NIC

PCI-e Attached SSD
Task #2: Print something on the screen

- On the screen or serial line

```c
printf() {

    ...
    if (vga) {
        asm("mov ["magic constant 1"], char");
    } else if (serial) {
        asm("out "magic constant 2", char");
    }
    ...
}
```

OS
A more general interface

- First device driver

```c
printf() {
    ...
    putchar(char);
    ...
}
```
Device drivers

• Abstract hardware
  • Provide high-level interface
  • Hide minor differences
  • Implement some optimizations
    – Batch requests

• Examples
  • Console, disk, network interface
  • ...virtually any piece of hardware you know
OS is like a library that provides a collection of useful functions
How hard it is to boot into main and print something on the screen?

- If you want to run this demo
  
  https://github.com/mars-research/hello-os.git

  `printf("Hello world\n");`
Task #3: Want to run two programs

- What does it mean?
  - Only one CPU
- Run one, then run another one

```
main() {
    ...
    yield()
}

main() {
    ...
    yield()
}
```

Save/restore
Very much like car sharing
Time sharing

• Programs use CPU in turns
  • One program runs
  • Then OS takes control
  • Launches another program
  • Then another program runs
  • OS takes control again
  • ...

Task #3: Want to run two programs

- Exit into the kernel periodically
- Context switch
  - Save state of one program
  - Restore state of another program
What is this state?
State of the program

- Roughly it’s
  - Registers
  - Memory

- Plus some state (data structures) in the kernel associated with the program
  - Information about files opened by the program, i.e. file descriptors
  - Information about network flows
  - Information about address space, loaded libraries, communication channels to other programs, etc.
Saving and restoring state

• Note that you do not really have to save/restore in-kernel state on the context switch
  • It’s in the kernel already, i.e., in some part of the memory where kernel keeps its data structures
  • You only have to switch from using one to using another
    – i.e., instead of using the file descriptor table (can be as simple as array) for program X start using at file descriptor table for program Y
What about memory?
Two programs, one memory?

main() {
    ...
    yield()
}

main() {
    ...
    yield()
}

Save/restore
Time-share memory

- Well you can copy in and out the state of the program into a region of memory where it can run
  - Similar to time-sharing the CPU
Time-share memory

• Well you can copy in and out the state of the program into a region of memory where it can run
  • Similar to time-sharing the CPU

• What do you think is wrong with this approach?
Time-share memory

- Well you can copy in and out the state of the program into a region of memory where it can run
  - Similar to time-sharing the CPU

- What do you think is wrong with this approach?
  - Unlike registers the state of the program in memory can be large
  - Takes time to copy it in and out
Virtual address spaces

- Illusion of a private memory for each application
  - Keep a description of an address space
  - In one of the registers

- OS maintains description of address spaces
  - Switches between them
Address spaces with page tables
Page tables high-level idea

- Break up memory into 4096-byte chunks called pages
  - Modern hardware supports 2MB, 4MB, and 1GB pages
- Independently control mapping for each page of linear address space
Staying in control
• What if one program fails to release the CPU?
• It will run forever. Need a way to preempt it. How?
Scheduling

- Pick which application to run next
  - And for how long
- Illusion of a private CPU for each task
  - Frequent context switching
Isolation
• What if one faulty program corrupts the kernel?
• Or other programs?
No isolation: open space office
Isolated rooms
Each process has a private address space
Each process maps the kernel

- It's not strictly required
  - But convenient for system calls
  - No need to change the page table when process enters the kernel with a system call
- Things are much faster!
P1 and P2 can't access each other memory
• What about communication?
• Can we invoke a function in a kernel?
Files and network
• What if you want to save some data to a file?
• What if you want to save some data to a file?
• Permanent storage
  • E.g., disks
• Disks are just arrays of blocks
  • write(block_number, block_data)
I/O Devices

- Memory Bus
  - PCI Bus
    - "South Bridge"
      - PCH
        - SATA
        - USB
      - NIC
    - PCI-e Attached SSD
• File system and block device provide similar abstractions

• Permanent storage
  • E.g., disks

• Disks are just arrays of blocks
  • write(block_number, block_data)

• Files
  • High level abstraction for saving data
    • fd = open(“contacts.txt”);
    • fprintf(fd, “Name:%s\n”, name);
main() {
    ...
    open("contacts.txt");
    ...
}

File system
File system and block layer

- Reliable storage on top of raw disc blocks
- Disks are just arrays of blocks
  
  ```c
  write(block_number, block_data)
  ```

- Human readable names (files)
  
  ```c
  fd = open("contacts.txt");
  fprintf(fd, "Name:%s\n", name);
  ```

<table>
<thead>
<tr>
<th>System calls</th>
<th>File descriptors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pathnames</td>
<td>Recursive lookup</td>
</tr>
<tr>
<td>Directories</td>
<td>Directory inodes</td>
</tr>
<tr>
<td>Files</td>
<td>Inodes and block allocator</td>
</tr>
<tr>
<td>Transactions</td>
<td>Logging</td>
</tr>
<tr>
<td>Blocks</td>
<td>Buffer cache</td>
</tr>
</tbody>
</table>
What if you want to send data over the network?

• Similar idea
  • Send/receive Ethernet packets (Level 2)
  • Two low level

• Sockets
  • High level abstraction for sending data
• Linux/Windows/Mac
Recap

- Run multiple programs
  - Each has illusion of a private memory and CPU
    - Context switching
    - Isolation and protection
  - Management of resources
    - Scheduling (management of CPU)
    - Memory management (management of physical memory)

- High-level abstractions for I/O
  - File systems
    - Multiple files, concurrent I/O requests
    - Consistency, caching
  - Network protocols
    - Multiple virtual network connections
Questions?