Lecture 01: Introduction

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

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

Hyper-Threading (logical threads)

Cores (4)

Socket
## Memory abstraction

<table>
<thead>
<tr>
<th>WRITE( (addr, value) \rightarrow \emptyset )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store \textit{value} in the storage cell identified by \textit{addr}.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>READ( (addr) \rightarrow value )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return the \textit{value} argument to the most recent WRITE call referencing \textit{addr}.</td>
</tr>
</tbody>
</table>
What does CPU do internally?
CPU execution loop

- CPU repeatedly reads instructions from memory
- Executes them

Example

```
ADD EAX, EBX
// EAX = 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");
    ...
}
```

OS
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
Task #3: Want to run two programs

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

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

```c
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?

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

```java
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
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
P1 and P2 can't access each other memory
• What about communication?
• Can we invoke a function in a kernel?
Files and network
• Want to save some data to a file?
• Want to save some data in a file?
• Permanent storage
  • E.g., disks
• Disks are just arrays of blocks
  • write(block_number, block_data)
• 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);
File system

main() {
    ...
    open("contacts.txt");
    ...
}

File system
- 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?
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");