Introduction to Operating Systems

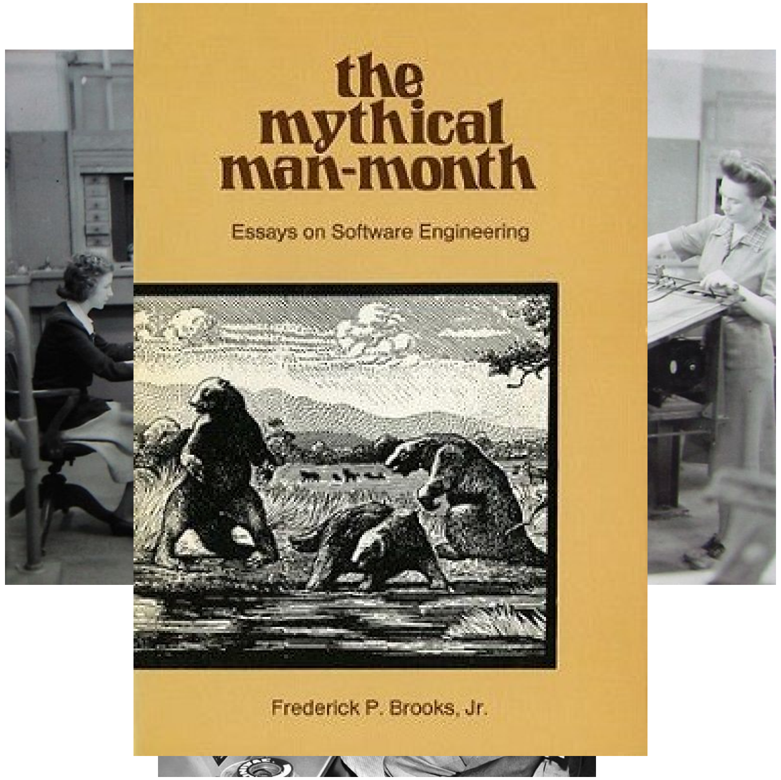
Dr Sana Belguith

History of Operating Systems

- Phase 0: no operating systems (1940-1955):
  - Program in machine language
  - Program manually loaded via card decks
- Phase 1: Introducing Terminals (1955-1970)
  - Move the users away from the computer, give them terminal
  - OS is a batch monitors, a program that: Load a user job, Run it, Move to the next

- First OS (1963- 1968)
  - Operating Systems did not really work
  - OS/360 introduced in 1963… worked in 1968
  - Systems were enormously complicated
  - Written in assembly code
  - No structured programming

  - (extra read) The Mythical Man-Month



What is an operating system?

- A computer program that
  - **Multiplexes** hardware resources
  - Implements resource **abstractions**

It is just another program… but a large and complex one

  - most complex piece of code you would have seen so far

- **Multiplexing:** allows multiple people or programs to use the same set of hardware resources—processors, memory, disks, network connection—safely and efficiently.
- **Abstractions:** processes, threads, address spaces, files, and sockets—simplify the usage of hardware resources by organizing information or implementing new capabilities.

Why study OS?

- **Reality:** this is how computers really work, and as a computer scientist or engineer you should know how computers really work.
- **Ubiquity:** operating systems are everywhere, and you are likely to eventually encounter them or their limitations.
- **Beauty:** operating systems are examples of mature solutions to difficult design and engineering problems. Studying them will improve your ability to design and implement abstractions.

What is an OS?

Protection Boundary

Kernel
File System Networking
Device Drivers Processes
Virtual Memory

Hardware/Software Interface


Protection Boundaries

- Multiple privilege levels
- Different software can run with different privileges
- Processors provide at least two different modes
  - User space: how “regular” program run
  - Kernel mode: How the kernel run
- The mode determine a number of things
  - What instructions may be executed
  - How addresses are translated
  - What memory locations can be accessed (through translation)

Example Intel

- Four modes
  - Ring 0: most privileged run the kernel here
  - Ring 1/2: ignored in most environment. Can run less privileged code (e.g. third party device drivers or virtual machine monitors etc.)
  - Ring 3: where “normal” processes live
- Memory is divided in segments
  - Each segment has a privilege level (0 to 3)
  - Processor maintain a current privilege level (CPL) generally the CPL of the segment containing the instruction currently executing
  - Can read/write in segment when CPL >= segment privilege
  - Cannot directly call code in segment where CPL < segment privilege

Example MIPS*

- Standard two modes processor
  - User mode: access CPU registers; flat uniform virtual memory address space
  - Kernel mode: can access memory mapping hardware and special registers

* Microprocessor without Interlocked Pipelined Stages

Changing Protection Level

- How do we transfer control between applications and kernel?
- When do we transfer control between applications and kernel?

Changing Protection Level: When?

- Sleeping beauty approach
  - Wait for something to happens to wake up the kernel
  - What might that be?
    - System calls: an application wants the operating to do something on its behalf (e.g. access some hardware)
    - Trap: an application does unintentionally something it should not (e.g. divide by zero, read an address it should not etc.)
    - Interrupts: An asynchronous event (e.g. I/O completion)
- Alarm clock approach
  - Set a timer that generate an interrupt when it finishes

System Call

- A system call is a way for programs to interact with the operating system. A computer program makes a system call when it makes a request to the operating system’s kernel.

- System call provides the services of the operating system to the user programs via Application Program Interface (API). It provides an interface between a process and operating system to allow user-level processes to request services of the operating system.

Trap

- Traps are the general means for invoking the kernel from user code.
- We usually think of a trap as an unintended request for kernel service, say that caused by a programming error such as using a bad address or dividing by zero.
- Traps can be triggered by an exception or error in a process when executing a function. Conditions like division by zero, a breakpoint, or invalid memory access occur synchronously with the execution of a program. They can result in an exception that changes the operation of the processor. Once the cause of the exception has been handled, the processor returns to its previous activity.

Interrupt

- An interrupt is a hardware or software signal that demand attention from the OS.
- Unlike a trap, which is handled as part of the program that caused it (though within the operating system in privileged mode), an interrupt is handled independently of any user program.
- For example, a trap caused by dividing by zero is considered an action of the currently running program; any response directly affects that program. But the response to an interrupt from a disk controller may or may not have an indirect effect on the currently running program and has no direct effect (other than slowing it down a bit as the processor deals with the interrupt).

Operating System Abstractions

- Abstractions **simplify applications design** by:
  - Hiding undesirable properties;
  - Adding new capabilities;
  - Organizing information.
- Abstractions provide an **interface** to programmers that separates **policy** – what the interface commits to accomplish – from the **mechanism** – how the interface is implemented.

Abstractions

- Threads
  - Abstract the CPU
- Address space
  - Abstract the memory
- Files
  - Abstract the disk

Abstraction example: File

- What **undesirable properties** file systems hide?
  - Disk are slow!
  - Chunk of storage are distributed all over the disk.
  - Disk storage may fail.
- What new **capabilities** do files add?
  - Growth and shrinking.
  - Organization into directories.
- What **information** files help to organize?
  - Ownership and permission.
  - Access time, modification time, type etc.

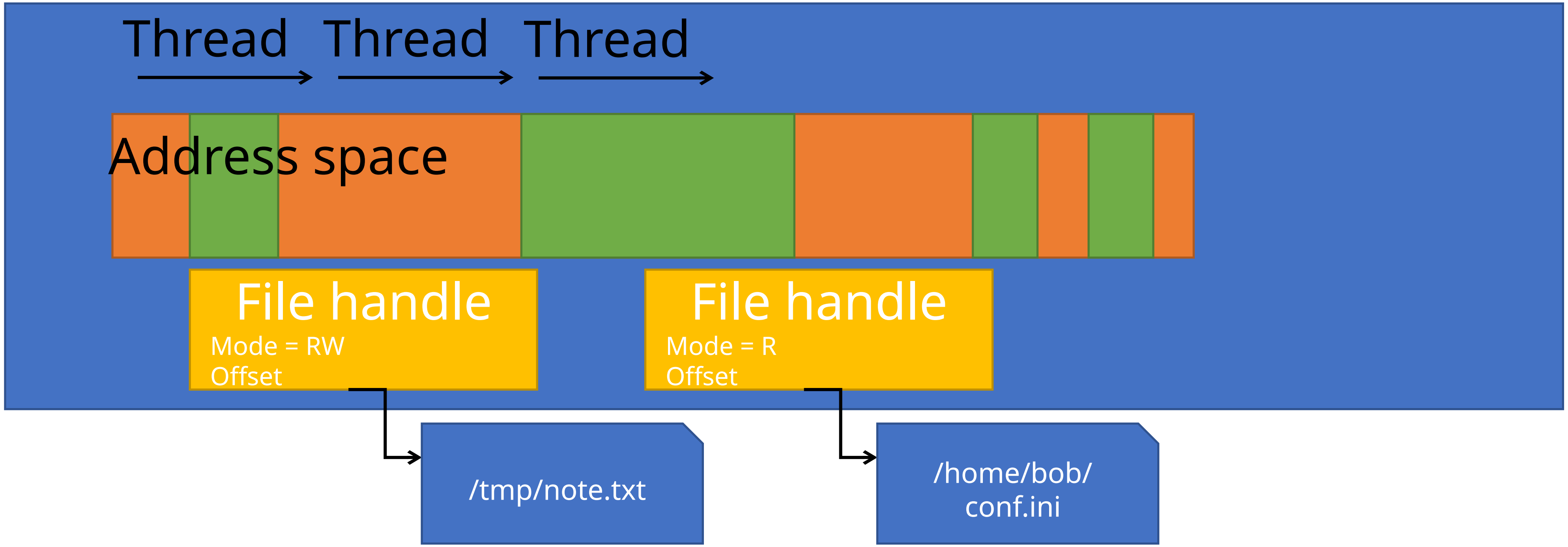
The process abstraction

Processes are the most fundamental abstraction

- What the computer “is doing”.
- Help organize other abstractions.
- You know processes as “applications”.

The process abstraction

- Processes are **not tied to a hardware component**.
- They contain and organize other abstractions.



Processes vs Threads

- Potentially confusing due to terminology
  - both described as **running**
- Some terminology useful to remember the distinction
  - Processes require multiple resources: CPU, memory, files
  - Threads abstract the CPU
- A process contains threads, threads belong to a process
  - Except kernel threads who do not belong to a user space process
- A process is running when one or more of its threads are running

Process Example: Firefox

- Firefox has multiple threads. What do they do?
  - Waiting and processing interface events (e.g., mouse click)
  - Redrawing the screen as necessary (responding to user inputs)
  - Loading web pages (generally multiple elements in parallel)
- Firefox is using memory. For what?
  - The executable code itself
  - Shared library: web page parsing, TLS/SSL etc.
  - Stacks storing local variables for running threads
  - Storing dynamically allocated memory
- Firefox has files open. Why?
  - Fonts
  - Configuration files