Operating System

Yuqiao Meng

2021–9–124

Contents

1 Overview

1.1 What?

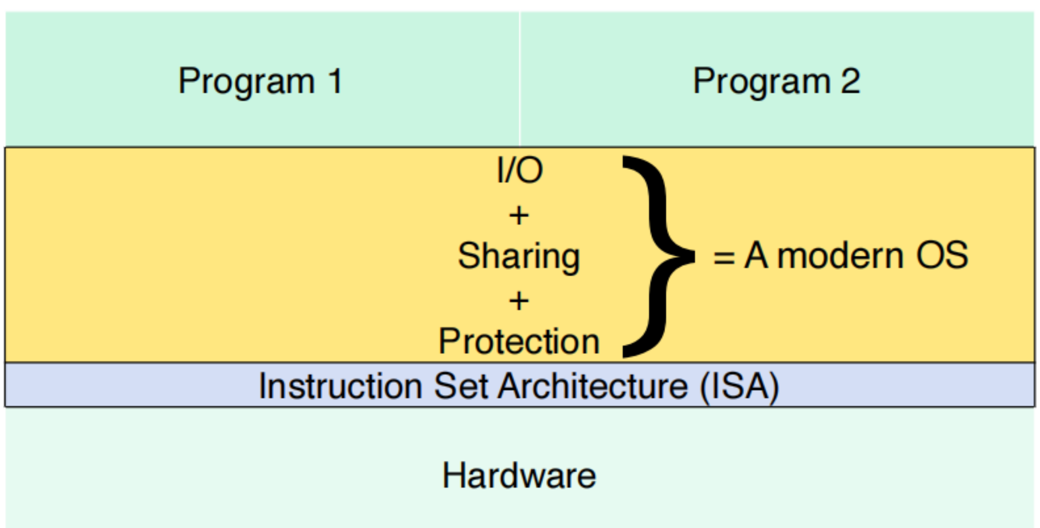
What is an Operating System? What's its reponisibility?

- A bunch of software and data residing somewhere in memory.
- The most privileged software in a computer. It can do special things, like write to disk, talk over the network, control memory and CPU usage, etc
- Manages all system resources, including CPU, Memory, and I/O devices.

1.2 Why?

Why do we need an OS?

- OS helps program to control hardwares.
- OS determines the way programs share resources.
- OS protects hardwares and programs from getting attacked.
- OS stores files persistently.



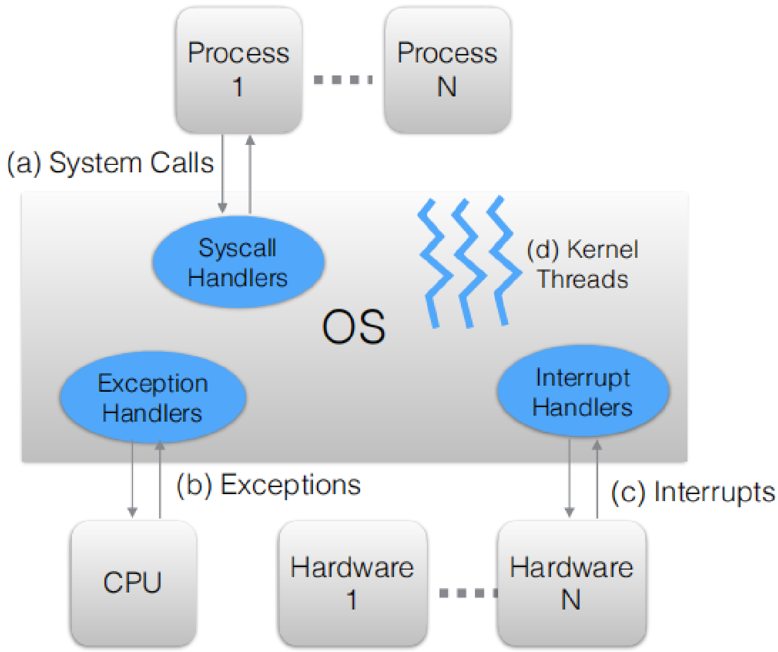
1.3 How

1.3.1 Virtulization

- Definition: OS takes a physical resource (such a sthe processor, or memory, or a disk) and transforms it into a more general, powerful, and easy-to-use virtual form of itself.
- Resource Virtulization
  - Many(virtual)-to-one(physical): CPU Virtulization
  - One-to-many: Disk Virtulization
  - Many-to-many

1.3.2 How to invoke OS code?

- System calls: Function calls into the OS, that OS provides these calls to run programs, access memory and devices, and other related actions.
- Exceptions: CPU will raise an exception to the OS when the running program do something wrong
- Interrupts: Hardwares send interrupts to invoke OS
- Kernel Threads: Programs run in the kernel context, executing kernel level functions.

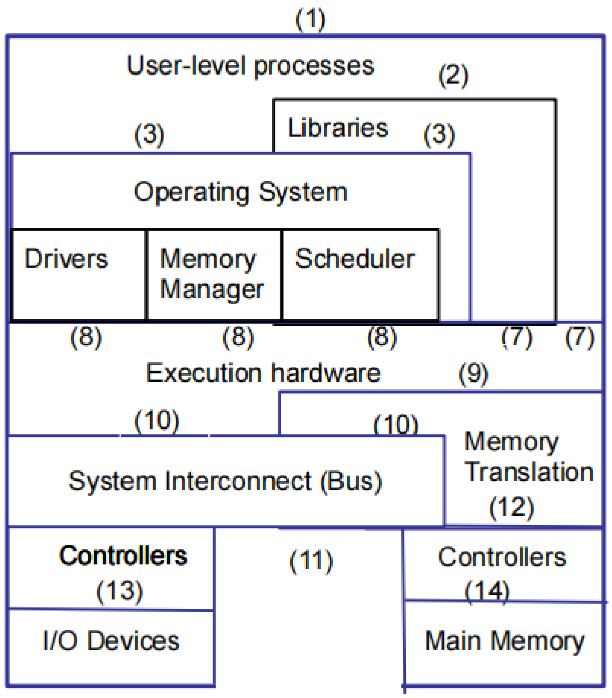


1.4 Interface

1.4.1 Explaination

- Instruction Set Architecture(ISA): the language CPU understand
- User ISA: ISA that any program can execute, it's accessible for all programs, doesn't need the service of operating system
- System ISA: ISA that only operating system is allowed to execute.
- Application Binary Interface(ABI): the combination of syscalls and User ISA(3, 7), it's the view of the world, seen by programs.
- Application Programmers' Interface(API): the combination of libraries and User ISA(2, 7), it's the tools programmer use to write codes.

1.4.2 Interfaces in a Computer System



- User ISA: 7
- System ISA: 8
- Syscalls: 3
- Application Binary Interface: 3, 7
- Application Programmers' Interface: 2, 7

1.5 History

- First Computer: Atanasoff–Berry computer, or ABC.
- First OS: GM-NAA I/O, produced in 1956 by General Motors' Research division for its IBM 704.
- First language: Plankalkül, developed by Konrad Zuse for the Z3 between 1943 and 1945.

- First programmer: Ada Lovelace

2 Processes

2.1 Process

2.1.1 What?

What is a process?

- A process is a program in execution. A program is a set of instructions somewhere (like the disk).
- Once created, a process continuously does the following:
  - **Fetches** an instruction from memory.
  - **Decodes** it. i.e., figures out which instruction this is.
  - **Executes** it. it does the thing that it is supposed to do, like add two numbers together, access memory, check a condition, jump to a function, and so forth.

2.1.2 Process versus Program

How is a process different from a program?

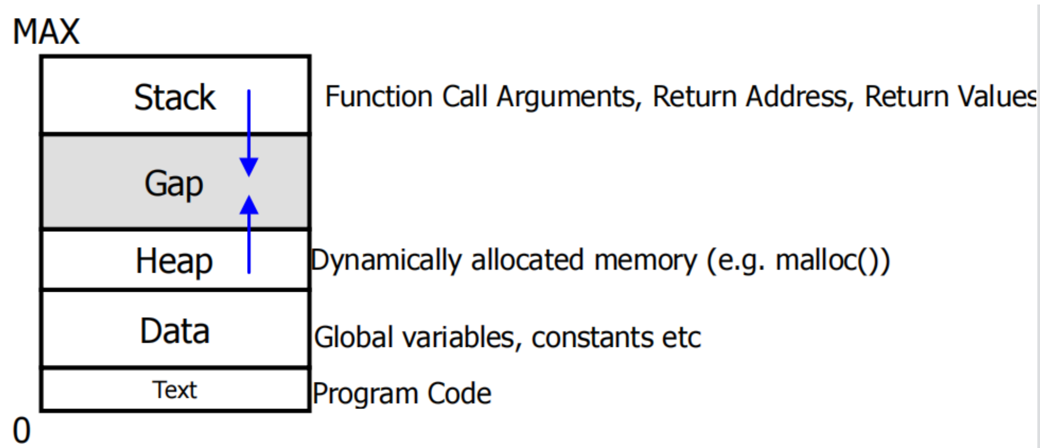
- Program: A passive entity stored in the disk, has static code and static data.
- Process: Actively executing code and the associated static and dynamic data.
- Program is just one component of a process.
- There can be multiple process instances of the same program

2.1.3 Constitution

- Memory space
- Procedure call stack

- Registers and counters
- Open files, connections
- And more.

2.1.4 Memory layout



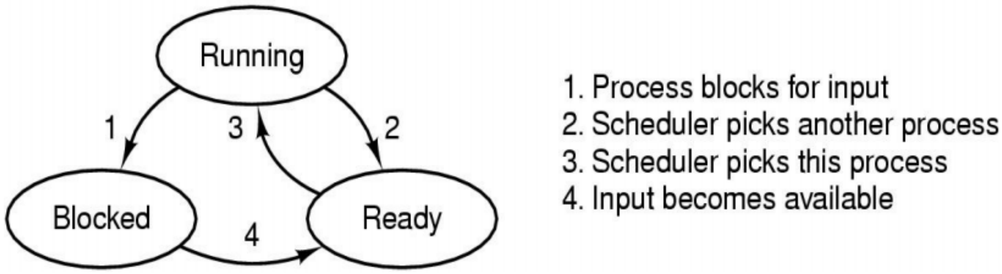
In this picture, Stack and Heap grow toward each other, that's because every process has a limited amount of space, thus let heap and stack grow toward each other from two direction can make the best use of space.

2.2 System calls

- fork(): create new process. **called once but return twice**. Usage:
  - User runs a program at command line
  - OS creates a process to provide a service: Check the directory /etc/init.d/ on Linux for scripts that start off different services at boot time.
  - One process starts another process: For example in servers
- exec(): execute a file. **replaces the process' memory with a new program image. All I/O descriptors open before exec stay open after exec**.
- wait()/waitpid(): wait for child process.

- exit(): terminate a process

2.3 Lifecycle



- Ready (runnable; temporarily stopped to let another process run)
  - Process is ready to execute, but not yet executing
  - Its waiting in the scheduling queue for the CPU scheduler to pick it up.
- Running: (actually using the CPU at that instant)
- Blocked (unable to run until some external event happens).
  - Process is waiting (sleeping) for some event to occur.
  - Once the event occurs, process will be woken up, and placed on the scheduling queue

1. Running → Blocked: Occurs when the operating system discovers that a process cannot continue right now.
2. Running → Ready: Occurs when the scheduler decides that the running process has run long enough and it is time to let another process have some CPU time.
3. Ready → Running: Occurs when all the other processes have had their fair share and it is time for the first process to get the CPU to run again.
4. Blocked → Ready: Occurs when the external event for which a process was waiting (such as the arrival of some input) happens

2.4 Special Process

- Orphan process
  - When a parent process dies, child process becomes an orphan process
  - The init process (pid = 1) becomes the parent of the orphan processes
- Zombie process
  - When a child dies, a SIGCHLD signal is sent to the OS, If parent doesn't wait()on the child, and child exit()s, it becomes a zombie.
  - Zombies hang around till parent calls wait() or waitpid().
  - Zombies take up no system resources, it's just a integer status kept in the OS.
  - Ways to prevent a child process from becoming a zombie:
    * Parent call wait()/waitpid() before child process exit()
    * Child parent sleep() before exit() until parent process give it a message.

2.5 Cold-start Penalty

2.6 Context Switch

3 Inter-Process Communication

3.1 Overview

Inter-Process Communication mechanisms

- Pipe:
- Signals: Event notification from one process to another
- Shared momery: Common piece of read/write memory, needs authorization for access

- Parent-child: Command-line arguments, including waitpid(), wait(), exit()
- Reading/modifying common files
- Semaphores: Locking and event signaling mechanism between processes
- Sockets: Not just across the network, but also between processes

3.2 Pipe