Introduction to Operating Systems

What are Operating Systems?

* Middleware between user programs and hardware
* Since, user programs run on hardware, operating systems also manages hardware: CPU, main-memory, I/O devices (disc, network-card, mouse, keyboard etc.)

What happens when you run a program?

* A compiler translates high-level programs into an executable (“.c” to “a.out”)
* The executable contains instructions that the CPU can understand, and data of the program (all numbered with addresses)
* These instructions are specific to underlying CPU hardware. Every CPU has a certain instruction set of architecture.

What does CPU do?

* CPU fetches instruction pointed by PC from memory.
* Loads data required by the instructions to the registers.
* Decodes and executes the instruction.
* Stores the result into memory.

Most recently used instructions and data are in CPU caches for faster access.

What does the Operating Systems do?

* OS manages program memory – loads program executable (code, data) from disc to main memory.
* OS manages CPU – initializes program counter (PC) and other registers to begin execution. Examples include:
  + PC points towards current instruction
  + Stack pointer points towards current position in the stack.
  + File descriptors points to open files and devices.
* OS manages external devices; read/write files from the disk.

1. OS manages CPU

* OS provides the process abstraction

Process in the context of OS, means the instance of a running program.

* OS creates and manages processes
* The role of OS is to create an “illusion” that a process has all it needs to be executed.
* Timeshares CPU between processes.
* Coordinates between processes.
* OS manages the memory of the process: code data, stack, heap etc.
* Each process thinks it has a dedicated memory space for itself, numbers code and data starting from 0 (using virtual addresses).
* OS abstracts out the details of the actual placement in memory, translates from virtual addresses to actual physical addresses.

2.) OS manages devices

* OS has code to manage disk, network card, and other external devices via device drivers.
* Device drivers are a piece of a kernel code that talks the language of the hardware devices
* Issues files instructions to devices (fetch data from a file)
* Responds to interrupt events from devices (user has pressed a key on a keyboard.)

3.) OS manages file-system

It means that OS ensures that there is a persistent data organized as a file-system on a disk.

4.) OS loads at start-up

* Bootstrap program is loaded at power-up or reboot
* It is typically stored in ROM or EPROM, generally known as firmware; it initializes all aspects of system, loads operating system kernel, and starts execution.

Design goals for an operating system

* Convenience, abstraction of hardware resources for user programs.
* Efficiency of usage of CPU, memory, etc.
* Isolation between multiple processes.

Process abstraction of Operating Systems

* When you run an executable file, the OS creates a process (i.e., a running program)
* OS timeshares CPU across multiple processes, *virtualizes* CPU;

Gives an illusion that each process is running on the CPU alone, whereas in the background multiple resources are receiving and sharing the CPU.

In other words, an OS creates an “illusion” that a process has all it needs to be executed.

* OS has a CPU scheduler that picks one of the many active processes to execute on a CPU.

Policy: which process to run

Mechanism: how to “context switch” between processes.

What constitutes a process?

* Every process that is created has a unique identifier called process-id (PID)
* A process has real or virtual memory
* Threads – a lightweight process, a process may have multiple threads which shares the same system resources that can assist in faster creation, termination, switching and communication.
* A process has some memory footprint that is in the RAM, (i.e., a memory image)
* Inside a memory image, there is

1. code & data (static) (code means compiled instructions)

* Code means binary instructions to be executed.
* Usually read-only.
* Global/constant/static variables are shared between threads

1. stack & heap (dynamic)

* Arguments and function/procedure calls are put into stack.
* Stack has stack pointer, and it is FILO (first-in last-out)
* Heap dynamically allocates memory at run-time.

How does OS create a process?

Allocates and initialises the memory for PCB and creates memory-image (code, data, stack & heap)

* + Loads code and data from the executable
* Creates stack and heap on runtime as they are dynamic.

Opens basic files

* STD IN, OUT, ERR

Initializes CPU registers, PC points to first instruction.

Link the PCB to process queue.

States of a process

A process exists in different states.

* Running: currently executing on CPU
* Ready: waiting to be scheduled *(Ready to run, but they have not yet run.)*
* Blocked: suspended, not ready to run

Why blocked? Waiting for some event, process wants to read a file from a disk, it has issued a request to a disk for example.

When is it unblocked?

When the disk issues an interrupt that the data is ready.

* New: being created, yet to run.
* Dead: terminated.

Process state transition

A process moves from one state to another as a result of OS scheduling, external interrupts, OR requests from other programs.

Diagram

Description automatically generated

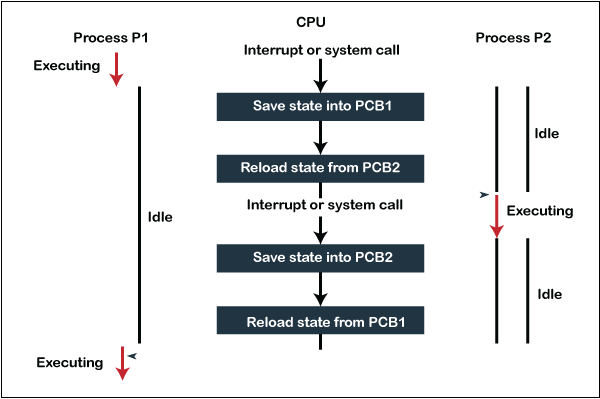
* Admit: process is fully loaded into memory and control is established
* Dispatch: scheduler assigns CPU to the process
* Time out: expired or pre-empted, pushed back to the queue
* Event Wait/Event Occurs: generally, requests that cannot be met at the moment, has to wait until something occurs
  + OS not ready for a service
  + Unavailable resource
  + Waiting for an input
* Release: release resources and end the process

OS Data structures

* OS maintains a data-structure (e.g., list) of all active processes.
* Information about each process is stored in a process control block (PCB) which contains:
  + Process identifier
  + Process state
  + PC, SP, and other registers (stored)
  + Pointers to other related processes (parent)
  + CPU context of the process (saved when the process is suspended)
  + Scheduling information (how the running of a process is prioritised)
  + Memory management information
  + Accounting information
  + User information
  + Inter-process communication (IPC)
  + Pointers to the memory locations
  + Pointers to open the files

PCB is stored in memory and its location pointed to by something called the process ID pointer.

Context switching



* Context switching happens whenever a process leaves or enters a running state.
* Processes may make a transition voluntarily or involuntarily e.g., end the program vs interrupt.
* OS typically maintains queue or queue-like (list) structures for the processes in the same states (many pointers in the PCB)

Diagram

Description automatically generated

PCB makes context switching a bit easier.

* Scheduler will start or stop a process accordingly.
* Stores necessary information in the PCB to stop.
* Similarly, loads back necessary information from the PCB on context switch back.

Context Switching limitations

* It still consumes some time – could be up to several thousand CPU clock cycles
* Overhead and bottleneck
* Hardware support is also needed