CS 347: Operating Systems

Amit Rajaraman

Last updated July 27, 2021

§1. Introduction

1.1. What makes a good operating system?

An **operating system** is the middleware between user programs and system hardware. It manages hardware such as the CPU, main memory, I/O devices, etc.
When one runs an OS, a *compiler* translates high-level programs to an executable. This executable contains instructions that the CPU can understand, and data of the program (numbered using addresses). These instructions run on a CPU – the hardware implements an *instruction set architecture* (ISA). The CPU also has a few registers, a pointer to the current instruction (known as the **program counter**), operands of instructions, and memory addresses.
When we run an executable, the CPU

- *fetches* the instruction pointed at by the PC from memory,
- loads the data required by the instructions,
- *decodes* and *executes* the instruction, and
- stores the result back to memory.

The above is known as the *Von Neumann* model of computing.
Further, recently used instructions and data are stored in an instruction cache and data cache for fast access.

Where does the OS fit into this picture? It

- manages program memory by loading the program executable from disk to the memory.
- manages the CPU by initializing the PC and other registers to ready for execution.
- manages external devices such as reading/writing files from/to the disk.

Due to this, an OS is also sometimes known as a *resource manager*.

The OS provides a *process abstraction* – it creates and manages processes (a running program). Each process is under the illusion of having access to the complete CPU, since the OS *virtualizes* the CPU. It also timeshares the CPU between and coordinates multiple processes.

The OS manages the memory of the process, including code (compiled instructions written by the user), data (variables created by the user), stack (storage of arguments/return addresses from function calls), heap (`mallocs` for example), etc.
Individual processes think they have a dedicated memory space for themselves, with numbers, code, and data starting from 0 (**virtual addresses**). The OS abstracts out details of the actual placement in memory, translating between virtual addresses (which the process sees) and physical addresses (actual address in the hardware).

The OS also has code (**device drivers**) to communicate with hardware devices such as disk and network card. They issue instructions to devices (fetching data from a file, for example) and respond to interrupt events from devices (when the user has pressed a key on a keyboard, for example). Persistent data is organized as a *file system* on disk.

What do we want when designing an OS?

- Convenience via abstraction of hardware resources for user programs.
- Efficiency of usage of CPU, memory, etc.
- Isolation of multiple processes (so one program cannot interfere with another).

That is, we want the virtualization to be effective and secure.
Operating systems started out as just a library to provide common functionality across programs. Later, we evolved from procedure calls to system calls – these are called when we want to run OS code and are executed at a higher privilege level. These interfaces between the OS and the user are called *APIs*. It also evolved from running a single program to multiple processes concurrently.

1.2. Process abstractions

When we run an executable, the OS creates a process, a running program, virtualizing the CPU and timesharing it across processes. The OS has a *CPU scheduler* that picks one of the several active processes to execute, which has two parts: the **policy** that decides which process to run and the **mechanism** which knows how to "context switch" between processes and ensures that the process chosen to run is running.

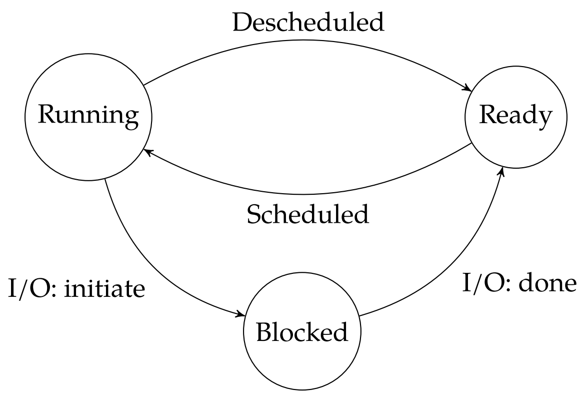
Any process has a unique identifier, known as the *process ID* (PID).
It also has a certain memory footprint or *memory image*, consisting of the code and data, which are static, and the stack and heap, which are dynamic. The stack is used for function calls and function arguments, while the heap is used for dynamic memory allocation (`malloc`s in C for instance).
It has a *CPU context*, which consists of the registers used in the process – the program counter (which points to the current position in the code), current operands (which looks at the data), and stack pointer (which points to the current position in the stack).
Finally, there are file descriptors that are pointers to open files/devices. `stdout` and `stdin` are examples of this.

When the OS creates a process,

- it allocates memory and creates a memory image. This loads code and data from a disk executable, and creates the runtime stack and heap.
- it opens some basic files for the program such as `stdin`, `stdout`, and `stderr` for example.
- it initializes various registers such as the PC which should point to the first instruction.

The primary states that any process is in are:

- *running* – currently executing on the CPU.
- *ready* – waiting to be scheduled.
- *blocked* – suspended and not ready to run. This may be because they are waiting for some event (such as a read from a disk). They can eventually be unblocked when the event is done (when the disk issues an interrupt, for example).
- *new* – being created and is yet to run.
- *dead* – terminated processes.



The OS maintains a data structure (a list) of all active processes. Information about each process is stored in a **process control block** (PCB) which has the

- process identifier,
- process state,
- pointers to other related processes (parent for example),
- CPU context of the process (which is saved when the process is suspended),
- pointers to memory locations, and
- pointers to open files.