Operating Systems

Author: Guanzhou (Jose) Hu 胡冠洲 @ UW-Madison

This note is a reading note of the book: Operating Systems: Three Easy Pieces (OSTEP) by Prof. Remzi Arpaci-Dusseau and Prof. Andrea Arpaci-Dusseau. Figures included in this note are from this book unless otherwise stated.

Introduction to OS

An **Operating System** (OS) is a body of software sitting in between *software applications* and a *Von Neumann computer architecture*. An OS connects applications to physical hardware. It makes **abstractions** of the underlying hardware and provides an easy-to-use *interface* for running portable software on physical hardware.

An OS does this through three general techniques:

1. **Virtualization**: taking a physical resource (processor, memory, storage, ...) and transforms it into a more general, portable, and easy-to-use virtual interface for user applications to use
2. **Concurrency**: acting as a resource manager which supports multiple user applications to run concurrently and coordinates among running applications, ensuring correct, fair, and efficient sharing of resources
3. **Persistence**: data can be easily lost on volatile devices such as DRAM. An OS allows users to talk to external devices - including persistent storage drives - through *Input/Output* (I/O)

> Abstraction is a great idea in both computer architecture and operating systems. It hides implementation complexity about the underlying layer and exposes a unified model of how to use the underlying layer to the upper layer. Check out the first section of this note.

A modern operating system also pursues some other goals apart from the above three. These are:

- *Performance*: minimize the overheads brought by the OS
- *Security*: protect against bad behavior; provide *isolation*
- *Reliability*: properly recover on fail-stops
- *Connectivity*: networking support; connect with the Internet
- *Energy-efficiency*, *Mobility*, ...

Virtualizing the CPU: Processes & Scheduling

One of the most important abstractions an OS provides is the **process**: a running instance of a program. We typically want to run many processes at the same time (e.g., a desktop environment, several browser windows, a music player, and a task monitor), more than the number of available CPU cores (and probably other physical resources as well). The OS must be able to *virtualize* a physical resource and let multiple processes share the limited resource. This section focuses on sharing the CPU, which is the most fundamental resource required to kick off any process.

Abstraction of Process

A process is simply an *instance* running on a processor (the dynamic instance, doing actual work) of a piece of *program* (the static code + data, residing on persistent storage). There can be multiple processes of the same piece of program code.

Machine State

Running a process instance of a program requires the OS to remember the *machine state* of the process, which typically consists of the following information:

- **Address space**: memory space that a process can address, typically also virtualized, which contains at least:
    - *Code*: compiled machine code of the program
    - *Data*: any initial static data/space the program needs
    - *Stack*: space reserved for the run-time function *stack* of the process
    - *Heap*: space reserved for any new run-time data
- **Registers context**: CPU registers' values; particularly special ones include:
    - *Program counter* (PC, or *instruction pointer*): which instruction of the program to execute next
    - *Stack pointer* (SP) & *Frame pointer* (FP): for managing the function stack
- **I/O information**: states related to storage or network, for example:
    - List of currently open files (say in the form of *file descriptors*)

Process Status

A process can be in one of the following states at any given time:

- (optional) *Initial*: being created and hasn't finished initialization yet
- *Ready*: is ready to be scheduled onto a CPU to run, but not scheduled at this moment
- *Running*: scheduled on a CPU and executing instructions
- *Blocked*: waiting for some event to happen, e.g., waiting for disk I/O completion or waiting for another process to finish, hence not ready to be scheduled at this moment
- (optional) *Terminated*: has exited/been killed but its information data structures have not been cleaned up yet

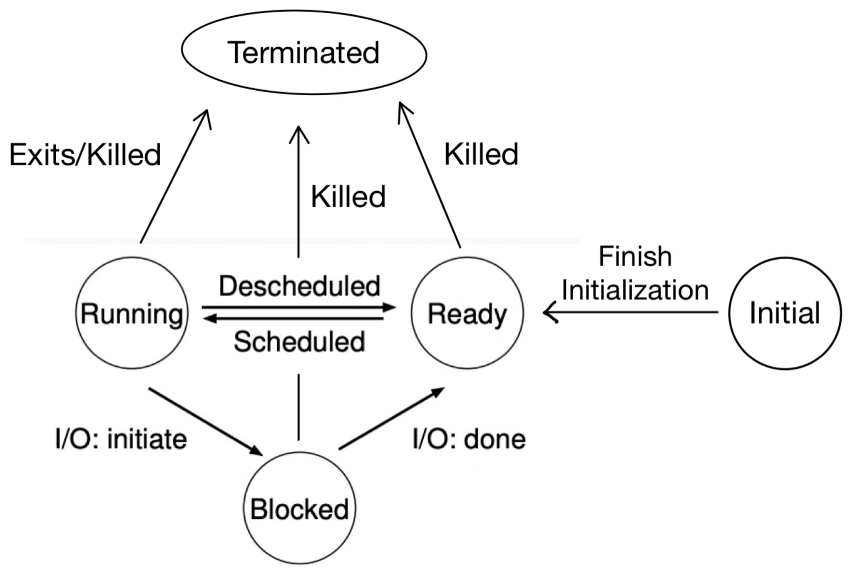

Figure 4.2: **Process: State Transitions** (extended)

Process Data Structures

The OS must have some data structures to hold the information of processes. We call the metadata structure of a process a **process control block** (PCB, or *process descriptor*). This structure must include the machine state of the process, the status of the process, and any other necessary information related to the process. For example, the xv6 OS has the following PCB struct:

```c
struct context {
    int eip;
    int esp;
    int ebx;
    int ecx;
    int edx;
    int esi;
    int edi;
    int ebp;
};

enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

/** The PCB structure. */
struct proc {
    char *mem;                  // Start of process memory
    uint sz;                    // Size of process memory
    char *kstack;               // Bottom of kernel stack
    enum proc_state state;      // Process state
    int pid;                    // Process ID
    struct proc *parent;        // Parent process
    void *chan;                 // If !zero, sleeping on chan
    int killed;                 // If !zero, has been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;          // Current directory
    struct context context;     // Register values context
    struct trapframe *tf;       // Trap frame of current interrupt
};
```

The collection of PCBs is the **process list** (or *task list*), the first essential data structure we meet in an OS. It can be implemented as just a fixed-sized array of PCB slots as in xv6, but can also take other forms such as a linked list or a hash table.

Process APIs

These interfaces are available on any modern OS to enable processes:

- `create` - *initialize* the above mentioned states, *load* the program from persistent storage in some *executable format*, and get the process running at the entry point of its code; the process then runs until completion and exits (, or possibly runs indefinitely)

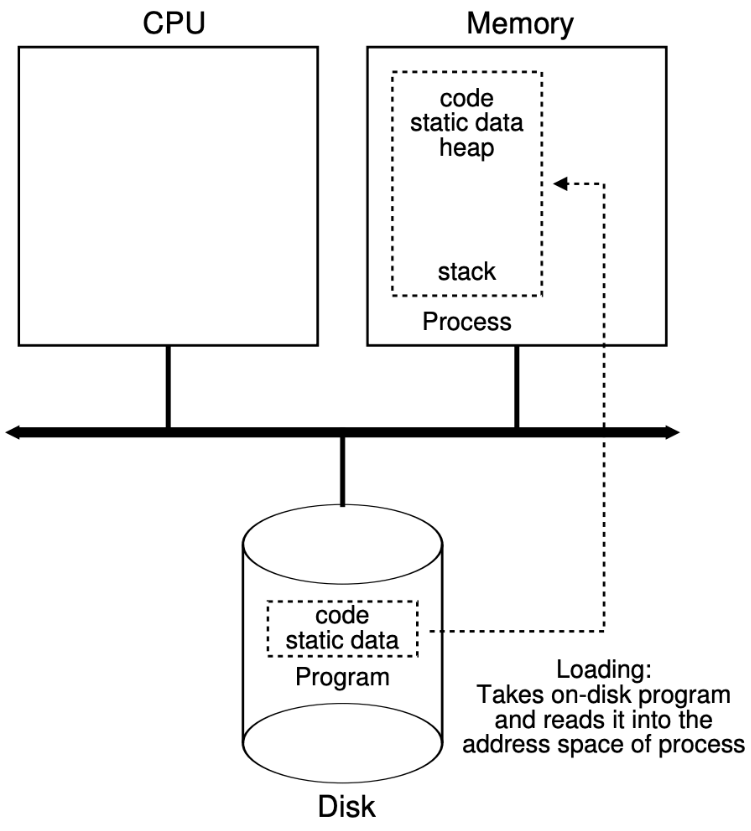

Figure 4.1: **Loading: From Program To Process**

- `destroy` - forcefully destroy (kill, halt) a process in the middle of its execution
- `wait` - let a process wait for the termination of another process
- `status` - retrieve the status information of a process
- other control signals like suspending & resuming, ...

There is typically a special `init` process created by the OS at the end of the booting process, and other user processes, e.g., the command-line shell and the desktop GUI, are created by the `init` processes through `fork` + `exec`.

TODO parent process, fork, exec

Time-Sharing of the CPU

TODO context switch

Scheduling Policies

TODO

Virtualizing the Memory: Memory Management

TODO

Concurrency: Multi-Tasking & Synchronization

TODO

Persistence: Storage Devices & File System

TODO

Advanced Topics

TODO a list of notes, blogs, and books

Implementation

> I've made a brief OS history tree in XMind which is available HERE.