Part 1 Virtualization

✓ Virtualization

Assume there is one physical CPU in a system (though now there are often two or four or more). What virtualization does is take that single CPU and make it look like many virtual CPUs to the applications running on the system. Thus, while each application thinks it has its own CPU to use, there is really only one. And thus the OS has created a beautiful illusion: it has virtualized the CPU.

4 The Abstraction : A Process

4.1 Process

✓ Process : a running program

(The abstraction provided by the OS of a running program is called a process.)

✓ Virtualizing the CPU

By running one process, then stopping it and running another, and so forth, the OS can promote the illusion that many virtual CPUs exist when in fact there is only one physical CPU (or a few).

This basic technique, known as time sharing of the CPU, allows users to run as many

concurrent processes as they would like; the potential cost is performance, as each will run more slowly if the CPU(s) must be shared.

✓ Time Sharing

Time sharing is a basic technique used by an OS to share a resource. By allowing the resources to be used for a little while by one entity, and then a little while by another, and so forth, the resource in question can be shared by many. It is space sharing, where a resource is divided among those who wish to use it.

✓ Low - level machinery : mechanisms.

- Mechanisms are low - level methods or protocols that implement a needed piece of functionality.
- Example :

  Context Switch gives the OS the ability to stop running one program and start running another on a given CPU.

✓ High - level intelligence : policies.

- Policies are algorithms for making some kind of decision within the OS.
- Example:

  A scheduling policy in the OS will make this decision, likely using historical

information, workload knowledge, and performance metrics to make its decision.

✓ Separate Policy and Mechanism

- Mechanism provides the answer to a how question about a system.

  For example, how does an operating system perform a context switch?

- The policy provides the answer to a which question.

  For example, which process should the operating system run right now?

- Separating the two allows one easily to change policies without having to rethink the mechanism and is thus a form of modularity , general software design principle.

✓ Machine state : what a program can read or update when it is running.

(At any given time, what parts of the machine are important to the execution of this program ?)

- Memory (Address Space)
  - Instructions lie in memory.
  - The data that running program reads and writes sits in memory.
- Registers :
  - Many instructions explicitly read or update registers and thus clearly they are important to the execution of the process

- Special Registers :
  - the program counter (PC) or the instruction pointer (IP) - tells us which instruction of the program will execute next;
  - the stack pointer and the associated frame pointer - are used to manage the stack for function parameters, local variables, and return addresses.

4.2 Process API

✓ Create

An operating system must include some method to create new process. When you type a command into the shell, or double-click on an application icon, the OS is invoked to create a new process to run the program you have indicated.

✓ Destroy

As there is an interface for process creation, systems also provide an interface to destroy processes forcefully. Of course, many processes will run and just exit by themselves when complete;when they don't, however, the user may wish to kill them, and thus an interface to halt a runaway process is quite useful.

✓ Wait

Sometimes it is useful to wait for a process to stop running; thus some kind of waiting

interface is often provided.

✓ Miscellaneous Control

Other than killing or waiting for a process, there are sometimes other controls that are possible.

For example, most operating systems provide some kind of method to suspend a process (stop it from running for a while) and then resume it (continue it running).

✓ Status

There are usually interfaces to get some status information about a process as well, such as how long it has run for, or what state it is in.

4.3 Process Creation

✓ Question

How programs are transformed into processes?

How does the OS get a program up and running?

How does process creation actually work?

✓ Answer

The OS will do under four operations.

- 1st. Loading : Takes on-disk program and reads it into the address space of process.

Before running anything, the OS clearly must do some work to get the important program bits from disk into memory.

So the first thing that the OS must do to run a program is to load its code and any static data (e.g., initialized variables) into memory, into the address space of the process.

Programs initially reside on disk(or, in some modern systems, flash-based SSDs) in some kind of executable format; thus, the process of loading a program and static data into memory requires the OS to read those bytes from disk and place them in memory somewhere (as shown in Figure 4.1).

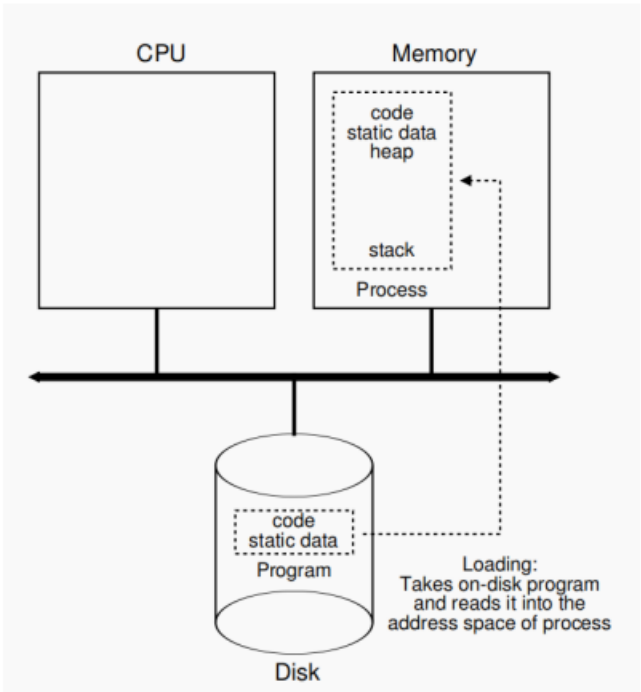

Figure 4.1: **Loading: From Program To Process**

[In early (or simple) operating systems, the loading process is done **eagerly**,(i.e., all at once before running the program; modern OSes perform the process **lazily**,(i.e., by loading pieces of code or data only as they are needed during program execution.]

❖ 2nd. Memory Allocation for program's stack and heap :

Once the code and static data are loaded into memory, there are a few other things the OS needs to do before running the process.

Some memory must be allocated for the program's **run-time stack** (or just **stack**).

For example, in C programs, it uses the stack for local variables, function parameters, and return addresses; the OS allocates this memory and gives it to the process. The OS will also likely initialize the stack with arguments; specifically, it will fill in the parameters to the main() function, i.e., argc and the argv array.

The OS may also allocate some memory for the program's heap.

For example, in C programs, the heap is used for explicitly requested dynamically-allocated data; programs request such space by calling malloc() and free it explicitly by calling free(). The heap is needed for data structures such as linked lists, hash tables, trees, and other interesting data structures. The heap will be small

at first; as the program runs, and requests more memory via the malloc() library API, the OS may get involved and allocate more memory to the process to help satisfy such calls.

❖ 3rd. Initialization tasks , particularly as related to input/output(I/O) :

For example, in UNIX systems, each process by default has three open file descriptors, for standard input, output, and error; these descriptors let programs easily read input from the terminal and print output to the screen.

[I/O, File Descriptors will learn in the third part of the book on persistence.]

❖ 4th. Start the program running at the entry point, namely main(). :

By loading the code and static data into memory, by creating and initializing a stack, and by doing other work as related to I/O setup, the OS has now (finally) set the stage for program execution.

It thus has one last task: to start the program running at the entry point, namely main().

By jumping to the main() routine (through a specialized mechanism that we will discuss next chapter), the OS transfers control of the CPU to the newly-created process, and thus the program begins its execution.

4.4 Process States

- Running :

In the running state, a process is running on a processor. This means it is executing instructions.

- Ready :

In the ready state, a process is ready to run but for some reason the OS has chosen not to run it at this given moment.

- Blocked :

In the blocked state, a process has performed some kind of operation that makes it not ready to run until some other event takes place.

A common example : when a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.

- Process State Transitions :

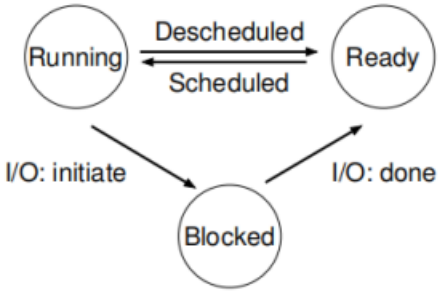

Figure 4.2: **Process: State Transitions**

Map these states to a graph, then arrive at the diagram in Figure 4.2.A process can be moved between the ready and running states at the discretion of the OS.

Being moved from **ready to running** means the process has been **scheduled**; being moved from **running to ready** means the process has been **descheduled** .

Once a process has become **blocked** (e.g., by initiating an I/O operation), the OS will **keep it as such until some event occurs** (e.g., I/O completion); at that point, the process moves **to the ready state again** (and potentially immediately to running again, if the OS so decides).

❖ An example of how two processes might transition through some of these states :

First example, imagine two processes running, each of which only use the CPU (they do no I/O). In this case, a trace of the state of each process might look like this Figure 4.3.

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	$Process_0$ now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	$Process_1$ now done

Figure 4.3: **Tracing Process State: CPU Only**

Second example, the first process issues an I/O after running for some time. At that point, the process is blocked, giving the other process a chance to run. Figure 4.4 shows a trace of this scenario.

Time	Process_0	Process_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process_0 initiates I/O
4	Blocked	Running	Process_0 is blocked,
5	Blocked	Running	so Process_1 runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process_1 now done
9	Running	–	
10	Running	–	Process_0 now done

Figure 4.4: **Tracing Process State: CPU and I/O**

More specifically, Process0 initiates an I/O and becomes blocked waiting for it to complete; When reading from a disk or waiting for a packet from a network, processes will become blocked. So the OS recognizes Process0 is not using the CPU and starts running Process1. While Process1 is running, the I/O completes, moving Process0 back to ready. Finally, Process1 finishes, and Process0 runs and then is done.

Note that there are many decisions the OS must make, even in this simple example. First, the system had to decide to run Process1 while Process0 issued an I/0; doing so improves resource utilization by keeping the CPU busy. Second, the system decide not to switch back to Process0 when its I/O completed; it is not clear if this is a good decision or not.

These types of decisions are made by the OS scheduler, a topic will discuss a few chapters in the future.

4.5 Data Structures

The OS is a program, and like any program, it has some key data structures that track various relevant pieces of information.

To track the state of each process, for example, the OS likely will keep some kind of **process list** for all processes that are ready and some additional information to track which process is currently running.

The OS must also track, in some way, blocked processes; when an I/O event completes, the OS should make sure to wake the correct process and ready it to run again

[Figure 4.5 shows what type of information an OS needs to track about each process in the xv6 kernel [CK+08]. Similar process structures exist in "real" operating systems such as Linux, Mac OS X, or Windows; look them up and see how much more complex they are.]

From the figure, you can see a couple of important pieces of information the OS tracks about a process.

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  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;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

Figure 4.5: **The xv6 Proc Structure**

✓ The **register context** will hold, for a stopped process, the contents of its registers. When a process is stopped,its registers will be saved to this memory location; by restoring these registers (i.e., placing their values back into the actual physical registers), the OS can resume running the process. [We' ll learn more about this technique known as a context switch in future chapters.]

- ✓ There are some other states a process can be in, beyond running, ready, and blocked. Sometimes a system will have an **initial** state that the process is in when it is being created.

Also, a process could be placed in a **final** state where it has exited but has not yet been cleaned up (in UNIX-based systems, this is called **the zombie state : just like real zombies, these zombies are relatively easy to kill**). This final state can be useful as it allows other processes (usually the parent that created the process) to examine the return code of the process and see if the just-finished process executed successfully (usually, programs return zero in UNIX-based systems when they have accomplished a task successfully, and non-zero otherwise).

When finished, the parent will make one final call (e.g., wait()) to wait for the completion of the child, and to also indicate to the OS that it can clean up any relevant data structures that referred to the now-extinct process.

- ❖ The Process List (Task List) :

  It is one of the simple ones, but certainly any OS that has the ability to run multiple programs at once will have something akin to this structure in order to keep track

of all the running programs in the system. Sometimes people refer to the individual structure that stores information about a process as a Process Control Block(PCB), a fancy way of talking about a C structure that contains information about each process (also sometimes called a process descriptor).

4.6 Summary

The process is the most basic abstraction of the OS. It is quite simply viewed as a running program. With this conceptual view in mind, we will now move on to the nitty-gritty : the low-level mechanisms needed to implement processes, and the higher-level policies required to schedule them in an intelligent way. By combining mechanisms and policies, we will build up our understanding of how an operating system virtualizes the CPU.

4.7 Key Process Terms

- The process is the major OS abstraction of a running program. At any point in time, the process can be described by its state: the contents of memory in its address space, the contents of CPU registers (including the program counter and stack pointer, among others), and information about I/O (such as open fifiles which can be read or written).

- The process API consists of calls programs can make related to processes. Typically, this includes creation, destruction, and other useful calls.
- Processes exist in one of many different process states, including running, ready to run, and blocked. Different events (e.g., getting scheduled or descheduled, or waiting for an I/O to complete) transition a process from one of these states to the other.
- A process list contains information about all processes in the system. Each entry is found in what is sometimes called a process control block (PCB), which is really just a structure that contains information about a specific process.

4.8 Homework(Simulation)

This program, process-run.py, allows you to see how process states change as programs run and either use the CPU (e.g., perform an add instruction) or do I/O (e.g., send a request to a disk and wait for it to complete). See the README for details.

4.9 Question

1. Run process-run.py with the following flflags: -l 5:100,5:100. What should the CPU utilization be (e.g., the percent of time the CPU is in use?) Why do you know this? Use the -c and -p flags to see if you were right.
2. Now run with these flags: ./process-run.py -l 4:100,1:0. These flags specify one process

with 4 instructions (all to use the CPU), and one that simply issues an I/O and waits for it to be done. How long does it take to complete both processes? Use -c and -p to find out if you were right.

3. Switch the order of the processes: -l 1:0,4:100. What happens now? Does switching the order matter? Why? (As always, use -c and -p to see if you were right)

4. We'll now explore some of the other flags. One important flag is -S, which determines how the system reacts when a process issues an I/O. With the flag set to SWITCH ON END, the system will NOT switch to another process while one is doing I/O, instead waiting until the process is completely finished. What happens when you run the following two processes (-l 1:0,4:100 -c -S SWITCH ON END), one doing I/O and the other doing CPU work?

5. Now, run the same processes, but with the switching behavior set to switch to another process whenever one is WAITING for I/O (-l 1:0,4:100 -c -S SWITCH ON IO). What happens now? Use -c and -p to confirm that you are right.

6. One other important behavior is what to do when an I/O completes. With -I IO RUN LATER, when an I/O completes, the process that issued it is not necessarily run right away; rather, whatever was running at the time keeps running. What happens when you run this combination of processes? (Run ./process-run.py -l 3:0,5:100,5:100,5:100 -S

SWITCH ON IO -I IO RUN LATER -c -p) Are system resources being effectively utilized?

7. Now run the same processes, but with -I IO RUN IMMEDIATE set,which immediately runs the process that issued the I/O. How does this behavior differ? Why might running a process that just completed an I/O again be a good idea?

8. Now run with some randomly generated processes: -s 1 -l 3:50,3:50 or -s 2 -l 3:50,3:50 or -s 3 -l 3:50,3:50. See if you can predict how the trace will turn out. What happens when you use the flag -I IO RUN IMMEDIATE vs. -I IO RUN LATER? What happens when you use -S SWITCH ON IO vs. -S SWITCH ON END?