【OS】Day7

【Ch2】Process(2)

4.1 The Abstraction: A Process

To understand what constitutes a process, we thus have to understand its machine state: what a program can read or update when it is running.

One obvious component of machien state is its memory. Instructions lie in memory; the data that the running program reads and writes sits in memory.

Also part of the process's machine state are registers. For example, the program counter(PC)*(sometimes called the instruction pointer or IP)*, tells us which instructions of the program will execute next; similarly a stack pointer and associated frame pointer are used to manage the stack for function parameters.

4.2 Process API

- Create: An operating system must include some method to create new processes.

  When you type a command into 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: Systems also provide an interface to destroy processes forcefully.

  When a process malfunction, the system destroys them

- Wait: Sometimes it is useful to wait for a process to stop running.

- Miscellaneous Control: Other than killing, creating, and waiting, systems provide other interfaces.

  For example, the system can suspend a process and then resume it later.

- Status: There are usually interfaces to get some status information about a process as well.

4.3 Process Creawtiong: A Little More Detail

How does OS create a process?

Load Data and Code from Disk

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 in some kind of executable format; the OS will load the executable file from the disk into the memory.

In early operating system, the loading process is done eagerly.(All at once before running the program)

Modern OSes perform the process lazily.(By loading pieces of code or data only as they are needed during program execution)

Allocate Stack

Once the code and static data are loaded, 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. The OS allocates 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)

Allocate Heap

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

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

I/O Setup

The OS will also do some other initialization tasks, particularly as related to input/output(I/O).

For example, in UNIX systems, each process by default has three open file descriptors.

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 will be discussed next chapter), the OS transfers control of the CPU to the newly-created process, and thsu the program begins its execution.

4.4 Process States

Now let us talk about the different states a process can be in at a given time. In a simplified view, a process can be in one of three 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 even takes place.

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

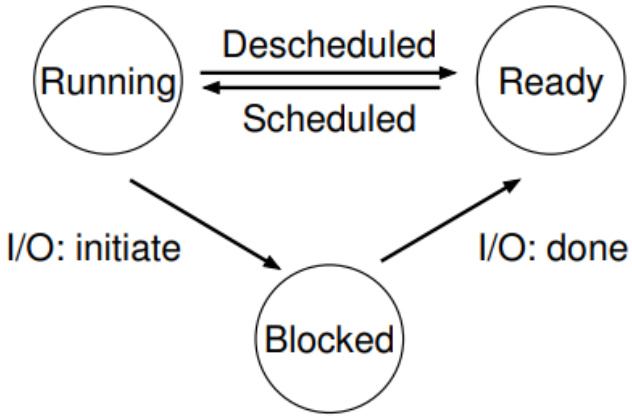

Figure 4.2: **Process: State Transitions**

A process can be moved between the ready and running states at the discretion of the OS.

Being moved from ready to running meants 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 moveds to the ready state again.

Let’s look at an example:

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**

$Process_0$ initiates an I/O and becomes blocked waiting for it to complete; processes become blocked, for example, when reading from a disk or waiting for a packet from a network.

The OS recognizes $Process_0$ is not using the CPU and starts running $Process_1$. While $Process_1$ is running, the I/O completes, moving $Process_0$ back to ready. Finally, $Process_1$ finishes, and $Process_0$ runs and then is done.