Chapter 5: Process Management in Theory and x86

Introduction

In the previous chapters, we have discuss the topics that helped us to understand basics in order to initialize the environment for a 32-bit protected mode kernel running on x86. Starting from this chapter we are going to discuss the topics that belong to the kernel itself, that is, the responsibilities of the kernel. We start with a quick look on theories that traditionally presented on academic textbooks, then we move to the practical part in order to implement these theories (or part of them) in 539kernel. A good place to start from is *process management*.

A kernel has multiple responsibilities, one of these responsibilities is to manage the resources and make sure they are managed well. One important resource of the computers is the time of the processor (AKA: CPU time), which is the component that execute the code of software that we would like to run on our machines. The part of process management is the one that studies how a kernel should manage and distribute CPU time among a bunch of *processes*.

The Most Basic Work Unit: A Process

Process is the term which is used in operating systems literature to point to a running program. In the previous chapters of this book we have encountered the concept of the process multiple times and you may recall from these encounters that every user-space software that we use in our computers are soulless sequence of bytes that are stored somewhere in the hard disk. When we decide to use a specific software, for example, the web browser, the first thing we do is to open it either through double clicking on its icon in graphical user interfaces or through writing its command in the shell. When we do that, the kernel is needed to be called through a *system call* and take the responsibility of "opening" this software, we can consider system calls as functions which are provided by the kernel to expose its services for the user-space software, one way of implementing system calls is to use interrupts, exactly the same way that we have used with BIOS. However, there are multiple steps that are needed to be performed to open the software, for example, reading its data from disk, but our current focus process-related parts, eventually, the kernel creates a new process for the software that we requested to open. After that, the kernel will have a table of current processes, each entry represents a process and contains the information which is needed by the kernel the manage the processes, this data structure which stores a process information is known as *process control block* (PCB). Of course, that most important part of the process is the code of the software that

this process represents and its data, both data [1] and code should be loaded into memory, after that, its code can be executed by the processor. We need to note that a process is an instance of a software, in other words, one software can be opened more than one time, for example, opening multiple windows of the web browser on the same time, the software is one which is the web browser, but each opened window is a separated process. While its well-known as process, specially in the literature, other names can be used for the same concept, for example *task* and *job* are other words which are used to point to the same concept.

Each process is known to have a *state*, when it is loaded to memory its state will be indicated by the kernel as *ready* and can be run anytime, when the CPU time is given to a process its state will be *running*, let's assume for example that the process has performed I/O request, its state will be *wait* since it's waiting for the I/O device to fulfill the request. These information about the process are maintained by the kernel by using a process table which has process control blocks as entries. Both process table and process control blocks are realized by some data structure of the kernelist choice. While this book is about operating system kernels, we will not cover the topic of data structures, but you need at least to know that a data structure is a way to store information in computers, usually, this way makes some operation better either in term of performance or space. For example, a data structure known as *binary search tree* (BST) which is designed to make the search operation fast.

Sometimes, a bunch of processes in a system need to communicate with each other, for example, to share some data or tell each other to perform a specific operation, this need led to a broad topic known as *inter-process communication* (IPC) which provides mechanisms to make this type of communication possible. The applications of IPC is not restricted to operating system kernels, they are used in distributed computing for instance. One well known mechanism of IPC is *shared memory*, that is, a region of the memory that is accessible by more than one process, they can read and write to this region in order to share data. The ability to write to same place by more than one process can cause a problem known as *race condition*, given a shared variable, the situation which two or more processes try to change the value of this variable at the same time is known as race condition. There are multiple solutions for this problem and this topic is studied in a branch known *concurrency control* which is a shared topic by many applications, one of the is database management systems (DBMS) which needs these mechanisms when two users try to update the same row at the same time.

Processes are not the only entities that need to communicate, there is another unit of work which is known as *thread* and it can be described as lightweight process. A process can have more than one thread and when a software uses more than one thread to do its job, it is described as *multithreaded*. Threads are everywhere in our usage of computers, and a world without them is unimaginable.

[1] We mean static data here, which are contained in the file of the software. While the data that are generated by the running process are not loaded from the binary file, instead they are created while the code is running.

For example, when you use a text editor, the main thread of the software lets you write your text, but when you click on save icon a separated thread within the text editor's process is used to perform this operation. Furthermore, another thread can be used for spell checker while you are typing. If all there functionalities were on one thread, you need to wait each one of them to finish in order to let you to perform the other functionality, that is, the software without threads is going to run sequentially while threads provide us concurrency within one process. Threads and processes have many similarities, for example, both of them are there to be executed, hence, they need to use the processor and both of them need to be scheduled to give every one of them time from the processor. In contrast to processes, threads run as a part of same process and share the same address space of the software which makes the communication between them much easier.

The Basics of Multitasking

When we write a kernel, multiple design questions should be answered [2] and the part of process management is not an exception of that. There are multiple well known answers for some basic design questions, each of those answer tries to solve a problem that faced the person who gave us this answer to solve this problem. For example, one of well-known features of the modern operating systems is *multitasking* which is a successor of *monotasking*, in the first one, the system can run multiple processes at a time even if there is one processor available, while in the second one, the system can run only one process at a time until it finishes its work or the user closes it, only after that, another process can be run.

In the days of monotasking we were facing a serious problem that led to the birth of multitasking, it has been noticed that the processes tend to have idle time, for example, when the process is waiting for the hard disk to give it some stored data, the process will be idle while it is taking over the processor, which means we are wasting the valuable resource of CPU time in waiting some action to happen, we need to utilize the processor as much as possible, and here came the solution of this problem, by letting the kernel to have a list of processes to be *ready* to run, assuming the machine has just one processor with one core, the CPU time will be given to, say process `A`, for some time, at some point of running time, process `A` requests from the disk some data and due to that it becomes idle waiting for disk to response, instead of keep the control of the processor for process `A`, which is doing nothing but waiting right now, the kernel suspend process `A` and give the CPU time to another process, say process `B`, this switching between two processes is known as *context switch*. The process `B` is going to use the processor while process `A` is waiting for the disk to respond. At some point, process `B` will perform some action that makes it idle which means that the kernel can switch to another ready process and so on. This solution

[2] Remember the job of a kernelist!

is known as *multiprogramming*, to sum it up, we have a list of ready processes, choose one, give it the CPU time and wait for it until it becomes idle, since it's waiting for something switch to another process which is not idle.

Better yet, multiprogramming has been extended for more processor utilization. Instead of waiting for the current process to perform something which makes it idle, why don't we suspend the current process after some period of running time whether it is idle or not and switch to another process? This solution is known as *time sharing* which is with multiprogramming represent the scheme that modern operating systems use for multitasking. In time sharing, a list of ready processes is there for the kernel, in each unit of time, say for example, every `1` second [3], the currently running process is suspended by the kernel and another process is given the CPU time and so on. You may recall from the previous chapter the system timer which emits an interrupt every unit of time, this interrupt can be used to implement time sharing to switch between the processes of the system, of course the kernel needs an algorithm to choose which process to run next, this kind of algorithms are known as *scheduling algorithms* and in general this part of the topic is known as *scheduling* in which we try to find the best way of choosing the next process to run in order to satisfy our requirements. The *scheduler* is the part of the kernel that schedules the next process by using some specific scheduling algorithm. There are many scheduling algorithms to deal with different requirements and one of them is known as *round-robin*. In this algorithm gives each process a constant amount of CPU time known as *quantum*, when the running process finishes its quantum the scheduler will be called and the CPU time will be given to the next process in the list until its quantum finishes and so on until the schedule reaches to the end of the process list where it starts with the first process again. The value of the quantum is decided by the kernelist, for example `50` milliseconds, which means each process will run `50` milliseconds then suspended to run the next one on the list and so on.

Both multiprogramming and time sharing solutions give us a type of multitasking known as *preemptive multitasking*, the processes are forced by the kernel to give the CPU time to another process and no process can take over the processor for the whole time. Another type of multitasking is known as *cooperative multitasking* (or *non-preemptive multitasking*), in this type the context switch is not perform forcibly as in preemptive multitasking, instead, the currently running process should cooperate and voluntarily tells the kernel when it should be suspended and a context switch should be performed. One of the obvious problem of this way, at least for the well-known workloads (e.g. servers or desktops), that a process, which runs a code that has been written by someone we don't know, cannot be trusted. It simply may take over the CPU time and never cooperate and give it up due to an error in the code or even in purpose [4].

[3] In practice, it is shorter.

[4] You may ask who would use cooperative multitasking and gives this big trust to the code of the software! In fact, the versions of Windows before 95 used this style of multitasking, also, Classic Mac OS used it. Why? You ask, I don't know exactly, but what I know for sure is that humanity is in a learning process!