Process Concept and State

1 Introduction

Process is one of the fundamental concepts in modern operating systems. It was first introduced by the designers of Multics in the 1960s. Many definitions have been given for this term since then, but the most common and simplest one is:

A process is an execution of a program.

Why do we need this concept and how this definition is obtained? The development of computer system made it introduced naturally. As we mentioned before, multiprogramming is designed to improve efficiency by keeping the processor and I/O devices, simultaneously busy. Due to the gap between the speed of processors and I/O devices, a program that performs I/O operations has to wait for the completion of the operations. To avoid the idle of CPU, another program may be loaded into main memory and be given the control of CPU to execute. Thus at some moment, we may have the main memory layout illustrated by Figure 1:

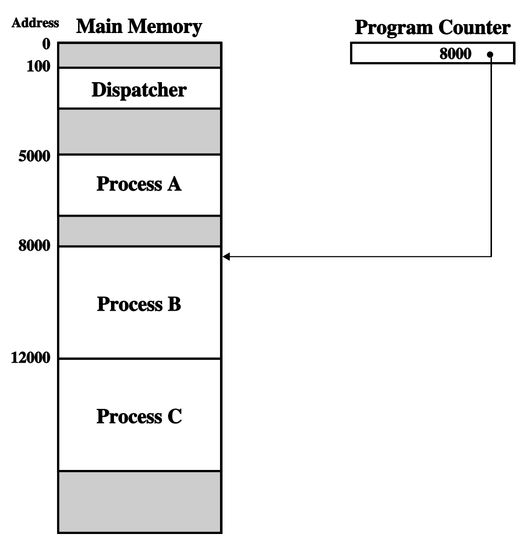

Figure 1: Main memory layout for multiprogramming

The programs that reside in main memory are absolutely different from their counter-parts, the program files on hard disks or tapes. The former are dynamic in the sense that they are running, while the latter are static. Thus we call processes these entities that have been loaded into main memory and are able to run or already in execution. If this is not convincing enough, let's consider an interesting case, suppose *Process B* and *Process C* in Figure 1 are different executions of a same program, say `nachos`. Then obviously, to distinguish the two `nachos`es, we definitely need a new term, instead of *program*, so the term *process* comes.

2 Process state

2.1 Traces of processes

For better control of processes, operating systems need to consider their dynamic behaviors. Figure 2 shows the typical behavior of a process.

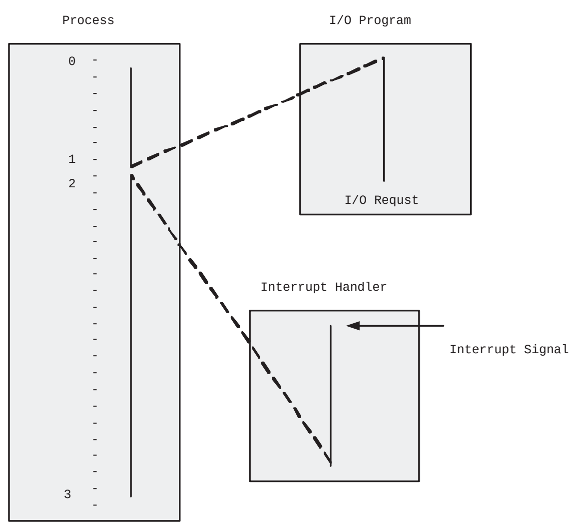

Figure 2: The typical dynamic behavior of a process

A process consists of a set of instructions, which may include ones related to I/O operations. In Figure 2, suppose at *Point 1* is an instruction that requests data from an I/O device. Thus when the processor executes to this point, the control is switched to the corresponding I/O program, which first prepares parameters for I/O operation, and finally makes the request. For efficiency, the processor then switches to another process while the I/O operation is going on. When the operation is finished, an interrupt signal will be generated. The processor then stops running the active process, and invokes the corresponding interrupt handler. When the interrupt handler routine is completed, the processor then may resume the execution of the process that has been inactive due to waiting for the completion of I/O operation.

If the processes in Figure 1 are taken as an example, we may trace their behaviors over time as illustrated in Figure 3. Solid line segments indicate that the corresponding processes are

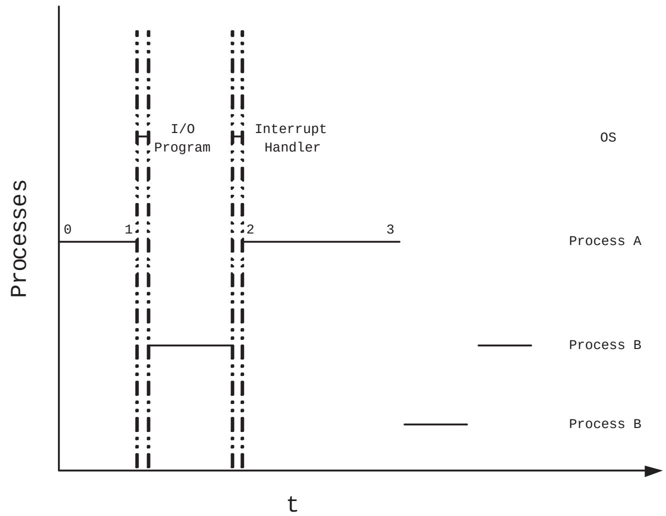

Figure 3: Traces of processes of Figure 1

occupying the processor, and otherwise not. Part of CPU time is also spent to execute I/O programs and interrupt handlers, which are generally considered as part of the operating system. Note that time sharing issue is not taken into account here, otherwise each process will achieve a series of short CPU time periods, instead of a long run as in the figure.

There is no magic when the switch of control happens from one process to another. Some facility in OS is needed to be in charge of this, which is often called *dispatcher*, dispatching the resource of CPU time to processes, or *scheduler*, scheduling processes to occupy the processor and run.

2.2 A two-state process model

By observing the above figure, we can come up with the simplest possible model of process behavior. As Figure 4 (a) shows, a process may be in one of the two states: *Running* or *Not Running*.

When the operating system creates a new process, it enters that process into the system in the *Not Running* state. From time to time, the currently running process will be interrupted due to either I/O operations or the end of its time slice, and the dispatcher will select another process to run. The former process moves from the *Running* state to the *Not Running* state, while the latter moves to the *Running* state.

To be scheduled by the dispatcher, each process should be represented in some way so that

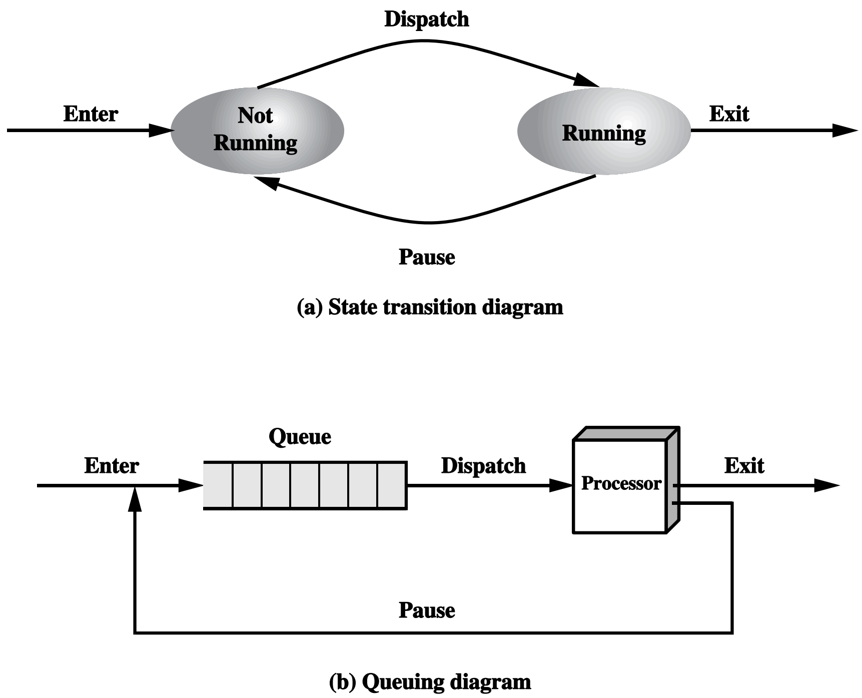

Figure 4: Two-State process model

the operating system can keep track of it. That is there must be some information relating to each process, including current state and location in the memory. At any time, there is at most one running process in an uni-processor system, but probably many processes that are not running. They must be kept in some sort of data structures, waiting their turn to execute. Figure 4 (b) suggests a queue structure. There is a queue in which each entry is a data block of some type including a pointer pointing to a particular process. When a process loses the processor for some reason, it is transferred to the queue and the dispatcher will then select a process from the queue.

2.3 The creation and termination of processes

Through the two-state process model, we have roughly discussed the trace of processes, but we haven't covered the two ends of a process's life, i.e. the creation and termination of processes.

Process creation

When a new process is to be added, the operating system builds the date structures that are used to manage the process and allocates space in main memory to the process.

There are several events leading to the creation of processes. In a batch environment, a process is created in response to a submission of a job; in an interactive environment, a process is created when a user types a command to execute. Users may also creates a process explicitly in a program by invoking a specific API. For example in Java,

```
Process p = Runtime.getRuntime().exec("md temp");
```

And when one process spawns another, the former is referred to as the *parent process*, and the spawned process is referred to as the *child process*.

Process termination

Similarly, a process may terminate in all kinds of ways. You may press the cross button at the top right corner of a Windows application to close it; You may type `exit` in a UNIX shell console to terminate the current session. Inside a program, it may call `exit(0)` to terminate normally or `exit(1)` to indicate an abnormal finish. When a process finishes, the operating system will free the memory space it occupies and remove the data structures it allocated to manage the process.

2.4 A five-state model

The above two-state model, to some extent, reflects the behaviors of processes; however the real situation is much more complex. We cannot simply deal with all the processes that are not running in the same way. For example, in a time sharing system, a process may lose control of the processor just because of timeout, i.e. its time slot is used up. Although it moves to the *Not Running* state, it is always ready to run right away. But some processes stopped running to wait for the completion of I/O operation. Before the interrupt signal notifies the completion, it cannot run and make any progress, and have to wait. Thus it is not proper to use a single state and a single queue for all the not-running processes.

Naturally, the *Not Running* state is to be split into two states: *Ready* and *Blocked*. Thus if the creation period and termination period of a process are also considered states, we will obtain a five-state model as depicted in Figure 5.

The *New* state indicates a process has just been created but has not been admitted to the pool of executable processes by the operating system. Typically, a new process has not yet been loaded into main memory. For example, the operating system may limit the number of processes that may be in the system for reasons of performance or main memory limitation. Note that the data structures for managing the new processes have already been allocated and maintained in main memory.

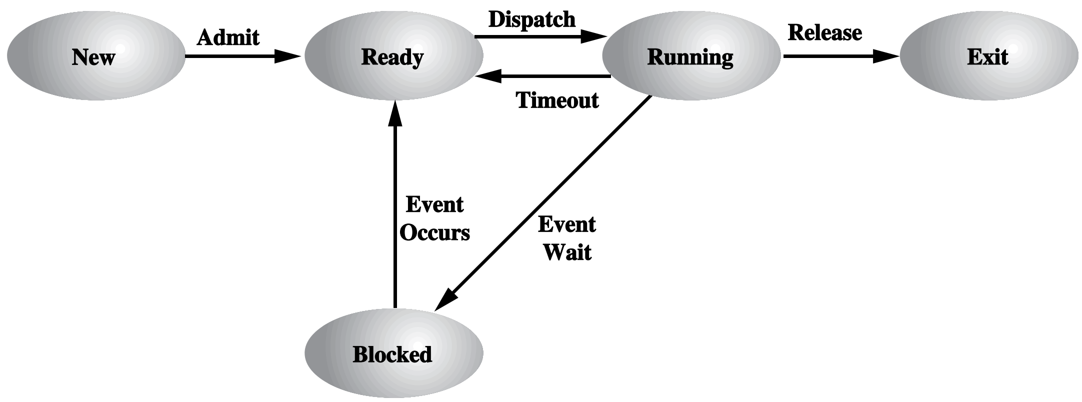

Figure 5: Five-State process model

The *Exit* state indicates a process has been released from the pool of executable processes by the operating system, either because it halted or because it aborted from some reason. This state enables some accounting programs to record the processor time and other resources utilized by the process for billing purposes, or utility programs to extract the information about the history of the process for purposes related to performance or utilization analysis.

Processes in the *Blocked* state cannot execute until some event occurs, such as the completion of I/O operation, while a ready process is always prepared to execute when given the opportunity.

Correspondingly, the queuing diagram in Figure 4 (b) may be extended to reflect this five-state model. Figure 6 (a) differentiates the blocked processes and the ones that may be dispatched again immediately by giving two paths from the *Running* state to the *Ready* state. An additional queue is set up for blocked processes. It means that when an event occurs, the dispatcher will go all the way through the queue for those processes waiting for that event. In some cases, there may be hundreds or even more processes in that queue, therefore it would be more efficient to have a number of queues, one for each event. Thus Figure 6 (b) is obtained.

2.5 Process swapping

The five-state model provides a systematic way of modelling the behavior of processes. Many operation systems are indeed constructed using this model.

However there are still problems. Recall that the reason for introducing the process concept and process state transition model is that there is a huge gap between the

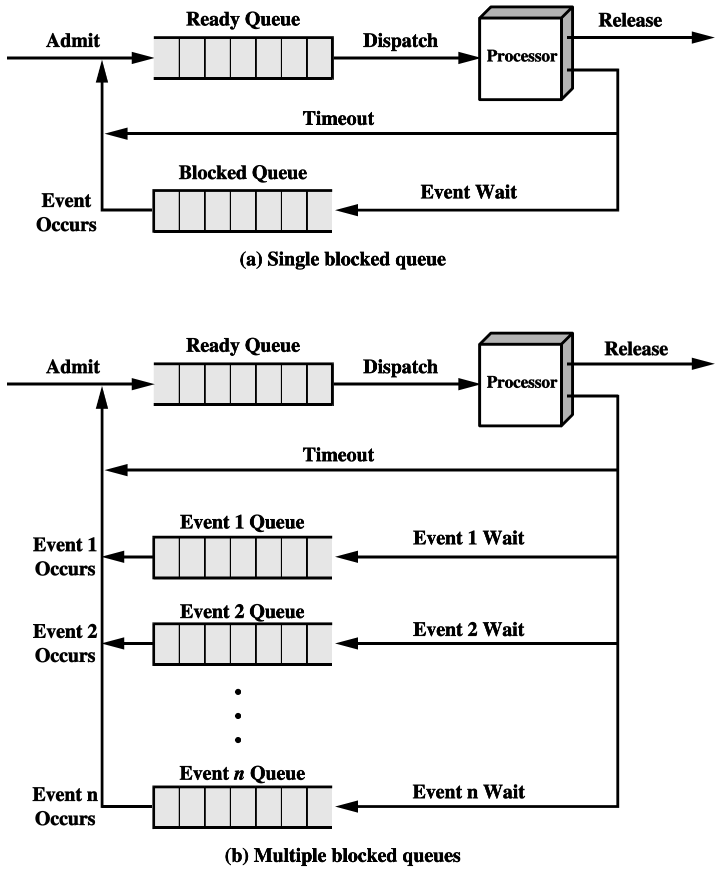

Figure 6: Queuing model of Figure 5

speed of CPU and I/O devices. When a process has to wait for the completion of I/O operation and thus gives up the processor, dispatching control to another process may avoid the idle of the processor. But this arrangement does not entirely solve the problem. The processor may be so much faster than I/O that all of the processes in memory are waiting for I/O. Thus even with multiprogramming, a processor could be idle for a long time.

Why not expand main memory so that more processes may be accommodated? It is possible, but cannot be a once-for-all solution since more memory means higher cost and with more memory the average size of programs is also likely to increase.

A real solution is *swapping*, which involves moving part or all of a blocked process from main memory to disk. Thus memory space may be freed for the system to bring in new processes to run. With swapping, a new state, *Suspend*, must be added to the process behavior model, as depicted in Figure 7 (a).

However a single *Suspend* state is still not enough, since the system needs to distinguish the suspended processes that remain blocked and those that are though sus-

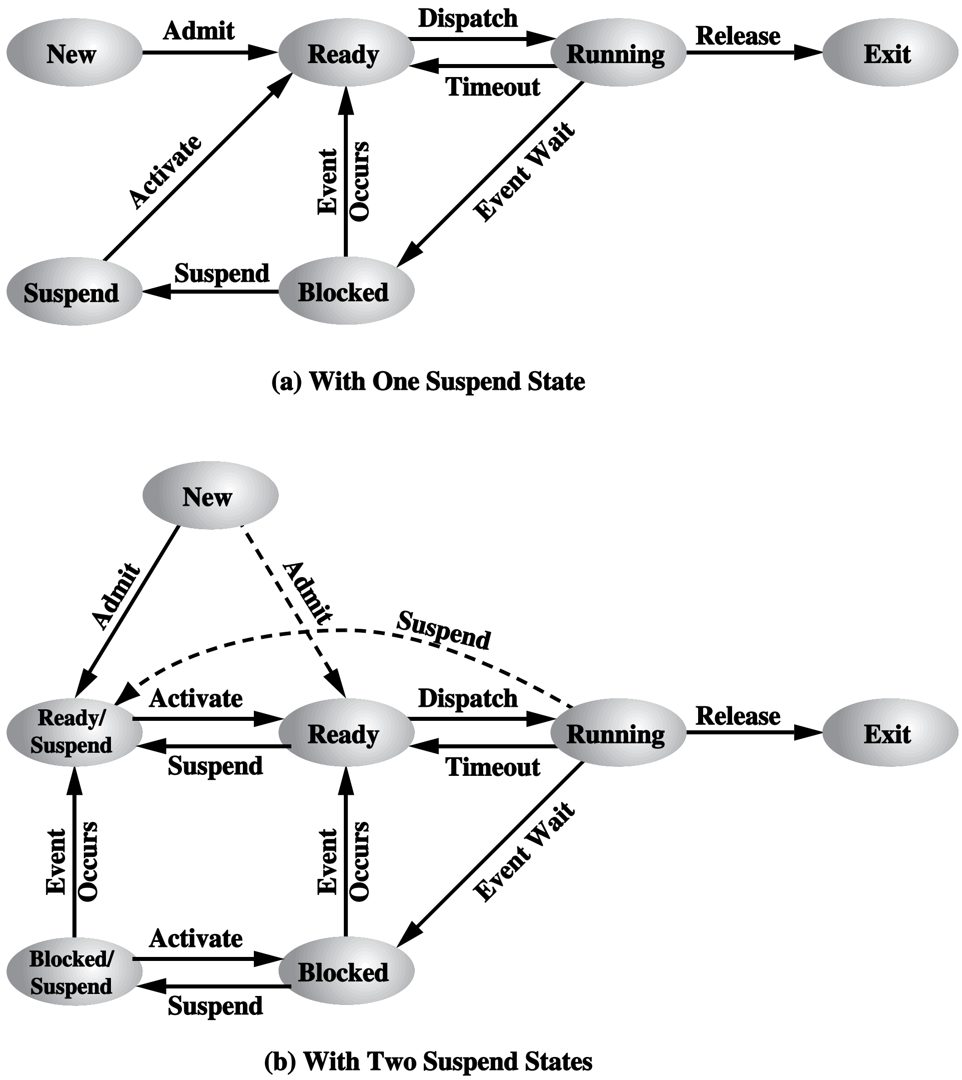

Figure 7: Process State Transition Diagram with Suspend States

pended and residing in secondary memory but are available for execution as soon as they are loaded into main memory. Accordingly, two *Suspend* states, *Blocked/Suspend* and *Ready/Suspend*, are introduced in Figure 7 (b). A process may move from *Blocked/Suspend* to *Ready/Suspend* when the event for which it has been waiting happens. All processes in either states may be brought back into main memory.