3. Processes as Resource Consumers

In this chapter we will explore the notion of "process" and introduce a powerful way to abstract the behavior of a process in a (computing) systems.

3.1 Concurrency in Computing Systems

As we alluded to before, the management of concurrency (coordination, resource management, synchronization, etc.) is a central theme of this course. Before we delve into details regarding mechanisms and approaches for the management of concurrency, we must first understand why concurrency is necessary in most modern computing systems.

3.1.1 Efficient Use of Resources dictates Concurrency

In a uni-programmed environment, a single program cannot make efficient use of a resource (e.g. CPU) when it is waiting for completion of some "work" on a different resource (e.g. a disk or a network interface). In Figure 3.1, while the program is waiting for service from some other resource (say the disk), CPU cycles are being wasted.

To make better use of the resource, we need to allow multiple programs to share in the use of the resource by interleaving their use of the resource as depicted in Figure 3.2.

If we focus on the amount of time the CPU and the disk are "busy", i.e. if we look at the combined execution of Program A and B, we obtain Figure 3.3. By comparing this with Figure 3.1 it is easy to see how interleaving the execution of multiple programs can lead to better *utilization* of system resources (in this case CPU and disk). Note that the increase in utilization is visible not only on the CPU, but also on the disk.

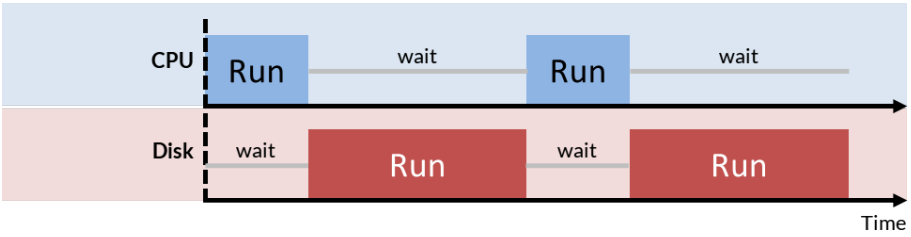

Figure 3.1: A uni-programming system. The considered program alternates between using the CPU and accessing the disk.

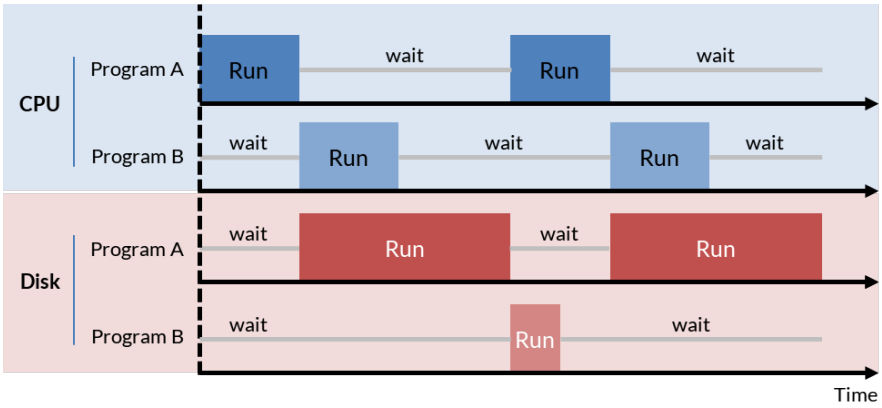

Figure 3.2: The execution of two programs that use disk and CPU is interleaved on the same system.

The co-existence of multiple executing programs sharing the same set of resources gives rise to the notion of a **multiprogramming** (or **multi-tasking**) environment.

3.1.2 System Scalability Dictates Concurrent Services

Consider a complex software system – say an operating system, a database system, or a web server. One way of building such a system is to think of it as a monolithic piece of software. This is obviously less than ideal because it does not allow for independent development (and separation) of functionality. A much better approach is to divide the system into basic components that are implemented independently, and which are run autonomously and communicate only when necessary. This is indeed the model most often used in the development of large software artifacts, whereby the system is designed as a collection of services that, together, constitute the "system."

One of the many advantages of this divide and conquer approach to software development is that it makes it possible to reuse services developed for one system in a different system. Another important advantage is that the issue of provisioning (deciding how much computing resources to dedicate to the system) becomes more manageable since, for example, one could have multiple services run on different computers in a distributed environment. This is indeed why the client-server model and variants thereof, (e.g. publisher/subscriber, producer/consumer, RPC) for building large software artifacts has been quite successful. All of these models imply that multiple services must

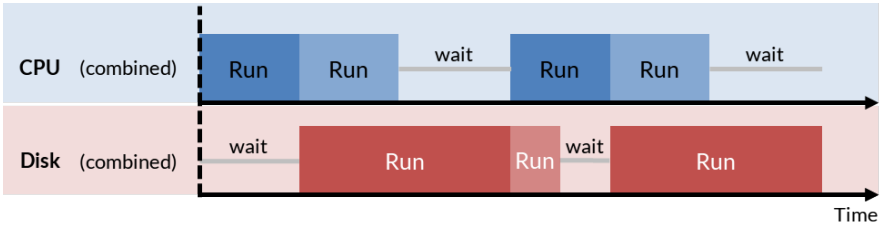

Figure 3.3: A combined view on CPU and disk of the execution of two programs interleaved on the same system.

co-exist and be executed concurrently, whether on the same computing infrastructure (i.e. processor) or on different ones.

3.1.3 Multiprogramming vs. Multiprocessing

Multiprogramming (or multi-tasking) implies the interleaved execution of programs without any portion of their execution actually overlapping in time. In fact, multiprogramming occurs when multiple programs must be executed on the same resource (or computing infrastructure – e.g. a processor). Since, on a single processor, only one program can be actively "executing" at any point in time, overlapped execution cannot occur. Figure 3.4 illustrates the interleaved execution of three programs on a single processor, whereby at any point in time, there is at most one program using the CPU.

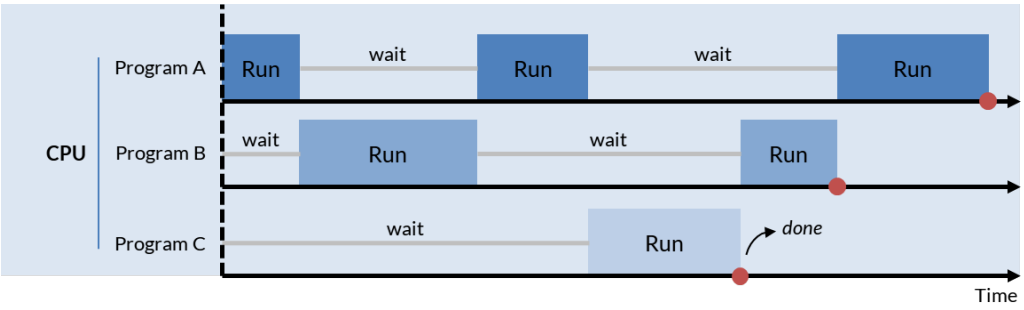

Figure 3.4: Multiprogramming system with three programs in interleaved execution on the same resource – a CPU.

In a distributed environment, however, it is possible for programs to simultaneously execute (e.g. in an environment with multiple processors such as the Internet or in a parallel computer system). This form of overlapped concurrent execution of programs is termed multiprocessing. Figure 3.5 illustrates this by showing the three processes executing on two processors, whereby at any point in time there could be upwards of two programs running at the same time.

It is important to realize that the concepts of "multiprogramming" and "multiprocessing" are not unique to program execution on a CPU. While these terms have been coined for purposes of studying concurrency of executing programs, the concepts of multiprogramming (a.k.a. interleaved used of a

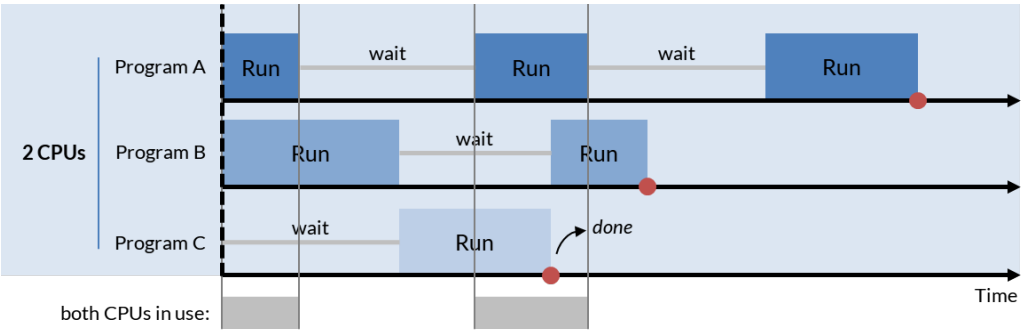

Figure 3.5: Multiprocessing system with three programs in interleaved and overlapped execution on 2 CPUs.

resource by multiple entities) and multiprocessing (a.k.a. overlapped use of multiple resources by multiple entities) are paramount in systems in general.

To support concurrency (i.e. interleaving with or without overlapping) we need to represent a "program being executed" as an entity that the operating system can manage. This entity is the notion of a **process**.

Box 3.1.1 Interleaved and Overlapped use of Resources in a Network

Consider, for example, the communication channels (i.e., links) before and after the router R1 Figure 3.6 and consider two senders Pa and Pb. Since each sender has its own link to the router R1, the transmission of packets from different senders can overlap in time before reaching R1 (left-hand side of the figure) – just like the scenario in which two processors are used in parallel to execute two programs[a]. In order to forward these packets to the Internet, however, a single link for both senders is used by R1 (right-hand side of the figure). On this (single) link, packets from different senders can only be interleaved but their transmission cannot overlap in time.

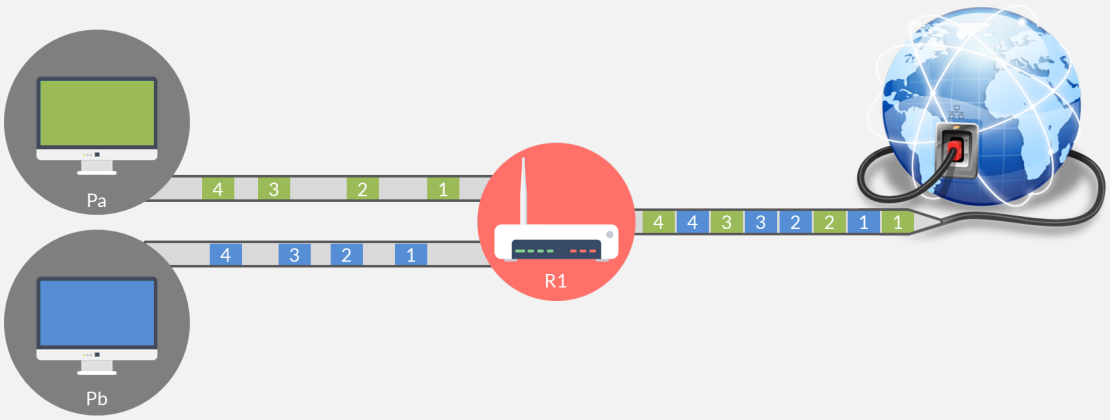

Figure 3.6: Example of overlapping (left-hand side) and interleaved (right-hand side) use of communication channels in a network.

[a] To distinguish between concurrency due to overlapping versus concurrency due to parallelism, the latter is often called "true concurrency." In most cases, for purposes of synchronization we do not distinguish between these two types of concurrency.

3.2 The Concept of a Process

A process is a program "in action." It is the sequence of instructions that specify the program along with the state of the memory associated with the program's execution. This memory consists of many data structures. Some of these are specified in the program (as in static and dynamic variables) or are created to support the execution of the program (as in the program stack). Others are necessary to allow the operating system to manage the process and the other resources in the system (as in the Process Control Block, or PCB).

As illustrated in Figure 3.7, the information in the PCB data structure includes "meta" information about the program being executed. This includes information such as the "owner," and "priority," as well as information about its resource usage for accounting purposes and monitoring. More importantly, the PCB also contains information about the "state" of the process.

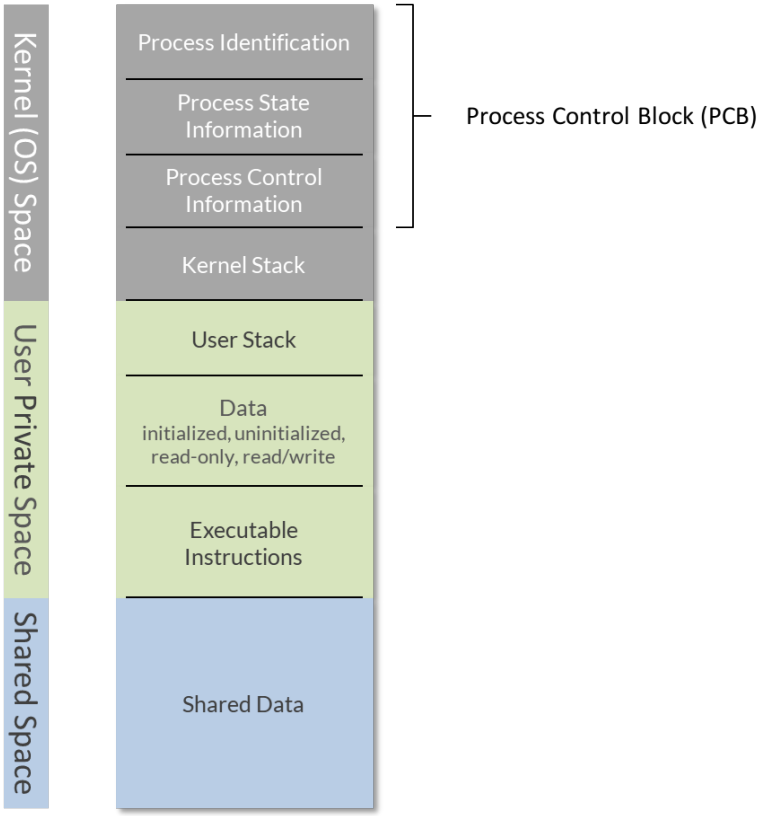

Figure 3.7: Structure of a process adressing space and location of the PCB.

3.2.1 Process (Execution) State

Often times, the operation of a process as well as the manner in which other entities in the system (including the operating system) interact with it depends on the "state" of that process. In the discussion below we focus on the state of a process as managed by an operating system for purposes of resource allocation.

Restricting our discussion to phases in the lifecycle of a process, we observe that a program becomes a **New** process when the necessary data structures for its execution are allocated (by the operating system). At this point, the process is **Ready** to start execution. At some point, a process is allowed to execute. At this point the demanded resource is allocated to the process and it is **Running**. The execution of a running process may have to be stopped to allow the process to wait for the occurrence of some type of event.(e.g. I/O, timer, signal from another process, etc.) A process that is stopped because it is waiting for an event is **Blocked**. Finally, when a process is finished, it **Exits**.

3.2.2 Process State Transitions

Figure 3.8 illustrates the "lifecycle" of a process. It shows not only the various "states" of a process (i.e. New, Ready, Running, Blocked, and Exit), but it also shows "why" a process may change its state.

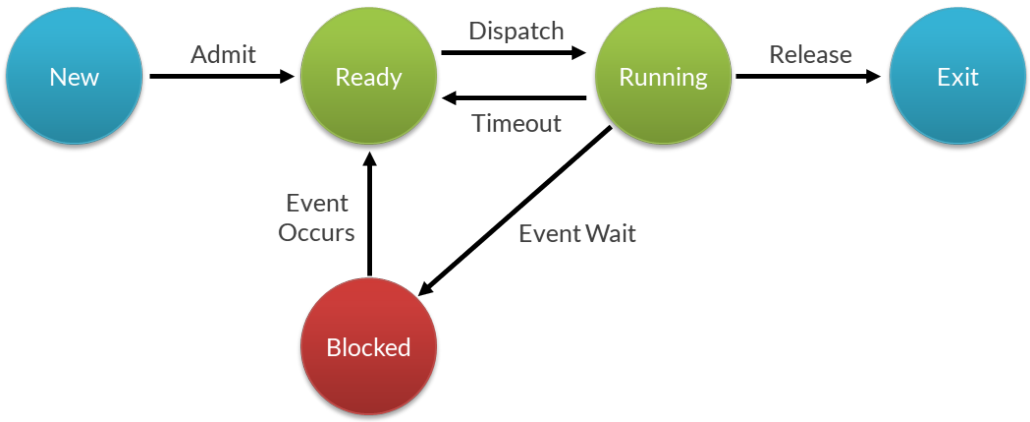

Figure 3.8: Life-cycle of a process expressed as a process state transition diagram.

For example, it is possible for a process to go from "Running" to "Ready" if the process's time slice expires. The expiration of the time slice is an event that occurs when a timer, which has been set by the dispatcher (the operating system's process responsible for scheduling various processes on the CPU), expires. The expiration of the timer results in an interrupt. The ISR for that "timeout" interrupt is none other than the dispatcher, which changes the state of the process from "running" to "ready" and awards the CPU to some other process that is "ready."

The state transition diagram in Figure 3.8 is perhaps too simplistic to account for all the states of a process in a modern operating system. However, it is sufficient to capture the most basic states in the lifecycle of a process (for the purposes of issues we will be dealing with in this course).

A more realistic state diagram will have to include states that account for a process that has been "swapped out" of main memory. A process may have to be swapped out of main memory if it stays blocked for a very long time (e.g. waiting for a user to come back from lunch to type a character on a keyboard) when memory is running out. In such a condition, it may be better for the operating system to suspend the process. A **Suspended** process (whether blocked or ready) is one which cannot be executed because it is swapped out of main memory.

Figure 3.9 shows the state diagram when process suspension is taken into consideration.

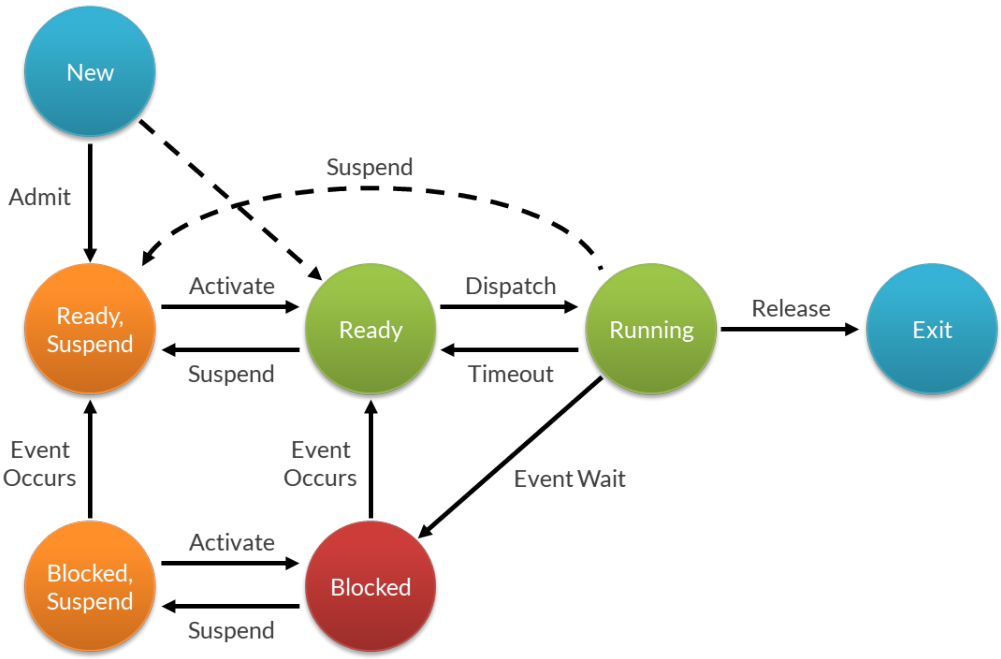

Figure 3.9: Process state transition diagram when suspension is supported.

3.2.3 Other Flavors of a Computing "Process"

To enable an efficient implementation of many of the resource management issues (in particular inter-process communication, sharing, etc.), it becomes handy to finesse the notion of a process. For example, a thread is a concept tightly related to that of a process. A thread (a.k.a. light process) is an independent execution of a function within a process. Threads enable concurrency within a process by allowing multiple parts of the process to execute concurrently. This is particularly valuable in multiprocessor environments. Threads within a process, however, share many of the resources "owned" by the process (e.g. file handles, locks, virtual-memory data structures, etc.)

Figure 3.10 shows how the use of multithreading enables a (multithreaded) server process to respond to two requests for service ("Remote Procedure Calls" or RPC) concurrently.

For the purposes of this class, however, we will not make a distinction between a thread and process. The same fundamental techniques that apply to processes (e.g. scheduling, synchronization, etc.) also apply to threads. Thus, we will view threads as a "nice hack" (a pretty important one,

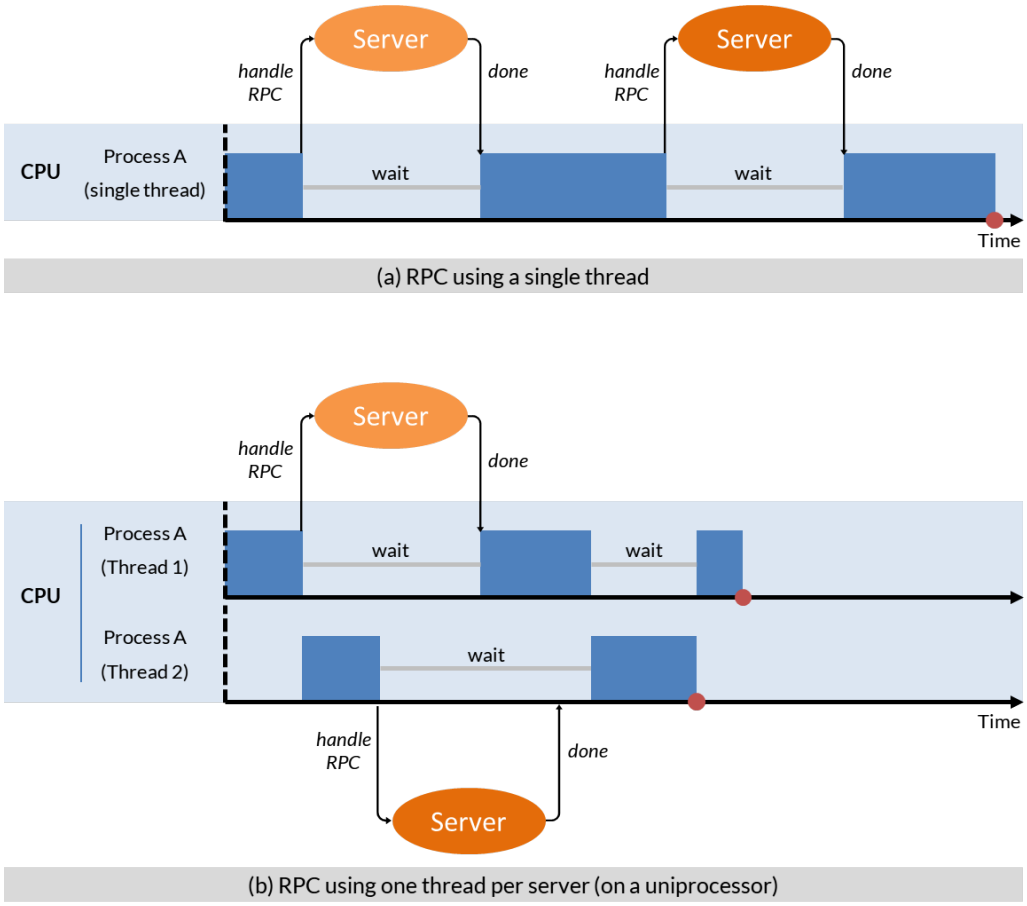

Figure 3.10: A multi-threaded server implementation where multiple threads handle incoming RPC requests concurrently.

actually), which does not really change how we think about the fundamental problems of concurrency management, scheduling, etc.

Another "hack" is related to the distinction between the "user" and "kernel" modes of a process. This distinction results in an even more complicated state transition diagram than the ones we have looked at so far. This distinction enables us to determine whether a process is "allowed" to do some "privileged" operations or functions. Privileged operations are those, which cannot be "trusted" to the user (e.g. because they involve access to critical shared resources such as disk, network, file system, etc.) Unix's answer to this is to allow a user process to be "promoted" and execute in "kernel mode" when it requests some special services (called "system calls"). Since these services are "written" by the OS developers, it is fair to assume that they will "do the right things." Thus, once a user process makes a system call, it is "promoted" until the service it requested is finished, at which point it is demoted back to user mode. This state of affairs is reflected in the process state diagram shown in Figure 3.11.

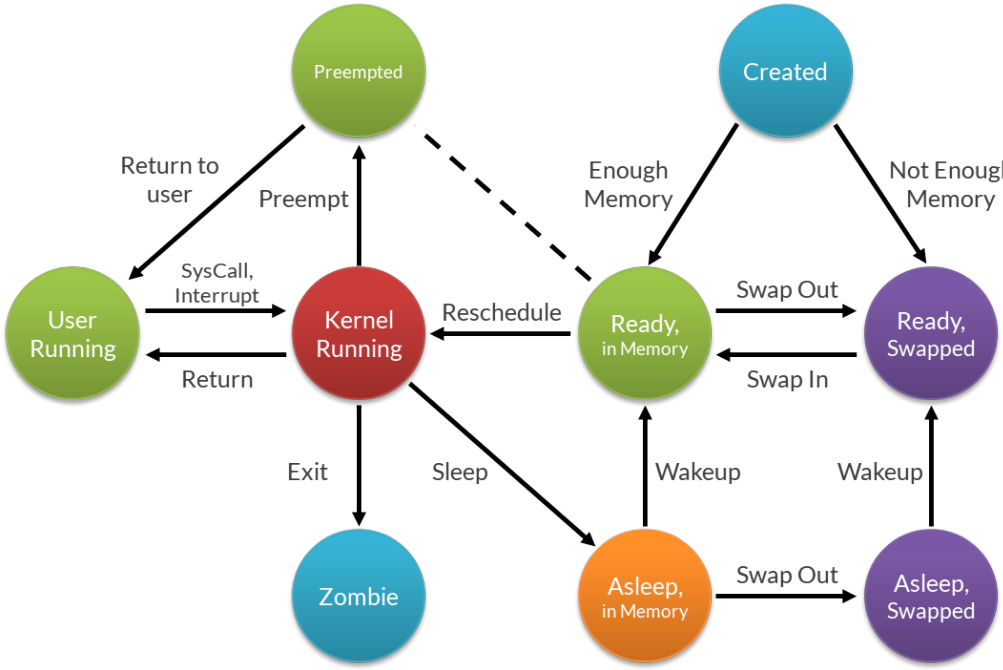

Figure 3.11: An even more elaborate process state transition diagram, taking inspiration from the way Unix handles processes.

3.2.4 Queues as a Discipline for Managing State Transitions

In order to manage the processes in a system (and there may be a huge number of those), queues come in handy.

For example, a dispatcher process (the operating system process that is responsible for awarding the CPU to one of the "Ready" processes) could be viewed as a simple queue manager (or server) that simply dequeues the next eligible process from a "Ready Queue" to the processor. Similarly, an ISR could be viewed as an event queue manager that (after doing all the housekeeping necessary to process an interrupt) will dequeue a process from its event queue to the ready queue.

Thus, it is very useful to think of the state of a system at any point in time as the state of a bunch of queues (possibly very many!) each of which has (pointers to) processes (PCBs) that are waiting for the services of the queue manager.

Figure 3.12 illustrates this perspective by showing various queues, each of which is being "served" by one of the system's resources. Processes in a given queue are waiting for the resource associated with that queue to become available. Until then, such processes are unable to make progress in their execution.

It is important to note that while we may be tempted to think of the items in a queue as "ordered" in a first-in first-out (FIFO) fashion, this may not be the only way items in a queue are dequeued. For example, in addition to a FIFO dispatching discipline, we may adopt a priority-based discipline, or a random policy, or any other policy. In later parts of this course we will examine examples of such

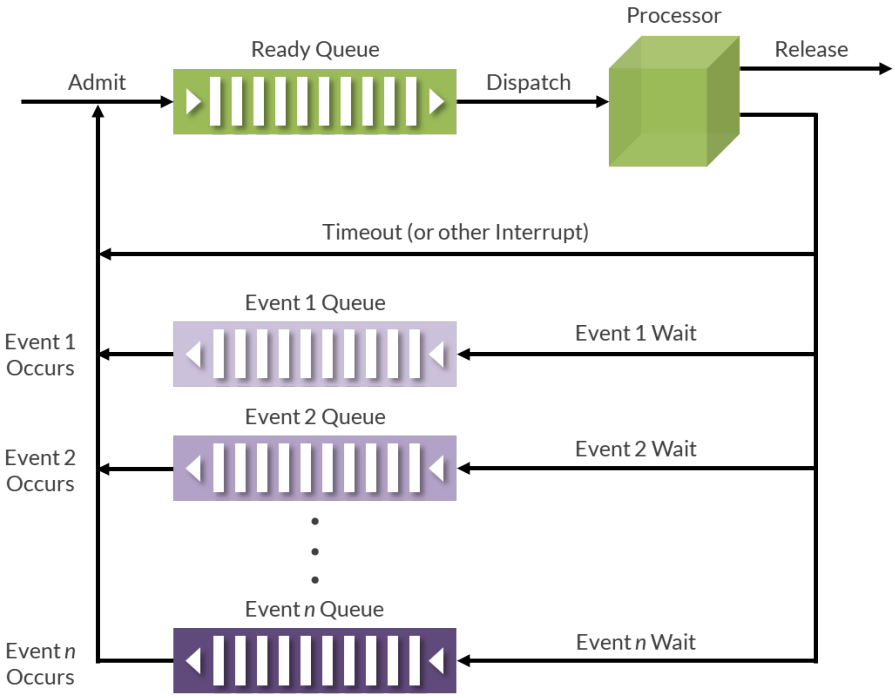

Figure 3.12: Process state transitions from a queuing perspective.

disciplines and their impact on system performance.

3.2.5 The Concept of a Process in Other Settings

As it should be evident by now, our use of the term "process" is not tied to any particular manner in which an operating system defines what constitutes a process. For example, we intentionally blurred the distinction between processes and threads—one can think of a thread as a process.

Our use of the word "process" should transcend systems in general. So, for the purposes of this course, we use the term "process" to refer to any consumer of resources that is "in action." A program being executed is a consumer of a resource—namely the CPU. Similarly, a connection between two network nodes (say a client and a server) going through a link in the network is a consumer of a resource—namely the link's capacity. And, just like a computing process in an operating system setting goes through different states in its lifetime (as illustrated by the various process state transition diagrams discussed before), a communication process (a.k.a. a connection between two network nodes) goes through a set of state transitions as well. In networking these states are defined by the protocol used for communication (see Box 3.2.1).

Box 3.2.1 TCP Connection State and State Transitions

The concept of a connection is a fundamental one in networking, as it encapsulates the notion of communicating processes. Just as processes on a computing system compete for underlying resources, connections between pairs of senders and receivers compete for network resources as well according to some established resource allocation protocol. One such protocol is the widely adopted TCP (Transport Control Protocol).

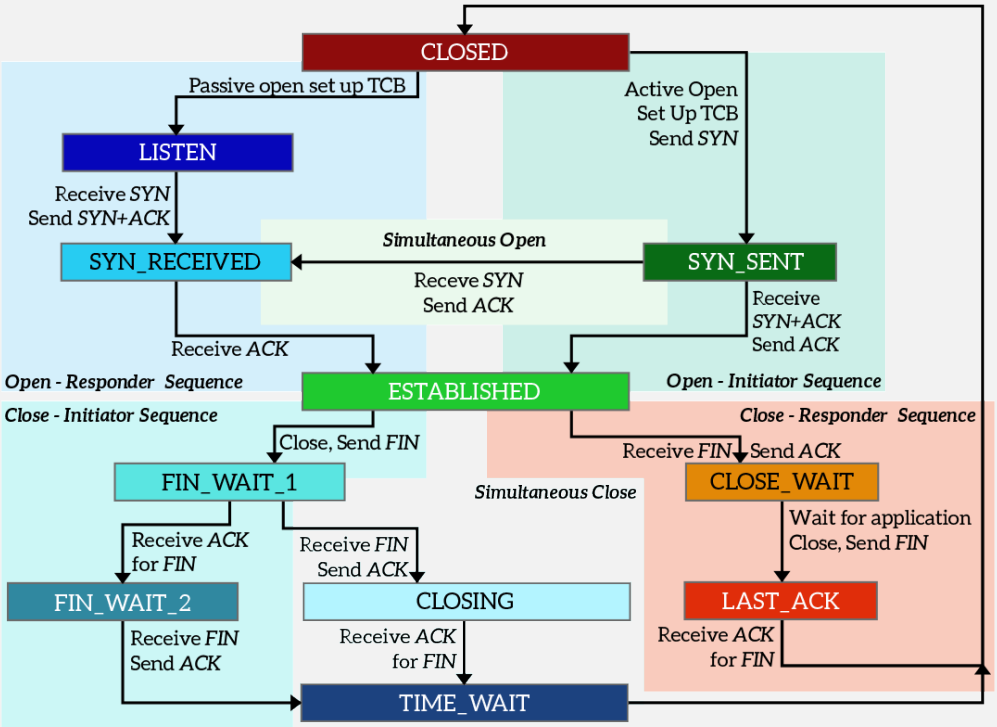

Figure 3.13: The state transition diagram of a TCP connection.

Figure 3.13[a] shows the various states that a TCP connection goes through in its lifetime as well as the possible state transitions taking it from one such state to the other. This is presented just for illustrative purposes—to underscore that, while in this course we will tend to think of processes as programs being executed (i.e., consumers of CPU cycles), the concepts we will consider are quite applicable to other settings of entities consuming other resources.

[a] Image adapted from `https://www.ictshore.com/category/free-ccna-course/`.