This chapter introduces the concepts of the process and concurrent execution. A process is a program in execution and is considered to be a unit of work in a modern time-sharing system. Such a system consists of a collection of different processes: operating-system processes executing system codes and user processes executing user codes. All these processes can execute concurrently, with the CPU (or CPUs) multiplexed among them. By switching the CPU among different processes, the CPU becomes more productive. This chapter also introduces the notion of a thread (lightweight process) and interprocess communication (IPC). Threads will be discussed in more detail in Chapter 4.

Process

- A process captures the entire duration of a program execution. Each process has a *life cycle*. It is an *active* entity in the sense that it can be in different **states** (*new, ready, running, waiting*, and *terminated*) during its lifetime; it requires many resources at different stages of the execution (CPU, memory, registers, files, I/O, possibly cooperating with other process) in order to execute a program. A program is considered as a *static* entity, which is the text part of a process.
- Each process is represented in the OS by a **Process Control Block** (**PCB**), which contains all information associated with a process. Each process is uniquely identified by a process ID or process number. This is a positive integer in Unix, and process ID=1 represents the root process, init().
- The state of a process can also be represented by the **queue** that a process is associated with at a given time instant. The OS manages all queues. A process during its lifetime can be in *different queues*, such as in the ready queue waiting for the availability of CPU, device or I/O request queues waiting for a specific device. The only exception is the process currently running on the CPU.
- In a multi-programming environment, in which multiple processes have been brought into the memory competing for CPU, memory and other resources, various scheduling algorithms are needed in order to handle the competition or/and resource sharing. We have so far only defined **long-term**, **medium-term** and **short-term schedulers**
- **Concurrency** refers to the fact that within a period of time, there are multiple processes running on the CPU, thus making progresses at *different time instants*. However, at a time instant, there could be at most one process running on the CPU if we have a single CPU. In another word, **concurrency** implies that there are multiple processes inside a system, their executions are *interleaved* or *multiplexed* on the CPU, and all of which can make progress within a period of time.
- The process defined here is referred as a *heavy-weight process*, in which there is no concurrency within the execution of this process itself; in another word, there is only one "thread" of execution, and instructions are executed sequentially.
- **Context switch** occurs when CPU switches from one running process to another process, in which the operating system must save the state of the old process and

load the saved state for the new process. Context of a process is represented by the PCB.

Multitasking in Mobile Systems

- Given the limited resources in mobile systems, such as processing capability, memory, and in particular battery, it usually restricts only one single *foreground process* to be running and on display while there could be a number of *background processes* (remains in memory, but do not occupy the display screen). There are a limited number of application types that can run in the background in iOS, but there is no such restriction in Android system, in which a background application must use a **service**, a separate application component that runs on behalf of a background process.

Operations on Process

- OS provides mechanisms for *process creation* and *termination*. Processes can execute concurrently; they can be created and terminated dynamically.
- A process may create new processes, via a system call during the course of its execution. The creating process is referred as the **parent process,** and the newly created processes are called the **children processes** of that process. There are a number of actions taken during a process creation including constructing a new PCB for the child process (i.e., create the data structure of PCB and allocate a unique process ID for the process), allocate memory space, copy data from the parent process and I/O state including all opened files.
- Once a child process is successfully created, it executes concurrently with other processes, possibly with different codes (exec() system call in Unix) and data.
- When a process is terminated, it has to return all resources to OS or OS has to *de-allocate* all resources from the terminating process. A process might need to wait for all of its children processes to be terminated before it can be aborted.

The fork() in Unix and Process Creation

- The fork() is the UNIX system call that creates a new process. This design might seem to be a bit odd, with some historical reasons. In the old days (20-30 years ago) when this was designed, there were not many concurrent processes running in a computer system and each process did not require a large memory space. The fork() expedites process creation by simply copying the address space (code and data) from its parent process.
- There is no parameter required when calling fork(), but fork() has return value(s).
- If fork() system call fails, no child process is created, a negative value (pid<0) is returned to the original process (the return value from the system call).
- If fork() is successful, a zero value (pid=0) is returned to the newly created child process, and the child process ID (i.e., a positive number) is returned to the parent

(original) process. Part of the reason that two different return values are needed is that the same codes right after fork() will be executed by both processes. In another word, unless the child process calls exec() system call to upload a new set of codes or a new program to run, the child process and the parent process will execute identical instructions starting right after the fork() (i.e., the program counters are the same). Therefore, this return value is used to distinguish which process (parent or child) is executing the context of the program after fork(). Meanwhile, the parent process also obtains the process ID (process identifier) of its child process in case it chooses to maintain in a table within its PCB.

- The codes right after fork() are executed by both original process (the parent process) and the newly created process (the child process), since the child process duplicates the address space of the parent (code, data, stack and etc.), such as PCB including registers and program counter from the parent process except the process ID. The child process must explicitly load a new program to run by using another system call (exec()) if it desires so.
- These two processes are concurrently executing. Either the parent process or the child process can run first (depending on the short-term scheduling while they are both on the ready queue). Their executions can be interleaved with each other, and with other processes in the system (Chapter 5 will discuss the CPU scheduling).
- Process creation system call CreateProcess() in Microsoft Window system is quite different, in which it requires loading a specific program into the address space of the child process at the time of process creation. It expects no fewer than 10 parameters (fork() has no parameter). Specifically, ZeroMemory() is for memory allocation. The equivalent of wait() is WaitForSingleObject(), which is passed handle of the child process, *pi.hProcess*, and waits for the child process to complete.

Process Termination

- A process that has terminated, but whose parent has not yet called wait(), is known as a **zombie process**. The Unix does not do anything as the parent is expected to call the system call wait() later.
- A parent process terminates without first calling wait(), its children are referred to as **orphan processes**. UNIX assigns the **init** process as the new parent of orphan processes and **init** periodically calls wait() that allows any resources allocated to terminated processes to be reclaimed by the operating system

Inter-Process Communication (IPC)

- Processes can be *independent* from or *cooperating* with other processes. The cooperation can bring several benefits (if desired) such as *information sharing, computation speedup, modularity* and *convenience*.
- Cooperating processes can exchange data and other information through an *inter-process communication* (IPC) mechanism. There are two basic modes of IPC:

1) **shared memory** and 2) **message passing**. Both are commonly used.

- In a shared-memory approach, communications are often implicit; for example two processes can share a variable and there is no explicit message exchanged between the two processes. While in the message-passing system, there are explicit message(s) delivered.
- Message-passing system can be further divided into *direct communication* and *indirect communication*. In the direct communication, the identity of the sender or the receiver needs to be known before the communication takes place. While in the indirect communication, the message is sent to a mailbox, no need to reveal the identity of the sender or receiver.
- In message-passing systems, communication between processes takes place through calls to send() and receive() primitives.

Client and Server Communications

- Communication in client-server systems may use (1) sockets, (2) remote procedure calls (RPCs), or (3) pipes.
- A socket is defined as an endpoint for communications. A connection between a pair of communications consists of a pair of sockets, one at each end of the communication channel. Each endpoint is uniquely specified by an IP address (host address) and a port number (identifying the specific process or application).
- In client-server systems, only clients can initiate the communications to servers. The servers with well-known sockets are always running and waiting for clients to contact them for services.
- The socket numbers smaller than 1024 are all reserved to be used by servers. The socket number used at a client side can be selected by the client arbitrarily as long as it is not used at the moment at the client. The socket number used at a server is prior-defined; for instance 80 used for Web server, 21 for ftp (file transfer protocol). The Internet Assigned Numbers Authority (IANA) is responsible for maintaining the official assignments of port numbers for specific uses.

Pipes

- **Ordinary pipes** allows two processes to communicate in a standard *Producer and Consumer* fashion. The producer writes to one end of the pipe (the **write-end** or fd(1) in Unix) and the consumer reads from the other end (the **read-end** or fd(0) in Unix). The communication is unidirectional.
- An ordinary pipe cannot be accessed from outside the process that creates the pipe. Typically, a parent process creates a pipe ad uses it to communicate with its child process.
- On Unix, ordinary pipes are constructed using the function pipe (int fd[]). Unix treats a pipe as a special type of file. Thus, pipes can be accessed using read() and write() system calls, the same way that files are accessed.
- **Name pipes** provide a much more powerful tool, which are bidirectional and does

not require parent-child relationship. Name pipes allow multiple processes to use it for communications and multiple processes can write to it.

- Name pipes are referred to as FIFOs in Unix system created with the mkfifo() system call. Once they are created, they appear as typical files in the file system.
- Name pipes can exist even if the processes using it for communications have terminated, while the ordinary pipe ceases to exist if the processes using the pipe have terminated.