This chapter introduces the concepts of 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. Processes and threads within a multi-threaded process execute ***concurrently***. By switching CPU between different processes, CPU becomes more productive. This chapter also discusses the basic operations on process – process creation and termination, and introduces the notion of a thread (referred as a lightweight process) and interprocess communication (IPC).

Process

- A process captures the entire duration of a program execution. Each process has a ***life cycle***. A process is considered to be an *active* entity in that it can be in different **states** (*new, ready, running, waiting*, and *terminated*) during its lifetime; it requires many resources during different stages of execution (CPU, memory, registers, files, I/O, possibly cooperating with other process) in order to execute a program. A program itself is a *static* entity, i.e., the text part of a process which is typically stored on hard drives.
- Each process is represented in the OS by a **Process Control Block** or **PCB** (a kernel data structure), which contains all information associated with one process. Each process is uniquely identified by a process ID (pid) or process identifier. It is a positive integer in Unix/Linux, and process ID=1 represents the root process, `init()` or `systemd()`.
- The OS uses a **Process List** (a kernel data structure) to keep track of all processes - indexed by PID and by pointing to the PCBs of all process currently in the system.
- The layout of a process in memory is represented by four different sections: (1) text, (2) data, (3) heap, and (4) stack. This defines the **address space** of a process.
- A process can also be represented by (1) a **thread** or multiple threads captured by the program counter (PC), registers and stack, and (2) **address space** includes data, text (program), and heap.
- The state of a process can also be determined by a **queue** that a process is associated with or resides on. The OS manages all queues. A process during its lifetime can be in *different queues*, e.g., in *ready queue* waiting for CPU, in a device queue waiting for a specific device ("waiting" state). The only exception is the process currently running on the CPU ("running" state). Each queue in the OS has its own scheduling disciplines or policies.
- In a multi-programming environment, multiple processes are brought into memory competing for CPU, memory and other resources, various scheduling algorithms are needed to manage the competition and facilitate resource sharing. We have insofar outlined the *long-term, medium-term* and *short-term* schedulers.
- The **long-term scheduler** is used primarily in mainframes or minicomputers which multiple users can submit their jobs to run (often remotely). It selects jobs from a *job queue* and bring them into the memory by allocating resources such as memory to jobs. Afterwards, the jobs or processes become "ready", thus join the ready queue. The long-term scheduler determines the **degree of multiprogramming** in a way, defined as the number of concurrent processes in a system, or simply the

system workload or the number of processes inside the memory.

- The **medium-term scheduler**, on the other hand, can be used to **swap** some partially executed processes from the memory out to the secondary storage temporarily, which reduces the degree of multiprogramming. This occurs in mainframes or minicomputers, when CPU utilization is too high or workload is heavy, and process execution becomes slow. This can effectively free up some resources in particular memory so remaining processes can have more resource. We will further discuss this issue in Chapters 9-10 (memory and virtual memory)
- The **short-term scheduler**, also referred as ***processor scheduler*** or ***CPU scheduler***, selects a process from the ready queue to run next and allocates CPU core to run it. The CPU scheduling will be examined in Chapter 5.
- **Concurrency** refers to the fact that, within a (short) period of time, there are multiple processes running on a CPU, thus making progresses all together during the time. However, at any time instant, there could be at most one process running if we have a single CPU core. In another word, **concurrency** implies that there are multiple processes, their executions are *interleaved* or *multiplexed* on the CPU, all of which can make progress within a period of time.
- The process defined here is referred to as a *heavy-weight process*, in which there is no concurrency within the execution of one process itself; in another word, there is only one thread of execution, and instructions are fetched and executed sequentially. For multi-threaded processes, different threads of the same process can execute *simultaneously* on different CPU cores in a multi-processor system.
- **Context switch** occurs when CPU switches from running one process to another, in which the OS must *save* the state of the old process and *load* the saved state from the new process. For instance, when there is an interrupt or system call. Context of a process is represented by its PCB. Context-switch time is an overhead, which varies from machine to machine and is highly dependent on the hardware support.

The Dual Mode

- The distinction between **kernel mode** and **user mode** distinguished by a *mode bit* provides basic means for **protection** in the following manner. Certain instructions (*privileged*) could only be executed when the CPU is executing in the kernel mode. In addition, hardware devices can only be accessed when executing in the kernel mode. Therefore, a user program has limited capability when executing in the user mode, hence enforcing protection of critical resources. Users rely on the services provided by operating system to access such resources. For example, the following operations are privileged and can only be executed in the kernel mode: to set value of timer, clear memory, disable interrupts, modify entries in device-status table, access I/O devices.
- Transitions from user mode to kernel mode can be caused by *system calls*, *interrupts*, *traps* and *exceptions*.

Operations on Process

- OS provides mechanisms (APIs along with the system calls) for *process creation* and

termination. In a multi-programmed system, processes execute concurrently, and they can be created and terminated dynamically.

- A process may create new processes, via system calls during the course of its execution. Please notice that any time a new program is executed, a child process is created. Once successful, the original process is referred to as a **parent process,** and the newly created process is called the **child process** of the original process.
- There are a number of actions that must be taken during a process creation: constructing a new PCB for the new process (i.e., create PCB data structure and allocate a unique process ID for the process), allocating memory, copying data from the parent process and I/O state including all opened files if any, and etc.
- Once a child process is successfully created, it executes ***concurrently*** with other processes, possibly with a different program (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 a UNIX/Linux system call that creates a process. This design might seem to be odd for historical reasons. In the (old) days when this was designed, there were not many concurrent running processes in a system and each process did not require a large memory space. fork() expedites process creation by simply copying the entire address space (code and data) from its parent process.
- There is no parameter required when calling fork(), but fork() has return values.
- If fork()system call fails, no child process is created, a ***negative value*** is returned to the original process (the return value from the system call).
- If fork() is successful, a ***zero value*** is returned to the newly created child process, and the ***child process ID*** (i.e., a positive number) is returned to the parent process. Part of the reason that two different return values are needed is that the next instruction right after fork() instruction will be executed by both parent and child processes concurrently and independently (the "odd" part). In another word, unless the child process calls exec() system call to upload a new program to execute, the child process and the parent process will execute the identical codes after the fork() (since the program counters or PCs are also the same). Therefore, this return value can be 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 of its child process, which is useful in process termination.
- The codes right after fork() are executed by both the original process (the parent process) and the newly created process (the child process). This is because the child process duplicates the entire address space of the parent including registers and program counter from the parent process. The child process must explicitly load a new program to run by calling system calls exec() or execlp() if it desires so.
- These two processes are concurrently executing. Either the parent or the child can run first (depending on the *short-term scheduler* while they are both on the ready queue). Their executions can even be interleaved with each other, along with other

processes in the system (Chapter 5 will discuss the CPU scheduling). In another word, a process itself does not control when it runs (i.e., the order of the execution) and how long it runs each time when it occupies the CPU.

Process Termination

- When a process terminates, its resources are deallocated by the OS. However, its entry in the Process Table (or Process List) remains until the parent calls `wait()`, as the process table contains the process's exit status. A process that has terminated, but whose parent has not yet called `wait()`, is known as a **zombie process**. In Unix, it does not need to do anything as the parent is expected to call `wait()` later to ask OS to collect resources, i.e., to delete the process identifier of the zombie process and its entry in the process table are released. All processes transition into this state briefly before termination, but generally they exist as zombies only briefly.
- If a parent process terminates itself without first calling `wait()`, its children are referred to as **orphan processes**. In UNIX/Linux, the `init()` or `systemd()` is assigned as a parent that periodically invokes `wait()`, allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and the corresponding entry in the process table.

Inter-Process Communication (IPC)

- Processes can be *independent* from or *cooperating* with other processes. The cooperation can bring several benefits 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 two processes. While in a message-passing system, there are explicit message(s) delivered. The Mach OS uses message passing as its primary form of interprocess communications. Windows also provides a form of message passing.
- Message-passing 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 usually sent to a mailbox, no need to reveal the identity of the sender or the receiver.
- In a message-passing system, communications between processes take place through system calls to `send()` and `receive()` primitives. The messages exchanged by communicating processes reside in a temporary queue, which can be implemented in three ways: (1) **zero capacity** – no messages are queued on a link; thus, sender must wait for receiver (rendezvous); (2) **bounded capacity** – finite number of messages, and sender must wait if link is full; (3) **unbounded capacity** – infinite length, in which sender never waits.

Pipes

- **Ordinary pipes** allow two processes to communicate in a standard *Producer and Consumer* manner. 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* – meaning the communication can only take place in one direction from a sender to a receiver.
- 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.
- In 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.
- Pipe makes Unix attractive in that it enables separate processes to communicate without being designed explicitly to work together. This allows tools limited in the functionality to be combined in complex ways. For instance, ls | grep x, in which shell run both commands, connecting the output of the first command to the input of the second command. Specifically in this example, **ls** produces a list of files in the current directory, while the **grep** reads the output of **ls** and prints only those lines containing the letter x.
- **A named pipe** is a more powerful tool, which are *bidirectional* and does not require parent-child relationship. A named pipe allows multiple processes to use it for communications and multiple processes can write to it.
- Named 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.
- A named pipe continues to exist after the processes using it have been terminated, while an ordinary pipe ceases to exist if the processes using the pipe have been terminated. In another word, named pipes must be explicitly deleted or destroyed.
- Windows systems also provide two forms of pipes—***anonymous*** and ***named pipes***. Anonymous pipes are similar to UNIX ordinary pipes. They are unidirectional and employ parent–child relationships between communicating processes.

Client and Server Communications

- Communication in a client-server system may use (1) sockets and (2) RPC. The key features in a client-server communication are: (1) the server is always available with a well-known address (name, and port number), waiting for clients to contact (e.g., a Web server), and (2) communication can only be invoked by clients (e.g., a browser) to initiate contact to a server.
- 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 a *client-server system*, only clients can initiate the communications to servers. The server with a well-known port number is always running and waiting for clients to contact it for services. For instance, Web servers and database servers.
- **RPCs** abstract the concept of function or procedure calls in such a way that a

function is invoked on another process that may reside on a separate computer. The Android OS uses RPCs as a form of interprocess communication.

- The RPC system hides the underlying details from the involving process that allow communication to take place by providing **stubs** on both sides: (1) the client-side stub locates the port on the server and marshalls parameters (in proper format), then transmits a message to the server using message passing; (2) the server-side stub receives this message, unpacks the marshalled parameters and performs the procedure on the server. If necessary, return values are passed back to the client.