Informally, process is a programm in execution. The status of the current activity of a process is represented by the valve of the programm counter and the contents of the processor's registers. The memory layout of a process is typically divided into multiple sections.

- **Text section**—the executable code
- **Data section**—global variables

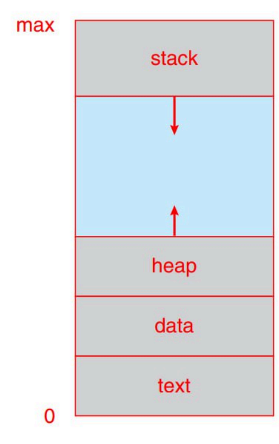

Figure 3.1 Layout of a process in memory.

- **Heap section**—memory that is dynamically allocated during program run time
- **Stack section**—temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)

If a text and data sections are fixed size, the stack and heap can shrink and grow dynamically during programm execution. The stack and heap sections grow toward one another, the OS must ensure that they do not overlap one another.

A program is not process by it self - it's a passive entity such as file containing instructions stored on disk.

In contrast, a process is an active entity. A program becomes a process when an executable file is loaded into memory. Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequenses.

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process.

- **New**. The process is being created.
- **Running**. Instructions are being executed.
- **Waiting**. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready**. The process is waiting to be assigned to a processor.
- **Terminated**. The process has finished execution.

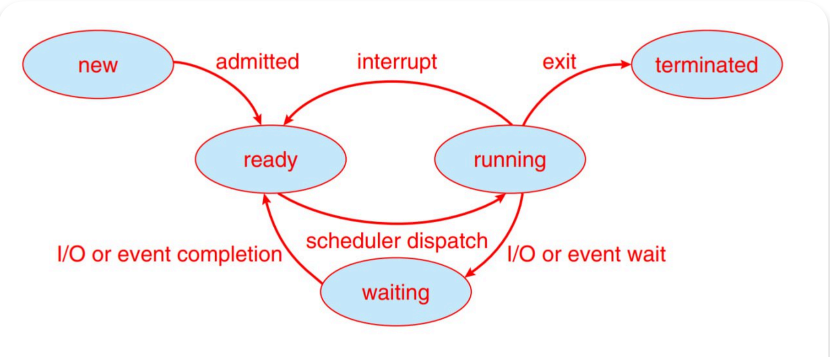

Figure 3.2 Diagram of process state.

Each process is represented in the OS by a process control block.

- **Process state**. The state may be new, ready, running, waiting, halted, and so on.
- **Program counter**. The counter indicates the address of the next instruction to be executed for this process.

Figure 3.3 Process control block (PCB).

- **CPU registers**. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward when it is rescheduled to run.
- **CPU-scheduling information**. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters. (Chapter 5 describes process scheduling.)
- **Memory-management information**. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system (Chapter 9).
- **Accounting information**. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information**. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

The objective of multiprogramming is to have some process running at all times so as to maximaze CPU utilization. The objective of time sharing is to switch a CPU core among processes so frequently that users can interact with each program while it's running. The process scheduler selects an avaible process for program execution on a core. Each CPU can run one process at time. The number of processes currently in memory is known as the degree of multiprogramming.

In general, most processes can be describe as either I/O bound or CPU bound. An I/O bound process is one that spends more of its time doing I/O than it spends doing computations. A CPU-bounded process generates I/O infrequently, using more of its time doing computations.

As processes enter the system, they are put into a ready queue, where they are ready and waiting to execute on CPU's core. A ready-queue header contains pointers to the first PCB in the list, and each PCB includes a pointer field that points to the next PCB.

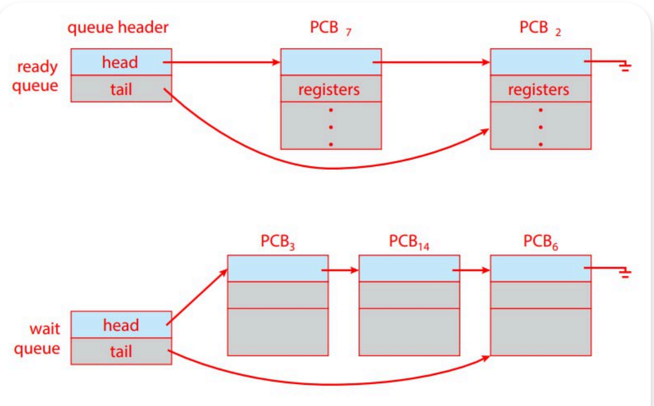

Figure 3.4 The ready queue and wait queues.

The system also includes other queues. When a process is allocated a CPU core, it executes for a while and eventually terminates, is interrupted, or waits for the occurence of a particular event (I/O for example). Processes that are waiting for a certain event to to occur are placed in a wait queue.

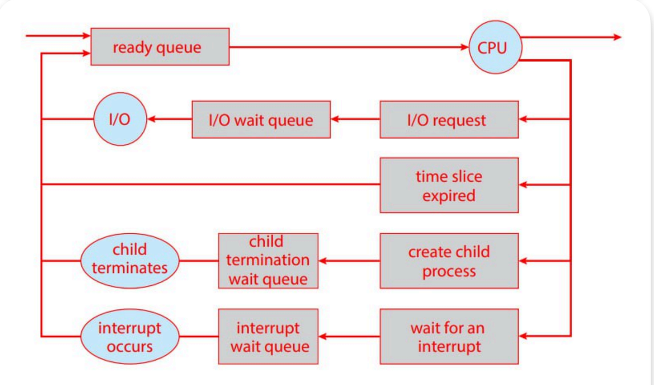

Figure 3.5 Queueing-diagram representation of process scheduling.

A ready process waits in ready queue until it is selected for execution, or dispatched. Once the process is allocated a CPU core and is executing, one of several events could occur:

- The process could issue an I/O request and then be placed in an I/O wait queue.
- The process could create a new child process and then be placed in a wait queue while it awaits the child's termination.
- The process could be removed forcibly from the core, as a result of an interrupt or having its time slice expire, and be put back in the ready queue.

The role of the CPU scheduler is to select from among the processes that are in the ready queue and allocate a CPU core to one of them. and must select a new process for the CPU frequently. It's likely designed to forcibly remove the CPU from a process and schedule another process to run.

Some OS have an intermediate form of scheduling, known as swapping, whose key idea is that sometimes it can be advantegeous to remove a process from memory (and from active contention for the CPU) and thus reduse the degree of multiprogramming. Later the process can be reintrodused into memory, and it's execution can be continued where it left off.

Interrupts cause the OS to change a CPU core form its current task and run a kernel routine. The system needs to save the current context of the process running on the CPU core so that it can restore that context when it's processing is done, essentially suspending the process and then resume it. The context is represented in the PCB of the process. It includes the valve of the CPU registers, the process state and memory-managment information. Generally, we perform a state save of the current state of the CPU core and then a state restore to resume operations.

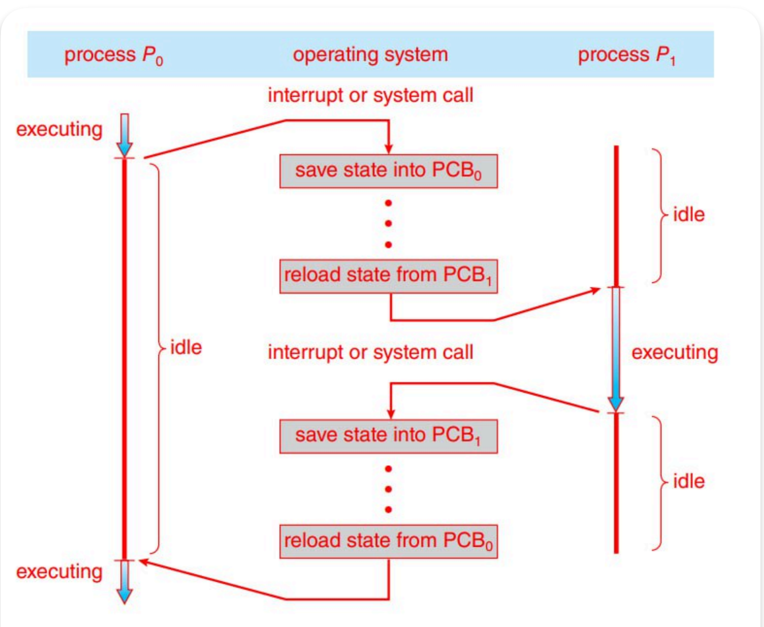

Figure 3.6 Diagram showing context switch from process to process.

Switching the CPU core to another process requires perfoming the state save of the current process and state restore of a different process. This task is known as a context switch.

Context-switch times are highly dependent on hardware support.

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus OS must provide a mechanism for process creation and termination.

A process can create new several processes.
parent — child of parents

Most OS identify processes according to a unique process identifier PID. Each of processes may in turn create other processes, forming a tree of processes.

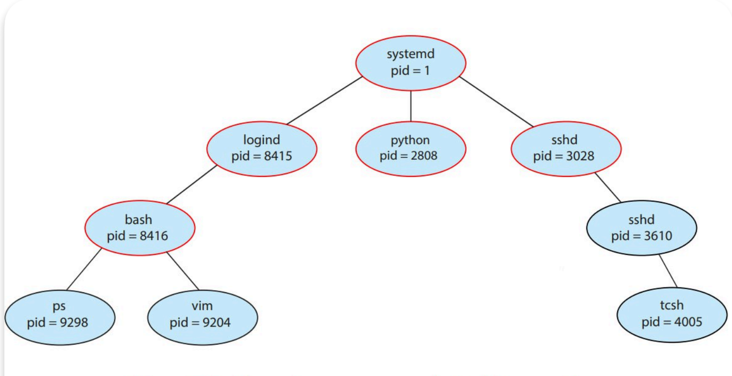

Figure 3.7 A tree of processes on a typical Linux system.

When a process creates a child process, that child will need certain resources (CPU time, memory, files etc) to accomplish its task. Child process may obtain resources directly from the OS, or it may be constrained to a subset of the resources of the parent process. The parent may partition it's resources among its children, or it may be able to share some resources among several of its children. Such restriction prevents any process from overloading the system by creating too many child processes. A parent process may pass along initialization data to the child process.

When a process creates a new process, two posibilities for execution exist:

1) The parent continues to execute concurrently with its children.
2) The parent waits until some or all of its children have terminated.

There are also two address-space posibilities for the new process:

1) The child process is a duplicate of the parent process (has the same program and data as the parent)
2) The child process has a new program loaded into it.

For instance in UNIX a new process is created by the fork() system call.

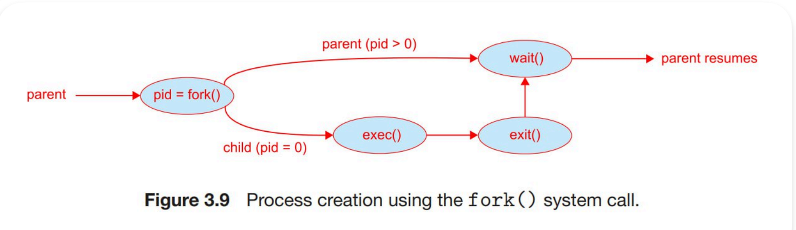

Figure 3.9 Process creation using the `fork()` system call.

A process terminates when it finishes executing it's final statement and asks the OS to delete it by using the exit() system call. At that point, the process may return a status valve to it's waiting parent process. All allocated resources are dealocated and reclaiming by the OS.

A parent may terminate the execution of one of its chidren for a variety of reasons, such as these:

- The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occured, the parent must have a mechanism to inspect the state of its children.)
- The task assigned to the child is no longer required.
- The parent is exiting, and the OS does not allow a child to continue if it's parent terminates

When a process terminates, it's resources are deallocated by the OS. However, it's entry in the process table must remain there until the parent calls wait(), because the process table contains the process's exit status. A process that has terminated, but whose parent hasn't yet called wait(), is known as a zombie process. Once the parent process calls wait(), the process identifier of the zombie process and its entry in the process table are released.

If a parent didn't invoke wait() and instead terminated, thereby leaving its child processes as orphans. Traditional UNIX systems has init process that invokes periodically invokes wait(), thereby allowing exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.