Process management

The main task of the OS (as a resource manager) is to give the CPU to a process that wants it. How does it know that a process wants the CPU?

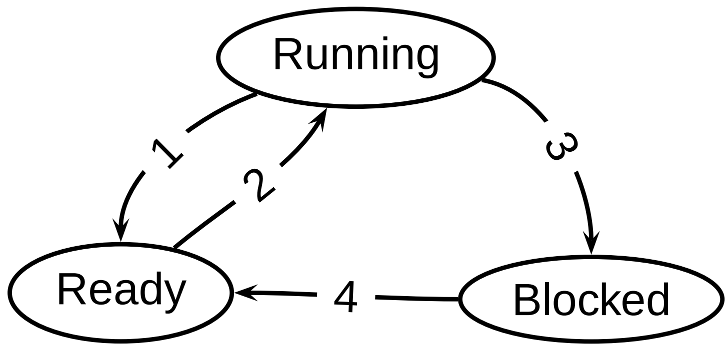
The OS keeps track of the current state of each process (usually in the PCB of the process). The state of a process changes as a result of the activities of the process itself or as a result of other events that originated elsewhere.

The basic states of a process are:

Running this process has control of the CPU.

Ready this process is ready to use the CPU but does not have control over it at this time.

Blocked this process is waiting for some external event to happen or for some service to be provided to it. At the present time it is not capable to use the CPU because it needs something else first.



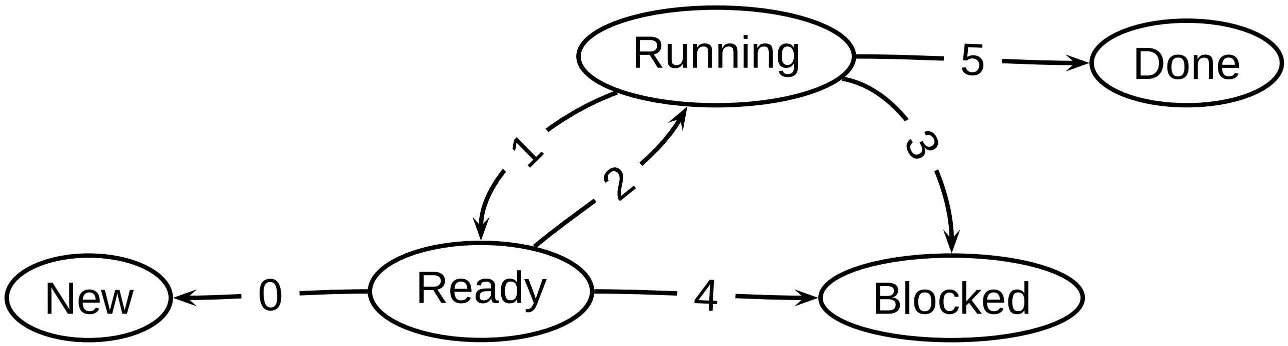
The transitions between states are:

1. A running process loses the CPU as a result of an interrupt. After processing this interrupt, the OS gives the CPU to another process.

2. The OS decided to give the CPU to a process and chose this one.

3. The process asked for something that could not be provided instantly (input/output, synchronisation with another process, etc.).

4. The process received whatever it was asking for.



The two new transitions are:

0 A new process was created by fork (or similar).

5 The running process called an exit or the execution of _exit was forced upon the process by a signal.

Process termination

Whether a process terminated willingly or not, chances are good that the process did not release all the resources it held.

These resources are released by the OS with the use of the u area. some resources cannot be released automatically (e.g. file locks) and may eventually create deadlocks.

_exit

The termination of a process is performed by a function called _exit; this function is called internally by exit and in many default signal handlers. It is also invoked directly by the SIGKILL signal.

_exit performs the following:

- closes any open file descriptors or handles.
- notifies the parent that the calling process is to be terminated if the parent process is blocked in a wait() or waitpid() call,
- If the parent process of the calling process is not inside a wait() or waitpid() function, _exit saves the exit status code for return to the parent process when the parent process calls wait() or waitpid().
- Terminating a process does not directly terminate its children. _exit assigns a new parent process ID (typically init, i.e. 1) to the children of a terminated process.
- A SIGCHLD is sent to the parent of the calling process. Note that the default action for SIGCHLD is SIG_IGN.

The zombies are coming to town

If the parent process has not expressed interest in the status of the (defunct) child, the child's PCB will have to be kept around until the parent dies (in case the parent issues a wait() call).

This may lead to the process lingering as a zombie for a long time (the state of such process is either defunct or plain zombie).

Note that init periodically calls wait() to dispose of orphaned zombies.

Worse than a zombie

_exit cancels any outstanding input/output if possible. If the operation is not cancellable (e.g. block i/o), it is executed as if the call to _exit had not yet occurred. Note that if the operation is not cancellable and does not complete, the exiting process will remain in a semi–zombie state forever, always claiming to be exiting (this state is marked **D**; it is labelled exiting).

There is no way to dispose of a process in this state because its wait for i/o is not interruptible. You cannot kill it and it will not go away on its own because it is never ready so it will never get the CPU.

If you are tired of seeing a **D** process around, all you can do is to reboot the system.

Other side effects

_exit also tries to get the exiting process out of any commitments it made to shared operations (semaphores, message queues, locks, etc.). This is not always possible to do it in a correct manner, so other processes may block forever as a result.

How does a process block?

A running process may want to get a resource. To get it, it must issue a system call to the OS. If the resource allocation requires some action, the process will lose the CPU and will have to wait until the resource is allocated or the request is denied.

Any process may be turned into a blocked process by an external signal SIGSTOP.

How does a process become ready?

A new process is not ready for execution until its creation stage (fork) is done. Then it becomes ready to use the CPU.

A process in progress (which already used some CPU time) becomes ready in two ways:

- It was running and the CPU was taken away from it. This happens in all contemporary systems: the CPU is allocated to a process for a ti,e slice after which the cpu is taken away and given to another process. The part of the OS which manages CPU allocation is called a CPU scheduler.
- A blocked process is waiting for a resource or an event. When that resource is granted or the event takes place, the process no longer has any reason to wait. It then becomes ready again and becomes eligible to get the CPU.

Note that eligible means just that: there is no guarantee that a process will get the CPU immediately after becoming ready.

Note also that a blocked process may be waiting for more than one condition to become true before it may continue. All these conditions must be met before the process becomes ready again.