ECE254: Operating Systems and Systems Programming Spring 2015

Lecture 5 — Process State

Jeff Zarnett

Process State

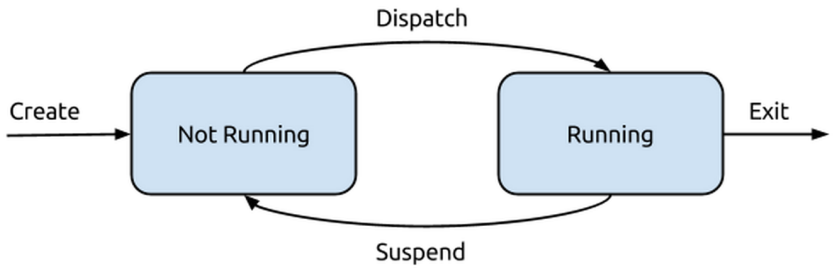
The OS is responsible for determining which programs run when and how to allocate resources. The current state of the process is therefore important information. To maintain the state of the process the PCB has a variable, but we will think about this as a finite state machine (FSM): there are a limited number of states and defined transitions between them.

The Two-State Model

Let us begin with the simplest possible model: the two state model. At any time, either a process is executing, or it is not. Thus we have the two states:

1. **Running:** Actively executing right now.
2. **Not Running:** Not currently executing.

When a new process is created, the PCB is allocated and the state of the process is Not Running. Whatever process is currently running, and when the process that currently running is finished or a process switch takes place, the running program's state is changed to Not Running, a new process is selected to run (according to scheduling), and it is put in the Running state while it executes.



State diagram for the two state model.

There are the following transitions in the diagram:

- **Create:** The process is created and enters the Not Running state.
- **Dispatch:** A process that is not currently running begins executing and moves to the Running state.
- **Suspend:** A running program pauses execution, for whatever reason, and moves to the Not Running state.
- **Exit:** A running program finishes, reaching the `exit`, and can be removed from the list of processes.

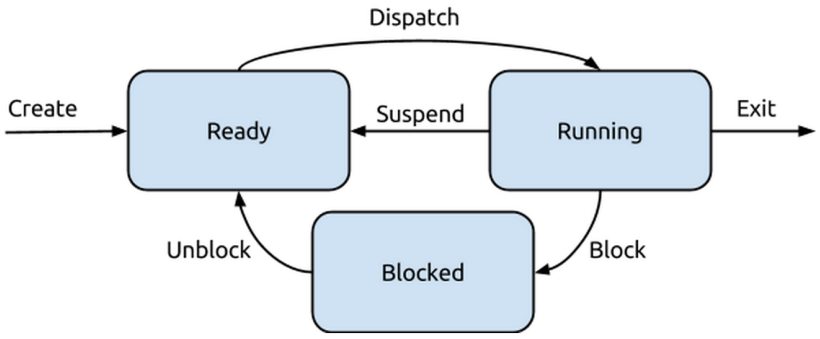
Thes two-state model, is, however, inadequate. It assumes that every process is constantly ready to run, which is not a safe assumption. We will need a way to indicate that a process is not ready to run, and thus a third state.

The Three-State Model

A program that requests a resource like I/O or memory may not get it right away. This is not to say the program will never get it, just that it does not have it right now. Sometimes the program needs user input, and as far as the computer is concerned, the user moves at glacial speed. In any case, the program wants to continue but cannot until it gets what it is waiting for. If the scheduler picks a process that is waiting for user input, nothing will be happening while the program is waiting for input, so the CPU's time would be wasted. Thus, we should be able to mark a process as "not ready to proceed", and we then go to a three-state model:

1. **Running:** Actively executing right now.
2. **Ready:** Not running, but ready to execute if selected by the scheduler.
3. **Blocked:** Not running, and not able to run until some event happens.

This new state, Blocked, indicates that the program is not ready to run for lack of a resource. It may also be called the Waiting state. The scheduler will not choose a Blocked process to run, even if the CPU has nothing else to do. Suppose process P_n is waiting for user input. When the user input is received, an interrupt is generated and the interrupt handler runs. The handler takes the input from the I/O device (keyboard), delivers it to P_n, then moves the state of P_n to Ready.



State diagram for the three-state model.

There are six transitions in the diagram:

- **Create:** The process is created and enters the Ready state.
- **Dispatch:** A process that is not currently running begins executing and moves to the Running state.
- **Suspend:** A running program pauses execution, but can still run if allowed, and moves to the Ready state.
- **Exit:** A running program finishes, reaching the `exit`, and can be removed from the list of processes.
- **Block:** A running program requests a resource, does not get it right away, and cannot proceed.
- **Unblock:** A program, currently blocked, receives the resource it was waiting for; it moves to the Ready state.

Though this three state model is good, it does not encompass everything we have already discussed, such as a zombie process. We can still improve on it by adding two additional states.

The Five-State Model

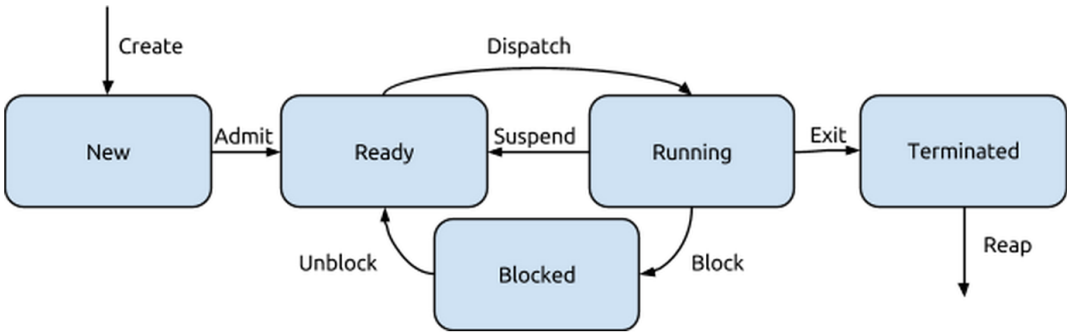
Earlier we discussed that a UNIX process may be finished but a zombie, because its parent has not (yet) come by to collect its return value. The process is not ready to run (it has finished running) and it is not waiting for a resource, so it does not qualify for Ready or Blocked. Thus, we need a state to represent that it is finished but not yet cleaned up: Terminated.

The fifth and final state will be the "New" state: a process that has just been defined. Suppose a user wants to run a new process. The OS will first perform the necessary administrative tasks: define an identifier for the process, instantiate the PCB object, and put the process in the New state. The OS has created the process but has not committed itself to execution thereof. This may be because the system limits the number of concurrent processes for performance reasons. When the process is in the New state is is typically not in memory, but on disk instead [Sta14].

Thus, with the two new states added, the five states of a process in the system are:

1. **Running:** Actively executing right now.
2. **Ready:** Not running, but ready to execute if selected by the scheduler.
3. **Blocked:** Not running, and not able to run until some event happens.
4. **New:** Just created but not yet added to the list of processes ready to run.
5. **Terminated:** Finished executing, but not yet cleaned up (reaped).

With five states, we will have significantly more transitions between the states. The diagram below shows the five-state model:



State diagram for the five-state model.

There are now eight transitions, most of which are similar to what we have seen before:

- **Create:** The process is created and enters the New state.
- **Admit:** A process in the New state is added to the list of processes ready to start, in the Ready state.
- **Dispatch:** A process that is not currently running begins executing and moves to the Running state.
- **Suspend:** A running program pauses execution, but can still run if allowed, and moves to the Ready state.
- **Exit:** A running program finishes and moves to the Terminated state; its return value is available.
- **Block:** A running program requests a resource, does not get it right away, and cannot proceed.

- **Unblock:** A program, currently blocked, receives the resource it was waiting for; it moves to the Ready state.
- **Reap:** A terminated program's return value is collected by a `wait` and its resources can be released.

There are two additional "Exit" transitions that may happen but are not shown. In theory, a process that is in the Ready or Blocked state might transition directly to the Terminated state. This can happen if a process is killed, by the user or by its parent (recall that parent processes can generally kill their children at any time, something the law thankfully does not permit). It may also happen that the system has a policy of killing all the children of a parent process when the parent process dies.

References

[Sta14] William Stallings. *Operating Systems Internals and Design Principles (8th Edition)*. Prentice Hall, 2014.

[Tan08] Andrew S. Tanenbaum. *Modern Operating Systems, 3rd Edition*. Prentice Hall, 2008.