Lecture 5 — Process State

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

August 21, 2016

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.

To maintain the state, there is a variable in the PCB.

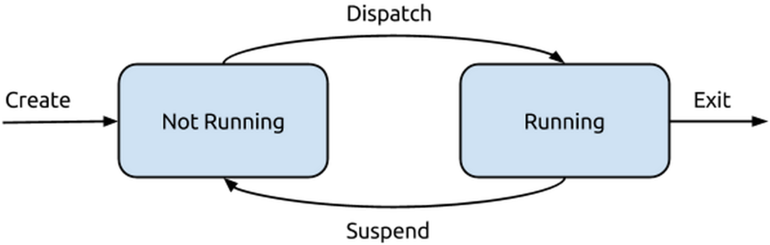
We will think of this as a finite state machine (FSM).

Two State Model

Let us begin with the simplest possible model: 2 states.

1. **Running**
2. **Not Running**

Two State Model



Two State Model

There are the following transitions in the diagram:

- **Create**
- **Dispatch**
- **Suspend**
- **Exit**

Two State Model

This simple model is inadequate.

It assumes every process is constantly ready to run.

We need a way to indicate a process is not ready to run...

Three State Model

A program that requests a resource may not get it right away.

This is not to say the program will never get it.
It's just that it does not have it right now.

The program wants to continue but can't until it gets what it's 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.
The CPU's time would be wasted.

Third state: "not ready to proceed".

Three State Model

1. **Running**
2. **Ready**
3. **Blocked**

The scheduler will not choose a blocked process.

Three State Model

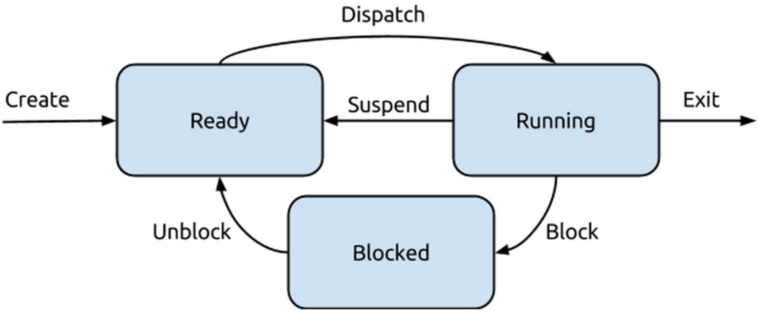
Processes get blocked when they are *waiting* for something.

Suppose process P_n is waiting for user input.

When the user input is received, an interrupt is generated.

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”.

Three State Model



Three State Model

There are six transitions in the diagram:

- **Create**
- **Dispatch**
- **Suspend**
- **Exit**
- **Block**
- **Unblock**

Five State Model

The three state model is good, but it does not cover things like zombies.

Life pro tip: the character who doubts that zombies are real dies first.

A UNIX process may be finished but its value yet uncollected.

It is not ready to run, but not waiting for a resource either.

New state to add, then: “Terminated[1]” – finished but not cleaned up.

[1] Say this in Arnold Schwarzenegger’s voice

Five State Model

That accounts for four states; what about the fifth?

The fifth is the “New” state: just created.

If the user creates a process, the OS has significant work to do.

- Define an identifier.
- Instantiate the PCB.
- Put the process in the New state.

The process is defined, but the OS has not started it yet.

Five State Model

Why bother with the “New” state?

The system may limit the number of concurrent processes.

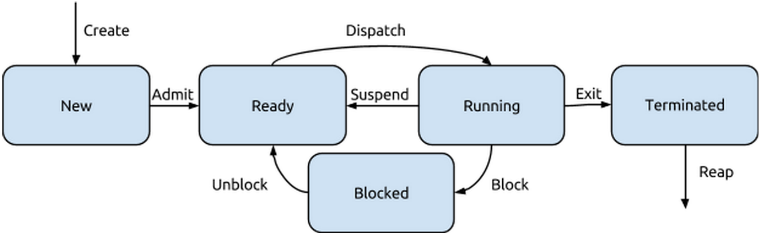
New processes are typically on disk and not in memory.

Five State Model

Thus, with the two new states added, the five states are:

1. **Running**
2. **Ready**
3. **Blocked**
4. **New**
5. **Terminated**

Five State Model



Five State Model

There are now eight transitions:

- **Create**
- **Admit**
- **Dispatch**
- **Suspend**
- **Exit**
- **Block**
- **Unblock**
- **Reap**

Five State Model

There are two additional exit transitions that are not shown.

A process can go directly from “Ready” or “Blocked” to “Terminated”.

This happens if a process is killed.

More is the New More

We can expand on the five state model by considering disk.

A process maybe swapped out to disk rather than in main memory.

If an executing process gets blocked, maybe swap it to disk.

Users often want more processes running than fit in memory.

The problem is not the PCBs, but stack & heap space of the programs.

Don’t Even Start...

A (non-)solution: do not start a program when insufficient memory.

Programs do not declare how much memory they will use.

They may not even know themselves – how could Microsoft Word know what document the user plans to open and edit?

Imagine being told a program cannot launch due to low memory.
  User complaints about “random” refusal to launch.

Let’s Throw Money at the Problem!

Another (non-)solution, then: buy more RAM.

Memory gets too expensive beyond a certain point.

But: “programs expand to fill the RAM available”.

Computers might have 1000× more RAM than it did 20 years ago, but programs today use 1000× more RAM than the programs of 20 years ago.

Increase the size of memory leads to larger, not more, processes.

The Last Resort: Disk

With no other place to put them, we have to put some processes on disk.

This is what we know as swapping.

When the demands for memory exceed the available memory, some of the processes will be moved to disk storage to make room.

This is a notably expensive operation.

Swapping to Disk

Swapping a process to disk might mean transferring several hundred megabytes of data, or even a few gigabytes.

This, from the perspective of the CPU, takes about seven eternities.

Then, when that process is going to run again, we need to load it back in to memory, which will take just as much time as it took to flush it out.

So this is something to be done only when necessary.

The Sixth Sens...State

We do not want to spend any more time swapping the process in and out of memory than is necessary.

We need to know if a particular process is in memory or on disk.

Thus we need a new state: swapped.

Ideally, we will only swap a process to disk if it is blocked.

A process swapped to disk then enters that sixth state, swapped.

- It is blocked and not in main memory.

Lucky Number Seven

There are two scenarios that tell us that six is not sufficient.

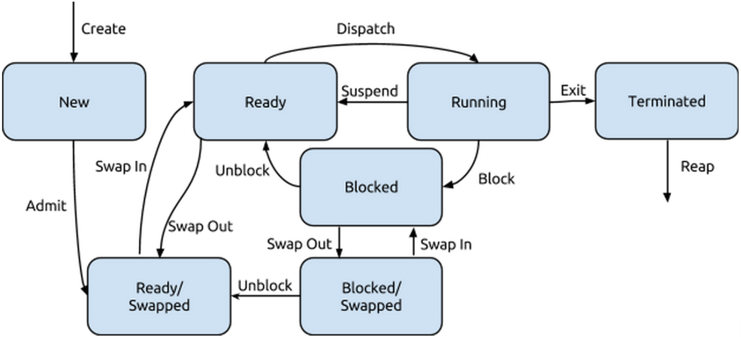
1. What if all processes are ready, but there is a memory space shortage?

2. What if the event a swapped process waited for took place?

Avoid bringing in a swapped process if it is just going to be blocked.

Solution: split swapped into “Ready/Swapped”and “Blocked/Swapped”.

Seven State Model



Seven State Model

A variant of the five state model.

The “Admit” transition is modified: by default the new process does not start in main memory.

Two new transitions: “Swap In” and “Swap Out”.

A second “Unblock” transition.

As in the five state model, some additional “Exit” transitions.

A Preview of Scheduling

Processes, represented by their PCB, are maintained in a data structure; typically a linked list or a queue of some sort.

For efficiency reasons, it would not make sense to have all processes in one single linked list that we would have to iterate over.

If there are 200 processes and the next ready one is number 137...

So it is logical to take non-ready processes out of this list.

E Pluribus... Pluribum?

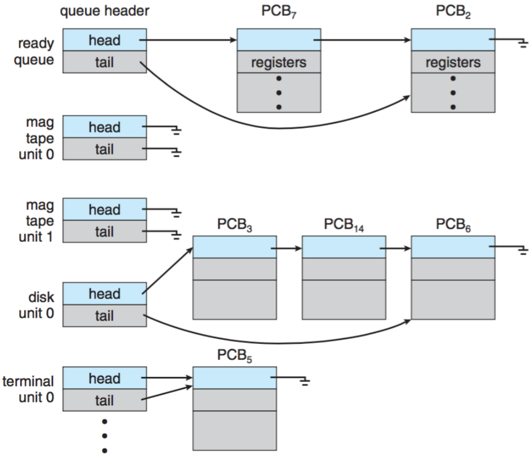
When a process is in some state like “Blocked”, where does it go?

This is a question we will come back to when we discuss scheduling.

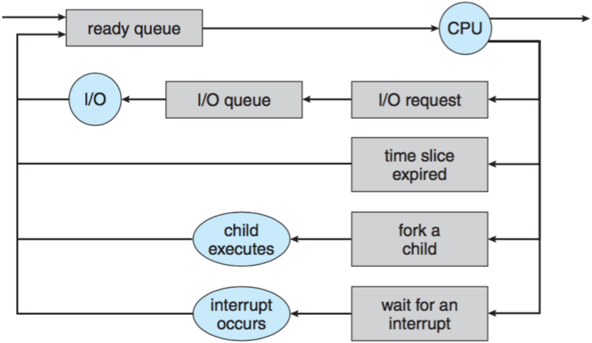
But for now, it’s convenient for the operating system to keep track of what processes are in which states separately.

If a process is waiting for a disk operation, when the disk becomes available, it would be convenient to know quickly what processes are waiting for the disk.

PCBs in Queues



Queueing Diagram


ready queue
CPU
I/O
I/O queue
I/O request
time slice expired
child executes
fork a child
interrupt occurs
wait for an interrupt