Lecture 4 — Processes

Prepared by Jeff Zarnett, taught by Seyed Majid Zahedi
jzarnett@uwaterloo.ca, smzahedi@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

Processes

Early computers did exactly one thing.

Or at least, exactly one thing at a time.

Now the OS supports multiple programs running concurrently.

To manage this complexity, the OS uses the process.

The Process

A process is a program in execution.

1. The instructions and data.
2. The current state.
3. Any resources that are needed to execute.

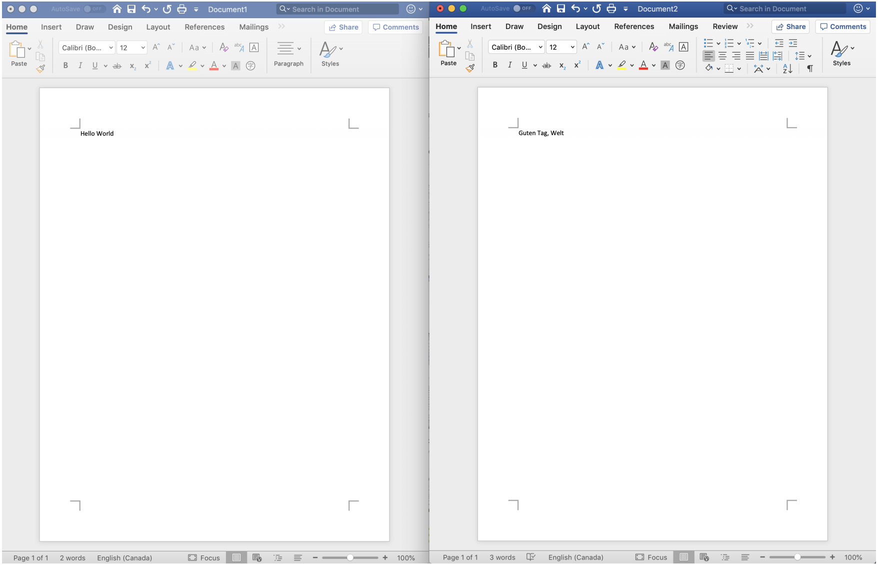
Processes vs. Programs

Note: two instances of the same program running equals two processes.

You may have two windows open for Microsoft Word, and even though they are the same program, they are separate processes.

Similarly, two users who both use Firefox at the same time on a terminal server are interacting with two different processes.

Two Documents, Two Processes


Document1
Home Insert Draw Design Layout References Mailings
Share Comments
Calibri (Bo... 12
Paste Paragraph Styles
Page 1 of 1 2 words English (Canada) Focus 100%
Document2
Home Insert Draw Design Layout References Mailings Review
Share Comments
Calibri (Bo... 12
Paste Styles
Page 1 of 1 3 words English (Canada) Focus 100%

Hello World

Guten Tag, Welt

Process Control Block

Data structure for managing processes: Process Control Block (PCB).

It contains everything the OS needs to know about the program.

It is created and updated by the OS for each running process.

It can be thrown away when the program has finished executing and cleaned everything up.

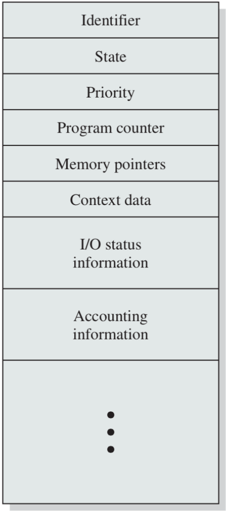
The blocks are held in memory and maintained in some container (e.g., a list) by the kernel.

Process Control Block

The process control block will (usually) have:

- **Identifier.**
- **State.**
- **Priority.**
- **Program Counter*.**
- **Register Data*.**
- **Memory Pointers.**
- **I/O Status Information.**
- **Accounting Information.**

Process Control Block (Simplified)



Identifier

State

Priority

Program counter

Memory pointers

Context data

I/O status information

Accounting information

•
•
•

Process Control Block

This data is kept up to date constantly as the process executes.

The program counter and the register data are asterisked.

When the program is running, these values do not need to be updated.

They are needed when a system call (trap) or process switch occurs.

PCB: Asterisked Fields

We will need a way to restore the state of the program.

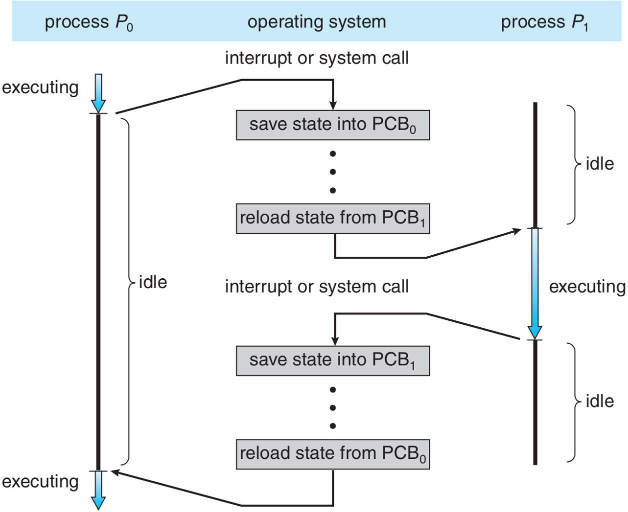
Save the state of the process into the PCB.

1. Save the state into the Program Counter variable.

2. Save the Register variables.

Process Switch

A switch between the execution of process P_0 and process P_1:



The Circle of Life

Unlike energy, processes may be created and destroyed.

Upon creation, the OS will create a new PCB for the process.
- Also initialize the data in that block.

Set: variables to their initial values.
- the initial program state.
- the instruction pointer to the first instruction in `main`

Add the PCB to the set.

After the program is terminated and cleaned up:
- Collect some data (like a summary of accounting information).
- Remove the PCB from its list of active processes and carry on.

Process Creation

Three main events that may lead to the creation of a process:

1. System boot up.
2. User request.
3. One process spawns another.

Process Creation: At Boot Up

When the computer boots up, the OS starts and creates processes.

An embedded system might have all the processes it will ever run.

General-purpose operating systems: allow one (both) of the other ways.

Process Creation: At Boot Up

Some processes will be in the foreground; some in the background.

A user-visible process: log in screen.

Background process: server that shares media on the local network.

UNIX term for a background process is Daemon.

Example: `ssh` (Secure Shell) command to log into a Linux system.

Process Creation: Users

Users are well known for starting up processes whenever they feel like it.

Much to the chagrin of system designers everywhere.

Every time you double-click an icon or enter a command line command (like `ssh` above) that will result in the creation of a process.

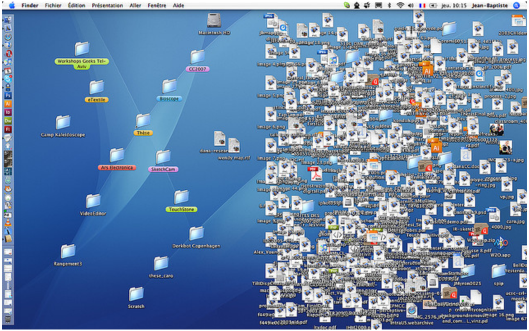


Image Credit: CS Tigers

Process Creation: Spawning

An already-executing process may spawn another.

E-mail with a link? Click it; the e-mail program starts the web browser.

A program may break its work up into different logical parts.
To promote parallelism or fault tolerance.

The spawning process is the parent and the one spawned is the child.

Process Destruction

Eventually, most processes die.

This is sad, but it can happen in one of four ways:

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal Error (involuntary)
4. Killed by another process (involuntary)

Process Destruction: Normal

Most of the time, the process finishes because they are finished.
Or the user asks them to.

Compiler: when compilation is finished, it terminates normally.

You finish writing a document in a text editor, click the close button.

Both examples of normal, voluntary termination.

Process Destruction: Error

Sometimes there is voluntary exit, but with an error.

Required write access to the temporary directory & no permission.

Compiler: exit with an error if you ask it to compile a non-existent file.

The program has chosen to terminate because of the error.

Process Destruction: Fatality

The third reason for termination is a fatal error.
Examples: stack overflow, or division by zero.

The OS will detect this error and send it to the program.

Often, this results in involuntary termination of the offending program.

A process may tell the OS it wishes to handle some kinds of errors.

If it can handle it, the process can continue.

Otherwise, unhandled exceptions result in involuntary termination.

Process Destruction: Killed

The last reason for termination: one process might be killed by another.
(Yes, processes can murder one another. Is no-one safe?!).

Typically, this is a user request:
a program is stuck or consuming too much CPU...
the user opens task manager (Windows) or `ps` (UNIX)

Programs can, without user intervention, theoretically kill other processes.

Example: a parent process killing a child it believes to be stuck.

Process Destruction: Killed

There are restrictions on killing process.

- A user or process must have the rights to execute the victim.

Typically, a user may only kill a process he or she has created.

Exception: system administrator.

While killing processes may be fun, it is something that should be reserved for when it is needed.

Maybe when a process is killed, all processes it spawned are killed too.

Process Family Tree

In UNIX, but not in Windows, the relationship between the parent process and child process(es), if any, is maintained, forming a hierarchy.

A process, unlike most plants and animals, reproduces asexually.

A process has one parent; zero or more children.

A process and all its descendants form a process group.

Certain operations like sending a signal can be sent to a whole group.

Process Family Tree: UNIX

UNIX the first process created is called `init`.

It is the parent of all processes (eventually).

Like the `Object` class in Java is the superclass of all classes.

Thus, in UNIX we may represent all processes as a tree structure.

Process Family Tree: Windows

In Windows, a parent process gets a reference to its child.

This allows it to exercise some measure of control over the child.

This reference may be given to another process (adoption).

No real hierarchy. A process in UNIX cannot disinherit a child.

Process Return Codes

When a process terminates, it does so with a return code.

Just as a function often returns a value.

On the command line or double-clicking an icon, return value is ignored.

In UNIX, a parent can get the code that process returns.

Usually, a return value of zero indicates success.

Other values indicate an error of some sort.

Normally there is some sort of understanding between the parent and child processes about what a particular code means.

The Walking Dead

Aaah! Zombies! Run!

When a child process finishes, until the parent collects the return value, the child continues in a state of “undeath” we call a zombie.

This does not mean that the process then shuffles around the system attempting to eat the brains of other processes.

It just means that the process is dead but not gone.

There is still an entry in the PCB list.
    And the process holds on to its allocated resources.

Only after the return value is collected can it be cleaned up.

Getting Rid of Zombies

Usually, a child process's result is eagerly awaited by its parent.

The `wait` call collects the value right away.

This allows the child to be cleaned up (or, more grimly, "reaped").

If there is some delay for some reason, the process is considered a zombie until that value is collected.

Orphans

If a parent dies before the child does, the child is called an orphan.

In UNIX any orphan is automatically adopted by the `init` process.

…making sure all processes have a good home.

By default, `init` will just `wait` on all its child processes

And do nothing with the return values.

A program can be intentionally orphaned: to run in the background.

This would be cruel, except that processes, as far as anyone knows, do not have feelings.

Five State Model

As you might imagine, at any given time, a process is running or not running.

The first two states of the model are therefore “Running” and “Ready”.

A program that requests a resource like I/O or memory may not get it right away. This gives us the “Blocked” state.

Five State Model

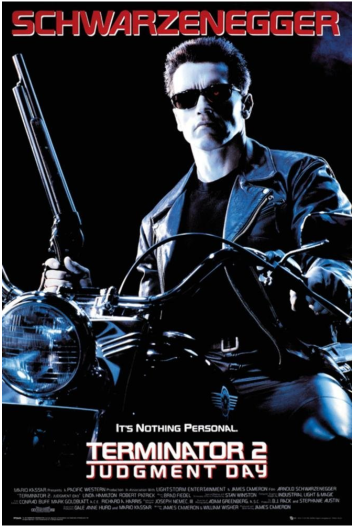
But we did not cover things like zombies.

- 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.

Come with me if you want to live



New state: Terminated.

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.

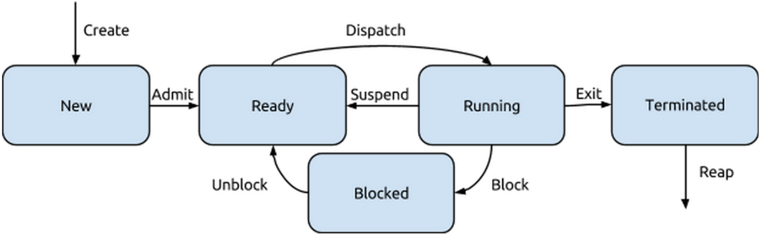
New processes are typically on disk and not in memory.

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 may not be entirely in memory!

Users often want more processes running than fit in memory.

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

The Last Resort: Disk

With no other place to put them, the OS puts part of processes on disk.

This is what we know as swapping.

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

This is a notably expensive operation.

Swapping to Disk

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

If the process needs it again, loading it back takes 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

The OS does 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.

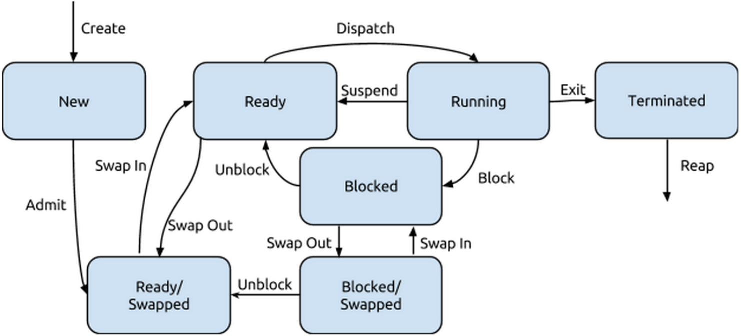
1. What if all processes are ready, but there is a memory space shortage?

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.