**Leandro Proença**
Posted on 12 de jul. de 2022 • Updated on 13 de jul. de 2022



# A brief history of modern computers, multitasking and operating systems

#unix #linux #operatingsystems #threads

**Computers, OS and networking (3 Part Series)**

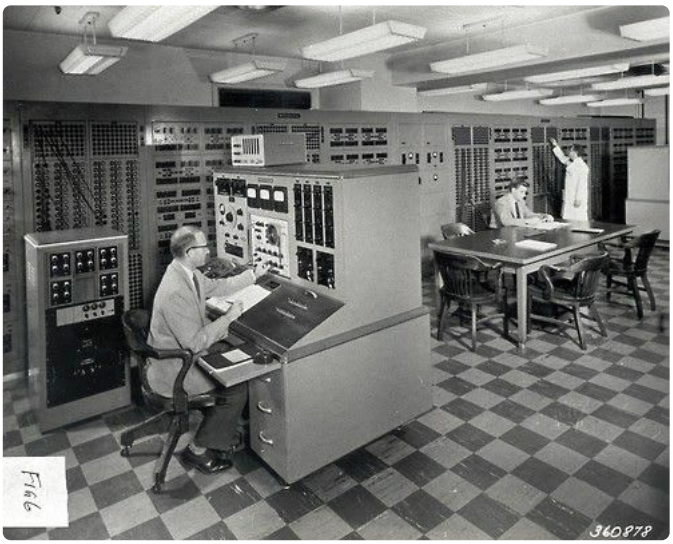
1. **A brief history of modern computers, multitasking and oper…**
2. Inter-process communication: files
3. Inter-process communication: pipes

In this article I'll try to write a brief history of modern computers, multitasking and how operating systems tackle concurrency.

## 40s

The first *modern* computers in the 40's were capable of running programs built in punched cards.

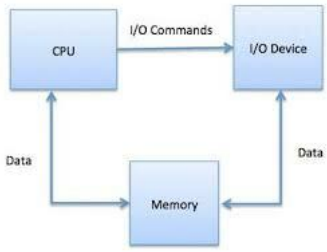
At that moment, computers used to load **one program at a time**, and because of very modest speeds, programs would took *days* to finish, thus leading to long *waiting queues* for programmers.



## 50s

A decade later, computers became a little faster and as such they could **enqueue multiple programs at once** using a FIFO system (first-in, first-out).

Despite of the importance of enqueuing jobs and solving the *programmers waiting queues*, programs would still **run one at a time**, meaning that when a specific program gets blocked on *input or output* (*i.e* waiting the printer to finish printing the output), the **CPU gets idle**.




In other words, there's no efficiency on the computer physical resources.

## 60s

Also called the "transistors era" or "third-generation computers", in the 60's we start seeing smaller yet even more faster computers.


The main goal here is to keep up the CPU **as busy as possible**, denoting that while a specific program *waits* on I/O, another one can use a "slice" of the CPU.

Two concurrent programs could use multiple computer resources *at the same time*.

But how is this achieved?

### Monitors, the former operating systems

Monitors were *primitive* systems that once installed in the computer, they could **manage concurrent programs** and give them a fair amount of the *CPU* while other programs are blocked on I/O.

This "fairness" is based on an arbitrary time of the CPU, so as long as the Monitor thinks a program used its *fair time* of the CPU, it **pauses** this program and prevents it from using the CPU, giving priority to another program which was on the **waiting queue**.

And this process repeats over and over again while programs finish their I/O operations.

Such technique is called time-sharing multitasking.

Make no mistake, **concurrency** is all about multiple concurrent programs getting a fair amount time of the CPU while they wait on I/O. It's still *one* CPU for all of them, but the *Monitor* helps to **keep the CPU busy as much as possible**.

By achieving multitasking and increasing the efficiency of resource utilization, we also increase the *volume of information being processed over time* (throughput).


## 70s

This is the decade where more sophisticated "Monitor Systems" like Unix were born. Those systems are called Operating Systems, and this is the "cambrian explosion era" of computers.

## Operating Systems

Like "Monitors", Operating Systems were created to manage computer resources (CPU, memory, I/O) and guarantee a fair amount of computer resources (mainly CPU) to multiple concurrent programs running in the computer.

Year after year, more operating systems are created. Let's dig in the architecture of a modern operating system (OS):

### Programs are isolated

Having *concurrency* in its realm, operating systems (OS from now on) need to guarantee that two different programs don't use the same memory address. Otherwise, it would lead to a race condition.

For solving that problem and avoid race condition, a modern OS has to follow some rules:

- the program needs to be isolated, meaning it must have its own memory space
- programs can communicate to each other only via **message passing**
- thus, programs need a unique identifier

### OS Processes

These traits for a program to being isolated and having a unique identifier are what makes it a **process**.

Then, basically, processes are *instances* of programs.

Let's check some processes on our OS:

```
$ ps ax

PID TTY      STAT   TIME COMMAND
  1 pts/0    Ssl    0:00 /usr/bin/qemu-x86_64 /usr/bin/bash
 53 pts/0    Sl+    0:00 /usr/bin/qemu-x86_64 /usr/bin/sleep 10
 62 ?        Rl+    0:00 /usr/bin/ps ax
```

Note the process ID `53` which is running the command `sleep 10`. This process is *waiting* on the computer clock and is **not using the CPU**, and as soon as it finishes, the process is gone and completely removed from the OS.

Now, let's raise one more question: what if we write a program but within this program we want some specific **blocks of code** to being executed concurrently?

Specifically saying, it's a scenario where *a block of code is waiting on I/O, but another one in the same process is "free" to compete on CPU*.

Enter **threads**.

### OS Threads

Operating Systems also bring a concurrent primitive called **Thread**, which is **bound to a process** and can be treated as a concurrency unit like OS processes.

Different threads **running in the same process** are not isolated, because they *share the same memory space*.

Threads are created within programs developers use to write and do share the same process memory. Hence, Threads are bound to **race conditions**.

```
$ ps a -o pid,tid,command

PID   TID COMMAND
  1     1 /usr/bin/qemu-x86_64 /usr/bin/bash
197   197 /usr/bin/qemu-x86_64 /usr/bin/sleep 10
200   200 /usr/bin/ps ax -o pid,tid,command
```

We can see the `TID` (thread ID) column, having the same identifier as the `PID` (process ID). Because every OS process by default, *has a main thread* running on it.

Other threads might be created within the program by the application programmer.

Okay, all of these stuff about *processes and threads* are cool and nice, but **how do the OS manage those concurrency units**?

### OS Scheduler

The OS scheduler is a program which manages OS processes/ threads in the waiting queue and give them a fair amount of the CPU while others wait on I/O.

Similar to the primitive "Monitor", a scheduler uses **time-sharing multitasking** by pausing/resuming OS processes and threads through a **context switch**.

OS Processes/Threads are preempted multiple times on CPU, and because of this nature of preempting concurrency units by *time*, most modern operating systems employ **Preemptive Schedulers**.

### Cooperative scheduling

In the 70s/80s, a small amount of operating systems used to have a different scheduling strategy. Those schedulers **do NOT preempt** OS processes by time-sharing, but instead delegate to the OS process to make its own "context switch", based on the process rules and requirements.

Such scheduling is called "Cooperative Scheduling", as the scheduler gives control to the process for making the context switch.

However, most modern operating systems use preemptive multitasking, because they can have complete control over the concurrency units on the computer.

## Race condition

As said earlier, OS processes are isolated *by design*, so they are not bound to race-conditions.

However, threads do share the same memory space (the OS process memory), then programmers have to carefully design multi-threading systems.

In the presence of a potential race condition, a single Thread can acquire a "lock", which is an OS primitive that prevents other Threads in the same process from being preempted in the CPU.

Still, locks can be cumbersome and lead to **deadlocks**, where two different Threads are *blocked forever* because they are waiting locks from each other.

To avoid locking, other techniques which employ **optmistic locking** arise, where instead of acquiring OS locks, two different "versions" of data are generated then "compared" before updating.

Yet another alternative to OS locks, is by making the Thread more "safe", having its own isolated space, then communicating outside via *message passing*, similar to OS processes. Those threads follow the actor-model definition and can be called "actors".

Both **optmistic locking** and **actor-based threads** rely on algorithms that can be implemented by runtimes and programming libraries.

## Multi-core era

As the Moore's Law comes to an end due to physical limitations of computer resources, CPU's stopped increasing clock rates, which means they are **not getting much faster** since the mid 2000's.

CPU engineers then came to a great solution which is building multiple "CPU cores" into a single CPU-unit, where each CPU-core has the same clock speed. That's why since the mid 2000's we've seen the blast of multi-core processors as the number of cores are getting cheaper and increasing faster.

Now that we understand concurrency and the importance to make the CPU busy, how can we **increase the throughput of applications** in such a scenario where CPU's are not faster but do have multiple CPU-cores?

Yes, using all the CPU-cores at the same time. It's called **parallelism**.

In a modern multiple-core CPU architecture, concurrent processes/threads that need CPU work can be executed **in parallel**.

## Non-blocking era

On the other hand, in the lands of I/O, network bandwidth and SSD's have gotten faster year after year.

Then operating systems started to offer capabilities where process do not need to be "blocked" on I/O. These processes could be freed to do other work **asynchronously** and, as soon as the I/O operation is *completed*, the process is notified by the OS.

Such technique is called "non-blocking I/O", or "async I/O".

Runtime implementations such as NodeJS and other projects like Loom, PHP Swoole and Ruby3 employ concurrency primitives for taking advantage on async I/O, thus helping to increase the system overall throughput.

## Conclusion

I hope this article helped you to understand a bit more about operating systems (OS) and how OS processes/threads are crucial units for tackling concurrency on computers.


**Computers, OS and networking (3 Part Series)**

1. **A brief history of modern computers, multitasking and oper…**
2. Inter-process communication: files
3. Inter-process communication: pipes


## Top comments (2)

Chris Greening • 12 de jul. de 22

Love seeing how far we've come :~) insightful article Leandro - thank you!

Leandro Proença • 12 de jul. de 22

Thanks for the feedback Chris! Indeed, it's great seeing how operating systems evolved over time, and how those pieces connect in our daily basis writing software!

**Leandro Proença**

Programmer • I occasionally write blog posts in both English and Brazilian Portuguese.

**LOCATION**
Brazil

**EDUCATION**
BSc, Information Systems

**WORK**
Software Developer

**JOINED**
8 de ago. de 2017

### More from Leandro Proença

# 🌚 Life is too short to browse without dark mode