Processes

Any modern computer system is capable of executing multiple programs at the same time. However, in most cases, the central processing unit (CPU) can only execute one program at a time. Therefore, the task of running multiple programs at the same time falls to the operating system, which must introduce a model through which the execution of programs, viewed from the user's perspective, can take place in parallel. In fact, a pseudoparallelism is achieved, through which the processor is allocated in turn to the programs that need to be run, a quantum of time for each, so that from the outside they appear to be running effectively at the same time.

🔷 Basic concepts

🔶 Process-based model

The most widespread model that introduces parallelism in the execution of programs is the process-based model. This model is the one adopted by the Unix operating system and will be the subject of this paper.

🔶 What is a process?

A process is a sequentially executing program, together with its data area, stack and instruction counter (program counter). A distinction must be made from the outset between a process and a program. A program is, in essence, a series of instructions to be executed by the computer, while a process is an abstraction of the program, specific to operating systems. It can be said that a process executes a program and that the operating system works with processes, not programs. The process includes, in addition to the program, state information related to the execution of the respective program (stack, CPU register values, etc.). It is also important to emphasize that a program (as a software application) can consist of several processes that may or may not run in parallel.

🔷 Older CPUs with single core architecture

🔶 Sequentially execution

Any process is executed sequentially, and several processes can run in parallel (with each other). Most of the time, parallel execution is achieved by allocating the processor to one process in turn. Although only one process is executed at a time, within a second, for example, portions of several processes can be executed. From this diagram it follows that a process can be, at a given moment, in one of the following three states [Tanenbaum]:

- Running
- Ready for execution
- Blocked

🔶 Example

The process is in execution when the processor executes its instructions. Ready for execution is a process that, although it would be ready to continue its execution, is left waiting because another process is executing at that moment. Also, a process can be blocked for two reasons: it suspends its execution intentionally or the process performs an operation outside the processor, which is very time-consuming (as is the case of input-output operations these are slower and in the meantime the processor could execute parts of other processes).

🔷 Newer CPUs with multicore architecture

The **Apple M4 Pro** processor executes processes in **true parallel**, not sequentially, thanks to its **multi-core architecture** and **out-of-order execution**. Here's how it works:

🔶 Multi-Core Execution (True Parallelism)

The M4 Pro has multiple performance (P) cores and efficiency (E) cores. Each core can execute its own process independently. If multiple processes are running, macOS schedules them across different cores, so they run truly in parallel.

🔶 Multi-Threading & Hyper-Threading

Apple Silicon does not use traditional Hyper-Threading (SMT) like Intel. Instead, it has many cores that can run real parallel workloads instead of faking extra "virtual" threads.

🔶 Out-of-Order Execution

Even within a single core, the M4 Pro reorders instructions to maximize CPU efficiency. This makes execution faster than strict sequential processing.

🔶 Unified Memory Architecture (UMA)

The high-bandwidth, low-latency memory allows multiple cores to access shared data efficiently. This further enhances parallel execution by reducing memory bottlenecks.

🔶 GPU & Neural Engine Parallelism

The GPU (many cores) and Neural Engine run workloads massively in parallel. These are optimized for graphics, AI, and machine learning tasks.

🔶 Final Answer:

If I run multiple processes, the M4 Pro will schedule them across different cores, meaning true parallel execution.

If a single-threaded process, it runs sequentially within one core, but still benefits from out-of-order execution for efficiency.
So, yes, the Apple M4 Pro executes in true parallel when multiple cores are involved!

🔷 Using Processes in UNIX

🔶 The fork() System Call

From the programmer's perspective, the UNIX operating system provides an elegant and simple mechanism for creating and using processes. Any process must be created by another process. The creating process is called the parent process, and the created process is called the child process. There is only one exception to this rule, namely the init process, which is the initial process, created when the operating system starts and which is responsible for creating subsequent processes. The command interpreter, for example, also runs inside a process.

🔶 Unique numerical identifier(Process Identifier)

Each process has a numerical identifier, called the process identifier (PID). This identifier is used when referring to the respective process, from within programs or through the command interpreter.

🔶 A process must be created using the system call: pid_t fork()

Through this system function, the calling process (the parent) creates a new process (the child) that will be a faithful copy of the parent. The new process will have its own data area, its own stack, its own executable code, all copied from the parent in every detail. As a result, the child's

variables will have the values of the parent's variables at the time of the fork( ) function call, and the child's execution will continue with the instructions that immediately follow this call, the child's code being identical to that of the parent. However, from now on, there will be two independent processes in the system, (although identical), with distinct data areas and stacks. Any modification made, therefore, to a variable in the child process will remain invisible to the parent process and vice versa. The child process will inherit from the parent all the file descriptors opened by it, so any subsequent processing of files will be performed at the point where the parent left them.

Because the parent and child code are identical and because these processes will continue to run in parallel, a clear distinction must be made within the program between the actions that will be executed by the child and those of the parent. In other words, a method is needed to indicate which portion of the code is the parent's and which is the child's.
This can be done simply by using the return value of the fork( ) function. It returns:
- -1, if the operation could not be performed (error)
- 0, in the child code
- pid, in the parent code, where pid is the process identifier of the newly created child.

Therefore, a possible calling scheme for the fork( ) function would be:

```
...
if( ( pid=fork() ) < 0)
{
    perror("Error");
    exit(1);
}

if(pid==0)
{
    /* child code */
    ...
    exit(0)
}

/* parent code */
...
wait(&status)
```

🔷 The wait() and waitpid() functions

pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int flags)
The wait( ) function is used to wait for the child to terminate and retrieve its return value. The status parameter is used to evaluate the return value, using some specially defined macros (see the manual pages corresponding to the wait( ) and waitpid( ) functions). The waitpid( ) function is similar to wait( ), but it waits for the termination of a given process, while wait( ) waits for the termination of any child of the current process. It is mandatory that the state of the processes be retrieved after their termination, so the functions in this category are not optional.

🔷 Functions of the exec() type

The fork( ) function creates a process identical to the parent process. To create a new process that runs a different program than the parent, this function will be used together with one of the exec( ) system calls: execl( ), execlp( ), execv( ), execvp( ), execle( ), execve( ).

All of these functions receive as a parameter a filename that represents an executable program and execute the program. The program will be launched in such a way that the code, data and stack of the process that calls exec( ) will be overwritten, so that immediately after this call the initial program will no longer exist in memory. The process will remain, however, identified by the same number (PID) and will inherit all possible redirections made previously on file descriptors (for example, standard input and output). It will also maintain the parent-child relationship with the process that called fork( ).

The only situation in which the calling process returns from the exec( ) function call is when the operation could not be performed, in which case the function returns an error code (-1).

Consequently, launching a program from disk into a separate process is done by calling fork( ) to create the new process, after which one of the exec( ) functions will be called in the code portion executed by the child.

Note: consult the manual pages corresponding to these functions.

🔷 The system() and vfork() functions

🔶 int system(const char *cmd)

Launches a program from disk, using for this purpose a fork( ) call, followed by exec( ), together with waitpid( ) in the parent.

🔶 pid_t vfork()

Creates a new process, just like fork( ), but does not copy the entire address space of the parent to the child. It is used in conjunction with exec( ), and has the advantage of not consuming the time required for copy operations that would be useless anyway if exec( ) is called immediately afterwards (however, the child process will be overwritten with the program taken from the disk).

🔷 Other functions for working with processes

- pid_t getpid() returns the PID of the current process
- pid_t getppid() returns the PID of the current parent process
- uid_t getuid() returns the identifier of the user who launched the current process
- gid_t getgid() returns the identifier of the group of the user who launched the current process

🔷 Managing processes from the command line

The UNIX operating system has some very useful commands that refer to processes:

ps displays information about the processes currently running on the system
kill process signal sends a signal to a process. For example, the command kill -9 123 will kill process number 123
killall -signal name send signal to all processes with name name

There are other useful commands; for their use, it is recommended to consult the UNIX man pages.

🔷Copied and shared data after forking a new process

> Unix is built on two fundamental abstractions: files and processes. A file is a stream of bytes. A *process* is an active entity, while a program is a static collection of bits. A program, more specifically, is a collection of bits that could be executed on the given computer under the given OS.

When writing to a terminal, the kernel flushes the output buffers when there is a newline character. But when writing to a file, buffering takes place in larger units that are a better match to the disk block size. After the fork(), the child process gets a copy of all the parent's data, including the unflushed user-level I/O buffers. When each process exits, it flushes it's I/O buffers, so each process sends a copy of this data to disk.

> **Moral:** Be careful about user-level data that will be copied across a fork(). In particular, I should be mindful of open files and unflushed I/O buffers. The easest thing to do is to flush and close open files before forking.

Keep in mind that the child gets a *copy* of the address space of the parent. (Read only memory, like the text segment, may be shared.) This means that any modifications either process makes to its data is not visible to the other. This is a feature that greatly simplifies concurrent programming with processes (which contrasts sharply with the shared memory model of threads). Generally, this is good from the point of view of software engineering and abstraction. However, questions of efficiency aside, it does mean that it can be difficult for cooperating processes to communicate.

One thing that *is* shared is the file table entry for any open files. That means that parent and child also share a file pointer for any open files. This is good if I want both processes to write to, for example, a common standard output, standard error, or a log file. I should be careful about I/O however, because there are no guarantees about which process will run when or for how long at a time, so I/O operations can be unpredictably interleaved.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

void print_chars(char *s)
{
        for ( ; *s != '\0'; putchar(*s++)) ;
}

int main()
```

```
{
        pid_t pid;
        setbuf(stdout, NULL);  /* Want unbuffered output */

        if ( (pid = fork()) < 0 )
                fprintf(stderr, "Can't create fork.");
        else if (pid == 0)
                print_chars("Child says:  \"Hello there!\"\n");
        else
                print_chars("Parent says:  \"Is anyone home?\"\n");

        return EXIT_SUCCESS;
}
```

Output

```
Child saParent says:  "Is anyone home?"
```

🔷 The wait() function

The wait() function is used in **parent processes** to wait for their **child processes to finish execution**. It prevents **zombie processes** and allows the parent to retrieve the child's exit status.

🔶 Basic Behavior of wait() function

- Blocks the parent process until one of its child processes terminates.
- Returns the PID of the terminated child.
- Stores the child's exit status (if needed).
- Prevents zombie processes (defunct processes that have finished execution but still occupy an entry in the process table).

🔶 Variations of wait() function

Function	Behavior
wait(&status)	Blocks parent until **any** child exits.
waitpid(pid, &status, 0)	Waits for **a specific child** (PID).
waitpid(-1, &status, WNOHANG)	**Non-blocking** version; returns immediately if no child has exited.

🔷 Orphan processes

If a parent creates multiple child processes but calls wait() only once, it will wait for only one child to finish. The other children will become zombie processes if the parent exits before them.

🔶 Key Takeaways

- If wait() is called only once, only one child is reaped, and the others may become zombies if the parent exits.

- If the parent exits before the children, the children become orphans and get adopted by init (PID 1).
- Zombies happen if a child exits but the parent doesn’t call wait().
- Orphans keep running but get reparented to init, which will wait() for them automatically.
- Zombies are not active processes, but they occupy the process table.
- They exist because the parent hasn't collected the child's exit status using wait().
- If the parent exits, init takes over and cleans up the zombie.
- Too many zombie processes can exhaust system resources, preventing new processes from starting.

🔶 What Happens to Zombies in Different Scenarios?

Scenario	What Happens?
Parent calls wait()	Zombie is removed immediately.
Parent exits before wait()	Zombie is adopted by init, which cleans it up.
Parent stays alive without calling wait()	The zombie persists indefinitely, occupying a process table entry.

🔷 States of a process

🔶 Process State Transitions

```
New (created) → Ready → Running → (Sleeping / Waiting) → Terminated (or Zombie)
```

- A running process can become sleeping if waiting for I/O.
- A sleeping process can return to ready when the event is completed.
- A zombie is cleaned up when the parent calls wait().

🔶 Summary Table

State	Meaning
New	Process created but not started
Running	Executing on CPU
Ready	Ready but waiting for CPU
Sleeping (Interruptible)	Waiting, can be interrupted
Sleeping (Uninterruptible)	Waiting, cannot be interrupted (I/O wait)
Stopped	Paused by signal (SIGSTOP, Ctrl+Z)
Zombie	Terminated, but parent didn’t wait()

🔷 Interprocess Communication (IPC) Using Pipes

Pipes are one of the simplest Interprocess Communication (IPC) mechanisms used in Unix/Linux systems. They allow processes to send data to each other through a unidirectional or bidirectional communication channel.

Interprocess Communication (IPC) Using Pipes

🔶 Definition of a Pipe

A pipe is a unidirectional communication channel used for interprocess communication (IPC). It allows one process to send data to another in a synchronized manner.

🔶 Characteristics of Pipes

Unidirectional: Data flows in one direction.
Used between related processes (e.g., parent-child processes).
Blocking behavior:
  Reader blocks if no data is available.
  Writer blocks if the pipe's buffer is full.
Implemented in the kernel as a circular buffer.

🔶 Creating a Pipe

```
int pipe(int fd[2]);
fd[0]: Read end of the pipe.
fd[1]: Write end of the pipe.
```

🔶 Process Communication Using Pipes

- After a fork, both parent and child share the same pipe.
- One process writes data using `write(fd[1], buffer, size)`.
- The other process reads using `read(fd[0], buffer, size)`.
- The unused ends of the pipe should be closed in each process.

🔶 Bidirectional Communication with Pipes

Since pipes are unidirectional, two pipes are required for full-duplex communication:

- First pipe: Parent → Child
- Second pipe: Child → Parent

🔶 Closing Unused Pipe Ends

- To prevent deadlocks and resource leaks, close the unused ends:
- The writer should close `fd[0]`.
- The reader should close `fd[1]`.

🔶 Named Pipes (FIFOs)

A named pipe (FIFO) allows communication between unrelated processes. It is created with:

```
int mkfifo(const char *pathname, mode_t mode);
Can be used by multiple processes.
Behaves like a file, supporting `open()`, `read()`, and `write()`.
```

🔶 Pipe Limitations

- Unidirectional: Requires two pipes for two-way communication.
- Limited buffer size: Data is temporarily stored in the kernel.
- Parent-child scope: Anonymous pipes work only between related processes.

🔷 File Descriptor Redirecting and Duplication

🔶 File Descriptors Overview

A **file descriptor (FD)** is a numerical identifier for an open file, socket, or device in Unix-like systems. The standard file descriptors are:

- **0** → Standard Input (stdin)
- **1** → Standard Output (stdout)
- **2** → Standard Error (stderr)

🔶 File Descriptor Redirection

Redirecting file descriptors means changing where input/output operations are directed. This is commonly done with system calls like dup(), dup2(), and fcntl().

🔶 File Descriptor Duplication

Duplication allows multiple file descriptors to refer to the same open file description. This is useful for redirection.

dup() – Duplicates a file descriptor to the lowest available number

```
int dup(int oldfd);
```

- Returns a new file descriptor with the **lowest available** integer.
- Both descriptors now refer to the same open file.
- They share the same file offset and flags.

dup2() – Duplicates a file descriptor to a specific number

```
int dup2(int oldfd, int newfd);
```

- If newfd is already open, it is closed first.
- newfd now refers to the same file as oldfd.
- Ensures redirection to a specific file descriptor.

🔶 Common Use Cases

- Redirecting standard output to a file:
  1. Open a file.
  2. Use dup2() to replace stdout (1) with the file descriptor.
  3. Write operations now go to the file instead of the terminal.
- Redirecting standard error (stderr) to stdout:
  dup2(1, 2);
  - This merges stderr with stdout, so error messages go to the same output stream.

🔶 Closing and Managing File Descriptors

- Always close unnecessary file descriptors to avoid resource leaks.
- close(fd) should be used after duplication when the original descriptor is no longer needed.

> File descriptor duplication is primarily used for **redirection**, **process communication**, and **resource management** in Unix-like systems.

🔶 Summary: When to Use dup() and dup2()

Use Case	Function	Description
Redirect stdout to a file	dup2(file_fd, 1);	Writes go to a file instead of terminal
Redirect stderr to a log file	dup2(log_fd, 2);	Error messages go to a log file
Merge stdout and stderr	dup2(1, 2);	Both outputs are combined
Use pipes for IPC	dup2(pipe_fd[1], 1);	Writes go into a pipe instead of stdout
Implementing a shell (exec()redirection)	dup2(file_fd, 1); execvp(cmd, args);	Executes a program with modified output
Running daemon processes	dup2(devnull_fd, 1);	Prevents daemon from printing to terminal