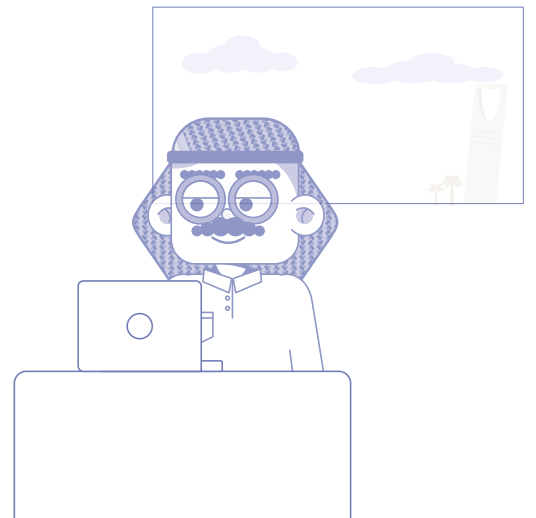
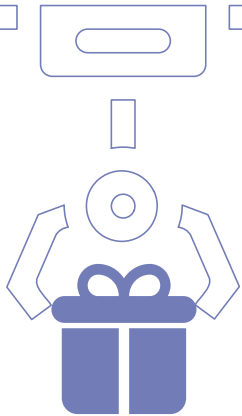
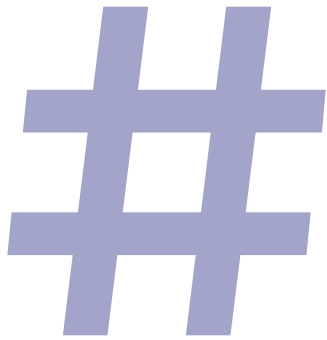


OPERATING SYSTEM

# Process and Thread Management





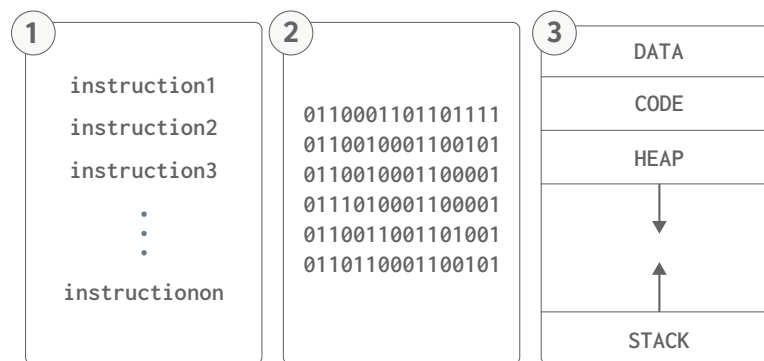
# **Process Introduction**

## Step 01 Process Introduction

A **process** is a program in execution. That means when a program is at run time, the program becomes a process. However, what was the program called before that?

The program passes through 3 phases in its lifetime which are as follows

- **Development phase:** source code.
- **Compile phase:** executable file.
- **Run-time phase:** a process.



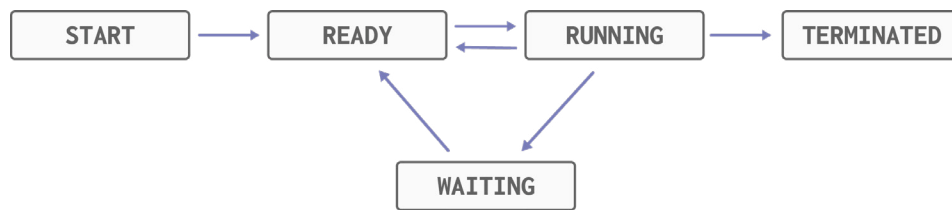
When a program is executed, it is loaded into the main memory as a process to be handled by the operating system, from execution to assigning resources. Once the process has completed its task, the operating system will free located memory addresses and resources from its use.

## Step 02 Process States

Once the program is loaded into memory and becomes a process, it will have one of the following states.

### 01 process states

- **New:** The process has just been created and it is in its initial state.
- **Ready:** The process is ready to be executed.
- **Running:** The CPU is working on this process's instructions.
- **Waiting:** The process can not run at the current time since it is waiting for an event or a resource to become available.
- **Terminated:** The process has finished.



All these states are used to manage and control processes efficiently.

### Step 03 Process Tree

Every process is part of another process. If a running program (process) executes another program, the first one will be the parent of the second one. Processes relationship forms a tree data structure.

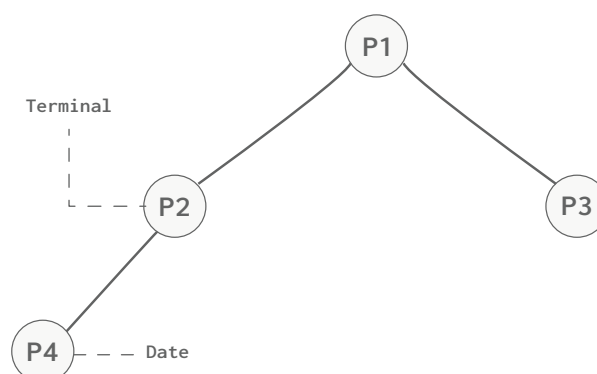
Let us have an example of running a date program from the terminal, the terminal is a program that enables users to command the operating system. For example, creating and accessing files and directories.

Here is an example of running a **date program** from the shell to display today's date.

```

Dev: date
Sun May 26 17:10:54 +03 2024
Dev:
  
```

The terminal is a process that executes another process. Therefore, the terminal will be the parent process of the date process. The tree will look like the following.



**NOTE**

The root process is the first process that the computer runs, in unix-based OS, it is typically init process.

## Step 04 Processes and Syscalls

Operating systems manage processes using **syscalls**. There are some services provided by the operating system to create, manage, and control processes.

In Unix-like systems, you can find the following syscalls for process management.

### 01 Process Creation

- **fork()**: Creates a new process that is a copy of the calling process. The new process is called the child process.
- **exec()**: it takes the current process and replaces it with a new program. This means the original program stops running, and the new program starts fresh in the same process.

### 02 Process Termination

- **exit()**: Ends the current process and sends a status code to the operating system, indicating how it finished (success or error)
- **kill()**: Sends a signal to a process, which can be used to terminate it or interrupt its execution.

### 03 Process Control

- **wait()**: Makes a parent process pause until one of its child processes finishes running.
- **waitpid()**: Similar to **wait()**, but allows the parent to wait for a specific child process to finish.

### 04 Process Scheduling

- The operating system uses scheduling algorithms to determine which processes run and for how long. Syscalls allow processes to yield control voluntarily or be interrupted based on priority.

### 05 Interruption and Signals

- **Signal Handling**: Allows processes to respond to events, like interruptions from the user or other processes. This is done using syscalls like `signal()`.

`pause()`

Suspends the process until a signal is received.

## 06 Process Information

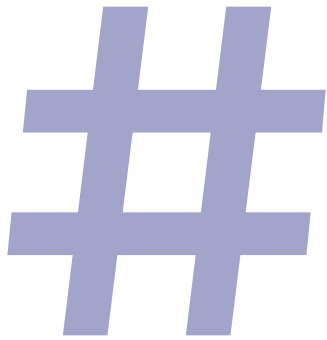
- **getpid()**: Returns the process ID of the calling process.
- **getppid()**: Returns the parent process ID.

### NOTE

There are some differences between operating systems and the names of their syscalls. For example, creating a process in Windows OS can be done using **CreateProcess()** syscall.

## Step 05 Summary

We have introduced the concept of the process for now, and we will delve into how to interact with the process using GUI, Command Line, and through the use of Programming Languages.



# **Process Interaction**

## Step 01 Process Management through GUI and Command Line

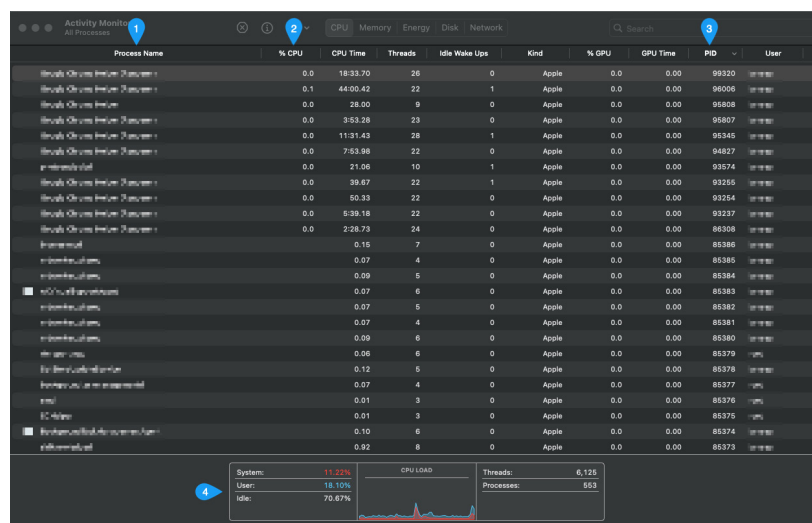
OS provides some services for programs or users to manage processes, such as creating, pausing, executing, or killing a process. To access these features we may use some libraries in our programming languages that communicate directly with the OS or even call syscalls directly, or we can use the command line and GUI to interact and manage processes. In our case, we will demonstrate the concept of managing processes through GUI using **Activity Monitor** in MacOS or **Task Manager** in Windows OS. And using the command line. However, we will discuss how to manage processes using the libraries in programming languages later on.

### NOTE

Both ways of interaction uses **syscalls** provided by the operating systems to manage processes.

## Step 02 Activity Monitor

It is a utility in MacOS devices that helps you analyze and manage processes and see their resource usage.



Above is an image of the activity monitor program, it displays the current activities of the computer including running programs (processes).

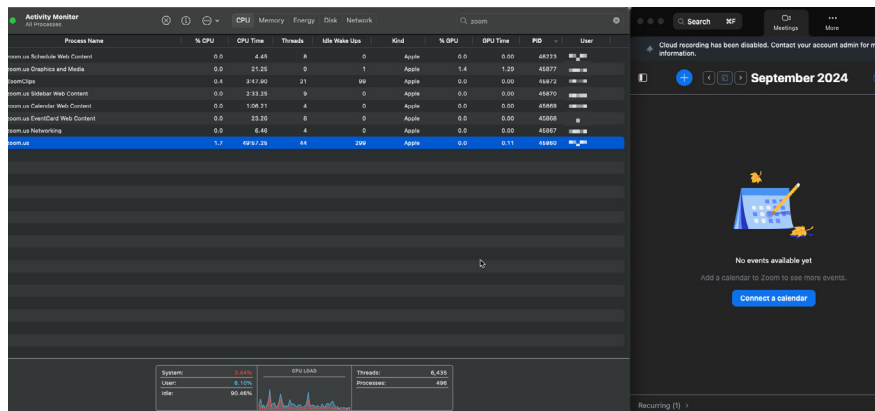
- 01 **Process Name** is a list of all currently running programs.
- 02 **CPU** is the CPU usage by each process.
- 03 **PID** each process is assigned to a process ID for identifying them.
- 04 **CPU Load** is an overall current CPU state that identifies how much CPU is used and by whom.

Resource for [Activity Monitor](#)

Resource for [Task Manager](#)

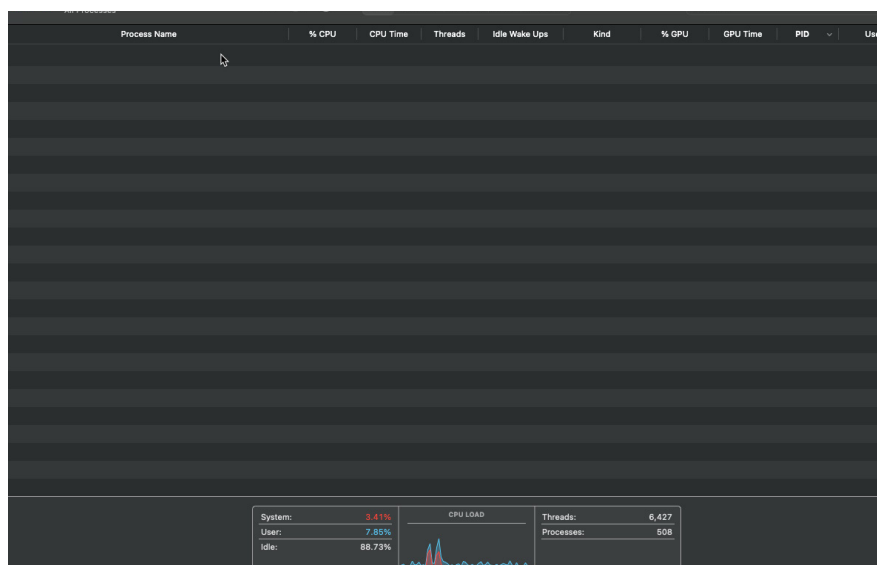


Let us open the Zoom program and search for it using the Activity Monitor.



As you can see, when Zoom is running, there must be a process which was displayed in the image above. What do you think will happen if we close Zoom?

The process must be terminated since the program is not running, let us terminate it and check again.



#### NOTE

To quit Zoom, you can double-click on the process and choose quit. Or, close the program from the zoom itself.

## Step 03 Command Line

Open your terminal on MacOS & Linux or command line prompt on Windows.

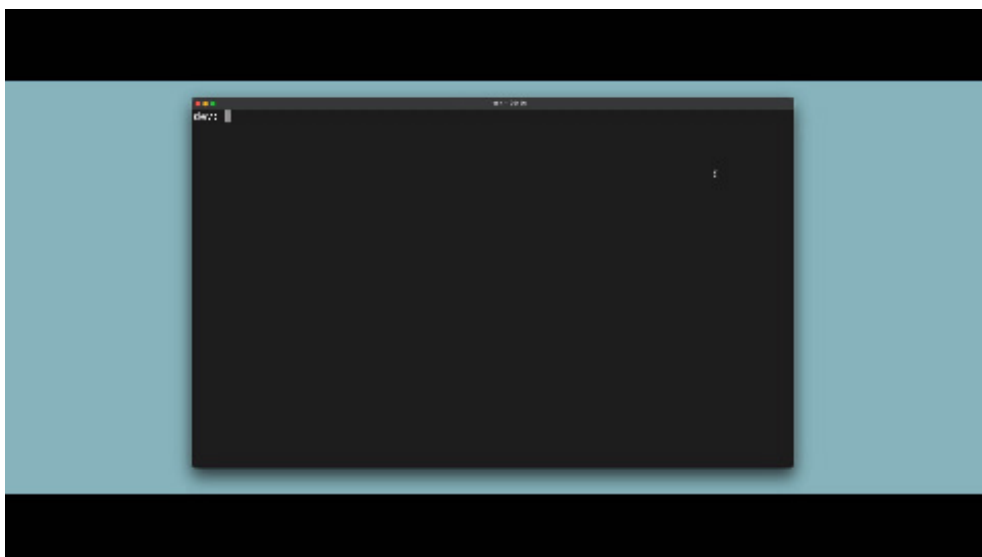
### 01 MacOS & Linux

Use the **top** command to display all the current processes and their data.

```
Processes: 566 total, 6 running, 560 sleeping, 6633 threads          13:17:13
Load Avg: 2.29, 2.72, 2.79  CPU usage: 14.63% user, 8.59% sys, 76.76% idle
SharedLibs: 459M resident, 91M data, 30M linkedit.
MemRegions: 2443861 total, 3986M resident, 135M private, 945M shared.
PhysMem: 156 used (2502M wired, 7920M compressor), 63M unused.
VM: 434T vsize, 4892M framework vsize, 233970947(439) swapins, 237545656(780) swapouts.
Networks: packets: 46800682/376 in, 20792368/5459M out.
Disks: 95969411/42246 read, 98398464/40566 written.
```

PID	COMMAND	%CPU	TIME	#TH	#WQ	#PORT	MEM	PURG	CMPRS	PGRP	PPID	STATE
94521	replayd	79.4	00:01.31	10/1	6/1	146+	15M+	0B	0B	94521	1	running
146	WindowServer	42.9	14:55:06	21	5	5924-	1156M-	1008K-	346M-	146	1	sleeping
703	SnagitHelper	24.6	04:07.11	42	9	624-	229M-	16K+	131M+	703	1	sleeping
0	kernel_task	19.0	18:42:07	588/12	0	0	56M+	0B	0B	0	0	running
94414	top	17.5	00:20.08	1/1	0	47+	9745K+	0B	2832K	94414	94377	running
84672	sysmond	17.5	02:48.96	3/1	2/1	26+	7969K+	0B	6576K-	84672	1	running
165	coreaudiod	11.7	03:11:07	12	3	2949	52M	0B	36M	165	1	sleeping
480	corespeechd	6.8	01:40:02	13/1	5	392	32M	0B	18M	480	1	running
463	WindowManage	3.9	20:50.47	5	2	1234+	38M-	0B	19M+	463	1	sleeping
500	Google Chrom	2.3	08:22:00	47	2	5147	1277M+	0B	793M-	500	1	sleeping
554	Finder	2.3	11:11.32	6	3	782+	128M+	0B	99M-	554	1	sleeping
91633	zoom.us	2.2	00:41.34	40	5	635	196M	0B	168M-	91633	1	sleeping
25517	Google Chrom	2.1	73:17.62	30	1	1102	1041M	0B	1020M+	500	500	sleeping
678	Google Chrom	1.9	02:08:36	20	1	1436	168M+	0B	82M-	500	500	sleeping
94293	ScreenFlow	1.8	00:19.44	11	2	601	288M	0B	134M-	94293	1	sleeping
8079	Google Chrom	1.6	02:05:11	26	1	415	300M	0B	287M	500	500	sleeping
154	runningboard	1.6	13:37.29	8	7	477-	11M+	0B	2496K-	154	1	sleeping
968	Google Chrom	1.6	43:08.52	26	1	394	207M	0B	176M+	500	500	sleeping
94374	Terminal	1.5	00:04.14	10	5	281	42M+	8960K+	13M+	94374	1	sleeping
49111	Google Chrom	1.4	39:18.81	24	1	774	554M	0B	535M+	500	500	sleeping
4245	Google Chrom	1.4	52:08.74	29	1	852	514M	0B	494M+	500	500	sleeping
63574	aned	1.3	00:06.10	4	2	93-	4609K-	0B	1760K-	63574	1	sleeping
96006	Google Chrom	1.3	46:13.91	22	1	829	761M	0B	696M-	500	500	sleeping

To kill a process, you can use the kill command and provide it with the process ID.  
The following video demonstrates the complete process of displaying and terminating a process  
(we terminated the terminal process as an example)

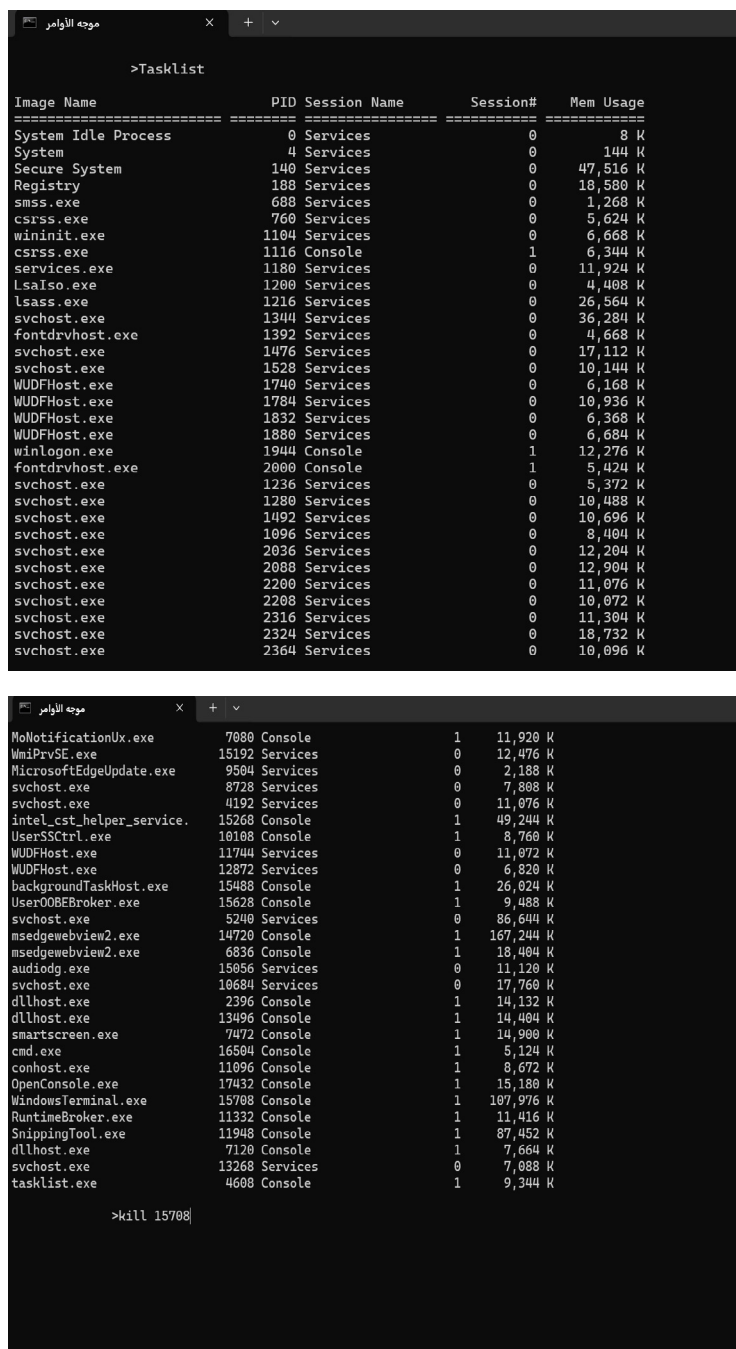


## 02 Windows

For Windows users, you can type the **Tasklist** command to display all processes.

### Resource for [Tasklist command](#)

Here is an example of listing current processes and terminating the CMD process using the command line.



The first screenshot shows the output of the `>Tasklist` command. The second screenshot shows the output of the `>kill 15708` command, which terminates the `cmd.exe` process.

Image Name	PID	Session Name	Session#	Mem Usage
System Idle Process	0	Services	0	8 K
System	4	Services	0	144 K
Secure System	140	Services	0	47,516 K
Registry	188	Services	0	18,580 K
smss.exe	688	Services	0	1,268 K
csrss.exe	760	Services	0	5,624 K
wininit.exe	1104	Services	0	6,668 K
csrss.exe	1116	Console	1	6,344 K
services.exe	1180	Services	0	11,924 K
lsass.exe	1200	Services	0	4,408 K
lsass.exe	1216	Services	0	26,564 K
svchost.exe	1344	Services	0	36,284 K
fontdrvhost.exe	1392	Services	0	4,668 K
svchost.exe	1476	Services	0	17,112 K
svchost.exe	1528	Services	0	10,144 K
WUDFHost.exe	1740	Services	0	6,168 K
WUDFHost.exe	1784	Services	0	10,936 K
WUDFHost.exe	1832	Services	0	6,368 K
WUDFHost.exe	1880	Services	0	6,684 K
winlogon.exe	1944	Console	1	12,276 K
fontdrvhost.exe	2000	Console	1	5,424 K
svchost.exe	1236	Services	0	5,372 K
svchost.exe	1280	Services	0	10,488 K
svchost.exe	1492	Services	0	10,696 K
svchost.exe	1096	Services	0	8,404 K
svchost.exe	2036	Services	0	12,204 K
svchost.exe	2088	Services	0	12,904 K
svchost.exe	2200	Services	0	11,076 K
svchost.exe	2208	Services	0	10,072 K
svchost.exe	2316	Services	0	11,304 K
svchost.exe	2324	Services	0	18,732 K
svchost.exe	2364	Services	0	10,096 K

Image Name	PID	Session Name	Session#	Mem Usage
MolNotificationUx.exe	7080	Console	1	11,920 K
MiniPrivSE.exe	15192	Services	0	12,476 K
MicrosoftEdgeUpdate.exe	9504	Services	0	2,188 K
svchost.exe	8728	Services	0	7,808 K
svchost.exe	4192	Services	0	11,076 K
intel_cst_helper_service.	15260	Console	1	49,244 K
UserSSCtrl.exe	10108	Console	1	8,760 K
WUDFHost.exe	11744	Services	0	11,072 K
WUDFHost.exe	12872	Services	0	6,820 K
backgroundTaskHost.exe	15488	Console	1	26,024 K
UserOOBEBroker.exe	15628	Console	1	9,488 K
svchost.exe	5248	Services	0	86,644 K
msedgewebview2.exe	14720	Console	1	167,244 K
msedgewebview2.exe	6836	Console	1	18,404 K
audiodg.exe	15056	Services	0	11,120 K
svchost.exe	10684	Services	0	17,760 K
dllhost.exe	2396	Console	1	14,132 K
dllhost.exe	13496	Console	1	14,404 K
smartscreen.exe	7472	Console	1	14,900 K
cmd.exe	16504	Console	1	5,124 K
conhost.exe	11096	Console	1	8,672 K
OpenConsole.exe	17432	Console	1	15,180 K
WindowsTerminal.exe	15708	Console	1	107,976 K
RuntimeBroker.exe	11332	Console	1	11,416 K
SnippingTool.exe	11948	Console	1	87,452 K
dllhost.exe	7120	Console	1	7,664 K
svchost.exe	13268	Services	0	7,088 K
tasklist.exe	4608	Console	1	9,344 K

`>kill 15708`

Once we kill the cmd web process, the cmd window will be closed same as the MacOS video.

## Step 04 Process Management through Programming Language

Since we have seen how can we manage processes through the use of GUI and Command Line, we will now look into how its managed through a programming language such as C or Java.

## Step 05 Process Management Code

Let us look into some examples in C indicating how to access syscalls that are described earlier for process management.

### 01 Process Management Syscalls

There are some header files that can be used to access syscalls such as,

```
01 | #include <unistd.h>    // For fork(), exec(), getpid(), etc.
02 | #include <sys/types.h> // For data types used in syscalls
03 | #include <sys/wait.h>  // For wait() and waitpid()
04 | #include <signal.h>    // For signal handling
```

Let's have an example of creating a simple C program printing **Hello World** and trying to modify it to use process syscalls.

```
01 | #include <stdio.h>
02 |
03 | int main(){
04 |     printf("Hello World\n");
05 |     return 0;
06 | }
```

#### OUTPUT

Hello World

- Create a Process

To create a process, we will use `fork()` syscall, to do so, we will need to include `#include <unistd.h>`.

```
01
02     #include <stdio.h>
03     #include <unistd.h>
04
05     int main(){
06         fork();
07         printf("Hello World\n");
08         return 0;
09     }
```

#### OUTPUT

```
Hello World
Hello World
```

The output got duplicated since the current program/process has been duplicated using the fork syscall and then executed.

The fork returns a number, which is the process id, we can store it in an integer variable using an `int` data type, or we can use `sys/types` and use `pid_t` as the type of the process id.

```
01
02     #include <stdio.h>
03     #include <unistd.h>
04     #include <sys/types.h>
05
06     int main(){
07         pid_t pid = fork();
08         printf("Hello World\n");
09         printf("PID: %d\n",pid);
10         return 0;
11     }
```

#### OUTPUT

```
Hello World
PID: 35638
Hello World
PID: 0
```

The values returned by the fork have meaning as follows.

- **Positive Value:** If `fork()` returns a positive value, this value is the Process ID (PID) of the child process. It indicates that you are in the parent process.
- **Zero (0):** If `fork()` returns 0, this means you are in the child process.
- **Negative Value:** If `fork()` returns a negative value (usually -1), it indicates that the process creation failed.

With the help of the returned pid value, we can write a program that executes some code if the child process is running, and another for the parent process.

```
01  #include <stdio.h>
02  #include <unistd.h>
03  #include <sys/types.h>
04
05  int main(){
06      pid_t pid = fork();
07
08      if(pid == 0 ){
09          printf("Hello from child process\n");
10      }else{
11          printf("Hello from parent process\n");
12      }
13
14      return 0;
15  }
```

#### OUTPUT

```
Hello from parent process
Hello from child process
```

- Read a Process

Reading a process means accessing the process data such as its id.

Let's have an example of reading child and parent processes ids.

```

01  #include <stdio.h>
02  #include <unistd.h>
03  #include <sys/types.h>
04
05  int main(){
06      pid_t pid = fork();
07
08      if(pid == 0 ){
09          printf("Hello from child process\n");
10          printf("PID: %d\n", getpid());
11          printf("PPID: %d\n", getppid());
12      }else{
13          printf("Hello from parent process\n");
14          printf("PID: %d\n", getpid());
15          printf("PPID: %d\n", getppid());
16          // sleep(2);
17      }
18      return 0;
19  }
```

#### OUTPUT

```

Hello from parent process
PID: 38542
PPID: 1120
Hello from child process
PID: 38543
PPID: 1
```

The ppid of the child process is the parent process id, which should have been 38542. Why did the output became 1?

As you can see, the parent process was executed and terminated before the child. Therefore, when the child process runs and tries to print its ppid, it can not find it. Its parent has just been terminated. But, with what we know about the process tree is that each process should have a parent, so when a child process loses its parent it becomes an orphan. Orphan processes are typically adopted by the root process (i.e. init) which has the id of 1.

To be able to print the correct ppid, we will make the parent process sleep a bit to make sure it does not terminate before the child process as the following.

```

01  #include <stdio.h>
02  #include <unistd.h>
03  #include <sys/types.h>
04
05  int main(){
06      pid_t pid = fork();
07
08      if(pid == 0 ){
09          printf("Hello from child process\n");
10          printf("PID: %d\n", getpid());
11          printf("PPID: %d\n", getppid());
12      }else{
13          printf("Hello from parent process\n");
14          printf("PID: %d\n", getpid());
15          printf("PPID: %d\n", getppid());
16          sleep(2);
17      }
18
19      return 0;
20  }

```

#### OUTPUT

```

Hello from parent process
PID: 38737
PPID: 1120
Hello from child process
PID: 38738
PPID: 38737

```

Now we notice, that the PPID of the child process is the same as the PID of our program since it did not terminate.

- **Terminate a Process**

To terminate a process, you must identify if you are terminating the current process you are in, or another process (such as a child process).



To terminate the current process you can use the `exit()` function as the following example.

```

01  #include <stdio.h>
02  #include <stdlib.h> // To call exit()
03
04  int main() {
05      printf("This process will be terminated now.\n");
06      exit(0); // Terminate with exit status 0 (success)
07  }

```

#### OUTPUT

This process will be terminated now.

To terminate another process, you may call the `kill()` function and pass it the PID of the process you want to kill.

```

01  #include <stdio.h>
02  #include <stdlib.h>
03  #include <unistd.h>
04  #include <signal.h>
05
06  int main() {
07      pid_t pid = fork();
08
09      if (pid == 0) {
10          // Child process
11          printf("Child process running. PID: %d\n", getpid());
12          sleep(10); // Simulate some work
13          printf("Child process finishing.\n");
14          exit(0);
15      } else {
16          // Parent process
17          sleep(2); // Wait for a moment
18          printf("Parent process terminating child with PID: %d\n", pid);
19          kill(pid, SIGTERM); // Send a terminating signal (SIGTERM) to the child
20          wait(NULL); // Wait for the child process to finish
21      }
22
23      return 0;
24  }

```

#### OUTPUT

Child process running. PID: 41932

Parent process terminating child with PID: 41932

As you can see, the kill happened before the child could have completed its task. Therefore, the rest of the code in the child process did not run. If we comment out the kill function you will see a different output which will make the parent wait until the child completes its task without interruption.

```

01  #include <stdio.h>
02  #include <stdlib.h>
03  #include <unistd.h>
04  #include <signal.h>
05
06  int main() {
07      pid_t pid = fork();
08
09      if (pid == 0) {
10          // Child process
11          printf("Child process running. PID: %d\n", getpid());
12          sleep(10); // Simulate some work
13          printf("Child process finishing.\n");
14          exit(0);
15      } else {
16          // Parent process
17          sleep(2); // Wait for a moment
18          printf("Parent process is not terminating the child with PID: %d\n", pid);
19          //kill(pid, SIGTERM); // Send a terminating signal (SIGTERM) to the child
20          wait(NULL); // Wait for the child process to finish
21      }
22
23      return 0;
24  }

```

**OUTPUT**

Child process running. PID: 42147

Parent process is not terminating the child with PID: 42147

Child process finishing.

**NOTE**

The wait(NULL) function in C is used to make a parent process wait for one of its child processes to terminate.

**NOTE**

SIGTERM (Signal Terminate) is a signal used in Unix-like operating systems to request the termination of a process.

- Execute Another Process

```
01  #include <stdio.h>
02  #include <stdlib.h> // To include system()
03
04  int main() {
05      // Execute the date command
06      int result = system("date");
07
08      // Check if the command was successful
09      if (result == -1) {
10          perror("Error executing date command"); //print error message
11          return 0;
12      }
13
14      return 1; // Return success
15  }
```

**OUTPUT**

Tue Dec 31 17:01:54 +03 2024

## 02 More examples in Java

- Create a Process

Create a program that runs the **date** process and displays its output

```
01  import java.io.BufferedReader;
02  import java.io.InputStreamReader;
03  import java.lang.Process;
04  import java.lang.ProcessBuilder;
05
06  public class Main {
07
08      public static void main(String[] args) {
09          try {
10              //Create a process to run date program
11              Process p = new ProcessBuilder("date").start();
12              //Read the process output data as an input
13              BufferedReader bfr = new BufferedReader
14                  (new InputStreamReader(p.getInputStream()));
15              String line = bfr.readLine();
16              //Loop through the process output and print it until it is finished
17              while (line != null) {
18                  System.out.println(line);
19                  line = bfr.readLine();
20              }
21              System.out.println(p.exitValue());
22          } catch (Exception e) {
23              e.printStackTrace();
24          }
25      }
26  }
27
28  }
```

- Read a Process

```
01  import java.io.BufferedReader;
02  import java.io.InputStreamReader;
03  import java.lang.Process;
04  import java.lang.ProcessBuilder;
05
06  // Get the information of date process.
07  public class ProcessInfo {
08
09      public static void main(String[] args) {
10          try {
11              //Create a process to run date program
12              Process p = new ProcessBuilder("date").start();
13              //Read the process output data as an input
14              BufferedReader bfr = new BufferedReader
15                  (new InputStreamReader(p.getInputStream()));
16              String line = bfr.readLine();
17              //Loop through the process output and print it until it is finished
18              while (line != null) {
19                  System.out.println(line);
20                  line = bfr.readLine();
21              }
22              System.out.println(p.pid());
23              System.out.println(p.isAlive());
24              System.out.println(p.exitValue());
25          } catch (Exception e) {
26              e.printStackTrace();
27          }
28      }
29  }
30
31 }
```

- Terminate a Process

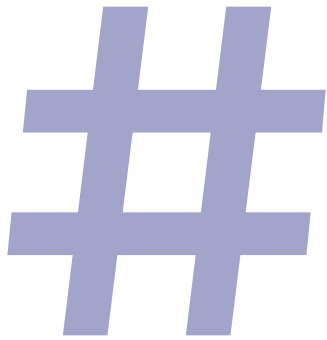
```

01  import java.io.BufferedReader;
02  import java.io.InputStreamReader;
03  import java.lang.Process;
04  import java.lang.ProcessBuilder;
05
06  // Terminate a process using destroy method.
07  public class ProcessTerminate {
08
09      public static void main(String[] args) {
10          try {
11
12              //Create a process to run date program
13              Process p = new ProcessBuilder("java", "--version").start();
14              //Read the process output data as an input
15              BufferedReader bfr = new BufferedReader
16                  (new InputStreamReader(p.getInputStream()));
17              String line = bfr.readLine();
18              //Loop through the process output and print it until it is finished
19              while (line != null) {
20                  System.out.println(line);
21                  line = bfr.readLine();
22              }
23              System.out.println(p.pid());
24              System.out.println("Is alive before termination: " + p.isAlive());
25              p.destroy();
26              System.out.println("Is alive after termination: " + p.isAlive());
27              System.out.println(p.exitValue());
28          } catch (Exception e) {
29              e.printStackTrace();
30          }
31      }
32  }
33
34  }
```

[Oracle docs: Class ProcessBuilder](#)

[Baeldung: Guide to java.lang.ProcessBuilder API](#)

[MacOS activity monitor](#)

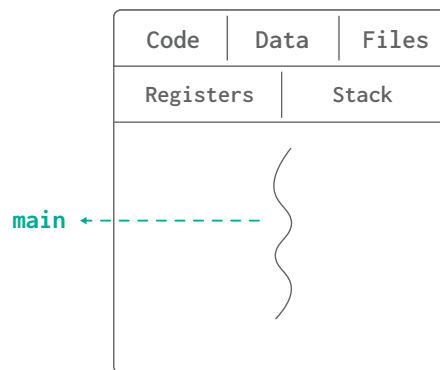


# **Thread Mangement**

## Step 01 Thread

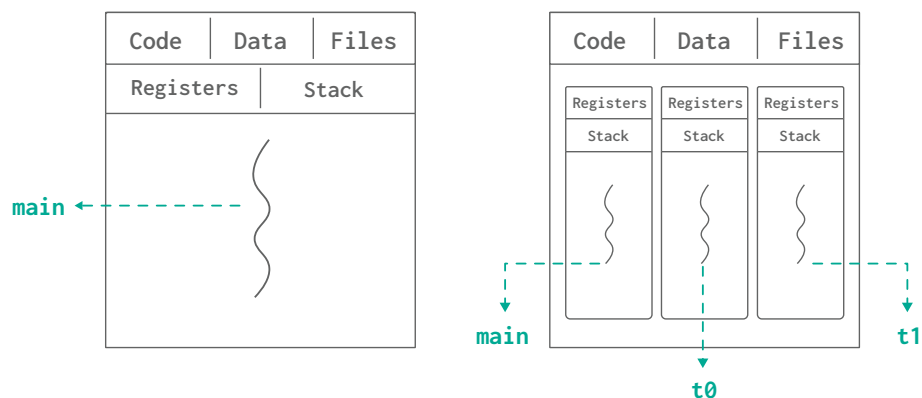
Have you ever used a program running several tasks at the same time? for example, a music player plays the music and at the same time, it shows notifications and recommendations. All of these tasks happen concurrently not sequentially (depending on the CPU cores). Also, the music player process does not utilize another process for other tasks but performs several tasks at the same time by itself. This can be achieved via threads.

A thread is a unit of execution. Every process has at least one thread to get executed. You can think of the thread as an execution arrow that reads instructions one after the other.



Having more than one thread in a single process means more independent execution units that can run concurrently at the same time.

Threads are small execution units inside of a process, and they share some resources as described in the image below.





## Step 02 Thread States

Threads go through a lifecycle that changes its state.

- **New:** A new thread have been created.
- **Ready:** the thread is simply ready for running.
- **Running:** the thread is currently running.
- **Waiting:** the thread is waiting for a specific event to execute.
- **Blocked:** the thread got blocked while running for a specific reason.  
For example, a sleep method have been executed, or it is waiting for an I/O device.
- **Dead/Terminated:** thread has been terminated.

## Step 03 Code

To create and manage threads in C, we will be using POSIX Threads (pthread) library as the following.

First, lets create a program without any threads. If there is no thread created, it means there is only one, which is the main thread to execute the program instructions

```
01 | #include <stdio.h>
02 |
03 |
04 | int main(){
05 |     printf("Hello from main thread\n");
06 |     return 0;
07 | }
```

### OUTPUT

Hello from main thread

Now, let's create a new thread (new execution flow) to print another statement which is "Hello from the new thread". This can be done by first including the thread library, creating the thread, and then joining the thread with the main execution flow as the following.

```

01  #include <stdio.h>
02  #include <pthread.h>
03
04  //A function to be executed by thread1
05  void* print_msg(){
06      printf("Hello from the new thread\n");
07      return NULL;
08  }
09  int main(){
10      //Declares a variable for the thread identifier
11      pthread_t thread1;
12      //Creates the new thread
13      pthread_create(&thread1, NULL, print_msg, NULL);
14      //Wait for thread1 to finish and then continue the execution
15      pthread_join(thread1, NULL);
16      printf("Hello from main thread\n");
17      return 0;
18  }

```

**OUTPUT**

```

Hello from the new thread
Hello from main thread

```

**pthread\_join()** is a function that will join two threads together, in our case, it will wait for thread1 to execute and then continue the execution of the main thread. If we changed its position to after the main thread print message, the output of the program may change, since we don't require thread1 to finish before printing Hello from main thread.

```

01  #include <stdio.h>
02  #include <pthread.h>
03
04  //A function to be executed by thread1
05  void* print_msg(){
06      printf("Hello from the new thread\n");
07      return NULL;
08  }
09  int main(){
10      //Declares a variable for the thread identifier
11      pthread_t thread1;
12      pthread_create(&thread1, NULL, print_msg, NULL); //Creates the new thread
13      printf("Hello from main thread\n");
14      //Wait for thread1 to finish and then continue the execution
15      pthread_join(thread1, NULL);
16      return 0;
17  }

```

**OUTPUT**

```

Hello from main thread
Hello from the new thread

```

The `pthread_create` takes several arguments, which are:

- **Thread ID:** A place to store the unique ID of the new thread.
- **Attributes:** Optional settings for the thread (like stack size). You can use default settings by passing **NULL**.
- **Function:** The function that the new thread will run.
- **Argument:** The data you want to send to the thread's function. You can pass NULL if you don't need to send anything.

You can also print the thread id using the variable we created as the following.

```

01  #include <stdio.h>
02  #include <pthread.h>
03
04  //A function to be executed by thread1
05  void* print_msg(){
06      printf("Hello from the new thread\n");
07      return NULL;
08  }
09  int main(){
10      pthread_t thread1;
11      pthread_create(&thread1, NULL, print_msg, NULL);
12      printf("Hello from main thread\n");
13      printf("thread id: %lu \n", (unsigned long)thread1); //Print thread1 id
14      pthread_join(thread1, NULL);
15      return 0;
16  }
```

#### OUTPUT

```

Hello from main thread
thread id: 6122795008
Hello from the new thread
```

#### NOTE

You can print the current thread id using `pthread_self()`

## Step 04 More examples in Java

To handle threads in Java, we will be working with the **Thread** class and its related classes and interfaces to create, read, and delete threads.

### 01 Create Thread

You can create a new thread by two ways.

- Extending Thread class

```

01 public class CreateThread1 {
02
03     public static void main(String[] args) {
04         Multithreading t0 = new Multithreading();
05         t0.start();
06     }
07 }
08
09 class Multithreading extends Thread{
10     @Override
11     public void run() {
12         System.out.println( "Thread is running");
13     }
14 }
```

#### NOTE

By extending the **Thread** class, you can use its methods such as **start()** to run the thread after creating it. Otherwise, the thread will not start running.

- Implementing Runnable interface

```

01 public class CreateThread2 {
02
03     public static void main(String[] args) {
04         Multithreading runnableThread = new Multithreading();
05         Thread t0 = new Thread(runnableThread);
06         t0.start();
07     }
08 }
09
10 class Multithreading implements Runnable{
11     @Override
12     public void run() {
13         System.out.println( "Thread is running");
14     }
15 }
```

There is a shortcut to implement an interface in Java without the effort to create a class as the following.

```

01 public class CreateThread2 {
02     public static void main(String[] args) {
03
04         Thread t0 = new Thread(new Runnable() {
05             @Override
06             public void run() {
07                 System.out.println( "Thread is running");
08             }
09         });
10         t0.start();
11     }
12 }
13

```

## 02 Read Thread

Every process has at least one thread running (main). Let us first print the current running thread in a java program.

```

01 public class ReadThread1 {
02
03     public static void main(String[] args) {
04         System.out.println(Thread.currentThread().getName()); //output: main
05     }
06 }

```

To create another thread and read it, you can use the Thread class methods such as `getName()`.

```

01
02 public class ReadThread2 {
03
04     public static void main(String[] args) {
05         Multithreading t0 = new Multithreading();
06         t0.start();
07     }
08 }
09
10 class Multithreading extends Thread{
11     @Override
12     public void run() {
13         System.out.println( "Thread " + this.getName() + " is running");
14     }
15 }

```

### OUTPUT

Thread Thread-0 is running

### 03 Terminate Thread

```

01 public class TerminateThread {
02
03     public static void main(String[] args) throws InterruptedException {
04         Multithreading t0 = new Multithreading();
05         t0.start();
06         System.out.println(t0.getState());
07     }
08 }
09
10 class Multithreading extends Thread{
11     @Override
12     public void run() {
13         System.out.println( "Thread " + this.getName() + " is running");
14     }
15 }

```

#### OUTPUT

RUNNABLE

Thread Thread-0 is running

As you can see, the RUNNABLE state was printed then the thread was executed. It did not terminate since the time the print statement was executed the thread was ready for execution hence RUNNABLE. Let's wait for 3 seconds before printing the thread state now.

```

01 public class TerminateThread2 {
02
03     public static void main(String[] args) throws InterruptedException {
04         Multithreading t0 = new Multithreading();
05         t0.start();
06         t0.sleep(3000);
07         System.out.println(t0.getState());
08     }
09 }
10
11 class Multithreading extends Thread{
12     @Override
13     public void run() {
14         System.out.println( "Thread " + this.getName() + " is running");
15     }
16 }

```

#### OUTPUT

Thread Thread-0 is running

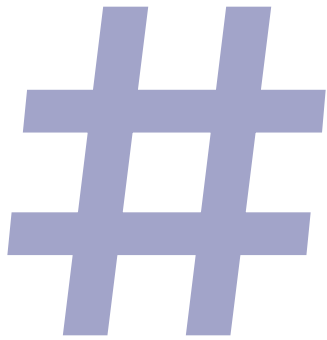
TERMINATED

we have given the thread the time to execute and terminate before printing its state. Therefore the current state of the thread is terminated.

## Step 05 Resources

[Threads in C](#)

[Runestone Academy](#)



# **Multiprocessing and Multithreading**



## Step 01 Multiprocessing and Multithreading

Creating processes and threads may and may not enhance the program or computer performance. It mainly depends on how you use them.

### 01 Multiprocessing

Multiprocessing means running several processes concurrently.

### 02 Multithreading

Multithreading is the process of creating multiple threads of a process to execute several functionalities concurrently.

### 03 Concurrency

Concurrency is the ability to execute more than one task at the same time; To utilize hardware resources, and speed up the execution process. It can be concluded using multiprocessing or multithreading concepts.

### 04 Context Switching

Context switching is the process of switching a running process or thread to another in the CPU.

Note that, switching between processes or threads which is called context switching is costly. Switching a running process will require a complete snapshot of the current state of the process execution and all its related data, which will be stored as a PCB (Process Control Block). Once the CPU is ready to execute the previously removed process, it will load all of its information and resume its execution. Context switching is time-consuming and costly, that is why you should apply multiprocessing and multithreading wisely.