A **process** is an executing instance of a program. The operating system (OS) manages multiple processes by allocating system resources such as CPU, memory, and I/O devices.

## ****Key Concepts in Process Management****

| **Concept** | **Description** |
| --- | --- |
| **Process ID (PID)** | A unique number assigned to each process. |
| **Parent Process** | A process that creates child processes using fork(). |
| **Child Process** | A new process created by a parent process. |
| **Process States** | A process goes through different states: New, Ready, Running, Blocked, Terminated. |
| **Process Table** | A data structure in the OS that tracks all running processes. |

## ****Process States & Life Cycle****

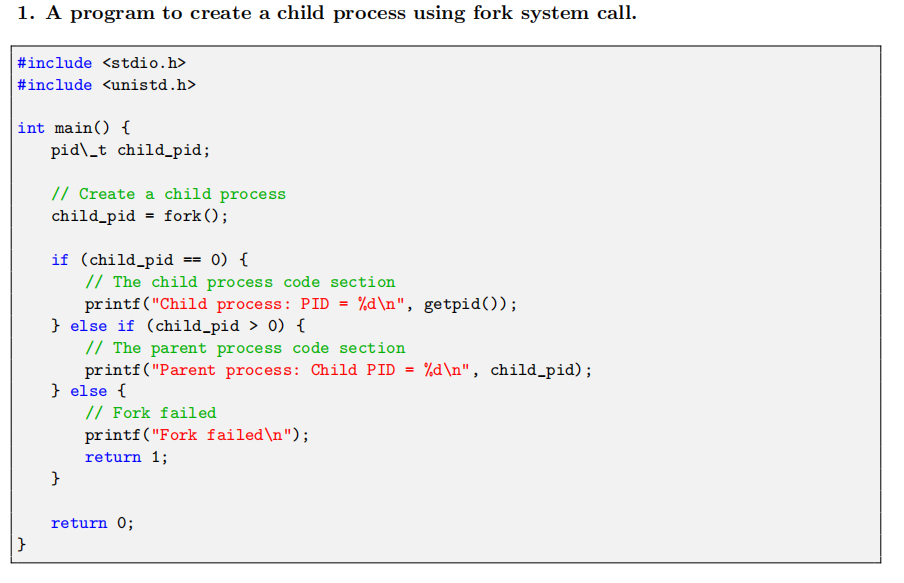
A process moves between different states:

* **New** → Process is being created.
* **Ready** → Process is waiting for CPU execution.
* **Running** → Process is currently executing.
* **Blocked** → Process is waiting for an event (e.g., I/O completion).
* **Terminated** → Process has completed execution.

# ****How a Process Works in an Operating System****

A **process** is an executing instance of a program. It includes:

* Code (Instructions)
* Data (Variables, Files)
* Resources (CPU, Memory)
* Execution Context (Registers, Program Counter)



| **Step No.** | **Code Statement** | **Process** | **Description** |
| --- | --- | --- | --- |
| 1 | #include <stdio.h> | - | Includes the standard input-output library for printf(). |
| 2 | #include <unistd.h> | - | Includes the unistd.h library for using fork(). |
| 3 | int main() { pid\_t child\_pid; } | - | Declares child\_pid variable to store the process ID. |
| 4 | child\_pid = fork(); | Parent | Creates a new child process. Returns:  **0** for the child process,  **>0** (child PID) for the parent process,  **-1** if fork fails. |
| 5 | if (child\_pid == 0) { | Child | Executes if the process is the **child** (fork returned 0). |
| 6 | printf("Child process: PID = %d\n", getpid()); | Child | Prints the **child process ID** using getpid(). |
| 7 | else if (child\_pid > 0) { | Parent | Executes if the process is the **parent** (fork returned child PID). |
| 8 | printf("Parent process: Child PID = %d\n", child\_pid); | Parent | Prints the **child’s process ID**. |
| 9 | else { printf("Fork failed\n"); return 1; } | - | If fork() fails, prints an error message and returns 1. |
| 10 | return 0; | Both | Ends both the **parent** and **child** processes. |

### fork()

Creates a **duplicate** process.

Parent gets **child’s PID**; child gets **0**.

If fork() fails, it returns -1.

* pid is the return value of fork().
* pid == 0: Code running in **child process**.
* pid > 0: Code running in **parent process** (value is PID of child).
* pid < 0: **Fork failed** (usually due to system resource limits).

### ****Key Concepts****

**Parent-Child Relationship:** The child inherits a copy of the parent’s memory but runs independently.

**Process Execution Order:** Parent and child **may execute in any order** (depends on CPU scheduling).

**Multiple** fork() **Calls:** Each fork() doubles the number of processes.

**2. C program to demonstrates the creation of an orphan process.**

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main() {

pid\\_t child\\_pid = fork();

if (child\_pid == 0) {

// Child process

printf("Child process: PID = %d\n", getpid());

sleep(2);

// Sleep to ensure the parent process terminates first

printf("Child process: My parent’s PID = %d\n", getppid());

}

else if (child\_pid > 0) { // Parent process

printf("Parent process: PID = %d\n", getpid());

printf("Parent process: Terminating...\n");

}

else {

printf("Fork failed\n");

return 1;

}

return 0;

}

| **Step** | **Code** | **Who Executes** | **What Happens** |
| --- | --- | --- | --- |
| fork() | pid\_t child\_pid = fork(); | Parent | Creates a child process. |
| if (child\_pid == 0) | Child process: | Child | Runs this block since fork() returned 0. |
| getpid() | printf("Child process: PID = %d\n", getpid()); | Child | Prints its own PID. |
| sleep(2); | sleep(2); | Child | Waits 2 seconds to let the parent exit. |
| getppid() | printf("Child process: My parent’s PID = %d\n", getppid()); | Child | Shows the PID of its current parent. Will likely be 1 if parent exited. |
| else if (child\_pid > 0) | Parent process: | Parent | Executes this block, prints own PID, and then terminates. |
| else | Fork failed | Both (if error) | If fork fails, prints an error message. |

This C program demonstrates how to create an **orphan process** using fork(), sleep(), and getppid().

* The **parent process** creates a child.
* The **parent exits immediately** after creation.
* The **child sleeps**, ensuring the parent exits before it continues.
* When the child wakes up, it checks its parent — now it has been **adopted by** init **(PID 1)**, making it an **orphan process**.

### ****Demonstration of Orphan Process in Tabular Form****

| **Step** | **Action** | **Parent (PID)** | **Child (PID)** | **Parent Status** | **Child Status** | **Child's Parent (PPID)** |
| --- | --- | --- | --- | --- | --- | --- |
| **1** | fork() is called | 1001 | 1002 (new process) | Running | Running | 1001 (Parent) |
| **2** | Parent prints its PID | 1001 | 1002 | Running | Running | 1001 |
| **3** | Parent prints "Terminating..." | 1001 | 1002 | Running | Running | 1001 |
| **4** | Parent terminates | ❌ (Terminated) | 1002 | Terminated | Running | **Reassigned to** 1 **(**init/systemd**)** |
| **5** | Child sleeps for 2 sec | ❌ | 1002 | ❌ | Sleeping | 1 (init/systemd) |
| **6** | Child wakes up and prints PID | ❌ | 1002 | ❌ | Running | 1 |
| **7** | Child prints "My parent’s PID = 1" | ❌ | 1002 | ❌ | Running | 1 |
| **8** | Child terminates | ❌ | ❌ | ❌ | ❌ | ❌ |

### ****What is a Zombie Process ?****

A **zombie process** is a child process that has **terminated**, but its **exit status has not been read by the parent**.

It remains in the **process table** until the parent process **collects its exit status using** wait() or **terminates**.

### ****How the Program Creates a Zombie Process****

### The ****child process terminates immediately**** (exit(0);)

The **parent process sleeps for 10 seconds**, keeping the child's **process ID in the system** but not calling wait().

During this **sleep period**, the child process is a **zombie**.

Once the parent process terminates, the **zombie process is cleaned up by the OS** (adopted by init).

**C program to demonstrate the creation of a Zombie process.**

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

#include <stdlib.h>

int main() {

pid\\_t child\\_pid = fork();

if (child\\_pid == 0) {

// Child process

printf("Child process: PID = %d\n", getpid());

exit(0); // Child process exits immediately

} else if (child\_pid > 0) {

// Parent process

printf("Parent process: PID = %d\n", getpid());

printf("Parent process: Child PID = %d\n", child\_pid);

sleep(10); // Sleep to allow time for the child to become a zombie

printf("Parent process: Terminating...\n");

} else {

printf("Fork failed\n");

return 1;

}

return 0;

}

| **Step No.** | **Code Statement** | **Process** | **Description** |
| --- | --- | --- | --- |
| 1 | #include <stdio.h> | - | Includes the standard input-output library for printf(). |
| 2 | #include <sys/types.h> | - | Includes system data types like pid\_t. |
| 3 | #include <unistd.h> | - | Includes the fork(), getpid(), sleep(), and exit() functions. |
| 4 | #include <stdlib.h> | - | Includes the exit() function for controlled termination of the child process. |
| 5 | int main() { pid\_t child\_pid = fork(); } | Parent | Creates a **child process** using fork(). Returns:  **0** for the child process,  **>0** (child’s PID) for the parent process,  **-1** if fork fails. |
| 6 | if (child\_pid == 0) { | Child | Executes only in the **child process** (when fork() returns 0). |
| 7 | printf("Child process: PID = %d\n", getpid()); | Child | Prints the **child process ID (PID)**. |
| 8 | exit(0); | Child | Child **terminates immediately**, leaving an **entry in the process table** (a zombie process). |
| 9 | else if (child\_pid > 0) { | Parent | Executes only in the **parent process** (when fork() returns child’s PID). |
| 10 | printf("Parent process: PID = %d\n", getpid()); | Parent | Prints the **parent process ID (PID)**. |
| 11 | printf("Parent process: Child PID = %d\n", child\_pid); | Parent | Prints the **child process ID** assigned to it. |
| 12 | sleep(10); | Parent | The **parent process sleeps for 10 seconds**, keeping the **zombie process in the system**. |
| 13 | printf("Parent process: Terminating...\n"); | Parent | Parent prints a termination message before exiting. |
| 14 | else { printf("Fork failed\n"); return 1; } | - | If fork() fails, prints an error message and returns 1. |
| 15 | return 0; | Both | Ends both parent and child processes. |

## ****Example Output****

## Child process: PID = 12345

## Parent process: PID = 12344

Parent process: Child PID = 12345

(Child process exits here)

(Parent sleeps for 10 seconds)

Parent process: Terminating...

**Before 10 seconds:** The child has exited but still appears in the **process table** (zombie).

**After 10 seconds:** The parent terminates, and the **OS cleans up the zombie process**.

**4. C Program to Demonstrate the Use of** execv() **System Call**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

pid\\_t pid; // Variable to hold the process ID

char \*args[] = {"ls", "-l", NULL}; // Arguments for the command

// Create a new process

pid = fork();

if (pid < 0) {

// If fork() fails, print an error message

printf("fork failed");

return -1;

}

if (pid == 0) {

// This block is executed by the child process

// Print a message to indicate the child process is running

printf("Child process (PID: %d) running ’ls -l’ command...\n", getpid());

// Execute the ’ls -l’ command

execv("/bin/ls", args);

// If execv() fails, print an error message

printf("execv failed");

exit(-1); // Exit the child process with an error status

} else {

// This block is executed by the parent process

// Wait for the child process to finish

wait(NULL);

// Print a message to indicate that the child process has finished

printf("Parent process (PID: %d): Child process has finished.\n", getpid());

}

return 0; // Return success

}

### ****1.**** fork()

Creates a **duplicate** process.

Parent gets **child’s PID**; child gets **0**.

If fork() fails, it returns -1.

### execv(): ****Replacing a Process with a New Program****

When you call exec(), the current process is **completely replaced** by the new program.The new program **inherits** the same process ID (PID) and open file descriptors of the original process.

* Replaces the **child process** with a **new program** (ls -l).
* If successful, execv() **never returns** (it replaces the process).
* If it fails, the error message "execv failed" is printed.

### ****Meaning of**** execv("/bin/ls", args);

This line **replaces the current process** with the /bin/ls program, which is the ls command used to list files in a directory.The execv() function is a system call in Unix/Linux that **replaces the current process** with a new executable program. It is part of the exec family of functions.

## ****Syntax:**** int execv(const char \*path, char \*const argv[])

path → The absolute or relative path to the executable program.

argv[] → An array of strings representing the command-line arguments. The first element (argv[0]) is conventionally the name of the program, and the last element **must be NULL**.

**Breakdown of the Components:** execv("/bin/ls", args);

execv → A system call that replaces the current process with a new program.

"/bin/ls" → The full path to the ls executable (which lists directory contents).args → An array of command-line arguments (char \*args[]).

### ****Meaning of**** execv("/bin/ls", args);

This line **replaces the current process** with the /bin/ls program, which is the ls command used to list files in a directory.

char \*args[] = {"/bin/ls", "-l", NULL};

args[0] = "/bin/ls" → The program itself.

args[1] = "-l" → A command-line option to display files in long format.

args[2] = NULL → Indicates the end of arguments (mandatory).

### ****3.**** wait()

Ensures the **parent waits** until the child **finishes execution**.

Prevents **orphan** or **zombie processes**.

| **Step No.** | **Code Statement** | **Process** | **Description** |
| --- | --- | --- | --- |
| 1 | #include <stdio.h> | - | Includes standard I/O functions like printf(). |
| 2 | #include <stdlib.h> | - | Includes functions like exit(). |
| 3 | #include <unistd.h> | - | Includes fork(), execv(), and getpid(). |
| 4 | #include <sys/types.h> | - | Includes data types like pid\_t. |
| 5 | #include <sys/wait.h> | - | Includes wait() for waiting on child processes. |
| 6 | pid\_t pid; | Parent | Declares a variable to store the process ID. |
| 7 | char \*args[] = {"ls", "-l", NULL}; | Parent | Defines an argument array for the ls -l command. |
| 8 | pid = fork(); | Parent | **Creates a child process**. Returns: **0** (child process), **>0** (child’s PID for parent), **-1** if failed. |
| 9 | if (pid < 0) { printf("fork failed"); return -1; } | Parent | Checks if fork() failed and exits if so. |
| 10 | if (pid == 0) { | Child | Executes only in the **child process**. |
| 11 | printf("Child process (PID: %d) running 'ls -l' command...\n", getpid()); | Child | Displays a message indicating the child process is executing. |
| 12 | execv("/bin/ls", args); | Child | **Replaces child process** with ls -l execution. |
| 13 | printf("execv failed"); exit(-1); | Child | If execv() fails, prints an error message and exits with -1. |
| 14 | else { wait(NULL); | Parent | **Parent waits** for the child process to finish execution. |
| 15 | printf("Parent process (PID: %d): Child process has finished.\n", getpid()); | Parent | Displays a message indicating child has finished. |
| 16 | return 0; | Both | Ends both parent and child processes. |

****Program Explanation: Sending a Signal to Terminate a Child Process in C****

This program demonstrates **inter-process communication using signals**. The **parent process** creates a **child process** using fork(). The child process runs indefinitely, waiting for a signal using pause(). The **parent process sends a** SIGTERM **signal** to terminate the child, then waits for it to exit.

**kill() **System Call in Linux****

The kill() function is used in Linux to **send signals to a process**. It can be used to **terminate, pause, or communicate with a process** using different signals.

**Syntax**

#include <sys/types.h>

#include <signal.h>

int kill(pid\_t pid, int sig);

| **Parameter** | **Description** |
| --- | --- |
| pid | The **Process ID (PID)** of the target process. |
| sig | The **signal number** to be sent (e.g., SIGTERM, SIGKILL, SIGSTOP). |

### ****Return Value****

### 0 ****on success****

* -1 **on failure** (e.g., if the process does not exist)

**Commonly Used Signals with** kill()

| **Signal** | **Number** | **Effect** |
| --- | --- | --- |
| SIGTERM | 15 | Gracefully asks the process to terminate (default signal for kill <pid>). |
| SIGKILL | 9 | Forcefully kills the process (cannot be ignored). |
| SIGSTOP | 19 | Stops (pauses) a process (can be resumed with SIGCONT). |
| SIGCONT | 18 | Resumes a stopped process. |
| SIGHUP | 1 | Sends a hangup signal (often used to reload configuration files). |
| SIGINT | 2 | Interrupts a process (same as Ctrl+C). |

## ****Step-by-Step Execution Breakdown****

| **Step No.** | **Code Statement** | **Process** | **Description** |
| --- | --- | --- | --- |
| 1 | #include <stdio.h> | - | Includes standard I/O functions like printf(). |
| 2 | #include <stdlib.h> | - | Includes functions like exit(). |
| 3 | #include <unistd.h> | - | Includes fork(), pause(), and kill(). |
| 4 | #include <signal.h> | - | Includes signal handling functions. |
| 5 | #include <sys/types.h> | - | Includes data types like pid\_t. |
| 6 | #include <sys/wait.h> | - | Includes wait() for waiting on child processes. |
| 7 | pid\_t pid; | Parent | Declares a variable to store the process ID. |
| 8 | pid = fork(); | Parent | **Creates a child process**. |
| 9 | if (pid < 0) { printf("fork failed"); return -1; } | Parent | Checks if fork() failed and exits if so. |
| 10 | if (pid == 0) { | Child | Executes only in the **child process**. |
| 11 | printf("Child process (PID: %d) is running...\n", getpid()); | Child | Displays a message indicating the child process is executing. |
| 12 | while (1) { pause(); } | Child | **Child waits indefinitely** for a signal. |
| 13 | else { printf("Parent process( %d) sending signal to child( %d)\n",getpid(),pid); } | Parent | Displays a message indicating the parent is sending a signal. |
| 14 | kill(pid, SIGTERM); | Parent | **Sends** SIGTERM **signal** to terminate the child process. |
| 15 | wait(NULL); | Parent | **Waits** for the child process to terminate. |
| 16 | printf("Parent process: Child process has been terminated.\n"); | Parent | Confirms that the child process has been terminated. |

### ****1.** fork()**

* Creates a **duplicate** process.
* Parent gets **child’s PID**; child gets **0**.
* If fork() fails, it returns -1.

### **2.pause():** Makes the **child process wait indefinitely** for a signal.The process **resumes only when a signal is received**.

### **3.** kill(pid, SIGTERM): Sends a **SIGTERM signal** to the child process to **gracefully terminate it**.

### **4.** wait(NULL): Ensures the **parent waits** until the child **finishes execution**.

### ****Analogy: An Employee Waiting for Instructions****

| Concept | Real-Life Equivalent |
| --- | --- |
| pause() | Employee waiting at their desk for a task. |
| kill() | Boss sending an email to the employee with instructions. |
| SIGTERM | Boss telling the employee to **stop working and leave**. |
| SIGKILL | Boss **firing the employee immediately**, no discussion. |