Process Management

Process

A process is a program in execution on a machine, defined as a sequential stream of execution in its own address space. It includes the code to be executed, the program's data, and a stack for function calls. Each process runs independently, and the operating system manages it.

Process Address Space

A process's address space is the range of memory locations it can access, typically from a minimum to a maximum address. It includes:

- **Executable program**: The code that the process executes.
- **Program's data**: Variables and dynamically allocated memory (heap).
- **Stack**: Holds function calls and local variables.

Process Components:

- **Code (Text) section**: The instructions to be executed.
- **Data section**: Stores global and static variables.
- **Stack**: Handles function calls and local variables.
- **Heap**: Dynamically allocated memory.
- **CPU state**: Registers like the Program Counter (PC) and Stack Pointer (SP), which store the current state of the CPU during process execution.
- **Environment**: Interaction with the external system, such as open files or communication channels with other processes.

Process vs. Program

- **Program**: A static set of instructions (e.g., C or assembly code).
- **Process**: A program in execution, with its own memory space and active state.

CPU State and Memory Contents

- **CPU registers**: Include the current state of the process, such as the **Program Counter (PC)**, which holds the address of the next instruction, and the **Stack Pointer (SP)**, which points to the top of the stack.
- **Memory segments**:
  - **Text/application code**: Instructions to execute.
  - **Data**: Variables.
  - **Heap**: Dynamically allocated memory.

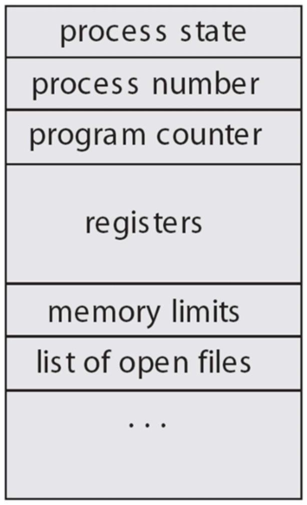
    - **Stack**: Function calls and local variables.

Process Control Block (PCB)

The **Process Control Block (PCB)** is a data structure in the OS that holds all the essential information about a process, making it the OS's representation of a process. The PCB includes:

- **Process state**: Current state (e.g., running, waiting).
- **Program counter**: Address of the next instruction.
- **CPU registers**: Current values of CPU registers.
- **Memory management information**: Addresses and permissions.
- **I/O status**: List of I/O devices allocated to the process.
- **Process ID (PID)**: Unique identifier.
- **Parent PID**: The process that spawned this process.

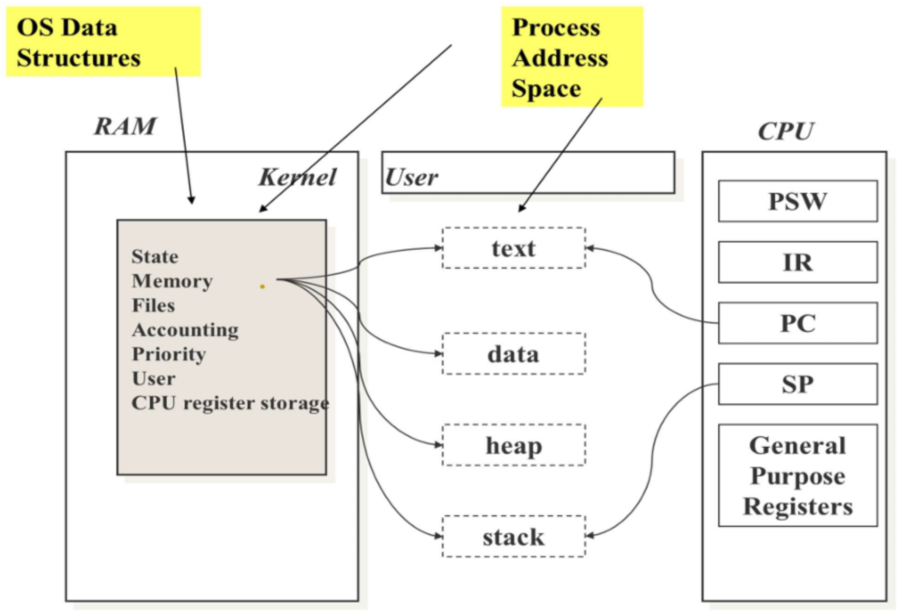
The PCB is stored in protected memory, ensuring user processes cannot tamper with it.



Process Identification

Each process is identified by a unique **Process ID (PID)** and a **User ID**, which helps the operating system determine access rights and security restrictions for that process.

Process Control



CPU Switching Between Processes

When the CPU switches from one process to another, it involves several steps to ensure that each process resumes execution correctly. This process is known as **Context Switching**.

Context Switch

Context Switching is the process of saving the state of the currently running process and loading the state of the next process to be executed. It involves:

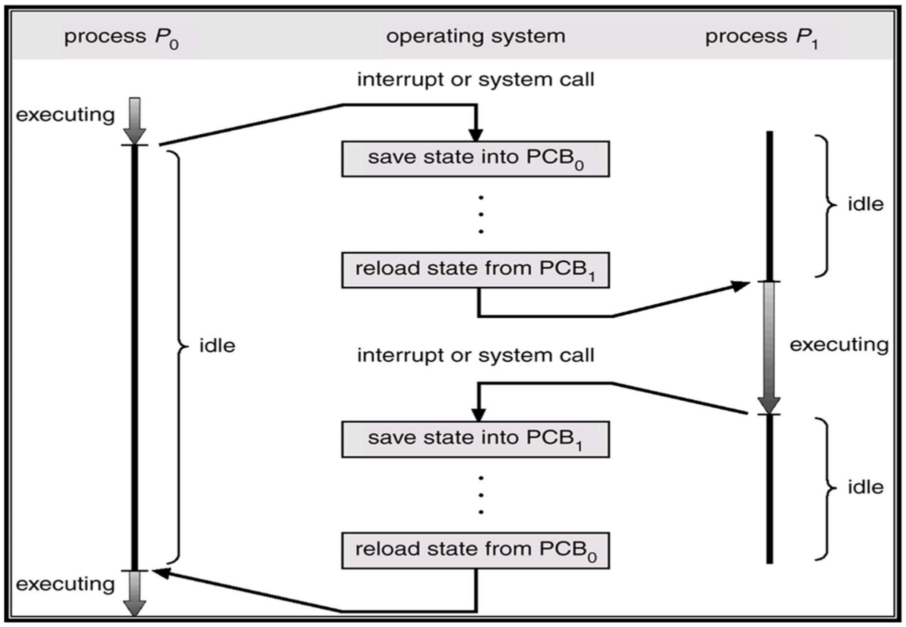
1. **Saving the State of the Old Process**:
    - **Registers**: The values of CPU registers, such as the Program Counter (PC) and Stack Pointer (SP), are saved into the Process Control Block (PCB) of the old process.
    - **Memory Management Information**: This includes information about the process's memory allocation.
2. **Loading the State of the New Process**:
    - **Registers**: The saved register values of the new process are loaded from its PCB into the CPU.
    - **Program Counter**: The PC is set to the address where the new process should resume execution.

Dispatcher

The **Dispatcher** is the part of the operating system responsible for performing context switches. Its tasks include:

- **Selecting** the next process to run based on the scheduling algorithm.
- **Saving** the state of the currently running process.
- **Loading** the state of the selected process.

- **Transferring** control to the newly loaded process, ensuring it resumes execution from where it was last stopped.

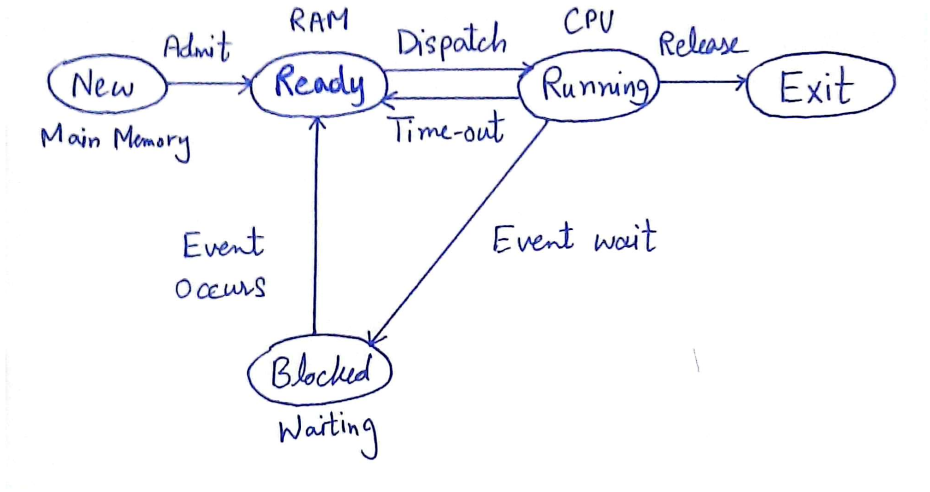


Process States

In process management, the state of a process determines its current status and is critical for process scheduling and execution. The process can be in various states depending on its lifecycle. Here's an overview of both the Five-State and Seven-State models.

Five-State Process Model

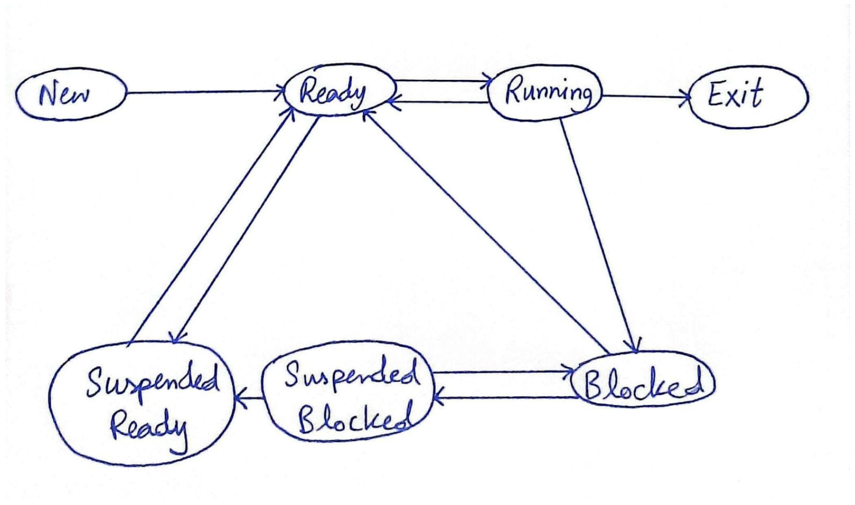
1. **Running**: The process is currently being executed by the CPU.
2. **Ready**: The process is in main memory and ready to run, but is waiting for CPU time.
3. **Blocked**: The process is waiting for an event or I/O operation to complete.
4. **New**: The process has just been created and has not yet been admitted to the set of runnable processes.
5. **Exit**: The process has completed execution or has been terminated due to an error.



Seven-State Process Model

The Seven-State Process Model expands on the Five-State Model by adding more granularity:

1. **New**: The process is being created and is not yet admitted to the system.
2. **Ready**: The process is in main memory and ready to execute, but is waiting for CPU allocation.
3. **Running**: The process is currently being executed by the CPU.
4. **Blocked**: The process is waiting for an I/O operation or another event to complete.
5. **Exit**: The process has completed execution or has been terminated due to an error.
6. **Suspended Ready**: Process that was initially in the ready state but was swapped out of main memory and placed onto external storage by the scheduler is said to be in suspend ready state. The process will transition back to a ready state whenever the process is again brought onto the main memory.
7. **Suspended Blocked**: Similar to suspend ready but uses the process which was performing I/O operation and lack of main memory caused them to move to secondary memory. When work is finished it may go to suspend ready.



Scheduling Queues

Scheduling Queues are used to manage processes in different states. They are typically implemented as linked lists:

1. **Job Queue**: Contains all processes in the system, including those that are waiting to be loaded into main memory.
2. **Ready Queue**: Contains processes that are in main memory and are ready to run. These processes are waiting for CPU time.
3. **Device Queues**: Contains processes that are waiting for I/O operations to complete.

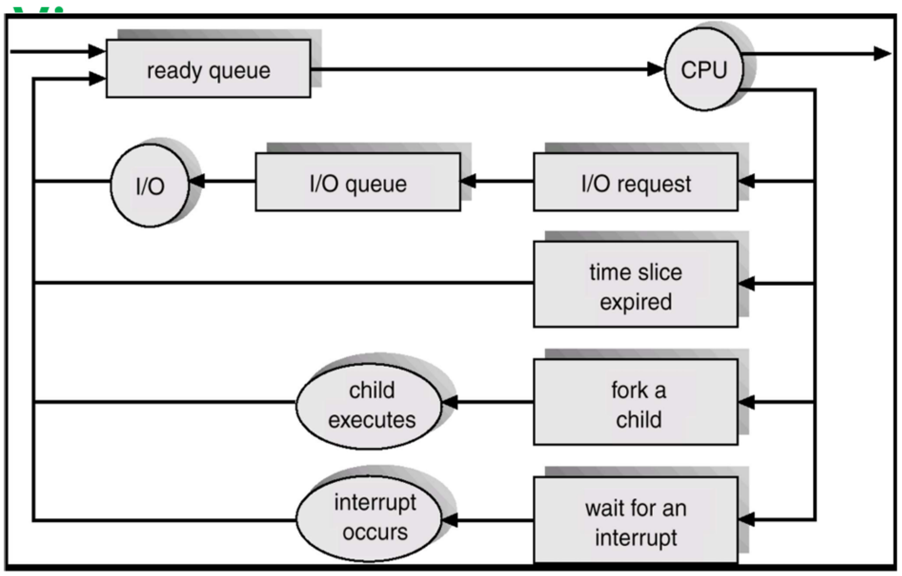
Queue Structure:

- **Queue Header**: Points to the first and last PCB (Process Control Block) in the queue.
- **PCB Pointer**: Each PCB has a pointer field that links it to the next PCB in the queue.

Process Migration: Processes move between these queues based on their state and system conditions, such as waiting for I/O or being scheduled for execution.

Schedulers

- **Long-Term Scheduler**: Determines which processes to admit into the system from the job queue.
- **Medium-Term Scheduler**: Handles the swapping of processes between main memory and secondary storage, managing their readiness.
- **Short-Term Scheduler**: Chooses which process from the ready queue should be executed by the CPU.



Operation on Processes

Process Creation in Operating Systems

When a process is created, it may generate other processes, forming a **tree of processes**. The process creation mechanism involves resource sharing, execution coordination, and system calls.

Key Elements of Process Creation:

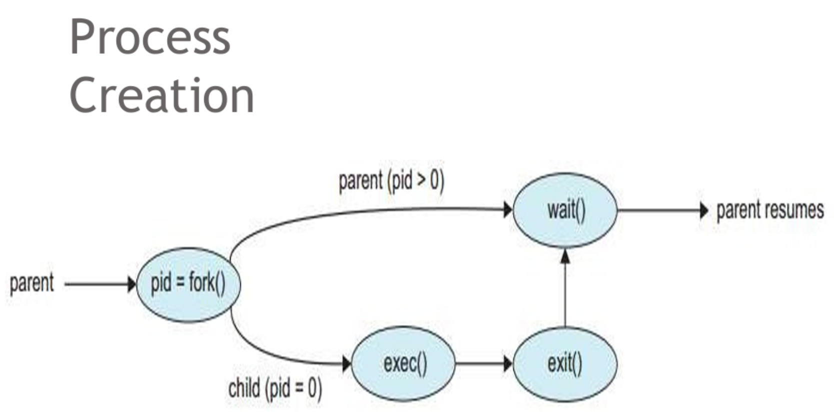
1. **Resource Sharing**:
    - **Parent and children share all resources**: Both the parent and child processes access and use the same resources (like memory or files).
    - **Children share a subset of the parent's resources**: Only some resources are shared between parent and child, while others are kept separate.
    - **Parent and child share no resources**: In this case, the parent and child processes have completely independent resources, meaning they do not access or interfere with each other's resources.
2. **Execution**:
    - **Concurrent execution**: The parent and child processes run simultaneously.
    - **Sequential execution**: The parent waits for the child process to terminate before proceeding.
3. **Address Space**:
    - The child process can be a **duplicate of the parent** (inheriting the same address space).
    - Alternatively, the **child can load a new program**, changing its address space.

System Calls for Process Creation:

1. `fork()`:

- Creates a new process (child), which is a duplicate of the parent.
- Both the parent and child execute the code following the `fork()` system call.
- The **parent receives the child's PID**, while the **child receives 0**.

2. `exec()`:
    - Used to replace the child process's memory space with a new program.
    - After a `fork()`, the `exec()` system call can be invoked to run a new program.



Example: `fork()` System Call

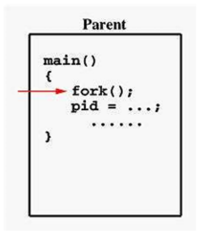
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    fork();
    printf("Hello world!\\n");
    return 0;
}
```

Output:

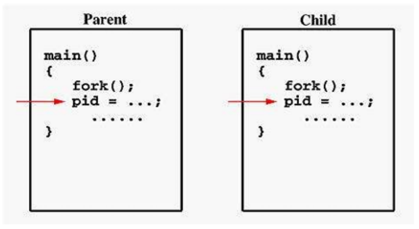
```
Hello world!
Hello world!
```

This occurs because both the parent and child processes execute the `printf()` statement after the `fork()` call.

□ Before calling fork()

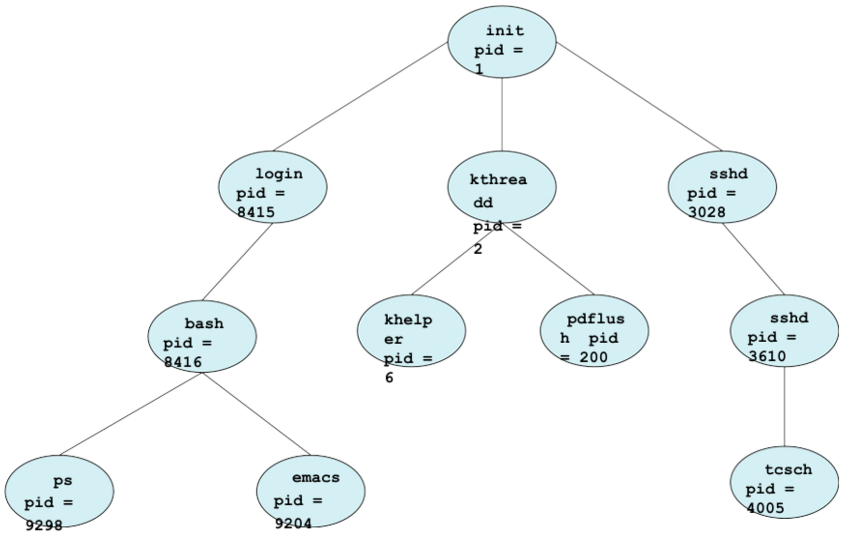


□ After calling fork()



Tree of Processes in Linux

In Unix/Linux systems, all processes are descendants of the **init process** (PID = 1), and they form a tree structure. This can be visualized using the `pstree` command.



Example: Fork Output with `forkexample()`

```
void forkexample() {
    int x = 1;
    if (fork() == 0)
        printf("Child has x = %d\\n", ++x);
    else
        printf("Parent has x = %d\\n", --x);
}

int main() {
    forkexample();
    return 0;
}
```

Output:

```
Parent has x = 0
Child has x = 2
```

Inheritance in Child Process:

- **Inherits**: Stack, memory, environment, file descriptors, and resource limits.
- **Does Not Inherit**: Process ID, parent process ID, process times, and resource utilization.

This allows the child process to function independently from the parent while inheriting necessary information for execution.

`Wait()` System Call:

The `wait()` system call allows a parent process to pause execution until one of its child processes has finished.

How the `wait()` System Call Works:

- `wait()`: This system call blocks the parent process until one of its child processes finishes.

  Syntax:

  ```
  wait(NULL);
  ```

- **`waitpid()`**: Unlike `wait()`, `waitpid()` allows the parent to specify which child process to wait for, using its process ID. This is useful when the parent has multiple child processes and needs to wait for a specific one.

Example of `wait()` in C:

```
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    if (fork() == 0) {
        printf("HC: hello from child\\n");
        return 0;
    } else {
        printf("HP: hello from parent\\n");
        wait(NULL);  // Parent waits for child to terminate
```

```
        printf("CT: child has terminated\\n");
    }
    printf("Bye\\n");
    return 0;
}
```

Possible Outputs:

1. **Case 1**:
    - `HP: hello from parent`
    - `HC: hello from child`
    - `CT: child has terminated`
    - `Bye`
2. **Case 2**:
    - `HC: hello from child`
    - `HP: hello from parent`
    - `CT: child has terminated`
    - `Bye`

Both cases are valid depending on which process (parent or child) gets scheduled first. However, the final two lines always appear after the child has terminated due to the `wait()` call ensuring synchronization.

Process Termination

A process can terminate for several reasons. When a process has finished executing all its statements or encounters an error, it requests the operating system to delete it. This process of termination can happen voluntarily (through normal completion) or be forced by its parent or the operating system.

Key Points of Process Termination:

1. **Process completes execution**:

    The process runs its final statement and uses the `exit()` system call to terminate.

    - **Output to parent**: The child process passes its exit status to the parent using the `wait()` system call.
    - **Resource deallocation**: The operating system deallocates all resources that were assigned to the process.
2. **Parent can terminate child processes**:

    In certain cases, the parent process may terminate its children (abort), including when:

    - The child has exceeded its allocated resources.
    - The task assigned to the child is no longer necessary.

    - The parent process itself is terminating.

3. **Cascading termination**:

   If a parent process terminates, the operating system often does not allow the child to continue running. This leads to the child also being terminated, which is known as **cascading termination**.

`exit()` System Call

The `exit()` system call is used by processes to terminate their execution.

Functionality:

```
exit(0);
```

When a process calls `exit()`:

- **Result value**: The process returns a result (exit status) to the parent.
- **Resource cleanup**:
    - All open files and connections are closed.
    - Memory allocated to the process is freed.
    - OS structures supporting the process are deallocated.

Zombie vs. Orphan Process

A **zombie process** and an **orphan process** are different types of processes in operating systems, particularly in Unix and Unix-like systems.

Orphan Process:

- An **orphan process** is a process that is still running but its parent has terminated.
- Orphan processes do not become zombies; instead, they are adopted by the **init** process (process ID 1).
- **Init** takes responsibility for waiting on orphan processes, ensuring that they are cleaned up when they terminate.

Zombie Process:

- A **zombie process**, also known as a **defunct process**, is one that has completed its execution but remains in the **process table**.
- The operating system keeps a record of the zombie process so the parent can read the child's **exit status** via the `wait()` system call.
- Zombie processes still occupy an entry in the process table, but unlike normal processes, they do not consume any other resources such as memory or CPU.
- If the parent does not call `wait()`, the zombie process remains in the process table, and its process ID (PID) is held until reaped by the parent.

Zombie Process Details:

- **Reaping zombies**: The parent process uses the `wait()` system call to retrieve the exit status of its terminated child and remove it from the process table.
- **Handling SIGCHLD**: Instead of sequentially calling `wait()`, the parent can handle the **SIGCHLD** signal to automatically reap zombies when a child process terminates.
- **Reallocation of PIDs**: Once a zombie process is removed, its PID can be reused by the system for new processes.
- **Preventing zombies**: If the parent explicitly ignores the **SIGCHLD** signal by using the `SIG_IGN` handler or sets the `SA_NOCLDWAIT` flag, the system will discard the child's exit status, preventing the creation of zombie processes.

Issues with Zombies:

- **Process table entries**: The primary concern with zombies is that they take up entries in the process table, which has a limited number of slots. If too many zombie processes accumulate, the system could run out of process IDs.
- **Removing zombies**: Zombies can be removed by manually sending the **SIGCHLD** signal to the parent. If the parent refuses to reap the zombie, the parent process may need to be terminated. When the parent is killed, `init` takes over, reaping any zombie processes.

Here are examples for both **zombie** and **orphan** processes.

Zombie Process Example:

Zombie process occurs when a child process terminates, but its parent hasn't yet called `wait()` to collect its exit status.

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

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        // Fork failed
        perror("Fork failed");
        exit(1);
    } else if (pid == 0) {
        // Child process
        printf("Child process (PID: %d) exiting.\\n", getpid());
        exit(0);  // Child terminates
    } else {
        // Parent process
        printf("Parent process (PID: %d) running. Child is now a zombie.\\n",
getpid());
        sleep(10);  // Parent sleeps, not calling wait(), making the child a
zombie
        // Now call wait() to reap the zombie
        wait(NULL);
        printf("Parent reaped the zombie. Exiting.\\n");
    }
    return 0;
}
```

Explanation:

- The child process terminates immediately after `fork()`, but the parent does not call `wait()` right away.
- During the sleep period, the child process becomes a **zombie**.
- After 10 seconds, the parent calls `wait()`, reaping the child and removing the zombie.

Orphan Process Example:

Orphan process occurs when a parent process terminates while the child process is still running, leaving the child to be adopted by the `init` process.

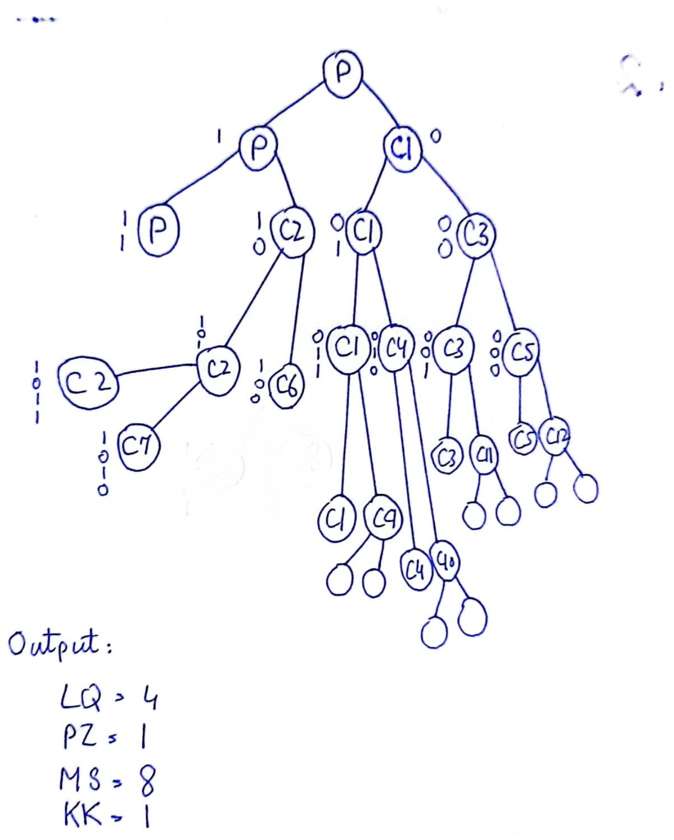
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        // Fork failed
        perror("Fork failed");
        exit(1);
    } else if (pid == 0) {
        // Child process
        sleep(5);  // Simulate the child still running after the parent exits
        printf("Orphan process (PID: %d) adopted by init (PPID: %d)\\n", getp
id(), getppid());
    } else {
        // Parent process
        printf("Parent process (PID: %d) exiting.\\n", getpid());
        exit(0);  // Parent terminates
    }
    return 0;
}
```

Explanation:

- The parent process terminates after forking the child.
- Since the parent has terminated, the child process becomes an **orphan**.
- After 5 seconds, the child process checks its parent's PID (`getppid()`), which now shows **1** (i.e., `init` has adopted it).

Process Tree:

```
#include <iostream>
#include <unistd.h>  // For fork()
using namespace std;

int main() {
    pid_t child_pid1;
    pid_t child_pid2;

    child_pid1 = fork();  // First fork
    child_pid2 = fork();  // Second fork

    // First condition: child process from the first fork
    if (child_pid1 == 0) {
        fork();  // Third fork inside the child process
        cout << "Lahore Qalandars" << endl;
    }

    // Second condition: child process from the second fork and not from the fir
    if (child_pid2 == 0 && child_pid1 != 0) {
        if (fork() && fork()) {
            cout << "Peshawar Zalmi" << endl;
        }
    }
    // Third condition: either child process from the first fork or second fork
    else if (child_pid2 == 0 || child_pid1 == 0) {
        if (fork() || fork()) {
            cout << "Multan Sultans" << endl;
        }
    }
    // Default case for the remaining processes
    else {
        cout << "Karachi Kings" << endl;
    }

    return 0;
}
```



Output:

```
LQ = 4
PZ = 1
MS = 8
KK = 1
```

`exec()` System Call:

The `exec()` family of functions in Unix-like operating systems replaces the current process image with a new process image. This is commonly used to run a different program within the same process. Here's a breakdown of the key points and an example:

Key Points of `exec()` Family:

1. **Variants:** The `exec` family includes functions like `execl`, `execv`, `execle`, `execve`, `execlp`, and `execvp`. They vary in how they take the program arguments and environment variables.
2. **No Return on Success:** When `exec()` successfully executes a new process, the current process image is replaced, and the new program runs in its place. This means that there is no return to the calling process unless an error occurs.
3. **Inheritance:** The new process retains certain attributes from the old process, such as:
    - Process ID

- Parent process ID
- Process group ID
- Real user ID, real group ID, supplementary group IDs
- Current working directory, root directory

4. **Common Usage:** Typically used in child processes to run a new executable file after a `fork()`.

Example:

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    printf("Using *execl* to exec ls -l...\n");
    execl("/bin/ls","ls","-l",NULL);
    printf("Program Terminated\n");
    return 0;
}
```

The above program illustrates the function of `exec()`. `execl()` is one of the family members of `exec()`, which is responsible for calling external programs.

First the program will print out `Using *execl* to exec ls -l...`

the program will execute `/bin/ls` program, with the supplied arguments.

After this step, the code of the process is *changed* to the target program and it **never** returns to the original code. As the result, the line `Program Terminated` is not printed.

1. `execl()`

- **Use:** Takes the file to be executed and a list of arguments. Each argument is passed individually.
- **Example:**

```
#include <stdio.h>
#include <unistd.h>

int main() {
    // Replace current process with the 'ls -l' command
    execl("/bin/ls", "ls", "-l", NULL);

    // If execl fails
    perror("execl failed");
    return 1;
}
```

- **Explanation:** This calls `ls -l`. The first argument is the path to the executable, and subsequent arguments are command-line parameters. The list of arguments must be terminated with `NULL`.

2. `execlp()`

- **Use:** Works like `execl()` but searches for the file in the directories listed in the `PATH` environment variable (like typing a command in the shell).
- **Example:**

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    // Replace current process with 'ls -l', searching in PATH
    execlp("ls", "ls", "-l", NULL);

    // If execlp fails
    perror("execlp failed");
    return 1;
}
```

- **Explanation:** Here, `execlp` searches for the `ls` command in the system's `PATH`, so you don't need to specify the full path to the executable.

3. `execv()`

- **Use:** Takes an array of arguments rather than listing them individually.
- **Example:**

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    char *args[] = {"ls", "-l", NULL};  // Array of arguments
    execv("/bin/ls", args);

    // If execv fails
    perror("execv failed");
    return 1;
}
```

- **Explanation:** Here, arguments are provided as an array, which is useful if the arguments are determined at runtime.

4. `execvp()`

- **Use:** Works like `execv()` but searches for the file in the directories listed in the `PATH`.
- **Example:**

```
#include <stdio.h>
#include <unistd.h>

int main() {
    char *args[] = {"ls", "-l", NULL};  // Array of arguments
    execvp("ls", args);

    // If execvp fails
    perror("execvp failed");
    return 1;
}
```

- **Explanation:** Similar to `execv()`, but `execvp()` searches for the command in the `PATH` directories.

5. `execle()`

- **Use:** Allows you to specify the environment variables explicitly.
- **Example:**

```
#include <stdio.h>
#include <unistd.h>

int main() {
    char *envp[] = {"MY_VAR=Hello", NULL};  // Custom environment
    execle("/bin/ls", "ls", "-l", NULL, envp);

    // If execle fails
    perror("execle failed");
    return 1;
}
```

- **Explanation:** Similar to `execl()`, but the last argument is an array of environment variables (`envp`), which is passed to the new process.

6. `execve()`

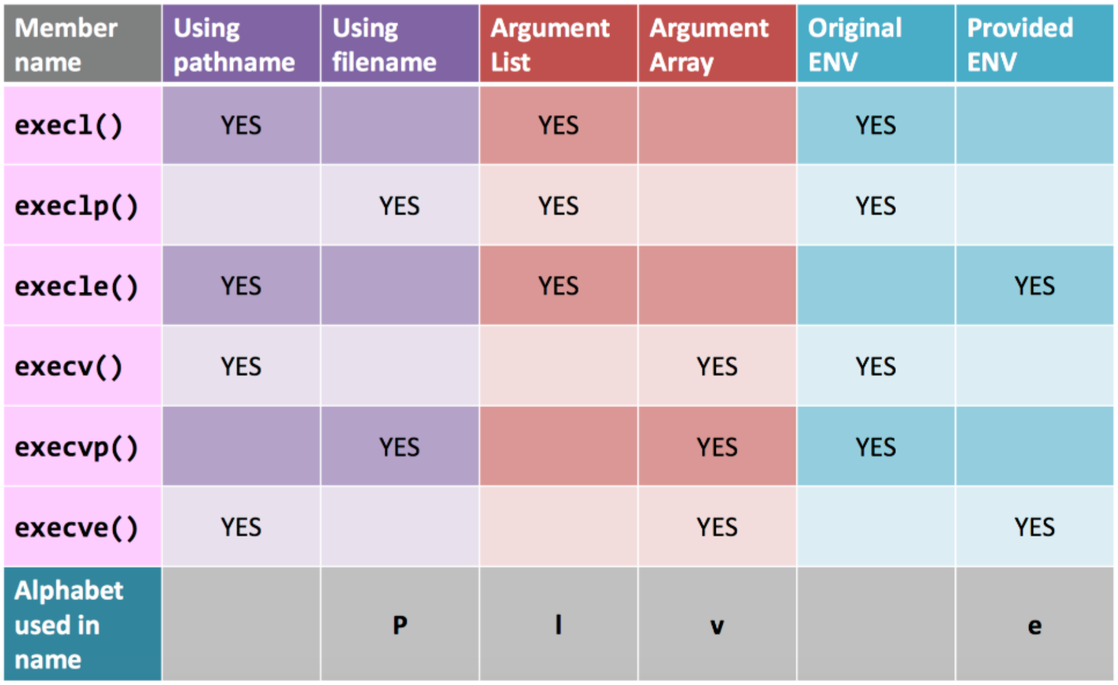
- **Use:** Takes both an array of arguments and an array of environment variables.
- **Example:**

```
#include <stdio.h>
#include <unistd.h>
```

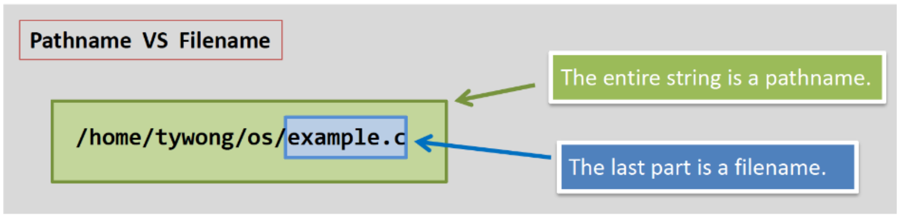
```
int main() {
    char *args[] = {"ls", "-l", NULL};       // Array of arguments
    char *envp[] = {"MY_VAR=Hello", NULL};  // Custom environment
    execve("/bin/ls", args, envp);

    // If execve fails
    perror("execve failed");
    return 1;
}
```

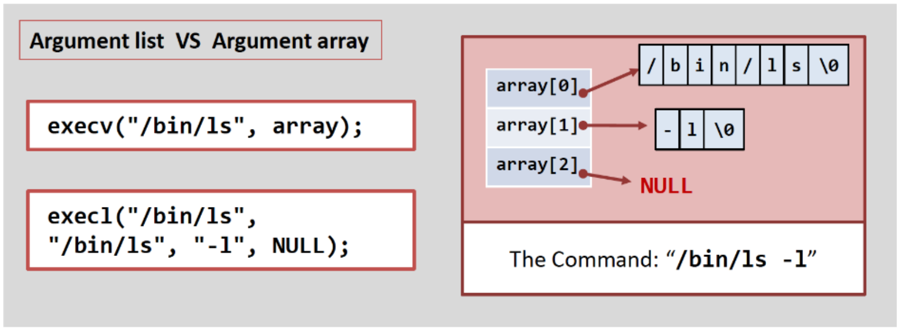
- **Explanation:** This is the most general form, allowing you to pass both an array of arguments and an array of environment variables to the new process.

Summary of Differences



Member name	Using pathname	Using filename	Argument List	Argument Array	Original ENV	Provided ENV
`execl()`	YES		YES		YES	
`execlp()`		YES	YES		YES	
`execle()`	YES		YES			YES
`execv()`	YES			YES	YES	
`execvp()`		YES		YES	YES	
`execve()`	YES			YES		YES
Alphabet used in name		**p**	**l**	**v**		**e**


Pathname VS Filename
/home/tywong/os/example.c
The entire string is a pathname.
The last part is a filename.


Argument list VS Argument array
execv("/bin/ls", array);
execl("/bin/ls", "/bin/ls", "-l", NULL);
array[0]
array[1]
array[2]
/ b i n / l s \0
- l \0
NULL
The Command: "/bin/ls -l"

`sleep()` System Call:

`sleep()` is a system call that can makes the process to *sleep* for a specified period.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
    printf("Before fork...\n");
    if(fork() == 0)
    {
        printf("Hello World!\n");
        exit(0);
    }
    sleep(1);
    printf("After fork...\n");
    return 0;
}
```

By using `sleep()`, parent can put to a *suspended* state and wait for the child. However, `sleep()` is not desirable (we need to specify the time...).

`getpid()` and `getppid()`

In C, you can use the `getpid()` and `getppid()` functions from the `unistd.h` library to get the process ID (PID) and the parent process ID (PPID) of the current process. Here's how to use them:

1. `getpid()`

- **Purpose:** Returns the process ID (PID) of the calling process.

2. `getppid()`

- **Purpose:** Returns the parent process ID (PPID) of the calling process.

Example Code:

```
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid = getpid();    // Get the current process ID
    pid_t ppid = getppid();  // Get the parent process ID

    printf("Current Process ID (PID): %d\\n", pid);
    printf("Parent Process ID (PPID): %d\\n", ppid);

    return 0;
}
```

Explanation:

- `getpid()`: Retrieves the PID of the running process.
- `getppid()`: Retrieves the PID of the parent process of the running process.