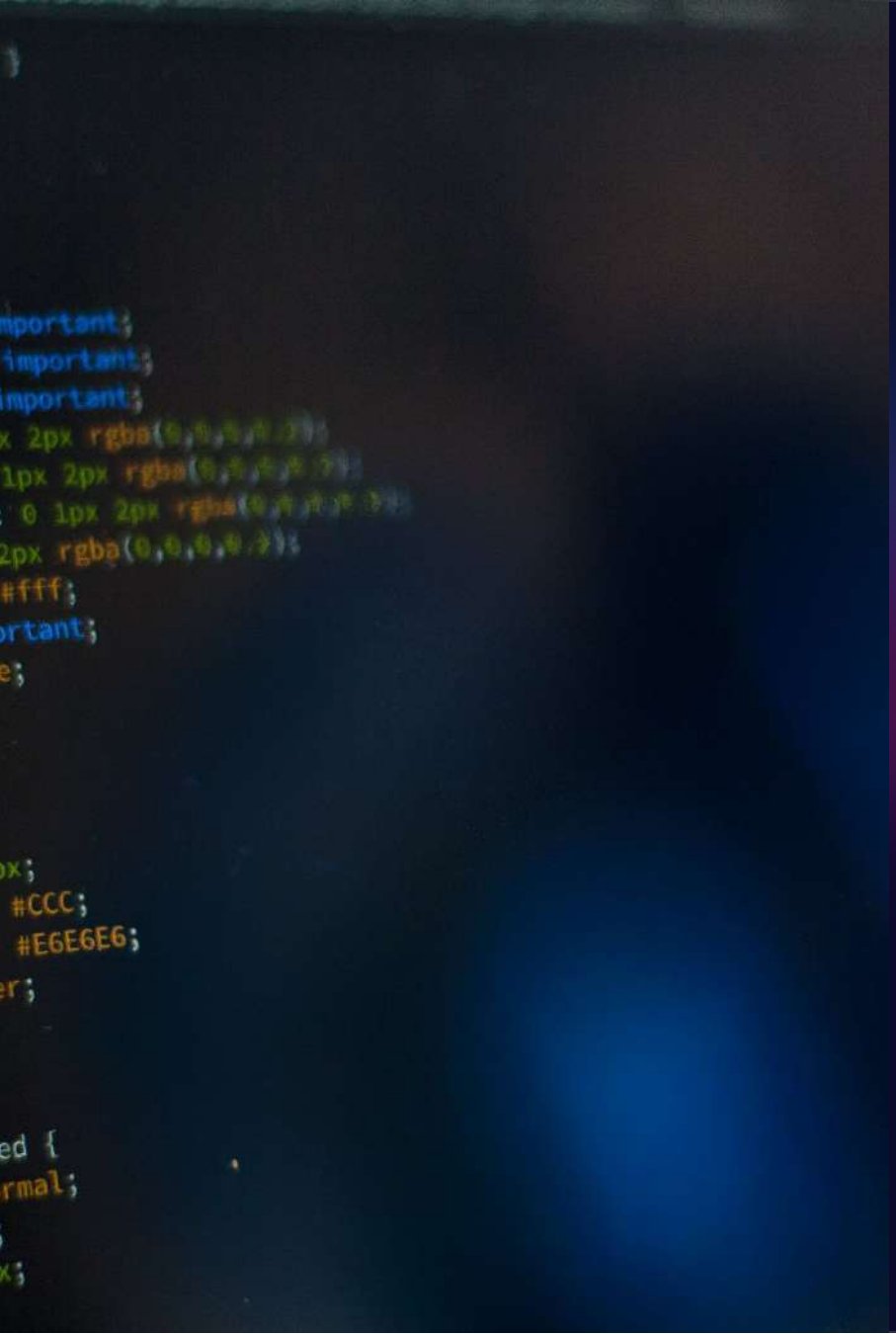


WEEK 7 and 8

In this C program, we explore the concept of creating child processes using the fork system call. By leveraging this mechanism, we can spawn a new process from an existing process, enabling parallel execution. Let's dive into the code and understand how it works.

by sahitya

Creating a Child Process: Step by Step

1

Fork System Call

The fork system call is used to create a new child process from the current parent process. It duplicates the parent process, including the code, data, and other resources.

2

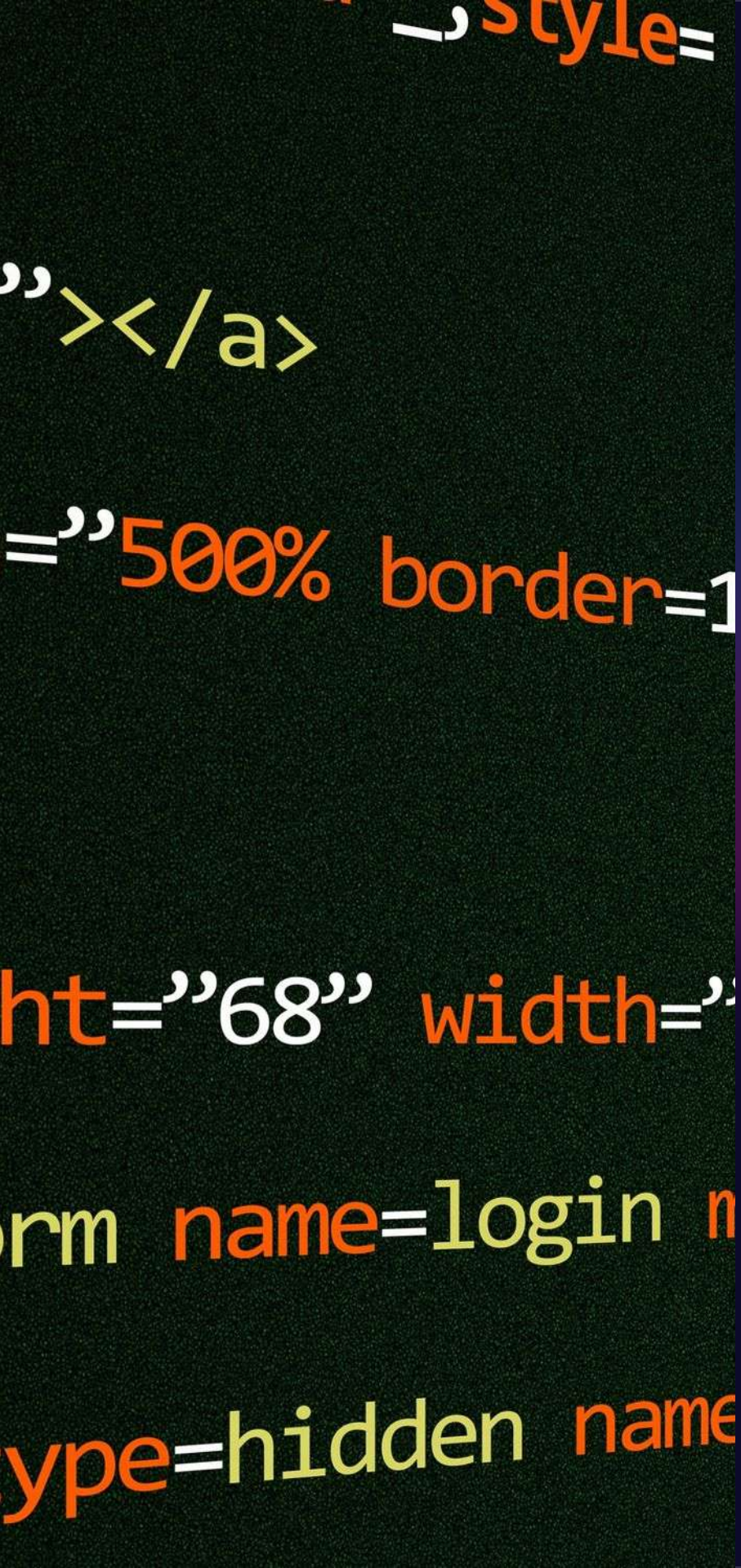
Communication

The child process and the parent process can communicate with each other through various mechanisms, such as pipes, shared memory, or message queues.

3

Executing Child Process

Once the fork call is made, both the parent process and the child process continue their execution from that point. However, the fork call returns different values for the parent and the child.



Understanding Fork System Call: Code Example

Let's take a look at a sample C program that demonstrates the creation of a child process using the fork system call:

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

int main(void) {
    int pid;
    int status;
    printf("Hello World!\n");
    pid = fork();
    if (pid == -1) {
        perror("bad fork");
        exit(1);
    }
    if (pid == 0)
        printf("I am the child process.\n");
    else {
        wait(&status);
        printf("I am the parent process.\n");
    }
}
```

Output of the Fork Program

When we run the program, we get the following output:

```
Hello World!
I am the child process.
I am the parent process.
```

This output showcases how both the child process and the parent process continue executing from the point of the fork, resulting in the interleaved output. The wait call ensures that the parent process waits for the child process to finish before proceeding.

Create a Zombie Process in C

A zombie process is a fascinating concept in the world of operating systems. It refers to a process that has terminated but still has an entry in the process table, as its exit status has not been collected by its parent. Let's explore a C program that intentionally creates a zombie process.

Creating a Zombie Process: Step by Step

1

Fork and Sleep

In the parent process, we fork a child process and make it sleep for a specified period of time using the sleep function. Meanwhile, the parent process waits.

2

Child Process Exits

The child process immediately exits without performing any further computation. At this point, it becomes a zombie process.

3

Parent Process Exits

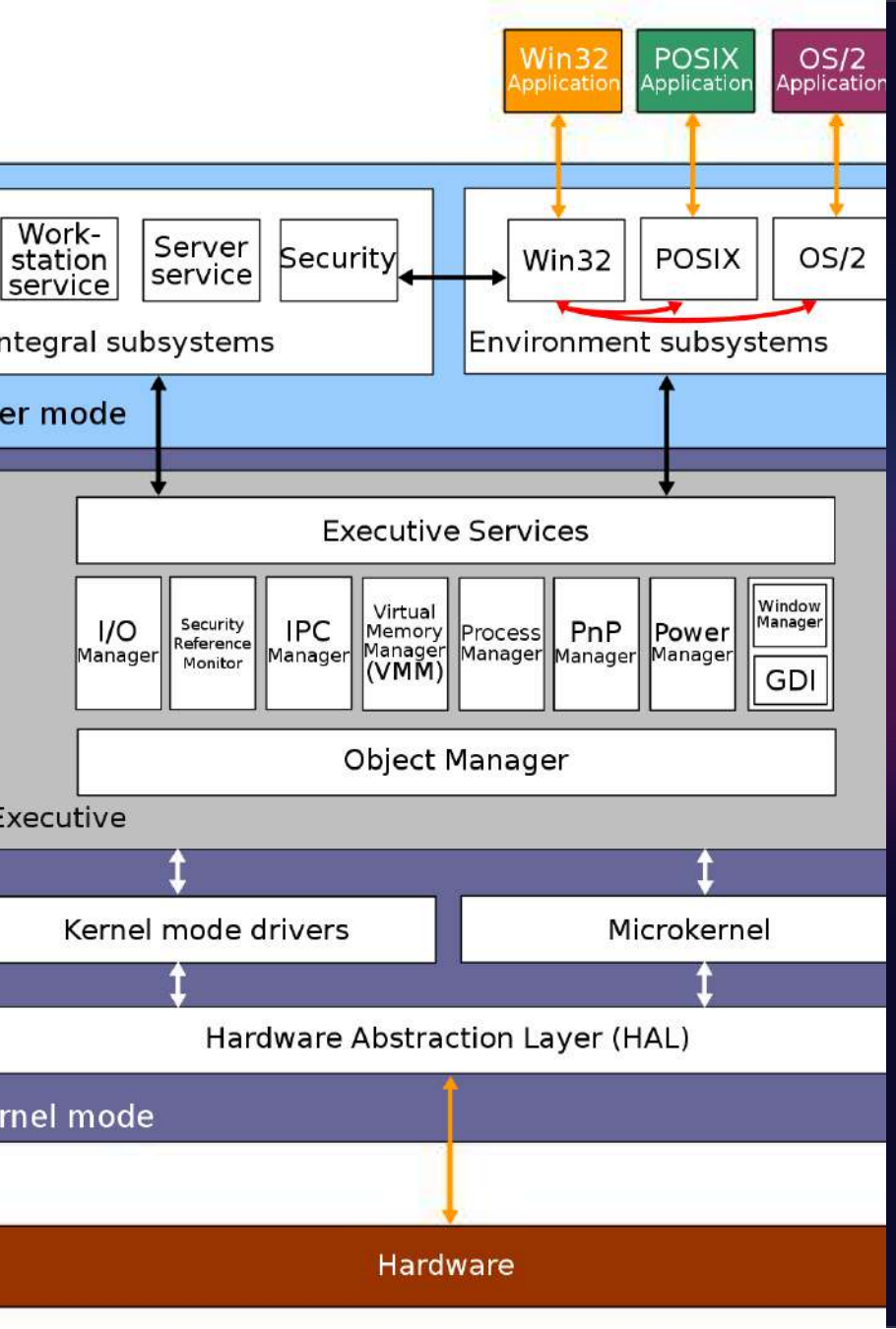
After the specified sleep time, the parent process wakes up and exits. The operating system automatically reaps the zombie child process.

Creating a Zombie Process: Code Example

Here's a C program that intentionally creates a zombie process:

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

int main() {
    pid_t child_pid;
    child_pid = fork();
    if (child_pid > 0) {
        sleep(60);
    }
    else {
        exit(0);
    }
    return 0;
}
```



Reaping a Zombie Process

During the execution of this program, the parent process sleeps for approximately 60 seconds, allowing the child process to become a zombie. Once the parent process exits, the operating system automatically reaps the zombie process.

Orphan Processes: When Kids Outlive Their Parents

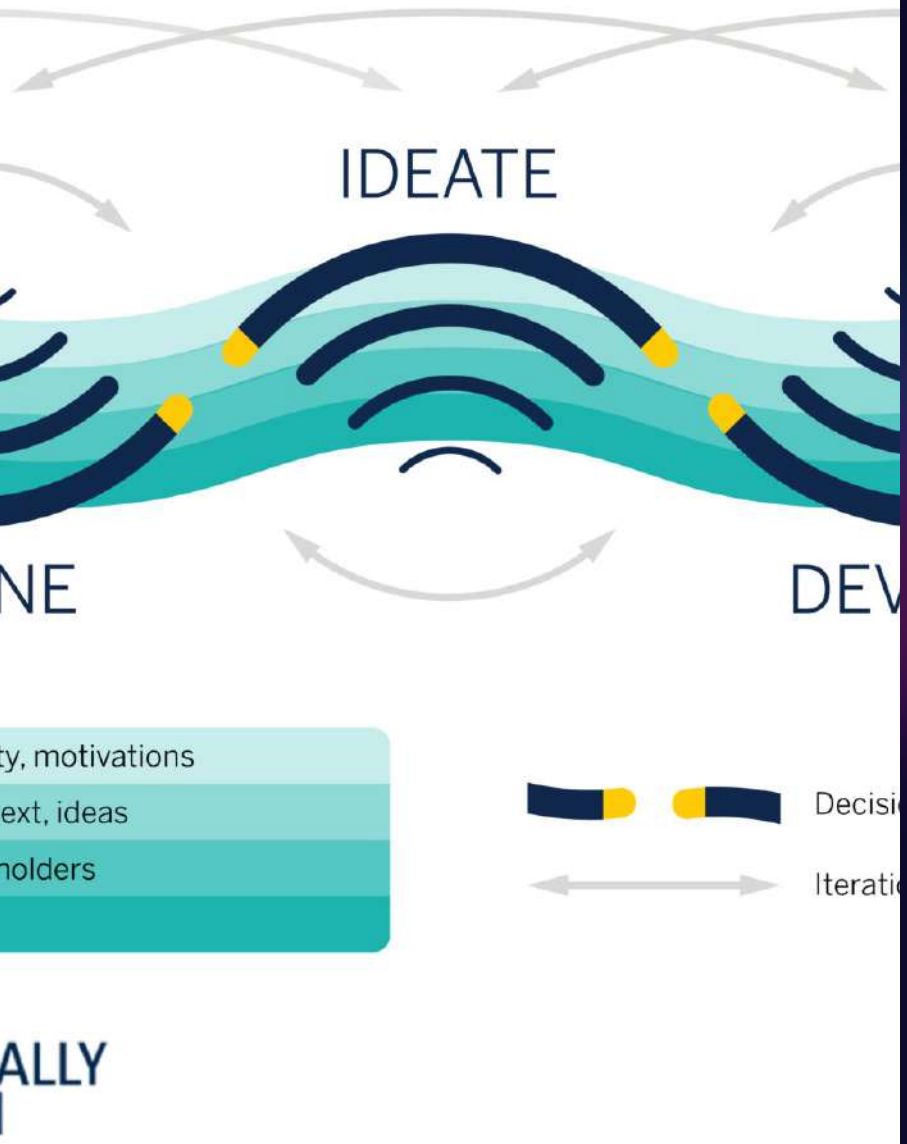
In the realm of process management, an orphan process is a process that continues to execute even after its parent process has terminated. This intriguing phenomenon occurs due to the existence of process groups and the adoption of orphaned processes by the init process.

Illustrating Orphan Process Creation: Code Example

Let's explore a C program that demonstrates the creation of an orphan process:

```
#include <stdio.h>

int main() {
    int pid;
    printf("I am the original process with PID %d and PPID %d.\n", getpid(), getppid());
    pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(1);
    }
    if (pid == 0) {
        printf("I am the child process with PID %d and PPID %d.\n", getpid(), getppid());
        sleep(10);
        printf("I am the child process. My parent process terminated.\n");
    }
    else {
        printf("I am the parent process with PID %d.\n", getpid());
        sleep(5);
        printf("I am the parent process. Exiting now.\n");
    }
    return 0;
}
```



An Orphan Process in Action

When we execute the above program, we observe the following behavior:

```
I am the original process with PID 1234 and PPID 5678.
I am the child process with PID 2345 and PPID 1234.
I am the parent process with PID 1234.
I am the parent process. Exiting now.
I am the child process. My parent process terminated.
```

As seen in the output, the child process continues execution even after the parent process has terminated. This showcases the existence of orphan processes and their behavior in the context of process management.