Chapter 5: Process API

This chapter delves into the practical aspects of process management in UNIX-like systems. It focuses on the core system calls that allow programs to create and control other processes, forming the foundation of how shells and many other system utilities work.

> **The Crux of the Problem: How to Create and Control Processes?**
>
> What interfaces should the OS present for process creation and control? How should these interfaces be designed to enable powerful functionality, ease of use, and high performance?

The UNIX solution is a powerful and elegant combination of three key system calls: `fork()`, `exec()`, and `wait()`.

1. The `fork()` System Call

The `fork()` system call is the primary method for creating a new process in UNIX. Its behavior is unique and fundamental to the system's design.

- **Action:** When a process (the **parent**) calls `fork()`, the OS creates a new, nearly identical process (the **child**).
- **State Duplication:** The child process receives a copy of the parent's address space (memory), CPU registers, and open file descriptors. It is a clone of the parent at the moment `fork()` was called.
- **Execution Point:** The child does **not** start execution at `main()`. Instead, it begins as if it also just returned from the `fork()` call.
- **The Key Difference (Return Value):** The only way to distinguish the parent from the child is by the return value of `fork()`:
  - In the **parent** process, `fork()` returns the **Process ID (PID)** of the newly created child.
  - In the **child** process, `fork()` returns **0**.
  - If `fork()` fails, it returns **-1**.

This differentiation allows a single piece of source code to execute different logic for the parent and child.

- **Example Program (`p1.c`):**

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

int main(int argc, char *argv[]) {
    printf("hello (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) { // fork failed
```

```
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("child (pid:%d)\n", (int) getpid());
    } else { // parent goes down this path
        printf("parent of %d (pid:%d)\n", rc, (int) getpid());
    }
    return 0;
}
```

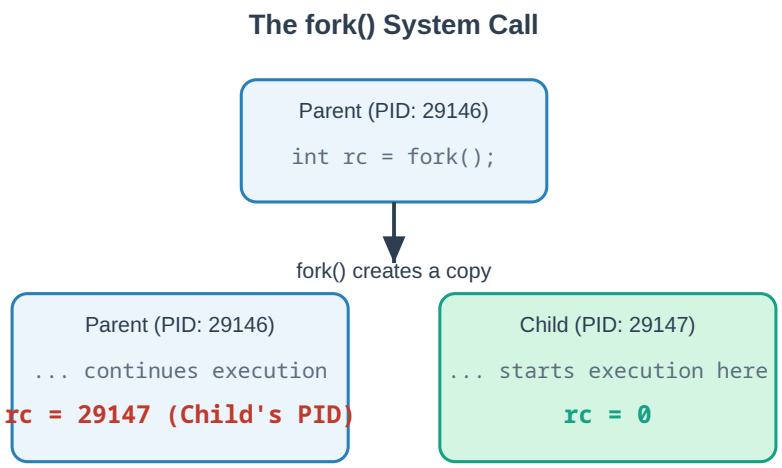

Figure 1: The fork() System Call

- **Non-Determinism:** After a `fork()`, both the parent and child are in the `Ready` state. The OS scheduler decides which one runs next. Because of this, the output order of the parent and child print statements is not guaranteed.

2. The `wait()` System Call

Often, a parent process needs to wait for its child to complete its task before continuing. The `wait()` system call facilitates this.

- **Action:** When a parent calls `wait()`, its execution is suspended until one of its child processes terminates.
- **Return Value:** `wait()` returns the PID of the terminated child.
- **Synchronization:** Using `wait()` allows a parent to synchronize its execution with its child, ensuring the child finishes first. This makes the execution order deterministic.

- **Example Program (`p2.c`):**

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

int main(int argc, char *argv[]) {
    printf("hello (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) { /* ... */ }
    else if (rc == 0) { // child
        printf("child (pid:%d)\n", (int) getpid());
    } else { // parent
        int rc_wait = wait(NULL); // Parent waits here
        printf("parent of %d (rc_wait:%d) (pid:%d)\n",
               rc, rc_wait, (int) getpid());
    }
    return 0;
}
```

- **Result:** With `wait()`, the parent will always print its message *after* the child has finished and printed its message.

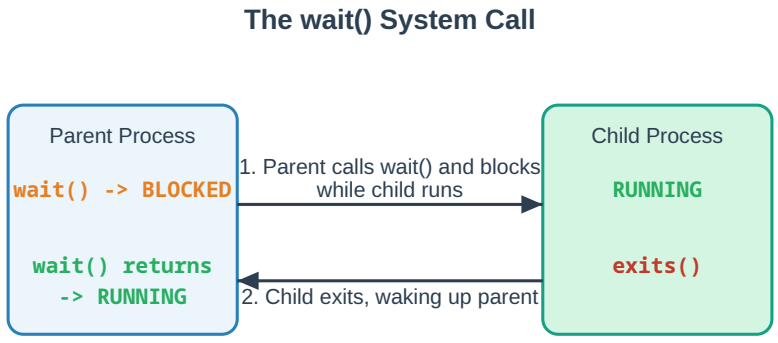

Figure 2: The wait() System Call

3. The `exec()` System Call

While `fork()` creates a copy of the current program, `exec()` transforms the calling process into a *new* program.

- **Action:** The `exec()` family of calls loads a new program from an executable file and replaces the current process's memory space (code, static data, stack, heap) with the new program's content.
- **No New Process:** `exec()` does **not** create a new process; it transforms the existing one.
- **No Return:** A successful `exec()` call **never returns** to the calling code, because that code has been overwritten.
- **Variants:** There are many variants (e.g., `execl()`, `execv()`, `execvp()`) that differ in how they accept arguments (as a list or an array) and whether they search the system's `PATH` for the executable.
- **Example Program (`p3.c`):** This program uses `fork()` to create a child, and the child then uses `execvp()` to run the `wc` (word count) command.

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

int main(int argc, char *argv[]) {
    printf("hello (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) { /* ... */ }
    else if (rc == 0) { // child
        printf("child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");       // program to run
        myargs[1] = strdup("p3.c");     // argument to wc
        myargs[2] = NULL;               // mark end of array
        execvp(myargs[0], myargs);      // run wc
        printf("this shouldn't print out");
    } else { // parent
        int rc_wait = wait(NULL);
        // ...
    }
    return 0;
}
```

4. The Power of `fork()` and `exec()` Separation

The separation of process creation (`fork()`) from program execution (`exec()`) is a cornerstone of UNIX design. It allows a shell (or any parent process) to manipulate the environment of a child process *after* the `fork()` but *before* the `exec()`.

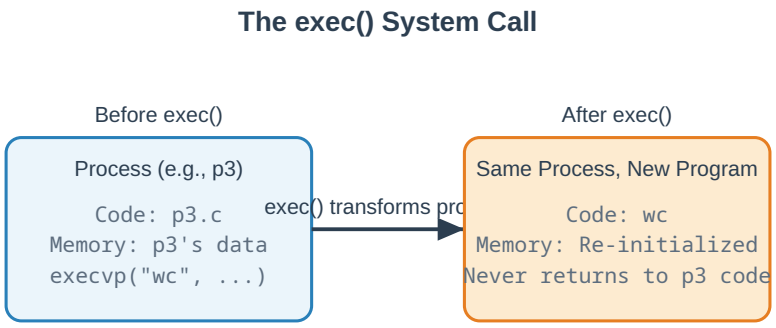

Figure 3: The exec() System Call

This enables powerful features like:

- **I/O Redirection:** A shell can redirect a child's output to a file.
  - **Mechanism:**
    1. Parent shell calls `fork()`.
    2. **In the child process:**
       a. Close the standard output file descriptor (`STDOUT_FILENO`).
       b. Open the desired output file (e.g., `newfile.txt`). The OS will assign the lowest available file descriptor, which is now `STDOUT_FILENO`.
       c. Call `exec()` to run the new program (e.g., `wc`).
    3. The `wc` program, unaware of the change, writes to its standard output, which now transparently points to `newfile.txt`.
- **Pipes (`|`):** The output of one command is connected to the input of another.
  - **Mechanism:** The shell uses the `pipe()` system call to create an in-kernel queue. It then `fork()`s two children and manipulates their file descriptors to connect one child's `stdout` to the pipe's write-end and the other child's `stdin` to the pipe's read-end before calling `exec()` on both.

5. Process Control and Users

- **Signals:** UNIX provides a signal mechanism for process communication. The `kill()` system call can send signals to processes to interrupt, terminate, pause (`SIGTSTP`), or continue them. Keyboard shortcuts like `Control-C` (sends `SIGINT`) and `Control-Z` (sends `SIGTSTP`) are common ways to send signals.

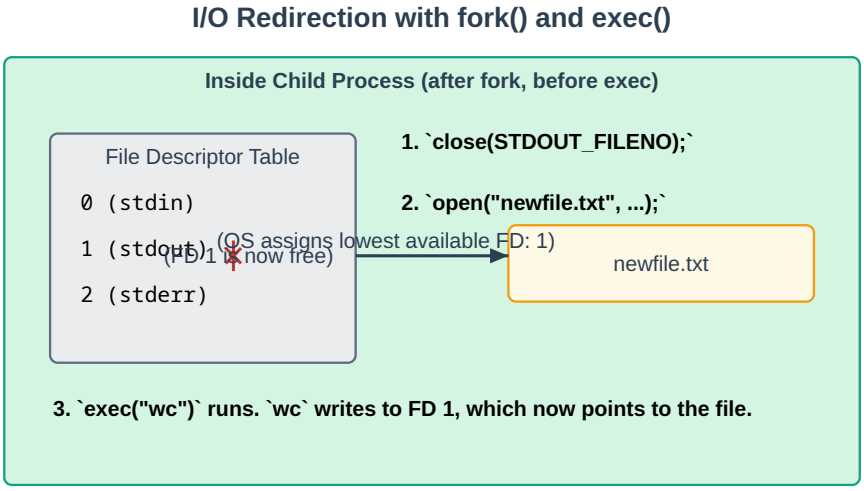

Figure 4: I/O Redirection with fork() and exec()

- **Users:** To maintain security and order, the OS has a concept of a **user**. A user can only control their own processes.
- **Superuser (root):** A special administrative user who has permission to control all processes and perform any system action. This power should be used with caution.

6. Useful Tools

- `ps`: Lists currently running processes.
- `top`: Displays system processes and their resource usage in real-time.
- `kill`, `killall`: Send signals to processes by PID or name.