Understand of Process and Process API

Sanghyeok Park
School of Electrical and Electronics Engineering
electronics@cau.ac.kr
20234690

September 22, 2025

1 Introduction

The operating system acts as an interface between hardware and the user. It manages resources to ensure that programs run efficiently and safely, and provides abstractions to simplify complex hardware operations.

1.1 CPU Virtualization

Physically, a CPU can only execute one instruction at a time. However, users perceive that multiple programs are running simultaneously. This illusion is created because the operating system uses CPU Virtualization, dividing CPU time among processes. The OS achieves concurrency through time sharing and scheduling.

1.2 API

There are two general ways for a user program to control hardware: directly or indirectly. If hardware were controlled directly, the OS could lose control of the system whenever a program failed to yield resources. Therefore, the indirect method is preferred. However, user programs sometimes need direct access to hardware. To safely provide this functionality, the OS exposes an Application Programming Interface (API) to interact with hardware.

1.3 System Call

User programs can only access resources through system calls provided by the OS. System calls are a type of API that provide controlled access to critical functions, such as reading/writing files, creating/terminating processes, and memory allocation. This mechanism allows the OS to maintain security and stability while enabling applications to use hardware functionality.

2 Process

A process is one of the most important abstractions provided by the OS, and is generally defined as a running program. A program is just a collection of instructions and static data on disk, but when the OS loads it into memory and executes it, it becomes an active process.

Users want to run multiple applications simultaneously. Even though modern systems usually have only a few CPUs, the OS uses CPU Virtualization to create the illusion that dozens or hundreds of processes can run at the same time. By applying time sharing, the OS executes one process briefly, suspends it, and switches to another, repeating this cycle to provide the illusion of concurrency.

2.1 Concept of Process

A process is essential to CPU Virtualization. It consists of the following elements:

- **Memory space:** code, static data, heap, stack
- **Registers:** PC, stack pointer.
- **Open file list:** including standard input, output, and error streams

A process can be in one of three states:

- **Running:** currently executing on the CPU
- **Ready:** prepared to run, but waiting for CPU allocation
- **Blocked:** waiting on an event such as I/O, unable to execute

2.2 Process Control Block (PCB)

The OS tracks each process via a Process Control Block (PCB), which includes:

- PID and process state (Running/Ready/Blocked)
- Saved register context (for context switching)
- Memory management info (code/data/heap/stack regions)
- Open file descriptors and current working directory
- Parent/child relations and pending signals

2.3 Process API

Modern operating systems provide the following APIs to control processes:

- **Create:** create and start a new process
- **Destroy:** terminate a running process
- **Wait:** wait until a specific process finishes
- **Miscellaneous Control:** suspend, resume, or otherwise control processes
- **Status:** retrieve execution time, state, or other process information

2.4 Process Creation

The transition from program to process consists of several steps:

1. Load the program code and static data into memory.
2. Initialize the runtime stack, including `argc` and `argv[]`.
3. Create a heap region for dynamic memory allocation.
4. Initialize file descriptors for standard input, output, and error.
5. Begin execution at the program entry point, `main()`.

3 Process API

The operating system provides system calls to create, execute, and terminate processes. `fork()`, `wait()`, and `exec()` are the core process-control primitives.

- `fork()`: creates a new process so that parent and child can run concurrently (with copy-on-write address space).
- `wait()`: blocks the parent until the child finishes, allowing the parent to reap the child's exit status (avoiding zombies).
- `exec()`: replaces the current process's memory image with a new program (PID is preserved).

By combining these three APIs, the OS can launch a new program when a user issues a command, while ensuring the parent process is synchronized with the completion of the child.

3.1 `fork()`

`fork()` creates a child process. The child has its own register state and program counter, and initially shares the address space via copy-on-write. At the point of the `fork()` call, both parent and child return from `fork()` to their respective control flows. Return values:

- In the parent: returns the child's PID
- In the child: returns 0
- On error: returns -1

Example code from [1]:

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

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int)getpid());
    int rc = fork();
    if (rc < 0)
    {
        fprintf(stderr, "fork failed\n");
        exit(1);
    }
    else if (rc == 0)
    {
        printf("hello, I am child (pid:%d)\n", (int)getpid());
    }
    else
    {
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int)getpid());
    }
    return 0;
}
```

3.2 wait()

`wait()` blocks the parent until the child terminates and reaps its exit status, preventing zombie processes. This ensures the child terminates first, allowing the parent to collect its exit status.

Example code from [1]:

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

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int)getpid());
    int rc = fork();
    if (rc < 0)
    {
        fprintf(stderr, "fork failed\n");
        exit(1);
    }
    else if (rc == 0)
    {
        printf("hello, I am child (pid:%d)\n", (int)getpid());
    }
    else
    {
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
               rc, wc, (int)getpid());
    }
    return 0;
}
```

3.3 exec()

Unlike `fork()`, `exec()` does not create a new process; it replaces the current process's memory image with a new program. The PID stays the same, but code, data, heap, and stack are replaced. `exec()` takes two parameters: (i) the pathname of the program to execute and (ii) an `argv` array of arguments. By convention, `argv[0]` is the program name and the array is `NULL`-terminated.

`exec()` takes two arguments:

- the name of the file to execute
- `argv[]`: the argument array passed to the program

Example code from [1]:

```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 world (pid:%d)\n", (int)getpid());
    int rc = fork();
    if (rc < 0)
    {
        fprintf(stderr, "fork failed\n");
        exit(1);
    }
    else if (rc == 0)
    {
        printf("hello, I am child (pid:%d)\n", (int)getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");
        myargs[1] = strdup("p3.c");
        myargs[2] = NULL;
        execvp(myargs[0], myargs);
        printf("this shouldn't print out");
    }
    else
    {
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
               rc, wc, (int)getpid());
    }
    return 0;
}
```

When `execvp()` is called, the child replaces its memory space with the program `wc p3.c`. If successful, the subsequent line `printf("this shouldn't print out")` is never executed. Meanwhile, the parent waits for the child to finish and then prints both PIDs.

References

[1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.10 edition, November 2023.