Chapter 5: Process API

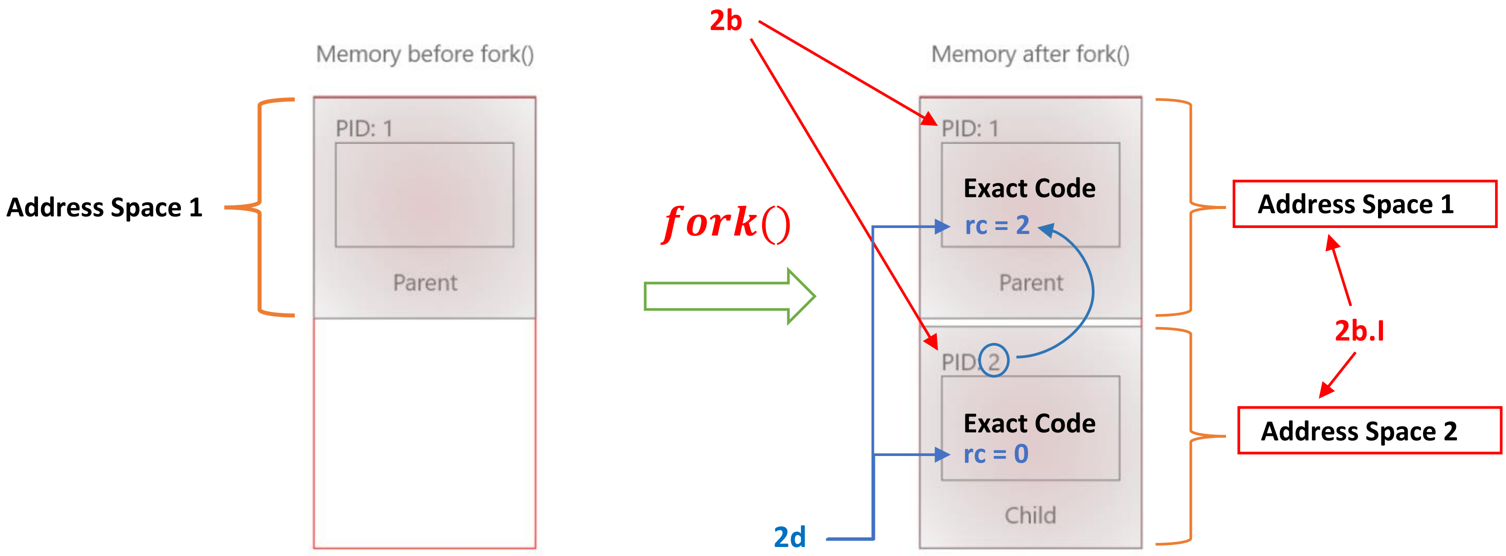
5.1 The $fork()$ System Call

$fork()$: a system call is used to create a new process.

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int
6  main(int argc, char *argv[])
7  {
8      printf("hello world (pid:%d)\n", (int) getpid());
9      int rc = fork();
10     if (rc < 0) {
11         // fork failed; exit
12         fprintf(stderr, "fork failed\n");
13         exit(1);
14     } else if (rc == 0) {
15         // child (new process)
16         printf("hello, I am child (pid:%d)\n", (int) getpid());
17     } else {
18         // parent goes down this path (original process)
19         printf("hello, I am parent of %d (pid:%d)\n",
20                rc, (int) getpid());
21     }
22     return 0;
23 }
```

1- When it first started running, the process prints out a hello world message. (Line: 8)
   a. Included in that message is its ***process identifier***, also known as a ***PID***.
   - **Process Identifier (PID):** a unique number that is used to name the process.
     - Used to do something with the process, such as, for example, stop it from running.

2- The process calls the $fork()$ system call, to create a new process. (Line: 9)
   - **Notes:**
     a. The process that is created is an ***(almost) exact copy*** of the calling process.
     b. Now there are two copies of the program p1 running, and both are about to return from the $fork()$ System call.
        I. **Notice:** the child isn't an exact copy. It now has its ***own copy of the address space (i.e., its own private memory)***, its own registers, its own PC, and so forth.
     c. The newly created process (called the child) doesn't start running at $main()$, rather, it just comes into life as if it had called $fork()$ itself.
     d. The value it returns to the caller of $fork()$ is ***different***.
        I. The parent receives the PID of the newly created child.
        II. The child receives a return code of zero.



You might also have noticed: **the output (of p1.c) is not deterministic**.

- When the child process is created, there are now ***two active processes*** in the system.
  - Assuming we are running on a system with a single CPU (for simplicity), then either the child or the parent might run at that point.

prompt> ./p1 hello world (pid:29146) hello, I am child (pid:29147) hello, I am parent of 29147 (pid:29146) prompt>	prompt> ./p1 hello world (pid:29146) hello, I am parent of 29147 (pid:29146) hello, I am child (pid:29147) prompt>

5.2 The *wait*() System Call

So far, we haven't done much: just created a child that prints out a message and exits.

Sometimes, it is quite useful for a parent to wait for a child process to finish what it has been doing.

- This task is accomplished with ***wait***() system call, or its more complete sibling ***waitpid***()

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

The parent process calls *wait*() to delay its execution until the child finishes executing.

When the child is done, *wait*() returns to the parent.

Now that you have thought a bit, here is the output:

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267) Always Child
hello, I am parent of 29267 (rc_wait:29267) (pid:29266)
prompt>
```

5.3 Finally, The $exec()$ System Call

$\boldsymbol{exec}()$: a system call that it ***transforms*** the <u>*currently*</u> running program into a <u>*different*</u> running program.

- It ***does not*** create a new process!

What it does? How $exec()$ works?

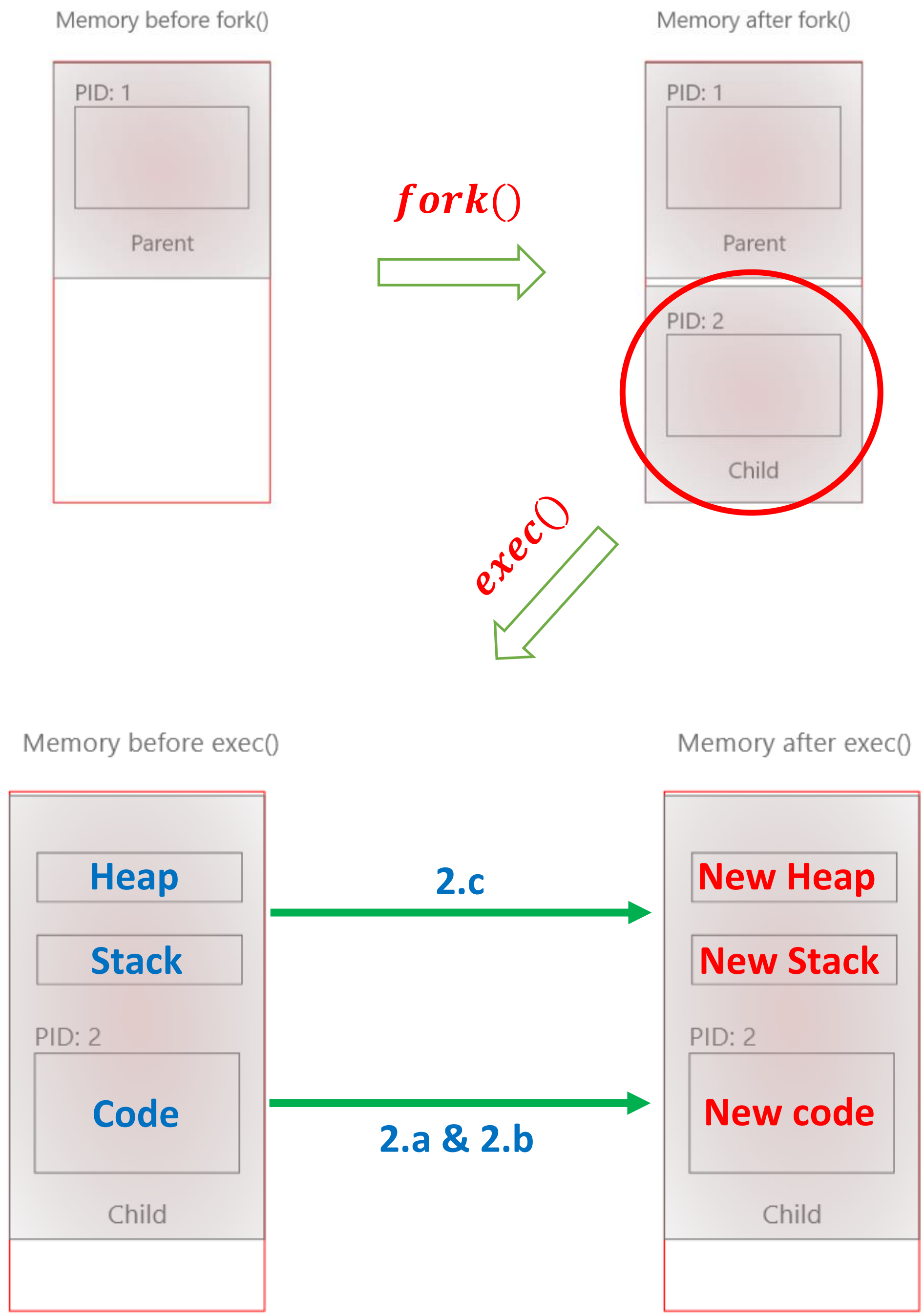
```c
int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("p3.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
        printf("this shouldn't print out");
    } else {
        // parent goes down this path (original process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
                rc, wc, (int) getpid());
    }
    return 0;
}
```

1- In this example, the child process calls $\boldsymbol{execvp}()$ in order to run the program wc.
- $\boldsymbol{wc}$: is the word counting program, which tells us how many ***lines***, ***words***, and ***bytes*** are found in the file.
- In this example, the child process runs $\boldsymbol{wc}$ on the source file $\boldsymbol{p3.c}$

2- Given the name of an executable (e.g., $\boldsymbol{wc}$), and some arguments (e.g., $\boldsymbol{p3.c}$)
   a. It **loads code** (and static data) from that executable ***(program binaries).***
   b. ***Overwrites*** its current code segment (and current static data) with the loaded code.
   c. The heap and stack and other parts of the memory space of the program are ***re-initialized***.

3- Then the OS simply ***runs*** that program, ***passing in any arguments as the*** <u>$argv$</u> of that process.

⇨ After the $\boldsymbol{exec}()$ in the child, it is almost <u>***as if p3.c never ran***.</u>
⇨ a successful call to $\boldsymbol{exec}()$ never returns.


Memory before fork()
PID: 1
Parent
fork()
Memory after fork()
PID: 1
Parent
PID: 2
Child
exec()
Memory before exec()
Heap
Stack
PID: 2
Code
Child
2.c
2.a & 2.b
Memory after exec()
New Heap
New Stack
PID: 2
New code
Child

5.4 Why? Motivating The API

Well, as it turns out, the separation of $\boldsymbol{fork}()$ and $\boldsymbol{exec}()$ is essential in building a UNIX shell, because it lets the shell run code after the call to $\boldsymbol{fork}()$ but before the call to $\boldsymbol{exec}()$.

⇨ The separation of $\boldsymbol{fork}()$ and $\boldsymbol{exec}()$ allows the shell to do a whole bunch of useful things rather easily.

Example 1:

The shell is just a user program. It shows you a prompt and then waits for you to type something into it. You then type a command (i.e., the name of an executable program, plus any arguments) into it.

Then the shell:

1- Figures out ***where*** in the file system ***the executable resides (your command).***
2- Calls $\boldsymbol{fork}()$ to create a new child process to run the command.
3- Calls some variant of $\boldsymbol{exec}()$ to run the command.
4- Waits for the command to complete by calling $\boldsymbol{wait}()$.
5- The shell returns from $\boldsymbol{wait}()$ and prints out a prompt again, ready for your next command.

Example 2:

$$\boldsymbol{prompt} \gg \boldsymbol{wc}\ \ \boldsymbol{p3.c} > \boldsymbol{newfile.txt}$$

In the example above, the output of the program $\boldsymbol{wc}$ is redirected into the output file $\boldsymbol{newfile.txt}$

- The greater-than sign is how said **redirection** is indicated.

The way the shell accomplishes this task is quite simple:

When the child is created ***(by fork)***, before calling $\boldsymbol{exec}()$:

1- The shell closes standard output.
2- Opens the file newfile.txt.

⇨ Any output from the ***soon-to-be-running program*** $\boldsymbol{wc}$ are sent to the file instead of the screen.

Figure 5.4 (page 8) shows a program that does exactly this.

```c
int
main(int argc, char *argv[])
{
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child: redirect standard output to a file
        close(STDOUT_FILENO);
        open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

        // now exec "wc"...
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("p4.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
    } else {
        // parent goes down this path (original process)
        int wc = wait(NULL);
        assert(wc >= 0);
    }
    return 0;
}
```

The reason this redirection works is due to an assumption about how the operating system manages ***file descriptors.***

1- Specifically, UNIX systems start looking for ***free file descriptors at zero***.
   - In this case, $\boldsymbol{STDOUT\_FILENO}$ will be the first available one and thus get assigned when $open()$ is called.
2- Subsequent writes by the child process to the standard output file descriptor, for example by routines such as $\boldsymbol{printf}()$, will then be routed transparently to the newly opened file ***instead of the screen***.

Here is the output of running the p4.c program:

```
prompt> ./p4
prompt> cat p4.output
      32      109      846 p4.c
prompt>
```

Notes:

1- When p4 is run, ***it looks as if nothing has happened***; you don't see any output printed to the screen because it has been redirected to the file $\boldsymbol{p4.output}$.

2- You can see that when we $\boldsymbol{cat}$ the output file, all the expected output from running $\boldsymbol{wc}$ is found.

UNIX pipes are implemented in a similar way, but with the $\boldsymbol{pipe}()$ system call.

- The ***output of one process*** is connected to an in-kernel pipe (i.e., queue)
- The ***input of another process*** is connected to that same pipe.

⇨ Thus, the output of one process seamlessly is used as input to the next.
  - long and useful chains of commands can be strung together.

Example:

$$\boldsymbol{grep - o\ foo\ file \mid wc\ \ -l}$$

Consider looking for a word in a file, and then counting how many times said word occurs; with pipes and the utilities $grep$ and wc, it is easy.

Topics to be covered later:

- The CPU ***scheduler***, a topic we'll discuss in great detail soon, determines which process runs at a given moment in time.
- This **non-determinism**, as it turns out, leads to some interesting problems, particularly in **multi-threaded programs**; hence, we'll see a lot more non-determinism when we study **concurrency** in the second part of the book.