~~~How does FORK() work?~~~

The **fork()** system call allows one process, the current running parent process to create a new child process. A duplicate is made of the parent process. The child obtains the exact copies of its parent's memory segment.

When the fork() system call is invoked, a new process is created with an identical memory layout to that of its parent.

Syntax of fork():

pid_t fork(void);

In parent, it returns the child's process ID is returned to the parent and 0 is returned to the child on success or returns -1 on error.

The execution continues from the point where the fork () is returned. But, it is important to remember that, upon the successful call of the fork function, two processes will be under execution at any given point in time. Note, once the fork is called, both the parent and child a race condition will start where whichever process gets the time slice, gets the CPU. We need to use synchronization techniques if we want to the fork in desired manner.

The code:

```c
#include<stdio.h>

void main()
{
    printf("Printing in Parent, only run once\n");
    printf("Process ID and parent process ID in parent: %d , %d\n",getpid(),getppid());
    fork();
    printf("Process ID and parent process ID in (runs in parent and child): %d , %d\n",getpid(),getppid());
    while(1);
}
```

Output:

```
Printing in Parent, only run once
Process ID and parent process ID in parent: 3401 , 3202
Process ID and parent process ID in (runs in parent and child): 3401 , 3202
Process ID and parent process ID in (runs in parent and child): 3402 , 3401
```

As you can see in the below image, the 3402 and 3401 are run by **a.out** concurrently. This process tree is obtained by running the command **ps axjf**.

```
   1  3125  3125  3125 tty8         0 Ss       0   0:00 /init
3125  3126  3126  3125 tty8         0 S     1000   0:00  \_ -bash
3126  3164  3164  3125 tty8         0 S        0   0:00      \_ su yash
3164  3165  3165  3125 tty8         0 S     1000   0:00          \_ bash
3165  3185  3185  3125 tty8         0 S        0   0:00              \_ su yash
3185  3186  3186  3125 tty8         0 S     1000   0:00                  \_ bash
3186  3201  3201  3125 tty8         0 S        0   0:00                      \_ su
3201  3202  3202  3125 tty8         0 S        0   0:00                          \_ bash
3202  3401  3401  3125 tty8         0 R        0   2:14                              \_ ./a.out
3401  3402  3401  3125 tty8         0 R        0   2:14                                  \_ ./a.out
```

In Parent

```
#include<stdio.h>

void main()
{
   printf("Printing in Parent, only run
once\n");
   printf("Process ID and parent process ID
in parent: %d , %d\n",getpid(),getppid());
   fork();
   printf("Process ID and parent process ID
in (runs in parent and child): %d ,
%d\n",getpid(),getppid());
   while(1);
}
```

In Child

```
#include<stdio.h>

void main()
{
   printf("Printing in Parent, only run
once\n");
   printf("Process ID and parent process ID
in parent: %d , %d\n",getpid(),getppid());
   fork();
   printf("Process ID and parent process ID
in (runs in parent and child): %d ,
%d\n",getpid(),getppid());
   while(1);
}
```

Let us understand how the fork works if called multiple time. In this section of code, we will call the fork line by line.

The code:

```
#include<stdio.h>
void main()
{
   printf(" Started from main, Calling forks\n");
   fork(); //fork1
   printf(" Fork 1 was called.\n");
   fork(); //fork2
   printf(" Fork 2 was called.\n");
   fork(); //fork3
   printf(" Fork 3 was called.\n");
   fork(); //fork4
   printf(" Fork 4 was called.\n");
   while(1);
}
```

Output:

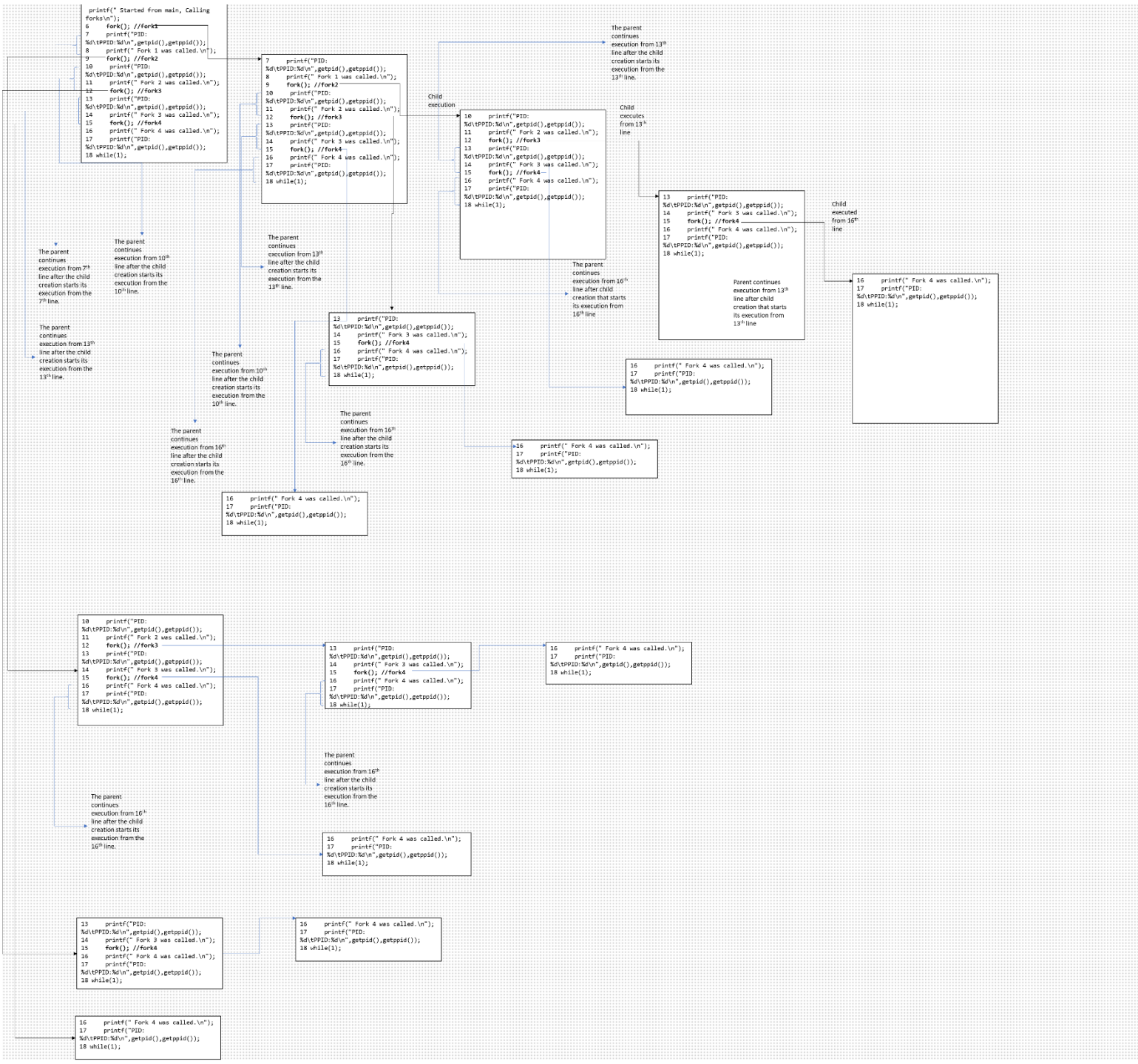
```
Started from main, Calling forks
Fork 1 was called.
Fork 1 was called.
Fork 2 was called.
Fork 2 was called.
Fork 2 was called.
Fork 2 was called.
Fork 3 was called.
Fork 3 was called.
Fork 3 was called.
Fork 3 was called.
Fork 3 was called.
Fork 3 was called.
Fork 3 was called.
Fork 3 was called.
Fork 4 was called.
Fork 4 was called.
Fork 4 was called.
Fork 4 was called.
Fork 4 was called.
Fork 4 was called.
Fork 4 was called.
Fork 4 was called.
Fork 4 was called.
Fork 4 was called.
Fork 4 was called.
Fork 4 was called.
Fork 4 was called.
Fork 4 was called.
Fork 4 was called.
Fork 4 was called.
Fork 4 was called.
```

It is indeed confusing. Let’s make a tree and see how it works.

```
   1  3125  3125  3125 tty8         0 Ss       0   0:00 /init
3125  3126  3126  3125 tty8         0 S     1000   0:00  \_ -bash
3126  3164  3164  3125 tty8         0 S        0   0:00      \_ su yash
3164  3165  3165  3125 tty8         0 S     1000   0:00          \_ bash
3165  3185  3185  3125 tty8         0 S        0   0:00              \_ su yash
3185  3186  3186  3125 tty8         0 S     1000   0:00                  \_ bash
3186  3201  3201  3125 tty8         0 S        0   0:00                      \_ su
3201  3202  3202  3125 tty8         0 S        0   0:00                          \_ bash
3202  3649  3649  3125 tty8         0 R        0   0:07                              \_ ./multi_fork
3649  3650  3649  3125 tty8         0 R        0   0:12                                  \_ ./multi_fork
3650  3652  3649  3125 tty8         0 R        0   0:15                                  |   \_ ./multi_fork
3652  3656  3649  3125 tty8         0 R        0   0:19                                  |   |   \_ ./multi_fork
3656  3658  3649  3125 tty8         0 R        0   0:14                                  |   |   |   \_ ./multi_fork
3652  3657  3649  3125 tty8         0 R        0   0:16                                  |   |   \_ ./multi_fork
3650  3655  3649  3125 tty8         0 R        0   0:06                                  |   \_ ./multi_fork
3655  3664  3649  3125 tty8         0 R        0   0:06                                  |   |   \_ ./multi_fork
3650  3663  3649  3125 tty8         0 R        0   0:08                                  |   \_ ./multi_fork
3649  3651  3649  3125 tty8         0 R        0   0:20                                  \_ ./multi_fork
3651  3654  3649  3125 tty8         0 R        0   0:07                                  |   \_ ./multi_fork
3654  3662  3649  3125 tty8         0 R        0   0:07                                  |   |   \_ ./multi_fork
3651  3661  3649  3125 tty8         0 R        0   0:07                                  |   \_ ./multi_fork
3649  3653  3649  3125 tty8         0 R        0   0:07                                  \_ ./multi_fork
3653  3660  3649  3125 tty8         0 R        0   0:07                                  |   \_ ./multi_fork
3649  3659  3649  3125 tty8         0 R        0   0:07                                  \_ ./multi_fork
```

The above tree was made by using the command **ps axjf.**

Upon running the program, we see that a total of 16 processes have been created. Check the below image for a detailed analysis of how the fork works for that particular code snippet.


printf(" Started from main, Calling forks\n");
fork(); //fork1
printf("PID: %d\tPPID:%d\n",getpid(),getppid());
printf(" Fork 1 was called.\n");
fork(); //fork2
printf("PID: %d\tPPID:%d\n",getpid(),getppid());
printf(" Fork 2 was called.\n");
fork(); //fork3
printf("PID: %d\tPPID:%d\n",getpid(),getppid());
printf(" Fork 3 was called.\n");
fork(); //fork4
printf(" Fork 4 was called.\n");
printf("PID: %d\tPPID:%d\n",getpid(),getppid());
while(1);
Child execution
The parent continues execution from 13th line after the child creation starts its execution from the 13th line.
Child executes from 13th line
Child executed from 16th line
Parent continues execution from 13th line after child creation that starts its execution from 13th line
The parent continues execution from 16th line after child creation that starts its execution from 16th line
The parent continues execution from 7th line after the child creation starts its execution from the 7th line.
The parent continues execution from 10th line after the child creation starts its execution from the 10th line.
The parent continues execution from 13th line after the child creation starts its execution from the 13th line.
The parent continues execution from 10th line after the child creation starts its execution from the 10th line.
The parent continues execution from 16th line after the child creation starts its execution from the 16th line.
The parent continues execution from 16th line after the child creation starts its execution from the 16th line.
The parent continues execution from 16th line after the child creation starts its execution from the 16th line.
The parent continues execution from 16th line after the child creation starts its execution from the 16th line.

Let us look at how the fork function call works when called iteratively.

```
void main()
{
	int i;

	for(i=1;i<=3;i++)
	{
		if(fork()==0)
		{
			printf("Loop %d - PID:%d\tPPID:%d\n",i,getpid(),getppid()  ) ;
		}

	}
	while(1);
}
```

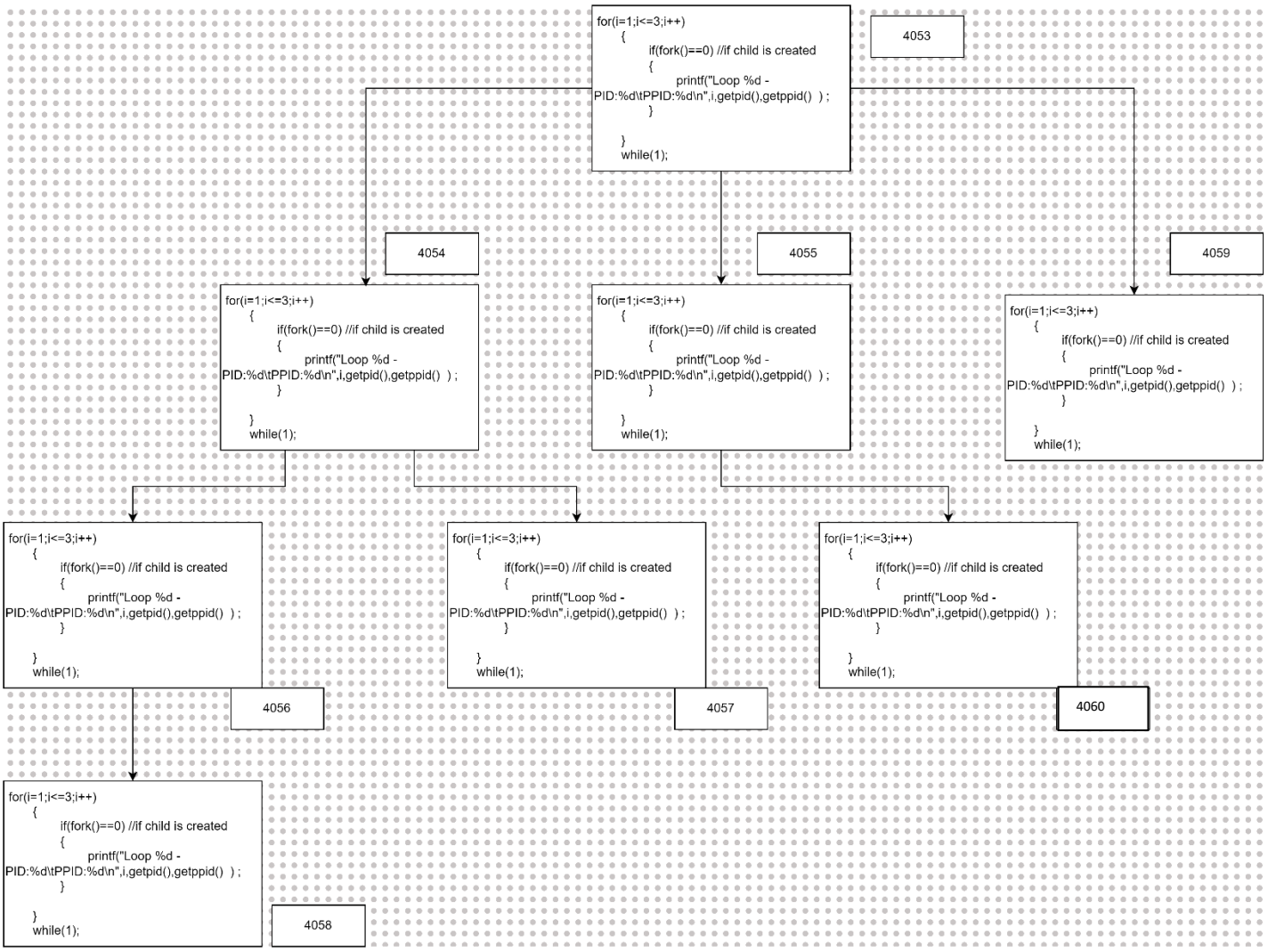
The output:

```
Loop 1 - PID:4054	PPID:4053
Loop 2 - PID:4056	PPID:4054
Loop 2 - PID:4055	PPID:4053
Loop 3 - PID:4060	PPID:4055
Loop 3 - PID:4057	PPID:4054
Loop 3 - PID:4058	PPID:4056
Loop 3 - PID:4059	PPID:4053
```

Using the PS command:

```
_ ./for_fork
   \_ ./for_fork
   |   \_ ./for_fork
   |   |   \_ ./for_fork
   |   \_ ./for_fork
   \_ ./for_fork
   |   \_ ./for_fork
   \_ ./for_fork
```

It can be visualized as the following diagram.



When the fork call is made, the kernel creates a text segment for the child process by setting up a set of per-process page table entries. These entries refer to the same virtual memory page frames that are already in use by the parent process. The text segment is designated as a read-only code segment.

In the case of data, heap, and stack segments, the kernel employs a copy-on-write mechanism. After the fork call, if either the parent or the child process attempts to modify one of these pages that are initially created by the kernel, the kernel intercepts the access and creates a duplicate copy of the page that can be modified. This involves assigning a new page and updating the corresponding page table entry for the child process.

At this point, the parent and child processes can independently modify their respective private copies of the page without affecting each other. The parent's process page is not copied for the child process; instead, it is shared between the parent and the child upon forking. When a process attempts to modify a shared page, a separate copy of the page is created exclusively for that process. This copy-on-write approach ensures that pages are only duplicated when a process actually writes to them, optimizing memory usage.

By utilizing the fork system call and implementing copy-on-write, the operating system efficiently manages memory resources and allows parent and child processes to safely and independently modify shared and private memory pages as needed.

~~~~~~

To receive updates regarding Embedded Systems, Space and Technology concepts, follow T Yashwanth Naidu.
Happy learning.

Follow #oswithyash to get updates on the concepts of embedded system os.
~~~~~~

Written by : T Yashwanth Naidu