Torry Harris

Torry Harris Business Solutions

Leaky Abstractions

Author – Manjunath Hanasi

www.thbs.com

One day, one of my friends came up to me with a piece of paper in his hand. He was wearing a winning smile on his face as if he had discovered water in the darkest corner of planet Mars.

He showed me the piece of paper; it had some C language code written on it. I could see lots of boxes, arrow marks, etc..indicating that he was working on demystifying some serious stuff.

```
/* buf.c */
#include <stdio.h>
#include <unistd.h>
int main()
{
        fprintf(stdout, "Hello World!");
        fork();
        return 0;
}
```

He asked me *'what would be the output of this program?'*

Before you see my answer, try to think on your own. You will be amazed by this little program.

Note: For those not from Unix background, 'fork' system call creates an exact replica of the process calling that function. The created process is normally referred to as 'child' process where as the process that invoked 'fork' is called 'parent' or 'mother' process. The execution in the child process, starts from the statement immediately after the 'fork' call.

In the above example, a process executes 'fprintf' statement, it creates the child process by 'fork' system call and returns. The Child process, however, will not start the execution from the beginning, it only starts after the 'fork' statement, i.e. child process will only execute 'return' statement in the above example. One more important fact about the 'fork' system call is that the 'child' process will inherit user/group id, environment, all the open file handles, address space and lots of other details.

As a side note: probably 'fork' is the only function which returns 2 return codes at the same time. 'Mother' process will get the process id of the 'child' process where as 'child' process gets a 0 as return value of 'fork'. Confusing??? Please read the main page for 'fork' system call. fork(2).

There were times when I considered myself blessed with C and Unix skills; people used to come to me whenever they found something fascinating in these areas. I replied without hesitation, *"Hello World!"*

"Only once?" was his next question.

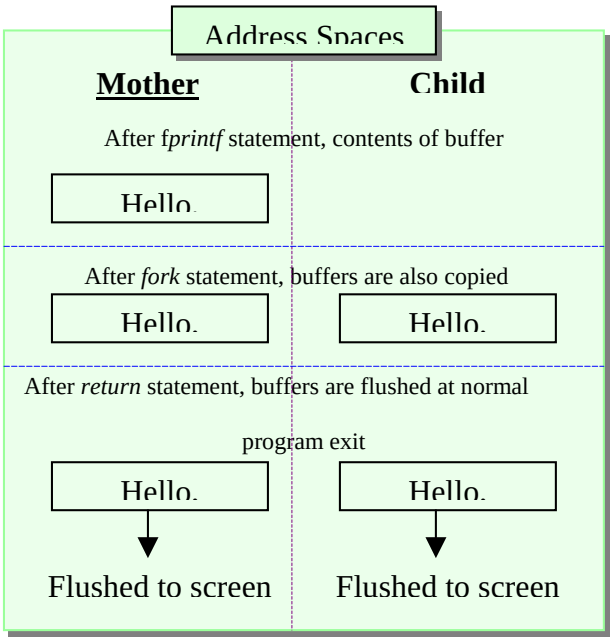
I said *"of course, I know 'fork' creates a child process but the execution in the child process will start from the statement following 'fork'. So child process would return without doing anything. Hence whatever the mother process printed, that would appear on the screen".*

My friend had a huge grin on his face, he briskly typed the above lines on my Linux PC, invoked the wonderful 'gcc' compiler. The program compiled without any errors and warnings. Then he executed the program; I was astonished to see the output. I couldn't believe my eyes; the output was showing 'Hello World!' not *once* but *twice*. I saw the program again and again to see whether we have made a mistake. No there was none!

```
$ gcc -o buf buf.c; ./buf
Hello World!Hello World!$
```

I quickly went back to my Unix collection and checked whether *'fork'* makes the child process start the execution from the start, it was *not* to be. This certainly defied my belief about sequential execution of statements following Von Neumann architecture [sequential architecture].

Seeing my frustration, my friend explained what was happening behind the scenes. Here it goes….



Its actually because of an abstraction provided by *fprintf* function in the standard C library. When we use *fprintf* to write something on the screen (stdout), it is immediately *not* sent to the console; it is stored in an area in memory referred to as 'buffer'. This buffer is specific to a process and is flushed when the buffer gets full or the program exits normally*.

When we *forked*, this buffer was also copied to the child address space and since we did not flush the buffer after *fprintf*, the contents were still there when *forked*. Now, we have the content in both the address spaces, mother as well as child. When these processes exited normally after *'return'* statement, the buffers were flushed to the screen. Since both the buffers had the content *'Hello World!'*, it was written twice.

* Read on to know more methods to flush the contents of buffer

"Wow !" I exclaimed, that's a wonderful explanation of what was happening inside without our knowledge.

But in our industry we should not believe in much of the theory, instead we should try to prove them. So I setout to prove this *theory*. We modified the program slightly, added a statement to flush the buffers immediately after the *fprintf* statement.

```
/* buf.c */
#include <stdio.h>
#include <unistd.h>
int main()
{
        fprintf(stdout, "Hello World!");
        fflush(stdout);
        fork();
        return 0;
}
$ gcc -o buf buf.c; ./buf
Hello World!$
```

This time the output showed only one *"Hello World!"*. The explanation was spot-on.

"That's wonderful. Now, lets go and play" I told him. But he was not done yet.

He modified the program again, removing *fflush* function call and adding a *newline* character after *"Hello World!*". Again he asked me *"What would be the output?"*

Now, this time I didn't want to face any embarassament, so I decided to think through deeply before giving an answer.

```
/* buf.c */
#include <stdio.h>
#include <unistd.h>
int main()

{
        fprintf(stdout, "Hello World!\n");
        fork();
        return 0;
}

$ gcc -o buf buf.c; ./buf

Hello World!
$
```

I tried to confirm the size of the *stdout* buffer, I found it to be fairly large (8192 on Linux [stdio.h]). So, this time boldly I told him *"It will print twice!"*.

I was eagerly waiting for him to execute the program and show me the output.

I was wrong again, it printed *only once* this time.

Not spending too much time, I shamelessly asked him *"Why is it so?"*, I had realized how little I knew about C and Unix.

He asked me to relax and executed the program again *without modifying the source code*; though this time redirecting the output to a file.

"What would be the output?" was his question.

I asked him to show me the output as I was not in a position to think and I had been wronged twice. He showed me the output, which was another shock to me that evening.

```
$ ./buf > out.log; cat out.log
Hello World!
Hello World!
```

The same program which prints *only once* on the screen, prints *twice* when the output is redirected to a file. I thought I would quit the software industry and open a dhaba on the roadside.

I didn't have any strength to think and solve the problem myself; so I asked my brilliant friend *"Whats happening? This is so weird"*. He explained for about half-an-hour on what was happening and here is the summary ….

The standard 'out' device (symbolically *stdout*) buffers the contents until the buffer *fills up* or a *newline* character is encountered or an explicit *flush* is done. It flushes the buffer when one of the above conditions is met. Such devices are said to be *line-buffered*. That is the reason when we put *'\n'* character in the *fprintf*, the output appeared only once on the screen; because '\n' character triggers the flushing of buffer. When *fork* copies the mother address space to child, the buffer is empty.

"Why did it print twice when the output was redirected to a file?" was my immediate question. He started answering as if he was already prepared for this question.

That's because, files are *not* line-buffered; they are *block-buffered*. This means, flushing to files would *not* occur as soon as a *'\n'* character is encountered. Block buffers are flushed only when the buffer fills up *or* an explicit flush is done on the buffer (file descriptor).

```
/* buf.c */
#include <stdio.h>
#include <unistd.h>
int main()
{
        fprintf(stderr, "Hello World!");
        fork();
        return 0;
}
```

I was enlightened. I thanked him for explaining such a wonderful thing to me. I asked him *"Shall we go out and have cool-drinks? I need to cool down"*. As we were starting off, he asked me *"You wanna try this?"*

Any ideas ??