Homework 19 - `Fork()`

Definition

The `fork-and-join` idea came up in the 1960s and later `fork()` became a system call that first appeared in Version 6 AT&T UNIX in the 1970s. It causes creation of a new process and is typically used in c or c++ programming languages. The original is the parent process, and the newly created one its child.

`fork()` receives no argument and returns an integer. On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created.

```
#include <iostream>
#include <unistd.h>

using namespace std;

int main() {
    cout << "current process" << " (pid: " << getpid() << ")" << endl;
    int rc = fork();
    if (rc < 0) {
        cout << "fork() failed";
        exit(1);
    } else if (rc == 0) {
        cout << "child process (pid: " << getpid() << ")" << endl;
    } else {
        cout << "parent process (pid: " << getpid() << ")" << endl;
    }
    return 0;
}
```

The code snippet above outputs:

```
current process (pid: 86612)
parent process (pid: 86612)
child process (pid: 86614)
```

Here it returns twice on success, because `fork()` returns `pid` of the created child and returns `0` in the created child.

What it does in the OS

Calling `fork()` creates a child process. Child is an exact copy of the parent process but the address space of child process differs from the parent. A technique called 'copy-on-write' or 'lazy copying' is used, so that the actual copying of stack, heap and registers is deferred until the child makes any changes to them during the execution time.

The child process is an exact duplicate of the parent process except for the following points:

- The child has its own unique process ID, and this PID does not match the ID of any existing process group or session.
- The child's parent process ID is the same as the parent's process ID.
- The child does not inherit its parent's memory locks.
- Process resource utilizations and CPU time counters are reset to zero in the child.
- The child's set of pending signals is initially empty.
- The child does not inherit semaphore adjustments from its parent.
- The child does not inherit process-associated record locks from its parent.
- The child does not inherit timers from its parent.
- The child does not inherit outstanding asynchronous I/O operations from its parent, nor does it inherit any asynchronous I/O contexts from its parent.

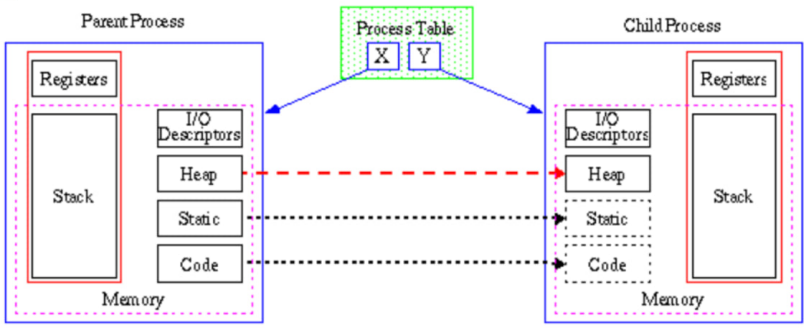

Figure-1

States between a parent process and a child process

- When a parent process in user mode is in the `running` state and calling `fork()`, the parent process is trapped into kernel mode and the OS takes over the work of creating a process. After that, the parent process will get back to the user mode and will be in the `ready` state waiting for its turn to run.
- The newly created process will be in the `new` state. OS will do process initialization and resource allocation.
- After that, the child process is in the `ready` state and is placed in a queue of processes that are ready to run. The CPU scheduler will later select a process from this queue to run.

- A parent process can wait for the child process to terminate before proceeding. If the parent process uses `wait()`, then the child process termination status and execution times are returned to the parent.
- When a child process completes but the parent process has not yet collected the exit status of the child, then the child process is a `Zombie` process. A Zombie process is a process that has completed its execution but still has an entry in the process table. Using `wait()` in the parent process could help us avoid leaving the child process in the `Zombie` state.
- A parent process can also run concurrently with the child, continuing to process without waiting. If the parent process terminates before its child processes, then the child processes become `orphaned`. In Unix, the init process (such as `systemd`) typically takes on the role of reaping orphaned child processes.

Practical use case

Parent Process	Child Process
Shell	Command Execution
Browser	Tabs and Extensions

- In a shell environment, if the command is a built-in command known by the shell (`echo`, `break`, `exit`, `test`, and so forth), it is executed internally without creating a new process. If the command is not a built-in, the shell treats it as an executable file. The parent process, the shell, waits until the child either completes execution or dies, and then it returns to read the next command.
- Web browsers like Google Chrome use multiple processes to enhance stability and security. Each tab, plugin, and extension typically runs in its own process.

Preferences

1. https://man7.org/linux/man-pages/man2/fork.2.html (2023-11-16)
2. ANDREW, S. Tanenbaum; HERBERT, Bos. Modern operating systems. Pearson Education, 2015.
3. ARPACI-DUSSEAU, Remzi H.; ARPACI-DUSSEAU, Andrea C. Operating systems: Three easy pieces. Arpaci-Dusseau Books, LLC, 2018.
4. LEI, Tony; LAM, Alfred. An Analysis on Google Chrome. 2010.
5. ROSEN, Kenneth H., et al. UNIX: the complete reference. McGraw-Hill, Inc., 2006.

6. SILBERSCHATZ, Abraham; GALVIN, Peter B.; GAGNE, Greg. Operating System Concepts, 9th ed. John Wiley & Sons, 2012.

7. Figure-1 source: https://sumeetjainengineer.wordpress.com/tag/processes/ (2023-11-16)