Analyzing the Process API: fork() vs. posix_spawn()

Kalki Srinivasan
Department of Computer Science, Mount Holyoke College

COMSC-322: Operating Systems

Prof. James McCauley

May 7, 2024

Abstract

This project is focused on the exploration and analysis of the operating system process API, centering on the fork() function and its modern alternative, posix_spawn(). Drawing inspiration from Baumann et al.'s 2019 work, "A fork() in a road," the project delves into the differences between the mechanisms, functionalities, advantages, and disadvantages of fork() and posix_spawn() in process creation code. It is motivated by existing research about the performance, efficiency, and security issues that fork() poses to operating systems. posix_spawn() is proposed as an alternative to fork(). This project aims to extend the proposed posix_spawn() alternative of fork() to practical applications. Open-source software code that involves process creation has been modified to leverage posix_spawn() for its performance, security, and resource benefits over fork(). Through this practical code adaptation, posix_spawn() is highlighted to have significant technical and functional implications and advantages for the process API over fork().

Analyzing the Process API: fork() vs. posix_spawn()

An API, or Application Programming Interface, is a set of rules, protocols, and tools for building software applications. It provides software components with policies and technologies for interaction and communication. Thus, they allow different software systems to communicate with each other by providing a standardized way for them to exchange data and instructions.

The process API is an API provided by an operating system (OS) or programming framework that allows users to interact with and manage any processes running on a computer system. It provides functionality for creating, terminating, scheduling, and communicating between processes. Specifically, it provides system calls to carry out process management. System calls are controlled jumps into kernel mode that are intentionally done by user mode code. In other words, system calls or syscalls are function calls into OS code that run at the kernel level, the level with the highest level of privilege and access within the system. They can be used for various process-related functionalities — creating new processes, waiting for processes to finish, retrieving information about processes, and managing process resources such as memory and CPU usage. To showcase the typical functionality of a syscall, consider the situation where a print statement is written in C and executed by the user. When the printing printf() function is invoked, the write() syscall is invoked within the OS, which signals the kernel that something should be written to the output stream.

fork() is one such syscall provided by an operating system process API that is used to create a new process. Once a new process is created, there exists an exec() syscall within the process API that allows the system to execute processes. Together, fork() and exec() work efficiently to create and execute processes — fork() creates a new process, and the new process is executed by exec().

fork()

When the fork() system call is called on by a program on a specific process, it creates a complete duplicate of the calling process. The newly created process, often referred to as the child of the parent calling process, is an exact copy of the parent process other than its process ID (PID). It doesn't require any parameters to be entered. Once the syscall is invoked, the PID of the copy created is returned. exec() can now be called on this PID returned, which would execute the copy process. A fork() syscall invoked on the child copy process would return 0.

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

```
int main() {
    pid_t pid;
    pid = fork();
    if (pid < 0) {
        // Fork failed
        fprintf(stderr, "Fork failed\n");
        return 1;
    } else if (pid == 0) {
        // Child process
        printf("Hello from child process %d!\n", getpid());
    } else {
        // Parent process
        printf("Hello from parent process%d!\n", getpid());
    }
    return 0;
}
```

Fig 1: Sample program using fork()

```
Hello from parent process 1234!
Hello from child process 1235!
```

Fig 2: Output of sample program in Fig 1
Here 1234 and 1235 are sample placeholders for actual PID’s

Here, the new child process created by fork is entirely independent of the original parent process. Thus, it runs entirely on its own after being made. This approach of fork() allows for multiple processes to run simultaneously. By giving the child process its own address space, it allows for each process to run independently of the others processes, as though they each had the full memory of the machine to themselves. However, with this provision of a new address space for the child process, fork() comes to be incompatible with single address space OS’s (Baumann et a., 2019, p. 4). While this creation of an independent child process poses both a benefit and a cost to the system, ultimately, the inherent mechanism of fork() to create a new process reduces program and system security. By default, the child inherits everything from its parent, and users become solely responsible for managing access and resources of the child. From a security perspective, this behavior of fork goes against the principle of least privilege which states that an entity should only have access to the specific data, resources and applications needed to complete a required task. By giving the child entity complete access to resources possessed by the parent, this principle is violated (Baumann et al., 2019, p.3).

posix_spawn()

posix_spawn() is a spawning function with mechanisms and functionalities highly similar to that of fork. However, it provides more control and flexibility, especially in terms of file actions and attributes (Baumann et al., 2019, p.5). An invocation of posix_spawn() both creates a new process and executes a specified program within that process. Through its direct creation and execution of a specified program within the created process, it has the combined functionality of fork() and exec(). Unlike fork(), posix_spawn() also takes in several parameters in order to run:

- A placeholder for the process ID for the child.
- A path of the program to be executed within the child process.
- An optional struct (posix_spawn_file_actions_t) that specifies actions to be performed on file descriptors during the spawn process.
- An optional struct (posix_spawn_file_actions_t) that specifies actions to be performed on file descriptors during the spawn process.
- An array of strings representing the arguments to be passed to the program being executed.
- An array of strings representing the environment variables to be passed to the new process.

It returns 0 on successful spawning (creation) of a new process. Otherwise, it returns an error code to signal failure to spawn.

The key difference between fork() and posix_spawn() is that posix_spawn does not create an exact copy of the parent process like fork(). While the copy process created by fork has the exact attributes as its parent, posix_spawn allows for more control over the attributes of the new process and the actions to be performed on file descriptors. Thus, if used for process management within an operating system's process API, posix_spawn() might pose a more efficient and concise way of creating and executing processes. However, it faces certain limitations (Baumann et al., 2019, p.5) that motivates the use of fork() over posix_spawn(). Firstly, it is not a complete replacement for the combined fork() and exec() mechanism because it lacks support for some common process management tasks, such as setting terminal attributes and switching into isolated directories. The second limitation concerns how posix_spawn handles any errors it comes across. Although it does return error codes to signal spawn failure, it lacks any explicit error-reporting mechanism making it difficult to identify the root of the error. Failures occurring in regards the created process are reported asynchronously and are indistinguishable from process termination.

```
#include <stdio.h>
#include <spawn.h>
#include <unistd.h>

int main() {
    pid_t pid;
    posix_spawn_file_actions_t actions;
    posix_spawn_file_actions_init(&actions);

    char *argv[] = {"./child_program", NULL};

    posix_spawn(&pid, "./child_program", &actions, NULL, argv, NULL);

    printf("Hello from parent process %d!\n", getpid());

    posix_spawn_file_actions_destroy(&actions);
    return 0;
}
```

Fig 3: Sample program using posix_spawn()

```
Hello from parent process 1234!
Child process reporting!
```

Fig 4: Output of sample program in Fig 3
Here 1234 and 1235 is a sample placeholders for a PID
'Child process reporting' is printed out due to the execution of the child_program program

Methods

To thoroughly understand differences in the performance, efficiency and usability of fork() and posix_spawn(), the current project examines three separate open source softwares that make use of fork() for software execution. These softwares were chosen based on whether software code uses fork() in a way that is modifiable and adaptable to alternative functions (like posix_spawn). After careful consideration of a range of softwares, the following three were choses to undergo code adaptation for the purposes of this project:

1. Sash (the Super Awesome SHell): Sash is a small and simple POSIX shell that can currently execute programs (Juliatto, 2019).
2. Toybox: Toybox is a Linux-based command-line interface that combines various common Linux command line utilities into a single executable software (Landley, 2019).
3. Htop: Htop is a cross-platform interactive process viewer. It allows for users to scroll along the list of processes vertically and horizontally to see their full command lines and related information like memory and CPU consumption (Htop-Dev, 2020).

Procedure

Github repositories of the selected open-source softwares were manually parsed through to understand the general purpose of the software. Simultaneously, software files within the repositories were checked for use of fork(). After confirming the utilization of fork() to create processes, ‘execute.c’ from Sash, ‘OpenFilesScreen.c’ from Htop, and ‘portability.c’ from Toybox were chosen to undergo the fork() to posix_spawn() adaptation. ‘execute.c’ was the first to undergo the transformation.

```
void execute(tokens t) {
pid_t pid = fork();
if (pid < 0) {
    //error checking code: for more information, please refer to code in cited repository
}
if (pid == 0) // we're on the child
 {
    execvp(t.tokens_list[0], t.tokens_list);
    print("An error has occurred. Could not launch command ");
    print(t.tokens_list[0]);
    print(".\n");
    exit(EXIT_FAILURE);
}

if (pid > 0){ // we're on the parent
   {
  int status;
  pid_t new_pid = waitpid(pid, &status, 0);
  if (new_pid != pid) {
      if (errno == ECHILD) {
         error_quit(NO_CHILD_EXECUTE,
             "FATAL ERROR: execute is waiting for a process that is not its child or it does not have one!");
       }
      if (errno == EINVAL) {
            error_quit(INVALID_EXECUTE,
                "FATAL ERROR: invalid arguments passed to waitpid in execute!");
        }
       if (errno != EINTR) {
            error_quit(UNKNOWN_ERROR,
                "FATAL ERROR: an unknown error has happened in the execute function!");
      }}}}
```

Fig 5: Original Sash code for program execute.c

Once the original code was analyzed for the way in which fork() was employed, a thorough modification to posix_spawn code was carried out. As mentioned previously, posix_spawn requires a number of parameters to be invoked. Parameters were identified from the original version of the code and altered to fit posix_spawn.

```
#include <spawn.h>
//parameters needed for posix_spawn
pid_t pid; //process ID
char *command = t.tokens_list[0]; //file or filepath
char **args = t.tokens_list; //arguments
posix_spawn_file_actions_t actions;
posix_spawn_file_actions_init(&actions); //file actions
```

Fig 6: posix_spawn()parameters for adaptation

Once the parameters were established, code regarding parent and child fork() processes were adapted to their posix_spawn counterparts. Because posix_spawn() doesn't create a copy process, fork() code parent and child processes were condensed into spawn management code. Aside from this difference, posix_spawn also makes use of waitpid() to wait on the spawned (created) process like fork() would.

```
/if posix_spawn() completes successfully, it would return 0
//test case for invalid posix_spawn (wouldn't have created a valid child process)
if (posix_spawn(&pid, command, &actions, NULL, args, NULL) != 0) {
   fprintf("posix_spawn");
   error_quit(UNKNOWN_ERROR, "FATAL ERROR: Failed to spawn process using posix_spawn!");
}

int status;

```

```
    //waiting for newly created process to return
    //if waitpid is successful, the created process returns
    if (waitpid(pid, &status, 0) == -1) {
        fprintf("waitpid");
        error_quit(UNKNOWN_ERROR, "FATAL ERROR: Failed to wait for spawned process!");
    }

    //handle the exit status of the created process
    if (WIFEXITED(status)) {
        printf("Spawned process exited with status %d\n", WEXITSTATUS(status));
    } else {
        printf("Spawned process did not exit normally\n");
    }

    posix_spawn_file_actions_destroy(&actions); //free resources related to file actions
}
```

Fig 7: posix_spawn()parameters for adaptation

Similarly, code for ‘OpenFilesScreen.c’ and ‘portability.c’ were adapted to facilitate the usage of posix_spawn for process creation. Once code modification was completed, the softwares were built and tested to ensure accuracy and consistency with previous behavior and expected functionality. Testing showed accurate and consistent results, confirming the reliability of the modified code.

Discussion

The project's primary purpose was to introduce the researcher to the process API in a research-focused format alongside a focus on utilizing the researched information in a practical setting. As the researcher involved in this project, I believe that I have been able to successfully explore the fork() aspect of the process API. Through this project, I have been able to conduct literature search and analysis, understand the intricacies of operating system process management and discover various novel utilizations of the process creation API in existing softwares. This process has strengthened my understanding of the larger concept of processes and how the OS interacts with the different programs and thereby, processes running on it. In the context of COMSC322, the research work involved in this project has enhanced my understanding of the programming assignments, worksheets and other class materials related to this topic. Moreover, working on this project has improved my skills and growth as a scientific researcher as it has allowed me to engage in extensive computer science research independently.

In terms of future research, it would be useful to explore other fork() alternatives. Baumann et al., 2019 introduced the idea of using vfork() and clone() as other potential alternatives alongside posix_spawn(). While vfork() is only a variant of the typical fork(), it creates a new process that shares the parent's address space until the child calls exec(). This would reduce the fork() cost of cloning the parent's address space. clone()'s behavior is highly similar to that of fork()'s but it provides more control over aspects of the child copy process. Modifying the open-source softwares used in this project to instead use these alternatives would allow researchers to explore the accessibility and usability of these functions. Researchers could also perform run-time analysis on the different versions of the softwares that use different process creation functions and identify the most optimal function for use in future software.

Ultimately, the project has enhanced my understanding of fork(), posix_spawn() and the process API in general. It has contributed to my growth as a scholar and a researcher. I hope that it will stimulate further investigation into the various aspects of the process API.

Acknowledgements

I would like to express my deepest appreciation to Prof. Murphy. Your constant support and assistance throughout my project journey has been incredibly important to its completion and success.

I would also like to express my unwavering gratitude to my friends and family for all the support they have given me during this stressful semester and my completion of this project.

Thank you.

References

Baumann, A, Appavoo, J., Krieger, O., & Roscoe, T. (2019, May 13-15). *A fork() in the road*. Workshop on Hot Topics in Operating Systems (HotOS '19), Bertinoro, Italy. ACM, New York, NY, USA. https://dl.acm.org/doi/10.1145/3317550.3321435

FORK(2) - Linux manual page. (n.d.). https://man7.org/linux/man-pages/man2/fork.2.html

Htop-Dev. (2020). *Htop-dev/htop: Htop - an interactive process viewer*. GitHub. https://github.com/htop-dev/htop

Juliatto, V. F. (2019). *VFabricio/Sash: A tiny POSIX shell, written in c*. GitHub. https://github.com/VFabricio/sash

Landley, R. (2019). *Landley/Toybox: Toybox.* GitHub. https://github.com/landley/toybox

POSIX_SPAWN(3) - linux manual page. (n.d.). https://man7.org/linux/man-pages/man3/posix_spawn.3.html

What is an API? - application programming interface explained - AWS. (n.d.). https://aws.amazon.com/what-is/api/