fork() and Alternatives

Maximilian Tim Schweikart

Abstract

The `fork()` system call is the de-facto way of creating processes in Linux. While it was designed to support the parallel execution of programs, it is today also often used for its copy-on-write capabilities. Even though `fork()` is used in a wide variety of use cases both in desktop and server software, it performs poorly in memory-intensive programs and violates fundamental design principles for operating systems.

This paper summarizes alternative approaches to process creation and discusses them with regard to these problems. We recommend to replace the `fork()` system call with a different set of system calls to create an API that has a reasonable complexity and respects security and memory management best practices.

1 Introduction

`fork()` is a Linux system call that creates an *almost* identical copy of the current process in a newly created address space. After `fork()`ing , both processes will continue execution at the `fork()` statement. The parent process and the newly created child process know whether they are the child process, allowing them to branch into different execution paths [14].

`fork()` is a powerful tool since it can be used in a wide variety of use cases: It was originally designed to start another program in a shell environment (in combination with the `exec()` system call) and to configure the standard input and output file descriptors of the created process (see §3.1) [3]. Numerous applications also make use of the copy-on-write optimization that `fork()` is implemented with. The copy-on-write behavior enables processes to duplicate their address space quickly which is useful both for starting up worker processes and for saving snapshots of the memory state in a time- and memory-efficient way (see §3.2, §3.3). The applications of these use cases range from shell programs to desktop and server software: In a study of the Ubuntu 15.04 Linux distribution, Tsai et al. found that `fork()` and its variant `vfork()` are used in 99.75 % of the applications and libraries [25].

The execution time of `fork()` is dominated by the time consumed by the address space copying process and grows linearly with the number of pages to copy. This has not been problematic so far due to the "copy-on-write" optimization and the small amount of memory processes used to take up. However, `fork()`'s execution time becomes problematic in high-performance or memory-intensive applications, which are increasingly common. For example, calling `fork()` in an application with 50 GB of allocated memory takes more than 250 ms of runtime. The performance of `fork()` significantly worsens if multiple processes attempt to fork concurrently (see §4) [27].

The `fork()` specification also violates multiple operating system design principles: `fork()`'s copy-by-default behavior encourages wasteful memory management. Accordingly, the inherit-by-default approach for access to resources like files and external devices conflicts with the zero-trust principle. `fork()` is both complicated to use and to implement. From the programmer perspective, the system call is not intuitive since it does not create an *exact* copy of the current process (e.g., threads cannot be copied and port access permissions are not inherited). From the operating system development perspective, `fork()` is hard to implement since the process state to copy is not limited to the address space of the process and might involve any external resources that are tied to the process [3].

We then describe both alternative implementations of the `fork()` specification and alternative APIs for process creation: `vfork()` and "on-demand-fork" are alternative implementations for `fork()` that mostly comply with its specification. They both reduce the execution time of the system call by avoiding copying a large number of pages on the system call invocation (see §6.1 and §6.2). A `spawn()` API in combination with a threading API separates the concerns of different use cases of `fork()` (starting independent processes and parallel programming) into different APIs (see §6.3 and §6.5). And a `clone()` API can be used to provide the quick process duplication capabilities that `fork()` was often used for while requiring the user to explicitly specify which parts of a process should be copied (see §6.4).

We argue that while faster implementations like `vfork()` and on-demand-fork can provide a performance that is sufficient for today's use cases, `fork()`'s flawed design can only be fixed by switching to a new API. Linux already implements many of the alternatives mentioned above although most of them rely on the `fork()` implementation internally (see §9).

2 The `fork()` system call

In the Linux operating system, the most common way to create a process from the user space is the `fork()` system call [2]. Calling `fork()` through its `glibc` wrapper function without any parameters will create a child process that is *almost* identical to the calling ("parent") process by duplicating the address space of the process [1]. Some resources of the parent process are copied (e.g., open file descriptors and memory mappings) where others are ignored (e.g., threads other than the one calling `fork()` and timers) [14]. Once the operating system has completed copying the address space of the parent, the execution in both the parent and the child process is resumed at the `fork()` statement. The child process will receive `0` and the parent process will receive the process identifier of the child as the return value of `fork()`. This allows the processes to identify whether they are the parent or the child process and to continue execution in different branches of the program [1, 14].

`fork()` is often used in combination with the `exec()` system call to start the execution of another program. When `exec()` is called, the operating system will replace the process image of the calling process with the process image from the executable file that is specified as a parameter of the system call. The operating system then resets some (e.g., memory mappings), but not all (e.g., some open file descriptors and the process identifier) process attributes and starts executing the image at the `main()` entry point of the program [22, 24].

Since duplicating the address space of a process can take a long time, Linux and many other UNIX operating systems (e.g., OpenBSD and Solaris) use a lazy copying approach called "copy-on-write" [14, 27, 7]. When calling `fork()`, only the kernel data structures of the process are duplicated and the page tables of both the parent and the child process will reference the same physical pages. All of these pages are marked as read-only and get a reference counter that denotes how many processes are currently referencing a certain page. Only when one of the referencing processes attempts to write to one of these pages, a page fault is triggered and the operating system will create an actual copy of the page, decreasing the reference counter of the original and setting the reference counter of the copy to `1`. Pages with a reference counter of `1` will be treated like normal pages again [1]. Copy-on-write requires hardware support in the memory management unit of the processor to translate virtual to physical addresses and call page fault handlers. However, most recent devices have a memory management unit and Linux does not support `fork()` at all if no memory management unit is available [1, 19]. A notable example for modern systems that do not have memory management units are embedded systems which often do not isolate processes [5].

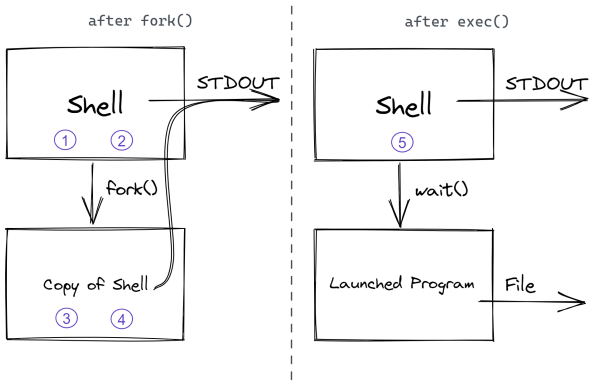

Figure 1: A shell `fork()`s to configure and `exec()`ute another program. The numbers show which process is responsible for which step.

3 Use cases for `fork()`

History suggests that the `fork()` system call was originally designed to create new processes in the context of shell applications [3] but today, numerous applications make heavy use of its address space duplication capability.

3.1 Starting programs in shells

Shell applications allow the user to run programs in a text-based interface. The user can start programs, configure them with arguments and chain them together using pipes and redirections.

Figure 1 shows how a shell makes use of `fork()` and `exec()` to run a program:

1. The user enters a command through the text interface. This command specifies which program to execute and whether to redirect the input or output to another program or file.

2. The shell `fork()`s, creating a copy of its process that still has access to the command data provided by the user.

3. The child process "Copy of Shell" then configures its input and output file descriptor as specified by the user command. In this case, the output is redirected to a file.

4. The child process calls `exec()` with the program path from the command to start this program.

5. The programs runs and the parent can `wait()` for the child to complete execution.

Baumann et al. argue that the design of the `fork()` and `exec()` system calls was influenced by how easy they could be implemented to support starting and configuring a program in a child process [3]. In this use case, the usage of the system calls is simple since it avoids (a) the need for a complex process creation system call that would allow configuring the child process with file descriptors etc. and (b) complex inter-process communication for passing configuration data to the child process.

Since the created process is a child of the shell process, the `wait()` and `kill()` system calls provide easy interfaces for monitoring and stopping the program [8, 15].

3.2 Snapshotting

Since `fork()` copies the address space of the process almost exactly, it can be used to create a consistent snapshot of the address space state, even in applications with frequently changing data that use multiple threads [14, 27]. This technique is commonly used in transactional systems like database servers where the process creates "checkpoint" of its memory state using a `fork()` system call. The child process can either write its memory state to secondary storage or allow the main process to retrieve data from its earlier state through interprocess communication [27]. Snapshotting is also used for applications that need to read a consistent state of the process's data without interrupting it. In this case, the application can `fork()` to create a consistent snapshot in the main memory and then analyze that data (commonly used in debuggers) or write it to secondary storage (used for backups in database servers). Both of these cases benefit from the copy-on-write implementation of `fork()` since most of the data does not actually need to be copied [27, 3].

3.3 Parallel programming

`fork()` can also be used for parallel programming: A program can distribute its workload to different processes by `fork()`ing (multiple times) and `wait()`ing for the child processes to complete their work. In master-worker architectures (commonly used in serverless computing platforms and web servers), `fork()` can be used to quickly spin up new worker processes [27, 3].

Thanks to copy-on-write, this approach has a small memory footprint since most parts of the program do not need be copied but can still be accessed by all child processes. For example, the source data and program instructions in a master-worker architecture are usually not modified by the worker processes and thus do not need to be copied. If the child processes do need to write a lot of data to complete their work, this approach still takes advantage of the isolated address spaces of the child processes: After `fork()`ing, the child processes can modify the source data set in memory without disturbing the execution of the other child processes. However, it is hard for the parent and the child processes to communicate about further instructions and computed results since they cannot access the same address space anymore and need to use interprocess communication instead.

3.4 Launching independent programs

Some programs start other programs without the need to monitor or interact with the other program afterwards. Specifically, this includes desktop applications where the user can request to open a file or web-link that the application does not support by itself. Some background processes like task schedulers also need to start other processes without further interactions.

These use cases can be implemented with `fork()` and `exec()` too by configuring open file descriptors to close upon the calling of `exec()` and by closing and disconnecting other resources that the child process would inherit.

Since System V, `fork()` can also be used to start daemon processes in Unix systems [18]. If a process wants to start a program as a daemon process, it can use a "double-`fork()`" to start a process that will continue its execution independently from the user. For that, the main process calls `fork()` to create a child process in the background and the child process assigns itself to a new session. The child process then calls `fork()` again to create a grandchild process and exits immediately. Since the child process died, the grandchild process is assigned to

the Linux init process. While daemon processes can still be created through a double-`fork()` in modern versions of Linux, the use of "System V Daemons" is discouraged in favor of "New-Style" daemons [18]. This alternative approach uses service units to start daemon processes that are supervised by the "systemd" system and service manager and does not require the user to use `fork()` any more.

4 `fork()`'s performance

Since `fork()` needs to copy the whole address space of the `fork()`ing process, the execution time of the system call grows linearly with the amont of pages that it takes up. With the copy-on-write optimization implemented, the execution time of `fork()` still grows linearly with the amount of pages of the original process, but on a smaller scale. In turn, the execution time of the page fault handler increases because it needs to copy pages that are referenced by multiple processes if one of the processes attempts to write to it.

Zhao et al. ran a benchmark on a 16-core AMD EPYC CPU with 256 GB of main memory and could observe such a linear growth both in sequential and concurrent executions of `fork()` [27]. The result of this benchmark can be seen in Figure 2. In this clinical example, small processes in the range of up to 1 GB of allocated memory can be `fork()`ed within up to 8 ms while it takes around 250 ms to `fork()` large processes in the range of 50 GB of allocated memory even without dirty pages. It it also notable that `fork()` performs significantly worse if multiple processes `fork()` concurrently. In the demonstrated example, three instances of the benchmark are run concurrently. For the range of allocated memory that has been covered by the benchmark, `fork()`ing concurrently increases the execution time by a factor of two to three, compared to sequential execution [27].

The authors argue that while this execution time is acceptable in many cases (i.e., desktop applications, background processes), it is becoming the bottleneck for performance-critical applications with a large memory footprint like database and caching servers since they rely on `fork()` for frequent snapshotting [27].

5 `fork()`'s design

`fork()` violates several design principles of operating systems. The copy-by-default behaviour encourages wasteful memory management and inattentive

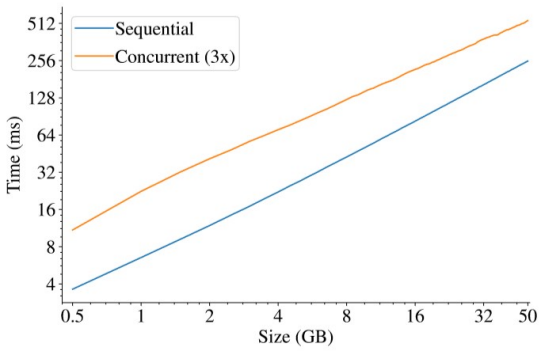

Figure 2: Execution time of the `fork()` system call, plotted as a function of the size of the forked process [27].

security practices. Additionally, both the implementation and usage of `fork()` are highly complex since the system call promises to create an equivalent copy of a process but makes numerous exceptions.

5.1 Memory management

According to its specification, `fork()` duplicates a process including its address space [14]. This approach is wasteful, considering that in many cases, the child process does not require access to all data from its parent. The extreme case, where another independent program is started and no data from the parent process other than the program path is required, is common in desktop applications and task schedulers (see 3.4).

Even with copy-on-write implemented, the operating system has to predict how much main memory the forked process will actually need: If the operating system reserves enough memory to copy all pages from the parent process eventually, it cannot take advantage of the shared parts between the parent and the child process (e.g., the code segment). If the operating system on the other hand only reserves as much memory as is needed immediately, it is likely that it needs to reserve additional memory in the near future. This can also lead to "memory overcommitment": If the processes share a lot of memory at first, a lot of them can fit inside the main memory. If the processes suddenly need to create copies of the shared memory (e.g., because a load of work is dispatched and distributed to them), the processes might need more space than available [3].

5.2 Complexity

As demonstrated above, `fork()` serves lots of different use cases, some of which center around using the

copy-on-write implementation of `fork()` rather than the creation of a new process. While `fork()` was simple to use in the context of a shell, the API has become unintuitive and complex to use since its behaviour is influenced by numerous system call flags [3]. In contrast, seemingly primitive operations like creating a new process independently from the current one require knowledge about what resources the current process is using and what process attributes are set [3].

The Linux `fork()` man page lists 16 process properties that are not duplicated exactly or not duplicated at all by the system call [14]. Furthermore, access to some external resources (e.g., the GPU) is not managed by the operating system and cannot be duplicated through the `fork()` system call itself [3]. Programmers need to carefully consider the numerous exceptions concerning the similarity of the duplicated process when using `fork()`.

5.3 Abstraction

When calling `fork()`, the child process is expected to inherit access to the resources that the parent has access to so that it could behave the same as its parent process if it executed the same instructions. In reality, this does not even work for many of Linux's built-in tools: the man page for `fork()` lists exceptions for numerous resources (like threads, timers, port access permissions) [14]. Additionally, if the process has access to resources on external stateful hardware (like graphics, machine learning or cryptography modules), the access to these resources and the state of the process on these resources would need to be duplicated too through the `fork()` system call. However, the operating system is not aware of the process state on all external devices and thus it is often unspecified whether the state on external devices is duplicated or shared correctly [3].

This violates the principle of abstraction on operating systems: It is impossible to change the underlying behaviour that implements this process creation functionality because the implementation is dictated by the specification. Specifically, this makes it impossible to implement the functionality of `fork()` on top of a system that does not support `fork()` itself for a number of reasons [3]:

- The state of new process's address space can neither quickly nor reliably be copied from the old process.
- The `fork()` implementation might not have access to all resources that the process is using.
- In operating systems without virtual memory addresses, the address space of new process cannot have the same range of addresses.

5.4 Security

The "inherit-access-by-default"-mechanism violates the principle of least privilege [3]. In use cases where another program should be started independently from the calling process, the child process must make sure to close file descriptors and disable access to external resources that are not intended to be available to the program that is executed afterwards [14]. This approach is error-prone because access to files and resources can be requested at any point in a program.

Additionally, since `fork()` duplicates address spaces, the parent process and the child process will necessarily have the same address space layout. This hinders address space layout randomisation and makes the system more vulnerable to memory corruption attacks [3].

6 Alternatives to fork()

As we have shown, both the performance and the design of Linux's `fork()` do not meet the expectations for a modern general-purpose operating system. We will now take a look at alternatives to both the implementation and the specification of `fork()` and what is needed in the Linux kernel to support them.

6.1 vfork()

In cases where `fork()` is used in combination with `exec()`, it can be unnecessary to create a copy of the process's address space in the first place if the child process never modifies the copied address space. The `vfork()` system call was first introduced into the BSD operating system for that exact reason: It can be used similarly to `fork()` but the operating system does not copy the address space and instead runs the child process in the address space of the parent process until `exec()` is called. The parent can resume its execution after the child process has started the new program through `exec()` [3].

It is noteworthy that the operating system does not stop a child process from writing to the parent address space and specifies this case as undefined behaviour. Since even the modification of a register might result in undefined behaviour, the use of `vfork()` is discouraged. `vfork()` has even been removed from the POSIX specification, although it is still present in the Linux specification and

not marked as obsolete to be used in performance-critical applications and by systems without a memory management unit [4].

As `vfork()` does not need to copy the address space of the process or the corresponding page table, its runtime is independent from the amount of allocated memory in the parent process. Thus, it can solve the performance issues with `fork()` – in cases where `vfork()` can be used. However, since `vfork()` works the same way as `fork()`, `vfork()` suffers from the same design issues as `fork()`, other than memory management.

6.2 On-demand-fork

Even with the copy-on-write optimization, the execution time of `fork()` is dominated by copying pages because `fork()` still needs to copy and modify the page table pages of the address space. Zhao et al. suggest to apply the copy-on-write optimization to first-level page table pages too [27]. Similar to the usual copy-on-write implementation, the last-level page tables are marked as read-only in memory and each get a reference counter when forking. Only if one of the referencing processes attempts to write to an address space page contained in a read-only page table, both the page table and the referenced page are actually copied (and their reference counter is decreased) [27].

Figure 3 shows a benchmark from the authors of on-demand-fork that compares the execution times of the Linux `fork()` implementation and their modified `fork()` implementation. The benchmark has been run on a 16-core AMD EPYC 7302P processor with 16 GB of main memory and averaged over 5 runs [27]. In this environment, the default Linux `fork()` implementation takes 65 to 270 times longer to execute compared to on-demand-fork in the same benchmark. The relative performance advantage of on-demand-fork increases with the size of the address space to copy and the absolute execution time is below 1 ms for all measured address space sizes.

Zhao et al. have implemented on-demand-fork as a separate system call in a copy of the Linux source code [26]. To the best of our knowledge, it has not been adopted in any major Linux distribution.

6.3 spawn()

A `spawn()` API creates a new process which is independent from the calling process and runs a program that is provided as a path through a parameter of the API call [3].

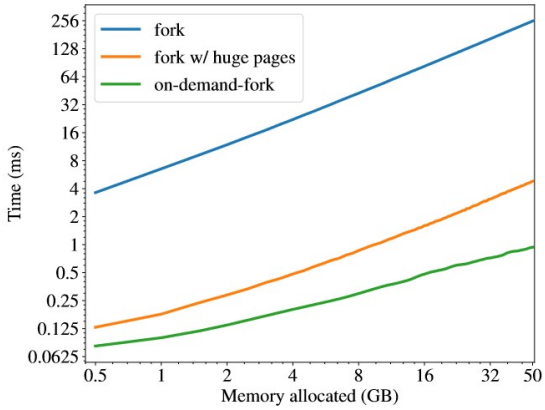

Figure 3: Benchmark by the authors of on-demand-fork, comparing the execution time of `fork()`, `fork()` with huge pages and on-demand-fork [27]

Linux implements the `posix_spawn()` API on top of the `clone()` and `exec()` system calls. The process created through `posix_spawn()` is not completely independent from the process that created it, e.g., file descriptors from the parent process are still be duplicated by default [9]. However, the `posix_spawn()` API does offer a set of abstractions for configuring the process that is created. For example, the user can pass a set of `posix_spawn_file_actions` as a parameter to specify which file descriptors should be opened, closed or even duplicated [9]. This can be considered as a first step towards creating a process spawning API that is independent from the parent process.

Oracle's Solaris uses a different approach than Linux: Since version 11.4 (released in 2018), the `posix_spawn()` implementation uses a new `spawn()` system call instead of `vfork()` and `exec()`. Notably, the `spawn()` system call does not duplicate the address space of the calling process and the threads in the calling process [7].

6.4 clone()

Some use cases of `fork()` benefit from the quick process duplication that is enabled through copy-on-write (see §3.2 and §3.3). But even though `fork()` is designed specifically for process duplication, there are numerous differences between the parent and the child process (see §5.2). Meanwhile, some similarities between the parent and the child process are not intended (e.g., access to resources, see §5.4).

A `clone()` API allows process duplication and gives the programmer more fine-grained control over what is being copied [3]. This increases awareness for the limitations of duplicating a process while still

allowing for a fast, `fork()`-like copy-on-write implementation.

Linux implements the `clone()` and `clone3()` system calls [12]. Both system calls allow the programmer to configure which resources of a process to duplicate. For example, the programmer can specify whether all threads are duplicated, what other process should be used as a parent process and whether the original and the new process should share an address space [12].

Snapshotting use cases (see §3.2) can benefit from the precision that this system call allows for in comparison to `fork()` by limiting what resources will be copied [3].

6.5 Threading

A process is an operating system abstraction for the execution of a program. Since the operating system can execute multiple processes concurrently or in parallel, they can be used for parallel programming (see §3.3). In these use cases, the isolation between different processes is usually more a hindrance than a benefit and interprocess communication is necessary to distribute workloads and to report results.

Kernel level threads can be used to execute multiple threads within the same process concurrently or in parallel. Since all threads of a process are located in the same address space, they can communicate easily and efficiently using the heap.

Linux implements the POSIX threads API, which allows the user to create, manage and communicate with threads within the current process in a similar manner to dealing with processes [13]. For example, POSIX threads can wait for the termination of other threads through the `pthread_join()` function [11]

7 Discussion

The on-demand-fork implementation (see §6.2) increases the performance of the system call without changing its specification. With an execution time of less than 1 ms for up to 50 GB of allocated address space and an expected linear growth, `fork()` will not be the bottleneck of high-performance and real-time applications any more. Consequently, the performance problem that Linux's current `fork()` implementation entails can be solved without changing the specification of the system call.

The violations of the `fork()` system call against operating design principles (see §5), however, cannot be solved through better implementations since they originate in the system call's specification. Baumann et al. suggest to replace `fork()` with a set of other APIs that each implement a well-defined subset of `fork()`'s capabilites: A spawn API (see §6.3) could be responsible for launching independent programs in a new process (see §3.2) while a threading API (see §6.5) should be used for parallel programming, if isolation between the threads is not required or even a hindrance [3]. Since this approach does not entail the copy-everything behaviour of `fork()`, the design issues described above do not apply to it.

Still, this replacement offers no performant method to duplicate processes and their address spaces in isolation. A `clone()` API (see §6.4) could be used to implement this functionality. If this API required the user to explicitly specify which parts of the process state to clone, it would not suffer from the complexity and security issues of `fork()` (see §5.2 and §5.4). The memory management issue (see §5.1) would not be solved but it would be limited to use cases of process cloning and would not affect all use cases of process creation. Similarly, the abstraction issue (see §5.3) would not be solved but through the opt-in API, nobody would assume that everything is cloned. Access to external resources that are not managed by the kernel directly would need to be requested again by the program.

8 Related Work

The Linux implementation of `fork()` conforms to the POSIX.1-2008 specification [14, 10]. Other Unix-like and Unix-based operating systems such as OpenBSD and macOS also have conforming implementations for the `fork()` system call [21, 20, 23]. While the POSIX specification of `fork()` is open to some extensions, the fundamental design problems outlined in §5 apply to all POSIX-compliant `fork()` implementations. The complexity of using `fork()` is even worse when writing code that should support multiple Unix-like platforms because the POSIX specification allows for additional exceptions.

Similarly, the `posix_spawn()`, `clone()`and `Pthread` APIs that were outlined as alternatives to the `fork()` system call (see §6) are also available on POSIX-compliant systems [10].

Microsoft Windows does not offer `fork()` or a similar system call through the Win32 API. Instead, it specifies the `CreateProcess()` API which works like a `spawn()` API (see §6.3) [3, 16]. In fact, Windows does not expose process duplication capabilities like those of `fork()` or `clone()` to the userspace at all [6, 17]. Thus, the Windows toolset for process creation does not suffer from the problems discussed above (see §4, §5) but it also cannot provide the quick process duplication capabilities that are used in snapshotting

use case (see §3.2).

9 Conclusion

While the limited performance of `fork()` can be mitigated through better implementations of the copying strategy of `fork()`, the underlying design problems cannot be solved without changing the API. Operating system developers need to carefully specify a set of new process creation primitives that respect the fundamental design principles of an operating system. Linux already implements multiple POSIX APIs that solve specific problems, even though all of these implementations still rely on the `fork()` or `clone()` system calls internally. Since these APIs can jointly replace the `fork()` system call, it is evidently not a necessity to support `fork()`. Linux should start the long deprecation process of `fork()` to encourage developers to switch to its other APIs. This will reduce the amount of bugs and security issues related to process creation and remove unnecessary constraints from the operating system itself.

References

[1] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, Aug. 2018. URL: http://www.ostep.org/ (visited on 11/26/2021).

[2] V. Atlidakis, J. Andrus, R. Geambasu, D. Mitropoulos, and J. Nieh. POSIX abstractions in modern operating systems: the old, the new, and the missing. In *Proceedings of the Eleventh European Conference on Computer Systems*. Association for Computing Machinery, 2016. URL: https://doi.org/10.1145/2901318.2901350.

[3] A. Baumann, J. Appavoo, O. Krieger, and T. Roscoe. A fork() in the road. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 14–22. Association for Computing Machinery, 2019. URL: https://doi.org/10.1145/3317550.3321435.

[4] A. Brouwer and M. Kerrisk. Linux programmer's manual: vfork(2), version 5.13, Mar. 22, 2021. URL: https://man7.org/linux/man-pages/man2/vfork.2.html (visited on 03/29/2022).

[5] X.-H. Cheng, Y.-m. Gong, and X.-z. Wang. Study of embedded operating system memory management. In *2009 First International Workshop on Education Technology and Computer Science*, volume 3, pages 962–965, 2009.

[6] Cygwin. Cygwin user's guide: highlights of cygwin functionality, version 3.3.4, Jan. 31, 2022. URL: https://cygwin.com/cygwin-ug-net/highlights.html#ov-hi-process (visited on 03/29/2022).

[7] C. Dik. Posix_spawn() as an actual system call. Oracle Solaris Blog. Feb. 12, 2018. URL: https://blogs.oracle.com/solaris/post/posix_spawn-as-an-actual-system-call (visited on 05/06/2022).

[8] D. Eckhardt. Linux programmer's manual: kill(2), version 5.13, Mar. 22, 2021. URL: https://man7.org/linux/man-pages/man2/kill.2.html.

[9] B. O. Gallmeister and M. Kerrisk. Linux programmer's manual: posix_spawn(3), version 5.13, Mar. 22, 2021. URL: https://man7.org/linux/man-pages/man2/mmap.2.html.

[10] IEEE standard for information technology - portable operating system interface (POSIX(r)). *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)*:1–3874, 2008.

[11] M. Kerrisk. Linux programmer's manual: pthread_join(3), version 5.13, Mar. 22, 2021. URL: https://man7.org/linux/man-pages/man3/pthread_join.3.html (visited on 03/23/2022).

[12] M. Kerrisk. Linux programmer's manual: pthreads(7), version 5.13, Mar. 22, 2021. URL: https://man7.org/linux/man-pages/man7/pthreads.7.html (visited on 03/23/2022).

[13] M. Kerrisk and D. Eckhardt. Linux programmer's manual: clone(2), version 5.13, Mar. 22, 2021. URL: https://man7.org/linux/man-pages/man2/clone.2.html.

[14] M. Kerrisk and D. Eckhardt. Linux programmer's manual: fork(2), version 5.13, Mar. 22, 2021. URL: https://man7.org/linux/man-pages/man2/fork.2.html.

[15] T. Koenig and M. Kerrisk. Linux programmer's manual: wait(2), version 5.13, Mar. 22, 2021. URL: https://man7.org/linux/man-pages/man2/wait.2.html.

[16] Microsoft. Windows app development: creating processes, Oct. 9, 2021. URL: https://docs.microsoft.com/de-de/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa.

[17] Microsoft. Windows subsystem for linux: pico process overview. Microsoft Blog Archive. May 23, 2016. URL: https://docs.microsoft.com/de-de/archive/blogs/wsl/pico-process-overview (visited on 03/29/2022).

[18] L. Poettering and K. Sievers. Systemd manual: daemon(7), version 249, July 7, 2021. URL: https://man7.org/linux/man-pages/man7/daemon.7.html (visited on 05/04/2022).

[19] The kernel development community. No-MMU memory mapping support, version 5.13, Mar. 22, 2021. URL: https://www.kernel.org/doc/html/latest/admin-guide/mm/nommu-mmap.html (visited on 05/02/2022).

[20] The Open Group. Open brand certificate for macOS version 12.0 monterey on apple silicon-based mac computers, Dec. 10, 2021. URL: https://www.opengroup.org/openbrand/certificates/1215p.pdf (visited on 03/28/2022).

[21] The Open Group. Open brand certificate for macOS version 12.0 monterey on intel-based mac computers, Dec. 10, 2021. URL: https://www.opengroup.org/openbrand/certificates/1214p.pdf (visited on 03/28/2022).

[22] The Regents of the University of California. Linux programmer's manual: exec(3), version 5.13, Mar. 22, 2021. URL: https://man7.org/linux/man-pages/man3/exec.3.html.

[23] The Regents of the University of California. OpenBSD manual pages: fork(2), version 7.0, Sept. 10, 2015. URL: https://man.openbsd.org/OpenBSD-7.0/fork.2.

[24] The Regents of the University of California. OpenBSD source code, version 7.1, Apr. 21, 2022. URL: https://github.com/openbsd/src (visited on 05/01/2022).

[25] C.-C. Tsai, B. Jain, N. A. Abdul, and D. E. Porter. A study of modern linux API usage and compatibility: what to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems*. Association for Computing Machinery, 2016. URL: https://doi.org/10.1145/2901318.2901341 (visited on 05/01/2022).

[26] K. Zhao and P. Fonseca. On-demand-fork: source code, Dec. 28, 2021. URL: https://github.com/rssys/on-demand-fork (visited on 02/17/2022).

[27] K. Zhao, S. Gong, and P. Fonseca. On-demand-fork: a microsecond fork for memory-intensive and latency-sensitive applications. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 540–555. Association for Computing Machinery, 2021. URL: https://doi.org/10.1145/3447786.3456258 (visited on 02/17/2022).