A fork() in the road

Andrew Baumann
Microsoft Research

Jonathan Appavoo
Boston University

Orran Krieger
Boston University

Timothy Roscoe
ETH Zurich

ABSTRACT

The received wisdom suggests that Unix's unusual combination of `fork()` and `exec()` for process creation was an inspired design. In this paper, we argue that fork was a clever hack for machines and programs of the 1970s that has long outlived its usefulness and is now a liability. We catalog the ways in which fork is a terrible abstraction for the modern programmer to use, describe how it compromises OS implementations, and propose alternatives.

As the designers and implementers of operating systems, we should acknowledge that fork's continued existence as a first-class OS primitive holds back systems research, and deprecate it. As educators, we should teach fork as a historical artifact, and not the first process creation mechanism students encounter.

ACM Reference Format:
Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. 2019. A `fork()` in the road. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3317550.3321435

1 INTRODUCTION

When the designers of Unix needed a mechanism to create processes, they added a peculiar new system call: `fork()`. As every undergraduate now learns, fork creates a new process identical to its parent (the caller of fork), with the exception of the system call's return value. The Unix idiom of `fork()` followed by `exec()` to execute a *different* program in the child is now well understood, but still stands in stark contrast to process creation in systems developed independently of Unix [e.g., 1, 30, 33, 54].

HotOS '19, May 13–15, 2019, Bertinoro, Italy

ACM ISBN 978-1-4503-6727-1/19/05…$15.00
https://doi.org/10.1145/3317550.3321435

50 years later, fork remains the default process creation API on POSIX: Atlidakis et al. [8] found 1304 Ubuntu packages (7.2% of the total) calling fork, compared to only 41 uses of the more modern `posix_spawn()`. Fork is used by almost every Unix shell, major web and database servers (e.g., Apache, PostgreSQL, and Oracle), Google Chrome, the Redis key-value store, and even Node.js. The received wisdom appears to hold that fork is a good design. Every OS textbook we reviewed [4, 7, 9, 35, 75, 78] covered fork in uncritical or positive terms, often noting its "simplicity" compared to alternatives. Students today are taught that "the `fork` system call is one of Unix's great ideas" [46] and "there are lots of ways to design APIs for process creation; however, the combination of `fork()` and `exec()` are simple and immensely powerful … the Unix designers simply got it right" [7].

Our goal is to set the record straight. Fork is an anachronism: a relic from another era that is out of place in modern systems where it has a pernicious and detrimental impact. As a community, our familiarity with fork can blind us to its faults (§4). Generally acknowledged problems with fork include that it is not thread-safe, it is inefficient and unscalable, and it introduces security concerns. Beyond these limitations, fork has lost its classic simplicity; it today impacts all the other operating system abstractions with which it was once orthogonal. Moreover, a fundamental challenge with fork is that, since it conflates the process and the address space in which it runs, fork is hostile to user-mode implementation of OS functionality, breaking everything from buffered IO to kernel-bypass networking. Perhaps most problematically, fork *doesn't compose*—every layer of a system from the kernel to the smallest user-mode library must support it.

We illustrate the havoc fork wreaks on OS implementations using our experiences with prior research systems (§5). Fork limits the ability of OS researchers and developers to innovate because any new abstraction must be special-cased for it. Systems that support fork and exec efficiently are forced to duplicate per-process state lazily. This encourages the centralisation of state, a major problem for systems not structured using monolithic kernels. On the other hand, research systems that avoid implementing fork are unable to run the enormous body of software that uses it.

We end with a discussion of alternatives (§6) and a call to action (§7): fork should be removed as a first-class primitive of our systems, and replaced with good-enough emulation for legacy applications. It is not enough to add new primitives to the OS—fork must be removed from the kernel.

2 HISTORY: FORK BEGAN AS A HACK

Although the term originates with Conway, the first implementation of a fork operation is widely credited to the Project Genie time-sharing system [61]. Ritchie and Thompson [70] themselves claimed that Unix fork was present "essentially as we implemented it" in Genie. However, the Genie monitor's fork call was more flexible than that of Unix: it permitted the parent process to specify the address space and machine context for the new child process [49, 71]. By default, the child shared the address space of its parent (somewhat like a modern thread); optionally, the child could be given an entirely different address space of memory blocks to which the user had access; presumably, in order to run a different program. Crucially, however, there was no facility to copy the address space, as was done unconditionally by Unix.

Ritchie [69] later noted that "it seems reasonable to suppose that it exists in Unix mainly because of the ease with which fork could be implemented without changing much else." He goes on to describe how the first fork was implemented in 27 lines of PDP-7 assembly, and consisted of copying the current process out to swap and keeping the child resident in memory.[1] Ritchie also noted that a combined Unix fork-exec "would have been considerably more complicated, if only because exec as such did not exist; its function was already performed, using explicit IO, by the shell."

The TENEX operating system [18] yields a notable counter-example to the Unix approach. It was also influenced by Project Genie, but evolved independently of Unix. Its designers also implemented a fork call for process creation, however, more similarly to Genie, the TENEX fork either shared the address space between parent and child, or else created the child with an empty address space [19]. There was no Unix-style copying of the address space, likely because virtual memory hardware was available.[2]

Unix fork was *not* a necessary "inevitability" [61]. It was an expedient PDP-7 implementation shortcut that, for 50 years, has pervaded modern OSes and applications.

3 ADVANTAGES OF THE FORK API

When Unix was rewritten for the PDP-11 (with memory translation hardware permitting multiple processes to remain resident), copying the process's entire memory only to immediately discard it in exec was already, arguably, inefficient. We suspect that copying fork survived the early years of Unix mainly because programs and memory were small (eight 8 KiB pages on the PDP-11), memory access was fast relative to instruction execution, and it provided a compelling abstraction. There are two main aspects to this:

Fork was simple. As well as being easy to implement, fork simplified the Unix API. Most obviously, fork needs no arguments, because it provides a simple default for all the state of a new process: inherit it from the parent. In stark contrast, the Windows `CreateProcess()` API takes explicit parameters specifying every aspect of the child's kernel state—10 parameters and many optional flags.

More significantly, creating a process with fork is orthogonal to starting a new program, and the space between fork and exec serves a useful purpose. Since fork duplicates the parent, the same system calls that permit a process to modify its kernel state can be reused in the child prior to exec: the shell opens, closes, and remaps file descriptors prior to command execution, and programs can reduce permissions or alter the namespace of a child to run it in restricted context.

Fork eased concurrency. In the days before threads or asynchronous IO, fork without exec provided an effective form of concurrency. In the days before shared libraries, it enabled a simple form of code reuse. A program could initialise, parse its configuration files, and then fork multiple copies of itself that ran either different functions from the same binary or processed different inputs. This design lives on in pre-forking servers; we return to it in §6.

4 FORK IN THE MODERN ERA

At first glance, fork still seems simple. We argue that this is a deceptive myth, and that fork's effects cause modern applications more harm than good.

Fork is no longer simple. Fork's semantics have infected the design of each new API that creates process state. The POSIX specification now lists 25 special cases in how the parent's state is copied to the child [63]: file locks, timers, asynchronous IO operations, tracing, etc. In addition, numerous system call flags control fork's behaviour with respect to memory mappings (Linux `madvise()` flags `MADV_DONTFORK/DOFORK/WIPEONFORK`, etc.), file descriptors (`O_CLOEXEC`, `FD_CLOEXEC`) and threads (`pthread_atfork()`). Any non-trivial OS facility must document its behaviour across a fork, and user-mode libraries must be prepared for their state to be forked at any time. The simplicity and orthogonality of fork is now a myth.

Fork doesn't compose. Because fork duplicates an entire address space, it is a poor fit for OS abstractions implemented in user-mode. Buffered IO is a classic example: a user must explicitly flush IO prior to fork, lest output be duplicated [73].

Fork isn't thread-safe. Unix processes today support threads, but a child created by fork has only a single thread (a copy of the calling thread). Unless the parent serialises fork with respect to its other threads, the child address space may

[1] Sharing memory between parent and child (as in Genie) was impractical, because the PDP-7 lacked virtual memory hardware; instead, Unix implemented multiprocessing by swapping full processes to disk.

[2] TENEX also supported copy-on-write memory, but this does not appear to have been used by fork [20].

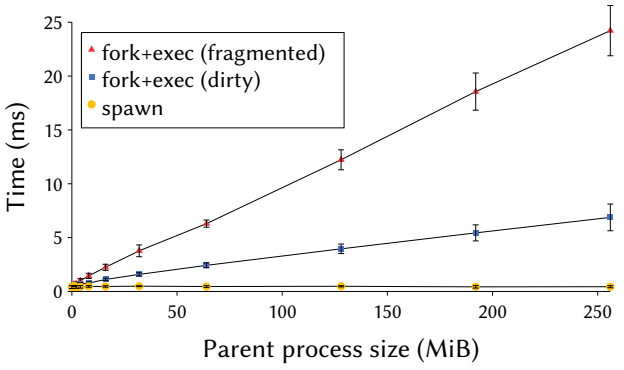

Figure 1: Cost of `fork()` + `exec()` vs. `posix_spawn()`

end up as an inconsistent snapshot of the parent. A simple but common case is one thread doing memory allocation and holding a heap lock, while another thread forks. Any attempt to allocate memory in the child (and thus acquire the same lock) will immediately deadlock waiting for an unlock operation that will never happen.

Programming guides advise not using fork in a multi-threaded process, or calling exec immediately afterwards [64, 76, 77]. POSIX only guarantees that a small list of "async-signal-safe" functions can be used between fork and exec, notably excluding `malloc()` and anything else in standard libraries that may allocate memory or acquire locks. Real multi-threaded programs that fork are plagued by bugs arising from the practice [24–26, 66].

It is hard to imagine a new proposed syscall with these properties being accepted by any sane kernel maintainer.

Fork is insecure. By default, a forked child inherits everything from its parent, and the programmer is responsible for explicitly removing state that the child does not need by: closing file descriptors (or marking them as close-on-exec), scrubbing secrets from memory, isolating namespaces using `unshare()` [52], etc. From a security perspective, the inherit-by-default behaviour of fork violates the principle of least privilege. Furthermore, programs that fork but don't exec render address-space layout randomisation ineffective, since each process has the same memory layout [17].

Fork is slow. In the decades since Thompson first implemented fork, memory size and relative access cost have grown continuously. Even by 1979 (when the third BSD Unix introduced `vfork()` [15]) fork was seen as a performance problem, and only copy-on-write techniques [3, 72] kept its performance acceptable. Today, even the time to establish copy-on-write mappings is a problem: Chrome experiences delays of up to 100 ms in fork [28], and Node.js applications can be blocked for seconds while forking prior to exec [56].

Fork is now such a performance liability that C libraries carefully avoid its use in `posix_spawn()` [34, 38], and Solaris implements spawn as a native system call [32]. However, as long as applications continue to call fork directly, they pay a high price. Figure 1 plots the time to fork and exec from a process of varying size under Ubuntu 16.04.3 on an Intel i7-6850K CPU at 3.6 GHz. The *dirty* line shows the cost of forking a process with dirty pages, which must be downgraded to read-only for copy-on-write mappings. In the *fragmented* case, the parent dirties only its stack, but simulates memory layout in a complex application using shared libraries, address space randomisation, and just-in-time compilation, by allocating alternating read-only and read-write pages. By contrast, `posix_spawn()` takes the same time (around 0.5 ms) regardless of the parent's size or memory layout.

Fork doesn't scale. In Linux, the memory management operations needed to setup fork's copy-on-write mappings are known to hurt scalability [22, 82], but the true problem lies deeper: as Clements et al. [29] observed, the mere specification of the fork API introduces a bottleneck, because (unlike spawn) it fails to commute with other operations on the process. Other factors further impede a scalable implementation of fork. Intuitively, the way to make a system scale is to avoid needless sharing. A forked process starts sharing everything with its parent. Since fork duplicates every aspect of a process's OS state, it encourages centralisation of that state in a monolithic kernel where it is cheap to copy and/or reference count. This then makes it hard to implement, e.g., kernel compartmentalisation for security or reliability.

Fork encourages memory overcommit. The implementer of fork faces a difficult choice when accounting for memory used by copy-on-write page mappings. Each such page represents a potential allocation—if any copy of the page is modified, a new page of physical memory will be needed to resolve the page fault. A conservative implementation therefore fails the fork call unless there is sufficient backing store to satisfy all potential copy-on-write faults [55]. However, when a large process performs fork and exec, many copy-on-write page mappings are created but never modified, particularly if the exec'ed child is small, and having fork fail because the *worst-case* allocation (double the virtual size of the process) could not be satisfied is undesirable.

An alternative approach, and the default on Linux, is to *overcommit* virtual memory: operations that establish virtual address mappings, which includes fork's copy-on-write clone of an address space, succeed immediately *regardless* of whether sufficient backing store exists. A subsequent page fault (e.g. a write to a forked page) can fail to allocate required memory, invoking the heuristic-based "out-of-memory killer" to terminate processes and free up memory.

To be clear, Unix does not require overcommit, but we argue that the widespread use of copy-on-write fork (rather

than a spawn-like facility) strongly encourages it. Real applications are unprepared to handle apparently-spurious out-of-memory errors in fork [27, 37, 57]. Redis, which uses fork for persistence, explicitly advises against disabling memory overcommit [67]; otherwise, Redis would have to be restricted to only half the total virtual memory to avoid the risk of being killed in an out-of-memory situation.

Summary. Fork today is a convenient API for a single-threaded process with a small memory footprint and simple memory layout that requires fine-grained control over the execution environment of its children but does not need to be strongly isolated from them. In other words, a shell. It's no surprise that the Unix shell was the first program to fork [69], nor that defenders of fork point to shells as the prime example of its elegance [4, 7]. However, most modern programs are not shells. Is it still a good idea to optimise the OS API for the shell's convenience?

5 IMPLEMENTING FORK

While it is hard to quantify the cost of implementing fork on existing systems, there is clear evidence that supporting fork limits changes in OS architecture, and restricts the ability of OSes to adapt with hardware evolution.

Fork is incompatible with a single address space. Many modern contexts restrict execution to a single address space, including picoprocesses [42], unikernels [53], and enclaves [14]. Despite the fact that a much larger community of OS researchers work with and on Unix systems, researchers working with systems not based on fork have had a much easier time adapting them to these environments.

For example, the Drawbridge libOS [65] implements a binary-compatible Windows runtime environment within an isolated user-mode address space, known as a picoprocess. Drawbridge supports multiple "virtual processes" within the same shared address space; `CreateProcess()` is implemented by loading the new binary and libraries in a different portion of the address space, and then creating a separate thread to begin execution of the child, while ensuring cross-process system calls function as expected. Needless to say, there is no security isolation between these processes—the meaningful security boundary is the host picoprocess. However, this model has been used, for example, to support a full multi-process Windows environment inside an SGX enclave [14], enabling complex applications that involve multiple processes and programs to be deployed in an enclave.

In contrast, fork is unimplementable within a single address space [23] without complex compiler and linker modifications [81]. As a result, Unikernels derived from Unix systems do not support internal multi-process environments [44, 45] and running multi-process Linux applications in an enclave is much more complicated. SCONE and SGX-LKL support only single-process applications [6, 50]. Graphene-SGX [79] implements fork by creating a new enclave in a new host process, then copying the parent's memory over an encrypted RPC stream; this can take seconds.

Fork is incompatible with heterogeneous hardware. Fork conflates the abstraction of a process with the hardware address space that contains it. In effect, fork *restricts the definition of a process* to a single address space and (as we saw earlier) a single thread running on some core.

Modern hardware, and the programs that run on it, just don't look like this. Hardware is increasingly heterogeneous, and a process using, say, DPDK with a kernel-bypass NIC [12], or OpenCL with a GPU, cannot safely fork since the OS cannot duplicate the process state on the NIC/GPU. This appears to have been a continuing source of bafflement among GPU programmers for a decade at least [58–60, 74]. As future systems-on-chip incorporate more and more stateful accelerators, this is only going to get worse.

Fork infects an entire system. The mere choice to support fork places significant constraints on the system's design and runtime environment. An efficient fork at any layer requires a fork-based implementation at all layers below it. For example, Cygwin is a POSIX compatibility environment for Windows; it implements fork in order to run Linux applications. Since the Win32 API lacks fork, Cygwin emulates it on top of `CreateProcess()` [31, 47]: it creates a new process running the same program as the parent and copies all writable pages (data sections, heap, stack, etc.) before resuming the child. This is neither fast nor reliable and can fail for many reasons, most often when memory addresses in parent and child differ due to address-space layout randomisation.

Ironically, the NT kernel natively supports fork; only the Win32 API on which Cygwin depends does not (user-mode libraries and system services are not fork-aware, so a forked Win32 process crashes). As an abstraction, *fork fails to compose*: unless every layer supports fork, it cannot be used.

Fork in a research OS: the K42 experience

Many research operating systems have faced the dilemma of whether (and if so, how) to implement fork, with the authors having direct experience of six [13, 36, 41, 48, 51, 80]. This choice has significant implications. Implementing fork opens the door to a large class of Unix-derived applications, first among them shells and build tools that ease the construction of a complete system. However, it also ties the researchers' hands: we conjecture that a system that implements fork, particularly one that attempts to do so efficiently, or early in its life, inexorably converges to a Unix-like design.

K42 [48] built on our experience with Tornado [36] that demonstrated the value of a multi-processor-friendly object-oriented approach, per-application customisable objects, and microkernel architecture [5] to enable pervasive locality and

concurrency optimisations. Our goal was to construct a fully-fledged general-purpose OS supporting a wide range of applications using *multiple OS personalities* on (potentially) very large multiprocessors. In the end, K42 was POSIX compliant and Linux ABI compatible, but the quest to make fork perform for the Linux personality caused fork semantics to subvert the OS design to the detriment of other personalities.

We initially assumed that we would be able to implement fork much like Cygwin: as a user-level library function that would create a child copy of the existing process by appropriately constructing the necessary new object instances. This by itself was not necessarily problematic. Rather, it was the choice to allow any process at any point to be forkable, and to do so efficiently in the quest of competitive performance, that proved our undoing—the attendant complexity may very well have been key to us abandoning all but our support for Unix and our native personalities.

In particular, the following problems permeated almost every aspect of the system:

Anti-modularity: Each object implementation, of any type that may back a running process, needed to define its behaviour when the process forked. This greatly increased the complexity of implementing specialised components whose sole purpose might simply be to introduce a locality optimisation for a long-lived parallel scientific computation or server that has no real reason to fork.

Inherent need for laziness: Given that every core resource, from memory regions and files to personality-specific abstractions like file descriptors and signal handlers, all required fork support, we were driven to implement lazy copy-on-write-like behaviour to mitigate poor performance. Not only did this induce complexity within individual objects, it required object interactions to maintain the hierarchical relationships created by fork. This was counter to our aim of limiting sharing and synchronisation, harming locality.

Centralisation: OS scalability is helped by avoiding centralised policies and using mechanisms that eschew exact global knowledge [11]. Decomposing state and functionality across object instances and servers thus became our core philosophy. However, despite fork being coordinated in library code, it required communication with every server and object a process may be connected to.

Poor scalability: Besides violating our core scalability principles, fork in a NUMA system must either access memory at the parent's location or schedule its children in contended parts of the system; these are inherent issues we spent much effort addressing.

In hindsight, we made a mistake in not carefully assessing the real use-cases for fork. Had we special-cased K42's fork to single-threaded processes (e.g., shells), we could have avoided burdening core objects with its complexity.

6 REPLACING FORK

Given all the problems with fork, what should replace it? Creating a new process leads to a messy API design problem, as any option must, implicitly or explicitly, specify the initial state of every OS resource belonging to the new process. Fork has an easy answer: everything is copied, which as we've seen becomes its undoing. In its place, we propose a combination of a high-level spawn API and a lower-level microkernel-like API to setup a new process prior to execution. We then discuss alternatives to fork without exec.

High-level: Spawn. In our opinion, most uses of fork and exec would be best served by a spawn API. The refactoring required to make this change can be tricky, particularly when fork and exec are not proximate in code, but as we showed in §4 there are significant performance and reliability advantages, not to mention portability. Notably, major applications that fork (e.g., Apache, Chrome, PostgreSQL) have Windows ports that don't, so fork is clearly not essential.

The `posix_spawn()` API can ease such refactoring. Rather than requiring that all parameters affecting a new process be provided at a single call-site (as is the case for `CreateProcess()`), spawn attributes are set by extensibly-defined helper functions. A post-fork `close()`, for example, can be replaced by a pre-spawn call that records a "close action" to occur in the child. Unfortunately, this means that the API is specified as if it were implemented by fork and exec, although it is not actually required [32].

The main drawback of `posix_spawn()` is that it is not a complete replacement for fork and exec. Some less-common operations, such as setting terminal attributes or switching to an isolated namespace, are not yet supported. It also lacks an effective error-reporting mechanism: failures occurring in the context of the child before it begins execution (such as invalid file descriptor parameters) are reported asynchronously and are indistinguishable from the child's termination. These shortcomings can and should be corrected.

Alternative: `vfork()`. This widely-implemented fork variant was introduced by BSD as an optimisation [15]; it creates a new process that shares the parent's address space until the child calls exec, more like the original Genie fork [71]. To enable the child to use the parent's stack, it blocks execution of the parent until exec. This permits a similar style of programming to fork in which a new process modifies its kernel state prior to exec. However, because of the shared address space, `vfork()` is difficult to use safely [34]. Although `vfork()` avoids the cost of cloning the address space, and may help to replace fork where refactoring to use spawn is impractical, in most cases it is better avoided.

Low-level: Cross-process operations. While a spawn-like API is preferred for most instances of starting a new program, for full generality it requires a flag, parameter, or

helper function controlling every possible aspect of process state. It is infeasible for a single OS API to give complete control over the initial state of a new process. In Unix today, the only fallback for advanced use-cases remains code executed after fork, but clean-slate designs [e.g., 40, 43] have demonstrated an alternative model where system calls that modify per-process state are not constrained to merely the current process, but rather can manipulate any process to which the caller has access. This yields the flexibility and orthogonality of the fork/exec model, without most of its drawbacks: a new process starts as an empty address space, and an advanced user may manipulate it in a piecemeal fashion, populating its address-space and kernel context prior to execution, without needing to clone the parent nor run code in the context of the child. ExOS [43] implemented fork in user-mode atop such a primitive. Retrofitting cross-process APIs into Unix seems at first glance challenging, but may also be productive for future research.

Alternative: `clone()`. This syscall underlies all process and thread creation on Linux. Like Plan 9's `rfork()` which preceded it, it takes separate flags controlling the child's kernel state: address space, file descriptor table, namespaces, etc. This avoids one problem of fork: that its behaviour is implicit or undefined for many abstractions. However, for each resource there are two options: either share the resource between parent and child, or else copy it. As a result, clone suffers most of the same problems as fork (§4–5).

Fork-only use-cases. There exist special cases where fork is not followed by exec, that rely on duplicating the parent.

Multi-process servers. Traditionally the standard way to build a concurrent server was to fork off processes. However, the reasons that motivated multi-process servers are long gone: OS libraries are thread-safe, and the scalability bottlenecks that plagued early threaded or event-driven servers are fixed [10]. While process boundaries may have value from a fault isolation perspective, we believe that it makes more sense to use a spawn API to start those processes. The performance advantage of the shared initial state created by fork is less relevant when most concurrency is handled by threads, and modern operating systems deduplicate memory. Finally, with fork, all processes share the same address-space layout and are vulnerable to Blind ROP attacks [17].

Copy-on-write memory. Modern implementations of fork use copy-on-write to reduce the overhead of copying memory that is often soon discarded [72]. A number of applications have since taken a dependency on fork merely to gain access to copy-on-write memory. One common pattern involves forking from a pre-initialised process, to reduce startup overhead and memory footprint of a worker process, as in the Android Zygote [39, 62] and Chrome site isolation on Linux [4]. Another pattern uses fork to capture a consistent snapshot of a running process's address space, allowing the parent to continue execution; this includes persistence support in Redis [68], and some reverse debuggers [21].

POSIX would benefit from an API for using copy-on-write memory independently of forking a new process. Bittau [16] proposed `checkpoint()` and `resume()` calls to take copy-on-write snapshots of an address space, thus reducing the overhead of security isolation. More recently, Xu et al. [82] observed that fork time dominates the performance of fuzzing tools, and proposed a similar `snapshot()` API. These designs are not yet general enough to cover all the use-cases outlined above, but perhaps can serve as a starting point. We note that any new copy-on-write memory API must tackle the issue of memory overcommit described in §4, but decoupling this problem from fork should make it much simpler.

7 GET THE FORK OUT OF MY OS!

We've described how fork is a relic of the past that harms applications and OS design. There are three things we must do to rectify the situation.

Deprecate fork. Thanks to the success of Unix, future systems will be stuck supporting fork for a long time; nevertheless, an implementation hack of 50 years ago should not be permitted to dictate the design of future OSes. We should therefore strongly discourage the use of fork in new code, and seek to remove it from existing apps. Once fork is gone from performance-critical paths, it can be removed from the core of the OS and reimplemented on top as needed. If future systems supported fork only in limited cases, such as a single-threaded process [2], it would remain possible to run legacy software without needless implementation complexity.

Improve the alternatives. For too long, fork has been the generic process creation mechanism on Unix-like systems, with other abstractions layered on top. Thankfully, this has begun to change [32, 38], but there is more to do (§6).

Fix our teaching. Clearly, students need to learn about fork, however at present most text books (and we presume instructors) introduce process creation with fork [7, 35, 78]. This not only perpetuates fork's use, it is counterproductive—the API is far from intuitive. Just as a programming course would not today begin with `goto`, we suggest teaching either `posix_spawn()` or `CreateProcess()`, and then introducing fork as a special case with its historic context (§2).

ACKNOWLEDGEMENTS

We thank all who provided feedback, including: Tom Anderson, Remzi Arpaci-Dusseau, Marc Auslander, Bill Bolosky, Ulrich Drepper, Chris Hawblitzel, Eddie Kohler, Petros Maniatis, Mathias Payer, Michael Stumm, Robbert Van Renesse, and the anonymous reviewers.

REFERENCES

[1] *The BeBook: The Kernel Kit: load_image()*. ACCESS Co., 1.0 edition, March 2008. URL https://www.haiku-os.org/legacy-docs/bebook/TheKernelKit_Images.html#load_image.

[2] *The BeBook: Threads and Teams*. ACCESS Co., 1.0 edition, March 2008. URL https://www.haiku-os.org/legacy-docs/bebook/TheKernelKit_ThreadsAndTeams_Overview.html.

[3] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *USENIX Summer Conference*, pages 93–113, June 1986.

[4] Thomas Anderson and Michael Dahlin. *Operating Systems: Principles and Practice*. Recursive Books, 2nd edition, 2014. ISBN 978-0-9856735-2-9.

[5] Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, Amos Waterland, Robert W. Wisniewski, Jimi Xenidis, Michael Stumm, and Livio Soares. Experience distributing objects in an SMMP OS. *ACM Transactions on Computer Systems*, 25(3), August 2007. ISSN 0734-2071. doi: 10.1145/1275517.1275518.

[6] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 689–703. USENIX Association, 2016. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

[7] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*, chapter 5. Arpaci-Dusseau Books, 1.00 edition, March 2018. URL http://www.ostep.org/.

[8] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. POSIX abstractions in modern operating systems: The old, the new, and the missing. In *EuroSys Conference*, pages 19:1–19:17. ACM, 2016. ISBN 978-1-4503-4240-7. doi: 10.1145/2901318.2901350.

[9] Jean Bacon and Tim Harris. *Operating Systems: Concurrent and Distributed Software Design*. Addison Wesley, 2003. ISBN 0-321-11789-1.

[10] Gaurav Banga and Jeffrey C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *1998 USENIX Annual Technical Conference*. USENIX Association, 1998. URL https://www.usenix.org/legacy/publications/library/proceedings/usenix98/banga.html.

[11] Amnon Barak, Shai Guday, and Richard G. Wheeler. *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Springer-Verlag Berlin Heidelberg, 1993. doi: 10.1007/3-540-56663-5.

[12] Dotan Barak. *Libibverbs Programmer's Manual: ibv_fork_init(3)*, October 2006. URL https://github.com/linux-rdma/rdma-core/blob/master/libibverbs/man/ibv_fork_init.3.md.

[13] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schuepbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *22nd ACM Symposium on Operating Systems Principles*. ACM, October 2009. doi: 10.1145/1629575.1629579.

[14] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation*, pages 267–283, October 2014. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann.

[15] *2.9.1 BSD System Calls Manual: vfork(2)*. Berkeley Software Distribution, Berkeley, CA, USA, 1983. URL https://www.freebsd.org/cgi/man.cgi?query=vfork&manpath=2.9.1+BSD.

[16] Andrea Bittau. *Toward Least-Privilege Isolation for Software*. PhD thesis, Department of Computer Science, University College London, November 2009. URL http://www.scs.stanford.edu/~sorbo/bittau-phd.pdf.

[17] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *IEEE Symposium on Security and Privacy*, pages 227–242. IEEE Computer Society, 2014. ISBN 978-1-4799-4686-0. doi: 10.1109/SP.2014.22.

[18] Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, and Raymond S. Tomlinson. TENEX, a paged time sharing system for the PDP-10. In *3rd ACM Symposium on Operating Systems Principles*. ACM, 1971. doi: 10.1145/800212.806492.

[19] *TENEX JSYS Manual*. Bolt Beranek and Newman, Cambridge, MA, USA, 2nd edition, September 1973. URL http://www.bitsavers.org/pdf/bbn/tenex/TenexJSYSMan_Sep73.pdf.

[20] *TENEX 1.33 source code,* CFORK *system call*. Bolt Beranek and Newman, 1975. URL https://github.com/PDP-10/tenex/blob/master/133-tenex/forks.mac#L208.

[21] Bob Boothe. Efficient algorithms for bidirectional debugging. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 299–310. ACM, 2000. ISBN 1-58113-199-2. doi: 10.1145/349299.349339.

[22] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. OpLog: a library for scaling update-heavy data structures. Technical Report MIT-CSAIL-TR-2014-019, MIT CSAIL, September 2014. URL http://hdl.handle.net/1721.1/89653.

[23] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(4):271–307, November 1994. ISSN 0734-2071. doi: 10.1145/195792.195795.

[24] Chromium Project. Bug 36678, 2010. URL https://crbug.com/36678.

[25] Chromium Project. Bug 56596, 2010. URL https://crbug.com/56596.

[26] Chromium Project. Bug 177218, 2013. URL https://crbug.com/177218.

[27] Chromium Project. Bug 856535, 2018. URL https://crbug.com/856535.

[28] Chromium Project. Bug 819228, 2018. URL https://crbug.com/819228.

[29] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Transactions on Computer Systems*, 32(4):10:1–10:47, January 2015. ISSN 0734-2071. doi: 10.1145/2699681.

[30] *OpenVMS System Services Reference Manual: $CREPRC*. Compaq Computer Corporation, Houston, TX, USA, April 2001. URL http://h30266.www3.hpe.com/odl/vax/opsys/vmsos73/vmsos73/4527/4527pro_018.html#jun_147. Document number ZK4527.

[31] *Cygwin 2.11 User's Guide*. Cygwin, November 2018. URL https://cygwin.com/cygwin-ug-net/highlights.html#ov-hi-process.

[32] Casper Dik. posix_spawn() as an actual system call. Oracle Solaris Blog, February 2018. URL https://blogs.oracle.com/solaris/posix_spawn-as-an-actual-system-call.

[33] D. Eastlake, R. Greenblatt, J. Holloway, T. Knight, and S. Nelson. *ITS 1.5 Refereence Manual*. MIT Artificial Intelligence Laboratory, Cambridge, MA, USA, July 1969. URL https://hdl.handle.net/1721.1/6165. Memo number AIM-161A.

[34] Rich Felker. vfork considered dangerous. October 2012. URL https://ewontfix.com/7.

[35] Greg Gagne, Abraham Silberschatz, and Peter B. Galvin. *Operating Systems Concepts*. John Wiley & Sons, 9th edition, 2012. ISBN 978-1-118-06333-0.

[36] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *3rd USENIX Symposium on Operating Systems Design and Implementation*, February 1999. URL https://www.usenix.org/legacy/events/osdi99/gamsa.html.

[37] GNOME Project. Merge request 95, 2018. URL https://gitlab.gnome.org/GNOME/glib/merge_requests/95.

[38] GNU C Library. Bug 10354, 2016. URL https://sourceware.org/bugzilla/show_bug.cgi?id=10354.

[39] *Android Developer Documentation: Overview of memory management*. Google, 2018. URL https://developer.android.com/topic/performance/memory-overview#SharingRAM.

[40] Gernot Heiser and Kevin Elphinstone. L4 microkernels: The lessons from 20 years of research and deployment. *ACM Transactions on Computer Systems*, 34(1):1:1–1:29, April 2016. ISSN 0734-2071. doi: 10.1145/2893177.

[41] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software, Practice and Experience*, 28(9):901–928, 1998.

[42] Jon Howell, Bryan Parno, and John R. Douceur. How to run POSIX apps in a minimal picoprocess. In *2013 USENIX Annual Technical Conference*, pages 321–332. USENIX Association, 2013. URL https://www.usenix.org/conference/atc13/technical-sessions/presentation/howell.

[43] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *16th ACM Symposium on Operating Systems Principles*, pages 52–65, 1997. ISBN 0-89791-916-5. doi: 10.1145/268998.266644.

[44] Antti Kantee. On rump kernels and the Rumprun unikernel, August 2015. URL https://xenproject.org/2015/08/06/on-rump-kernels-and-the-rumprun-unikernel/.

[45] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv—optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference*, pages 61–72, 2014. ISBN 978-1-931971-10-2. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity.

[46] Eddie Kohler. Harvard University CS 61 problem set 4: WeensyOS, October 2018. URL https://cs61.seas.harvard.edu/site/2018/WeensyOS/. See also https://twitter.com/xexd/status/951977086331359232.

[47] David G. Korn. Porting UNIX to Windows NT. In *1997 USENIX Annual Technical Conference*, January 1997. URL https://www.usenix.org/legacy/publications/library/proceedings/ana97/korn.html.

[48] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a complete operating system. In *EuroSys Conference*, pages 133–145. ACM, 2006. ISBN 1-59593-322-0. doi: 10.1145/1217935.1217949.

[49] Butler W. Lampson. SDS 940 lectures. June 1966. URL http://archive.computerhistory.org/resources/text/SDS/sds.lampson.SDS_940_lectures.1966.102634499.pdf.

[50] *SGX-LKL*. Large-Scale Data & Systems Group, Imperial College London, 2018. URL https://github.com/lsds/sgx-lkl.

[51] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996. doi: 10.1109/49.536480.

[52] *Linux Programmer's Manual: unshare(2)*. Linux man-pages project, March 2019. URL http://man7.org/linux/man-pages/man2/unshare.2.html.

[53] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 461–472. ACM, 2013. ISBN 978-1-4503-1870-9. doi: 10.1145/2451116.2451167.

[54] *Windows API: CreateProcessW function*. Microsoft, April 2018. URL https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessw.

[55] Greg Nakhimovsky. Minimizing memory usage for creating application subprocesses. Sun Microsystems, May 2006. URL https://www.oracle.com/technetwork/server-storage/solaris10/subprocess-136439.html.

[56] Node.js. Issue 14917, 2018. URL https://github.com/nodejs/node/issues/14917.

[57] Node.js. Issue 25382, 2019. URL https://github.com/nodejs/node/issues/25382.

[58] Nvidia Developer Forum. CUDA and fork(), December 2007. URL https://devtalk.nvidia.com/default/topic/382954/cuda-programming-and-performance/cuda-and-fork-/.

[59] Nvidia Developer Forum. Linux fork() and CUDA OOM possible bug, March 2009. URL https://devtalk.nvidia.com/default/topic/453458/linux-fork-and-cuda-oom-possible-bug-/.

[60] Nvidia Developer Forum. (CUDA8.0 BUG?) Child process forked after cuInit() get CUDA_ERROR_NOT_INITIALIZED on cuInit(), October 2016. URL https://devtalk.nvidia.com/default/topic/973477/-cuda8-0-bug-child-process-forked-after-cuinit-get-cuda_error_not_initialized-on-cuinit-/.

[61] Linus Nyman and Mikael Laakso. Notes on the history of fork and join. *IEEE Annals of the History of Computing*, 38(3):84–87, July 2016. ISSN 1058-6180. doi: 10.1109/MAHC.2016.34.

[62] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference*, pages 57–70, 2018. ISBN 978-1-931971-44-7. URL https://www.usenix.org/conference/atc18/presentation/oakes.

[63] *Base Specifications POSIX.1-2017*. The Open Group, San Francisco, CA, USA, 2018. URL http://pubs.opengroup.org/onlinepubs/9699919799/functions/fork.html. IEEE Std 1003.1-2017.

[64] Damian Pietras. Threads and fork(): think twice before mixing them. June 2009. URL https://www.linuxprogrammingblog.com/threads-and-fork-think-twice-before-using-them.

[65] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library OS from the top down. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–304. ACM, 2011. ISBN 978-1-4503-0266-1. doi: 10.1145/1950365.1950399.

[66] Python Project. Issue 27126, 2016. URL https://bugs.python.org/issue27126.

[67] *Redis FAQ: Background saving fails with a fork() error under Linux*. Redis, 2018. URL https://redis.io/topics/faq.

[68] *Redis Persistence*. Redis, 2018. URL https://redis.io/topics/persistence.

[69] Dennis M. Ritchie. The evolution of the Unix time-sharing system. In Jeffrey M. Tobias, editor, *Language Design and Programming Methodology*, volume 79 of *Lecture Notes in Computer Science*, pages 25–35. Springer, 1980. ISBN 978-3-540-38579-0. doi: 10.1007/3-540-09745-7_2.

[70] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974. ISSN 0001-0782. doi: 10.1145/361011.361061.

[71] *SDS 940 Time-Sharing System Technical Manual*. Scientific Data Systems, Santa Monica, CA, USA, November 1967. URL http://bitsavers.org/pdf/sds/9xx/940/901116A_940_TimesharingTechMan_Nov67.pdf. Publication number 90 11 16A.

[72] Jonathan M. Smith and Gerald Q. Maguire, Jr. Effects of copy-on-write memory management on the response time of UNIX fork operations. *Computing Systems: The Journal of the USENIX Association*, 1 (3):255–278, 1988. URL https://www.usenix.org/legacy/publications/

compsystems/1988/sum_smith.pdf.

[73] Stack Overflow. printf anomaly after fork(), March 2010. URL https://stackoverflow.com/questions/2530663/printf-anomaly-after-fork.

[74] Stack Overflow. CUDA initialization error after fork, April 2014. URL https://stackoverflow.com/questions/22950047/cuda-initialization-error-after-fork.

[75] William Stallings. *Operating Systems: Internals and Design Principles*. Pearson, 6th edition, 2009. ISBN 978-0-13-603337-0.

[76] W. Richard Stevens and Stephen A. Rago. *Advanced Programming in the UNIX Environment*. Addison Wesley, 3rd edition, 2013. ISBN 978-0-321-63773-4.

[77] *Multithreaded Programming Guide*. Sun Microsystems, Santa Clara, CA, USA, 2002. URL https://docs.oracle.com/cd/E19683-01/806-6867/. Part number 806-6867-11.

[78] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Pearson, 4th edition, 2015. ISBN 978-0-13-359162-0.

[79] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference*, pages 645–658. USENIX Association, 2017. ISBN 978-1-931971-38-6. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

[80] Ronald C. Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Experiences with locking in a NUMA multiprocessor operating system kernel. In *1st USENIX Symposium on Operating Systems Design and Implementation*, November 1994. URL https://www.usenix.org/legacy/publications/library/proceedings/osdi/unrau.html.

[81] Tim Wilkinson, Ashley Saulsbury, Tom Stiemerling, and Kevin Murray. Compiling for a 64-bit single address space architecture. Technical Report TCU/SARC/1993/1, Systems Architecture Research Centre, City University, London, UK, 1993.

[82] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *2017 ACM Conference on Computer and Communications Security*, pages 2313–2328. ACM, 2017. ISBN 978-1-4503-4946-8. doi: 10.1145/3133956.3134046.