Direct-style process creation on Linux

Spencer Baugh
Two Sigma

Abstract

Traditional process creation interfaces, such as `fork` and `spawn`, are complex to use, limited in power, and difficult to abstract over. We develop a new process creation interface for Linux which allows a program to create a child process in a non-running state and initialize the new process by operating on it from the outside. This method of process creation results in more comprehensible programs, has better error handling, is comparably efficient, and is more amenable to abstraction. Our implementation is immediately deployable without kernel modifications on any recent Linux kernel version.

1 Introduction

The most well-known process creation interface on Linux is `fork`. When a parent process creates a child process through `fork` or other fork-style interfaces, both processes execute concurrently, and the child process typically inherits all but a few attributes from the parent process [43]. The child process can call arbitrary system calls to mutate the new process until it reaches the desired state, typically ultimately calling `exec`.

A user of `fork` must think carefully about the attributes of the process in which `fork` is called. `fork` can have poor performance when called in processes with many memory mappings [7]. This can be mitigated to some degree by using `vfork` or `CLONE_VM`, but these techniques are difficult to use correctly [17]. Multi-threaded processes can encounter deadlocks and other issues if they call `fork` [7] [34] [18]. Some thread libraries provide partial mitigations for issues caused by fork, but these must be enabled and used properly by users [45].

Since the child process runs concurrently with the parent process, communicating events in the child back to the parent requires some form of IPC [18]. Even the basic interface of `fork` — one function call which returns twice — is unusual; in other settings, such functions are considered to be complex, advanced topics. [28] [50].

Closely related to `fork` are process creation interfaces where the new process is launched running a function of the caller's choice, which can then call arbitrary system calls to mutate the new process [40] [6]. Such interfaces have similar issues as `fork`, and we class them together as "fork-style" interfaces.

Conversely, in a spawn-style interface, most details about the new process are provided up front as arguments to some function which creates the new process all at once. `posix_spawn` [23] is a typical example of this style [61] [15]. Some other details, such as security context, are inherited from the parent process [34] [15]. Spawn-style APIs can be called from any parent process without concerns about memory usage or multi-threading, when used correctly [18] [7].

Unlike with `fork`, normal system calls cannot be used to customize a process started with a spawn-style interface [24] [20]. Spawn-style interfaces therefore often lack support for some features, and they also require learning a new interface distinct from the existing system call API [4] [34]. Also, spawn-style interfaces, by their nature as a single call, don't allow programs to branch on the results of individual modifications to the new process, and have worse error reporting for such modifications [23] [18].

Table 1 summarizes the differences between fork-style and spawn-style. Each has its own advantages and disadvantages.

A few operating systems — KeyKOS [10] and seL4 [26], among others [16] [39] [1] — use another style of process creation. We use the new term "direct-style" to refer to this, since we know of no pre-existing generic term for this style of process creation.

In direct-style, a process is created by its parent in a non-running state, then supplied with various resources, and then started running once it is fully set up. In operating systems with true direct-style process creation, the syscalls that can mutate a process take explicit arguments to indicate the process on which they should operate [10] [26]. In this way, the same syscalls that can mutate a process while it is running are called by the parent process to mutate the process while it is being set up.

	fork-style	spawn-style	direct-style
Parent process requirements	✗:Single thread, small memory	✓:None	✓:None
Programming model	✗:Complex (returns twice)	✓:Simple (single call)	✓:Simple (imperative)
Maximally powerful	✓:Yes, can call any syscall	✗:No, limited interface	✓:Yes, can call any syscall
Reporting of results	✗:Requires IPC	✗:Not fine-grained	✓:From individual child syscalls
Non-inherited attributes	Mutated by code in child	Set by arguments	Mutated by code in parent

Table 1: Features of fork-style vs spawn-style vs direct-style

Listing 1: Create new process, change CWD, and exec

```
child = local.clone()
child.chdir("/dev")
child.execv("/bin/cat", ["cat", "./null"])
```

To perform direct-style process creation on Linux, we need an API where we explicitly specify in which process we want to make a syscall. This differs from the normal mode of operation on Linux, where programs making syscalls implicitly operate on the current process.

Our main contribution in this paper is such an API, in the form of the `rsyscall` library. `rsyscall` is a language-specific, object-capability-model, low-abstraction library for the Linux system call interface, bypassing libc [52] [51]. The `rsyscall` project is currently focused on supporting Python, but the library can be ported to other languages. The examples we show in this paper will be in Python, but generalize easily. `rsyscall` is open source and available from https://github.com/catern/rsyscall.

Unlike typical C libraries such as glibc or musl, the `rsyscall` library is organized based on the object-capability model [52] [51]. The capability to make syscalls in any specific process is reified as a language object. If the capability to make syscalls in a specific process is not passed (in some way) to a function, then that function cannot make syscalls in that process.

Due to this design, an `rsyscall` program can make use of multiple processes at once, by manipulating capabilities for multiple processes. The relevant two types of capabilities for this paper are the initial capability for the "local" process, and capabilities for child processes. The "local" process is the one which hosts the runtime for the running program, and in which a legacy libc would implicitly make syscalls. Every `rsyscall` program starts with the capability for the "local" process and uses it to bootstrap capabilities for other processes.

Listing 1 shows a Python program using `rsyscall`. We create a new child process using direct-style `clone` in the local process `local`. Here, `local.clone()` does not create another task running the same user program, as with the usual `clone` system call, but instead returns a capability through which we can control the new child process. We can then call various syscalls in the child process to mutate it until it reaches the desired state. In this example, we call `chdir` in the child process to change its working directory, then call `execv` in the child to execute a new executable. Calling `execv` consumes the capability, releasing the process from our control; later use of this child process capability will fail with an exception. The child process can now be monitored using normal Linux child monitoring syscalls, such as `waitid`.

These child process capabilities are created and managed in userspace, by launching new processes running a syscall server, which receives syscall requests, performs the syscall, and returns the response. No kernel modifications are required, and `rsyscall` is immediately deployable on recent Linux kernels. We discuss the implementation in depth in section 3, and examine several difficult process-creation details.

System calls called in a child process through `rsyscall` behave identically to system calls called implicitly in the current process through libc. A system call returning an error is reported in the typical way for system calls in Python: An exception is thrown at the point of the call. User code can branch as normal on the results of system calls, and implement fallbacks or other error handling logic.

Since we can call any syscall, we can access any feature available in Linux; we are therefore at least as powerful as `fork`. As we'll show in section 2, we can in fact use Linux features in ways that are impractical with the usual fork-style or spawn-style interfaces.

Direct-style `clone` has acceptable performance cost, and can outperform `fork` in some cases. Like a spawn-style interface, we can call `clone` without worse performance in the presence of large memory usage, and without the possibility of bugs in the presence of multi-threading. The performance overhead of creating processes with direct-style clone is comparable to the performance cost of creating processes while inside a Linux namespace. At Two Sigma, we've used direct-style process creation to implement a library for distributed system deployment. We evaluate in further detail in section 4.

In summary, this paper makes the following contributions:

- We coin the term "direct-style process creation" to refer to a previously-unnamed style of process creation which is present on several operating systems.
- We built `rsyscall`, a library for the Linux system call interface following the object-capability model.
- As part of `rsyscall`, we built the first implementation

of direct-style process creation on a Unix-like kernel. `rsyscall` is open source and available from https://github.com/catern/rsyscall.

2 Examples

In this section, we'll demonstrate direct-style process creation by using several sophisticated features of Linux processes. For concision, we assume that we are running with sufficient privileges, but these examples can all be run without privileges with appropriate use of `CLONE_NEWUSER` [33] [9].

In these examples and in this paper, a single program with a single flow of control is operating across multiple processes. We'll always describe operations from the perspective of this single synchronous program, never from the perspective of an individual parent or child process.

These examples are not novel, as such, as they use only conventional Linux system calls; but implementing them with `fork` or `posix_spawn` requires substantially more code, or even a complete change in implementation approach as in section 2.6. We demonstrate this by comparison to fork-style implementations of these examples in section 4.1.

Several of these examples can be implemented by configuring and invoking existing software; such software effectively provides a specialized spawn-style interface. But such tools are often inflexible. For example, a shell allows the creation of pipelines and container systems like Docker allow sandboxing, but the two are difficult to combine [53].

With improved process creation techniques, these features can be used directly by programmers instead of by configuring stand-alone software. In this way, individual real-world applications can pick and choose the features that are useful for them, as we describe in section 4.3.

2.1 Abstraction through FD passing

In Linux (and Unix in general), a program can pass file descriptors (FDs) from the parent process to a child process by using file descriptor inheritance [34]. When a process is created, all open file descriptors are copied from the parent process to the child process. The child process can then make independent use of the file descriptors. We'll discuss this important feature further in section 3.3.1.

Most resources in Linux are managed through file descriptors, so this allows the parent process to pass a variety of resources to the child process, such as files, network connections, pipes, or other resources [8]. Since the resource is created outside the child process and passed down as an opqaue file descriptor, the precise details and type of the resource are not available to the child. This is a form of abstraction, and so we know "for free" [67] that the program running in the child process will not rely on the details of how the file descriptor was created, such as the filename that we opened or the hostname to which we connected.

Listing 2: Passing down FDs

```
db_fd = local.open("/var/db/database.db", O_RDWR)
child = local.clone()
child_fd = child.inherit_fd(db_fd)
child_fd.fcntl(F_SETFD, 0)
child.execv("/bin/fooserver",
  ["fooserver", "--database-fd", str(int(child_fd))])
```

Listing 3: Creating a concurrent processing pipeline

```
def argfd(child, fd):
  child_fd = child.inherit_fd(fd)
  child_fd.fcntl(F_SETFD, 0) # unset CLOEXEC
  return str(int(child_fd))

audio_pipe = local.pipe()
video_pipe = local.pipe()
source = local.clone()
source.execv('/bin/source', ['source',
  '--audio-out', argfd(source, audio_pipe.write),
  '--video-out', argfd(source, video_pipe.write)])
video_sink = local.clone()
video_sink.execv('/bin/video_sink', ['video_sink',
  '--video-in', argfd(video_sink, video_pipe.read)])
audio_sink = local.clone()
audio_sink.execv('/bin/audio_sink', ['audio_sink',
  '--audio-in', argfd(audio_sink, audio_pipe.read)])
```

In Listing 2, we first open a file with read-write permission in the local process. Then we create a child process, which inherits all file descriptors from its parent. We indicate that we want to operate on the child's inherited copy of the file descriptor with `child.inherit_fd`. `inherit_fd` performs no system calls, it just updates bookkeeping to return a new handle for the inherited file descriptor; we'll discuss it in more depth in section 3.3.1. Then we disable `CLOEXEC` on the child's file descriptor, which on Linux we can do by clearing the FD flags with `fcntl(fd, F_SETFD, 0)`; this ensures that the file descriptor will be usable by the new program after we call `exec` [42]. Finally, we execute a new program in the child process, passing the file descriptor number as an argument. The new program will be able to use the file descriptor we opened through that file descriptor number.

2.2 Non-shared-memory concurrency

Processes run concurrently, which enables modularity and allows exploiting the parallelism of the underlying hardware. Since processes don't share memory, they can provide a less complex parallel programming environment than shared-memory thread-based approaches [58].

In listing 3, we execute a few programs concurrently, connected by pipes. The `source` program generates two outputs, which go to `video_sink` and `audio_sink`. We first create

Listing 4: Overriding absolute path using a mount namespace

```
child = local.clone(CLONE_NEWNS)
child.mount("/home/foo/custom_foo.conf",
  "/etc/foo.conf", "", MS_BIND, "")
child.execv('/bin/fooserver', ['fooserver'])
```

two pipes in the local process, then inherit them down to several child processes using a helper function, `argfd`, which uses `inherit_fd` and `fcntl` as described in section 2.1. We call `execv` in each child process to run the desired programs, passing some file descriptors as arguments to each. The whole system of connected processes then runs concurrently.

2.3 Customization without explicit support

While ideally all resources would be passed by file descriptor, traditional Unix has a number of resources that are global, such as the process identifier space or the mount table, which cannot be changed on a per-process basis [68].

In Linux, many global resources have been made process-local through the namespaces system [31]. Like other process-local resources such as the current working directory, namespaces are typically set up at process creation time.

One key use of namespaces is to implement large-scale container systems such as Docker [31]. However, this is far from the only use. With an adaquate process creation primitive, namespaces can be used easily on a much smaller scale.

Namespaces, like all process-local resources, can be used to customize a program's behavior in ways that were not anticipated at development time by customizing the environment the process runs in [59]. Other system calls such as `chdir` and `chroot` can also be used for this, but namespaces allow new ways of customization [30].

One classic form of customization is overriding files at paths that were hardcoded into a program, without changing those files for the rest of the system. In listing 4, we create a new child process as normal, but for the first time, pass an argument to `clone`.

`CLONE_NEWNS` causes `clone` to create the process in a new mount namespace, which allows us to manipulate the filesystem tree in a way that only this process will see, using the `mount` system call [30] [40]. We call `mount`, passing `MS_BIND`, to make the file `/home/foo/custom_foo.conf` appear at the path `/etc/foo.conf`; this is known as a bind mount [44]. Then we execute some program, which, when it opens `/etc/foo.conf`, will see the contents of our `custom_foo.conf`.

2.4 Sandboxing

At process creation time, we can not only pass resources and customize the process's environment, we can also deny the process access to resources that it otherwise receives by default. This is a key part of creating a secure sandbox for potentially malicious code [47] [54] [70].

Listing 5: Unmount all and run executable via fexec

```
child = local.clone(CLONE_NEWNS)
exec_fd = child.open("/bin/foo_static", O_RDONLY)
db_fd = child.open("/var/db/database.db", O_RDWR)
child.umount("/", MNT_DETACH)
db_fd.fcntl(F_SETFD, 0)
child.fexec(exec,
  ["foo_static", "--database-fd", str(int(db_fd))])
```

In listing 5 we create a new child process, again in a new mount namespace using `CLONE_NEWNS`. Since we won't be able to open files or use `execv` in the child process after the next step, we open several files in the child up-front. Then we use `umount`, passing `MNT.DETACH` to perform a recursive unmount of the entire filesystem tree, removing it all from the view of this process.

Then, as in listing 2, we prepare to pass `db_fd` to the new program by unsetting `CLOEXC`. Unlike in listing 2, we don't need to call `inherit_fd`, since the file descriptor was opened directly from the child process. We then run the executable we opened earlier using `fexec`, which allows executing a file descriptor, and pass the database file descriptor number as an argument [41]. Note that this executable must be statically linked, or it wouldn't work in an empty filesystem namespace with no libraries to dynamically link against.

By removing the filesystem tree from the view of this process, we can run this executable with greater confidence that it won't be able to tamper with the rest of the system. A sandbox which is truly robust against malicious or compromised programs requires additional steps, but this is a substantial start [54] [70]. Such a technique can also be used when a full sandbox is not relevant, to ease reasoning about the behavior of the program being run and protect against bugs.

Even if the process needs additional resources, those can be explicitly passed down through file descriptor passing, as we do here with the database file descriptor. This allows us to approximate capability-based security [68].

2.5 Nested clone and pid namespaces

Process-local resources can also be used to control the lifetime of resources used by that process. Some Linux resources are not automatically cleaned up on process exit; a poorly coded program may allocate global resources without ensuring that they will be cleaned up later, leaving behind unused garbage on the system when it exits.

One resource which is not automatically cleaned up is processes themselves. If we run a program which itself spawns subprocesses, those subprocesses may unintentionally leak, and be left running on the system even after the original program has stopped [11].

Listing 6: Nested clone and pid namespace

```
init = local.clone(CLONE_NEWPID)
grandchild = init.clone()
grandchild.execv('/bin/fooserver', ['fooserver'])
```

We can use a pid namespace to solve this issue. The lifetime of all processes in a pid namespace is tied to the first process created in it, the init process. When the init process dies, all other processes in the pid namespace are destroyed [32].

In listing 6, we create a child process which we'll name `init`, passing `CLONE_NEWPID` to create it in a new pid namespace [40].

To create another process in the pid namespace, we clone again, this time from `init`. This is the first example in which we've cloned from one of our child processes; as normal, this gives us a capability for a new process, a grandchild, which can be used exactly like a direct child process. We `exec` in the grandchild, and we can monitor the grandchild process from `init`, just as we would monitor `init` from its parent process.

The first process in a pid namespace (usually referred to as `init`) has some special powers and responsibilities [32]. We can handle these responsibilities ourselves directly from our program, or we can continue on by `exec`ing an init daemon in `init` to handle it for us [57] [69].

In either case, when `init` dies, the pid namespace will be destroyed, and `grandchild` will be cleaned up.

2.6 Shared file descriptor tables

For our final example, we'll create a child process which shares its file descriptor table with the parent process. This can be useful for a variety of purposes, such as accessing resources in other namespaces, or accessing multiple namespaces at once.

Our example will use the Filesystem in Userspace (FUSE) Linux subsystem. FUSE [29] allows a filesystem to be implemented with a userspace program. Many interesting filesystems [37] [22] [27] have been implemented using FUSE, representing a variety of resources as files.

For improved modularity, we might want to mount and use a FUSE filesystem in our program, in a way that no other process can see it, without entering a new namespace ourselves. We can do this by creating a child process that shares its file descriptor table with the parent process.

In listing 7, we first create a child process `ns_child` in a new mount namespace with `CLONE_NEWNS`, and this time also pass `CLONE_FILES`, which causes the file descriptor table to be shared between the parent process and the child process [40]. We create another child from `ns_child` and use it to `exec` a FUSE filesystem, which will appear only in the mount namespace of `ns_child` and its descendents. Then we can

Listing 7: Shared file descriptor tables

```
ns_child = local.clone(CLONE_FILES|CLONE_NEWNS)
server_child = ns_child.clone()
server_child.execve('/bin/foofs',
  ['foofs', "--mount-at", "/"])
fd = ns_child.open("/foo/bar", O_RDONLY)
parent_fd = local.use_fd(fd)
```

open FUSE files in `ns_child` and use those files in `local`, through the shared file descriptor table. We call `use_fd` which, like `inherit_fd`, updates bookkeeping to create a new handle for the file descriptor existing in the shared fd table; we'll discuss `use_fd` in section 3.3.1.

File descriptor passing over a Unix domain socket can allow similar behavior without sharing the file descriptor table, but is substantially more complex [35]. Nevertheless, for fork-style and spawn-style process creation, file descriptor passing over a Unix domain socket is our only option to implement this kind of sharing.

3 Implementation

In this section, we'll give a brief overview of the implementation of `rsyscall`, and then focus on implementation issues specific to process creation.

`rsyscall` can be conceptually divided in two parts: the basic syscall primitive, and a language-specific library built on top. The Python language-specific library has already been demonstrated above. Such libraries only need to be able to call syscalls and explicitly specify a process in some way; they are, for the most part, agnostic to how the syscall primitive is implemented. The syscall primitive takes a syscall number, and some number of register-sized integer arguments, and arranges to call that syscall in the specified process, returning the result as a single register-sized integer.

When making a syscall in the local process, the syscall is performed normally, directly on the local thread, as one might expect.

When making a syscall in another process, `rsyscall`'s default userspace cross-process syscall primitive sends the syscall request to a userspace "syscall server" running in the target process, which performs the syscall and sends the result back. Communication to the syscall server happens over file descriptors, typically a pair of pipes. The syscall server runs on the main and only thread of the target process, and is the only userspace code running in an rsyscall-controlled child process; when not executing a syscall or writing out the result, an rsyscall-controlled child process spends all its time blocked in `read`, waiting for new syscall requests.

We use file descriptors for transport of data rather than shared memory to support straightforward integration with existing event loops; this is a key design constraint. While

we only show synchronous programs in this paper, calling a syscall in another process may block at any time, and a complex program will likely have other requests to service concurrently. Even a relatively simple program may want to monitor multiple child processes at a time. To support this, our implementation fully supports asynchronous usage with a user-provided event loop, including Python `async/await` coroutine support [63].

We have chosen to implement our cross-process syscall primitive in userspace, rather than immediately implementing this in the kernel, to aid deployability and to allow fast iteration while developing userspace code using these features. In the past, attempts to upstream kernel support for novel features without extensive userspace usage have had a poor reception in the Linux kernel community [13] [64]. We believe proving viability in userspace first will be better in the long term.

`ptrace` [46] already provides an in-kernel way to perform arbitrary actions on a target process, including system calls, but is not suitable for us. As we discuss in section 5.2, a number of systems, such as `gdb` and `strace`, use `ptrace` to implement debuggers. But multiple processes cannot use `ptrace` on a single target process at the same time; thus, if we used `ptrace` to implement `rsyscall`, such debugging tools would not work on `rsyscall`-controlled processes, which is an unacceptable limitation for a general-purpose utility.

Many syscalls either take or return pointers to memory, and require the caller to read or write that memory to provide arguments or receive results. Therefore, an `rsyscall` library needs a way to access memory in the target process.

The most simple way to access memory is for the local process to be in the same address space as the target process. This is the case most of the time; we pass `CLONE_VM` to `clone` by default in the `rsyscall` wrapper for `clone`.

Sometimes, the target process may be in another address space; for example, if the target process is at a different privilege level, we will want it to be in a different address space for security reasons [17]. There are a number of available techniques in that scenario; we choose to copy memory over a pair of pipes, again using file descriptors for the sake of easy event loop integration.

3.1 `clone`

Now that we've established the basic implementation details of `rsyscall`, we'll consider specific issues related to process creation and initialization.

Besides `clone`, `vfork` and `fork` also create processes, but they are not suitable for flexible direct-style process creation. `vfork` [49] suspends execution of the parent process while waiting for the child process to exit or call `execve`, which is immediately unsuitable. `fork` lacks many features which are restricted to `clone`. For example, with `fork`, we could not create child processes which share the parent's address space, which would complicate memory access.

The raw `clone` system call creates a new process which immediately starts executing at the next instruction after the syscall instruction, in parallel with the parent process, with its registers in generally the same state as the parent process.

`clone` lets us change the stack for the new process. We can use this to make the new process call an arbitrary function, by storing its address on the new stack. Further arguments can also be passed on the stack, with the aid of a trampoline to match C calling conventions if necessary.

We use this to create processes running the syscall server. After this, the parent process can begin to call system calls in the child process. Most system calls work as normal; the new child process can be modified freely through `chdir`, `dup2`, and other system calls. From the system calls related to process creation, only `execve` needs substantial further attention.

3.2 `exec`

`execve` is unusual and requires careful design, because when it is successful, it does not return. We need a way to determine if `execve` is successful; naively waiting for a response to the syscall request may leave us waiting forever.

There is a traditional technique used with `fork` to detect a successful `execve` using a pipe, which unfortunately won't work for us. With this technique, the parent process creates a pipe before forking, ensures both ends are marked `CLOEXEC`, performs the `fork`, closes the write end of the pipe, and `read`s the read end of the pipe. The child process either successfully calls `execve` or exits; either way, the last copy of the write end of the pipe will be closed, which causes the `read` in the parent process to return EOF. The parent can then check that the child process hasn't exited; if the child hasn't exited, then it must have successfully called exec.

This trick works with `fork`, but it's not general enough to work with `clone`. Child processes can be created with the `CLONE_FILES` flag passed to `clone`, which causes the parent process and child process to share a single fd table; we showed an example of this in section 2.6. This means that when the parent process closes the write end of the pipe, it will also be closed in the child process, and the read end of the pipe will immediately return EOF, before the child has called `execve` or exited.

Fortunately, there is an alternative solution, which does work with `CLONE_FILES`. `clone` has an argument, `ctid`, which specifies a memory address [40]. If the `CLONE_CHILD_CLEARTID` flag is set, then when the child exits or successfully calls `exec`, the kernel will set `ctid` to zero and then, crucially, perform a futex wakeup on it.

Futexes are usually used for the implementation of userspace shared-memory synchronization constructs [25], but the relevant detail for us here is that we can wait on an address until a futex wakeup is performed on that address. This means we can wait on `ctid` until the futex wakeup is

performed, and in this way receive a notification when the child process has exited or successfully called `exec`.

A process can only wait on one futex at a time; to monitor multiple futexes from a single event loop, we need to create a dedicated child process for each futex we want to wait on, and have this child process exit when the futex has a wakeup. We can then monitor these child processes from an event loop using standard techniques [48] [14].

So, we pass `ctid` whenever we call `clone`, and set up a process to wait on that futex. Then, when we call any syscall, we wait for either the syscall to return an error or the futex process to exit, whichever comes first. For other syscalls, either result will indicate that the syscall fails, but for `execve`, if the futex process exits, without the child process itself exiting, then we know that the child has successfully completed the `execve` call.

If the futex process and child process both exit, it's ambiguous whether the child process successfully called `execve`; this ambiguity is unfortunate, but it is also present in the pipe-based approach. This ambiguous situation will only happen if the child receives a fatal signal while calling `execve`, which we believe will be rare.

Some other Unix-like systems natively provide the ability to wait for a child's `execve` [36]; our implementation would be simplified, and the ambiguity eliminated, if we had this ability on Linux. One approach would be to add a new `clone` flag to opt-in to receiving `WEXECED` events through `waitid`. A new `waitid` flag alone is not sufficient, since, to integrate this feature into an event loop, it's necessary to receive `SIGCHLD` signals or readability notifications on a pidfd when the `WEXECED` event happens.

3.3 Handling file descriptors

In Linux, there are many types of resources maintained inside the kernel which are referred to by identifiers which are valid only within a certain context. For example, file descriptors are referred to by file descriptor numbers which are valid only in a certain file descriptor table; memory and memory mappings are referred to by memory addresses which are valid only in a certain address space; the same pattern is followed for a number of other resources [34]. Each process has one file descriptor table, one address space, and one of each of the other contexts. Two or more processes can share individual contexts; for example, two processes might share an address space, but not a file descriptor table.

The core issues here are the same for all such resources, so we'll focus on file descriptors. For concision and clarity, we'll abbreviate "file descriptor" as "FD" in this section.

A typical POSIX program operates within only one process, and so only needs to concern itself with one of each kind of context. To identify and operate on an FD, it can use just an FD number, which is valid within the global implicit process's FD table. That's sufficient to make syscalls involving that FD.

A program using `rsyscall`, however, operates in multiple processes, and has to deal with resources across multiple contexts. To identify an FD, it needs not just a FD number, but also some kind of identifier for an FD table. To then operate on that FD, it needs a capability for a process with that FD table, so that it can make syscalls in that process.

To manage this, we add a new concept alongside FD, FD number, and FD table: The FD handle. An FD handle is a pair of an FD number and a process capability. This both precisely identifies an FD, and allows operating on it. We say an FD handle is associated with a process if that FD handle contains that process capability.

All system calls which would otherwise return FD numbers or take FD numbers as an argument, now instead return FD handles or take FD handles as an argument. For convenience, as shown in section 2 for `fcntl`, helper methods on the FD handle object are provided for many system calls dealing with a single FD; these methods simply call the corresponding underlying system call using the process capability contained in the FD handle.

In this way, we can work with FDs without the ambiguity of dealing with raw FD numbers in a multi-process context.

3.3.1 File descriptor inheritance and table sharing

The picture is complicated by the need to support some FD behaviors which implicitly cross between processes; specifically, FD inheritance and FD table sharing.

FD inheritance is a behavior exhibited by `clone` (without `CLONE_FILES`) and by `fork`. When these system calls are called, they create a new child process with a new FD table. The new FD table contains copies of all the FDs existing in the parent process's FD table at the time of the system call, at the same FD numbers. We show examples of this in sections 2.1 and 2.2.

FD table sharing occurs when `clone` *with* `CLONE_FILES` is called; this creates a new child process that shares its FD table with its parent. As a result, either process can operate on any FD in the table, through the same FD numbers. We show an example of this in section 2.6.

These behaviors are unusual in that they operate on all open FDs at once. There are other ways to pass FDs between processes, such as `SCM_RIGHTS` [35], but those interfaces operate on individual FDs through explicit system call arguments and return values. In those interfaces, one or more FDs are explicitly passed as arguments to a system call in one process, and one or more FDs are returned from another system call in another process. This differs from FD inheritance and FD table sharing, which happen for all FDs, implicitly; these behavior must be dealt with differently to be fully supported.

`rsyscall` supports both behaviors in the same way: by allowing us to create new FD handles referring to the already-open FDs created by FD inheritance or FD table sharing. By creating new FD handles associated with new processes, we

can call syscalls in those processes on those FDs.

`rsyscall` provides two functions `inherit_fd` (used in listings 2 and 3) and `use_fd` (used in listing 7), which support FD inheritance and FD table sharing, respectively. Both functions take as arguments a process capability A and an FD handle F associated with some other process B, and return a new FD handle associated with process A.

For `inherit_fd`, the returned handle has the same FD number as F, and refers to the copy of F's FD that was inherited at process creation time. `inherit_fd` will fail if B is not A's parent process, or if F was not open at the time A was created.

For `use_fd`, the returned handle again has the same FD number as F, and refers to the *same* FD as F, through the shared FD table. `use_fd` will fail if A and B don't share the same FD table.

`rsyscall`'s support for these behaviors is intended to expose the native functionality of the Linux kernel, while still being relatively straightforward to use. In this way, we hope to ensure that direct-style process creation with `rsyscall` supports everything that the underlying Linux system call interface can support.

4 Evaluation

Our goal is a powerful approach to process creation that is easy to use correctly. Performance of our implementation is secondary, but the overhead of our implementation relative to alternative approaches must be low.

We evaluate the degree to which we meet this goal by answering the following questions:

- Does direct-style process creation on Linux allow for simpler programs than the alternatives? We compare implementations of similar functionality using both direct-style and fork-style in 4.1, and find substantial benefits for direct-style.
- Does our `rsyscall`-based direct-style process creation interface have acceptable performance overhead relative to the alternatives? We evaluate a number of microbenchmarks in 4.2, and find an acceptable level of overhead.
- Does direct-style process creation perform well in the "real world"? We discuss our positive experience with using direct-style process creation at Two Sigma in 4.3.

4.1 Ease of programming with direct-style

Does direct-style process creation on Linux allow for simpler programs than the alternatives? To answer this, we compare the direct-style examples shown in section 2 to programs with the same behavior implemented in fork-style.

Most of the examples cannot be implemented with typical Linux spawn-style interfaces such as `posix_spawn`, so we compare only to fork-style.

Name	Listing	Direct-style	Fork-style
basic	1	3	14
fds	2	6	16
pipe	3	17	49
mount	4	4	15
unmount	5	7	18
pidns	6	3	23
fuse	7	6	27

Table 2: Line counts with direct-style vs fork-style

Listing 8: Fork-style: Creating a new process, changing CWD, and execing

```
pid = os.fork()
if pid == 0:
    try:
        os.chdir("/dev")
        os.execv("/bin/cat", ["cat", "./null"])
    except OSError as e:
        ipc.send(e)
        os.exit(1)
else:
    result = ipc.recv()
    if result.is_eof:
        pass # success
    elif result.is_exception:
        raise result.exception
```

The line counts of the direct-style and fork-style implementations are listed in table 2. Direct-style consistently takes under half the lines to express the same functionality.

Listing 8 shows one of our fork-style implementations, corresponding to the direct-style implementation in listing 1.

In our fork-style implementations, we assume substantial infrastructure is available to make fork-style simpler; despite this, fork-style implementations are still significantly more verbose than direct-style. We assume the existence of a robust IPC system, with communication channels set up automatically between the parent process and the child process, with an already-defined protocol which can cover all errors from all system calls.

The main source of additional code in fork-style implementations is reporting errors back to the main program using IPC. This pattern is common in complex usage of `fork` [18] [19]. Many users of `fork` avoid complex IPC by encoding a subset of the error information into the exit code of the child process when an error is encountered during child setup. Such an approach removes the need for IPC, but still requires similar code for catching errors, encoding them into the exit code, and detecting the error in the parent process by decoding the exit code.

Direct-style does not require any extra work to report errors from the child process; errors are reported just like any other

system call. This is the main immediately visible simplification of using direct-style process creation.

It is possible to build an abstracted wrapper for fork-style process creation which abstracts away this error reporting code. We have built several such wrappers [6], but have ultimately discarded them in favor of direct-style process creation.

We found that such wrappers, besides their complexity, don't provide any fundamental improvement for the issue of communication between the concurrently executing child process and parent process. As a result, while such wrappers can remove some boilerplate in easy cases, they remain complex in difficult cases like in sections 2.5 and 2.6, which rely on a single program being able to coordinate multiple processes. We believe only direct-style process creation is able to easily express such functionality.

Direct-style also removes the need to be concerned about the state of the calling process, though this simplification is not immediately visible in a small example. The fork-style implementations may break if there are multiple threads in the calling process, and may be slow if the calling process has large amounts of memory mapped, as we'll see in section 4.2. The direct-style implementations do not have these limitations.

4.2 Microbenchmark results

4.2.1 Basic process creation

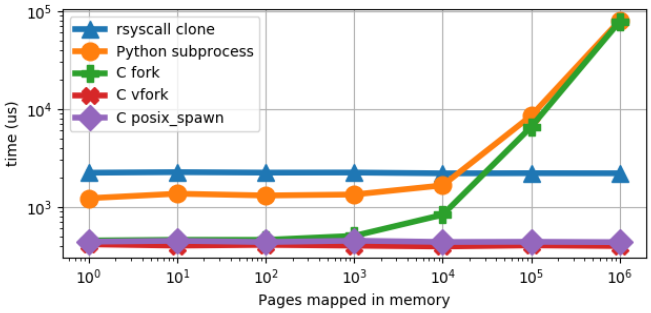

Figure 1: Python spawn-style vs direct-style performance under varying memory usage

We evaluate a simple process creation workload: create a child process, exec the `true` binary, and wait for it to exit. We implement this using Python `rsyscall` direct-style `clone`, the Python standard library's `subprocess.run` (which is implemented primarily with `fork`), `fork` from C, `vfork` from C, and `posix_spawn` from C. We run on CPython 3.7.7, glibc 2.30, Linux 4.19.93, pinned to an isolated single core on an Intel i9-9900K CPU at standard clock speed, with 60GB of RAM. We vary the amount of anonymous memory mapped in the parent process to demonstrate how each implementation scales with memory usage. The results are summarized in figure 1.

Our true baseline for performance is the Python standard library's `subprocess.run`; this takes an average of 1.4 milliseconds at low memory usage, while `rsyscall`'s `clone` takes an average of 2.2 milliseconds at any memory usage. As expected, the `fork`-based implementations scale linearly with memory usage, and the C implementations vastly outperform the Python implementations. glibc's `posix_spawn` takes an average of 440 microseconds to start a process, and `vfork` takes 400 microseconds, regardless of the memory usage of the calling process,

We perform another microbenchmark to evaluate the overhead of performing additional modifications of the child process. We call `getpid` from Python in both the child process and the parent process on the same benchmark setup. The average time per `getpid` call is 3 microseconds when called in the parent process without going through `rsyscall`, and 561 miroseconds when called through `rsyscall` in the child process. The difference, 558 microseconds, is the amount of overhead incurred for each child process setup system call performed through `rsyscall`.

These slowdowns in process creation and modification are substantial, but we found that this overhead is acceptable in practice. Process creation in Python is already slow, taking milliseconds of time, so it is not expected to be on the fast path. In that context, `rsyscall` has a reasonable cost compared to `subprocess.run`, and avoids the bad scaling of `fork` which might be unexpected by the naive programmer.

Furthermore, as we'll discuss in section 4.3, we've found that the greater expressivity of direct-style provides for programs that are more efficient on a large scale, which makes up for the performance cost in micro-benchmarks. We've also found that, for many interesting applications, the performance overhead of direct-style process creation (or Python, for that matter) is dwarfed by the execution time of the native-code programs we ultimately run.

As a result, though implementations in native code and in the kernel would likely remove most of this overhead, we have chosen to not invest effort into optimizing process creation at this micro-level, to preserve implementation simplicity; such optimization is reserved for future work.

4.2.2 Nested processes, `CLONE` flags

As demonstrated in sections 2.5 and 2.6, `rsyscall` supports nested operation. That is, we can spawn a child from one of our children; this can impact performance. `clone` also supports specifying a variety of flags to create new namespaces when creating the child, which can also impact performance.

We've benchmarked process creation using the same simple process creation workload as section 4.2, performed through `rsyscall` from a variety of parent processes: the lo-

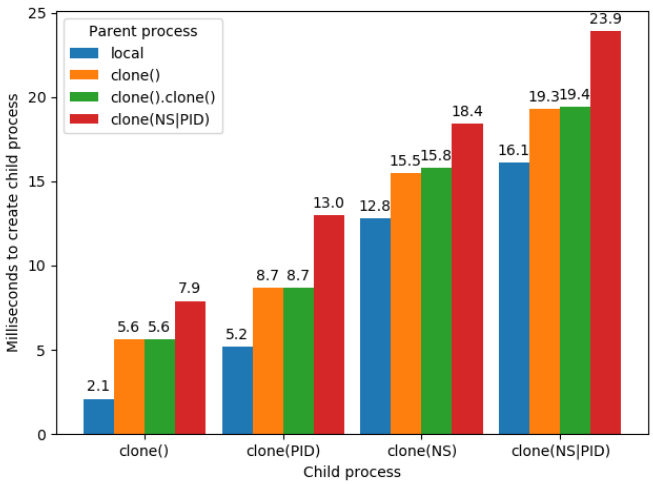

Figure 2: Time to create different processes from `rsyscall`, by variety of parent process and child process

cal process, a child (created with `clone()`), a grandchild (created with `clone().clone()`), or a child in a different namespace (created with `clone(CLONE_NEWNS|CLONE_NEWPID)`). We've also specified different combinations of flags for the child process being created in the benchmark. We don't include the Python standard library in these benchmarks as there's no way to create nested child processes or specify `CLONE` flags in the Python standard library. The results are in figure 2.

Process creation in a child has a significant performance impact, primarily because of the overhead of performing syscalls in a child. Performance of process creation in a grandchild is the same as in a child, because there is no additional overhead for syscalls in a grandchild relative to a child. Benchmarking `getpid` in a grandchild confirms that the average time per `getpid` call is still 561 microseconds on our benchmark setup.

The use of namespaces also has a significant impact, both when the parent process is in a namespace, and when the child process is created in a namespace. We perform the benchmark with no clone flags, and with the common clone flags `CLONE_NEWNS` and `CLONE_NEWPID`, first on their own and then together. This benchmark essentially measures the performance of the Linux namespaces system, not the work in this paper, but it is illustrative to compare it to the performance of `rsyscall`.

Note that the cost of creating a new child process from a parent process that is already inside a namespace (the difference between the `clone()` and `clone(NS|PID)` parent processes) is around 2 milliseconds, comparable to the cost of running with `rsyscall`. Note also that the overhead of creating a new process inside a new namespace (the difference between the `clone()` and `clone(NS|PID)` child processes) is around 15 milliseconds; subtantially greater than the overhead of `rsyscall`. As namespaces are widely used, this suggests our overhead is acceptable.

4.3 Usage in the real world

At Two Sigma we have used `rsyscall` and direct-style process creation to implement an internal distributed system deployment library, written in Python, which we'll refer to here as Toplevel. Toplevel is in use in production, and is extensively used as part of testing and development. Toplevel consists of a collection of functions and modules for each component in our system, which start up components using direct-style process creation, all from a single Python parent process.

One of the major benefits of direct-style process creation has been easy use of file descriptor inheritance. Our experience has been that with fork-style or spawn-style interfaces, file descriptor logic must ultimately be centralized, while with direct-style, file descriptors in a new process can be built up over time, simplifying the implementation. We've found that even programmers without substantial prior experience with Unix programming are able to use file descriptor passing as a normal feature of their development process with relative ease, writing code to support new programs and passing down file descriptors to those programs without issue.

Easy file descriptor passing has, in turn, allowed us to heavily use socket activation techniques to start up services in parallel, substantially speeding up system startup [60]. We are able to bootstrap connections over our internal shared memory transports over pre-created, passed-down file descriptors, and our management interfaces listen on passed-down sockets; this allows starting up services in parallel, rather than in dependency order. Benchmarking a representative example built with an old library and its equivalent built with Toplevel, the old example took an average of 221 seconds to run, mostly spent in system startup, while the new example takes an average of 16 seconds.

Easy file descriptor inheritance has also allowed us to treat file descriptors as a uniform interface in Toplevel. We pass around file descriptors freely inside Toplevel. This uniform interface allows us to flexibly swap implementations between in-process and out-of-process, allowing us to decide on a case-by-case basis whether we prefer the flexibility of an in-process Python implementation, or the performance of a native code implementation running in a child process.

We've also made use of namespaces, chiefly user namespaces and pid namespaces. For legacy applications which directly start subprocesses, we prevent process leaking with a pid namespace. This is part of a strategy of noninterference and cleanup that ensures that multiple users of Toplevel can coexist simultaneously on the same host, even if running the same system.

Toplevel also makes uses of other features of `rsyscall` which are not covered in this paper, and will be covered in future work. These include the ability to operate on remote hosts through capabilities for remote processes, and the ability to perform system calls in parallel across a pool of processes. These have significant synergy with direct-style process creation in forming an overall powerful system.

All this has allowed us to run our systems in a much more dynamic way than before. We can freely start up arbitrary subsets of our systems on-demand, spread across one or more hosts, for development, testing, or production, regardless of the configuration of that host, without worrying about interference between instances, and without a dependency on any external privileged services, such as container runtimes. This has become a key part of our development workflow.

5 Related work

5.1 Direct-style process creation

Baumann (2019) briefly suggests cross-process operations as a possible replacement for `fork` on Unix, positing many of the same advantages we find in practice [7]. Our work, though developed independently, is in many ways an elaboration on and implementation of that idea.

As we discuss in section 1, some operating systems natively have direct-style process creation. We are not aware of any other instances of direct-style process creation in a Unix-like environment.

The closest related work known to us is efforts to build Unix compatibility environments on operating systems which natively use direct-style process creation; examples include Exokernel and Fuchsia [16] [38]. Such compatibility environments could be bypassed to perform direct-style process creation using the underlying primitives, but regular Unix system calls could not be used to create new Unix processes in a direct-style way in such an environment.

5.2 Remote system calls

Many Unix-based systems have features described as "remote system calls". Most such systems do not allow a single program to manipulate multiple processes; rather, a program implicitly makes system calls in a single remote process which is distinct from the process the program's code runs in. Those systems that do allow manipulating multiple processes are generally oriented towards debugging and introspection, and are unsuitable for a general purpose system.

HTCondor [66] and Popcorn [5], among others, use system call forwarding to implement migration between hosts in a computing cluster. In these systems, processes can be live-migrated between hosts; when this occurs, the system will transparently forward IO-related system calls back to the original host.

gvisor [70] and ViewOS [21], among others, use system call interception as a means of virtualization. In these systems, system calls made in one process are intercepted and forwarded to another process, which performs the specified system call and returns the results back to the original process.

Many other systems have unusual system call invocation patterns, such as enclave systems like SCONE [3] [56] or systems for exception-less system calls like FlexSC [65]. In these systems, for a variety of reasons, a system call in one process is not evaluated by directly entering the kernel, but instead is sent to some other process or thread to be evaluated, and the results received from that process or thread.

A number of debugging or instrospection systems have the ability to perform or monitor system calls in other processes, typically using ptrace. Systems like gdb and CRIU retrieve information about the target process by forcing it to call various system calls to dump information. These systems typicaly use `ptrace`, which can be used by a single ptracer process to operate on multiple target processes at once. Unfortunately, as discussed in section 3, multiple ptracer processes cannot use `ptrace` on a single target process simultaneously, which means only one of these systems can be used at a time.

5.3 Process capabilities

Many capability-oriented operating systems, such as KeyKOS [10], seL4 [26], and others [39], have process capabilities which allow one process to operate on another process. These systems are our main inspiration for this work.

On several Unix systems, Capsicum [68] provides something called process descriptors, which is a file descriptor handle for a process. A similar feature has been recently added to Linux in the form of the pidfd API [14]. These notions of process capability are quite limited, however; pidfds and process descriptors only allow sending signals to a process or waiting for its death, rather than exercising full control over the process.

5.4 Capability-secure libc replacements

Capsicum [68] provides a set of system calls which can be used to provide a capability-secure sandbox. Latter efforts [2], including CloudABI [55] and WASI [12], developed this into a partial or full replacement for a POSIX libc. Like all other Unix libcs that we know of, these libc-replacements implicitly make system calls on the current process, rather than using an explicit process capability as `rsyscall` does.

PLASH [62] provides a spawn-style API, in the form of a shell, to launch processes in a capability-secure environment. PLASH, like all spawn-style APIs, abstracts over the native Linux environment, and is therefore limited in what kind of processes it can create.

6 Future work

6.1 Other applications of `rsyscall`

`rsyscall` was not developed solely for the purpose of this paper, and it has other uses unrelated to direct-style process creation, such as asynchronous system calls, exceptionless system calls [65], and cross-host operations, among others. We are actively exploring such applications, as well as broadening `rsyscall`'s language support.

6.2 Kernel support

`rsyscall`'s cross-process syscalls can be performed entirely in userspace, which has substantial benefits for deployability and flexibility. Nevertheless, direct support in the Linux kernel for creating a stub process and performing syscalls in the context of that process may provide efficiency benefits, as well as reducing overall complexity.

Other aspects of our implementation would also be improved by new features in the Linux kernel; one example is discussed in section 3.2.

6.3 Portability to other Unix systems

Other non-Linux systems could adopt the techniques of this paper to provide direct-style process creation. Currently, our focus is on Linux, but others may wish to explore porting these techniques to other operating systems.

6.4 Large scale open source usage

We would like to open source libraries built on top of direct-style process creation. From our experience using such libraries, discussed in section 4.3, we believe this would make it easier to develop complex systems involving processes.

7 Conclusions

We have introduced direct-style process creation on Linux for the first time, through `rsyscall`. This style of process creation is common on academic operating systems, but was previously not usable on Linux. We've provided a number of examples which demonstrate the usefulness of direct-style process creation in section 2. Our current implementation of `rsyscall` has excellent support for Python, works entirely in userspace, and is immediately deployable on today's Linux systems. `rsyscall` is open source and available from https://github.com/catern/rsyscall.

We've found that direct-style process creation is more expressive than fork-style and spawn-style, and even with an unoptimized implementation, has acceptable performance. We've used direct-style process creation successfully in production at Two Sigma. Direct-style process creation is our technique of choice for new applications involving process management, including in container runtimes and distributed systems. Future work can make direct-style process creation more efficient and decrease the complexity of its implementation.

We hope that a better process creation mechanism will help encourage more creative use of processes and the features available in Linux. Though processes are a long-standing, widespread feature, we believe there is still much to be learned about how to use processes to their full potential.

References

[1] Mark Aiken, Manuel Fahndrich, Chris Hawblitzel, Galen Hunt, and Jim Larus. Deconstructing process isolation. In *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, pages 1–10. ACM, October 2006. https://www.microsoft.com/en-us/research/publication/deconstructing-process-isolation/.

[2] Jonathan Anderson, Stanley Godfrey, and Robert NM Watson. Towards oblivious sandboxing with capsicum. *FreeBSD Journal*, 2017. https://www.semanticscholar.org/paper/Towards-oblivious-sandboxing-with-Capsicum-Anderson-Godfrey/453ba6d5377d2a9c48b75eb0a8dbdd2e44c33561.

[3] 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 (OSDI 16)*, pages 689–703, Savannah, GA, November 2016. USENIX Association. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

[4] Helge Bahmann. Re: posix_spawn is stupid as a system call. https://lwn.net/Articles/360556/.

[5] Antonio Barbalace, Binoy Ravindran, and David Katz. Popcorn: a replicated-kernel OS based on Linux. In *Proceedings of the Linux Symposium, Ottawa, Canada*, 2014. https://www.linuxsecrets.com/kdocs/ols/2014/ols2014-barbalace.pdf.

[6] Spencer Baugh. A synchronous, single-threaded interface for starting processes on Linux. https://github.com/catern/sfork.

[7] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A fork() in the road. In *17th*

Workshop on Hot Topics in Operating Systems. ACM, May 2019. https://www.microsoft.com/en-us/research/publication/a-fork-in-the-road/.

[8] D. J. Bernstein. Unix client-server program interface, 1996. https://cr.yp.to/proto/ucspi.txt.

[9] Eric W. Biederman. fuse: Allow fully unprivileged mounts, 2018. https://patchwork.kernel.org/patch/10435567/.

[10] Alan C Bomberger, Norman Hardy, A Peri, Frantz Charles, R Landau, William S Frantz, Jonathan S Shapiro, and Ann C Hardy. The keykos nanokernel architecture. In *In Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 95–112, 01 1992. https://www.researchgate.net/publication/221397234_The_KeyKOS_nanokernel_architecture.

[11] catern. The Unix process API is bad. http://catern.com/posts/fork.html.

[12] Lin Clark. Standardizing WASI: A system interface to run WebAssembly outside the web. https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/.

[13] Jonathan Corbet. Checkpoint/restart: it's complicated. https://lwn.net/Articles/414264/.

[14] Jonathan Corbet. Completing the pidfd API. https://lwn.net/Articles/794707/.

[15] Microsoft Corporation. Createprocessa function. https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa.

[16] Dawson R Engler. *The Exokernel operating system architecture*. PhD thesis, Massachusetts Institute of Technology, 1998. https://pdfs.semanticscholar.org/5f11/b6bd3f7dcb892b226ec734730081d5716c55.pdf.

[17] Rich Felker. vfork considered dangerous, 2012. https://ewontfix.com/7/.

[18] Rich Felker. Re: posix_spawn() can expose the error pipe to the spawned process, 2019. https://www.openwall.com/lists/musl/2019/07/08/3.

[19] Python Software Foundation. _posixsubprocess.c. https://github.com/python/cpython/blob/v3.8.0/Modules/_posixsubprocess.c#L781.

[20] Python Software Foundation. subprocess — subprocess management. https://docs.python.org/3/library/subprocess.html.

[21] Ludovico Gardenghi, Michael Goldweber, and Renzo Davoli. View-os: A new unifying approach against the global view assumption. In Marian Bubak, Geert Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors, *Computational Science – ICCS 2008*, pages 287–296, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-540-69384-0_34.

[22] Valient Gough. EncFS: an encrypted filesystem for FUSE. https://github.com/vgough/encfs.

[23] The Open Group. posix_spawn, posix_spawnp - spawn a process. https://pubs.opengroup.org/onlinepubs/9699919799/functions/posix_spawn.html.

[24] The Open Group. spawn.h - spawn. https://pubs.opengroup.org/onlinepubs/009695399/basedefs/spawn.h.html.

[25] Darren Hart. A futex overview and update. https://lwn.net/Articles/360699/.

[26] Gernot Heiser and Kevin Elphinstone. L4 microkernels: The lessons from 20 years of research and deployment. *ACM Trans. Comput. Syst.*, 34(1), April 2016. https://ts.data61.csiro.au/publications/nicta_full_text/8988.pdf.

[27] K Henriksson. mp3fs - FUSE-based transcoding filesystem from FLAC to MP3. https://github.com/khenriks/mp3fs.

[28] keegan. Continuations in C++ with fork, 2012. http://mainisusuallyafunction.blogspot.com/2012/02/continuations-in-c-with-fork.html.

[29] Linux kernel development community. FUSE. https://www.kernel.org/doc/html/latest/filesystems/fuse.html.

[30] Michael Kerrisk. Mount namespaces and shared subtrees. https://lwn.net/Articles/689856/.

[31] Michael Kerrisk. Namespaces in operation, part 1: namespaces overview. https://lwn.net/Articles/531114/.

[32] Michael Kerrisk. Namespaces in operation, part 3: PID namespaces. https://lwn.net/Articles/531419/.

[33] Michael Kerrisk. Namespaces in operation, part 5: User namespaces. https://lwn.net/Articles/532593/.

[34] Michael Kerrisk. *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press, 2010. http://1.droppdf.com/files/87BCs/the-linux-programming-interface.pdf.

[35] Vlad Krasnov. Know your SCM_RIGHTS. https://blog.cloudflare.com/know-your-scm_rights/.

[36] Jonathan Lemon. kqueue, kevent. https://www.freebsd.org/cgi/man.cgi?kqueue.

[37] libfuse. sshfs - a network filesystem client to connect to SSH servers. https://github.com/libfuse/sshfs.

[38] Google LLC. Fuchsia: libc. https://fuchsia.googlesource.com/docs/+/refs/changes/56/101356/4/libc.md.

[39] Google LLC. zx_process_create. https://fuchsia.dev/fuchsia-src/reference/syscalls/process_create.

[40] man-pages. clone(2). http://man7.org/linux/man-pages/man2/clone.2.html.

[41] man-pages. execveat(2). http://man7.org/linux/man-pages/man2/execveat.2.html.

[42] man-pages. fcntl(2). http://man7.org/linux/man-pages/man2/fcntl.2.html.

[43] man-pages. fork(2). http://man7.org/linux/man-pages/man2/fork.2.html.

[44] man-pages. mount(2). http://man7.org/linux/man-pages/man2/mount.2.html.

[45] man-pages. pthread_atfork(3). http://man7.org/linux/man-pages/man3/pthread_atfork.3.html.

[46] man-pages. ptrace(2). http://man7.org/linux/man-pages/man2/ptrace.2.html.

[47] man-pages. seccomp(2). http://man7.org/linux/man-pages/man2/seccomp.2.html.

[48] man-pages. signalfd(2). http://man7.org/linux/man-pages/man2/signalfd.2.html.

[49] man-pages. vfork(2). http://man7.org/linux/man-pages/man2/vfork.2.html.

[50] Matt Might. Continuations by example: Exceptions, time-traveling search, generators, threads, and coroutines. http://matt.might.net/articles/programming-with-continuations--exceptions-backtracking-search-threads-generators-coroutines/.

[51] M Miller. Robust composition: Towards a unified approach to access control and concurrency control. *Johns Hopkins: Baltimore, MD*, 2006. http://www.erights.org/talks/thesis/markm-thesis.pdf.

[52] Mark S Miller, Ka-Ping Yee, Jonathan Shapiro, et al. Capability myths demolished. Technical report, Technical Report SRL2003-02, Johns Hopkins University Systems Research, 2003. https://srl.cs.jhu.edu/pubs/SRL2003-02.pdf.

[53] moby/moby. Issue #14221: Attach stdout of one container to stdin of another (pipe-like). https://github.com/moby/moby/issues/14221.

[54] netblue30. firejail - Linux namespaces and seccomp-bpf sandbox. https://github.com/netblue30/firejail.

[55] Nuxi. CloudABI: secure and testable software for UNIX. https://cloudabi.org/.

[56] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for SGX enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 238–253, New York, NY, USA, 2017. Association for Computing Machinery. https://marksilberstein.com/wp-content/uploads/2020/02/cr-eurosys17sgx.pdf.

[57] Thomas Orozco. Tini - a tiny but valid init for containers. https://github.com/krallin/tini.

[58] John Ousterhout. Why threads are a bad idea (for most purposes). In *Presentation given at the 1996 Usenix Annual Technical Conference*, volume 5. San Diego, CA, USA, 1996. https://web.stanford.edu/~ouster/cgi-bin/papers/threads.pdf.

[59] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in Plan 9. *ACM SIGOPS Operating Systems Review*, 27(2):72–76, 1993. http://doc.cat-v.org/plan_9/4th_edition/papers/names.

[60] Lennart Poettering. Socket activation. http://0pointer.de/blog/projects/socket-activation.html.

[61] Eric Steven Raymond. *The Art of UNIX Programming: Security wrappers and Bernstein chaining*, chapter 6. Addison-Wesley, 2008. http://www.catb.org/~esr/writings/taoup/html/ch06s06.html.

[62] Mark Seaborn. PLASH: the principle of least authority shell, 2007. http://www.cs.jhu.edu/~seaborn/plash/html/.

[63] Yury Selivanov. PEP 492 – coroutines with async and await syntax. https://www.python.org/dev/peps/pep-0492/.

[64] Till Smejkal. Introduce first class virtual address spaces. https://lwn.net/Articles/717069/.

[65] Livio Baldini Soares and Michael Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *OSDI*, 2010. https://www.usenix.org/legacy/events/osdi10/tech/full_papers/Soares.pdf.

[66] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005. https://research.cs.wisc.edu/htcondor/doc/condor-practice.pdf.

[67] Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, page 347–359, New York, NY, USA, 1989. Association for Computing Machinery. https://people.mpi-sws.org/~dreyer/tor/papers/wadler.pdf.

[68] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for unix. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, page 3, USA, 2010. USENIX Association. https://www.usenix.org/legacy/event/sec10/tech/full_papers/Watson.pdf.

[69] Yelp. dumb-init: A minimal init system for Linux containers. https://github.com/Yelp/dumb-init.

[70] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The true cost of containing: A gVisor case study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association. https://www.usenix.org/conference/hotcloud19/presentation/young.