Enforced Deterministic Program Execution in the Linux Kernel

Chris Cotter
University of Texas

1 Introduction

As processes move from single to multiple cores, more and more applications are written parallel. Today, the dominant parallel programming model is nondeterministic. In this model, threads typically share an entire address space, file system, and other globally visible system managed resources like process IDs. The operating system freely schedules threads arbitrarily, and lock abstractions are not deterministic or predictable.

This model is popular, because threads can operate on shared data "in-place" instead of having to pack and unpack data structures [3]. Unfortunately, this model is error prone and has many drawbacks. Programmers spend a lot of time eliminating data races introduced by nondeterminism using unpredictable synchronization primitives. Programmers must worry about hardware side effects like in what order the processor commits memory write operations. Debugging and quality assurance are difficult without repeatability.

In general, a program is a function of both implicit and explicit inputs. We say that inputs to a program that are semantically relevant to the application are explicit and otherwise implicit. Most implicit inputs are random, arbitrary, and uncontrollable; examples are timing dependencies, quantum size, and cache size. We say a program is *deterministic* if it is a function of only its explicit inputs. Even though a program's schedule sequence may be deterministic, it may still be *arbitrary*. *sdf*

2 Background

Computer architects are under increasing pressure to produce multicore processors: the doubling of transistors every 18 months, or Moore's law, no longer implies increasing processor clock speeds. Rather, processors increasingly have more cores, with clock rates holding steady [4]. The multicore revolution is shifting program development from a sequential to parallel paradigm.

Many parallel applications are written in the conventional nondeterministic threading model. Features like shared memory state are attractive, since they facilitate easy application development. Unfortunately, writing correct parallel programs in this environment is hard, particularly due to nondeterminism.

Anyone who has written a multithreaded application using the conventional threading model is familiar with the drawbacks. Data races, both low and especially high level, introduce bugs that often must be solved with difficult to reason about lock abstractions. Nondeterministic, or even arbitrarily deterministic, scheduling of threads again introduces bugs and complicates reasoning about the program output.

Lee points out problems with the conventional threading model, and he would like to do away with it [11]. As more applications become parallel, we do not want a degradation in code quality. What we would like, then, is a new programming model that limits or eliminates the possibility of buggy writing programs. In search of a solution, Bocchino et al. make the case that all parallel programs should be written using a programming model that is "deterministic by default" [6].

In general, a program's output is a function of its inputs, both explicit and implicit. We say an input is explicit if it is semantically relevant to the program's output. We consider an input that is irrelevant to the program's intended goal, but nonetheless influences program output, to be implicit. In most cases, implicit inputs are random, arbitrary, and uncontrollable; timing dependencies, quantum size, and cache size are examples [5].

Since implicit inputs by definition are irrelevant, we would like programs to be functions of only explicit inputs. We call such programs *deterministic*. Running a deterministic program on the same input will always generate the same output, regardless of implicit inputs. Programs whose output depends on implicit inputs are considered nondeterministic.

Olszewski et al. characterize determinism into two categories: strong and weak [15]. *Strong determinism* guarantees a deterministic order of memory operations for a fixed program input and thus always provides deterministic execution. *Weak determinism* only guarantees a deterministic order of lock acquisitions. Weak determinism provides deterministic execution only when a program is written free of data races. In the presence of racy data unprotected by a lock, weakly deterministic systems fail to execute deterministically.

2.1 Motivation for Determinism

Determinism provides many benefits to application developers [5, 6, 15]. Bergan et al. suggest there are four main benefits in the following areas: debugging, fault tolerance, testing, and security.

Debugging Debugging multithreaded programs can be difficult, because often bugs are not easily reproducible, and tools such as `gdb` are not always useful for tracking down heisenbugs [14]. Finding a bug's root cause becomes easier when a program's execution can be replayed over and over. Deterministic execution naturally provides replay debugging as a benefit.

Fault tolerance Fault tolerance through replication obviously relies on the assumption that running a program multiple times will always return the same output. Determinism again provides this benefit naturally.

Testing The difficulties in testing multithreaded applications are compounded by racy nondeterministic scheduling. Developers and automated test systems must consider the exponential blow up of possible scheduling sequences. Determinism helps alleviate this problem by guaranteeing a one-to-one correspondence between input and output. For each input, there is exactly one possible logical scheduling sequence of threads. This observation eliminates the need to consider what scheduling interactions can occur, and ultimately helps developers design test strategies [5].

In addition to explicit inputs, schedule sequences are affected by implicit inputs like quantum size, data races, and memory ownership granularity. Deterministic systems like Kendo [15] and DMP [7] are especially dependent on quantum size [5]. In other words, these systems provide deterministic execution, but the scheduling sequence is still a function of implicit inputs.

On the other hand, Determinator's programming model provides something stronger than determinism: *predictability*. Whereas systems like DMP seek to "control, detect, or reproduce" data races, Determinator's programming model is *naturally deterministic*: it avoids "introducing data races or other nondeterministic behavior in the first place" [3]. With predictability, programmers can reason about every relevant aspect of a program, including scheduling. Determinism, and the stronger property of predictability, thus make testing applications easier.

Security Processes sharing a CPU or other hardware should be conscious about leaking sensitive data. A malicious thread can exploit covert timing channels to extract sensitive data from other, perhaps privileged, threads [2]. Determinism eliminates covert timing channels, since a program is purely a function of explicit inputs and cannot possible rely subtly on the timings of hardware operations.

To further motivate determinism, we consider systems that solve that above problems. So called "point solutions" solve in problems in single areas at once. Record and replay debuggers, like Leblanc et al.'s `Instant Replay` system, aid in debugging parallel programs by logging scheduling sequences and other relevant interactions in order to replay an execution sequence exactly. However, these debuggers are costly in terms of storage and performance. In general, these "point solutions...do not compose well with one another" [5]. On the other hand, determinism provides benefits in all four areas at once with a single mechanism.

2.2 Determinator

Aviram et al. set out to provide

> a parallel environment that: (a) is "deterministic by default," except when we inject nondeterminism explicitly via external inputs; (b) introduces no data races, either at the memory access level or at higher semantic levels; (c) can enforce determinism on arbitrary, compromised or malicious code for security reasons; and (d) is efcient enough to use for "normal-case" execution of deployed code, not just for instrumentation during development. [3]

To this end, they presented Determinator, a novel OS written from the ground up. For most of the remainder of this section, we will recapitulate Aviram et al.'s work and contributions; first we will discuss aspects that influenced Determinator's design. Then, we will look at the actual kernel design itself and the accompanying user library.

The primary cause of nondeterminism is data races introduced by timing dependencies. Each source of implicit nondeterminism must be accounted for in designing a deterministic programming model. We discuss them here, and describe how Determinator handles them.

Explicit Nondeterminism Often, programs rely on nondeterministic inputs such as network packets, user input, or clock time. These inputs are essential to a program being useful; therefore, a deterministic programming model must incorporate these inputs while still enforcing determinism. Determinator addresses these "semantically relevant" inputs by turning them into explicit I/O [3]. Applications have complete control over these input sources and can even log the inputs for reply debugging.

Shared program state Traditional multithread programming models provide shared state: threads using the pthreads API share the entire memory state, and Linux's file system is shared by all running programs. Data races and incorrect synchronization lead to nondeterministic execution traces and often introduce unpredictable bugs.

Determinator eliminates data races caused by shared program state by eliminating shared state altogether. Threads operate using a private workspace model and synchronize program state at explicitly defined program points. When two or more threads begin executing, each has identical private virtual memory images. Writes to memory are not visible to other threads until the threads synchronize.

Nondeterministic scheduling abstractions Traditional multithreaded synchronization abstractions are often neither deterministic nor predictable. Random hardware races determine the next thread to acquire a mutex lock, and as mentioned before this has debugging and testing implications. Even though we can record lock acquisition sequences to replay program execution or use some arbitrary device to choose a deterministic sequence, the order of acquisition is not predictable. Determinator restricts itself to only allow naturally deterministic and predictable synchronization abstractions, such as fork-join.

Globally shared namespaces Operating systems introduce nondeterminism by using namespaces that are shared by the entire system. Process IDs returned by `fork()` and files created by `mktemp()` are examples. Since these identifiers are nondeterministic, and only the resource itself, not the identifier, is semantically relevant to the application, Determinator disallows the system from choosing resource identifiers from globally shared namespaces. Instead, applications themselves choose identifiers deterministically.

2.3 The Determinator Kernel

Determinator organizes processes in a nested process model [9]. Processes cannot outlive their parents and can communicate only with their parents and children. In line with the earlier discussion of nondeterminism, the kernel "provides no file systems, writable shared memory, or other abstractions that imply globally shared state" [3]. Only "the distinguished root [process] has direct access to nondeterministic inputs" [3]. It is this root process that can control explicitly nondeterministic inputs. All other processes must communicate directly or indirectly with the root process to access I/O devices.

Kernel Interface Processes communicate with the kernel via three syscalls, `Put`, `Get`, `Ret`. Table 1 and Table 2, reproduced from [3], summarize how the syscalls work and the options available to `Put` and `Get`.

Determinator enforces a deterministic schedule by requiring programs to explicitly define synchronization points; this mechanism is described here. Since the kernel does not manage any global namespaces, user programs specify a child process ID parameter to `Put` and `Get`. The first `Put` syscall with a previously unused child ID creates a new child process. `Put` calls can start a child's execution, and the child will continue to execute until it invokes `Ret` or generates a processor exception (e.g. divide by zero). `Put` and `Get` calls block until the child process stops.

Process state is composed of register state and its entire virtual memory. The Regs option copies register state from a parent to child or vice versa. Determinator provides more sophisticated virtual memory options: the Zero option zeros a virtual memory region in a child; the Copy option copies virtual memory regions between a parent and its child; the snapshot-merge mechanism is similar to Copy, but more complicated.

Snap copies the calling process's entire virtual memory state into the specified child. Invoking `Get` with Merge copies bytes from the child that have changed since the previous Snap invocation into the parent. Bytes that changed in the parent but not the child are not copied. Bytes that changed in both the parent and child generate a *merge conflict*. The kernel implements Merge efficiently by examining page table entires.

Aviram et al. conclude their discussion of Determinator's kernel by mentioning the three syscall "primitives reduce to blocking, one-to-one message channels, making the space hierarchy a deterministic Kahn network" [10].

2.4 Determinator's User Library

The Determinator kernel alone is enough to enforce deterministic program execution; however, Aviram et al. provide a high-level user library to make writing deterministic programs easier and more natural. In this section, we will go over the five main areas discussed by Aviram et al. in their "Emulating High-Level Abstractions" section: process API, file system, I/O, shared memory multithreading, and legacy thread APIs [3].

Process API Determinator provides an interface similar to that of `fork/wait`. All of these functions are implemented in user space instead of kernel space. To do a `fork()`, the parent does a single `dput()` to copy its register and memory into a child. The user library must manage a list of "free" process ID numbers, since the user must specify the child process ID. A `waitpid()` works by entering a loop querying the status of the child; if the child needs more input to continue running (through a mechanism described below), it sets its status appropriately and issues a `dret()`. The parent gives the child more input and sets it in motion again. Once the child finishes executing, it marks itself done and issues a `dret()`.

Call	Interacts with	Description
Put	Child	Copy register state and/or virtual memory range into child, and optionally start child executing.
Get	Child	Copy register state, virtual memory range, and/or changes since the last snapshot out of a child.
Ret	Parent	Stop and wait for parent to issue a Get or Put. Processor traps also cause implicit Ret.

Table 1: System calls comprising Determinators kernel API.

Put	Get	Option	Description
X	X	Regs	PUT/GET childs register state.
X	X	Copy	Copy memory to/from child.
X	X	Zero	Zero-fill virtual memory range.
X		Snap	Snapshot childs virtual memory.
X		Start	Start child space executing.
	X	Merge	Merge childs changes into parent.
X	X	Perm	Set memory access permissions.
X	X	Tree	Copy (grand)child subtree.

Table 2: Options/arguments to the Put and Get calls.

File System Since processes can only access their register set and memory, Determinator provides an in-memory file system. Each process has a private copy of the system's file system. A `fork()` copies the parent's file system state into the child. The parent and child work on private copies of the file system.

At synchronization points, the parent reconciles the changes using file versioning techniques [16]. Two files may not be concurrently modified; such cases lead to a reconciliation conflict. The parent and child may, however, perform *append-only* changes to the same file. The file is reconciled by appending the child's additions to the end of the parent's file, and vice-versa.

The file system is limited, however. The total file system size is limited by the process's address space; on 32-bit systems, this is a serious limitation. Since the file system resides in virtual memory, buggy programs can write to the memory where the file system resides, corrupting the file system. Lastly, the file system only supports up to 256 files, each with a max of 4MB in size [2].

I/O Since Deterministic processes have no access to external I/O, Determinator emulates I/O as a special case of the file system. Library functions like `getchar()` and `printf()` read and write from special files `stdin` and `stdout`, respectively.

A `printf()` appends output to `stdout`. At program synchronization points, the file system is merged. `stdout` output is forwarded to the parent and eventually reaches the root process where the root can actually write the output to the system's I/O device. A program does a `read()` to obtain the next unread character(s) in `stdin`. If the file is out of unread characters, `read()` does a `dret()` to ask for more input from the parent.

Since the file system supports "append-only" conflicts, the above strategy works for handling I/O. As all processes reconcile their file systems, each process will see all other process's `printf()`s.

Legacy Multithreading APIs Determinator can emulate shared memory multithreading and other legacy multithreaded APIs like pthreads. However, we will not discuss either here, since we do not use these techniques in deterministic Linux. However, we note that since Determinator emulates these legacy thread APIs using its three syscall interface, deterministic Linux could support these APIs.

2.5 Deterministic Linux

With Determinator presented, we can now motivate a deterministic Linux. Determinator was written from scratch in an academic environment with determinism as the main OS design goal. In some sense, Determinator is not a *real* operating system. The potential uptake outside the academic world is minimal. On the other hand, Linux is a mature and widely deployed nondeterministic operating system. Linux is installed on millions of systems including desktop computers, embedded systems, and mobile devices. In other words, Linux is a real operating system used in the real world, and by adding determinism to Linux, we are able to take advantage of its widespread use and adoption; the potential userbase for a deterministic Linux is much greater than that of Determinator.

Adding an inherently deterministic and predictable programming model to Linux is a huge step in attempting to influencing how developers write the parallel applications of the future.

3 Overview

We begin the discussion of adding determinism to Linux by discussing overall design goals of the project. Next we look at the challenges of making nondeterministic Linux deterministic. For the rest of this thesis, we distinguish between *legacy* Linux (the unmodified Linux kernel) and *deterministic* Linux.

3.1 Design Goals

We wish to make 64-bit Linux deterministic, and in doing we will present an interface similar to that of Determinator. We also would like to run legacy Linux applications alongside deterministic applications, for this is one of the motivating factors applying Determinator's design to Linux. We want to run legacy applications without modification, but we won't make any attempt to force legacy applications to run deterministically. Legacy applications will run in a legacy nondeterministic mode. In order to take advantage of determinism in Linux, legacy programs must be rewritten using the new operating system abstractions.

We would also like to write a user level C library with familiar abstractions such as fork-join and an in memory file system based on those of Determinator. In some cases, we improve upon Determinator's user library, especially the limitations of the in memory file system [2, 3].

We do not wish to apply all of Determinator's features to a deterministic Linux. Determinator supports deterministic distributed cluster computing by extending its nested process model to a cluster of nodes. Determinator also supports a "tree" copy operation for `Put` and `Get`. Lastly, Determinator allows threads to place an instruction limit on children threads. We have no intention of supporting these features, but we note this limitation does not detract from making Linux deterministic.

Since the primary goal is to make Linux deterministic, we may decide to limit or ignore features of the Linux kernel internals. For example, Linux supports huge pages of memory alongside "normal" 4-KB pages. Since this is an internal optimization that is hidden from user applications and for reasons of implementation complexity, we may not allow deterministic applications to take advantage of certain kernel features. We would like to keep all existing functionality available to applications running in legacy mode, however.

Moreover, some useful features may be unavailable to deterministic processes. For example, shared dynamic libraries require nontrivial support from the Linux kernel and standard C library. There is nothing limiting us from devising ways to support features like this, but we feel it is out of scope of our primary goal.

3.2 Challenges

We have already discussed the four categories of nondeterminism identified by Aviram et al; these obvservations are general enough that they also apply in making Linux deterministic. In adapting Determinator's design to Linux, however, we must address the following additional issues.

Inherent nondeterminism in Linux In order to run legacy Linux applications, we cannot enforce that all but a single root process operate in deterministic mode; this design aspect must be reexamined to allow legacy and deterministic applications to run side-by-side. Furthermore, Linux's process model allows reparenting and for children to outlive parents, directly opposed to Determinator's nested process model.

Linux's threading model is inherently nondeterministic and provides many additional sources of nondeterminism than those already addressed by the above discusion: Linux supports signals and System V IPC. To address most sources of nondeterminism (implicit and explicit), Determinator's designers simply did not add support for these features, since Determinator was written from scratch. On the other hand, Linux already provides extensive support for nondeterministic features (e.g. the `gettimeofday()` syscall).

Memory subsystem Linux supports a wide range of virtual memory features including memory mapped files, huge pages, and swapping to disk; all of these features are layered on top of an abstraction for supporting memory management units for a wide range of processor types. Compared to Determinator, Linux's memory subsystem uses much more complicated abstractions to support these features. Memory operations (Zero, Copy, and Snap/Merge) are central to Determinator's design, so understanding and overcoming this complicated system is essential to implementing determinism in Linux.

Standard C Library Many Linux applications written in C use the standard C library. This library in large part functions as a wrapper around legacy Linux syscalls, and thus is highly nondeterministic. Whereas some functions, such as `strlen()` might not use nondeterministic syscalls, many other functions do use nondeterministic syscalls (e.g. `printf()`). Thus, deterministic programs may be forced to use a completely different library. Moreover, the libraries in Linux are often linked dynamically with shared libraries, but Determinator does not provide any native kernel support for dynamic linking. We may lose the ability to dynamically link shared libraries.

3.3 High level approach

To address concerns about Linux's more flexible process model, we present a *hybrid process model*. A Linux process invokes a syscall to become a *master* process, akin to Determinator's lone root process. This master process has full access to the legacy Linux kernel API, with some minor restrictions noted below. Master processes then create *deterministic* children. We call this master process and its entire subtree a *deterministic process group*. Within this

process group, processes abide by Determinator's nesting rules (e.g. children cannot outlive parents). A deterministic process's death automatically triggers reaping that process's subtree.

Legacy Linux applications run alongside deterministic applications with absolutely no kernel restrictions. In some sense, each deterministic application resembles an entire Determinator "virtual machine" of sorts.

We also add three new syscalls, `dput()`, `dget()`, and `dret()` and restrict deterministic processes to only use the new syscalls. These syscalls function exactly as their Determinator counterparts, excepting cluster support, the copy (grand)child subtree option, and instruction count limits. By restricting deterministic processes to these three syscalls, we can nearly remove all sources of nondeterminism; we only have to modify the kernel to ignore all signals sent to deterministic processes, and thus we have effectively blocked all sources of nondeterminism.

At the expense of predictability, but without harming determinism, master processes in deterministic Linux are not restricted to blocking `dput()` and `dget()`. A master can invoke these syscalls with a special flag to poll whether or not the child process has reached a synchronization point yet. We have to chosen to allow this, since some parallel applications fail to exploit optimal concurrency in Determinator's design, like running a `make -j2` [3]. Applications that use the non-blocking syscalls can still log schedule sequences to reproduce program output in a deterministic fashion.

We also allow signals to reach master processes, and master processes can specify a set of signals that can interrupt a blocking `dput()` or `dget()`. Allowing signals for master processes again introduces nondeterminism, but we note that it is explicit and controllable. We also note the usefulness of signals: terminal operators can send a `SIGINT` to kill an application immediately.

Once this kernel work is done, we begin work on a C user library. We won't use the standard C library with deterministic processes, since many library calls invoke disallowed legacy syscalls. The common use case of multithreaded applications is to `fork()` a child with a copy of the parent's virtual memory image, thus giving the child access to the same library API as the parent. This is undesirable for our system, since this would automatically let deterministic children use the standard C library. To avoid this and namespace problems (we want deterministic processes to use familiar function names like `printf()`), we require master and deterministic processes must use our new deterministic library. Unfortunately, many functions must be rewritten (e.g. `sprintf()`, `strlen()`).

To increase the usefulness of the system, we provide an in memory file system just as Determinator does. Whereas Determinator's file system used fixed file size [2], our file system design is similar to that of the BSD Fast File System [13]. The file system is divided into 4096-byte *blocks*. The first block is a *superblock* containing metadata about the file system. A region of fixed size following the superblock is reserved for *inodes* and a bitmap for managing free blocks. The rest of the blocks are data blocks. In addition to direct block pointers, inodes support a singly and doubly indirect block of data block pointers. Directories are files containing a list of files within the directory.

We also note that since master processes have access to the system file system, our user library can save the in memory file system to permanent storage if so desired. Thus, our file system improves upon that of Determinator by supporting hard linking, supporting larger file sizes, and being more flexible in managing underlying resources (inodes, blocks).

4 Implementation

With a high level design in mind, we now discuss the final version of the implementation. We started by forking Linux from a `git` repository, and worked on x86_64 Linux 3.0. We developed and tested incrementally on an 8-core machine with 8 gigabyets of RAM running Arch Linux.

This section is divided into two logical parts. We first discuss the kernel implementation, then the user library and in memory file system.

4.1 Process synchronization

The first step was adding the three new syscalls: `dput()`, `dget()`, and `dret()`; iteratively, we added the various features to the syscall implementations. Our initial focus was adding process creation functionality. `dput()` relies heavily on existing `fork()` kernel code to create new processes. We also used existing `do_exit()` code to delete processes and enforce that a deterministic processes death implies the death of its process subtree. We block all external signals generated by user applications, but allow signals generated by the kernel itself; it is through this mechanism that exceptions, such as divide-by-zero faults that generate a `SIGFPE`, cause an implicit `dret()`.

We augment Linux's `task_struct` process structure with a *deterministic PID* and synchronization primitives. `dput()` and `dget()` use these synchronization primitives to correctly synchronize deterministic process communication within the hybrid process model.

We also placed some restrictions on what a master process can do. Processes become the master of a deterministic process group by invoking `dput()` with a special parameter. The kernel makes sure that process is not the parent of any other nondeterministic process, and the process isn't using any unsupported virtual memory features,

like "kernel samepage merging" [1] or huge pages. Once a process becomes a master, it can no longer `fork()`

Master processes can use `dput()` to specify a set of signals to block while in a blocking `dput()` or `dget()` (i.e. waiting for a child to stop). These syscalls will ignore signals specified in the block set sent to the master until the child issues a `dret()`. This can be useful when a console operation wishes to kill an application with a `SIGINT`, but does not want other signals to interrupt the master process.

The last feature relating strictly to process organization is register state copying. The initial `dput()` call that creates a child automatically copies register state, since we effectively delegate work to `fork()`. Subsequent calls to `dput()` and `dget()` pass general purpose register state structure pointer to set or get a child's register set.

4.2 Memory operations

As mentioned in the challenges section, Linux's memory subsystem is very complex and feature filled. Before discussing how we implemented the memory operations, we will give some technical background on Linux's memory subsystem. Our implementation reused many existing functionality, so this discussion will be useful.

Background A process's entire virtual memory is managed by a `mm_struct`. `mm_structs` maintain a list of contiguously mapped memory regions with identical permission bits (e.g. read/write/execute); these regions are stored as `vm_area_structs`. Anonymous memory (memory not backed by a file) is mapped to a read only global zero page, and new pages are allocated *on demand* [12]. In order to swap a single physical page of memory to disk, Linux requires all mappings of the page have the same offset from the start of the enclosing `vm_area_struct`. This requirement makes swapping space and time efficient, but it places unfortunate restrictions on how aggressively we can apply copy-on-write.

Linux provides `dup_mm()` to copy a virtual memory image of a process. This is used by `fork()`. Of course, Linux takes advantage of copy-on-write. `dup_mm()` copies each `vm_area_struct` of the source by invoking `copy_page_range()`. This function walks Linux's four level page table structure to perform copy-on-write by actually changing page table entries and marking the pages as read only.

Linux's memory subsystem does lack one important generality that is essential to the core of Determinator's memory operations: in Linux, most virtual memory kernel functions act on behalf of the calling process (inside a syscall, the calling process is accessed via the C macro `current`) instead of operating on *any* process's virtual memory. For example, `do_mmap()`, which maps files into memory and creates anonymous memory regions, will only perform work on `current`.

Zero Since `dput()` and `dget()` operate on a child process, not the calling process, the first step was to generalize Linux's memory subsystem. Functions like `do_mmap()` were enhanced to take an extra argument specifying the target process's virtual memory structure.

Implementing the Zero operation was thus very simple. do_mmap() maps anonymous memory to a zeroed out region automatically, so to perform a Zero operation, the kernel unmaps then remaps the region in question. Most of the memory subsystem deals in page aligned regions, so the kernel needs to handle non page aligned begin and end regions with what amounts to a `memset()`.

Copy The copy operation was more complicated. At a high level, Copy takes an arbitrary virtual memory region from the source and copies it to the destination virtual memory, with an optional offset in the destination. To be efficient, we only allow page aligned offsets so that we can take advantage of copy-on-write.

We generalized `copy_page_range()` to map pages copy-on-write with a destination offset. The Copy operation works by unmapping the specified region in the destination, then invoking a `copy_page_range()`. To satisfy the swapper subsystem's requirement about how physical pages can be mapped, care must be taken to ensure the source and destination have their `vm_area_structs` sharing start and end boundaries (with respect to the destination offset). This can be accomplished with a helper function, `split_vma`. Finally, as before, we must handle non page aligned begin and end regions with a `memcpy()`.

Snap/Merge The Snap/Merge combination is the most complicated feature, and unfortunately we were not able to reused as much existing code as with Zero and Copy. Performing a Snapshot is relatively easy. We destroy the target's `mm_struct`, then mimic `fork()`: we make a copy of the source's `mm_struct` and attach it to the destination. This effectively copies the source's virtual memory image into the destination. We also make a second `mm_struct` copy for use as a reference virtual memory image later. Using `dup_mm()` only has to create a copy of the `mm_struct` and page tables; pages used by the process are only copied when written to, so this method of creating a reference snapshot is space efficient.

Upon a Merge request, the kernel first must ensure `vm_area_structs` are aligned, just as for Copy. We then iterate over `vm_area_structs` and walk the page table hierarchy. Instead of doing a naive byte by byte comparison, we check page table entries to quickly determine if two pages have diverged since the snapshot; if a page was written to,

a new page would have been allocated via copy-on-write, thus indicating the kernel must do a byte by byte comparison. Pages that have changed only in the source are copied via copy-on-write to the destination, when a byte by byte combination must be used, changed bytes are also copied over. Writes by the source and destination to the same byte location generate an exception. The child can no longer run, and the parent must acknowledge that exception by killing the child. As usual, we handle non page aligned start and end regions manually by doing a direct byte comparison.

4.3 User library

We require that deterministic Linux applications use a custom user library. We model the library design and API off of the standard C library. Many simple functions must be rewritten, like `strlen()`, and indeed we borrowed many header and implementation source files from the instructional JOS operating system [8].

Designing and implementing what amounts to be a replacement for the standard C library is no easy task, and indeed a fully functional library merits an entirely separate discussion. Thus, we did not set out with a specific plan or set of functionality to implement. Much of the library was constructed in a reactive manner. New functionality was added only when necessary for building applications.

Aviram et al. devote an entire section, "Emulating High-Level Abstractions" discussing how to implement a traditional Unix API (`fork()`, `open()`, etc.) [3]. We do not have any new insight in this area, so we will limit our discussion here to Linux-specific details as they apply to writing the user library.

Before executing `main()`, the library detects whether the executing process is the master or deterministic and sets up internal variables. The in-memory file system is initiated (described in detail below), and special `stdin` and `stdout` files are created so that processes can emulate functions like `getchar()`. When returning from main, or performing an `exit()`, the file system is cleaned up and the process signals to the parent that it has finished by issuing a `dret()` and setting its exit status code. The parent must acknowledge the child's death before proceeding.

Many functions have dual roles depending on whether the executing process is the master or deterministic. For example, since master processes have direct access to nondeterministic resources like file I/O, functions like `printf()` write directly to the system's `stdout` via the `write()` syscall. On the other hand, when deterministic processes invoke `printf()`, the output is buffered in the special `stdout` file in the in-memory file system. At synchronization points, the file system forwards this `stdout` to the parent; eventually, the output reaches the master space and is directed to the system's `stdout`.

4.4 File System

The in memory file system for deterministic Linux has significant improvements over that of Determinator. Whereas Determinator uses a fixed file size and all files are mapped to a known location in memory, we chose to implement the more general BSD Fast File System design.

The file system is divided into 4096-byte *blocks*. The first block is a *superblock* containing metatata about the file system. A region of fixed size following the superblock is reserved for *inodes* and a bitmap for managing which blocks are used. The rest of the blocks are data blocks. Each inode represents a traditional Unix file object and contains ten direct block pointers and a singly and doubly indirect block pointer. A file may be up to 1GB on 64-bit systems.

Deterministic applications can also take advantage of the master process's access to the system file system. When an application starts up, it can read an in-memory file system image from permanent storage. When the application finishes, it can save the in-memory image back to permanent storage for use later.

5 Conclusion

Remind reader about the contributions of the proposed work, and what the proposed work will actually look like.

References

[1] A. Arcangeli, I. Eidus, and C. Wright. Increasing memory density by using ksm. In *Proceedings of the linux symposium*, pages 19–28, 2009.

[2] A. Aviram, S. Hu, B. Ford, and R. Gummadi. Determinating timing channels in compute clouds. In *CCSW*, 2010.

[3] A. Aviram, S. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *OSDI*, 2012.

[4] C. BANDWIDTH and T. USABLE. Multicore cpus for the masses. 2005.

[5] T. Bergan, J. Devietti, N. Hunt, and L. Ceze. The deterministic execution hammer: How well does it actually pound nails? In *WoDet*, 2011.

[6] R. Bocchino, V. Adve, S. Adve, and M. Snir. Parallel programming must be deterministic by default. In *First USENIX workshop on hot topics in parallelism (HotPar)*, 2009.

[7] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: Deterministic shared memory multiprocessing. In *ASPLOS*, 2009.

[8] F. K. et al. Operating system engineering.

[9] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *OSDI*, 1996.

[10] G. Kahn. The semantics of a simple language for parallel programming. 1974.

[11] E. Lee et al. The problem with threads. *Computer*, 39(5):33–42, 2006.

[12] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[13] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A fast file system for unix. *ACM Transactions on Computer Systems (TOCS)*, 2(3):181–197, 1984.

[14] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[15] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ACM Sigplan Notices*, volume 44, pages 97–108. ACM, 2009.

[16] D. Parker Jr, G. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *Software Engineering, IEEE Transactions on*, (3):240–247, 1983.