Enforced Deterministic Program Execution in the Linux Kernel

Chris Cotter
University of Texas

Abstract

This is Determinator in Linux. Hope you read it all.

1 Introduction

As processors 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. Operating systems schedule 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 [3]. Unfortunately, this model is error prone and has many drawbacks. Programmers spend a lot of time eliminating data races introduced by nondeterminism by using unpredictable synchronization primitives. Programmers must worry about hardware side effects like commit ordering of memory write operations. Without repeatability, debugging and ensuring software quality assurance become difficult.

To overcome the challenges of nondeterminism, Aviram et al. presented a deterministic operating system called Determinator [3]. Programs are written using a novel parallel programming mode that is "naturally and pervasively deterministic" and in fact predictable. Through a three syscall approach, the Determinator microkernel can run programs written in general-purpose languages like C on conventional hardware. Determinator also constributes a high-level library with familiar abstractions and an in-memory file system. Evaluations of Determinator against Linux show that such a model can be implemented to run coarse-grained parallel applications efficiently with little overhead, but fine-grained parallel applications have unacceptable overhead.

Whereas Determinator was written from scratch in an academic environment, we would like a deterministic environment in a more widely deployed operating system. The research presented in this thesis is based on applying Determinator's model to Linux. Before we describe the work of this thesis, we will present background information on determinism and argue why we chose to base our work on Determinator.

In general, a program is a function of both implicit and explicit inputs. We call semantically relevant inputs *explicit* and otherwise *implicit*. Most implicit inputs are random, arbitrary, and uncontrollable; examples are timing dependencies, quantum size, and cache size [5]. [1] 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*. The next thread to acquire a mutex lock could depend on hardware implementation or instruction counting [14]. If we consider a program as a relation in the mathematical sense, determinism guarantees the relation is a one-to-one function, but we may not know how to compute the function without actually running the program. If we can determine a program's output from examining the program code, we say the program is *predictable*. For the rest of this thesis, we modify our definition of determinism to have the property of predictability and make the distinction explicit only when required. [2]

There are many factors to consider in designing a deterministic environment. Some systems provide determinism through special hardware, but these systems have limited uptake, We want to write programs on widely available CPUs. Some environments provide determinism through special programming languages, but again these have limited uptake. We want to write programs in conventional general-purpose programming languages. Enforcing deterministic scheduling through user space libraries has the drawback that programming bugs or malicious code can interfere with the scheduler. Instead, we want a system that cannot be compromised by misbehaved code. With system enforced deterministic, the burden shifts from the programmer to the system. Lastly, we consider the two variations of determinism: strong and weak [14]. *Strong* determinism ensures a deterministic order of all memory accesses, and every program is guaranteed to run deterministically. *Weak* determinism ensures a deterministic sequence of lock acquisitions, but the existence of data races can lead to non deterministic program execution.

For a deterministic Linux, we prefer strong determin-

[1] The reader should note the subtle distinction in classifying inputs as explicit or implicit and deterministic or nondeterministic. Time of day is often an explicit nondeterministic input, since it is semantically relevant to many applications. For a given CPU, cache size is an implicit deterministic input.

[2] The usual definition of determinism does not include the stronger notion of predictability, but it will be useful for our purposes to include it.

ism that is predictable and enforced by the system. Our choice is affirmed by Bochino et al., who claim that all parallel programming environments must be "deterministic by default" [6].

This thesis is about adapting Determinator's operating system design and programming model to Linux. We implement Determinator's three syscall interface inside the Linux kernel, then we write a high-level user library based on that of Determinator. We can run legacy nondeterministic programs alongside deterministic programs. We can write new parallel applications using familiar abstractions. Legacy applications must be rewritten using the new API.

We make the following contributions:

- A presentation of a deterministic Linux kernel heavily based on that of the Determinator kernel. This is the first known adaptation of Determinator's kernel design in a widely deployed operating system.
- A deterministic high level user library for use by application programmers. This library is motivated by Determinator's user library and the usefulness of developing programs using a familiar API like the standard C library.
- An improvement of Determinator's in-memory file system. Our file system is modeled off of the BSD Fast File System [11], and it provides persistence.
- We evaluate the performance of deterministic Linux against traditional nondeterministic parallel Linux. We also demonstrate a case study of the benefits of determinism.

Aviram et al. were motivated by meeting the "software development, debugging, and security challenges" of writing future parallel applications [3]. Determinator was a huge step towards this. Unfortunately, Determinator has limited uptake outside the academic community. Linux is a mature and widely deployed operating system available for desktops, servers, mobile, and embedded systems. Adding determinism alongside nondeterministic Linux will be a huge next step. If we are lucky, we might be able to influence how future parallel applications are written.

2 Background

With an understanding of the goal of this thesis, we will now discuss the benefits of deterministic execution. Then, we will present the Determinator kernel and user library design.

2.1 Benefits of determinism

Determinism provides many benefits to application developers [5, 6, 14]. 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. Tools such as `gdb` are not always useful for tracking down heisenbugs [12]. Finding a bug's root cause becomes easier when a program's execution can be replayed.

Fault-tolerance Fault tolerance through replication relies on the assumption that running a program multiple times will always return the same output. Repeatability is a natural benefit of determinism.

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, since 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].

Since schedule sequences may still be arbitrarily deterministic, developers may still have a hard time designing test suites. Predictable programming models like Determinator allow developers to reason about code beforehand to design a more sophisticated testing strategy.

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 possibly rely subtly on the timings of hardware operations.

We would also like to point out the importance of repeatability in simulators. Physicists rely on repeatability of simulated experiments to verify results, and video game developers often benefit from repeatable physics engines.

Whereas individual tools like record and replay debuggers aid programmers in single areas, these so called "point solutions…do not compose well with one another" [5]. On the other hand, determinism provides benefits in all four areas with a single mechanism without any overhead besides that inherent in the deterministic environment itself.

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 efficient 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 semantically relevant nondeterministic inputs such as network packets, user input, or clock time. A deterministic programming model must incorporate these inputs while still enforcing determinism. Determinator addresses these inputs by turning them into explicit I/O [3]. Applications have complete control over these input sources and can log the inputs for reply debugging.

Shared program state Traditional multithread programming models provide shared state: processes share the system's file system and threads using the pthreads API share the entire memory state. Data races and buggy application of synchronization primitives lead to nondeterministic execution traces and 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 [3]. 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 explicitly 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 in section 2.1, 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 [13]. In the fork-join paradigm, threads *fork* children threads to perform some computation. The original thread gathers the results by doing a *join* by blocking until the child thread completes.

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. For example, when performing a `fork()`, the user program must specify the child ID as an argument instead of the operating system returning the process ID of the newly created child.

2.3 The Determinator kernel

Determinator organizes processes in a nested process model [3, 8]. Processes cannot outlive their parents and can communicate only with their parents and children. The kernel "provides no file systems, writable shared memory, or other abstractions that imply globally shared state". Only "the distinguished root [process] has direct access to nondeterministic inputs" [3]. It is this root process that can control explicitly nondeterministic inputs like network packets. 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. 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 (via the `Start` flag), and the child will continue to execute until it issues a `Ret`. Processor exceptions (e.g. divide by zero) generate an implicit `Ret` that must be acknowledge by the parent. Both `Put` and `Get` calls block until the child process stops.

Process state is composed of its register state and virtual memory image. The `Regs` option copies register state from a parent to child or vice versa. The `Zero` option zeros a virtual memory region in a child. The `Copy` option copies virtual memory into or out of a child.

Determinator provides a more sophisticated memory utility: snapshot-merge. `Snap` copies the calling process's entire virtual memory state into the specified child. In-

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. [3]

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. [3]

voking `Get` with the `Merge` parameter performs a three-way diff. The kernel 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* exception. The kernel implements `Merge` efficiently by examining page table entires. This snapshot-merge mechanism is useful for providing applications with a fork-join model, where a process forks many children to perform some computation and simply `Merge` the results back into itself.

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" [3, 9].

2.4 Determinator's user library

The Determinator kernel alone is enough to enforce deterministic program execution; to make writing deterministic programs more natural, however, Aviram et al. provide a high-level user library that wraps around the three syscall interface. 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/exec/wait`. All of these functions are implemented in user space instead of kernel space. To `fork()` a child process, the parent invokes `dput()` to copy its register and memory state into a new child. The user library must manage a list of "free" process ID numbers, because the system itself does not manage process IDs. `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()`. The parent collects the child's status and kills the process.

`exec()` works by forking a child process and loading the new program the new program's memory image into the child. This child is never actually run. Instead, `exec()` enters a trampoline code segment that does a `Get` to copy the new program into the existing process. The trampoline code is mapped at identical locations in both processes so that after executing the `Get`, the process begins executing valid code.

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 and merge their changes at synchronization points using file versioning techniques [15]. 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 has limitations compared to traditional file systems. 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, Determinator's implementation of 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`. When a parent merges its file system with a child, `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()` issues a `dret()` to ask for more input from the parent.

Since the file system supports "append-only" conflicts, the above strategy works well 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 very well be extended to support these APIs. The reader is referred to Aviram et al.'s sections 4.4 and 4.5 [3].

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 and Non-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 sup-

port 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 the `dput()` syscall (introduced below) 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 the master can control the signals and write them to a log file for replay. 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 [11]. 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 implementation details. We started by forking Linux from the GitHub `git` mirror repository and worked on x86_64 Linux 3.0. We developed and tested incrementally on an 8-core machine with 8BG 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.

The first step was adding the three new syscalls: `dput()`, `dget()`, and `dret()`. Our initial focus was adding process management related functionality, then we added the `Zero`, `Copy`, and `Snapshot/Merge` operations in that order.

4.1 Process synchronization

The first step was adding the three new syscalls: `dput()`, `dget()`, and `dret()`. Our initial focus was process creation and related functionality. `dput()` relies heavily on existing `do_fork()` and `do_exit()` kernel code. We added logic to enforce the requirement that deterministic processes cannot outlive their parents. We blocked all external signals generated by user applications, but allowed kernel-generated signals. We used the kernel-generated signal mechanism to trigger implicit `dret()`s on process faults like divide-by-zero (`SIGFPE`) and memory access violations (`SIGSEVG`).

We augmented Linux's `task_struct` process structure with a new *deterministic PID* and low-level synchronization primitives. `dput()` and `dget()` use these synchronization primitives to synchronize using the fork-join model. When a parent starts a child with `dput()`, any subsequent `dput()` or `dget()` call blocks until the child issues a `dret()`.

Even though master processes have direct access to kernel I/O devices, we 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 the process is not the parent of any other nondeterministic process and 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 use the legacy `fork()` family of syscalls. It can only create new processes through the deterministic family of syscalls.

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* [10]. 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 [7].

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.

Component	Lines of code added
Primary syscall implementation	1187
Memory subsytsem	1081
Kenrel miscellaneous	296
New user library	W
Borrowed user library	U
Total	2916

Table 3: Count of number of lines of code added.

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.

4.5 Implementation Statistics

We began with a `git` fork of Linux 3.0. Table 3 lists lines of code added (including comments but excluding newlines) for various kernel and user library components. The "new user library" category counts code added by us, and the "borrowed user library" category counts code that was reused from `JOS`. In total, the kernel required 2564 additional lines of code.

The kernel [3] and user library [4] are available on GitHub as two separate repositories. The deterministic Linux kernel is maintained as a separate branch `v3.0-det`.

5 Evaluation

6 Related Work

Tools to alleviate nondeterminism

Similar deterministic environments

What is Determinator based on?

7 Future work

8 Undergraduate Research Experience

Before we conclude, this section will reflect on the undergraduate research journey as experience by the primary author, Chris Cotter. I initially read Aviram et al.'s "Efficient System-Enforced Deterministic Parallelism" paper in July of 2011 and wrote my first line of code in the Linux kernel that August. After many iterations, my final implementation of deterministic Linux took up the month of September 2012, and I wrote this thesis soon after.

Learning the Kernel There is no "Linux Kernel 101" course at the University, and most existing documentation and comments in kernel code are written for seasoned kernel programmers. Thus, I very often found myself lost and frustrated. It wasn't until I spent a year (August 2011 - 2012) of kernel hacking and until I finally felt comfortable implementing my third and final iteration of deterministic Linux.

8.1 First Iteration

In August 2011, I began by downloading Linux 2.6.32 source and learned to compile and run the kernel with `QEMU` [4]. I ran `QEMU` with an ramdisk containing a single program to run as *init*. Mike Walfish gave me my first goal: to implement a new syscall in Linux. After scouring the Internet, I learned how to do this, and I wrote skeleton code for my three syscalls: `dput()`, `dget()`, and `dret()`.

I iteratively implemented Determinator's functionality in these syscalls, starting with process organization and moving to memory operations. I had a particularly difficult time with memory operations. Even though I was a wizard with my operating system's instructional `JOS` OS and page table management, I had no idea how to maneuver Linux's memory subsystem.

After fumbling around with countless kernel panics, I set out with a simple goal: to change a single page table entry. After accomplishing this goal, I was ready to implement Determinator's Zero and Copy operations. In fact, I eventually found I could reuse and adapt a lot of existing code for these operations. Implementing Merge took considerably more effort, since Merge in Linux is an entirely novel operation.

Unfortunately, this version of my implementation was buggy, primarily due to misuse of internal kernel API and race conditions in my kernel code.

8.2 Second Iteration

In November 2011, I downloaded Linux 2.6.38 source and began rewriting my code, copying and pasting most of my first iteration code. I also started running an Arch Linux distribution on QEMU, since I was able to run more sophisticated tests with an actual Linux distribution running my kernel. With a few months of kernel hacking knowledge under my belt, I identified many logic bugs and had a better understanding of how things worked "under the hood". Unfortunately, I encountered many setbacks.

New Memory Features Moving from Linux 2.6.32 to 2.6.38 introduced new memory subsystem features. Notable among these was transparent huge pages (THP). When processes map a large enough region of virtual memory (e.g. at least 4MB), the kernel will sometimes fulfill demand paging requests with huge pages transparently without the user knowing the difference.

Since my original code did not account for THP, a lot of my existing kernel code broke, and I spent days trying to understand what went wrong and perhaps a week

[3] `https://github.com/ccotter/linux-deterministic`
[4] `https://github.com/ccotter/libdeterm`

devising a solution. In the end, I came to realize I still lacked a great deal of knowledge about Linux's memory subsystem, and this lack of knowledge would continue to plague my second iteration's quality.

Condition Variable Usage Violation My operating systems professor Mike Walfish taught us to always surround the testing of condition variables with a while-loop and not an if-statement. Unfortunately, I completely disregarded this lesson at some point in my implementation of `dput()` synchronization, and I found threads being woken up prematurely.

Snapshot/Merge Bug When I ran a stress test to fork hundreds of processes than did a simple Snapshot and Merge, I encountered a kernel panic that caused unrelated processes to crash (i.e. `bash`). Through the course of a month, I never identified the issue except to say that my lack of a thorough understanding of the memory subsystem was at fault. This, and a general lack of organization in my code lead me to write a third iteration.

8.3 Final Iteration

In September 2012, I forked a copy of the Linux `git` repository and started with Linux 3.0. I started running my code on an 8-core machine with 8GB of ram; the 8 cores maximized parallelism to help expose concurrency bugs in my kernel code. I also decided to do a complete rewrite - no old code from previous iterations would be copied and pasted.

After a year of kernel hacking, I had never felt more confident in the code I wrote; whereas in my second iteration I was not confident in my code's correctness, in my third iteration I could explain nearly every part of the kernel that my code interacted with.

General Success As I wrote and tested code, I often found that my code the first or second try. This was primarily due to careful and thoughtful reasoning about anything I wrote. In previous iterations, I often wrote code and ran it without fully knowing what to expect.

Squashing the Snapshot Bug Through the process of rewriting, I identified the Snapshot/Merge bug described above: I did not acquire a spinlock when operating on sensitive kernel data structures in my Snapshot code path. Unfortunately, this spinlock had very little accompanying documentation, and it was only through many months of kernel hacking that I even knew to use the spinlock.

9 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*, 2010.

[4] F. Bellard. Qemu open source processor emulator. *URL: http://www. qemu. org*, 2007.

[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] F. K. et al. 6.828: Operating system engineering. `http://pdos.csail.mit.edu/6.828`.

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

[9] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing*, pages 471–475, 1974.

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

[11] 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.

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

[13] R. Nelson and A. Tantawi. Approximate analysis of fork/join synchronization in parallel queues. *Computers, IEEE Transactions on*, 37(6):739–743, 1988.

[14] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *14th ASPLOS*, Mar. 2009.

[15] 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.