Enforced Deterministic Program Execution in the Linux Kernel

Chris Cotter
University of Texas

Abstract

[to be done]

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 neither deterministic nor predictable. This model is popular, because threads can operate on shared data "in-place" instead of having to pack and unpack data structures [4]. Unfortunately, this model is error prone and has many drawbacks [2, 19, 21]. Programmers must eliminate data races introduced by nondeterminism. Without repeatability, debugging, testing, and ensuring software quality assurance become difficult.

Existing attempts to provide determinism require special hardware [12, 13], custom programming languages [9], or specialized compilers [6]. Record-and-replay systems [18, 23] incur high overhead and do not offer any insight into the inherently nondeterministic behavior of a program. Systems that rely on a deterministic scheduler residing in user space are often arbitrary and unpredictable [26]; buggy or malicious application code can compromise the scheduler [4, 11]. Some relaxed deterministic models only enforce determinism on synchronization and permit unsynchronized low-level memory accesses [26], thus leading to nondeterministic traces in programs that do not properly protect critical sections.

To overcome the challenges of nondeterminism, Aviram et al. presented a deterministic operating system called Determinator [4]. Programmers write parallel applications using a novel parallel programming model that is "naturally and pervasively deterministic" and in fact predictable. Determinator run programs written in general-purpose languages on conventional hardware and enforces determinism at the system level. Determinator also contributes 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.

For a deterministic Linux, we prefer strong determinism 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" [8].

This thesis is about adapting Determinator's operating system design and programming model to Linux. 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. The bulk of the work in this area is implementation of an existing model. Additionally, we identify and eliminate Linux-specific sources of nondeterminism and contribute the *hybrid process* model, an enhancement of Determinator's *private workspace*. This allows us to run legacy nondeterministic and deterministic applications side-by-side in Linux.
- A deterministic high-level user library for deterministic Linux applications. This library is motivated by Determinator's user library, and again the bulk of the work in this area is implementation.
- An improvement of Determinator's in-memory file system. Determinator's file system maps a fixed number of files to static locations in memory and does not provide persistence. Our file system is modeled on the BSD Fast File System [22]; thus, we support features like hard-linking and dynamic allocation of data blocks. Moreover, we provide persistence by saving the in-memory file system to Linux's disk-backed file system.
- An evaluation of deterministic Linux against traditional nondeterministic parallel Linux. [preview what we found] We also demonstrate a case study of the benefits of determinism by exhibiting a bug that manifests nondeterministically in legacy Linux, but deterministically in deterministic Linux.

Aviram et al. were motivated by meeting the "software development, debugging, and security challenges" of writing future parallel applications [4]. 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 [7, 8, 26]. 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 [24]. 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 [7].

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 [3]. 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.

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" [7]. 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. [4]

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 [4]. 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 [4]. 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 [25]. 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 [4, 15]. Processes cannot outlive their parents and can communicate only with their parent 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" [4]. 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.

Processes execute in a *private workspace* model. Each process (or thread) keeps a private copy of the application's virtual memory. Processes do not see another process's writes until reconciliation at explicit synchronization points. This model eliminates read/write races, and write/write races are still deterministic since the program defines reconciliation order.

Kernel interface Processes communicate with the kernel via three syscalls, `Put`, `Get`, `Ret`. Tables 1 2, reproduced from [4], 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. Invoking `Get` with the `Merge` parameter performs a three-way diff and merge. The kernel compares bytes in the child that have changed since the previous `Snap` invocation. Bytes that have changed in the child only are copied 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 an easy way to implement fork-join. 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 that the three syscall primitives "reduce to blocking, one-to-one message channels, making the space hierarchy a deterministic Kahn network" [4, 17].

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 [4].

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

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. [4]

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. [4]

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 [27]. 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 [3].

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 [4].

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 its nondeterministic API) and *deterministic* Linux.

3.1 Design goals and non-goals

We wish to make 64-bit Linux deterministic, and in doing so we will present an interface similar to that of Determinator. We would like to run legacy Linux applica-

tions without modification alongside deterministic applications, for this is one of the motivating factors applying Determinator's design to Linux. We won't make any attempt to force legacy applications to run deterministically. 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 and an in-memory file system based on those of Determinator. To address Determinator's in-memory file system's limitations [3, 4], we would like to improve the file system by adapting the BSD Fast File System [22] and having deterministic applications utilize Linux's disk-backed persistent storage.

We do not wish to apply all of Determinator's features to our deterministic Linux. The Determinator kernel: (a) extends its nested process model to support deterministic distributed cluster computing; (b) supports a "tree" copy operation for `Put` and `Get`; and (c) allows threads to place an instruction limit on children threads. We have no intention of supporting these features, but we note these limitations do not detract from our goal of making Linux deterministic.

Since the primary goal is determinism, we may decide to limit or ignore Linux kernel internal features. For example, Linux supports huge 2MB pages of memory alongside "normal" 4KB 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.

Other useful library-level features may be unavailable to deterministic programs. 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 outside the scope of our primary goal.

3.2 Challenges

We already discussed the four categories of nondeterminism identified by Aviram et al. in section 2.2; these observations are general enough that they also apply in making Linux deterministic. The Linux kernel presents additional challenges, and we discuss them here.

In order to run legacy Linux applications, we cannot enforce Determinator's requirement that all but a single root process run in deterministic mode. Linux's process model is also too flexible: it allows reparenting and for children to outlive parents, directly opposed to Determinator's nested process model.

Since Determinator was written from scratch, the designers addressed sources of nondeterminism by simply not providing kernel support for such features. On the other hand, Linux's existing threading model is inherently nondeterministic and already provides extensive support for nondeterministic features (e.g. the `gettimeofday()` syscall). We must also address nondeterminism not covered by Aviram et al. Linux supports a wider range of process communication utilities including signals, pipes, and System V IPC.

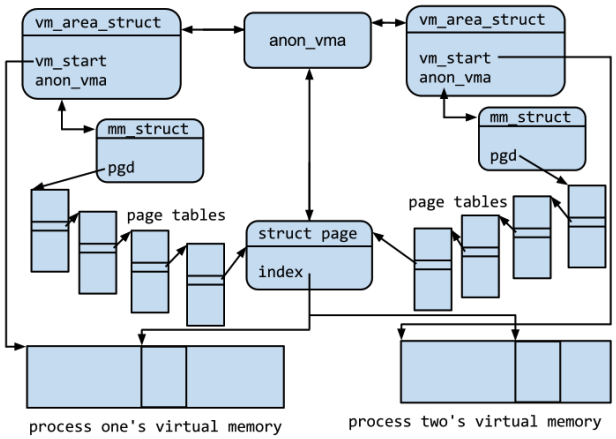

Figure 1—Data structure relationships associated with object-based reverse mapping. Two processes map virtual memory to the same page read only (possibly at different addresses). In order for the kernel to swap a given page to disk, object-based reverse mapping assumes each process maps the page to `vm_area_struct->start_addr + page->index` in virtual memory.

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 of many different processor architectures (e.g. x86_64, Alpha). Whereas Determinator's memory management subsystem deals only with page table management for a single architecture, Linux's memory subsystem uses more complicated data structures and nontrivial algorithms. Figure 1 demonstrates the complicated data structures associated with object based reverse mapping [10] used to find all processes that map a given page of memory. From an implementation perspective, Linux's memory subsystem's complexity presents a potentially formidable challenge in implementing Determinator's three memory operations (`Zero`, `Copy`, and `Snap/Merge`).

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 func-

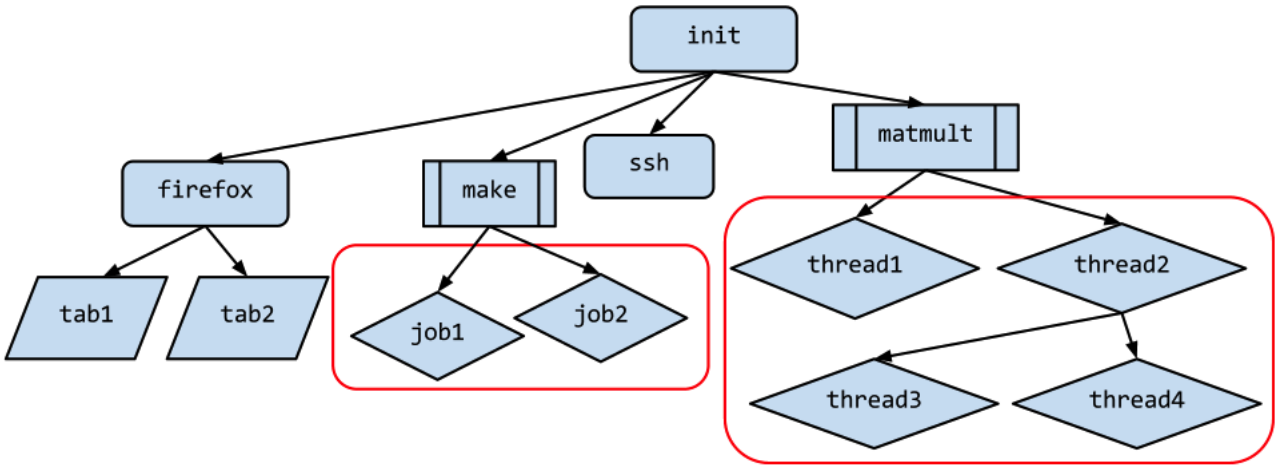

Figure 2—Illustration of the hybrid process model with two deterministic process groups. Legacy nondeterministic processes (`firefox`, `ssh`) run as children of `init`. `make` and `matmult` are masters of deterministic applications. The deterministic children (the diamond processes) are isolated from the rest of the system.

tions, 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, libraries in Linux are often linked dynamically with shared libraries, but since 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). 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. Figure 2 illustrates the hybrid process model in a hypothetical environment.

At the expense of predictability, but without harming determinism, master processes can use nonblocking `dput()` and `dget()` (invoked 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. Since threads are responsible for scheduling children, a CPU might become available while the scheduler thread remains blocked waiting for another thread to finish. Applications that use the nonblocking syscalls can still log schedule sequences to reproduce program output in a deterministic fashion.

We also allow signals to reach master processes. 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. Deterministic applications won't use the standard C library, because many library calls invoke disallowed legacy syscalls. 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 uses fixed file size [3], our file system design is based on the BSD Fast

File System [22]. Our file system is organized as a rooted tree and supports hard linking, dynamic file sizes that can grow up to 1GB, and better resource management (inodes, data blocks). Since a disk-backed file system is standard on Linux systems, master processes can save and load the in-memory file system to disk to provide persistence.

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.

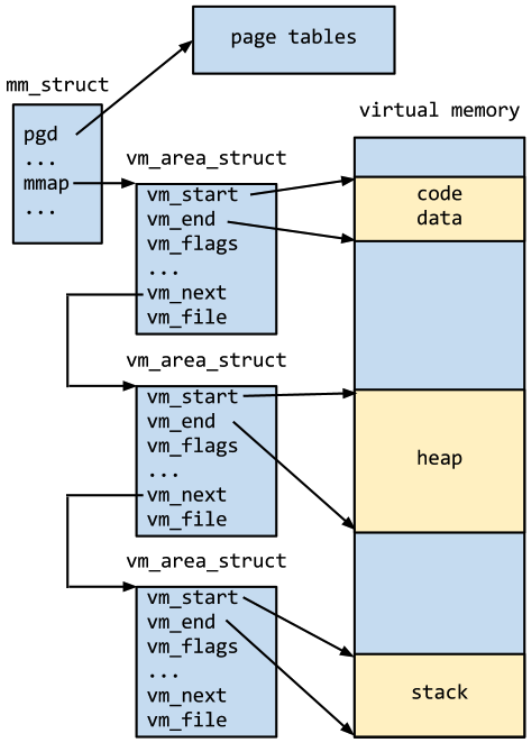

Figure 3—Illustration of Linux's memory management data structures.

Master processes can use `dput()` to specify a set of signals to ignore while in a blocking `dput()` or `dget()` (i.e. waiting for a child to stop). 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()` can put or retrieve a child's general-purpose register state.

4.2 Memory operations

As mentioned in the section 3.2, Linux's memory subsystem is very complex. Before discussing how we implemented the memory operations, we will give some technical background on Linux's memory subsystem. Our implementation reused many existing functions, 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 (see Figure 3). Anonymous memory (memory not backed by a file) is mapped to a read only global zero page, and new pages are allocated *on demand* [20]. 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 (see Figure 1). 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 the virtual memory image of a process. fork() uses this to copy a process's memory via 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 on the C standard 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 [14].

Designing and implementing what amounts to be a replacement for the C standard 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 where 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 [4]. 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.

Linux specific considerations Before executing `main()`, the library runtime 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 settings its status code and issuing a `dret()`. 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`. The library runtime choses the appropriate action by checking if the executing process is the master.

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. We since implemented deterministic Linux on x86_-64, we do not run into the address space limitations imposed by 32-bit systems.

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.

Component	Lines of code added
Primary syscall implementation	1187
Memory subsytsem	1081
Kenrel miscellaneous	296
New user library	2492
Borrowed user library	1727
Total	7135

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

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 [1] and user library [2] are available on GitHub as two separate repositories. The deterministic Linux kernel is maintained as a separate branch `v3.0-det`.

5 Evaluation

This section evaluates deterministic Linux by running compute-bound applications using deterministic Linux and nondeterministic pthreads. We conclude by considering a case study demonstrating the qualitative debugging benefits of determinism.

5.1 Empirical experiments

Aviram et al. already demonstrated course-grained applications in Determinator performed comparable to nondeterministic Linux equivalents, but fine-grained applications did not scale nearly as well, incurring high performance costs owing to memory synchronization costs.

The primary goal of our evaluations is to determine if Linux can efficiently run applications using Determinator's programming model. Since the deterministic Linux kernel reuses a lot of existing code, we expect reasonable performance compared to the equivalent nondeterministic application. However, applications that heavily rely on memory-intensive kernel operations like Snapshot/Merge might incur performance penalties.

5.1.1 Methodology

We evaluated four benchmark programs. *md5* searches for a string whose md5 hash yields a particular hash value (i.e. a brute force password cracker). *md5* creates $N + 1$ threads to perform parallel work where N is the number of processors available. The *qsort* benchmark recursively

[1] `https://github.com/ccotter/linux-deterministic`
[2] `https://github.com/ccotter/libdeterm`

	lu				matmult			
Dimension	$N = 1$	$N = 2$	$N = 4$	$N = 8$	$N = 1$	$N = 2$	$N = 4$	$N = 8$
16×16	23.7 (49.9%)	27.3 (45.2%)	33.3 (45.7%)	25.5 (31.7%)	14.1 (49.9%)	33.9 (58.2%)	30.7 (51.9%)	24.9 (38.1%)
32×32	23.3 (49.8%)	26.6 (45.7%)	30.4 (45.2%)	24.4 (31.3%)	14.3 (49.8%)	29.7 (56.5%)	26.3 (52.4%)	20.1 (37.0%)
64×64	22.2 (49.0%)	26.5 (45.2%)	33.2 (44.5%)	24.9 (31.0%)	10.7 (48.4%)	26.3 (55.7%)	33.9 (51.7%)	24.0 (39.5%)
128×128	15.4 (46.7%)	22.4 (44.0%)	28.6 (42.4%)	27.2 (30.8%)	4.1 (37.6%)	14.6 (53.9%)	21.6 (46.9%)	24.1 (39.8%)
256×256	5.3 (38.4%)	11.7 (38.7%)	14.1 (36.8%)	14.9 (31.8%)	1.5 (16.8%)	4.0 (28.5%)	7.9 (29.2%)	8.2 (27.0%)
512×512	1.8 (17.9%)	4.0 (24.1%)	6.3 (22.2%)	5.8 (19.2%)	1.1 (16.7%)	1.5 (13.5%)	2.6 (12.6%)	2.3 (15.6%)
1024×1024	1.2 (4.3%)	1.5 (11.8%)	2.1 (12.5%)	1.8 (7.8%)	2.6 (0.6%)	0.9 (0.5%)	0.6 (1.0%)	0.7 (1.0%)
2048×2048	1.0 (0.4%)	1.1 (0.9%)	1.2 (1.2%)	1.1(1.0%)	-	-	-	-

Table 4: Deterministic overhead for *lu* and *matmult*. The numbers in parentheses indicate time spent in the kernel performing a virtual memory merge as a percentage of overall runtime.

	qsort			
Input size	$N = 1$	$N = 2$	$N = 4$	$N = 8$
1K	27.51	39.22	24.14	17.38
4K	14.07	46.84	37.86	20.80
8K	8.57	28.73	31.38	23.13
10K	7.37	21.17	30.60	26.58
40K	2.52	8.43	13.41	16.80
80K	1.73	4.31	8.37	10.57
100K	1.61	3.10	7.57	11.08
400K	1.17	1.80	2.20	2.70
800K	1.11	1.41	1.63	1.65
1M	1.10	1.36	1.52	1.75
4M	1.07	1.22	1.26	1.35
8M	1.04	1.05	1.22	1.36
10M	1.04	1.01	1.11	1.31
40M	1.04	1.09	1.18	1.27
80M	1.03	1.15	1.16	1.30

Table 5: Deterministic overhead for *qsort*.

sorts an integer array by forking children threads going $log_2(2 \cdot N)$ levels deep, after which the recursive computation is carried out sequentially. *matmult* multiplies two square matrices by dividing the input into 64 blocks and forking off threads to multiply the individual blocks. *lu* performs LU decomposition of a square matrix by breaking the input into 256 blocks and creating a thread for each block. Our LU algorithm breaks the blocks up into discontiguous blocks of memory (i.e. we represent our matrix in row-major order).

All benchmarks are designed so the deterministic and pthread versions operate on identical inputs generated from a pseudorandom number generator (*matult*, *lu*, *qsort*) or "hardcoded" input (*md5*). For a given input size, *qsort* runs three tests on different inputs and averages the run times. Both matrix benchmarks run ten tests on different inputs and average the results. *md5* searches for eleven fixed ASCII strings whose values are distributed randomly over a fixed alphabet. The run times for all eleven runs are averaged to a single value.

The *matmult*, *qsort*, and *md5* benchmarks are modified versions of benchmarks found in Determinator's source on GitHub. [3] The *lu* benchmark was written from scratch using an algorithm describe on the Internet [16]. We tested on a 2 socket $\times$ 4 core 2.33 GHz Intel Xeon PC running Arch Linux with 8GB of RAM. All benchmarks use the same modified x86_64 Linux 3.0 kernel.

The deterministic benchmarks create threads by issuing a `dput()` specifying the `Snap` option. We join thread results by performing a `Merge` on the program's entire static data segment (initialized and uninitialized). Thus, deterministic applications incur the performance penalty of creating two copies of the parent thread's page tables and associated kernel data structures as well as scanning the page tables and potentially the pages themselves of the child's entire static data. The pthread programs incur no such penalties, since `pthread_create()` does not make copies of page tables and `pthread_join()` does the same work as `dget()` without the `Merge`.

5.1.2 Results

Applications that require a lot of memory synchronization show considerable overhead for small inputs, but have much more acceptable overheads for large inputs. Table 4 shows deterministic overheads for the *lu* and *matmult* benchmarks. Inputs smaller than 1024×1024 spend anywhere from 12.6% to 58.2% of their run time in the kernel doing a memory merge. Small inputs in general have a hard time seeing parallel benefits: the pthread versions of both benchmarks did not see parallel speedup benefits until the input matrix reached at least 64×64. Still, it isn't until even larger input sizes (at least 1024×1024) when we begin to see more "acceptable" deterministic overheads of at most $2.1\times$ for $N > 1$. *matmult* performs better than pthreads on the largest input size 1024×1024; we cannot account for this, but a good place to begin investigating this occurrence might be comparing how the kernel schedules pthreads threads and deterministic threads.

The *qsort* benchmark also shows unacceptable overhead for small inputs, but once the input array size becomes at least 800K, overheads stayed under $2\times$ (Table 5). The embarrassingly parallel *md5* benchmark exhibited very little overhead; it had overhead of $1.17\times$ for $N = 1$

[3] `https://github.com/bford/Determinator`

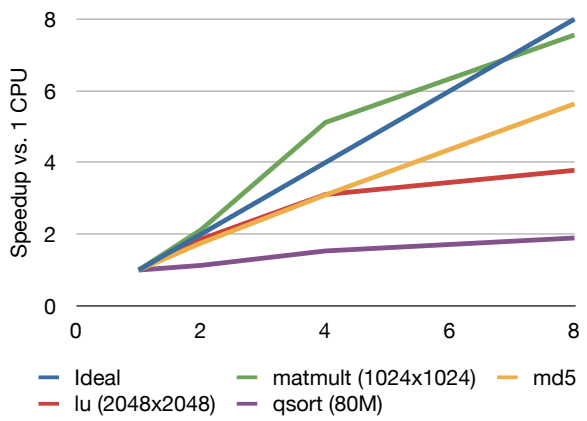

Figure 4—Deterministic speedup for the parallel benchmarks.

and $1.06\times$ for $N = 8$. This can likely be attributed to the little amount of information transfered back and forth between threads (a single boolean and the matching string when it is found).

Figure 4 shows speedup over one CPU for the deterministic benchmarks. We show the results when run on the largest input size available (e.g. 80M for *qsort*). The *matmult* benchmark appears to show speedup better than the ideal case. This can be attributed to the base case $N = 1$ running exceptionally slow. *matmult* always creates 64 threads, and we believe the overhead of managing 64 threads with a single processor plus the cost of merging adds significant overhead. *md5* scales well, *lu* shows less scalability, and *qsort* scales poorly, barely reaching $2\times$ when $N = 8$. *matmult* shows promise in its ability to scale, but we cannot be too sure due to irregular behavior in the base case.

5.1.3 Fine-tuning the benchmarks

The apparent overhead for managing threads in the pthread and deterministic benchmarks appears to have affected our results. Deterministic *matmult* runs faster for 1024×1024 matrices than the pthread version and appears to scale better than the ideal case. Regardless of the number of available processors, the main thread in both matrix benchmarks starts by forking a large number of threads (64 for *matmult*, 256 for *lu*), then joining on each thread. A more sane approach might be to base the number of threads created on N. We could also distribute a large number of tasks among a thread pool.

We may attribute some of the exceptionally high overhead for small inputs to the cost of merging. For simplicity, our deterministic thread join merges the entire static data segment. *qsort*, *matmult*, and *lu* all declare static arrays with sizes as large as the maximum expected input. For example, *lu* declares two `long double` arrays of size 2048×2048 for a total of 128MB on our 64-bit machine. When we join on all 256 threads for the 16×16 input, we merge over all 128MB. Page table optimizations (Section 4.2) enable the kernel to only have to do a byte-by-byte merge for the single page containing the input matrix, but the kernel must still check at least 32767 other page table entries, assuming 4KB pages!

Thankfully, the syscall API allows us to specify a range of virtual memory to merge; we could fine-tune our programs to merge only what we know has changed. For the *qsort* benchmark, we could opt to use the `Copy` to achieve the same effect. `Copy` copies page tables directly without checking page table entries and possibly the bytes themselves as `Merge` does. By having our high-level join function merge the entire static data segment, our join method is very general in the types of programs it can be used with. We could likely trade this generality for performance gains.

5.2 Finding bugs deterministically

To demonstrate one of the key benefits of deterministic execution (Section 2.1), we consider a Gaussian elimination program written using the deterministic API and pthreads. Figures 5a and 5b show nondeterministic and deterministic portions of Gaussian elimination code. There is a crucial synchronization bug, however: both algorithms create `nrows - k` worker threads but only join `nrows - k - 1` threads. We purposefully inserted this bug, but the reader can imagine a programmer making this typo by mistake.

We ran both versions multiple times and examined the resulting matrix. The deterministic program *always* produced the wrong answer. `djoin()` merges changes back into the main thread for all but the last worker thread. When the last worker thread finishes, its changes are private and never seen by the main thread.

On the other hand, the pthreads program executes nondeterministically. We observed three different output matrices, one of which was the correct result. If the final worker thread finishes before we observe the final result, the output will be correct. If the final worker thread does not finish by the time we examine the output, we get an intermediate result. The output is nondeterministic, owing to a race condition, since all threads share memory.

Determinism provides a clear benefit in debugging, since the deterministic version always behaves the same, whereas the buggy behavior might exhibit itself rarely in the pthread version. We also consider testing our application to ensure its correctness. When testing our Gaussian elimination application, determinism allows us to see right away that, in general, all inputs give incorrect answers. On the other hand, it may be the case that the nondeterministic version always gives the correct answer for matrices we test on a development machine, but as soon

```
pthread_t thread[MAXTHREADS];
struct thread_data data[MAXTHREADS];
void pthread_reduce(void) {
   for (i = 1; k <= nrows - 1; ++k) {
      for (i = k + 1; i <= nrows; ++i) {
         data[i] = /* Setup worker. */;
         pthread_create(&thread[i], NULL, worker,
            &data[i]);
      }

      /* Bug! Should be i <= nrows */
      for (i = k + 1; i < nrows; ++i)
         pthread_join(thread[i], NULL);
   }
}
```

Figure 5a—pthread Gaussian elimination.

```
/* Forks a deterministic child. Returns 0 into the
 * child and 1 into the parent. */
int dfork(pid_t childid);
/* Merges a child's changes into the parent after
 * the child issues a dret(). */
void djoin(pid_t childid);

void det_reduce(void) {
   for (i = 1; k <= nrows - 1; ++k) {
      for (i = k + 1; i <= nrows; ++i) {
         data[i] = /* Setup worker. */;
         if (!dfork(i)) { worker(&data[i]); dret(); }
      }

      /* Bug! Should be i <= nrows */
      for (i = k + 1; i < nrows; ++i)
         djoin(i);
   }
}
```

Figure 5b—Deterministic Gaussian elimination.

as we ship the program to a client's machines with a different number of processors (for example), the bug exhibits itself.

6 Limitations and future work

[not finished]

I'll discuss Linux specific limitations like deterministic processes not using the full memory subsystem feature set (e.g. huge pages). File system merging is slow: because data blocks live in arbitrary regions throughout virtual memory, reconciliation involves chasing data block pointers and invoking dput/dget many times (linear in file size). This could be alleviated by adding kernel support for an in-memory file system.

We don't support instruction limits, so malicious or buggy children could execute forever, blocking parents indefinitely.

Implementation is not optimized. Example: killing a child process is surprisingly slow (I'd like to find out why, since this should be a very quick nonblocking operation).

6.1 Future work

Recent efforts by Determinator's authors have extended the OS to support arbitrary process interactions (not just hierarchical). Determinator and deterministic Linux focused on compute-bound applications using fork-join. What about I/O intensive apps or other parallel paradigms?

7 Related Work

7.1 Tools to alleviate nondeterminism

Record-and-replay

7.2 Similar deterministic environments

Kendo, dOS, CoreDET, DPJ, ...

8 Reflections on my research experience

Before we conclude, this section will reflect on the undergraduate research journey as experience by the 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 implementation 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 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` [5]. I ran `QEMU` with an ramdisk containing a single program to run as *init*. My advisor 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 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

Hm...

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] C. Artho, K. Havelund, and A. Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, 2003.

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

[4] A. Aviram, S. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *OSDI*, 2010.

[5] F. Bellard. Qemu open source processor emulator. `http://www.qemu.org`.

[6] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. *ACM SIGARCH Computer Architecture News*, 38(1):53–64, 2010.

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

[8] 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.

[9] R. Bocchino Jr, V. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *ACM SIGPLAN Notices*, volume 44, pages 97–116. ACM, 2009.

[10] J. Corbet. The case of the overly anonymous anon_vma. `http://lwn.net/Articles/383162/`.

[11] H. Cui, J. Wu, C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. *9th OSDI*, 2010.

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

[13] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. Rcdc: a relaxed consistency deterministic computer. *ACM SIGARCH Computer Architecture News*, 39(1):67–78, 2011.

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

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

[16] M. T. Heath. Parallel numerical algorithms. `http://www.cse.uiuc.edu/courses/cs554/notes/06_lu.pdf`.

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

[18] T. LeBlanc and J. Mellor-Crummey. Debugging parallel programs with instant replay. *Computers, IEEE Transactions on*, 100(4):471–482, 1987.

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

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

[21] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ACM Sigplan Notices*, volume 43, pages 329–339. ACM, 2008.

[22] 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.

[23] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution ef? ciently. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*, pages 289–300. IEEE, 2008.

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

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

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

[27] 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.