Enforced Deterministic Program Execution in the Linux Kernel

Chris Cotter
University of Texas

1 Introduction

This thesis describes adding kernel enforced deterministic program execution to Linux. In our solution, at a high level, programs enter a *deterministic* mode where the kernel provides a very restricted subset of syscalls designed to enforce determinism. The kernel enforces determinism, thus making it impossible for user programs to possibly behave nondeterinistically, even by deliberate design. Internal nondeterministic inputs like hardware data races are entirely eliminated, and explicitly nondeterministic inputs like user input or time of day become controllable explicit I/O.

Nondeterministic parallel programming is difficult: nondeterministic inputs, such as data races, force programmers to use difficult to reason about synchronization primatives such as semaphores and condition variables. Misuse of these primatives can lead to buggy code and deadlock. Even correct use cannot guarantee deterministic execution: conventional synchronization primatives are not predictable. This has debugging, testing, and security implications.

Deterministic execution overcomes the challenges of nondeterminism, and according to Bergan et al. it provides benefits in four main areas: debugging, fault tolerance, testing, and security. Thus, we have a strong motivating factor for adding determinism to Linux.

The research presented and discussed in this thesis is based on Determinator. We adopt Aviram et al.'s operating system design to make Linux deterministic. We choose this approach since it enforces determinism from the kernel level using a straightforward syscall interface. This thesis also presents an accompanying user level C library, akin to the library utilities discussed by Aviram et al. This user library is intended to simplify writing user programs in C using familiar high level abstractions such as fork-join. We also provide an in memory file system, improving upon Arivam et al.'s in memory file system design.

The section following this introduction gives background on Determinator and further motivates making Linux deterministic. The next section will describe design goals for this research project, followed by a discussion of challenges in accomplishing the goals. Then, we will discuss the actual implementation details of the kernel and user library work.

In later sections, we evaluate deterministic Linux against legacy Linux and make some conclusions about Determinator's design in Linux compared to Linux's traditional multithreaded runtime. We will also discuss limitations and related work. Finally, we will begin to conclude by discussing the overall undergraduate research experience.

2 Background

Writing correct parallel programms is difficult in the absense of determinism; this section motivates determinism by describing its benefits in software development, then describes the Determinator operating system.

2.1 Motivation

Bergan et al. suggest there are four main benefits of deterministic execution in the following areas: debugging, fault tolerance, testing, and security. While there exist so called "point solutions" that solve problems in single areas at once, those solutions do not "compose well with one another" [3]. Determinism solves problems in all four areas at once with a single mechanism.

Debugging Debugging multithreaded programs can be difficult since often bugs are not easily reproducible and tools such as `gdb` are not always useful for tracking down heisenbugs [6]. Finding a bug's root cause becomes easier when a program's execution can be replayed over and over. Deterministic execution naturally provides replay debugging as a benefit.

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

Testing The difficulties in testing multithreaded applications are compounded by racy nondeterministic scheduling. Developers and automated test systems must consider the exponential blow up of possible scheduling sequences. Determinism helps alleviate this problem by guaranteeing a one-to-one correspondence between input and output. For each input, there is exactly one possible logical scheduling sequence of threads. Bergan et al. notes that this obvervation can help in designing test strategies [3].

Deterministic execution by itself it not as useful as the stronger guarantee of *predictability*. Some deterministic environments provide synthesized and arbitrary schedulers [2, 4]. These models do not allow one to reason about a program and determine the output beforehand, which makes designing tests difficult. The stronger claim of predictability allows for developers to design test suites that can accurately check a program's output.

Security Processes sharing a CPU or other hardware should be conscious about leaking sensitive data. Covert timing channels can be exploited by a malicious thread to extract sensitive data from other threads [1]. Determinism eliminates covert timing channels, since a program is purely a function of explicit inputs and cannot possible rely on the timings of hardware operations.

2.2 Determinator

Aviram et al. set out to provide

> a parallel environment that: (a) is "deterministic by default," except when we inject nondeterminism explicitly via external inputs; (b) introduces no data races, either at the memory access level or at higher semantic levels; (c) can enforce determinism on arbitrary, compromised or malicious code for security reasons; and (d) is efcient enough to use for "normal-case" execution of deployed code, not just for instrumentation during development. [2]

To this end, they present Determinator, a novel OS written from the ground up. First, Determinator identifies sources of nondeterminism, then defines a simple three syscall kernel interface, and finally recreates higher level programming abstractions from the low level syscalls.

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

Explicit Nondeterminism Often, programs rely on nondeterministic inputs such as network packets, user input, or clock time. These inputs are essential to a program being useful; therefore, a deterministic programming model must incorporate these inputs while still enforcing determinism. Determinator addresses these "semantically relevant" inputs by turning them into explicit I/O [2]. Applications have complete control over these input sources and can even log the inputs for reply debugging.

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

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

Nondeterministic scheduling abstractions Traditional multithreaded synchronization abstractions are often not deterministic or predictable. Random hardware races determine the next thread to acquire a mutex lock, and as mentioned before this has debugging and testing implications. Even though we can record lock acquisition sequences to replay program execution or use some arbitrary device to choose a deterministic sequence, the order of acquisition is not predictable. Determinator restricts itself to only allow naturally deterministic and predictable synchronization abstractions, such as fork-join.

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

2.3 The Determinator Kernel

Determinator organizes processes in a nested process model [5]. Processes cannot outlive their parents and can only communicate with their parents and children. In line with the earlier discussion of nondeterminism, the kernel "provides no file systems, writable shared memory, or other abstractions that imply globally shared state" [2]. Only "the distinguished root [process] has direct access to nondeterministic inputs" [2]. 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 summarized in Table 1: `Put`, `Get`, and `Ret`. `Put` and `Get` take parameters that specify various operations outlined in Table 2.

...obviously needs more content

2.4 Deterministic Linux

With Determinator's design presented, we can now motivate a deterministic Linux. Determinator was written from scratch in an academic environment with determinism as the main OS design goal. In some sense, Determinator is

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

Table 1—System calls comprising Determinators kernel API.

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

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

not a *real* operating system, and the potential uptake outside the academic world is minimal. On the other hand, Linux is a mature and widely deployed nondeterministic operating system. Linux is installed on millions of systems including desktop computers, embedded systems, and mobile devices. In other words, Linux is a *real* operating system used in the real world. By adding determinism to Linux, we are able to take advantage of the widespread use and adoption of Linux; the potential userbase for a deterministic LInux is much greater than that of Determinator. Furthermroe, we can evaluate Determinator's design in a real operating system.

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 (an unmodified Linux kernel) and *deterministic* Linux.

3.1 Design Goals

We wish to make 64-bit Linux deterministic, and in doing we will present an interface similar to that of Determinator. We also would like to run legacy Linux applications alongside deterministic applications, for this is one of the motivating factors applying Determinator's design to Linux.

Whereas Determinator forces all but one process to operate in *deterministic* mode, we wish to be able to run legacy Linux programs without modification; however, we won't make any attempt to force legacy programs to run deterministically, so legacy applications will run in 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 [1, 2].

We do not wish to apply all of Determinator's features to a deterministic Linux. Determinator supports deterministic compute clusters 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 the goal of a deterministic Linux.

Snce 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, for reasons of implementation complexity, we may not allow deterministic applications to take advantage of certain kernel features.

3.2 Challenges

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

Inherent nondeterminisim 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 some sources of nondeterminism, 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 of a wide range of processor types. Compared to Determinator, Linux's memory subsystem uses much more complicated abstractions to support these features. Understanding and overcoming this complicated system is essential to implementing determinism in Linux.

Standard C Library Many applications written in C on Linux 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 nondeterminstic syscalls, many other functions do use nondeterministic syscalls (e.g. printf()). Thus, deterministic programs may be forced to use a completely different library. Moreover, the libraries in Linux are often linked dynamically with shared libraries, but Determinator does not provide any native kernel support for dynamic linking. We may lose the ability to dynamically link shared libraries.

3.3 High level approach

To address concerns about Linux's more flexible process model, we present a *hybrid process model*. A Linux process invokes a syscall to become a *root* process, akin to Determinator's single root process. This root process has full access to the legacy Linux kernel API, with some minor restrictions noted below. Root processes then create *deterministic* children. Within the process hierarchy, 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 then add three new syscalls: dput(), dget(), and dret(); we restrict deterministic processes to only using these three 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 nondeterminisim; we only have to modifiy the kernel to ignore all signals sent to deterministic processes, and thus we have effectively blocked all sources of nondeterminism.

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 root and deterministic processes must use our new deterministic library, even though 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 [1], our file system design is similar to that of the BSD Fast File System [–cite–]. 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 pointers. Directories are files containing a list of files within the directory.

We also note that since root 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 Kernel Implementation

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

4.1 Process synchronization

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

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

In Determinator, *all* processes that issue a dput() or dget() block until the child in question issues a dret(). This is a limitation for applications that benefit from concurrency, such as running make -j2 [2]. As noted by Aviram et al., Determinator might miss opportunities to start a parallel job, because a deterministic make in Determinator schedules itself and might be blocked waiting for a thread to finish when other children are runnable. In our implementation we allow the root process to perform a special non-blocking dget() to determine whether or not a child is still running. This allows more optimal scheduling, since the root can find a runnable thread and give it work. Determinism is achieved by recording the scheduling sequence for replay, if desired. This feature is desirable for inherently nondeterministic applications like web servers that may wish to exploit as much concurrency as possible.

Root processes can use dput() to specify a set of signals to block while in a blocking dput() or dget(). Blocking versions of these syscalls ignore signals specified in the block set sent to the root 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 root 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

5 Conclusion

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

References

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

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

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

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

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

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