Finding Concurrency Errors in Sequential Code —OS-level, In-vivo Model Checking of Process Races

Oren Laadan, Chia-Che Tsai, Nicolas Viennot, Chris Blinn, Peter Senyao Du, Junfeng Yang, Jason Nieh
Columbia University

Abstract

While thread races have drawn huge attention from the research community, little has been done for *process races*, where multiple—possibly sequential—processes access a shared resource, such as a file, without proper synchronization. We present a preliminary study of real process races and show that they are numerous, dangerous, and difficult to detect. To address this problem, we present the design of RACEPRO, an *in-vivo* model checking system for automatically detecting process races in deployed systems, along with preliminary results from a RACEPRO prototype. To the best of our knowledge, we are the first to study real process races, and RACEPRO is the first system to detect them.

1 Introduction

The rise of multicore has created a huge buzz around thread races in multithreaded programs.[1] Thread races are numerous, dangerous, and nondeterministic [10]; they are widely regarded as some of the worst possible errors. Many have flocked to the area of reliable multithreaded software and proposed a plethora of approaches to detect, diagnose, and avoid thread races.

Just like threads have races with load/store, processes have races with `read()`/`write()`. However, the research community has done little for such *process races*, where multiple—possibly sequential—processes access an OS resource (*e.g.*, a file or device) without proper synchronization. For instance, while over 30 papers in recent premier systems/PL conferences (OSDI, SOSP, PLDI, and ASPLOS) are on thread races and reliability of multithreaded software, only two deal with process races and neither helps detecting them.

Process races are much broader than time-of-check-to-time-of-use (TOCTOU) races. A typical TOCTOU race is an atomicity violation where the permission check and the use of a resource are not atomic, so that a malicious process may slip in. In contrast, a process race may be any form of race. Some real examples include a shutdown script unmounting a file system before another process writes its data, "`ps | grep X`" shows N or $N+1$ lines depending on the timing of the two commands, and "`make -j`" failures.

[1]In this paper, we broadly define races to include read-write and write-write races, atomicity violations, and order violations [10].

An important message of this paper is that process races are bad and we urge the systems community to give them their due share of attention. We have conducted a preliminary study of real process races and show that they are numerous, dangerous, and difficult to detect. A simple search on Launchpad [3], Ubuntu's software management site, returns hundreds of potential process races. This number has risen due to multicore, evident by the many process races in Ubuntu's parallel boot support [4]. Our study identifies many specific process races and shows that they can cause program crashes, data loss, and other failures. Our study also shows that process races are difficult to detect and many occur only due to rare runtime and deployment factors. The result is that dangerous process races often sneak into deployed systems, causing "heisen" production failures.

Our goal is to build effective tools to automatically detect, diagnose, and avoid process races. As a first step, this paper outlines how we may detect process races. Two challenges make detection difficult. The first is scope: process races are extremely heterogeneous. They may involve many different programs. These programs may be written in different programming languages, run within different processes, access diverse resources, and use diverse synchronization operations (*e.g.*, fork-wait, pipe, and signal). Existing thread or TOCTOU race detectors are unlikely to work well with this heterogeneity.

The second challenge is coverage: process races are highly elusive. Like thread races, they are timing-dependent, and tend to surface only in rare executions. Worse than thread races, they may occur only under specific software, hardware, and user configurations at specific sites. It is hopeless to rely on a few software vendors and beta testing sites to create all possible configurations and executions for checking.

To address these challenges, we present the design of RACEPRO, a system for automatically detecting process races, without requiring source code or modifications to the checked systems. RACEPRO addresses the scope challenge by transparently and pervasively tracking shared resource accesses (*e.g.*, the file and byte range accessed by `write()`) at the OS-level, for both user processes and the kernel. It mitigates the coverage challenge by checking *deployed* systems. The insight is that all user machines together can create a much larger and more diverse set of configurations and executions for checking. Alternatively, if a configuration or execution never occurs, it is probably not worth checking.

To detect process races, RACEPRO employs an approach we call *in-vivo model checking*. While a deployed system is running, RACEPRO records the execution without doing any checking. RACEPRO then systematically checks this recorded execution for errors *offline*, *e.g.*, when the deployed system is idle or by replicating it to a dedicated checking machine. To check a recorded execution, RACEPRO first analyzes this execution to identify a set of *execution branches* that may lead to process races. It then checks each potentially buggy branch by replaying the recorded execution, making it *go live*—*i.e.*, resume live execution—down this branch, and testing whether any process race occurs.

Our approach has three key benefits. First, it decouples execution recording from checking, thus incurring little overhead on the deployed systems it checks.[2] Second, it increases coverage by checking many execution branches. Third, by creating a live execution and verifying that a race does occur and lead to harm, it reduces false positives.

We have implemented a preliminary RACEPRO prototype in Linux, which consists of a record-replay kernel module and a user-space exploration engine for systematically checking execution branches. Our preliminary results show that RACEPRO incurs little overhead (under 2.5% for server and 15% for desktop applications) and detects two known process races.

This paper makes two main contributions. First, through an initial study (§2) of process races in real software, we show that they are a real threat. Second, we have designed (§3) and implemented (§4) a preliminary prototype of RACEPRO. To the best of our knowledge, we are the first to study process races, and RACEPRO is the first system to detect them.

2 Process Race Study and Examples

To better understand process races, we have conducted a preliminary study with two key questions in mind:

- How serious is the problem of process races?
- What are their characteristics that hint towards potential methods to detect them?

We collected the races by searching for the term "race" on Launchpad, Ubuntu's software management site. Our search query returned 3,330 pages. We then sampled 300 pages, which yielded 69 unique bugs. We then manually examined these bugs and classified them. Raw data for all bugs studied is available [1]. §2.1 describes general findings and §2.2 presents four example process races, from the most to the least serious.

[2]Previous work [6] also decouples recording and checking, but it checks the exact execution recorded and does not detect process races.

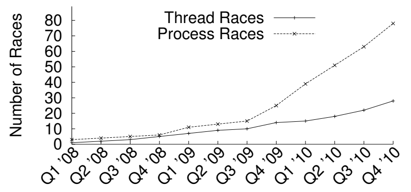

Figure 1: *Sampled thread and process races over time.*

2.1 Findings

Process races are numerous. Of the 69 sampled bugs, 42 are process races, a dominant majority; 24 are potential process races but the page did not contain enough information for us to understand the cause, so we did not count them when computing the statistics below; the other 3 bugs are thread races.[3] Of the 42 process races, only 1 is a TOCTOU race; the other 41 races are not TOCTOU races, which existing TOCTOU detectors cannot catch. Based on this sample, the 3,330 pages that our simple search returned may extrapolate to about seven hundred process races. Figure 1 shows that the number of process races has also risen in recent years.

Process races are dangerous. Compared to thread races that corrupt volatile application memory, process races corrupt persistent/system resources, making them potentially more dangerous than thread races. As described in §2.2, the process races can cause programs to read garbage, processes to get stuck in infinite loops, and files and databases to become corrupted. These races are just the tip of the iceberg; please refer to [1] for the effects of all studied races.

Process races are heterogeneous. The sampled races spread across 74 programs, ranging from server applications such as MySQL, to desktop application such as OpenOffice, to shell scripts in Upstart [4], an event-driven replacement of System V initialization scripts. 35 of the 42 races, including all races described in A§2.2, require interactions of at least two programs. These programs are written in different programming languages such as C, Java, and PHP, run in different processes, synchronize via `fork()` and `wait()`, pipes, sockets, and signals, and access resources such as files and devices.

This heterogeneity makes it difficult to apply existing detection methods for thread or TOCTOU races to process races. For instance, static thread race detectors (*e.g.*, [8]) work only with one program written in one language, and dynamic thread race detectors (*e.g.*, [13]) work only with one process.

Process races are highly elusive. Many of the sampled process races (*e.g.*, Race 1 and Race 3 described in §2.2) occur only for specific combined software, hard-

[3]The number of thread races is likely underrepresented because Launchpad is heavily used by Ubuntu distribution maintainers, not developers of individual applications.

```
fork child
setjmp(loc)
wait(...) blocks
            <−− child exits
wait(...) returns
            <−− signal
longjmp(loc)
wait(...) error! no child
```

Figure 2: *dash-MySQL race.*

```
fd = open(H, RDONLY);
read(fd, buf, ...);
close(fd);
... // update buf
fd = open(H, WRONLY|TRUNC);
read(fd, buf, ...);
close(fd);
```

Figure 3: *bash race.*

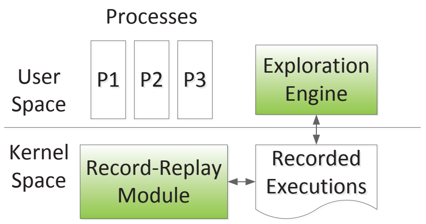

Figure 4: RACEPRO *Architecture.* Its components are shaded (and in green).

ware, and user configurations. Moreover, many of the sampled races (*e.g.*, all races described in §2.2) occur only due to rare runtime factors, for example, when a signal is delivered right after a child process exited (Race 2 in §2.2) and when a database shutdown takes longer than usual (Race 1 in §2.2). These races illustrate the advantage of checking deployed systems: we can rely on real users to create the diverse configurations and executions for us to check.

Process race patterns. Classified based on the causes, the sampled process races fall into two categories: atomicity violations or execution order violations.[4] Of the 42 process races, 37 are execution order violations, and only 4 are atomicity violations. These patterns suggest how we may detect process races.

2.2 Examples

Race 1: Upstart-MySQL. The symptom is that `mysqld` is forcibly killed during shutdown, resulting in a corrupted file system. This race is an execution order violation when `S20sendsigs`, the shutdown script that terminates processes, does not wait for MySQL to cleanly shutdown, before proceeding to file system unmount scripts. Its occurrence requires a combination of many factors, including the mixed use of System V initialization scripts and Upstart, a misconfiguration so that `S20sendsigs` does not wait for any daemons started by Upstart, insufficient dependencies specified in MySQL's Upstart configuration file, and a large MySQL database that takes a long time to shut down.

Race 2: dash-MySQL. The symptom is that the shell wrapper `mysql_safe` of the MySQL server daemon `mysqld` goes into an infinite loop with 100% CPU usage after a MySQL update. This race is an atomicity violation in *dash*, a small shell Debian uses to run daemons [2]. This race occurs when `dash` is interrupted by a signal unexpectedly. Figure 2 shows the event sequence surfacing this race. To run a background job, `dash` adds the job to an internal job list and forks a child process to carry out the job. It then calls `setjmp()` to save an execution context and and `wait()` to wait for the child to exit. After the child exits, `wait()` returns, and `dash` is supposed to remove the child from its job list. However, if a signal is delivered at this time, dash's signal handler will call `longjmp()` to go back to the saved context, and the subsequent `wait()` call will fail because the child's exit status has been collected by the previous `wait()` call. However, dash still has a nonempty job list, thus it calls `wait()` repeatedly to wait for the nonexistent child to exit. Although this race is in `dash`, it is triggered in practice by a combination of dash, the `mysql_safe` shell wrapper, and `mysqld`.

Race 3: Mutt-OpenOffice. The symptom is that OpenOffice displays garbage when a user tries to open a MS Word attachment in the Mutt mail client (though the attachment is not lost). This race is an execution order violation when Mutt prematurely deletes a file before OpenOffice uses this file. Its occurrence requires a combination of Mutt, OpenOffice, a user configuration entry in Mutt, and the `openoffice` shell wrapper. Specifically, the user first configures Mutt to use the `openoffice` wrapper to open MS word attachment. Then, to show an MS Word attachment, Mutt saves the attachment to a temporary file, spawns the configured viewer in a process, and waits for the viewer process to exit. The `openoffice` wrapper simply spawns the actual OpenOffice binary and exits at once; Mutt mistakes this exit as the exit of the actual viewer, and deletes the temporary file holding the attachment.

Race 4: bash. This race is an atomicity violation: `bash` writes to `.bash_history` without synchronization, allowing this file to be corrupted when multiple shells write concurrently. When `bash` appends to the history file, it correctly uses `O_APPEND`. However, when it exits, it reads back the history file and overwrites it to keep the history file under user-specified size. This problematic sequence of system calls is shown in Figure 3. When multiple `bash` processes exit at the same time, for instance, when a user exits `emacs` with multiple `bash` processes, the history file may become corrupted.

3 RACEPRO Design

Figure 4 shows the architecture and Figure 5 the work flow of RACEPRO. It has two main components: a kernel module for recording and replaying executions and a user-space exploration engine for systematically checking recorded executions.

[4] An execution order violation [10] occurs when a set of events must occur in a fixed order, but no synchronization operations enforce so.

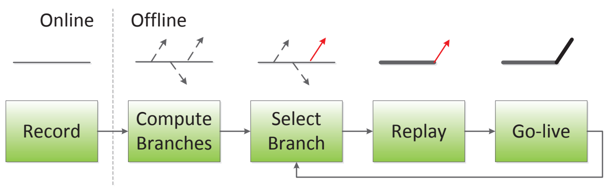

Figure 5: RacePro *Work Flow.* Thin solid lines represent recorded executions; thick solid lines represent replayed executions. Dotted arrows represent potentially buggy execution branches. The solid arrow (in red) represents the branch RacePro selects to explore.

The record-replay module implements four low-level mechanisms used by the exploration engine: (1) recording all interactions, such as system calls and signal delivery, between a group of processes and the kernel; (2) tracking what operations access what shared resources; (3) deterministically replaying a recorded execution; and (4) making a replayed execution go-live by resuming live execution.

The exploration engine implements three algorithms: (1) detecting when a machine becomes idle, so that it can switch RacePro from recording mode to checking mode; (2) computing the set of execution branches that may lead to process races using the accesses tracked by the record-reply module; and (3) checking whether these execution branches do lead to process races using the replay and go-live mechanisms.

RacePro provides a generic algorithm for detecting potential process races, which can be extended by refining what system calls may conflict. This algorithm need not give accurate results, because for each potential process race, RacePro validates that it is a real race by creating a real execution and verifying that the race does occur and lead to a failure [11].

To identify failures, RacePro can use three strategies. First, it can detect generic failures such as process crashes and error messages in system logs. Second, it can run user-written checks, for instance, a script that detects a corrupted `.bash_history`. Third, it can run existing "reader" programs (*e.g.*, `fsck`) on the result and detect if the result is different from that of the recorded execution or an execution in which all operations are serialized.

While our initial design of RacePro records and checks executions on the same machine, we intend to build a distributed checking infrastructure so that RacePro can migrate recorded executions to other machines for checking. New challenges and tradeoffs naturally arise in this distributed setting, which we will investigate.

4 Preliminary Implementation

We have implemented a preliminary RacePro prototype in Linux. Its record-replay module leverages our previous work on lightweight OS-level deterministic replay on multiprocessors [9]. This module is application-transparent, does not require changing, relinking, or recompiling applications, libraries, or the kernel, does not require any specialized hardware support, and does not require a VMM nor incur its associated costs. To record the execution of a multiprocess and multithreaded application, the module records all nondeterministic interactions between the application and the OS. To be able to go live at any point during replay needed for in-vivo model checking, it faithfully replays the effects of the system calls on the kernel, because replaying just the user-space effects is not enough.

Our RacePro prototype detects potential process races as follows. Given two events in a recorded execution, it first determines if they are concurrent. Intuitively, it checks that no prior dependencies make one event always happen before the other. It considers two concurrent events "racy" if (1) both operate on the same file and at least one modifies the file or (2) for catching signal races, one sends a signal and the other may be interrupted by this signal.

If a pair of racy events runs in one order during recording, RacePro considers the opposite order a possible execution branch. To check this branch, it uses the replay and go-live mechanisms to force the opposite order. Note that this branch may not lead to any failure or this branch may not even be feasible because RacePro's race detection is imprecise. However, RacePro reports this race only when it can create a real failure, thereby avoiding false positives.

We are continuing the development RacePro. Currently it does not detect when the machine is idle, and we currently switch from recording to checking manually. Its race detection algorithm can be made more precise and extended to other shared resources. It switches to random exploration of execution branches when the number of them goes beyond a threshold (1000 by default). We will address these issues in our future work.

5 Preliminary Results

Record and Replay Overhead. Low recording overhead is crucial because RacePro runs with deployed systems. Low replay overhead is desirable because RacePro can check more execution branches within the same amount of time. To evaluate RacePro's record and replay overhead, we applied it to a wide range of real applications on a 4-CPU multiprocessor. These applications include (1) server applications such as Apache in multi-process and multi-threaded configurations, MySQL, an OpenSSH server, (2) utility programs such as SSH clients, make, untar, compression programs such as gzip and lzma, and a vi editor, and (3) graphical desktop applications such as Firefox, Acrobat Reader, and MPlayer. Our results show that RacePro's recording overhead was under 2.5% for server and 15% for desktop applications. Replay speed was in all cases at least as fast as native execution and in some cases up to two orders of magnitude faster. This speedup is particularly useful for enabling rapid model checking.

Error Detection. Our RACEPRO prototype has successfully detected the `dash` and the `bash` races described in §2.2. Our methodology is as follows. To create an initial execution for RACEPRO to record, we downloaded the shell script contained in a bug report. Then, to show that RACEPRO can detect errors by exploring execution branches despite the fact that the recorded execution succeeds, we modified this script so that it no longer triggered the bug when we ran it manually.

For the `dash` race, our script forks two children; one of them sends a signal to the parent and the other simply exits. We recorded one execution of this script with RACEPRO, which included 277 system calls and 6 processes (some processes are created by the shell to run commands). However, RACEPRO computed only 2 branches because the signal had to be delivered after the corresponding child was forked. By exploring these branches, RACEPRO successfully created a real execution that looped infinitely. We detected this infinite loop using a timeout of one second.

For the `bash` race, our script forks two `bash` shells which run five commands and then exit at the same time. This script triggered a total of 1588 system calls and 13 processes. RACEPRO identified 132 potentially buggy execution branches, among which 115 branches corrupted the history file. We detected this corruption by comparing the number of lines of the history file to a history file where the two shells run sequentially.

6 Related Work

As discussed (§1), existing work on thread or TOCTOU races (*e.g.*, [8, 13]) may not be able to deal with the scope and the coverage challenges of process races. In this section, we discuss other closely related work.

Determinator [5] advocates a new, radical programming model that converts all races, including thread races and process races, into exceptions, to achieve pervasive determinism. This programming model is not designed to be backward-compatible; it deals only with low-level read-write or write-write races, not high-level atomicity and order violations. RACEPRO differs because it is designed to automatically detect general process races in legacy systems.

Several tools can also check deployed systems. CrystalBall [12] detects and avoids errors in a deployed distributed system using an efficient global state collection and exploration technique. Porting CrystalBall to detect process races is difficult because it works only with programs written in a special language, and it does checking synchronously while the deployed system is running, relying on network delay to hide the checking overhead. In-vivo testing [7] uses live program states, but it focuses on unit testing and lacks the systematic exploration and multiprocess support of RACEPRO.

7 Discussion and Future Directions

We have described our initial study of process races and our initial work towards effectively detecting them. Our immediate future work is to strengthen our study by studying more process races and studying them more thoroughly, to complete our RACEPRO prototype, and to detect numerous process races in real systems.

Detection of process races is only the first step. More research questions arise. For instance, how do we help developers diagnose process races RACEPRO detects? Given an execution where a process race surfaces, developers still have to figure out the cause of the race. Fixing it can take time, and before developers produce a fix, systems remain vulnerable. Thus, how can process races be temporarily worked around before developers fix them? Or, more aggressively, how can process races be fixed automatically? We hope that our initial work will inspire others to work on these research directions.

Acknowledgments

This work was supported in part by NSF grants CNS-1054906, CNS-1012633, CNS-0914845, and CNS-0905246, AFRL grants FA8650-10-C-7024 and FA8750-10-2-0253, and AFOSR MURI FA9550-07-1-0527.

References

[1] All resource races studied. `http://rcs.cs.columbia.edu/projects/racepro/`.

[2] The Debian Almquist Shell. `http://gondor.apana.org.au/~herbert/dash/`.

[3] Launchpad software collaboration platform. `https://launchpad.net/`.

[4] Upstart, an event-based replacement for System V init scripts. `http://upstart.ubuntu.com/`.

[5] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[6] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, June 2008.

[7] M. Chu, C. Murphy, and G. Kaiser. Distributed in vivo testing of software applications. In *1st IEEE International Conference on Software Testing, Verification, and Validation*, April 2008.

[8] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 237–252, Oct. 2003.

[9] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '10)*, June 2010.

[10] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Thirteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '08)*, Mar. 2008.

[11] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI '08)*, June 2008.

[12] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the Sixth Symposium on Networked Systems Design and Implementation (NSDI '09)*, Apr. 2009.

[13] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Oct. 2005.