Reproducible Concurrency for NPTL based applications

by

Praveen Srinivasan

B.Tech. (Information Technology), University of Madras, Madras, India, 2002

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science

Dr. Jerry James, Chair

Dr. Douglas Niehaus, Member

Dr. David Andrews, Member

Date Thesis Accepted

Dedicated to my parents

Acknowledgments

I would like to express my gratitude to Dr. Jerry James, my advisor and committee chair, for giving me the opportunity to work on this project and providing me excellent guidance throughout the course of the project.

I would also like to thank Dr. Douglas Niehaus and Dr. David Andrews for serving as members of my project committee. Special thanks to Dr. Douglas Niehaus and his research assistants for having developed and maintaining Datastreams without which this project would not have been completed in time. I would like to thank Tejasvi for helping me on numerous occasions and Ashwin for developing a post-processing filter in short notice.

I would like to take this opportunity to thank my mom for taking bold decisions to help me realize my academic dreams. I would also like to thank my sister and her family for their love, support and encouragement.

I would also like to thank Marilyn Ault and my fellow colleagues at ALTEC for supporting me throughout my graduate studies. Finally, I would like to thank my roommates and friends for making my stay in Lawrence fun-filled and memorable.

Abstract

Single-threaded programs are conventionally debugged by tracing the program's execution repeatedly until the bug is found. The debugging process is based on the assumption that the repeated executions of the program are identical. Multi-threaded programs, however, cannot be traced in such a manner because the execution path that the program takes is affected by several non-deterministic factors and, therefore, could be different for each run. This project provides a framework to reproduce the execution of Linux Pthread applications by controlling the non-determinism due to scheduling.

The framework has two phases: recording and replay. The context switch events of the threads, occurring in the kernel, are logged with essential data to produce a schedule order during the recording phase. The program's execution is then reproduced in GDB using the recorded schedule order during the replay phase.

Contents

List of Figures

List of Programs

Chapter 1

Introduction

Some software applications require a threaded programming model by nature. For example, a web server requires a manager thread to handle incoming connections and several worker threads for servicing each successful connection. The correct behavior of such multi-threaded (MT) programs is to produce the same result with same input for repeated executions. However, the execution path taken to produce the result could be different for each run. Concurrency control techniques offered by thread libraries limit the number of such valid execution paths. But, improper use of concurrency control techniques could lead to bugs that show up only for a certain set of execution paths. The thread libraries, in general, do not provide information about the execution trace which leaves the developers guessing about the interleaving order that resulted in the bug.

Developers generally follow cyclic debugging techniques, which is nothing but tracing a program's execution repeatedly, to find out the cause for bugs. This process is based on the underlying assumption that repeated executions of a program are always identical. Though it is true for single-threaded programs (if inputs supplied are the same), the case is not same for multi-threaded programs because a program's execution cannot be replayed in a deterministic way. A program can deviate from the path that it took during the previous run due to several non-deterministic events. These non-deterministic events have to be recorded and should be used to replay an execution of a MT program in an identical fashion for repeated runs so that cyclic debugging techniques and hence the features available in current debuggers can be used.

The factors that affect an execution path of a MT program are scheduling decisions, signals and I/O. The goal of this project is to reproduce an interleaving order for MT programs by recording and replaying scheduler events. The interleaving order provides us information about the order in which the threads executed. This information is necessary, though not sufficient, to reproduce an execution path. The execution path itself can be reproduced only when signals and I/O, in addition to scheduling decisions, are recorded and triggered at appropriate times. Furthermore, the scope of this project is limited to MT programs using POSIX thread library on uni-processor systems running Linux.

1.1 Thread Library Models

Thread library is an important component of a multi-threaded application. It provides a programming interface that is used by developers. In general, thread libraries can be classified as M:1, 1:1 or M:N.

In a M:1 thread library model, all user threads are mapped to a single kernel process. The thread library should therefore provide a scheduler that determines the order in which the threads are scheduled or mapped to the kernel process. In a 1:1 model each thread is mapped to a process. In this case, the library can choose to be simple and thin because the kernel already handles the scheduling job. In the last case several threads are mapped to one or more processes which, therefore, requires a scheduler component in the thread library. Bthreads [10], Pthreads and Next Generation POSIX Threads (NGPT) [2] are examples for M:1, 1:1 and M:N models respectively.

1.2 Pthread (NPTL)

Linuxthreads is a popular thread library that offers POSIX thread implementation to Linux developers. It is a 1:1 thread library where every thread created in user space is mapped to a kernel process. Even though it is viewed as a process in kernel mode, a thread does not have its own address space .Instead, it shares its address space with other threads created by a parent process. As expected of a 1:1 thread library, Linuxthreads takes advantage of the scheduling offered by the kernel and does not have a scheduler of its own.

Native POSIX Thread Library (NPTL) [6] is the latest implementation of POSIX threads on Linux which will eventually replace Linuxthreads. NPTL requires significant support from the kernel. So it is advisable to run NPTL on kernel versions 2.6 and higher. The new implementation addressed several issues like faster thread creation and destruction, futexes etc which were considered to be shortcomings in the earlier implementation.

1.3 Proposed Solution

The goal of this project can be accomplished in two steps.

- First, a log of time-stamped scheduler events have to be generated.

  Just recording scheduler events when they occur is not enough. Additional information about threads that are involved in the context switch and a valid address in user mode have to be logged to precisely know when to interrupt an execution of a thread during replay. DataStreams Kernel Interface (DSKI) [3] is a framework that can be used to log such events and additional data (related to performance and control-flow) from the kernel. This topic has been dealt extensively in Chapter 3.

- Second, the program replay has to be guided using recorded information.

  The recorded information can be used to set replay breakpoints at appropriate addresses in thread execution path. When such breakpoints are hit, the context has to be forced to the next thread in the recorded interleaving order. The GNU debugger, GDB already has several key features to support deterministic replays for NPTL. Chapter 4 provides more details about how GDB has been extended to support deterministic replays.

1.4 Document Organization

In the next chapter, we discuss some related work that has already been done in this area. The design and implementation details for recording and replay are discussed in chapters 3 and 4 respectively. We then discuss the evaluation procedures that have been used to test the proposed solution in chapter 5 and conclude with possible extensions to this project in chapter 6.

Chapter 2

Related Work

Ronsse, Bosschere and Kergommeaus[12] classify the approaches that address the execution replay problem into two main categories: content-based and data-based. Content-based techniques record values of shared variables whenever they are modified and force threads to read the recorded values during replay. Order-based approaches depend on recording the order in which threads executed and forcing them to execute in the same order during replay. Since our approach falls into the latter category, we present some order-based approaches already available and compare and contrast them against ours. We believe, to our knowledge, that this is the first project that is directed towards developing a record/replay system for POSIX threads on Linux.

Ronsse, Bosschere and Kergommeaus[12] also provide an exhaustive list of references for both content-based and order-based approaches that can be reviewed for more information.

2.1 DejaVu

DejaVu [4] is a modified JVM that supports execution replay for multi-threaded Java programs. The authors identify synchronization events that could affect the order in which threads access shared variables and refer them collectively as critical events. The JVM increments a global clock every time a critical event is executed (in an atomic fashion). A logical thread interval is then defined as `<a, b>` where `a` and `b` are equal to the global clock value when a critical event was executed for the first time and last time during that physical interval for that thread. An ordered set of such logical schedule intervals form a logical thread schedule which uniquely determines the execution path taken by a program. The recorded logical thread schedule is then used to achieve deterministic replays. In essence, the system records the order of synchronization events and forces the same order during replay.

Though the approach is independent of underlying operating system, more tracing information (than required) is generated and additional computation has to be performed to identify logical thread schedules as opposed to physical thread schedules. Moreover, the global clock has to be incremented for each shared memory access in an atomic way which introduces additional overhead. The execution time could increase considerably if the number of shared memory accesses for a MT application is high.

2.2 JaRec

JaRec [7] is also a framework which can be used to replay multi-threaded Java programs deterministically. JaRec, similar to DejaVu, records the order in which threads executed synchronization sections and forces the same order during replay. However, the authors have chosen not to modify the JVM (if it supports JVM Profiler Interface) for portability concerns. Instead, they instrument the Java classes through a Profiler Agent just before the Java classes are loaded into JVM. The synchronization instructions viz. `monitorenter` and `monitorexit` are wrapped to do actions related to recording/replay.

During the recording phase, a logical clock value is updated every time a thread performs a synchronization operation. This value is then written to a trace file for the corresponding thread. During replay, a thread reads the logical clock values one by one from its trace file and is forced to wait till all threads with lower clock values have completed their synchronization operations. Though the thread order is maintained, the additional delay caused due to the forced wait slows down the replay.

2.3 Record/Replay for Bthreads

Ramanasankaran [11] has developed a record/replay framework for Bthreads. Bthreads [10] is a user-level thread library based on Reactor, an event demultiplexing framework. Both the thread library and the underlying Reactor framework are instrumented to trigger events whenever a context switch occurs, a signal is delivered or an I/O system call is made (the I/O system calls are wrapped for this purpose). The program's execution is then controlled and reproduced in GDB using the recorded information.

The framework provides a way to control all sources of non-determinism and can reproduce the execution path of a MT program deterministically. However, since the thread library is not yet popular among developers, applications should be ported to Bthreads library to make use of this framework.

2.4 RecPlay

RecPlay [13] is a tool that allows developers to use cyclic debugging techniques to debug multi-process applications under Sun multiprocessor systems. This tool also records the order in which synchronization operations are carried out and forces the same order during the replay. However, data races (data races occur when synchronization mechanism is not used to protect access to shared variables), if present in the program, could make a replay unreliable as the information about the order in which the shared memory was accessed is lost. Hence this approach requires race detection during replay which slows down the debugging process considerably.

Chapter 3

Recording

The execution path of a MT program can be deterministically reproduced if

- the order in which the threads interleaved is recorded
- the inputs given to the program are recorded (this includes all network, file and system input)
- the places at which the signals were delivered are recorded.

For better understanding, consider a simple program 3.1 where a main thread creates another thread and both of them perform a simple arithmetic on a shared variable.

The output of program 3.1 could be different for each execution based on the order in which the threads accessed the shared variable. Consider the execution in figure 3.1, where thread 1 executes till line 14[*]. Thread 2 runs next, executes line 7 and exits from the system. Thread 1 is scheduled again and it completes the rest of the instructions (16, 18 and 20) and exits from the system. In this case the value of `shared_var` printed in line 18 is 8.

Consider another execution in figure 3.2, where thread 1 executes till line 18 followed by thread 2 and thread 1 again. In this case the value of `shared_var` is printed as 3.

[*]The line numbers are considered to be actual instruction addresses for the sake of simplicity

Program 3.1 A simple program to illustrate the importance of recording interleaving information.

```
 1: #include <stdio.h>
 2: #include <pthread.h>
 3:
 4: int shared_var = 0;
 5:
 6: int thread_func(void) {
 7:  shared_var += 5;
 8: }
 9:
10: int main() {
11:
12:  pthread_t t1;
13:
14:  pthread_create(t1, NULL, thread_func, null);
15:
16:  shared_var += 3;
17:
18:  printf("In main: shared_var is %d", shared_var);
19:
20:  return 0;
21: }
```

In order to reproduce the program's execution in a manner identical to its previous run, the order in which the threads were scheduled and the last instruction they executed in each schedule should be recorded.

For the execution in figure 3a, this would be

```
<thread 1, 14>
<thread 2, exit>
<thread 1, exit>
```

Similarly, for the execution in figure 3b, the information that has to be recorded looks something similar to

```
<thread 1, 18>
<thread 2, exit>
<thread 1, exit>
```

This interleaving information called schedule order (SO) hereafter, can be used to uniquely identify and, therefore, reproduce the execution path during replay.

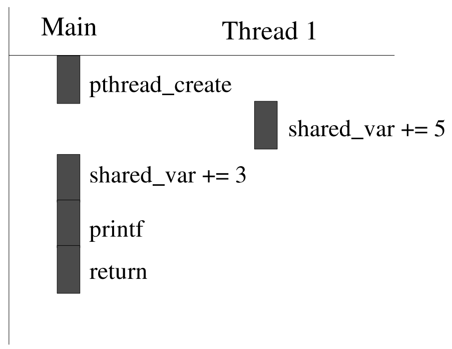

Figure 3.1: Execution Sequence 1

Consider a small modification to program 1 above. An if construct in line 17 checks if a command line argument is provided just before printing the value of `shared_var`. If so, the value of `shared_var` is printed.

For program 3.2, a schedule order such as `<thread 1, 18>, <thread 2, exit>, <thread 1, exit>` is necessary but not sufficient to reproduce the execution path. Because there is no assurance that thread 1 will execute line 18 until the inputs are recorded and fed at appropriate places during replay. Signals in addition to inputs, can deviate an execution path and therefore have to be recorded whenever they are delivered to a MT program.

Hence, a schedule order can be used to determine an execution path if and only if inputs and signals are recorded and re-fed (or re-delivered) at appropriate times during replay. However, the scope of this project is limited to recording the schedule order and forcing the replay to follow the same order.

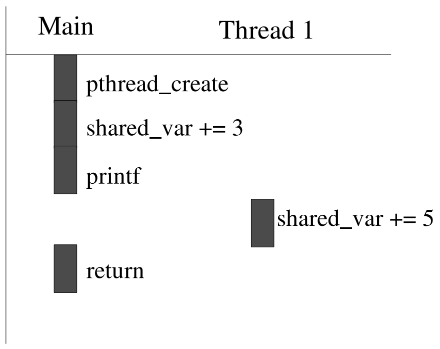

Figure 3.2: Execution Sequence 2

Program 3.2 A simple program to illustrate the importance of recording inputs

```
#include <stdio.h>
#include <pthread.h>

int shared_var = 0;

int thread_func(void) {
 shared_var += 5;
}

int main(int argc, char **argv) {

 pthread_t t1;

 pthread_create(t1, NULL, thread_func, null);

 shared_var += 3;
 if(argc > 1)
 printf("In main: shared_var is %d", shared_var);

 return 0;
}
```

3.1 Background

As discussed in chapter 1, the NPTL implementation (similar to LinuxThreads implementation) of POSIX threads follows a 1:1 model. Threads created using NPTL thread library have individual `PID`s, user and kernel mode stacks, but share the address space with other threads. The following are some of the points that are relevant to the design of recording framework.

- Threads are created using `pthread_create` API which finally makes the `clone` system call. The kernel then creates a new process, but keeps track of its parent using a variable called `TGID` in the `task_struct`(PCB). Normally, the `TGID` of all "real" processes is equal to their respective `PID`s. But, for threads created using `clone` system call, `TGID` is equal to `PID` of the parent process.
- Since each thread is a process by itself in the kernel, the kernel treats them as any other process as far as scheduling is concerned. However, individual threads of a MT application do not show up when one examines the processes in the system using `ps` command.
- Threads (like any other process) enter the kernel mode either due to a system call or when the system receives an interrupt. The threads could be context switched when they are in the kernel mode.

3.2 Schedule Order

Figure 3.3 shows an execution sequence of a program that has two threads. The horizontal axis denotes the timeline. As it can be seen in the figure, the threads can either be in user or kernel mode at any point of time. The points where the threads transition to the kernel mode are denoted by labels that start with T. This would be T11, T12, T13 etc. for thread 1 and T21, T22, T23 etc. for thread 2 and so forth. As mentioned above, a context switch could occur during those transitions and the points where it happens is denoted by labels that start with C. The points at which the threads resume execution in user mode is denoted by labels that start with R.

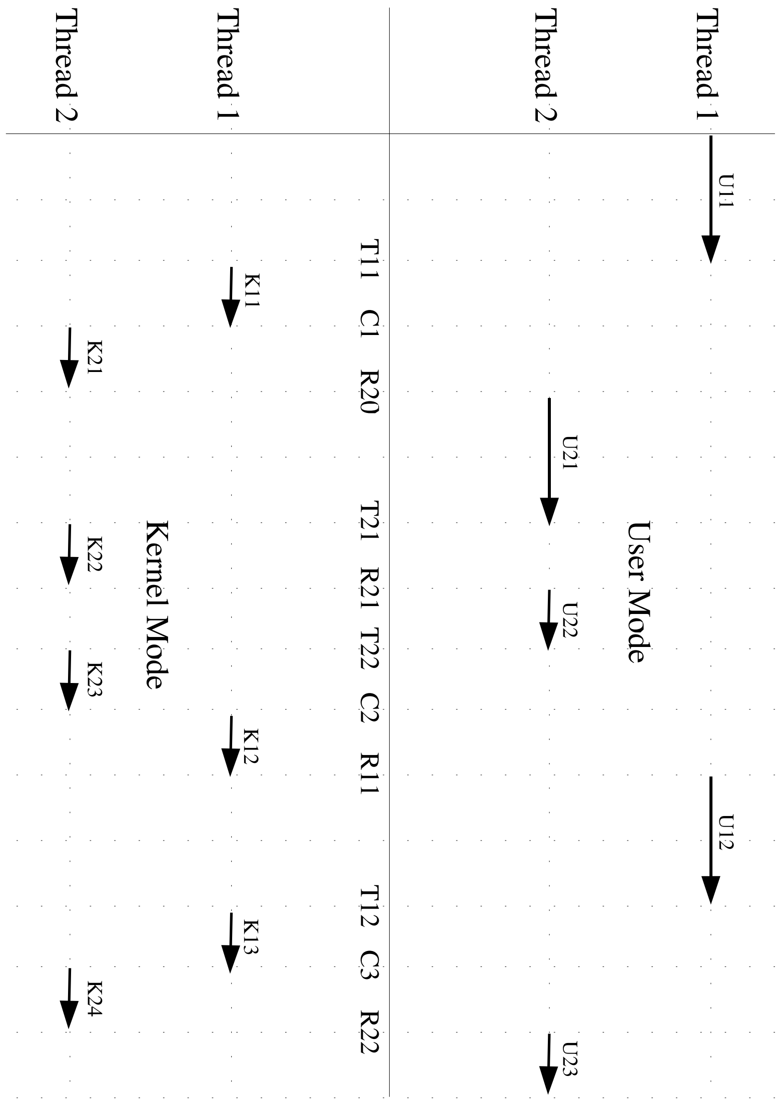

Figure 3.3: A detailed execution sequence

3.2.1 Transition and Resumption points

A transition point is nothing but the address of last executed instruction in user mode. Similarly a resumption point is the address of an instruction to be executed next when a thread resumes execution in user mode. Program 3.3, a part of assembly code generated by an `objdump` of C library, shows the transition and resumption points for a `fork` system call.

Program 3.3 An objdump of libc.so to illustrate transition and resumption addresses

```
79239:      ba 01 00 00 00          mov    $0x1,%edx
7923e:      b9 01 00 00 00          mov    $0x1,%ecx
79243:      89 f0                   mov    %esi,%eax
79245:      87 fb                   xchg   %edi,%ebx
79247:      cd 80                   int    $0x80
79249:      87 fb                   xchg   %edi,%ebx
7924b:      90                      nop
7924c:      8b 45 cc                mov    0xffffffcc(%ebp),%eax
```

The instruction at `79243` moves the system call number to the accumulator and the instruction at `79247` (`int 0x80`) generates a software interrupt to let the kernel know that a system call is being made. So, in this case, address `79247` is the transition address and address `79249` is the resumption address. It is also evident that transition and resumption addresses always point to consecutive instructions.

There are some instances where a breakpoint has to be set at a transition address and some instances where it should be set at a resumption address during replay. If a context switch occurs after a thread transitions to the kernel mode due to an interrupt, a breakpoint should be set at the resumption address. For most system calls it is reasonable to set a breakpoint at the transition address, because we do not want a process to enter the kernel mode. For system calls like futex, a thread may never return to user mode (and the resumption address) if the lock is held by another thread. However, we want some system calls like `clone` and `exit` to actually go through so that threads are actually created and destroyed. In such cases, breakpoints have to be set at respective resumption addresses.

It is, therefore, important that both transition and resumption addresses are recorded during a context switch. The schedule order for the execution sequence in figure 3.3 would now look like

```
<Thread 1, T11, R11>
<Thread 2, T22, R22>
<Thread 1, T12, R12>
```

Assuming that C1 in figure 3.3 happened after thread 1 made a transition (at T11) due to a system call (other than clone and exit) and C2 occurred after thread 2 made a transition due to an interrupt, the replay sequence would look like figure 3.4

When thread 1 reaches T11, it is stopped and GDB instructs the kernel to resume thread 2. As it is evident from the figure, the actions that thread 1 would have done in the kernel mode after T11 (K11) is delayed till GDB resumes thread 1 again. Due to this the overall execution sequence is different from the original one. The effects that this part of execution has on the overall application is therefore not reproducible and this problem is beyond the scope of this project. But the execution sequence in the user mode is reproduced exactly as shown in the figure 3.4.

3.2.2 System call wrappers

As discussed above, both transition and resumption addresses have to be recorded at the time of a context switch. Resumption points are nothing but return addresses stored in the kernel stack frame. So they can be easily obtained at the time of a context switch. Though a transition point is just the address of previous instruction, the length(in bytes) of the previous instruction cannot be determined automatically.

So we need another way to record a transition address (at the time of a context switch). We can exploit the fact that transition addresses are required only for system calls and wrap them to generate an event whenever the system call is being made. Also, we now need a way to distinguish between context switch events and events generated from system call wrappers (SYSCALL). The schedule order for the same sequence in figure 3.3 would now be

```
SYSCALL <thread 1, S11>
```

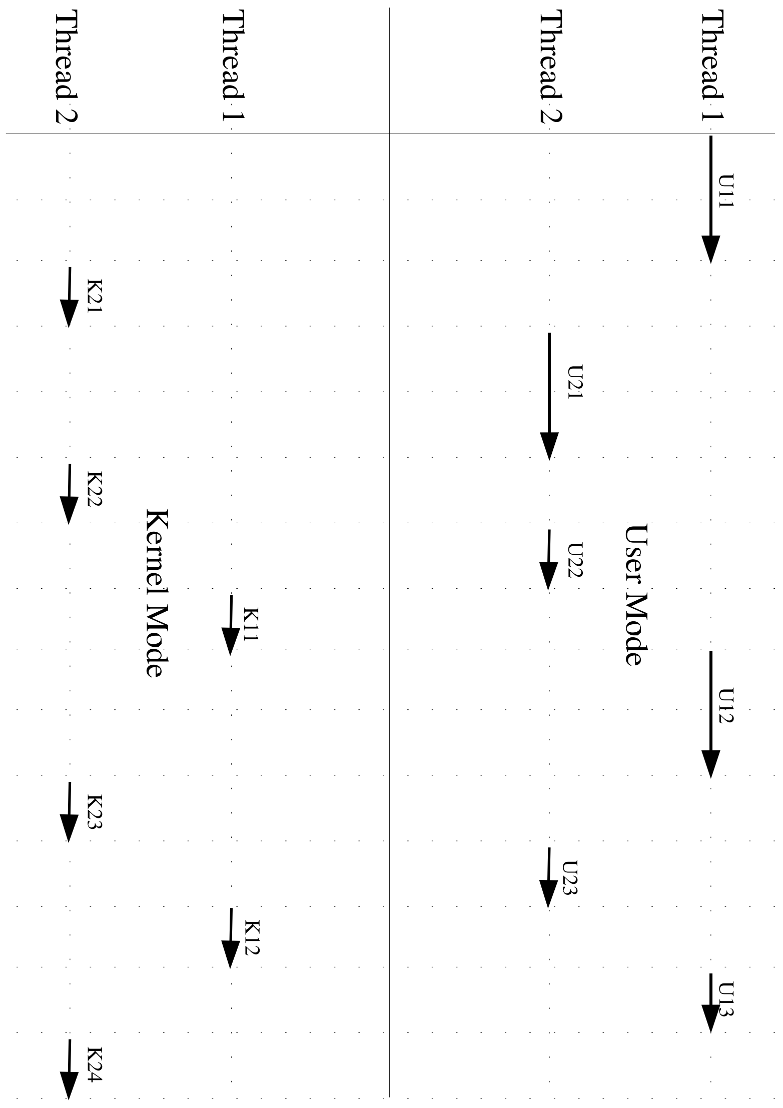

Figure 3.4: A detailed replay sequence

```
CS <thread 1, R11>
SYSCALL <thread 2, S21>
CS <thread 2, R22>
```

It should be noted that CS events no longer have transition address. It is generated by the system call wrappers instead. The information recorded above is by no means complete to replay the execution. Figure 3.3 indicates that there was no context switch after T21. But a system call event is generated just before the transition (assuming T21 is due to a system call). This SYSCALL event should be ignored during replay as the context switch did not happen during that transition. So we need a way to ignore system call events that did not result in context switch. This can be done by recording additional information with a context switch event which indicates whether the corresponding transition was due to an interrupt or a system call. The schedule order would now be

```
SYSCALL <thread 1, S11>
CS <thread 1, R11, SC>
SYSCALL <thread 2, S21>
CS <thread 2, R22, INT>
```

Using this additional information, all system call events that did not result in context switch can be ignored (in this case the second SYSCALL event is ignored as the CS following it is due to an interrupt). However, a method to obtain this important information (at the time of context switch) is currently unknown.

The current record/replay framework ignores system call events due to this reason. The replay framework, instead, uses a mapping file which contains transition addresses (previous instruction's address of a resumption address) of system calls that led to context switches. Chapter 4 discusses more about this issue. The rest of the discussion in this chapter will focus on recording just return addresses.

3.2.3 Basic Blocks

It is possible for any instruction to be executed more than once due to loops present in a program. Therefore, just knowing the address of an instruction is not enough to uniquely identify a transition or resumption address. It is actually determined by the `(address, count)` pair where `count` indicates the number of times the instruction at this address was executed before.

To determine the value of count, the concept of basic blocks can be used. A basic block is a set of assembly instructions that has a single point of entry and exit with no branches in between. The block profiling feature in GCC can be used to maintain a per-thread basic block count which can then be logged together with the resumption address when a context switch occurs. The modified schedule order would now look like

```
CS <thread 1, R11, basic block count>
CS <thread 2, R22, basic block count>
```

Summarizing the discussion above, the following things have to be accomplished to complete the recording framework.

- A per-thread basic block count has to be maintained.
- A context switch event has to be triggered every time the context is switched from and to a "pthread" process.
- A return address and a per-thread basic block count has to be recorded along with a context switch event.

The next few sections discuss about how these were implemented.

3.3 Maintaining Basic Block count

GCC (GCC 2.95 to be specific) has an in-built block profiler which can be used with a -ax option. It is normally used to find portions of code which can be optimized based on execution statistics. GCC organizes the resulting executable into basic blocks and assigns them a label and adds additional routines to do book-keeping. GCC adds a new function called `__bb_trace_func` in each basic block so that block profiling

Program 3.4 A sample objdump showing basic block profiling

```
8048b2a:       c7 45 fc 00 00 00 00    movl   $0x0,0xfffffffc(%ebp)
8048b31:       9c                      pushf
8048b32:       c7 05 b0 b2 04 08 01    movl   $0x1,0x804b2b0
8048b39:       00 00 00
8048b3c:       c7 05 b4 b2 04 08 80    movl   $0x804b180,0x804b2b4
8048b43:       b1 04 08
8048b46:       e8 7d 17 00 00          call   804a2c8 <__bb_trace_func>
8048b4b:       9d                      popf
8048b4c:       8b 45 f0                mov    0xfffffff0(%ebp),%eax
8048b4f:       8b 55 fc                mov    0xfffffffc(%ebp),%edx
8048b52:       3b 50 08                cmp    0x8(%eax),%edx
8048b55:       7c 20                   jl     8048b77 <infloop+0x83>
8048b57:       9c                      pushf
8048b58:       c7 05 b0 b2 04 08 02    movl   $0x2,0x804b2b0
8048b5f:       00 00 00
8048b62:       c7 05 b4 b2 04 08 80    movl   $0x804b180,0x804b2b4
8048b69:       b1 04 08
8048b6c:       e8 57 17 00 00          call   804a2c8 <__bb_trace_func>
8048b71:       9d                      popf
8048b72:       e9 99 01 00 00          jmp    8048d10 <infloop+0x21c>
8048b77:       9c                      pushf
8048b78:       c7 05 b0 b2 04 08 03    movl   $0x3,0x804b2b0
8048b7f:       00 00 00
8048b82:       c7 05 b4 b2 04 08 80    movl   $0x804b180,0x804b2b4
8048b89:       b1 04 08
8048b8c:       e8 37 17 00 00          call   804a2c8 <__bb_trace_func>
8048b91:       9d                      popf
8048b92:       c7 45 f8 00 00 00 00    movl   $0x0,0xfffffff8(%ebp)
8048b99:       8d b4 26 00 00 00 00    lea    0x0(%esi),%esi
8048ba0:       9c                      pushf
8048ba1:       c7 05 b0 b2 04 08 04    movl   $0x4,0x804b2b0
8048ba8:       00 00 00
8048bab:       c7 05 b4 b2 04 08 80    movl   $0x804b180,0x804b2b4
8048bb2:       b1 04 08
8048bb5:       e8 0e 17 00 00          call   804a2c8 <__bb_trace_func>
```

related data structures can be updated just after entering a basic block. The following piece of code shows `__bb_trace_func` being called upon the entry of each basic block.

Malladi[9] made modifications to the basic block profiling feature by registering a new function called `__bb_new_trace_func` which, in addition to doing the usual book-keeping work, updates a basic block count variable in the internal thread structure maintained by the Bthread library. This feature can be used by compiling a Bthread program with a "-at" option.

A similar approach has been followed to maintain a per-thread basic block count. A new variable called `bb_count` has been added to the internal thread data structure maintained by the pthread library. Two functions `pthread_incr_bb_count` and `pthread_get_bb_count` that increment and return the value of `bb_count` respectively have also been added to the pthread API. When a pthread program is compiled using the "-at" option, GCC inserts a call to `__bb_new_trace_func` upon entering a basic block. The `__bb_new_trace_func` in turn calls `pthread_incr_bb_count` function which increments the `bb_count` in pthread structure.

3.4 Recording kernel events using DSKI

Datastreams Kernel Interface (DSKI)[3] is a framework to collect status and performance related data from the kernel. Kernel components can be instrumented to trigger an event when a thread of execution reaches that point. Each instrumentation point can be uniquely identified by a Family and Event ID, where a family is a collection of related events. It also has support for counters and histograms. DSKI also has a device driver interface through which the user can choose to log certain subset of events for an experiment. Data is collected through the same device driver interface and is logged to a file in binary format which can then be post-processed.

The following DSKI macro has to be added at points where we want an event to be triggered.

DSTRM_EVENT(Family ID, Event ID, Tag, Extra Data length, Pointer to extra data)

As mentioned above Family and Event ID uniquely identify this instrumentation point. `PID` is usually passed as the third argument. The fourth and fifth arguments can be used to log additional data at every instrumentation point. Events are automatically time-stamped which can then be used to generate a chronological order.

DSKI already has several pre-defined families and events. But events like `SWITCH_FROM` and `SWITCH_TO` under `SCHEDULER` family are very useful for the functionality of this project. Event `SWITCH_FROM` is triggered whenever the scheduler chooses to interrupt the execution of a process and event `SWITCH_TO` is triggered whenever the execution of a process is resumed back. Section 3.5 discusses how these events (or a variation) have been used to implement the recording framework.

3.5 Identifying Pthread processes in the kernel

Identifying pthread processes in the kernel is very important to the functionality of the recording framework. Logging context switch events (`SWITCH_FROM` and `SWITCH_TO`) of all processes is not of much use since it is difficult to separate out events of a multi-threaded application during post-processing. Therefore, it is essential that pthread processes are identified in the kernel so that context switch events of only those processes can be logged. For this purpose, two new events called `PTHREAD_SWITCH_FROM` and `PTHREAD_SWITCH_TO` have been added. These events are triggered only when execution of a pthread process is stopped or resumed respectively.

A new `clone flag` called `CLONE_SET_BBCOUNT` has been added to both GLIBC and kernel for this purpose. The `clone` system calls from GLIBC are now made with this new flag turned on. The kernel, while processing the system call, checks for this flag and turns on a new status flag called `pthread_flag` in the `task_struct`, which serves for identifying pthread processes. But such an approach cannot be followed for a main thread because it is not created from a `clone` system call. The status flag for a main thread, therefore, has to be updated only when it creates a thread(using `clone` system call).

3.6 Accessing Basic block count in kernel mode

Return addresses have to be logged as `(address, basic block count)` pair during a context switch event, which requires the kernel to know the address that it has to fetch the basic block count from. A new variable that points to this address called `bb_count_address` has been created in the `task_struct`. The value of this variable is set to the address of per-thread basic block count variable maintained by the pthread library, during the `clone` system call (again, this is done only when `CLONE_SET_BBCOUNT` flag is on) for threads created by a main thread. For a main thread, the `bb_count_address` is set during the `set_thread_area` system call.

3.7 Fetching Return addresses

The next task is to fetch return addresses. It is important to note, at this point, that every process has both user and kernel level stack. Each process is allocated a kernel stack of certain size (which can be set during kernel configuration). The `task_struct` of a process itself is stored in this kernel stack and it occupies the lower addresses. The remaining memory is used to store stack frames and the stack grows downwards from higher to lower addresses.

When a process enters the kernel mode (either due to a system call or an interrupt), the user space return address is stored in first stack frame, which is usually called a trap frame. This frame can be accessed in the following manner.

```
t_regs = ((struct pt_regs *)
           (
             THREAD_SIZE +
             (unsigned long)current->thread_info)
           ) - 1;
```

where,

- `current` points to the `task_struct` of the process that is executing currently

- "`current− > thread_info`" points to the lower most address of the kernel stack of the process
- `THREAD_SIZE` is equal to the size of the stack
- `pt_regs` refers to the structure of a stack frame.

The (return address, basic block count) pair can then be recorded as extra data along with `PTHRED_SWITCH_FROM` event. It is enough to record just `PTHREAD_SWITCH_FROM` events to create a schedule order. `PTHREAD_SWITCH_TO` event was added just for testing purposes.

3.7.1 Virtual System Calls

The newer versions of linux kernel (2.6 and above) have support for virtual system calls [1]. They have been introduced to reduce the time it takes to switch from user to kernel mode during a system call. Under this new approach, a kernel page is mapped to every user process. This is evident by doing an `ldd` on any application. The following example shows the libraries that are linked to `ls` on a system that supports virtual system calls.

```
testbed62 [42] % ldd /bin/ls
        linux-gate.so.1 =>  (0xffffe000)
        librt.so.1 => /lib/tls/librt.so.1 (0x00535000)
        libacl.so.1 => /lib/libacl.so.1 (0x4118a000)
        libselinux.so.1 => /lib/libselinux.so.1 (0x4189c000)
        libc.so.6 => /lib/tls/libc.so.6 (0x006fd000)
        libpthread.so.0 => /lib/tls/libpthread.so.0 (0x00932000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x006e4000)
        libattr.so.1 => /lib/libattr.so.1 (0x410da000)
```

The first library, linux-gate.so.1, does not exist and it refers to the `vsyscall` page in the kernel memory. A process can now make a system call as though it makes a normal function call and , in such cases, the return address in the stack always points to an address on this page referred by the macro `SYSENTER_RETURN`. Logging this address is not of much use because breakpoints cannot be set at `SYSENTER_RETURN` during replay(because it belongs to the kernel memory).

To get a return address where a breakpoint can be set, the user level stack of the process has to be unwinded once, so that the return address of the function which actually made the virtual system call can be obtained. The way the user-level stacks are maintained is dependent on the underlying architecture and therefore, the rest of this section is i386 specific. The changes should be simple to port for other architectures.

Figure 3.5 [5] shows how user level stack frames are organized. In i386, the `ebp` register is used for storing frame pointers. Every stack frame has `ebp`, a pointer to previous stack frame, at the top. It is followed by `eip`, which is the return address for that stack frame.

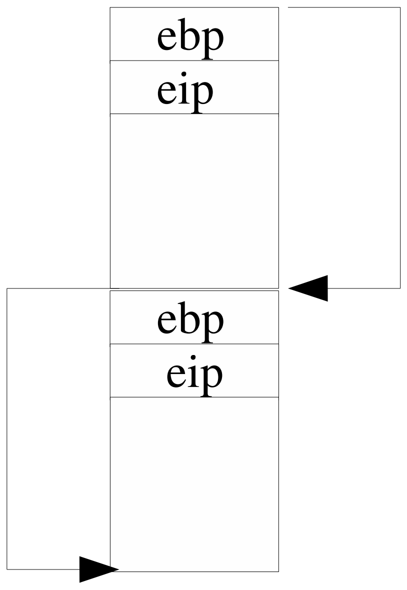

Figure 3.5: User Stack Organization

Therefore, the return address of a function which made the virtual system call can be obtained in the following way.

```
get_user(
  temp_eip,
  (unsigned long __user *)
  (t_regs->ebp + sizeof(unsigned long))
);
```

`tregs` points to the trap frame (see section3.7). ″$tregs- > ebp$″ points to the desired stack frame in the user stack and the address of the return address can be found by adding the size of one `unsigned long`.

3.8 Sample Schedule Order

Consider a simple multi-threaded program in Program 3.5. The main thread creates three threads and waits till all the threads exit after they count from 0 to 99. It then prints ″Hello World!″ and then exits.

The following is a part of the schedule order that was obtained while tracing a sample run of program 3.5 in XML format. The binary schedule order files can be converted to XML format (and vice versa) using Python scripts in the Datastream package.

```
<?xml version = "1.0" encoding="iso-8859-1" standalone="no"?>
<!DOCTYPE COMPOSITE_STREAM SYSTEM>
<COMPOSITE_STREAM>
<ENTITY number="0" tag="0" ... time_std="0" type="ADMINISTRATIVE_EVENT">
<EVENT name="EVENT_CPU_TIME_INFO" family="DSTREAM_ADMIN" id="9">
<EXTRA_DATA  format="base64">
IJBxQqCY2yFJvQK6Zd8D...
</EXTRA_DATA>
<EXTRA_DATA format="custom">
tv_sec=1114738720
tv_nsec=568039584
tsc=1090052935564617L
per_sec=0L
per_jiffy=0L
per_subjiffy=0L
```

Program 3.5 A simple multi-threaded program to illustrate schedule order

```
#include <stdio.h>
#include <pthread.h>

#define NUM_THREADS 3

void *count() {
        int i;

        for (i=0; i < 100; i++);

        return NULL;
}

int main() {

        pthread_t pt[NUM_THREADS];

        int i;

        for(i=0; i<NUM_THREADS; i++) {
                pthread_create(&pt[i], NULL, count, NULL);
        }

        // Wait for the threads to exit
        for(i=0; i<NUM_THREADS; i++) {
                pthread_join(pt[i], NULL);
        }

        printf("\nHello World\n");

        return 1;
}
```

```
16 </EXTRA_DATA></EVENT>
17 </ENTITY>
18 <ENTITY number="1" tag="2540" ... time_std="0" type="EVENT">
19 <EVENT name="PTHREAD_SWITCH_FROM" family="SCHEDULER" id="66">
20 <EXTRA_DATA  format="base64">
21 TIALQNDq/7846/+/AwAAAA==
22 </EXTRA_DATA>
23 <EXTRA_DATA format="custom">
24 from_eip(hex)=400b804c
25 bb_count=3
26 </EXTRA_DATA></EVENT>
27 </ENTITY>
28 <ENTITY number="2" tag="2541" ... time_std="0" type="EVENT">
29 <EVENT name="PTHREAD_SWITCH_FROM" family="SCHEDULER" id="66">
30 <EXTRA_DATA  format="base64">
31 GEcRQMQakkAcG5JAzQAAAA==
32 </EXTRA_DATA>
33 <EXTRA_DATA format="custom">
34 from_eip(hex)=40114718
35 bb_count=205
36 </EXTRA_DATA></EVENT>
37 </ENTITY>
38 <ENTITY number="3" tag="2540" ... time_std="0" type="EVENT">
39 <EVENT name="PTHREAD_SWITCH_FROM" family="SCHEDULER" id="66">
40 <EXTRA_DATA  format="base64">
41 TIALQNDq/7846/+/BgAAAA==
42 </EXTRA_DATA>
43 <EXTRA_DATA format="custom">
44 from_eip(hex)=400b804c
45 bb_count=6
46 </EXTRA_DATA></EVENT>
47 </ENTITY>
```

Lines 1 to 17 represent the datastream header. It is then followed by `ENTITY` tag which encapsulates events (counters, and histograms). Lines 18 to 27, 28 to 37 and

38 to 47 show details about three `PTHREAD_SWITCH_FROM` events. The `tag` argument in the `ENTITY` tag holds the `PID` (passed as third argument to `DSTRM_EVENT` macro) of the process for which the event was triggered. The extra data passed as the fifth argument is encapsulated within the `EXTRA_DATA` tag. The return address is displayed in hexadecimal format and the basic block count is shown in decimal format. The three events in the SO correspond to main thread (PID: 2540), first thread (PID: 2541) and main thread again. The return addresses correspond to `clone`, `exit` and `clone` system calls.

The next chapter discusses in detail about how the schedule order can be used to reproduce the execution path.

Chapter 4

Replay

This chapter provides details about the replay framework and how a schedule order can be used to reproduce an execution path in user mode. This can be done by executing the target program within the context of GDB and setting breakpoints at appropriate addresses. Section 4.4 provides more detail about this. Section 4.2 provides background information about GDB and existing support for thread debugging. Section 4.2 describes the intermediate steps that have to be completed before replay. Section 4.3 describes the changes that have been made to Insight to support deterministic replays.

4.1 Background

In GDB terminology, the program to be debugged is called the `inferior process`. GDB starts to trace the `inferior process` by first attaching to it using `ptrace` system call (`PTRACE_ATTACH` flag is passed along). GDB (or any process that traces) can then control the `inferior`'s execution and examine its stack, local and global variables, registers and other data structures. A sample debugging session of a single-threaded program, represented as a state machine is shown in figure 4.1. Though GDB has rich set of features, only those that are relevant to the scope of this project are discussed here.

GDB starts in state 1 where the `inferior`'s symbols have already been loaded and initial breakpoints have been set. A `run` command starts the `inferior` and puts GDB

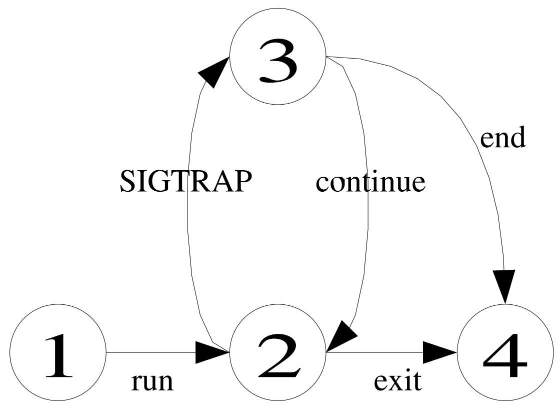

Figure 4.1: GDB State diagram

in state 2 where it waits for any `inferior`'s event to be reported. This event is usually a `SIGTRAP` (or `SIGINT` if a program performs something illegal) caused by hitting a breakpoint. At this point the `inferior` process is stopped and control returns to GDB which moves it to state 3 where the user can check the inferior's stack, variables etc. A `continue` command puts GDB back in state 2 (wait state) till a breakpoint is hit again. This process is usually referred as cyclic debugging. An user can also choose to `end` the debugging session which puts GDB in state 4 from state 3. A transition from state 2 to state 4 is also possible if the `inferior` exits. The basic system calls used are `wait` (`waitpid`) and `ptrace`. More information about these system calls can be obtained from their respective `man` pages.

4.1.1 Existing Thread debugging support

As stated in chapter 1, GDB simply follows the cyclic debugging process even for multi-threaded programs (which cannot be replayed deterministically). However, GDB already has some support for multi-threaded debugging which can be used to achieve deterministic replays.

GDB has a well structured framework through which support for any threading library can be incorporated. In addition to adding the target thread library to the GDB source tree, the thread library should also provide a debugger support library. For NPTL, the target is called `linux-nat` and the debugger support library is called `libthread_db.so`.

GDB inserts breakpoints at thread creation and exit addresses (the debugger support library helps GDB to find the addresses) and hence gets notified when those events occur. GDB also updates its internal thread list when such events occur. The threads can be identified using a variable called `threadnum` that is specific to GDB (it is not related to `PID`). It is usually assigned in the order in which threads are created. The main thread is 1, first thread created is 2 and so on. Users can then use the following thread specific commands by specifying the threadnum.

- break `address` thread `threadnum`:

  A thread specific breakpoint can be set by specifying a `threadnum`. A `SIGTRAP` is generated every time the breakpoint is hit, no matter which thread hits it, but GDB returns control to the user only when the specified thread hits it. In other cases GDB simply resumes execution of all threads. It is important to note, at this point, that execution of all threads are stopped by the kernel when any thread hits a breakpoint. As stated in chapter 3, the kernel maintains a `TGID` list which keeps track of threads created by a process. The kernel supports group stop and group exit feature through which the entire thread group can be stopped or killed respectively.

- thread `threadnum`:

  This changes the context from current thread that reported an event to a different thread specified by `threadnum`.

- set schedlock on:

This is one of the key features necessary to support deterministic replays. When this feature is turned on, instead of resuming all threads when a `continue` command is issued, GDB resumes only the thread that has context. The context can be changed to a desired thread using thread `threadnum` command mentioned earlier.

Section 4.4 describes how deterministic replay can be achieved using the schedule order (obtained during the recording phase) and a combination of commands mentioned above. But before moving to that section, it is important to understand the schedule order cannot be used as such. Some post-recording/pre-replay processing has to be done so that it can be used for replay. The next section describes the details.

4.2 Intermediate Steps

This section lists some intermediate actions that have to be done before replaying a program's execution.

4.2.1 Thread number mapping

Scheduler order is a binary file which contains all context switch events. But the `PID`s in those events cannot be used as such because threads will be assigned new `PID`s during replay. So a mapping procedure to identify threads becomes essential. For this purpose, a new instrumentation point has been added in the `copy_thread` function (which gets called by `sys_clone` eventually) of the kernel source tree. It triggers a `PTHREAD_THREAD_CREATE` event every time a pthread is created (As stated in Chapter 3 a pthread can be identified using the `CLONE_SET_BBCOUNT` flag that gets passed along with the `clone` system call) and logs the newly created thread's ID as extra data.

A post-processing filter (available with the datastreams package) then makes a pass through the schedule order file and replaces the `PID`s with number of thread create events processed. Main thread always creates the first thread. So, when a thread create event is processed for the first time, the `PID` of the main thread is assigned number 1 and the newly created thread is assigned number 2. Threads created thereafter are assigned numbers in increasing order. This intermediate phase essentially creates a mapping through which desired threads can be identified because the numbers assigned in this phase are going to match the `threadnum` values that GDB will assign during replay (see section 4.1.1). After post-processing, a new (binary) schedule order file with revised thread numbers is created.

This file can be specified as input file using the `replay_schedsgnl_file` command which has been created for this purpose. Ramanasankaran [11] created this command to achieve reproducible concurrency for BThreads. It has been modified to fit the requirements of this project.

4.2.2 Transition Address mapping

As stated earlier in section 3.2.1, breakpoints have to be set at transition addresses for most of the system calls. It also mentioned that a method to fetch it at the time of context switch is currently unknown. However, it can be found by going through the schedule order file (in XML format) and finding a corresponding transition address (if required, transitions caused by interrupts and some system calls do not require transition address) by looking at `objdump` files manually. A list of such mappings (return address, transition address) have to be saved as a file and specified as input to GDB so that it can set a breakpoint at the transition address instead of a return address it reads from the schedule order file.

A new command by name `syscall_address_file` has been added to GDB for this purpose. This command takes a transition address mapping file (the transition addresses are nothing but addresses where the system calls are being made from and hence the name `syscall_address_file`) as an argument and then builds a mapping array which can then be used during replay.

4.3 Clever Insight

Another feature that is required to support deterministic replays is the ability to attach a group of commands (including setting next replay breakpoint) to be executed when a breakpoint is hit (as described in the next section). GDB lacked the feature of nested breakpoints which was addressed by Ramanasankaran [11]. Furthermore, Clever Insight, which in turn is derived from SmartGDB [8], added the capability to attach TCL scripts to be executed when breakpoints are hit. Support for deterministic replay has been added in such a way that the feature can be used in both GDB and Insight.

4.4 Deterministic Replay Support

The schedule order can be used to achieve execution replays by setting replay breakpoints at appropriate transition or resumption points and forcing the context to a desired thread. Special conditions like thread exits have to be dealt specially. Program 4.1 describes the pseudo code for automatic (replay) breakpoint insertion process.

Program 4.1 Pseudo code for Automatic Breakpoint insertion

```
 1: Set breakpoint at main and start the inferior
 2: Wait for the inferior to report an event
 3: If Event = Thread Exit
 4:     Add threadnum to exited thread list
 5:     Goto 10
 6: Else If Event = Replay Breakpoint
 7:     Goto 10
 8: EndIf
 9: Continue Inferior

10: Read next schedule event from schedule order file
11: If threadnum is in exited thread list
12:     Goto 10
13: EndIf
14: Switch to threadnum
15: If breakpoint return address has a corresponding transition address
16:     Set Next Replay Breakpoint at Transition address
17: Else
18:     Set Next Replay Breakpoint at Return address
19: EndIf
20: Goto 9
```

Before setting a breakpoint at main, the transition address mapping and schedule order file have to specified as input. First a breakpoint is set at main and a group of commands that have to be executed when the breakpoint is hit is specified. Next, the inferior is started. When the breakpoint at main is hit, the command group is executed which contains command to set next replay breakpoint and this process continues till the inferior exits. There is a possibility that the schedule order can contain schedule events for exited threads. This is because GDB/Insight receives a thread exit event slightly earlier before a thread actually exits. Therefore it is important to maintain an exited thread list and update it when a thread exits so that schedule events for those `threadnum` values can be ignored. A sample group of commands that have to be executed when a replay breakpoint is hit is shown in Program 4.2

Program 4.2 A sample command group that is executed upon hitting a replay breakpoint

```
1: info threads
2: thread 1
3: break *1074174393 thread 1 if pthread_get_bb_count() == 68
4: commands
5: disable_last_breakpoint_hit
6: set_next_replay_breakpoint
7: end
8: continue
```

The `info threads` command in line 1 is necessary because, a breakpoint has to be set for a thread sometimes even before the thread creation event is reported in GDB. By executing this command, threads events can be detected even though it is not reported and not added to the internal thread list maintained by GDB. The command in line 2 makes thread 1 to be in current context. A replay breakpoint is inserted for thread 1 after reading the schedule order and transition mapping files. It is then followed by the set of commands that have to be executed when this breakpoint is hit. It starts with `commands` command in line 5 and is followed by a list of commands and is ended by `end` command. The commands specified in line 5 and 6 disable last replay breakpoint set and insert next replay breakpoint respectively. This file is then input to GDB through the `source` command which executes commands in line 1 through 3 and schedules commands in line 5 and 6 to be executed when the breakpoint in line 3 is hit. The `continue` command in line 8 is specified only when the replay is set to `auto` mode which is described in section 4.5.

4.5 Experimental Interleaving

The execution replay procedure can be controlled. When run in a controlled mode, control is returned to the user when a replay breakpoint is hit. The next replay breakpoint is already set before yielding the control to the user. The user can just continue an `inferior` at this point which will continue the execution replay process. Or the user can choose to experiment and switch to another thread that he/she wishes and achieve a new interleaving pattern. Thus the new feature also allows an user to experiment with new interleaving pattern in addition to achieving execution replay.

The next chapter describes the how the record/replay framework has been evaluated.

Chapter 5

Evaluation

The recording framework produces a schedule order file, which can be checked to see if return addresses and basic block counts that it recorded are valid. The replay framework is more obvious to evaluate because a program's execution has to match the execution in the recorded run. The evaluation procedures that have been used to test the framework are presented in the next few sections.

5.1 Evaluation of Recording framework

The recording framework has two major components that can be tested: basic blocks and return addresses.

5.1.1 Testing basic blocks

A new function, `pthread_get_bb_count`, has been added to the pthread library for this purpose. A program with 4 threads, each executing a for loop for a different number of times was written for this purpose. The difference between the basic block counts before and after the loop was printed. This matched the number of times the for loop executed in each of the threads.

5.1.2 Testing Return Addresses

The schedule order file can be converted into XML format using post-processing filters that come with the Datastreams package. The XML file can then be opened in an editor and the return addresses can be checked to see if they are valid. A quick method to do that is to open the executable in GDB and set a breakpoint at that particular address and see if GDB can actually insert a breakpoint at that location.

A simple program was written for this purpose and its execution was recorded. The return addresses were checked manually using the `objdump` tool. As expected, all context switches that occurred due to system calls were due to blocking system calls which proves the credibility of the recording process.

Finally, because of the nature of the problem being addressed, a successful execution replay indicates the success of the recording framework. The next section describes how the entire framework has been tested.

5.2 Evaluation of Replay framework

The credibility of the execution replay system can be tested by checking it against programs that have data races. Such programs produce different results for different executions. When an execution of such a program is recorded, the framework should produce the same result obtained during the recorded execution (for any number of times) if it is valid.

The framework was tested using simple multi-threaded programs that had data races. The output of the program under test was saved while recording the schedule order. When replayed using the framework it produced an output that was identical to the saved output.

As stated earlier, the framework requires a transition address mapping file that cannot be produced automatically. The user has to go through a tedious process of manually identifying the transition addresses as explained in section 4.2.2. For this reason, the framework has not been tested using test-suites and complex multi-threaded programs.

Chapter 6

Conclusions and Future Work

A record/replay framework has been implemented to achieve deterministic execution replays for NPTL based multi-threaded applications. Using this feature, developers can resort to cyclic debugging techniques and use wealth of features already available in GDB/Insight to debug multi-threaded programs. The following features/changes have been made to open source code-bases to achieve this.

- GCC and GLIBC code-bases have been modified to support per-thread basic block count.
- Small changes to GLIBC and the kernel have been made to identify pthread processes and to inform the kernel about the address where it can fetch basic block count from.
- New instrumentation points that trigger and record context switch events of pthread processes have been added to the kernel.
- The kernel source has been modified to record user-mode return addresses (from the kernel/user stack of a process) during the context switch events.
- Automatic breakpoint insertion feature and several new commands have been added to GDB to achieve deterministic execution replays.

The following are some pointers to additional work that are required to make this tool complete.

- As stated earlier, the transition addresses for some system calls have to be found manually. An alternative approach through which this can be automated was mentioned in section 3.2.2. This approach, however, requires the ability to distinguish context switch events that occur due to system calls or interrupts. Such an approach would completely eliminate the manual lookup that is required currently.

- Though pthread processes have been identified in the kernel, events of multiple applications could be recorded if they are running at the same time. A method to distinguish processes/threads of a particular application is therefore required to separate events. Progenitor feature, which is available with the KUSP source can be used this purpose with slight modifications. However, it has not been ported and tested to work with 2.6 kernels at the time of this writing.

- As stated in Chapter 3, system calls and signals have to recorded and re-fed at appropriate points during replay to deterministically replay the execution path.

- GCC's (2.95 version) block profiler feature has been used to maintain a per-thread basic block count for this project. Unfortunately, this feature has been removed and replaced with an edge profiler in higher versions of GCC. Methods to implement the basic block feature using an edge profiler must be designed and implemented for higher versions(3.0 and above) of GCC.

Bibliography

[1] Common Name for the Kernel DSO. http://www.uwsg.iu.edu/hypermail/linux/kernel/0306.2/0674.html.

[2] Next Generation Posix Threading. http://www-124.ibm.com/pthreads.

[3] B. Buchanan, D. Niehaus, G. Dhandapani, R. Menon, S. Sheth, Y. Wijata, and S. House. The Datastream Kernel Interface (Revision A). Technical Report ITTC-FY98-TR-11510-04, Information and Telecommunication Technology Center, University of Kansas, 1994 June.

[4] Jong-Deok Choi and Harini Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, August 1998.

[5] Ulrich Drepper. glibc-2.3.3/sysdeps/i386/backtrace.c. Comments in source code.

[6] Ulrich Drepper and Ingo Molnar. The Native POSIX Thread Library for Linux. http://people.redhat.com/drepper/nptl-design.pdf.

[7] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. Jarec: a portable record/replay environment for multi-threaded java applications. *Softw. Pract. Exper.*, 34(6):523–547, 2004.

[8] S. Halbhavi. Thread debugger implementation and integration with the Smartgdb debugging paradigm. Master's thesis, University of Kansas, 1995.

[9] S. Malladi. A thread debugger for testing and reproducing concurrency scenarios. Master's thesis, University of Kansas, January 2003.

[10] S. Penumarthy. Design and Implementation of a User-Level Thread Library for Testing and Reproducing Concurrency Scenarios. Master's thesis, University of Kansas, December 2002.

[11] R. Ramanasankaran. A Framework for Recording and Replay of Software that Performs I/O. Master's thesis, University of Kansas, May 2004.

[12] Michiel Ronsse, Koen De Bosschere, and Jacques Chassin de Kergommeaux. Execution replay and debugging. In *Proceedings of the Fourth International Workshop on Automated Debugging*, August 2000.

[13] Michiel Ronsse, Mark Christiaens, and Koen De Bosschere. Cyclic Debugging Using Execution Replay. In *Lecture Notes in Computer Science*, volume 2074, page 851, January 2001.