Performance Engineering of Software Systems
Massachusetts Institute of Technology 6.172
Prof. Charles E. Leiserson and Prof. Julian Shun Handout 14

Homework 9: Deterministic Execution

Please answer the questions in this handout and submit an *individual* writeup.

[Note: This assignment makes use of AWS and/or Git features which may not be available to OCW users.]

1 Introduction

We'll explore the power of deterministic execution, which may or may not help you debug your final project. Before we even get to parallel code, there will be plenty of bugs to root out in your serial code.

2 Environment dependencies

The most entertaining bugs to debug are the ones that depend on the environment. Why would a bug happen only on `awsrun` but not on your development machine? Why would code that works great for you, occasionally crash for your partner, and always for your TA?

Let's dig deeper into this simple program excerpt from `undef.c`.

```
01 main() {
02   int i;
03   printf("value of i=%d\n", i);
04   printf("address &i=%p\n", &i);
05 }
```

Figure 1: An undef.c excerpt

The `Makefile` contains a number of useful targets for this exercise. Whenever you use a target, you may find it useful to examine the `Makefile` in detail and understand why certain targets have the result they do.

Run it a few times using `make undef-compare` and compare the results. Are they the same? Why or why not? Now, let's fix these nondeterministic outputs line by line.

Whenever your program is using undefined state, it's often going to produce non-deterministic results. Undefined variables (or bits within one) are one such nondeterminism source. Fix the code to define the variable and rerun `make undef-compare` to make sure that it worked.

Next, we will look at nondeterminism in addresses. The higher order bits of a pointer are random due to Address Space Layout Randomization (ASLR) (look on Wikipedia for more details), which is an important security feature now in all modern operating systems. However, for testing how it impacts your address-dependent nondeterministic program, you can run your program without ASLR with the command `setarch x86_64 -R ./undef`. Run `make undef-noaslr` to run the program a few times without ASLR. Is this program deterministic now?

Even after considering these two factors, there are even more aspects that affect a nondeterministic program. Run `make undef-env` to run the command with differing environments. What is printed now? To see why, run `make whoami` and take a look at `user.c` :

```
06 int main(int argc, char *argv[], char* envp[]) {
07   char *username=genenv("USER");
08 }
```

Figure 2: An excerpt from user.c

You can see that the stack location is perturbed by the environment block, which affects the lower order bits of the stack variable *i* in `undef.c`.

Thus, we learned that program addresses are thus an important source of nondeterminism. These are a derivative of environmental, time, and randomness sources beyond our control.

3 Nondeterministic Hashtable

Now let's see a more complicated example in `hashtable.c`. It has a mostly working implementation of an open address hashtable with linear probing. This technique is a lot more cache-friendly compared to hashtables with linked lists.

Run `make hashtable1` a few times. It may work for you most of the time. How about `make hashtable10` or `make hashtable1000`?

> **Checkoff Item 1:** What do you need to make this program deterministic? Modify the Makefile target for `hashtable1000-good` so that all runs succeed.

To fix our bugs, we want the bugs to be reproducible.

Checkoff Item 2: Modify `hashtable1-bad` target so the program always fails. Show your modified Makefile target. You may find it useful to examine `hashtable.c` to see what system arguments it takes.

4 Replay Debugging

To fix serial code, you can use the GCC Record/Replay debugging facilities:

http://www.sourceware.org/gdb/wiki/ProcessRecord/Tutorial

```
(gdb) break main
(gdb) run
(gdb) record
(gdb) continue
(gdb) reverse-next
(gdb) reverse-next
(gdb) reverse-continue
```

If you don't like typing, you can just do

```
(gdb) b main
(gdb) r
(gdb) rec
(gdb) c
(gdb) rn
(gdb) rn
(gdb) rc
```

You may find it useful to watch some variables when debugging:

```
(gdb) set can-use-hw-watchpoints 0
(gdb) watch some_variable_name
```

You don't necessarily have to use good tools—if you have plenty of time, you can always figure out any bug by staring at the code long enough (static analysis by eyeballing). However, a much easier approach is to run your code with asserts, printf statements, or replay debugging techniques like gdb. Use any of the above techniques to find and fix the bug in `hashtable_insert`.

Checkoff Item 3: What was the bug? What is your fix? Rerun `make hashtable1000` to ensure that your fix always works.

Remember that this same bug may appear in other functions that have similar routines. This is an important reason why you should try not to write the same piece of code twice. An easy way to fix this is to refactor your code or use macros.

5 Hashlocking

Assuming that the serial code works, let's move on to the parallel version with `make hashtable-mt`. If you run `make hashtable-mt1000`, you may find some errors. We will need some synchronization here to fix the errors.

One possibility is to use a single hashtable lock, as in `hashtable_insert_locked`. However, this lock would be very highly contended. A good technique for reducing lock contention on a shared data structure is to split the lock into multiple locks. For example, instead of a single "tablelock", we consider having a separate lock for each section of the hashtable. A lock per hashtable line ("rowlock") would have too much memory and cache overhead, but a lock per hash ("hashlock") gives us the amount of overhead that we want.

Modify `hashtable_fill` to use `hashtable_insert_fair`, which is a rather basic implementation of a hashlock.

Write-up 1: Is the hashlock implementation in `hashlock.c` vulnerable to false sharing? Why or why not?

Write-up 2: What problem is the fairness solution in `hashtable_insert_fair` introducing? Explain or demonstrate the behavior you see.

6 Lockless Hashtable

Now, let's try to avoid using a lock altogether. On modern processors, the assembly instruction `CMPXCHG16B` supports the compare-and-swap operation for 16 bytes, and similar instructions exist for 4 bytes and 8 bytes. However, early 64-bit processors didn't support the 16 byte instruction. In this vein, we'll use the `CMPXCHG8B` in our implementation to only work with 64-bit words. This does mean that we can't atomically write a whole `entry_t`. Assume that `size` can be a positive integer only. The relevant definitions are in `common.h`.

Write-up 3: Use `InterlockedCompareExchange64` to implement your changes in `hashtable_insert_lockless`. Don't forget to modify `hashtable_fill` to use your new code. Run `make hashtable-mt1000` to ensure that it works. What changes are necessary in `hashtable_lookup` so you can use just a 8-byte compare-and-swap in `hashtable_insert_lockless`?

7 Deterministic Hashtable

Write-up 4: (No implementation required.) How would you implement a deterministic hashtable structure, such that the order of insertions does not change the final hashtable state?