Course ‘Operating Systems Architecture’ – tutorial: Concurrency problems

UFAZ, L2
Lecturer: Pierre Parrend, Rabih Amhaz
This material is made available by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau on:
http://pages.cs.wisc.edu/~remzi/OSTEP/
http://pages.cs.wisc.edu/~remzi/OSTEP/Homework/homework.html

This homework lets you explore some real code that deadlocks (or avoids deadlock). The different versions of code correspond to different approaches to avoiding deadlock in a simplified `vector_add()` routine.
See the `README` for details on these programs and their common substrate.

1. Exercise 1

First let’s make sure you understand how the programs generally work, and some of the key options. Study the code in `vector-deadlock.c`, as well as in `main-common.c` and related files.

1.1 Now, run `./vector-deadlock -n 2 -l 1 -v`, which instantiates two threads (`-n 2`), each of which does one `vector_add` (`-l 1`), and does so in verbose mode (`-v`). Make sure you understand the output.

1.2 How does the output change from run to run?

2. Exercise 2

Now add the `-d` flag, and change the number of loops (`-l`) from 1 to higher numbers.

2.1 What happens?

2.2 Does the code (always) deadlock?

3. Exercise 3

3.1 How does changing the number of threads (`-n`) change the outcome of the program?

3.2 Are there any values of `-n` that ensure no deadlock occurs?

4. Exercise 4

Now examine the code in `vector-global-order.c`.

4.1 First, make sure you understand what the code is trying to do; do you understand why the code avoids deadlock?

4.2 Also, why is there a special case in this `vector_add()` routine when the source and destination vectors are the same?

5. Exercise 5

Now run the code with the following flags: `-t -n 2 -l 100000 -d`.

5.1 How long does the code take to complete?

5.2 How does the total time change when you increase the number of loops, or the number of threads?

6. Exercise 6

6.1 What happens if you turn on the parallelism flag (`-p`)?

6.2 How much would you expect performance to change when each thread is working on adding different vectors (which is what `-p` enables) versus working on the same ones?

7. Exercise 7

Now let's study `vector-try-wait.c`. First make sure you understand the code.

7.1 Is the first call to `pthread_mutex_trylock()` really needed?

7.2 Now run the code. How fast does it run compared to the global order approach?

7.3 How does the number of retries, as counted by the code, change as the number of threads increases?

8. Exercise 8

Now let's look at `vector-avoid-hold-and-wait.c`.

8.1 What is the main problem with this approach? How does its performance compare to the other versions, when running both with `-p` and without it?

9. Exercise 9

Finally, let's look at `vector-nolock.c`.

9.1 This version doesn't use locks at all; does it provide the exact same semantics as the other versions?

9.2 Why or why not?

10. Exercise 10

Now compare its performance to the other versions, both when threads are working on the same two vectors (no `–p`) and when each thread is working on separate vectors (`–p`).

10.1 How does this no-lock version perform?