CS 45: Operating Systems

CLab 9: threads

Write your answers in `README.md`. Create a `COLLAB.md` file to keep track of any outside resources you might use. Be sure to push to the repo after class (even if you are not done).

1 Race Conditions

Use the `threads-intro` directory (in `ostep-homework/`) to see if you understand how race conditions arise.

0. Either `chmod +x x68.py` or run `python3 x86.py`.
1. Let's examine a simple program, `loop.s`. First, just read and understand it. Then, run it with these arguments (`./x86.py -t 1 -p loop.s -i 100 -R dx`) This specifies a single thread, an interrupt every 100 instructions, and tracing of register `%dx`. What will `%dx` be during the run? Use the `-c` flag to check your answers; the answers, on the left, show the value of the register (or memory value) *after* the instruction on the right has run.
2. Same code, different flags: (`./x86.py -p loop.s -t 2 -i 100 -a dx=3,dx=3 -R dx`) This specifies two threads, and initializes each `%dx` to `3`. What values will %dx see? Run with `-c` to check. Does the presence of multiple threads affect your calculations? Is there a race in this code?
3. Run this: `./x86.py -p loop.s -t 2 -i 3 -r -R dx -a dx=3,dx=3` This makes the interrupt interval small/random; use different seeds (`-s`) to see different interleavings. Does the interrupt frequency change anything?
4. Now, a different program, `looping-race-nolock.s`, which accesses a shared variable located at address `2000`; we'll call this variable value. Run it with a single thread to confirm your understanding: `./x86.py -p looping-race-nolock.s -t 1 -M 2000` What is value (i.e., at memory address `2000`) throughout the run? Use `-c` to check.
5. Run with multiple iterations/threads: `./x86.py -p looping-race-nolock.s -t 2 -a bx=3 -M 2000` Why does each thread loop three times? What is final value of value?
6. Run with random interrupt intervals: `./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 0` with different seeds (`-s 1`, `-s 2`, etc.) Can you tell by looking at the thread interleaving what the final value of value will be? Does the timing of the interrupt matter? Where can it safely occur? Where not? In other words, where is the critical section exactly?
7. Now examine fixed interrupt intervals: `./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1` What will the final value of the shared variable value be? What about when you change `-i 2`, `-i 3`, etc.? For which interrupt intervals does the program give the "correct" answer?
8. Run the same for more loops (e.g., set `-a bx=100`). What interrupt intervals (`-i`) lead to a correct outcome? Which intervals are surprising?

2 pthreads

Use the `threads-api` directory (in `ostep-homework/`) to see if you understand how race conditions occur and can be detected using `helgrind`.

9. First build `main-race.c`. Examine the code so you can see the (hopefully obvious) data race in the code. Now run helgrind (by typing `valgrind --tool=helgrind main-race`) to see how it reports the race. Does it point to the right lines of code? What other information does it give to you?
10. What happens when you remove one of the offending lines of code? Now add a lock around one of the updates to the shared variable, and then around both. What does `helgrind` report in each of these cases?

11. Now let's look at `main-deadlock.c`. Examine the code. This code has a problem known as deadlock (which we discuss in much more depth in a forthcoming chapter). Can you see what problem it might have?

12. Now run `helgrind` on this code. What does `helgrind` report?

13. Now run `helgrind on`main-deadlock-global.c`. Examine the code; does it have the same problem that main-deadlock.c has? Should helgrind be reporting the same error? What does this tell you about tools like`helgrind‘?

14. Let's next look at `main-signal.c`. This code uses a variable (done) to signal that the child is done and that the parent can now continue. Why is this code inefficient? (what does the parent end up spending its time doing, particularly if the child thread takes a long time to complete?)

15. Now run `helgrind` on this program. What does it report? Is the code correct?

16. Now look at a slightly modified version of the code, which is found in `main-signal-cv.c`. This version uses a condition variable to do the signaling (and associated lock). Why is this code preferred to the previous version? Is it correctness, or performance, or both?

17. Once again run `helgrind` on `main-signal-cv`. Does it report any errors?