CPSC 457 Fall 2021 – Assignment 2

Due date is posted on D2L.
Individual assignment. Group work is NOT allowed.
Weight: 19% of the final grade.

Imagine you are writing a program that needs to call a function written by someone else. Unfortunately, this function occasionally misbehaves; sometimes this function crashes, sometimes it runs for far too long, and occasionally it never returns. How can you call such unsafe code in your program?

A possible answer to this question is that you can safely call unsafe code in your own program by a clever use of the `fork()` system call. In this assignment you will be given an unsafe code, and you will be asked to write code that invokes such unsafe code in a safe manner.

Overall description

For this assignment you need to write a function `safecall` with the signature:

```
int safecall(int n);
```

The purpose of `safecall()` function is to call another function, `unsafe()`, and return the result of that call. If the `unsafe` function was well behaved, you could write `safecall` like this:

```
int safecall(int i) {
  return unsafe(i);
}
```

However, the `unsafe()` function can misbehave, and you need to address this in your implementation of `safecall`. The misbehavior of `unsafe()` is that it sometimes runs for too long, sometimes it runs forever, and other times it even crashes. You need to write `safecall()` in such a way that it calls `unsafe()` similar to the above code, but also detects if `unsafe` misbehaves. If you detect no misbehavior, you simply return the result `unsafe()`, but if you detect misbehavior, you return special values (-1 or -2) as follows:

- If `unsafe(i)` runs for longer than 1 second, you stop the execution of `unsafe()`, and `return -1`
- if `unsafe(i)` crashes, you `return -2`
- If `unsafe(i)` returns a `result` in under 1 second, you `return the result`.

In order to deal with `unsafe()`'s misbehavior, you need to create a child process (using `fork` system call) and run `unsafe()` in the child process. The parent will then monitor the child process for misbehavior. You will also need a mechanism for transferring the results obtained by calling `unsafe()` in the child process back to the parent process. Please see the hints at the end of this documents for suggestions on how to accomplish this.

Starter code

Download the starter code from GitLab, and compile it:

```
$ git clone https://gitlab.com/cpsc457f21/safecall.git
```

```
$ cd safecall
$ make
```

The starter code includes an incomplete implementation of `safecall()` in the file `safecall.cpp`. The starter code also includes several example test inputs (see description below on their format). To run the code on `test1.txt` file, you can:

```
$ ./safecall < test1.txt
Finished in 0s
Correct results, good job.
```

Please note that some of the test files will cause the incomplete `safecall` function to misbehave (crash, run forever, or run for too long). Once you fix the `safecall` implementation, all test files should work fine.

Also, please note that the test files provided in the starter code are only samples. It is your job to design your own test files and test your implementation thoroughly before submitting.

Driver program

The included driver program (`main.cpp`) configures the `unsafe` function based on data from standard input. The driver then calls your `safecall()` function repeatedly. The driver then displays the outputs from your `safecall()` and compares them to the expected outputs. By default, the driver will only show outputs that are incorrect:

```
$ ./safecall < test4.txt
Only displaying outputs with errors
================================================
      | Expected   | Observed
Index | Output     | Output
------------------------------------------------
    1 | -1         | 1          ! wrong
    3 | -1         | 3          ! wrong
    5 | -1         | 5          ! wrong
    7 | -1         | 7          ! wrong
    8 | -1         | 8          ! wrong
   10 | -1         | 10         ! wrong
   11 | -1         | 11         ! wrong
   12 | -1         | 12         ! wrong
   13 | -1         | 13         ! wrong
   14 | -1         | 14         ! wrong
================================================
Finished in 50.004s
Wrong results :(
```

To see more debugging information, and to see all outputs (including the correct ones), you can pass `all` as a command line argument to the executable:

```
$ ./safecall all < test4.txt
```

If you do not want to see the outputs at all, and only want to see whether your `safecall()` function worked correctly, pass `none` keyword to the executable:

```
$ ./safecall none < test4.txt
Finished in 50.004s
Wrong results :(
```

Input format

The input format is described here to help you design your own test files.

The input to the driver is a text file containing a set of integers, separated by white spaces (spaces, tabs or new lines). The driver loads these integers into an array `input[]` and then uses that array to configure the `unsafe()` function as follows:

`input[i] >= 1`	`unsafe(i)` immediately returns `input[i]`
`input[i] == 0`	`unsafe(i)` sleeps for 0.5s, then returns `i`
`input[i] == -1`	`unsafe(i)` sleeps for 5s, then returns `i`
`input[i] == -2`	`unsafe(i)` never returns
`input[i] <= -3`	`unsafe(i)` crashes

The driver then calls `safecall(i)` for every `i` in range [0, `input.size()`-1].

For example, if the input contains 7 numbers: `55 0 3 -1 -2 10 -50`, then the driver will call `safecall(i)` on every `i` in range [0, 6], and `usafe()` will behave as follows:

`i`	`input[i]`	`unsafe(i)` behavior	Expected behavior of `safecall()`
0	55	Returns 55	returns 55 immediately
1	0	Returns 1 after 0.5s delay	returns 1 after 0.5s
2	3	Returns 3 immediately	returns 3 immediately
3	-1	Returns 3 after 5s delay	returns -1 after 1s
4	-2	Never returns	returns -1 after 1s
5	10	Returns 10 immediately	returns 10 immediately
6	-50	Crashes	returns -2 immediately

Submission

Submit a single file to D2L for this assignment:

`safecall.cpp`	Containing your implementation of `sefacall()`

Write your entire implementation of `safecall()` in the file `safecall.cpp`. Please, only modify the `safecall.cpp` file, and **do not modify** any other source files. The `safecall.cpp` that you submit must compile and run with the rest of the code in the GitLab repository. During marking, we may use a slightly different driver to test your code, so it is important that you follow the above instructions.

Appendix - Hints

Sharing results between child/parent and dealing with crashes

In the child process you can call `unsafe()`, and if it returns, you can write its result to a temporary file. If `unsafe()` does not return, i.e. it crashes, your child will automatically exit, and the temporary file will not be created. The parent can detect whether the child finished or crashed by detecting whether a file was created or not.

You can use `fopen`, `fclose`, `fread`, `fwrite` and `unlink` system calls for the above. I recommend you use the `/tmp` directory for your temporary file, as it will be faster to read/write files there than in your home directory.

Note: If you know how to use `mmap` and `munmap` to setup shared memory, you can use those instead of temporary files to make your code run slightly more efficiently. This is **not required**.

Detecting if child runs for too long

I suggest using a busy loop with sleep in the parent process to detect when child runs for longer than 1s. Inside the busy loop you should periodically check whether the child is finished. To this end you should use the `waitpid()` system call, in non-blocking mode. I suggest you use it like this:

```
auto res = waitpid (pid, NULL, WNOHANG);
```

where `pid` is the child process ID (returned by `fork`). The `WNOHANG` option forces `waitpid` to return immediately. If the child finished, `waitpid` will return a positive integer, otherwise it will return a number smaller than 1.

To sleep for small amount of time in your loop, I suggest you use:

```
std::this_thread::sleep_for(std::chrono::microseconds(nap_time));
```

where `nap_time` is the number of microseconds you would like to sleep. You can experiment with different values for this delay, but I found that 1ms worked well for me. Do not set the delay too long though, as it would slow down your code when `unsafe()` returns quickly.

In order to exit the busy loop in case `unsafe` runs longer than 1s, you will need to measure elapsed time since creating the child. To this end, I suggest you save the current time before calling fork:

```
auto start_time = std::chrono::steady_clock::now();
auto pid = fork();
```

Then, inside the busy loop, you can find out how much time has elapsed using code like this:

```
auto curr_time = std::chrono::steady_clock::now();
double elapsed = std::chrono::duration_cast<std::chrono::microseconds>
   (curr_time - start_time).count() / 1000000.0;
if( elapsed > 1.0) { /* unsafe() is taking too long */ }
```

To kill the child process if it runs longer than 1s, use:

```
kill( pid, SIGKILL);
```

Alternative way for detecting if child runs for too long

An alternative way of detecting whether the child runs for too long is to use a 2nd child process, instead of a busy wait loop. The 2nd child will simply sleep for 1 second and then exit. In the parent process you can then detect which child finished first by calling `pid=wait(NULL)`, which will return the pid of the child that finished first. If the first child finishes first, that means it finished under 1s. If the second child finishes first, that means the first child is still running. Please remember to send `SIGKILL` to the child that did not yet finish, and then call another `wait()` for it to terminate.

This method can be simpler to program than the one in the previous hint, but it can also be quite a bit harder to debug. Therefore I recommend you use hint 1.

Appendix – correct solution

I made my sample solution available as an executable on linuxlab:

```
~pfederl/public/cpsc457f21/safecall
```

Here is how you can run it, for example, on `test7.txt`:

```
~pfederl/public/cpsc457f21/safecall all < test7.txt
```

General information about all assignments

1. All assignments are due on the date listed on D2L. Late submissions will not be marked.
2. Extensions may be granted only by the course instructor.
3. After you submit your work to D2L, verify your submission by re-downloading it.
4. You can submit many times before the due date. D2L will simply overwrite previous submissions with newer ones. It is better to submit incomplete work for a chance of getting partial marks, than not to submit anything. Please bear in mind that you cannot re-submit a single file if you have already submitted other files. Your new submission would delete the previous files you submitted. So please keep a copy of all files you intend to submit and resubmit all of them every time.
5. Assignments will be marked by your TAs. If you have questions about assignment marking, contact your TA first. If you still have questions after you have talked to your TA, then you can contact your instructor.
6. All programs you submit must run on `linuxlab.cpsc.ucalgary.ca`. If your TA is unable to run your code on the Linux machines, you will receive 0 marks for the relevant question.
7. Unless specified otherwise, you must submit code that can finish on any valid input under 10s on linuxlab.cpsc.ucalgary.ca, when compiled with `-O2` optimization. Any code that runs longer than this may receive a deduction, and code that runs for too long (about 30s) will receive 0 marks.
8. **Assignments must reflect individual work**. Here are some examples of what you are not allowed to do for individual assignments: you are not allowed to copy code or written answers (in part, or in whole) from anyone else; you are not allowed to collaborate with anyone; you are not allowed to share your solutions (code or pseudocode) with anyone; you are not allowed to sell or purchase a solution; you are not allowed to make your code available publicly. This list is not exclusive. For further information on plagiarism, cheating and other academic misconduct, check the information at this link: http://www.ucalgary.ca/pubs/calendar/current/k-5.html.

9. We will use automated similarity detection software to check for plagiarism. Your submission will be compared to other students (current and previous), as well as to any known online sources. Any cases of detected plagiarism or any other academic misconduct will be investigated and reported.