UCLA Computer Science 35L Final Exam - Winter 2022
Open book, open notes, closed computer.
180 points total, 180 minutes total
Write answers on the exam in spaces provided.

Name:

Student ID:

1 (8 minutes). Briefly explain why GDB watchpoints can greatly slow down debugging compared to GDB breakpoints, and why there are important special cases on the SEASnet GNU/Linux hosts where GDB watchpoints can be implemented efficiently anyway.

2 (12 minutes). Explain why Git and zlib Huffman-encode the result of dictionary compression, rather than the reverse (i.e., dictionary-encode the result of Huffman compression).

3 (16 minutes). The 'cp' command lets you copy many files to the same directory. For example, if G/H is a directory, the command:

```
cp A B/C D/E/F G/H
```

copies A to G/H/A, B/C to G/H/C, and D/E/F to G/H/F. Suppose you want to do the reverse, i.e., to copy many files *from* the same directory. Write a shell script fromcp to do that. For example, if G/H is a directory, the command:

```
fromcp G/H A B/C D/E/F
```

should copy G/H/A to A, G/H/C to B/C, and G/H/F to D/E/F. If any of the individual copying actions fail, fromcp should continue to attempt the rest of the actions, but it should exit successfully only if all the copying actions succeed.

4 (12 minutes). Suppose we have a toy C program implemented as follows. The file prog.in.h is empty, the file prog.c contains just the single line 'int main (void) { return 0; }', and we have the following Makefile:

```
prog: prog.o prog.h
        $(CC) $@.o -o $@

prog.o: prog.c prog.h
        $(CC) -c prog.c

prog.h: prog.in.h prog
        ./prog <prog.in.h >$@
```

We run the shell command 'make' and see the following output:

```
make: Circular prog.h <- prog dependency dropped.
./prog <prog.in.h >prog.h
/bin/sh: line 1: ./prog: No such file or directory
make: *** [Makefile:5: prog.h] Error 127
```

We think, "What in the world happened? I'm not sure, let's try it again." So we run the shell command 'make' again and see the following:

```
make: Circular prog.h <- prog dependency dropped.
cc -c prog.c
cc prog.o -o prog
```

and we then think "Oh, it's just some sort of 'make' bug that generates a false alarm, and it's not a real problem because 'prog' got built so I'll just move ahead to my next problem."

Explain why the first 'make' failed but the second 'make' succeeded, and why our problem diagnosis is incorrect.

5a (4 minutes). Briefly explain how the annualized failure rate (AFR) should affect your backup policy.

5b (4 minutes). If you use GitHub to store your class project source code, what is your failure model for backups, and what are the most important failures to worry about?

5c (4 minutes). What backup strategy should work well for these failures? Briefly explain.

6 (10 minutes). The command 'git merge-base', which was not discussed in class, is documented as follows:

> git merge-base finds best common ancestor(s) between two commits to use in a three-way merge. One common ancestor is better than another common ancestor if the latter is an ancestor of the former. A common ancestor that does not have any better common ancestor is a best common ancestor, i.e. a merge base. Note that there can be more than one merge base for a pair of commits.

Diagram an example repository where there is more than one merge base for a pair of commits, and specify the 'git merge-base' command that might output one merge base and might output another.

7a (5 minutes). The command 'git diff' exits with status 0 if there are no differences, 1 if there are differences, and some other status if there is trouble (e.g., a bad option or file name). Suppose you have a freshly cloned repository and working tree, containing a source file named 'f'. Give shell commands that will modify your repository and/or working tree so that the following shell command will succeed:

```
git diff && ! git diff HEAD
```

7b (5 minutes). Similarly, if you start again from a freshly cloned repository, give shell commands that will cause the following shell command to succeed:

```
git diff HEAD && ! git diff
```

8. Consider the following Solaris shell command and its output:

```
$ ls -lid / /.. . /usr/bin sparc*
 65138 drwxr-xr-x  4 root bin   716 2021-12-01 17:43 .
     3 drwxr-xr-x 35 root root   47 2022-01-25 12:54 /
     3 drwxr-xr-x 35 root root   47 2022-01-25 12:54 /..
 65138 drwxr-xr-x  4 root bin   716 2021-12-01 17:43 /usr/bin
242177 -r-xr-xr-x 29 root bin  9912 2011-04-01 15:37 sparc
241929 drwxr-xr-x  2 root bin    11 2020-11-04 11:40 sparcv7
241930 drwxr-xr-x  2 root bin    61 2021-12-01 17:43 sparcv9
```

8a (4 minutes). What is the the significance of the '4' in the 3rd column of the /usr/bin line? Which, if any of the other lines of output help to explain why that value is 4 instead of something else?

8b (4 minutes). How many subdirectories does '/' have on this system? Briefly explain.

9 (20 minutes). You are working on a software development project that is in maintenance mode. You're the only person working on it. When you access the repository, you almost always want the most recent version; it's rare to access older versions. And when you modify the source code, you almost always change just one source file.

For your project, compare and contrast the strengths and weaknesses of the following approaches for doing software maintenance. Assume that you're equally familiar with all the approaches and do not have to worry about compatibility with other projects in your organization.

* A file system with version numbers
* A file system with snapshots
* the Source Code Control System (SCCS)
* the Revision Control System (RCS)
* Git

10 (10 minutes). Consider the following Emacs Lisp source code:

```
(defun what-line ()
  (interactive)
  (let ((start (point-min))
        (n (line-number-at-pos)))
    (if (= start 1)
        (message "Line %d" n)
      (save-excursion
        (save-restriction
          (widen)
          (message "line %d (narrowed line %d)"
                   (+ n (line-number-at-pos start) -1) n))))))
```

Suppose we remove the '(save-excursion' and '(save-restriction' lines, and remove two ')'s from the end of the last line. How will this affect the user experience, for those who use 'what-line'?

11 (15 minutes). Node supports both synchronous I/O, which blocks the V8 thread until the I/O operation completes, and asynchronous I/O, which doesn't. For example, fs.readFileSync is synchronous whereas fs.readFile is not. In general, when is it better to use synchronous I/O, and when is it better to use asynchronous I/O, and why?

12 (15 minutes). ECMAScript 2020 (ES2020) has a feature dynamic imports, which let you do something like this:

```
document.getElementById("helpbutton")
.addEventListener("click", async () => {
   const { nextPage } = await import("./HelpPage.js");
   nextPage();
});
```

This lets you delay importing HelpPage.js until the user clicks on the help button.

Discuss some advantages and disadvantages for your project's using dynamic imports versus the more-traditional static imports.

13 (8 minutes). Naively one might wonder why GCC has so many optimization options, and why GCC doesn't simply generate optimal code without your having to tell it to. One answer is that there's a tradeoff: the more time GCC spends "thinking" about your code, the higher-performance the generated code can be, and you may prefer faster compile-time or faster run-time but you can't have both and so the -O flag lets you choose.

This is not the only tradeoff in GCC optimization, though. Give two other tradeoffs, specify what options are used to control GCC's behavior towards those tradeoffs, and say which choice student code will typically prefer and why.

14 (12 minutes). As mentioned in class, when you debug a program with GDB there are ordinarily two processes running. Would it make sense, or even be possible, for GDB to debug itself? Would this involve one process or two? Or if it's not possible, can some other debugger debug GDB? Briefly explain.

15. Consider the following two implementations of the Square component, discussed in the React tutorial.

```
class Square extends React.Component {
  render() {
    return (
      <button className="square"
              onClick={() => this.props.onClick()}>
        {this.props.value}
      </button>
    );
  }
}

function Square(props) {
  return (
    <button className="square" onClick={props.onClick}>
      {props.value}
    </button>
  );
}
```

15a (6 minutes). Explain why the class implementation needs 'this.' in places where the function implementation does not, and why the class implementation needs '() =>' and '()' in places where the function implementation does not.

15b (6 minutes). Since the function implementation is shorter and easier to read than the class implementation, why would developers bother using class implementations in React? Shouldn't they invariably prefer function implementations? Briefly discuss.