

22W-COM SCI-35L-LEC-1 Final

TOTAL POINTS

132.5 / 180

QUESTION 1

1 GDB watchpoints vs breakpoints **4.5 / 8**

Incorporates definition of GDB watchpoint: ("You can use a watchpoint to stop execution whenever the value of an expression changes, without having to predict a particular place where this may happen. (This is sometimes called a data breakpoint.) The expression may be as simple as the value of a single variable, or as complex as many variables combined by operators.") (https://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_29.html)

✓ + **3 pts** Correct

+ **1.5 pts** Good Attempt

+ **0 pts** Poor Attempt/Missing

Why software watchpoints are slow (have to single step through the instructions and check if variable were updated)

+ **3 pts** Correct

✓ + **1.5 pts** Good Attempt

+ **0 pts** Poor Attempt/Missing

Special Case: Hardware watchpoints

+ **2 pts** Correct

+ **2 pts** Another correct answer

+ **1 pts** Good Attempt

✓ + **0 pts** Poor Attempt/Missing

QUESTION 2

2 Huffman-encode and dictionary-encode **12 / 12**

✓ + **1.5 pts** Basic understanding of dictionary compression

✓ + **1.5 pts** Basic understanding of huffman encoding

✓ + **3 pts** Explained how dictionary compression

works before huffman-encode (We can naturally apply huffman encoding on dictionary compressed strings)

+ **1.5 pts** Moderate try to explain how dictionary compression works before huffman-encode

+ **3 pts** Explained why dictionary compression could not work after huffman-encode (Adaptive dictionary compression in zlib)

✓ + **3 pts** Explained why dictionary compression could not work after huffman-encode (strings of bits would be hard to be encoded by dictionary)

✓ + **3 pts** Explained briefly why dictionary compression could not work after huffman-encode (Efficiency)

+ **1.5 pts** Moderate try to explain why dictionary compression could not work after huffman-encode (Others)

+ **0 pts** No answer

QUESTION 3

3 cp command **0 / 16**

Loop through each argument (except \$1)

+ **6 pts** Correct

+ **4 pts** Good Attempt

+ **2 pts** Moderate Attempt

✓ + **0 pts** Poor Attempt/Nothing

Loop Logic (capture base name of the rest of the arguments, append base name to the dest dir, copy the arg to the new destination)

+ **7 pts** Correct

+ **5.83 pts** Very minor bug

+ **4.67 pts** Good Attempt

+ **2.33 pts** Moderate Attempt

✓ + **0 pts** Poor Attempt/Nothing

Exit function with correct return status

+ 3 pts Correct

+ 1.5 pts Good Attempt

✓ + 0 pts Poor Attempt/Nothing

QUESTION 4

4 makefile 4.5 / 12

+ 2 pts Explains circular dependency and how it was dropped both the times

✓ + 1.5 pts Moderate try in explaining circular dependency

+ 4 pts Explains why the first call failed correctly (prog not present after dependency dropped)

✓ + 2 pts Decent try in explaining why first call failed.

+ 1 pts Incorrectly explained but tried why first call failed

+ 2 pts Includes point "prog.h was made in the first call"

+ 1 pts Decent try in making the point that prog.h was made in the first call

+ 4 pts Explains why next call succeeded correctly.

+ 2 pts Decent try in explaining why next call succeeded correctly.

✓ + 1 pts Incorrectly explained but tried why second call succeeded

+ 1 pts Written something relevant

+ 0 pts No answer

QUESTION 5

backup 12 pts

5.1 AFR 4 / 4

✓ + 4 pts Correct

+ 2.5 pts Didn't explain how.

+ 2 pts Incorrect explanation.

+ 0 pts No attempt.

5.2 github failure 2 / 4

+ 4 pts Correct

+ 2.5 pts Somewhat correct

✓ + 2 pts Incorrect

+ 0 pts No attempt

5.3 strategy 4 / 4

✓ + 4 pts Correct

+ 3 pts Somewhat correct

+ 2 pts Incorrect

+ 0 pts No attempt

QUESTION 6

6 git merge-base 10 / 10

✓ + 10 pts Correct

+ 7 pts Correct diagram

+ 2 pts Incorrect

+ 0 pts No attempt

QUESTION 7

git diff 10 pts

7.1 diff 1 1 / 5

+ 5 pts Correct

✓ + 1 pts Incorrect

+ 0 pts No attempt

7.2 diff 2 5 / 5

✓ + 5 pts Correct

+ 1 pts Incorrect

+ 0 pts No attempt

QUESTION 8

ls command 8 pts

8.1 3rd column 4 / 4

✓ + 4 pts Correct

+ 1 pts Incorrect

+ 0 pts No attempt

8.2 subdir 1 / 4

+ 4 pts Correct

✓ + 1 pts Incorrect

+ 0 pts No attempt

QUESTION 9

9 compare software maintenance 20 / 20

✓ + 20 pts Correct

- + **13 pts** Not quite correct
- + **3 pts** Incorrect or no explanation
- + **0 pts** No attempt

QUESTION 10

10 emacs lisp 10 / 10

- ✓ + **10 pts** Correct
- + **6 pts** Not quite correct
- + **2 pts** Incorrect
- + **0 pts** No attempt

QUESTION 11

11 sync/async IO 11 / 15

- ✓ + **3.5 pts** [When] Sync IO: has data dependencies; e.g.: when reading the configuration files (before finishing, you can do nothing else)
- + **1.5 pts** [When] Sync IO: moderate try
- + **4 pts** [Why] Sync IO: easier; avoid error/ensure correctness when there are data/logical dependencies
- + **2 pts** [Why] Sync IO: moderate try
- ✓ + **3.5 pts** [When] Async IO: no dependency problem, don't want to hang/block other operations; e.g.: read a file and answer a request at the same time
- + **1.5 pts** [When] Async IO: moderate try
- ✓ + **4 pts** [Why] Async IO: more efficient/faster when there's no correctness problem
- + **2 pts** [Why] Async IO: moderate try
- + **0 pts** No Answer

QUESTION 12

12 dynamic/static imports 11.5 / 15

- ✓ + **7.5 pts** Advantage of dynamic import (Full points if one advantage mentioned; e.g.: start faster; no need to load if not touched; ...)
- + **7.5 pts** Disadvantages of dynamic import (At least two disadvantages mentioned; describing from the same perspective with two different sentences is counted as one disadvantage; e.g.: less coupled with the main page; need extra waiting time to load when needed; hard to develop, less likely to catch

bugs/problems since it's not triggered every single time)

- + **4 pts** Advantage of dynamic import - Good attempt or minor errors
- ✓ + **4 pts** Disadvantage of dynamic import - Good attempt or minor errors
- + **2 pts** Advantage of dynamic import - Moderate attempt or some obvious errors
- + **2 pts** Disadvantage of dynamic import - Moderate attempt or some obvious errors / Or mention some points but didn't make it clear/sufficient/correct/convincing enough
- + **0 pts** No answer

QUESTION 13

13 gcc optimization options 8 / 8

- ✓ + **8 pts** Correct
- + **6 pts** Didn't explain which is preferred.
- + **4 pts** One correct tradeoff
- + **1 pts** Incorrect
- + **0 pts** No attempt

QUESTION 14

14 gdb efficiency 8 / 12

- + **12 pts** Answer is fully correct or has is largely correct with very minor errors. Great work!
- ✓ + **8 pts** Good attempt - student makes good arguments and is very close to correct answer but is missing detail or gets things incorrect, such as the fact that GDB can be used to meaningfully debug itself if there are two separate processes running, or that a known stable GDB could be used to debug a buggy GDB
- + **6 pts** Fair attempt - answer demonstrates some degree of understanding about debugging and GDB, but does not make convincing arguments for GDB being able to debug itself, misses some details, or is not fully correct
- + **3 pts** Basic Attempt - some degree of correctness but crucial details are missing or incorrect
- + **1 pts** Basic Attempt
- + **0 pts** Click here to replace this description.

QUESTION 15

React 12 pts

15.1 this 6 / 6

✓ + **6 pts** Fully correct, or very close - demonstrates very good understanding of class components vs functional components

+ **3 pts** Moderate attempt - some degree of understanding of class vs. functional components

+ **1 pts** Basic attempt

+ **0 pts** no attempt

15.2 class implementation 6 / 6

✓ + **6 pts** Fully correct, or very close - demonstrates very good understanding of advantages of class components

+ **3 pts** Moderate - demonstrates some degree of understanding of advantages of class components

+ **1 pts** Basic attempt

+ **0 pts** No attempt

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.

GDB breakpoints can greatly slow down debugging b/c the program will stop executing each time the value of the variable in the expression changes. For example, if the expression was $m < 0$ and initially $m = -1$, the program would stop executing if $m = 1$. On the other hand, breakpoints will only stop at the line the debugger desires the program to stop at.

GDB watchpoints can be implemented efficiently anyway if you are debugging your code and trying to trace the value of variable who's value should remain constant, however, ^{is specifically changing} its changing, so one would use watchpoints to find out where it

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).

One reason to use Huffman-encoding of dictionary-compression rather than the reverse is because the Huffman coding strategy uses a fixed size RAM, as there are only 0-255 possible numbers that can represent the different strings and substrings in the dictionary. However, if the dictionary were to be the top layer, the process would require much more memory.

In addition, it makes more sense to encode using Huffman as the top layer as the Huffman encoding would assign values to substrings found in the dictionary compression process based on the frequency of a substring, making decompressing faster.

However, if we used Huffman compression first, it would decrease the speed as we would assign symbols to the most common characters, and then in the dictionary there would be translation. However the translation would be for individual characters and not substrings making the process much longer.

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.

Pseudocode

Go to the string in fromcp keep track of the first component. Look at the next component, if it is just a directory(A), then concatenate to end of path G/H → G/H/A and do cp command accordingly (cp G/H/A A). Continue to the next component. If the component is a directory rooted with subdirectories/files, loop through to the last character in the sequence. Concatenate the last character to the first component G/H → G/H/c and do the cp command for the entire component (cp G/H/c B/c). continue this process with all components.

Finally, if all copies performed successfully exit with no error, else exit with some error code or some error displayed.

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.

The first `make` failed because there is circular dependency in the `MAKEFILE`. In order to make `prog`, it depends on `prog.h` to be made. However, `prog.h` also depends on `prog` to be made. So, `prog` depends on `prog.h` and `prog.h` depends on `prog`, so neither can be built as one needs the other in order to execute.

The second `make` succeeded as this circular dependency dropped, as to build `prog.h`, the compiler dropped the dependency of `prog` and continued the process. In this way, nothing conflicts and the `make` process continues smoothly. `prog.h` is made, `prog.o` uses `prog.c` and `prog.h` to be made, and finally `prog` uses `prog.o` and `prog.h` to be made.

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

The annualized failure rate at which your flash drive/disks either partially fails or completely fails. This percentage is usually a 1%. If the AFR was high, one's backup policy must account for this by taking backups on multiple flash drives/disks. Another requirement may be that these drives are ^{of} independent manufacturers, so one doesn't run into the issues of using many drives that were consequence of a bad batch, causing one to lose all their backups.

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?

The most important failures to worry about are if someone ^{on the team} accidentally deletes files in the source code or if an outsider tries to hack in and delete files on the repo.

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

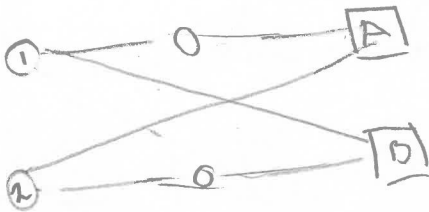
One backup strategy that could work is every time a ^{decently} large change is made, the project could be backed up to an external hard drive, so all the info could be stored there.

However, since Git maintains every version that is uploaded to the server, it is very simple to revert to a previous state of the project before all the failures occurred.

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.



git merge-base A B
 may output 1 as a merge base one time
 and it may output 2 as a merge base
 a second time.
 A and B both share 1. and 2 as
 common ancestors and 1 and 2 are
 not ancestors of each other.

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) ^{changes w/ most recent commit}
 0 1

```
'a' > f.c
git add f.c
git commit -m "added 'a' to f.c"
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
 0 1

~~git checkout -b Nodiff
 git checkout main
 'a' > f
 git checkout Nodiff
 git diff HEAD && !git diff~~

```
'a' > f.c
git add f.c
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 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?

The significance of the 4 in the 3rd column of the /usr/bin line is the inode count. This means that the directory has 4 hard links to it. The line that may explain its value is the first line with the same ID 65138 referenced as .. This may be one of the hard links.

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

/ has two subdirectories which are /usr and /usr/bin. This is because ls -ld will print out all the directories. Since . and /usr/bin have the same metadata, we know we are currently on the /usr/bin directory. The other directory is /.. .. represents the directory above and the directory above is /usr

Note there might be 4. The other two being sparcv7 and sparcv9 as they both have drwxr-xr-x.

(usr/bin/sparcv7)
(usr/bin/sparcv9)

↑
represents directory.



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

File system with version numbers

Advantages: There is data grooming, so if older versions of the code are not as frequently referenced, this will save memory.

The application decides when there should be a new version. Since you are the only one working, who made the change doesn't need to be saved.

Disadvantages: If you always want to access the most recent version, each time you make a change, a new version might be created which will take up a lot of memory, for even a minute change. There is no metadata.

File system with snapshots

Advantages: This design allows for cheap snapshots of filesystem, which saves memory. Does not save the name of who authored change.

Disadvantages: Snapshots are not taken at every change. They are taken periodically, so at some point, if a big change gets deleted by accident, that snapshot ^{with changes} may not be saved causing you to have to recreate. Metadata is not saved. If needed to make changes in older version you can't, old versions are immutable.

SCCS

Advantages: First introduction of metadata, can see what time change was authored and other details about change.

Disadvantages: The time required to retrieve any version is $O(1s.F1)$, so to retrieve recent version it is $O(1s.F1)$

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'`?

If we remove `save-excursion` and `save-restriction`, the user will no longer be able to use the functionality in Emacs, where you can narrow the window to a smaller size and focus on a specific piece of text. In addition the user may still see the message `line # (narrowed line #)` which would not make sense without the ability to narrow the size of the window.

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?

It is better to use synchronous I/O when you know the next steps in the process absolutely depend on the information retrieved by the synchronous I/O process such as `fs.readFileSync`.

However if the asynchronous I/O into retrieval is not needed immediately, it is smarter to use asynchronous I/O as the process will be running in the background until it completes without halting the execution of the thread/program.

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.

Advantages: Dynamic imports allows for conditionally importing something. This saves time in the process. For example, if we statically imported HelpPage.js, but never clicked a button, there would be no use of the import which wasted time when initially building the program. However, when we do click the button and use dynamic imports, the import will only happen if it is needed.

Disadvantages: Static imports allows the program to use whatever you import immediately. With dynamic imports, we must wait for the import to occur in the background until we can use it.

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.

One tradeoff is the speed in the debugging process. When you optimize, GCC will do calls in out of order execution making it hard to debug. Thus, if a student wanted to debug, they would use the -O0 option which turns off optimization, however the execution of the function would be relatively slow. Speed vs debug, as O# ↑, debugging becomes harder, but speed increases.

Another tradeoff is whether you want your compile time to be longer or shorter. Using the -fHo flag enables link-time optimization that allows compile time to be shorter, however this increases the link time tremendously as this is when the optimizations begin to be applied.

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.

GDB runs in its own shell, so it may be possible for GDB to debug itself. Like GDB does with debugging, if its debugging itself, it still makes sense for there to be multiple processes, so if one fails the other continues.

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.

The class implementation needs 'this' to know that it's editing its own value, however Square does not need it because the value that is getting edited is getting passed on as a parameter props. The parentheses are given because the onClick behaves as a function editing a pointer in to. The pointer being prop, while the function takes in prop as a parameter, which can directly modify prop.

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.

Developers can only use function implementations over class implementations when what they are trying to create does not need to render and does not have its own state. We can not render within a function implementation.