CS360 Exam #1: March 10, 1995

Answers

General Comments

So, this turned out to be another in a series of tests that are too long. These tests are difficult to take, because unless you're well on top of the material and are a quick thinker, you're not going to have time to finish the test. This means you have to say as much as you know, and hope that the grading will take the length of the test into account. Since I have the propensity to give long tests, I do this, and am usually very lenient in the grading.

That said, this test was kind of disappointing — especially question 1. I guess the TA's (and the lab scores) had led me to believe that you all were more on top of the details of C coding, so I gave a test that assumed as much. The answers to question 1 were by and large very disappointing. Question 2 and 3 were better, although there were still quite a lot of bad scores, and that was with generous grading.

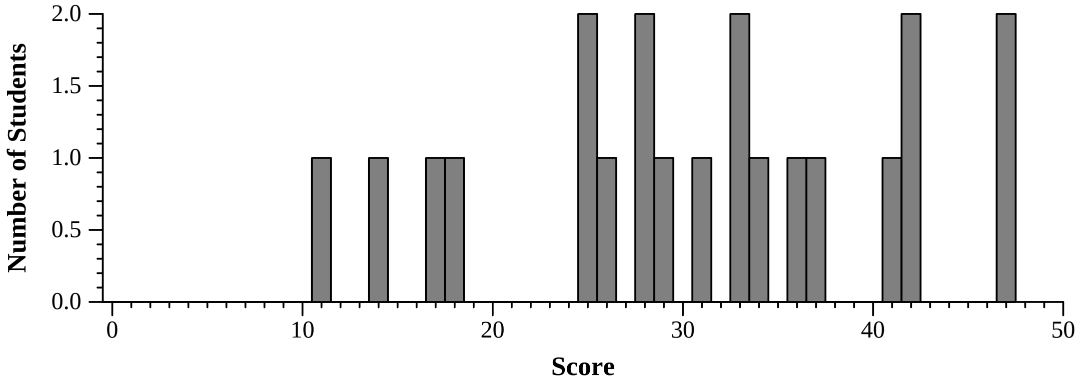
So, what I'm going to do is the following. There is going to be another exam this friday (3/17). It is going to consist of nothing but questions like Question 1, and I will make sure that it is not too long. If you scored higher than a 40 on this exam, you do not have to take it. This will give more data for assigning a final grade for the course — it will have the potential to help you if you bombed the midterm.

I would assign letter grades to this exam as follows:

- Above 40: A
- 30 – 39: B
- 25 – 29: C
- 20 – 24: D
- Below 19: F

That said, should you be worried if you got a really low score on this test? Well, frankly, yes. However, it can be made up — if you do well on Friday's test and the final, and you're doing well in your labwork, I'll ignore the results of this test. If you're worried about this, come and see me.

Histogram of test scores is as follows:



Total Score Histogram

Question 1: 15 points

Answers

Implementation A

```
UNIX> a.out
A
B
Segmentation fault
UNIX>
```

`subr(0)` returns the value zero, which gets placed into `ip[0]`. Note that this in itself doesn't cause a segmentation fault. Instead, when you try to print out `*ip[0]`, it will try to read the 4 bytes starting at address zero, which will cause the segmentation fault.

Implementation B

```
UNIX> a.out
A
B
XXXX
UNIX>
```

`XXXX` can a segmentation fault, a bus error, or a line of the form "*ip[0] = YYY" where `YYY` is some integer. This is because `subr(0)` returns a random integer, which turns into a random pointer. If the pointer is to a non-existing but aligned address, then you get a segmentation fault. If an ut unaligned address, then you will get a bus error. Otherwise, if it is an existing and aligned address, then it will point to some 4 bytes in memory, which will get printed out. If printing out the first line does not cause a segmentation fault or bus error, then it will try to print out the second line (`*ip[1]`), and so on either until a segmentation fault or bus error occurs, or until five lines are printed.

Implementation C

```
UNIX> a.out
A
B
*ip[0] = XXX
*ip[1] = XXX
*ip[2] = XXX
*ip[3] = XXX
*ip[4] = XXX
UNIX>
```

Here `subr(i)` returns a pointer to the argument `i`. This is a legal address (i.e. dereferencing it does not cause a segmentation violation or bus error), however, its value is not valid once `subr(i)` returns. Thus, it will print out some integer `XXX` (and that will be the same integer for every line), but we can't predict what that integer will be.

The `malloc()` there is useless. The statement after the `malloc()` sets `ptr` to some other value, and thus the 4 bytes that `malloc()` returns are lost.

Implementation D

```
UNIX> a.out
A
B
*ip[0] = 0
*ip[1] = 1
*ip[2] = 2
*ip[3] = 3
*ip[4] = 4
UNIX>
```

Here `subr(i)` mallocs 4 bytes and copies `i` to those 4 bytes. Some of you say that there is a problem here because you return a pointer to non-`malloc()`'d storage. This is not true – you have `malloc()`'d space for `j`, set `ptr` to `j` and returned `ptr`. Thus you are indeed returning a pointer to `malloc()`'d memory.

Implementation E

```
UNIX> a.out
A
B
*ip[0] = 0
Bus Error
UNIX>
```

Here `subr(i)` mallocs $4 + \mathtt{i}$ bytes and copies `i` into the last 4 of those bytes. Then it returns a pointer to those four bytes. The problem is that when `i` is not a multiple of 4, that pointer will be unaligned. Thus, when it tries to print `*ip[1]`, it will generate a bus error.

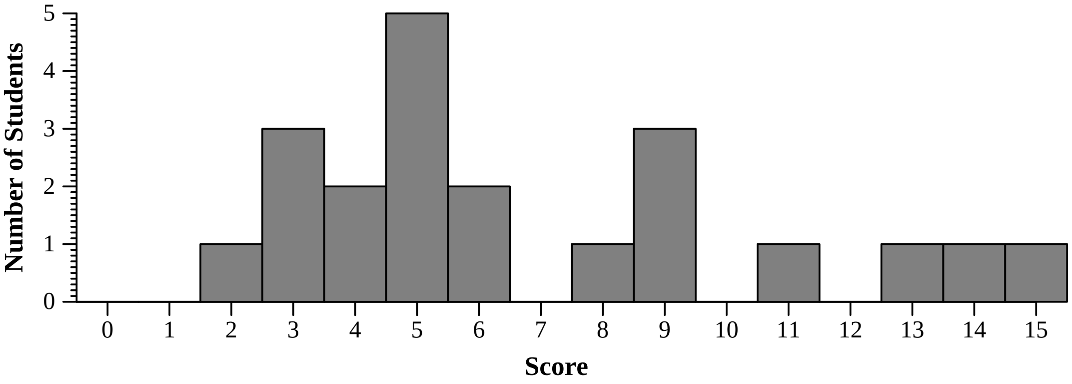
Grading

So, this turned out to be a very difficult question for most of you. The key thing to remember is that pointers are numbers — they are indices to the big memory array. Some of these indices are illegal — these are the ones that give you segmentation violations. When the pointer is to a 4-byte quantity, then that quantity must be aligned or you'll get a bus error. This question was to test your understanding of these points, and to test your code-reading ability. I graded accordingly.

The grading here is 3 points per part. If you said something reasonable but not correct, you got a point or two, depending on how reasonble it was. If it was really reasonable, you got all three. I was somewhat charitable here – if you said that it printed out

```
A
B
...
```

in Implementation A, that was worth a point, and if you mentioned anything having to do with randomness on Implementaiton B, that was worth a point.



Question 1 Histogram

Question 2: 20 points

Answers

Part 1 (5 points)

```
UNIX> chmod 04755 /mahogany/homes/plank/cs360/bin/jtetris
UNIX> chmod 0644 /mahogany/homes/plank/cs360/jt/scores
```

In other words, the setuid bit of `jtetris` should be set, and the `scores` file should be set so that only the owner can write it (0644 and 0600 are both ok). This way, the only way that one can change the `scores` file is by running `jtetris`. Were I not to set the setuid bit of `jtetris`, I'd have to make the `scores` file be writable by everybody, which would let anyone edit the file, and would corrupt the integrity of the high scores.

Part 2 (9 points)

For the most part, both implementations are fine. The problems come when more than one user runs `log_score()` at the same time. In `log_score1()`, the following scenario could occur:

- User 1 reads the high scores into his rb-tree.
- User 2 reads the high scores into his rb-tree.
- User 1 writes out the high scores with his score included
- User 2 writes out the high scores with his score included

In such a scenario, user 1's high score gets lost because user 2's rb-tree does not include it. Even worse things could happen if user 1 and user 2 interleave `write()` calls. Indeed the opening of the file using the `O_SYNC` flag means that each `write()` gets flushed instantly to disk; however, if the `write()`'s get interleaved, then the `scores` file could grow beyond 10 lines, and some lines could be included twice. In other words, if more than one user runs `log_score1()` at the same time, then the `scores` file can get messed up.

Instead, look at `log_score2()`: This procedure waits for until there is no file `cntl`, and the it creates that file. Then it reads/writes the `scores` file, and deletes the `cntl` file. The testing to see whether `cntl` exists and the creating of that file is done atomically by use of the `O_EXCL` flag. Note that now you can't have a scenario where different users read/write the `scores` file simultaneously. Only one can do it at a time, because in order to read/write the `scores` file, a user has to wait for `cntl` to go away. When the `open(CF, WCE, 0644)` statement actually returns with a non-negative value, we are guaranteed that no other user is reading/writing the `scores` file.

Part 3 (4 points)

This is a very primitive kind of locking. Basically, if you want to have a piece of code execute atomically with respect to another piece of code, you use a file like `cntl` which each piece of code must open exclusively when it wants to exectue, and which each piece of code must delete when it's finished.

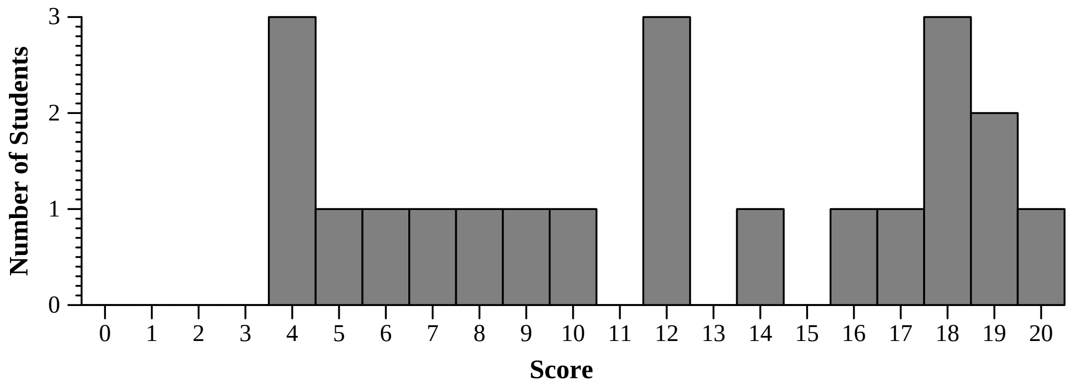
This is limited because all programs must cooperate to achieve this atomicity. If one program `A` wants certain pieces of code to execute atomically by using a file like `cntl` all other programs must also use `cntl` when they want to execute. Otherwise, there is no reason why the other programs cannot execute in the middle of `A`'s supposed "atomic" code.

Part 4 (2 points)

`Log_score2()` will fail if somehow the `cntl` file gets created and does not get deleted. For example, suppose a user kills his `jtetris` program while it is reading the `scores` file. Then the `cntl` file will exist and not be deleted, and therefore all other `jtetris` users will hang waiting for the `cntl` file to be deleted when they call `log_score2()`.

Grading

Again, I gave points if you said reasonable things. In part 1, you had to mention the setuid bit to get more than 1 point.



Question 2 Histogram

Question 3: 15 points

Answer

This has a structure just like `jtar` and `prsize`: Traverse the current directory using `opendir/readdir`. For each file, print out the complete pathname if it matches `fn`, and put it into a dlist if it is a directory. You must use `lstat` in order to ignore soft links. After traversing the directory, you should call `closedir` and then traverse the dlist, making recursive calls to `findfile` for each directory.

There are a few subtleties — you need to build up pathnames, and you need to check for `"."` and `".."` so that you don't traverse them recursively.

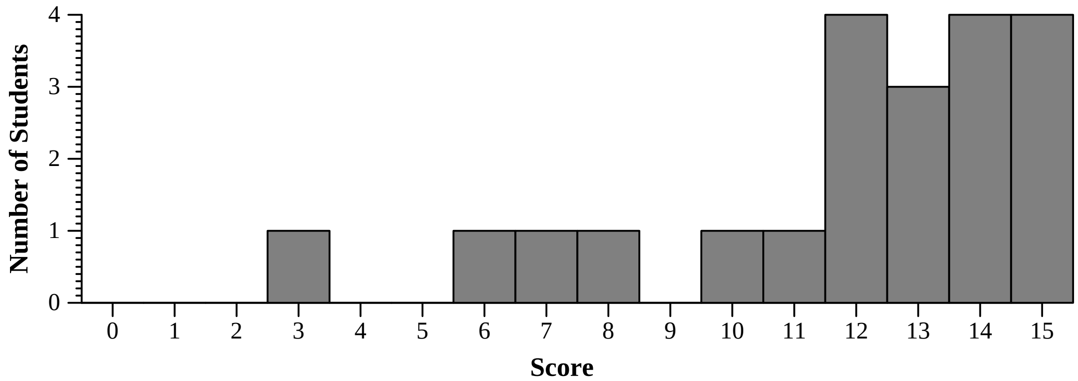
```
#include <stdio.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include "dlist.h"

findfile(char *fn, char *dir)
{
  Dlist dirs, tmp;
  struct stat buf;
  DIR *d;
  struct dirent *de;
  char *s;

  s = (char *) malloc(strlen(dir)+257);

  dirs = make_dl();
  d = opendir(dir);
  if (d == NULL) { perror("opendir"); exit(1); }
  for (de = readdir(d); de != NULL; de = readdir(d)) {
    sprintf(s, "%s/%s", dir, de->d_name);
    if (lstat(s, &buf) >= 0) {
      if (strcmp(de->d_name, fn) == 0) {
        printf("%s\n", s);
      }
      if (S_ISDIR(buf.st_mode) &&
          strcmp(de->d_name, ".") != 0 &&
          strcmp(de->d_name, "..") != 0) {
        dl_insert_b(dirs, strdup(s));
      }
    }
  }
  closedir(d);
  dl_traverse(tmp, dirs) {
    findfile(fn, tmp->val);
    free(tmp->val);
  }
  dl_delete_list(dirs);
  free(s);
}
```

Grading

This question was relatively straightforward. Grading goes as follows. There are two camps of answers — those with the correct general form, and those without. If your answer had the correct general form, you started with a base of 10 points, and your score was adjusted as follows:

- You remembered to build pathnames: 1 point
- You remembered to ignore `"."` and `".."`: 1 point
- You remembered to call `lstat`: 1 point
- You remembered to call `strdup` when inserting into the dlist: 1 point
- You used `malloc()` to allocate storage for the pathname instead of allocating it statically: .5 points
- You actually freed storage (e.g. the dlist, the `strdup`'d `val` fields): .5 points
- You didn't write code, but instead used pseudo-code/text: -3 points.
- You made the recursive subcall in the `opendir/readdir` loop: -1 point
- I gave an extra point if your program looked especially good (i.e. I'm fairly lenient about giving the above points. If your program looks like it would compile and run with just a few modifications, I gave you an extra point).
- I also took off between .5 and 2 points if there were parts of your code that looks particularly convoluted/incomprehensible/random.
- Round up and take the minimum of that number and 15.

If your answer didn't have the correct general form, you started with a base of zero points and I gave points for reasonable things said/done.



Question 3 Histogram