CS360 Final Exam

May 5, 1995

Answers

Jim Plank

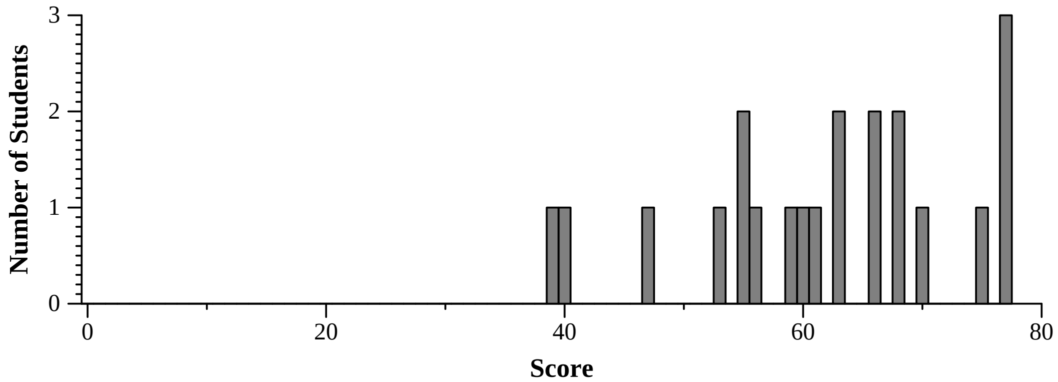
General Comments

So, this was a good exam, and I was very pleased with the quality of most answers. I'd assign the following letter grades to the exams:

- 75 – 80: A+
- 65 – 74: A
- 60 – 64: B+
- 53 – 59: B
- 45 – 52: C
- Below 44: D

I enjoyed teaching y'all, and I encourage you to send me email if you have any comments about ways to improve the class. Enjoy the summer.

Jim Plank



Total Score Histogram

Question 1: 5 points

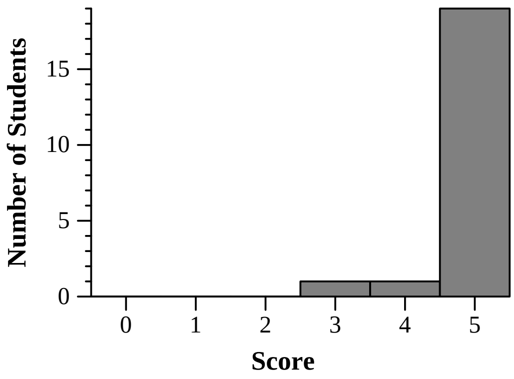
"An orphan can never become a zombie."

Explain.

The Answer

An orphan process is one whose parent process has exited. All orphans are inherited by `/sbin/init`. A zombie process is one who has died, but whose parent has not called `wait()`. Orphans cannot become zombies because children of `/sbin/init` go away instantly when they exit.

Good to see that everyone did will on this one:



Question 1 Histogram

Question 2: 15 points

The subroutine `free()` frees a block of memory so that it can be reused by `malloc()`. However, `free()` is passed only a pointer to the block to be freed. Answer the following questions about `free()`:

1. How does `free()` know how big this block is?
2. "Bad arguments" can be passed to `free()`. Describe what a "bad argument" to `free()` is, and what are the possible results of passing "bad arguments" to free.
3. `Free()` be fixed so that "bad arguments" are recognized (or are almost always recognized). Discuss how this can be achieved.

The Answer

1. `malloc()` allocates 8 extra bytes in front of the pointer that it returns. The first four of these bytes contain the size of the amount of memory that `malloc()` allocated. Thus, `free9)` can use this size to put this chunk of memory back on `malloc()`'s free list.
2. If `free()` is given an address that was *not* returned from `malloc()`, or an address that has already been `freed()`'d, then it will do bad things. It will assume that the address was returned by `malloc()`,

and it will look 8 bytes back from that address and treat whatever is there as the size of the region to free. There are many possible results of calling `free()` on a bad address. These are:

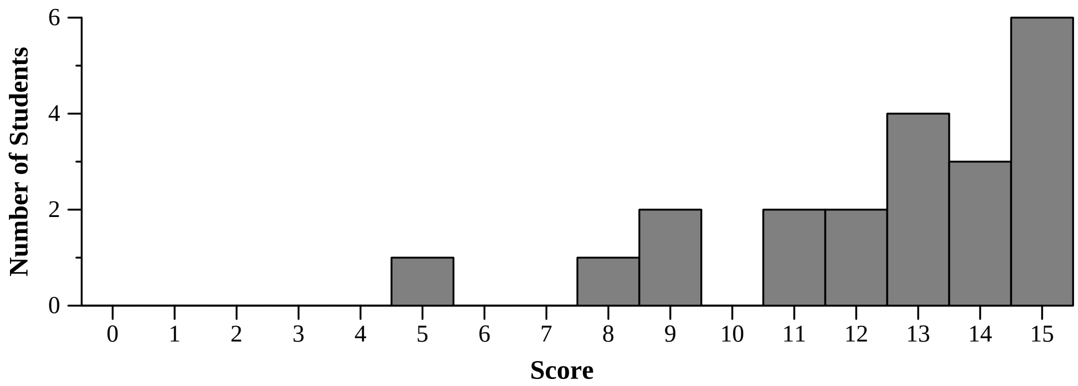
- Segmentation violation – the address is not a legal address.
- Bus error – the address is not aligned, so when `free()` tries to glean the size, it will generate a bus error.
- Unknown – it will put some chunk of memory on the free list that is perhaps still in use, and will then get `malloc()`'d later. Or it will put an already-`free()`'d chunk of memory on the free list, and it will get `malloc()`'d twice. In any case, the `free()` call will succeed, but subsequent `malloc()`'s will cause very strange results.

3. One good way is to put a checksum in the 4 bytes following the size. That way when `free()` is first called, it can check the 4 bytes preceeding the address and see if they equal the checksum. If so, then `free()` can be reasonably certain that the address is a good one. Otherwise, `free()` can flag the error instantly.

   Another way is to keep a list or hash table or rb-tree of `malloc()`'d chunks, and check to see if the address given in `free()` has been `malloc()`'d. This is less efficient than the checksum given above.

Grading

Point allocation was 3/8/4, and grading was pretty straightforward. On part 2, four of the points are for defining what a bad address is, and the other four is for what can happen.



Question 2 Histogram

Question 3: 15 points

Answer the following three questions about the program `telnet`:

1. What does it do? (i.e. give a man-page description of `telnet`).
2. We call `telnet` a "generic client program." Why? How does one structure a server to work with `telnet` clients?
3. Describe the `select()` system call — why does `telnet` have to use `select()`, and how does it do so?

The Answer

1. `telnet` accepts two command line arguments — a host name and a port number. It calls `request_connection()` on that host/port combination, assuming that there is a program that is serving a socket there. Once the connection is made, `telnet` performs two things — it reads in lines from standard input, and writes them to the socket, and it reads in lines from the socket, and prints them on standard output. If the socket closes, or standard input closes, then `telnet` exits (actually, this is only true in `jtelnet` – the real `telnet` exits when the socket closes, or when the user tells `telnet` to quit by hitting control-], and then typing "quit."

2. `telnet` is a generic client program because most Unix (and other) systems have it built-in, and thus one can write a server program that anyone on the internet can use without having to port a special client program. Structuring servers to work with `telnet` clients is simple – you serve a socket and then wait for connections. Once you recieve a connection, you can service it in an iterative or concurrent fashion. In either case, you receive lines of input along the socket connection, and print your output along the socket as well. If your server is iterative, then you don't process any new clients until the current client is done. If your server is concurrent, then you service multiple clients simultaneously, either via `select()` (as in `jtalk`) or by forking off a child process for each client.

3. `select()` is a system call that you make when you want to perform one of a number of I/O events, and you don't know which I/O event is going to happen first. You pass three sets of file descriptors to `select()` – one for reading, one for writing, and one for errors. Select will return when:

   - A file descriptor in the reading set has input to read and/or
   - A file descriptor in the writing set is able to process output (i.e. the OS buffer is not full) and/or
   - A file descriptor in the error set has an error condition on it.

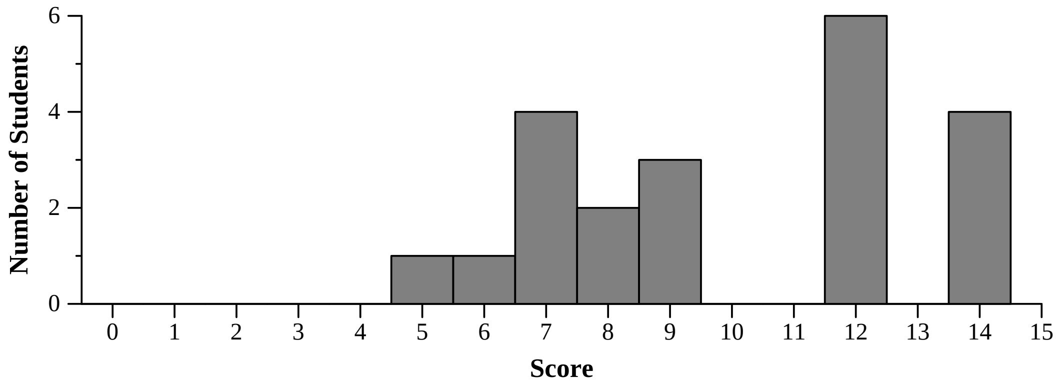
   You can also specify a timeout with `select()` so that it will return after a certain amount of time even if none of the file descriptors are ready for action.

   `telnet` has to use `select()` because it needs to process two I/O events simultaneously – reading from standard input, and reading from the socket. The only way that it can do this (without forking off a new process) is to use `select()`.

Grading

Point allocation here was 4/6/5:

- **Part 1** 2 points for saying that `telnet` requests a connection on the given host/port, and 2 points for saying how it works once the connection was made.
- **Part 2** 3 points for the description of "generic client." 3 points for saying how the server is structured: 1 for saying that it has to serve a socket; 1 for saying that it accepts connections along this socket and performs I/O along this connection, and 1 for some mention of serving multiple connections (i.e. iterative vs concurrent servers).
- **Part 3** 3 points for the description of select, and two points for describing why `telnet` needs to use `select()`. You did not need to define the write/error sets or mention the timeout or max field width argument.



Question 3 Histogram

Question 4: 25 points

The following are three implementations of `malloc()` and `free()`:

- **Implementation A**: Unix's implementation as described in class.
- **Implementation B**:

```
char *malloc(int size)
{
  return sbrk(size);
}

free(char *x)
{
  return ;
}
```

- **Implementation C**:

```
char *malloc(int size)
{
  if (size%4 != 0) size = size + 4 - (size%4);

  return sbrk(size);
}

free(char *x)
{
  return;
}
```

On the next page is a program `reverse.c` that reads in lines from standard input, and prints them out reversed. Suppose you call `reverse` on an input file containing 1,000,000 lines and 4,000,000 characters. Discuss the performance of `reverse` for each of the implementations of `malloc()` above. Included in this discussion should be the following:

- Does it work?
- *Exactly* how much memory is used? (If there's ambiguity, give me minimum and maximum values).
- The relative speed of execution.

After discussing each implementation, describe an implementation of `malloc()` that would work better than any of the above on this instance of `reverse`.

For your discussion, you should assume that the buffer size for standard input is 4K bytes as is the buffer size for standard output. Each `Dlist` node is 12 bytes. Obviously, pointers and integers are 4 bytes each, and characters are 1 byte each. We also assume that no line of the input file is larger than 1000 bytes.

A thing to remember: The procedures `make_dl()` and `dl_insert_b()` both call `malloc()` to allocate a `Dlist` node. `Strdup()` calls `malloc()` as well. Finally, `gets()` and `printf()` may both call `malloc()` in order to allocate a buffer for standard input or standard output.

```
#include <stdio.h>
#include <string.h>
#include "dlist.h"

main()
{
  char line[1000];
  Dlist d, tmp;
  char *s;

  d = make_dl();
  while(gets(line) != NULL) {
    s = strdup(line);
    dl_insert_b(d, s);
  }
  for (tmp = d->blink; tmp != d; tmp = tmp->blink) {
    s = (char *) tmp->val;
    printf("%s\n", s);
  }
}
```

`reverse.c`

The answer

Do they work?

Obviously, Implementation A works fine.

Implementation B will only work if all of the strings in the input have lengths that are multiples of 4 (including the newlines). Otherwise, this implementation will cause a bus error. Why? Well, assume that a string has a length of x which is not a multiple of 4. The `strdup()` call will call `sbrk(`x`)`, which means that the next call to `malloc()` will return an unaligned pointer. `dl_insert_b()` calls `malloc(12)`, and will generate a bus error trying to put this newly allocated node onto the dlist.

Implementation C also works fine – it pads the length so that `sbrk()` always returns an aligned pointer. Although implementation C's `free()` doesn't do anything, that's not a concern here because `free()` is not called.

Many of you said that the stdio library calls `free()` to free up its buffer when it's full or flushed. This is not true. You need to reread the lecture notes/book on buffering I/O if you don't understand this.

How much memory is used?

We won't worry about implementation B any further because it doesn't work. There are 5 places where `malloc()` is called:

1. In `make_dl()` – 12 bytes are allocated.
2. The first time `gets()` is called – this allocates 4K bytes for the standard input buffer. All subsequent calls to `gets()` use this buffer — thus, it only calls `malloc()` once.
3. Each time `strdup()` is called. `strdup()` is called 1,000,000 times. We know that each `malloc()` call can have any value from one to 1,000, and that the sum of these values over the 1,000,000 calls will be 4,000,000. Note that each newline in stdin is stripped by `gets()` so that indeed 4,000,000 bytes (and not 5,000,000) are allocated.
4. Each time `dl_insert_b()` is called. `dl_insert_b()` is called 1,000,000 times, and each call allocates 12 bytes for a total of 12,000,000 bytes.
5. The first time `printf()` is called – this allocates 4K bytes for the standard output buffer. All subsequent calls to `printf()` use this buffer — thus, it only calls `malloc()` once.

So, `malloc()` is called 2,000,003 times. In total, 16,008,204 bytes are allocated with these calls.

Were Implementation B to work, it would allocate exactly 16,008,204 bytes, since it does neither buffering nor padding.

Implementation C will allocate anywhere between 16,008,204 and 19,008,204 bytes. got these figures in the following way:

- The `make_dl()` call allocates 12 bytes.
- The first `gets()` and `printf()` calls allocate 4096 bytes each.
- Each `dl_insert_b()` call allocates 12 bytes, for a total of 12,000,000 bytes.
- Each `strdup()` call allocates anywhere from zero to three extra bytes for padding the `sbrk()` call to a multiple of four. This makes for 4,000,000 total bytes at minimum, and 7,000,000 bytes at maximum.

Implementation A allocates even more:

- The `make_dl()` call allocates 24 bytes (padding 12 to 16, and then adding 8 bytes for bookkeeping).
- The first `gets()` and `printf()` calls allocate 4104 bytes each.
- Each `dl_insert_b()` call allocates 24 bytes, for a total of 24,000,000 bytes.

- Each `strdup()` call allocates anywhere from 8 to 15 extra bytes (8 for bookkeeping and up to 7 for padding to a multiple of 8. This makes for 12,000,000 total bytes at minimum, and 19,000,000 bytes at maximum.

Add up to 8K for buffering the `malloc()` calls, and this means that between 36,008,232 and 43,016,424 bytes are allocated with Implementation A.

Relative speeds

Assuming that Implementations A and C won't run out of memory and that we don't see the effects of paging (which we actually would), Implementation A is going to run faster than Implementation C because of buffering. `sbrk()` is called 2,000,003 times in Implementation C. In Implementation A, lets just consider the `dl_insert_b()` and `strdup()` calls, since these account for 2,000,000 of the 2,000,003 calls. Each `dl_insert_b()` allocates 24 bytes in `malloc()`, and each `strdup()` call allocates 16 bytes on average (4 bytes, 4 padding, 8 bookkeeping). This means that the average `malloc()` call uses up 20 bytes of the buffer, resulting in $8192/20 \approx 410$ `malloc()` calls per `sbrk()` call. This should mean that Implementation C calls `sbrk()` around 4880 times. Since system calls are very expensive, we would expect Implementation A to run much faster than Implementation C.

A better approach

Obviously, a better approach should mix buffering with some strategy for reduced fragmentation. Since we're not using any `double's`, we don't need to worry about aligning pointers on 8-byte quantities. Thus, we only have to pad to 4 bytes instead of 8, as in Implementation C. Fixing the Unix implementation to pad to 4 bytes, and only include 4 bytes in the bookkeeping (since 4 of those bytes are unused anyway) will result in a savings of between 8,000,000 and 12,000,000 bytes in Implementation A. If we decide to throw away the the bookkeeping bytes entirely since our program never calls `free()`, we'll save another 4,000,000 bytes, and Implementation A will be as efficient space-wise as Implementation C (modulo 8K for the buffer). We can make it more efficient time-wise by increasing the buffer size from 8K to something larger, as long as we're willing to absorb the extra space of an unfilled buffer.

A still better approach can eliminate all wasted space due to padding. We can make the following assumption – if one calls `malloc(`x`)` and x is not a multiple of 4, then the resulting pointer does not need to be aligned because it is probably a `(char *)`. Then we can maintain two buffers – one for aligned memory, and one for unaligned memory. If we call `malloc(`x`)` and x is a multiple of 4, then we return a chunk from the aligned buffer, and if x is not a multiple of 4, then we return a chunk from the unaligned buffer. This way, no padding needs to be done, although `malloc()` will be a little more complex because it has to manage two buffers. Moreover, `free()` won't work, but that's ok, because we don't ever call it.

Grading

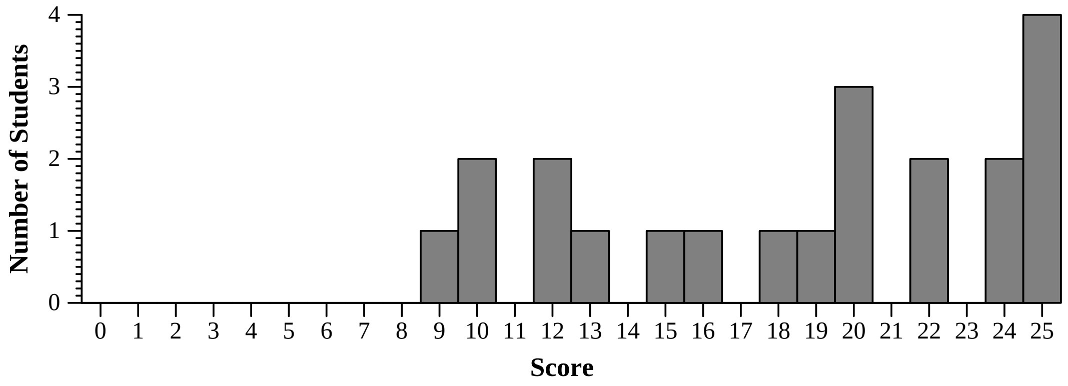
So, there are a few important parts of this question, which I expected everyone to get. Thus, I allocated 15 points for these basic things.

- Implementation A will work but will have a lot of internal fragmentation: 3 points.
- Implementation C will work but will be slow because of all the calls to `sbrk()`: 3 points.
- Implementation B has a problem with returning unaligned pointer accesses: 3 points. Whether or not this actually causes a bus error is subtle, but it does.

- Implementations B and C don't really implement `free()`, but that's ok because free is never called: 3 points.
- The best solution is one where there is buffering for performance, but not as much bookkeeping, and thus less fragmentation: 3 points.

The other points were allocated as follows:

- Implementation C/B uses on the order of 16-20M of memory. You have to explain why: 3 points.
- Implementation A uses 24M for dlists: 1.5 points.
- Implementation A uses between 12M and 19M for the `strdup()` calls: 1.5 points.
- Implementation B probably generates a bus error: 1 point.
- You gave a good mechanism for the best solution: 2 points.
- Everything else is in good order: 1 point.



Question 4 Histogram

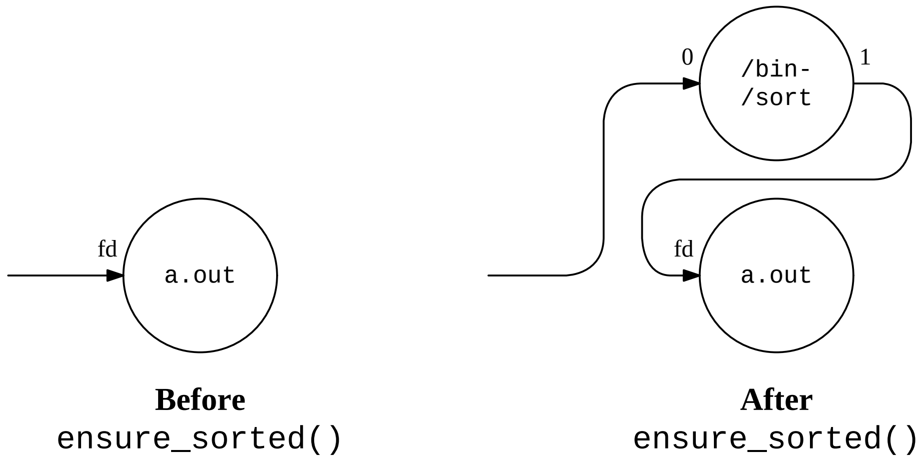
Question 5: 20 points

You are trying to join a nerd fraternity. As part of the hazing process, you have been asked to write the procedure `ensure_sorted()`, which is defined as follows:

One of your "brothers" has written a subroutine `process_input(int fd)` that reads its input from the file descriptor `fd` and does some complex processing on that input. There's one hitch — `process_input()` only works if the input is sorted lines of text. You must write the procedure `ensure_sorted(int fd)`, that will be the first thing called in `process_input()`. `Ensure_sorted()` must make sure that the input coming from `fd` is sorted.

This seems impossible to you, and you wonder why you wanted to be in this fraternity in the first place. But then the following idea hits you: I can fork off a process that executes `"/bin/sort"` to do the sorting for me. It can use `fd` for standard input, and will pipe standard output back to my process. I can then make things work so that when `process_input()` makes subsequent `read()` calls on `fd`, it will be reading from the pipe!

Write `ensure_sorted()`.



The Answer

So, the above picture tells the whole story. You need to first call `pipe` so that the `/bin/sort` process can talk to the `a.out` process. Next, you call fork. In the parent, you close the write end of the pipe, and `dup` the read end of the pipe onto `fd`. Finally you close the read end of the pipe. In the child, you `dup` `fd` onto standard input, and the write end of the pipe onto standard output. Then you close both ends of the pipe and close `fd`. Finally, you `exec` the program `/bin/sort`, and you're done:

```
ensure_sorted(int fd)
{
  int p[2];

  if (pipe(p) < 0) {
    perror("ensure_sorted");
    exit(1);
  }

  if (fork() != 0) {  /* Parent */
    dup2(p[0], fd);
    close(p[0]);
    close(p[1]);
    return;

  } else {            /* Child */
    dup2(fd, 0);
    dup2(p[1], 1);
    close(fd);
    close(p[0]);
    close(p[1]);
    execlp("/bin/sort", "sort", NULL);
    perror("ensure_sorted child -- execlp failed");
    exit(1);
  }
}
```

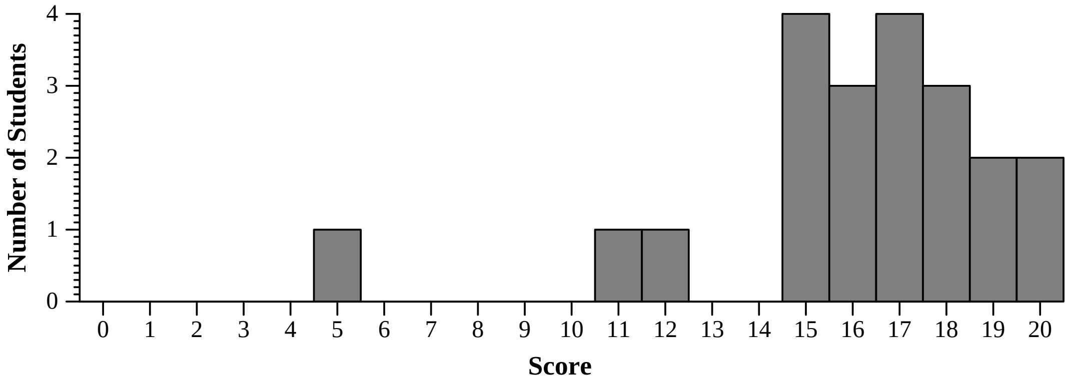
Note that I didn't call `wait()`. You don't need to – indeed, the `sort` process might hang around as a zombie, but you didn't have to worry about that. Since `sort` has to look at all of its input before making any output, calling `wait()` is ok, however.

Grading

Basically, grading worked as follows: If your code was basically correct (one `fork()`, one `pipe()`, you started at level 1 with 20 points. Otherwise, you started at level 2 with 15 points. Then, the following deductions were made for the following reasons:

- Not calling `exit()` if `execlp()` fails: -2
- Not closing `fd` in the child: -.5
- Not closing `p[0]` in the child: -.5
- Not closing `p[1]` in the child: -.5
- Not closing `p[0]` in the parent: -.5
- Not closing `p[1]` in the parent: -.5
- Not calling `dup2(p[0], fd)` in the parent, but calling some other erroneous `dup2()`: -1
- Not calling `dup2(p[0], fd)` or any other `dup2()` in the parent: -2
- Getting the `dup2()` backward (i.e. `dup2(1, p[1])`): -.5

- Bad or no argument to `wait()`, which will cause a seg fault: -1
- Not `dup()`-ing `fd` to zero in the child: -2
- Extraneous `fgets()`, `printf()`, or `fprintf()` calls: -1.5
- Calling `wait()` twice: -1
- `dup()`-ing `fd` to zero in the parent: -1
- Using `stdin`/`stdout` instead of 0/1: -1
- Closing 0 in the parent: -1
- Extraneous `dup2()`'s: -1 each
- Closing/`dup()`-ing `fd` before calling `fork()`: -1



Question 5 Histogram