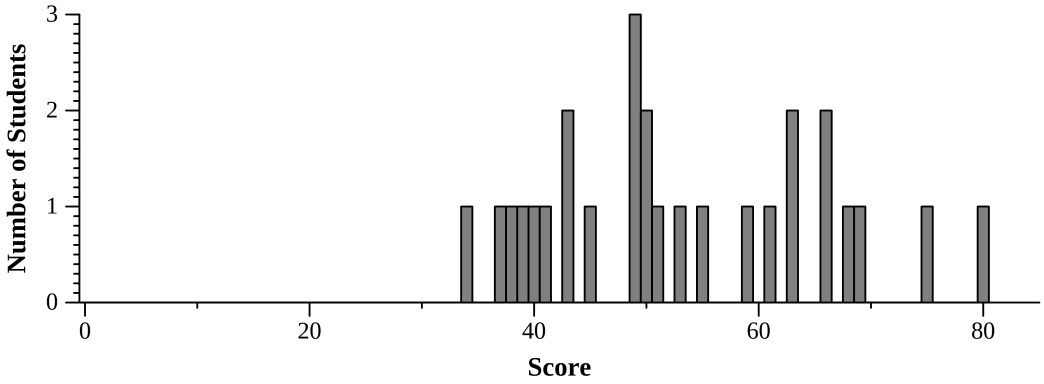
CS360 Final Exam: December 15, 1994

Answers

Jim Plank

Grading and general comments

So, this was another very difficult exam, which is good and bad. It's bad because I imagine it stresses y'all out a good bit, and sometimes it's hard to grade because your answers show the effects of time-panic. However, it's good because when you answer something well, it's extremely pleasing to see as a teacher, and when I make exams too hard, I tend to be very lenient with the grading. The final score histogram is as follows:



Total Score Histogram

Grade cutoffs look as follows:

- Above 65: A+ – a very impressive score.
- 56 – 65: A
- 53 – 55: B+
- 45 – 52: B
- 35 – 44: C
- Below 35: D

Question 1: 15 points

So, this was question 3 from exam #1 verbatim. As such, I expected to see everyone get parts 1 & 3 correct, and get about half of the assembler or more. The answer and grading guide is borrowed from exam #1.

Part 1: The bug is that `time()` expects an argument, and procedure `b` uses `time()` without an argument. You might try to say that this is not a bug, but from the description of `time()` in the man page, it is clear that `time()` always uses its argument — when it says "if tloc is non-NULL, the return value is also stored in the location to which tloc points", that in no way implies that tloc cannot exist. What it says is that `time()` will check its argument to see if it equals NULL (which is zero). If so, it returns the time. Otherwise, it assumes `tloc` is a pointer, and sets its value to the time, and returns that value.

Part 2:

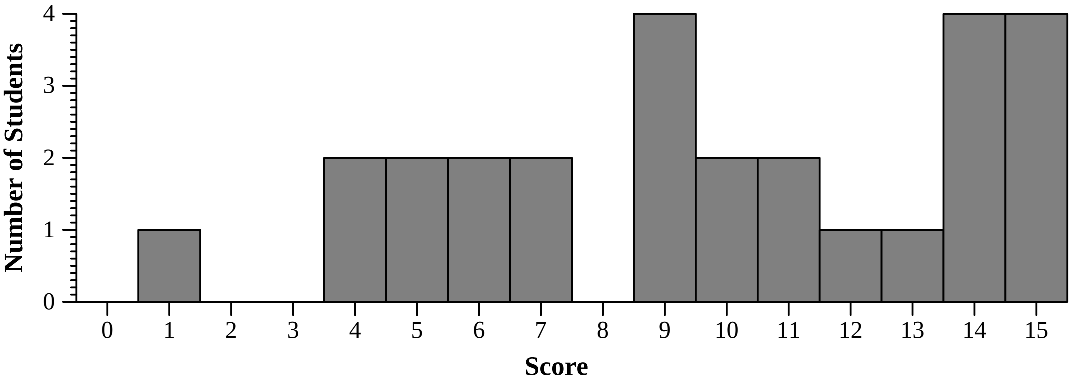
```
    .globl t1

 b:
    add #-4, sp -> sp          / Allocate room for i

    ld [fp+16] -> %r0          / Push k on the stack
    st %r0 -> [sp]--

    ld [fp+12] -> %r0          / Push j on the stack
    st %r0 -> [sp]--

    jsr proc1                  / Call proc1

    add #8, sp -> sp           / Pop j and k off the stack

    st %r0 -> [fp]             / Store the return value of proc1 in i

    jsr time                   / Call time

    st %r0 -> t1               / Store the return value of time to
                               / global variable t1

    ld [fp] -> %r0             / Set the return value to i
    ret
```

Part 3: Were there no bug, this code would return `proc1(j, k)` to the user, and set `t1` to the current time. However, there is a bug – that is that `time()` expects there to be an argument. It is going to look for that argument in `[fp+12]`, and what will be there? The value of `i`! So, if `i` is zero, then this code will run fine and won't need to be debugged. Otherwise, `time()` will try to treat `i` as a pointer and dereference it. This can have one of three results, depending on the value of `i`:

- If `i` is not a multiple of 4, you will get a bus error. This will be easy to find with a debugger.
- If `i` is not a legal address, you will get a segmentation violation. This will also be easy to find with a debugger.
- Otherwise, `time()` will overwrite some location in memory that it shouldn't be overwriting. This bug might go undetected, or it might be detected somewhere else in the program. It will be a nightmare to debug — the best way to find this would be to use the program `lint`, which tries to find things like argument mismatches. In this case, `link` will actually say `''time:  variable # of args.''`.

Grading: For part 2, grading was broken down into the following:

- 1 point for allocating room for `i` on the stack.
- 1 point for pushing `j` and `k` on the stack.
- 1 point for calling `jsr` for `proc1` and `time`.
- 1 point for popping `j` and `k` off the stack.
- 1 point for getting the return value of `proc1` into `i`
- 1 point for putting the return value into `r0`.
- 1 point for referencing `i`, `j`, and `k` as offsets to the frame pointer.
- Minus 1 point for changing `fp` or accessing the `pc`, or pushing the `fp` / `pc` on the stack.
- Half points for any of the above if you did them, but not really in the correct way.
- I rounded up in the end.



Question 1 Histogram

Question 2: 15 points

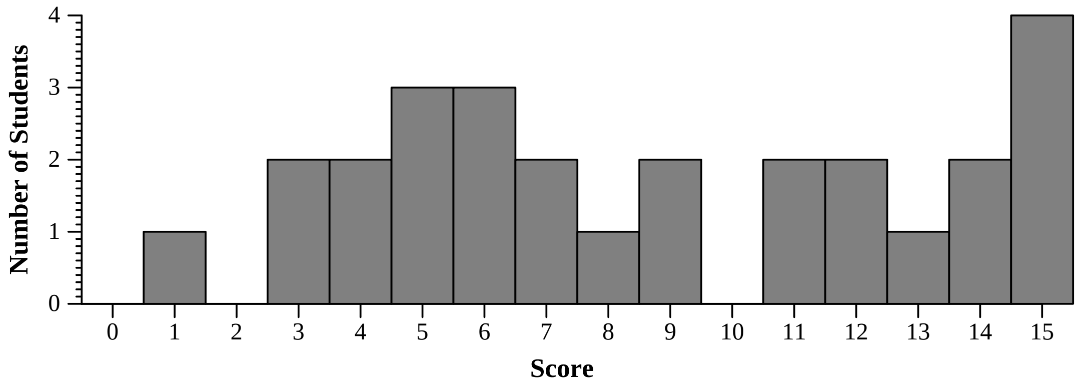
I anticipated that this question would be straightforward, since you all had written `jsh` previously. Your program just needs to act like a shell. It should first call `pipe`, and then fork off two processes – one for `a.out` and one for `sort`. The `a.out` process opens `input` and `dup`'s this file onto standard input, and the pipe onto standard output. The `sort` process opens `output` and `dup`'s this file onto standard output, and the pipe onto standard input. Both processes call `execlp`. Now, the main process must call `wait()` twice to wait for both of them to finish, and then can either fork off a process to execute `ls`, or it can just call `execlp` for `ls`. The code looks as below. Note the code below checks for errors. I didn't care whether or not yours did:

```
#include <fcntl.h>
#include <stdio.h>

main()
{
  int p[2];
  int fd;
  int statusp;

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

  if (fork() == 0) {  /* a.out process */

    fd = open("input", O_RDONLY);   /* Open and hook up input to stdin */
    if (fd < 0) { perror("open(input)"); exit(1); }
    dup2(fd, 0);
    close(fd);

    dup2(p[1], 1);   /* Hook up the pipe to stdout. Close pipe fd's */
    close(p[0]);
    close(p[1]);

    execlp("a.out", "a.out", 0);   /* Execute a.out */
    exit(1);                       /* Just in case the exec fails */
  }

  if (fork() == 0) {  /* sort process */
                                         /* Open and hook up input to stdout */
    fd = open("output", O_WRONLY | O_CREAT | O_TRUNC, 0666);
    if (fd < 0) { perror("open(output)"); exit(1); }
    dup2(fd, 1);
    close(fd);

    dup2(p[0], 0);   /* Hook up the pipe to stdout. Close pipe fd's */
    close(p[0]);
    close(p[1]);

    execlp("sort", "sort", 0);   /* Execute sort */
    exit(1);                       /* Just in case the exec fails */
  }

  close(p[0]);
  close(p[1]);
  wait(&statusp);   /* Wait for the two processes to finish -- we don't */
  wait(&statusp);   /* care about the order */
```

```
  execlp("ls", "ls", 0);    /* Execute ls -- alternatively, you could fork
                               off a child and have the child do it */
}
```

Grading of this question went as follows:

- Forking off a process for `a.out`: 4 points. This is divided into the following parts:
  - Calling `fork()`: .5 points
  - Opening/duping the input file: 1.5 points
  - Duping the pipe: 1 points
  - Calling exec: .5 point
  - Closing unnecessary fd's: .5 point
- Forking off a process for `sort`: 4 points. This is divided into the following parts:
  - Calling `fork()`: .5 points
  - Opening/duping the output file: 1.5 points
  - Duping the pipe: 1 points
  - Calling exec: .5 point
  - Closing unnecessary fd's: .5 point
- Calling `pipe()` in the right place was worth 1 point.
- Calling `exec` or `fork/exec` for `ls` was worth 1 point.
- Calling `wait()` was worth 1 points.
- Getting all the pieces together was worth 4 points.



Question 2 Histogram

Question 3: 25 points

Part 1

The two problems I was looking for are the following:

1. `free()` doesn't free anything. Thus, code that shouldn't run out of memory will.
2. This code calls `sbrk()` at every `malloc()` call. `sbrk()` is a system call, which means that this is very expensive. `malloc()` should do some buffering instead.

Some of you said that this `malloc()` would not return aligned pointers. This is a good answer, and you could use it in place of either of the above. Another decent answer is that if you call `malloc()` with a negative number it could wreak all sorts of havoc. I liked that. Finally, some of you said that when we run out of memory, `sbrk()` will return -1, instead of 0 like `malloc()` should. This got you a point, but not full credit as that's not really an important problem.

If you got one but not two of these, you received 3 points.

Part 2

`malloc()` acts as an interface to `sbrk()`, handling memory allocation with a free-list implementation. At all times, a list of chunks of free memory is maintained using a global variable. Initially, this list is empty. The list is implemented so that the first 12 bytes of each chunk contains the node information – a forward pointer, a backward pointer, and the size of the chunk. When `malloc(n)` is called, a chunk of at least `k` bytes is sought on the list, where `k` is eight plus `n`, padded to a multiple of eight. If no such chunk is found, `sbrk()` is called with to allocate a new huge chunk (i.e. 8K or so). Once a chunk is found, it is either broken into two pieces – `k` and the remainder – or kept whole if the remainder is under 12 bytes. The size `k` chunk is returned to the user, and the remainder is put onto the free list. The first 8 bytes of the `k`-sized chunk are initialized to contain the chunk size (`k`) and a checksum. A pointer to the 8th byte is returned to the user (giving him/her at least) `n` bytes aligned on an 8-byte quantity.

When `free(x)` is called, the chunk is put back onto the free-list. It knows the size of the chunk is at the address `x-8`. If the chunks adjacent to the newly freed chunks are also free, `free()` should coalesce them to minimize external fragmentation.

The grading here was generous – I took off points if you missed important things like where the size was kept, how `free()` found this size, etc.

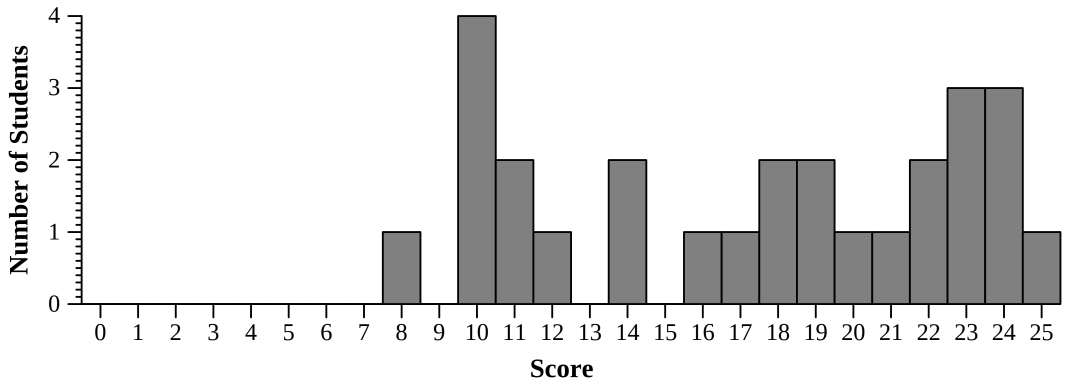
Part 3

The problem is internal fragmentation – 16-byte chunks are going to be allocated for each `twochar`. This means that you are going to use up 16M of memory for 2M of `twochar`'s. This is quite wasteful.

Part 4

You need to cut down the fragmentation. There are several ways to do this. One is to kill 4 of the bookkeeping bytes. Better is to deal with 2-byte quantities separately. The best thing is to allocate 4096-byte quantities and dole them out 2 bytes at a time. The problem is `free()` – how can it identify when its address is a

2-byte quantity, and then how does it mark it as free. You had to give some lip service to `free()` if you said to deal with 2-byte quantities separately. The way I'd do it is to keep the 4096-byte chunks of 2-byte quantities in a list, and free() would check this list to see if its address is in there. Then I'd maintain a 2048-entry bitmap for each 4096-byte chunk which denotes whether or not the entries are free. This would mean that for every 4096-byte chunk of 2-byte quantities, you'd have 8 bytes for a linked list structure, and 256 bytes for the bitmap. This is far less infrastructure than Unix's malloc which has 14 extra bytes for each 2 byte quantity, meaning 28,672 extra bytes for every 4096 bytes of `twochar`'s.



Question 3 Histogram

Question 4: 30 points

Part 1

The key here is to catch `SIGPIPE` and delete the current client file descriptor from the client list. Here's code that will do this:

```
int Sigpipe_caught;
pipe_handler()
{
  Sigpipe_caught = 1;
}

broadcast_info(Dlist clients)
{
  char line[100];

  signal(SIGPIPE, pipe_handler);

  while(fgets(stdin, line, 100) != NULL) {

    for (tmp = clients->flink; tmp != clients; tmp = tmp->flink) {
      Sigpipe_caught = 0;
      write(tmp->val, line, strlen(line));
      if (Sigpipe_caught) {
        close(tmp->val);
        tmp = tmp->blink;
        dl_delete_node(tmp->flink);
      }
    }
  }
}
```

You can also make `tmp` global and delete it in the handler. Note though that you have take care in deleting the node that `tmp` points to the right place (the node previous to the deleted one) so that the `for` loop will go the right place.

Grading for this question was as follows:

- Calling `signal` and setting up a handler: 1 point
- Deleting the node from the client list: 1 point
- Closing the socket: 1 point
- Making sure that `tmp` is updated properly: 1 point
- General approach: 3 points

Part 2

Now, you need to call `select` to read from two file descriptors at once: standard input, and the pipe. If `select` returns that standard input is ready to be read, then you read a line and broadcast the line as before. However, if `select` returns that the pipe is ready to be read, then you know two things – there is a newline on the pipe that needs to be read, and there is a client waiting for a connection, so you should call `accept_connection(s)`, and put the resulting file descriptor on the client list. The code is below:

```
broadcast_info(int *p, int s, Dlist clients)
{
  Dlist tmp;
  fd_set readset;
  int i;
  char line[100];

  signal(SIGPIPE, pipe_handler);

  while (1) {
    FD_ZERO(&readset);
    FD_SET(0, &readset);
    FD_SET(p[0], &readset);

    select(p[0]+1, &readset, NULL, NULL, NULL);
    if (FD_ISSET(p[0], &readset)) {
      i = accept_connection(s);
      dl_insert_b(clients, i);
    } else {
      if (fgets(line, 100, stdin) == NULL) {
        fprintf(stderr, "Stdin closed.  Bye\n");
        exit(1);
      }
      for (tmp = clients->flink; tmp != clients; tmp = tmp->flink) {
        Sigpipe_caught = 0;
        write(tmp->val, line, strlen(line));
        if (Sigpipe_caught) {
          close(tmp->val);
          tmp = tmp->blink;
          dl_delete_node(tmp->flink);
        }
      }
    }
  }
}
```

Grading for this question was as follows:

- Calling `select`: 1 point
- Dealing with the `fd_set`'s correctly: .5 points
- Checking for standard in and dealing with that properly: 1 point
- Checking for the pipe: 1 point
- Calling `accept_connection(s)`: 1 point
- Inserting the new fd onto the client list: 1 point
- Reading the newline: .5 points
- General form: 2 points

Part 3

This is a very subtle piece of code. First the program sets up a pipe and forks off a child process. The parent serves a socket at port 200, and loops as follows: It waits for a connection on the socket. When it receives a connection, it sends a newline along the pipe to its child. It also forks off another child process, which calls `request_connection` on port 5000. After this child receives the connection, it sends bytes from the connection at port 5000 to the client process that has called telnet. The initial child process serves a socket at port 5000 and runs `broadcast_info`.

Thus, when a client joins, the parent returns from `accept_connection()`, sends a newline to the `broadcast_info` process, and forks off a child process which connects to the `broadcast_info` process via port 5000, and shuttles bytes from that process to the client. When the client quits, the child process that is doing the byte-shuttling will generate a `SIGPIPE` signal and die. Then the `broadcast_info` process will generate a `SIGPIPE` signal and deal with it gracefully.

Why do we need this convoluted process structure? Because the `broadcast_info` process cannot call `select()` to wait for either stuff to come in on standard input or for someone to request a connection on port 200. Therefore, we add the extra parent process to wait on port 200. Since the connection is made with the parent, and since we want a connection to be made with the `broadcast_info` process, we fork off the extra process which shuttles bytes.

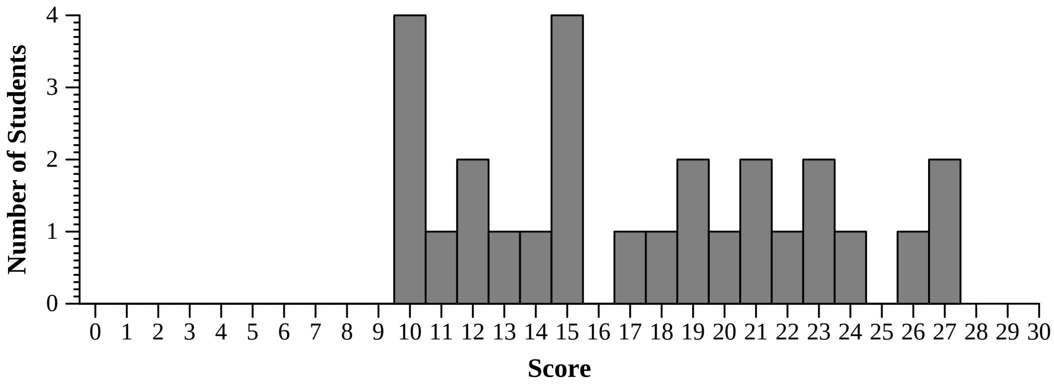
Part 4

There is one process forked off per client to do byte-shuttling. When this process dies (via `SIGPIPE`), it becomes a zombie, because its parent process is still alive and never calls `wait()`. However, the parent really can't know when the child dies, and can't call `wait()` indiscriminately, since it has to wait for `accept_connection()` to return. Therefore, to fix this, we can use the hack I described in class – fork off a child which forks off a grandchild that does byte-shuttling and then returns instantly. Since the child returns instantly, the parent process can wait for it, and the grandchild will then be inherited by `init`, and will not be a zombie when it dies. The new code, (starting at the `while(1)` statement) is as follows:

```
while(1) {
  i = accept_connection(mainsock);
  if (fork() == 0) {
    if (fork() == 0) {
      close(p[1]);
      j = request_connection("stocks.dowjones.com", 5000);
      while(read(j, &c, 1) != 0) write(i, &c, 1);
      exit(0);
    } else {
      exit(0);
    }
  } else {
    wait(&statusp);
    close(i);
    write(p[1], "n", 1);\
  }
```

Grading for this part was as follows:

- If you gave some coherent lip service to zombies, you received 1 point.
- If you stated why the problem existed, you received 1 points.
- If you mentioned that you had to call `wait`, you received 1 point.
- If you gave a working solution to fix the problem, you received 2 points.



Question 4 Histogram