Answers and Grading for CS360 Final, Spring, 1994

So, this test was way too hard. It would have made a good 3 hour general exam for graduate students. I apologize for that, but the grading was lenient, and the difficulty of the exam was considered when final grades were doled out. When you encounter an exam like this, the best thing to do is write as much as you can showing what you know, and trying to display that you are thinking along the correct lines. I know that's hard to do when under pressure, but it's the best thing. That said, I was pleased with the performance of the class as a whole. It looks like most of you understood a great deal of the contents of class, and that you have learned quite a bit about systems programming. Good luck with future classes and future endeavors in computer science. I enjoyed teaching you all.

Jim Plank

Question 1: 20 points

This question was to test your command of Unix primitives for process creation and maniplulation. The key was to open two pipes – one to connect stdout of the command with the return value of the caller, and one to connect stdout of the caller with stdin of the command. You also had to save standard output into another file descriptor when calling `jpopen()`, and restore it on `jpclose()`. That saving and restoration had to be performed with `dup()` and `dup2()`.

The Answer

```
int saved_fd;

int jpopen(char *command)
{
  int p1[2], p2[2];

  if (pipe(p1) < 0) { perror("jpopen: pipe(p1)"); exit(1); }
  if (pipe(p2) < 0) { perror("jpopen: pipe(p2)"); exit(1); }

  pid = fork();
  if (pid < 0) { perror("jpopen: fork"); exit(1); }

  /* If child, hook up stdout to p1, stdin to p2, close all extraneous
     fd's (all except 0, 1, and 2, and call exec */

  if (pid == 0) {
    dup2(p1[1], 1);
    dup2(p2[0], 0);
    close(p1[0]);
    close(p1[1]);
    close(p2[0]);
    close(p2[1]);
    execlp("/bin/sh", "sh", "-c", command, (char *) 0);
    perror("jpopen: execlp");
    exit(1);

  /* If parent, save the current stdout, hook up stdout to p2, and return p1 */

  } else {
    saved_fd = dup(1);  /* This saves the current stdout */
    dup2(p2[1], 1);
    close(p2[0]);
    close(p2[1]);
    close(p1[1]);
    return p1[0];
  }
}

/* This restores stdout from its saved value */

int jpclose(int fd)
{
  int status;
```

```
  wait(&status);
  close(fd);
  dup2(saved_fd, 1);
  close(saved_fd);
  return status;
}
```

Comments

So, this was a very difficult question. Grading was as follows. You started at a certain plateau:

- Plateau #1: Good overall form, two pipes: 20 points.
- Plateau #2: Good overall form, one pipe. 15 points.
- Plateau #3: Random: 7 pts.

After that, I took off deductions for the following:

1. Not closing extraneous files in `jpopen()`: 1 point.
2. Calling `pipe` after fork (that can't do anything): 1 point.
3. Calling `exec` in the parent, and returning in the child (that's wrong because the child can't call `wait()` for the parent: 1 point.
4. Mixing up the order of the `dup2` arguments was ok, unless you were inconsisent. If you did a `dup2(1, fd[1])`, I took off a point because it doesn't make sense to call `pipe(fd)` and then `dup2(1, fd[1])`.
5. Calling `dup2(fd[0], 1)` or `dup2(fd[1], 0)`: 1 point. This is bad because again, it makes no sense – fd[0] is read only, and stdout is write-only. Thus `dup2(fd[0], 1)` cannot do anything constructive.
6. No `jpclose()`: 1 point.
7. No attempt to reconstruct `stdout`: 1 point.
8. Using a global variable to save `stdout` – you got an extra point here if you did something vaguely correct with this.
9. Making non-error-flagging calls after `exec()`: 1 point.
10. No return values or other really lame syntax errors: 1 point.
11. Returning a write-only fd to the caller: 1 point.
12. No error-flagging. I didn't take off for lack of error-flagging.
13. For the really random answers, I took off one point each for omitting `pipe()`, `fork()`, `exec()`, `dup2()`, `wait()`, and `close()`.

Question 2: 25 points

This question was confused a bit by my use of the word "interactive." What I had meant was that the game was user-vs-computer – interactive in the sense that the computer will wait for input and produce output. Many of you took "interactive" to mean multi-user. I.e. the game is user-vs-user (or many users). I graded this so that you could make either assumption, although the question is kind of confusing if you made the latter assumption. In grading, I basically graded out of 28 points, and then gave the grade of `min(25, your score)`. The breakdown was 6 points for each distribution method, and 4 points for explaining a better method. I graded on whether you said reasonable things that were pertinent. If you made false statements, or said nothing really meaningful, I deducted points.

The Answer

1. **Exporting the executable**. This method has problems with the high-score file. If users have access to the file itself, they can only update it if it is writable by the universe. That is not good security. If they don't have access to the file itself, then they cannot be included in it.

   - Running the program – the user owns the program, and should have no problems running it (although if a binary is exported, users can only run it on the machine that supports the binary).
   - The high score file – discussed above. One thing that must be considered is that with a shared high score file, multiple users may try to write it simultaneously. Thus, you will have to use the `O_SYNC` flag when you open it.
   - Security – security is terrible. The high score file is essentially meaningless. For those who assumed the game was multi-user, a viable security problem is that users can alter their local version to cheat. Of course, if just the executable is exported, this will be pretty difficult, but not impossible.
   - Resources – with each user having a copy of the executable, there is significant disk waste if multiple users are on the same system. As each user can run on his/her own machine, however, processors shouldn't get bogged down.
   - Availability – the program is always available, since a user has his/her own copy of the executable. Some people said something about maintainability, which is that making updates to the game will be a nightmare, since it means getting a new executable to all users.

2. **Public executable with `setuid` bit set**: This solves some problems as now the high score file can be readable only by the owner of the executable, and everything will work.

   - How a user runs the program: He/she runs the public executable, perhaps on a local machine.
   - The high score file works as above, only it can be protected read-only to the universe, since access is restricted via the setuid bid.
   - Security. Security is much better. However, with all programs that use the setuid bit, the programmer must take care, lest a subtle bug give a user potential control over the programmer's file.
   - System resources – much better than above – only one version of the executable. Users can still execute the program on their own machines.
   - Availability – not good. Now, only users who have access to the programmer's files can play the game. At least before, anyone could play, although not everyone could get at the high score file.

3. **Iterative Server**: With an iterative server, any user can connect to the server and the server will execute the copy of the game. Since the user connects using `telnet` all he/she can do is send input to and receive output from the server. As the server is iterative, only one user can play at a time. This is really bad, because if one user has a network problem with the server, all other users are shut out of the game. How this method would work for a multi-user game is confusing, so I won't try to make something up. If you said something reasonable, I accepted it.
   - How a user runs the program – the user calls telnet.
   - How the high score file stays maintained – the server can do this. Since the server is iterative, the programmer doesn't have to worry about multiple-users changing the file at once. It would make programming easier.
   - Security – security is fine, unless you want to verify who users are. Most games don't attempt to do this.
   - System resources – fantastic – only one instance of the server is running at any one time. No exported executable or multiple copies – all users have telnet.
   - Availability – availability is terrible. Only one user can play at a time. Also, you can only play if your machine can talk with the server's.
4. **Concurrent Server**: With a concurrent server, each time a user connects to the server, a copy of `game` is forked off for the user. This is a nice interface, except that now the programmer's machine may have as many `game` processes has he has allowable socket connections. That's not too good for the programmer's machine.
   - How the user runs the program – same as above.
   - High score file – since there can be multiple `game` processes, the programmer has to be careful to use `O_SYNC` again.
   - Security – same as above.
   - Resources – bad for the server as there will be one process per user.
   - Availability – same as above.

A better way: The best way to deal with a program like this is to have each user have its own client code, and have the server perform minimal functionality. In this program, the only thing that you really need a server for is dealing with the high score file. Thus, you export an executable to users that lets them play the game, and when they need to update the high score file, they connect to a server. The server should be concurrent because otherwise, a hung connection to one user can affect other users. This uses up more disk space as each user needs his/her own executable, but it distributes processes well, and limits the server so that it only does what it has to.

Question 3: 25 points

Part 1: 10 Points

The Answer: From the March 14 lecture:

```
What setjmp does is saves the contents of the registers so that longjmp can restore them
later.  In this way, you "return" to the state of the program when setjmp was called.

#include <setjmp.h>
int setjmp(jmp_buf env);

This says to save the current state of the registers into env.  If you
look in /usr/include/machine/setjmp.h, you'll see that jmp_buf is defined as:

#define _JBLEN  9
typedef int jmp_buf[_JBLEN];

So, when you call setjmp, you pass it the address of an array, and it stores the value of
the registers in that array.  Setjmp returns 0 when you call it in this way.


longjmp(jmp_buf env, int val);

Longjmp states to reset the registers to the values saved in env.  THIS INCLUDES THE SP,
FP, and PC.  What this means is that longjmp doesn't return -- instead, when you call it,
you return as if you have just called the setjmp() that saved env.  This is because the
PC is restored along with the other registers.  Setjmp returns the 'val' argument of
longjmp, which is not allowed to be zero (read the man page).  Thus, you know when setjmp
returns a non-zero value that longjmp was called, and is returning to setjmp.
```

I changed `jmp_buf` a little, but the answer still applies. `setjmp` and `longjmp` are used most often to deal gracefull with error conditions when a program is deep in procedure calls. It can also be used to jump out of signal handlers.

Grading here is straightforward. I took off 2 points if you neglected to say how they are used.

Part 2: 5 Points

The Answer:

```
a=0
a=1
a=1. Done
```

I took of three points if you missed the second line, as that is pretty fundamental.

Part 3: 10 Points

So, the key here is to do three things:

- Reset the general purpose registers
- Reset the special registers
- Set the return value

This can be tricky. Especially because my `setjmp` didn't give you enough information. In particular, there was no way for you to reset the `pc` in `longjmp`. Instead, after saving `r9`, `setjmp` needed to save both the old `fp` and `pc`. Thus, after the following line:

```
st %r1 -> [r0+36]
```

there should also have been:

```
ld [fp+8] -> f1
st %r1 -> [r0+40]
```

(and `jmp_buf` should have been 11 words long).

Ok. Now, the form of `longjmp()` should be as follows: First, put the `env` argument into `r0`, and push the `val` argument onto the stack. Then restore registers `r1` through `r9`. Next, restore change the old `fp` and `pc` to which `longjmp` would normally return, to those saved in `env`. Then when `longjmp` returns, it will be using the return `fp` and `pc` for `setjmp`, thus returning from `setjmp` as desired. Finally, before calling `ret`, `r0` must be restored from the stack, so that the return value is the argument `val`.

Note that using the stack is really the only safe way to remember the return value. However, I accepted other ways that didn't quite work...

The correct code looks as follows:

```
ld [fp+12] -> %r0 / Put env into r0
ld [fp+16] -> sp-- / Push val onto the stack
ld [r0+4] -> %r2 / Restore the general registers except r1
ld [r0+8] -> %r3
ld [r0+12] -> %r4
ld [r0+16] -> %r5
ld [r0+20] -> %r6
ld [r0+24] -> %r7
ld [r0+28] -> %r8
ld [r0+32] -> %r9
ld [r0+36] -> %r1       / Put setjmp's return fp into this return fp
        st %r1 -> [fp+4]
ld [r0+40] -> %r1       / Put setjmp's return pc into this return pc
        st %r1 -> [fp+8]
ld [r0] -> %r1 / Restore r0
ld [++sp] -> %r0 / Pop the return value into r0
ret / Return to the caller of setjmp
```

The grading for this took account of my bug in `setjmp`. To get full credit you had to properly get `env` and `val` from the arguments, restore the general purpose registers, do something reasonable that showed you were attempting to reset the `fp` (and `pc`), and put `val` into the return value.

If you just restored the general purpose registers, and did it totally correctly, you received 5 points. Points were otherwise added or subtracted according to how well it seemed you understood what was going on.

Question 4: 30 points

Part 1: 10 points

The Answer: `malloc8()` and `free8()` maintain two data structures: a doubly linked list (dlist) and a red-black tree. Each dlist node contains a pointer to a structure with three fields – an 8K buffer, a 1024-bit bitmap, and an integer. The 8K buffer holds 1024 8-byte chunks, which are designated free or allocated by the appropriate bit in the bitmap – 0 means free, and 1 means allocated. The integer holds how many free 8-byte chunks are in the buffer. If a buffer is in the dlist, then there is guaranteed to be at least one free chunk in the buffer.

The rb-tree holds all structures ever created by `malloc8()`. The tree is keyed by the starting address of the structure.

Now, `malloc8()` initialized the dlist and rb-tree on its first invocation. After that, it checks to see if the dlist is empty. If so, there are no free 8-byte chunks. Thus, it calls `malloc()` to get a new struct. Now there are 1024 free 8-byte chunks. It puts this struct into the dlist. Now that there is guaranteed to be at least one element in the dlist, `malloc8()` finds a free 8-byte chunk in the first dlist node. That chunk is marked allocated, and if that was the last chunk in the struct (i.e. if `n_free == 0`), then the struct is removed from the dlist. Finally, a pointer to the chunk is returned.

`Free8()` must first find the struct corresponding to the given address. This is done using the rb-tree. `free()` does a bunch of error checking to make sure that the address is indeed in one of these structs. If not, it dumps core. Once it finds the struct, it frees the chunk in the bitmap by setting the proper element to zero. If the struct previously had no free chunks (i.e. if `n_free` was 0), then it is placed onto the dlist.

Grading: You had to come up with something reasonable that resembled the above.

Part 2: 10 points

The Answer: For megabytes of 8-byte chunks, `malloc8()` and `free8()` will show vastly less fragmentation than `malloc(8)` and `free()`. Why? External fragmentation for the two will be the same. They both allocate 8K buffers, and thus might on the order of 8K unused bytes due to buffering. However, the internal fragmentation is quite different. For each 8K of chunks allocated, `malloc8()` will have basically 200 bytes of infrastructure: 128 bytes for the bitmap, 4 bytes for `n_free`, 12 bytes for a dlist node, and 56 bytes for the rb-tree node. When `malloc(8)` is called, it returns a chunk of 8 bytes to the user prefaced by 8 bytse of infrastructure. This means that for every 8K of chunks allocated, `malloc(8)` will have 8K of infrastructure. This is horrendous, and a good reason to use something like `malloc8()` when you are allocating a lot of small chunks.

Grading: So the correct answer is that they are equal with respect to external fragmentation, and `malloc(8)` is much better with respect to internal fragmentation. I gave partial credit if you said reasonable things, but came to the wrong conclusions. I gave little credit for incorrect declarations.

Part 3: 10 points

`malloc8()` is $O(\log n)$. Look at all the things that `malloc8()` can do:

- Set up the initial dlist and rb-tree: $O(1)$.

- Malloc and initialize a new struct. This involves calling `malloc(8192)`, which is probably $O(1)$, although maybe is a little more because malloc has to do list traversal. Then clear the bitmap – this will take 128 operations, making it $O(1)$. Set `n_free`, and insert the struct into the dlist: both $O(1)$. Finally, insert the struct into the rb-tree – this will be $O(\log n)$, where n is the number of structs allocated. (This is equivalent to $O(\log n)$, where n is the number of 8-byte chunks allocated. Think about it).
- Find the first node in the dlist: $O(1)$.
- Find a free chunk: This will take a traversal of the bitmap, which will take at most 1024 operations if implemented poorly: Still $O(1)$.
- Set the chunk as allocated and return: $O(1)$.

Thus, `malloc8()` is $O(1)$ in the normal case, and $O(\log n)$ in the worst case.

`free8()` has a bunch of constant time operations and one rb-tree search. Thus it is $O(\log n)$.

Why is a rb-tree needed? Otherwise, you might have to search every struct that you have allocated to see if the memory to free is in that struct – this is $O(n)$, which is much worse thatn $O(\log n)$.

Grading: Again, you received partial credit for saying reasonable things, but not finding the right answer. Saying `malloc8()` was $O(n^2)$ without giving any reasons gained you little. Not saying what n was cost you a point. Not answering why a rb-tree was needed cost you 4 points.