CS360 Final Exam

May 5, 1995

2:45–4:45

Jim Plank

80 Points: Answer all questions.

Question 1: 5 points

"An orphan can never become a zombie."

Explain.

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.

Question 3: 15 points

Answer the following 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?
3. Describe the `select()` system call — why does `telnet` have to use `select()`, and how does it do so?

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

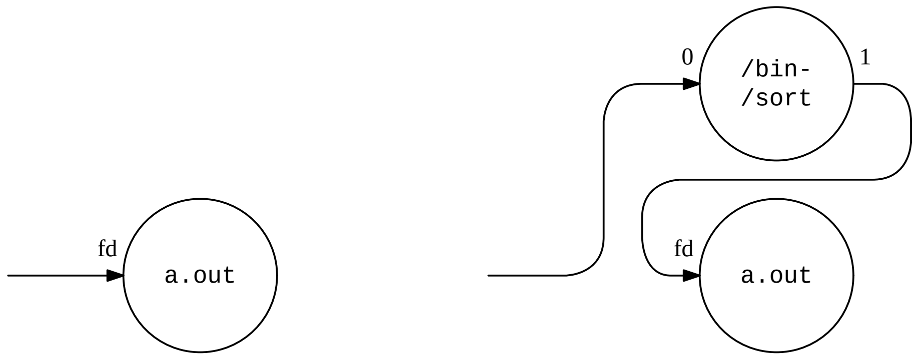
Question 5: 20 points

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 can't just spray-paint a building or steal a car like they do in other fraternities. 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()`.



Before
`ensure_sorted()`

After
`ensure_sorted()`

From string.h:

You should already know what these do:

- `char *strcmp(char *s1, char *s2);`
- `char *strcpy(char *s1, char *s2);`
- `char *strdup(char *s);`
- `int *strlen(char *s);`

From dlist.h:

```
typedef struct dlist {
  struct dlist *flink;
  struct dlist *blink;
  void *val;
} *Dlist;
```

- `Dlist make_dl()`: Mallocs and returns a new dlist
- `void dl_insert_b(Dlist d, void *val)`: Creates a new dlist node with val `val` and inserts it into the list before node `d`. If `d` is the head of the list, the new node is inserted at the end of the list.
- `void dl_delete_list(Dlist d)`: Deletes and frees the dlist whose head is the node `d`. Note that it does *not* free the `val` fields.
- Tree Traversal Macro:
  ```
  #define dl_traverse(ptr, lst) \
      for(ptr = lst->flink; ptr != lst; ptr = ptr->flink)
  ```

Syntax of various system calls:

```
int fork()
int execlp(char *file, char *arg0 [ , char *arg1,...  , char *argn ] (char *)0)
int wait(int *statusp)
int dup2(int fd1, int fd2)

     With dup2(), fd2 specifies the  desired  value  of  the  new
     descriptor.   If  descriptor  fd2  is  already in use, it is
     first deallocated as if it were closed by close(2V).

int pipe(int fd[2])
int select (int width, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)
char *sbrk(int incr)

     In function sbrk(), incr more bytes are  added
     to  the  program's  data space and a pointer to the start of
     the new area is returned.
```

puts/gets

```
puts(char *s)
char *gets(char *s)
```