CS360 Final Exam: May 6, 2004

Question 1

Suppose we have a program called **ticker**. You call it with no arguments. Whenever it receives a newline on standard input, it writes a floating point number between zero and one on standard output. That number will be written with exactly 7 decimal places, and will be followed by a newline. If it receives EOF on standard input, it quits. If it receives any other characters on standard input, it ignores them.

Your job is to write a program called **onesec**. **Onesec** forks off a child process that calls **ticker**, and sets up pipes so that it can both write to **ticker**'s standard input, and read from **ticker**'s standard output. Then **onesec** does the following. It iterates forever, doing the following steps:

- It sleeps for one second.
- Then it writes a newline to **ticker**'s standard input.
- Then it reads **ticker**'s standard output, and prints the number with exactly 5 decimal places (and a newline) to standard output.

Write **onesec**. Note, it will not require threads, sockets, or signal handlers.

Question 2

Write either a program or a shell command that will fork off a process running **ticker** that exits because it generates **SIGPIPE**. You may not use the system call **kill()** or the program **kill** to do this!

Make sure that there is no way for **ticker** to write to an existing and open pipe buffer (that will never be read), and then exit because it reads EOF.

Question 3

Behold the following program. Note, it **does not** have any typographical errors. Yes, it is not a great program.

```
main()
{
  int *a, *b, *c;
  int i;

  a = (int *) malloc(16);
  b = (int *) malloc(12);
  c = (int *) malloc(20);
  for (i = 0; i < 4; i++) a[i] = 100+i;
  for (i = 0; i < 5; i++) b[i] = 200+i;
  for (i = 0; i < 5; i++) c[i] = 300+i;
  free(b);
  b = (int *) malloc(8);
  free(a);
}
```

Suppose **malloc()** works as described in class:

- Its buffer size is 8192 bytes.
- It manages a doubly-linked, null terminated free list with no header node.
- It does not do coalescing or checksumming.
- It stores the size of the allocated buffer 8 bytes before the pointer that is returned.

Suppose that **sbrk(8192)** returns 0x6000.

Part A

Use the answer sheet to draw the values in memory between addresses 0x6000 and 0x6080 just before the first **free()** statement. You may assume that when **sbrk(8192)** returns, all those memory values are zero. Draw *every* memory value.

Part B

Use the answer sheet to draw the values in memory between addresses 0x6000 and 0x6080 right after the last **free()** statement. You may assume that when **sbrk(8192)** returns, all those memory values are zero. Draw *every* memory value.

Question 4

Circle the correct answer on the answer sheet:

Part A - *True or False:* In a non-preemptive thread system, you do not have to worry about race conditions.

Part B - *True or False:* A thread cannot see the variables on another thread's stack.

Part C - *True or False:* There are programs for which mutexes do not provide flexible enough synchronization operations.

Part D - *True or False:* **pthread_detach()** allows you to have a thread exit and have its resources be cleaned up without having any other thread call **pthread_join()** on it.

Part E - *True or False:* A thread may only call **pthread_join()** on threads which it has created with **pthread_create()**.

Part F - *True or False:* On our machines, a non-preemptive thread may be turned into a preemptive thread automatically by making a blocking system call.

Part G - *True or False:* With mutexes, you may have a thread execute instructions atomically with respect to other threads that lock the mutex.

Part H - *True or False:* It is a programming mistake to call **pthread_cond_signal()** on a condition variable that has no threads waiting on it.

Part I - *True or False:* There are programs that will work correctly in a non-preemptive thread system that will not work correctly in a preemptive thread system.

Part J - *True or False:* There are programs that will work correctly in a preemptive thread system that will not work correctly in a non-preemptive thread system.

Question 5

Suppose we write the following procedure to provide a more convenient mechanism to use **setjmp()/longjmp()** to catch errors.

```
jmp_buf *set_up_exception(char *id_string)
{
  jmp_buf *buf;

  buf = (jmp_buf *) malloc(sizeof(jmp_buf));
  if (buf == NULL) { perror("set_up_exception"); exit(1); }
  if (setjmp(*buf) == 0) return buf;
  fprintf(stderr, "Exception caught: %s\n", id_string);
  free(buf);
  return NULL;
}
```

We will use it in code like the following:

```
  ...
  buf = set_up_exception("Activity #1");
  if (buf != NULL) {
    do_activity(buf, ......);
  } else {
    clean_up_exception(....);
  }

  ...
```

In this code, if **do_activity()** or any of its subprocedures detects an error, it will call **longjmp(*buf, 1)** so that the error may be caught gracefully. The **longjmp()** will go to **set_up_exception()**, which will return **NULL**, and any clean-up code will be executed by **clean_up_exception()**.

There is a serious problem with this way of doing things. State what the problem is, and as precisely as you can what will happen when **do_activity()** or one of its subprocedures calls **longjmp()**.