Full Name: _____

CSCI 340, Fall 2015

Practice Midterm Exam

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name on the front. Put your name or student ID on each page.
- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.
- You may **not** use your text, notes, or any other reference material.
- No electronic devices (music, phone, calculator, etc).
- **Do not turn this page until instructed to do so.**

Advice:

- The amount of space after a question does not always indicate how long the answer should be. Sometimes I add space so questions fit well on pages.
- Some questions have multiple parts such as Explain your answer. Make sure you answer all the parts of each question.
- If you can't answer a question easily, move on and come back to it later.
- If you have time left over, use it to review your answers. Students who turn tests in early often make trivial mistakes that they would catch if they went back over their answers.
- If you think a question has a mistake make sure to ask me during the test. It is too late to have a clarification or fix the problem after the exam.

Problem	Page	Possible	Score
1	1	20	
2	2	25	
3	4	20	
4	5	15	
5	6	20	
Total		100	

1. **[20 Points]** This problem tests your understanding of process creation and event ordering

```
void end(void)
{
   printf("2");
}

int main()
{
  if ( fork() == 0 ) {
     atexit(end);
  }
  if ( fork() == 0) {
     printf("0");
  } else {
     printf("1");
  }
  exit(0);
}
```

The `atexit()` routine takes a pointer to a function adds that function to a list of functions (initially empty) to be called when the `exit()` function is called.

What outputs are possible? Circle the possible outputs in the list below.

(a) 112002

(b) 211020

(c) 122001

(d) 100212

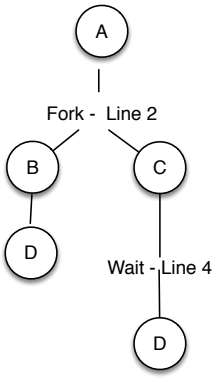
2. **[25 Points]** This problem tests your understanding of process creation and event ordering

```
void P(char *s) { fprintf(stderr,"%s ", s); }

main()
{
  int status;

  P("B");
  if (fork() ) {
    P("L");
    if ( fork() == 0 ) {
      P("G");
    } else {
      P("S");
      wait(&status);
    }
  } else {
    P("X");
    if ( fork() ) {
      P("W");
      wait(&status)
      P("Z");
    } else {
      P("T");
      if (fork()) {
         P("H");
      } else {
         P("M");
      }
      exit(0);
    }
  }
  P("R");
}
```

(a) **[10 Points]**

Draw a "process tree" diagram. Each `fork()` in the program should cause a fork in the tree labeled with the line number of the `fork()`. The output of print statements should be shown linearly beneath the tree in the order they would be output. `wait()` statements should also be included in the tree.

Model your diagram after the diagram on the right, which shows a program that prints "A", forks a child using an —verb—if(fork())—, has the child output "B" and the parent output "C" and then waits for the child; following the if-then-else is a statement to print "D".



(b) **[4 Points]** Line 14 contains a `wait` call. Other lines (8, 10, 18, 24) contain fork() statements that spawn child precesses. What are the line numbers of the corresponding `fork()` calls that create processes that might be successfully harvested by the `wait()` statement on line 14?

(c) **[4 Points]** Repeat part the previous problem for the `wait()` on line 20?

(d) **[4 Points]** If the program is executed, can W ever appear before H?

(e) **[4 Points]** Can Z ever appear before W?

(f) **[4 Points]** Can R ever appear before L?

3. **[20 Points]** Consider the following C program. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
pid_t pid;

void bar(int sig) {
  printf("wabbit");
  kill(pid, SIGUSR1);
}

void baz(int sig) {
  printf("tweribble");
}

void foo(int sig) {
  printf("waskly");
  kill(pid, SIGUSR1);
  exit(0);
}

main() {
  signal(SIGUSR1, baz);
  signal(SIGCHLD, bar);
  pid = fork();
  if (pid == 0) {
    signal(SIGUSR1, foo);
    kill(getpid(), SIGUSR1);
    for(;;);
  }
  else {
    pid_t p; int status;
    if ((p = wait(&status)) > 0) {
      printf("that");
    }
  }
}
```

What is the output string that this program prints?

4. **[15 Points]**

(a) **[5 Points]** What is Virtual Memory?

(b) **[5 Points]** What is it called when we resolve a virtual memory address and it's in physical memory?

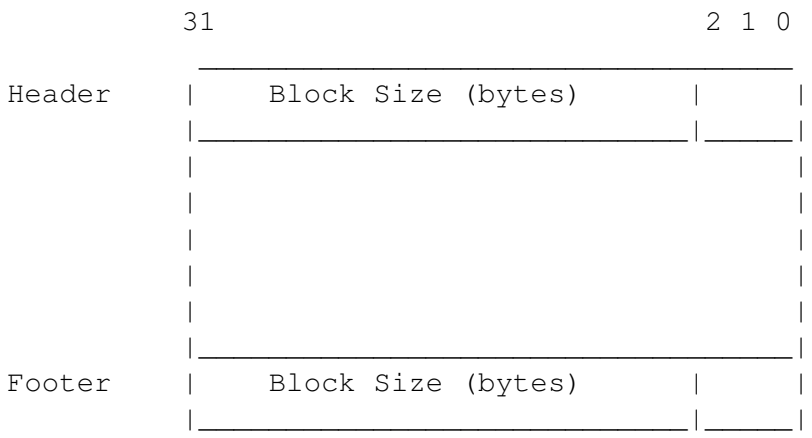
(c) **[6 Points]** What are the is it that Virtual Memory and Caching take advantage of that makes them work?

5. **[20 Points]**

Dynamic storage allocation

The following problem concerns dynamic storage allocation.

Consider an allocator that uses an implicit free list. The layout of each allocated and free memory block is as follows:



Each memory block, either allocated or free, has a size that is a multiple of eight bytes. Thus, only the 29 higher order bits in the header and footer are needed to record block size, which includes the header and footer. The usage of the remaining 3 lower order bits is as follows:

- `bit 0` indicates the use of the current block: 1 for allocated, 0 for free.
- `bit 1` indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
- `bit 2` is unused and is always set to be 0.

Given the contents of the heap shown on the left, show the new contents of the heap (in the right table) after a call to `free(0x400b010)` is executed. The size is indicated in a decimal value and the allocation bits are in binary. Note that the address grows from bottom up. Assume that the allocator uses immediate coalescing, that is, adjacent free blocks are merged immediately each time a block is freed.

Address	Size	Alloc Bits
0x400b028	16	010
0x400b024	n/a	n/a
0x400b020	n/a	n/a
0x400b01c	16	010
0x400b018	16	011
0x400b014	n/a	n/a
0x400b010	n/a	n/a
0x400b00c	16	011
0x400b008	16	011
0x400b004	n/a	n/a
0x400b000	n/a	n/a
0x400affc	16	011

Address	Size	Alloc Bits
0x400b028		
0x400b024	n/a	n/a
0x400b020	n/a	n/a
0x400b01c		
0x400b018		
0x400b014	n/a	n/a
0x400b010	n/a	n/a
0x400b00c		
0x400b008		
0x400b004	n/a	n/a
0x400b000	n/a	n/a
0x400affc		