Name:

COS 217 Final
Fall 2008
January 20, 2009

Pledge:

Directions:

- Please answer each question in the space provided. The amount of space should be sufficient for a correct answer. If you need more space, please use the backs of pages, and make a note to that effect. If you run out of space, exam books are provided at the front of the room.
- This exam is open-book, open-notes, and is covered by the Honor Code. No electronic devices may be used for this exam. Please write and sign the pledge after you finish your exam.
- There are a total of five sections, with the number of points for each shown by the question. While it is not the intent for the exam to be a race, spending too much time on a single question may preclude finishing the exam. Budget your time wisely.
- To be fair, I will try to avoid answering content-related questions during the exam, unless it's to correct a mistake on my part.
- If you feel that a question **requires** additional assumptions or information to answer, please state them. Your guiding principle should be *Occam's razor*, which loosely translated states that you should allow as few assumptions as necessary to explain the situation.
- Answers should assume C/Unix and our standard development environment unless otherwise stated or implied
- Please first read over the entire exam and then begin to answer questions. I will wait outside the exam room for the first twenty minutes, and then will be available in my office (room 322).
- Please write legibly

#	Name	Points Available	Score
1	True or False	20	
2	Short answer	21	
3	Code reading	18	
4	Bug hunt	20	
5	Putting It Together	21	
	Total	100	

1. True or False (20pts) For each statement, write "true" if you believe the statement is correct, or false if you believe the statement is incorrect. If you believe the statement does not have a clear answer, give whichever choice is more appropriate and explain why.

- for maximum flexibility, programmers use unconditional jumps for function calls and returns

- placing commonly-used variables in registers typically improves program speed

- programmers typically try to underflow buffers to save memory

- using big-endian byte order provides space for larger numbers

- the number of registers in a machine is determined by the amount of RAM installed

- the `wait()` system call makes computers slower and should be avoided when possible

- memory fragmentation can waste resources, leading to slower programs

- right-shifts on unsigned integers is undefined

- signals are the mechanism the programmer uses to tell the operating system what actions to perform

- page tables store information about the virtual memory pages

2. Short Answer (21pts) Answer each item *well* in no more than 3 sentences. Each question is worth 3 points.

- A programmer replaces `pushl $0` with `subl $4, %esp` to allocate a variable on the stack. Give one side-effect of this change.

- Give a second side-effect of the change mentioned above.

- Why might a "best-fit" allocation strategy for malloc be a good idea?

- Why might a "best-fit" allocation strategy for malloc be a bad idea?

- What's worse for instructions, **and why** – a 1% page fault rate or a 10% cache miss rate?

- What are callee-saved registers?

- Why are registers split into caller-saved and callee-saved?

3. Code reading (18pts)

The following assembly code was automatically generated by compiling a very short, nonsensically-named, C function that takes a single parameter of type unsigned int and has a return type of type int. You may assume the formal parameter passed in is always in the range of zero to one hundred.

```
        .file   ‘‘wipe.c’’
        .text
.globl wipe
        .type   wipe, @function
wipe:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $0, -4(%ebp)
        jmp     .L2
.L3:
        movl    8(%ebp), %eax
        andl    $1, %eax
        testb   %al, %al
        je      .L4
        movl    -4(%ebp), %eax
        addl    8(%ebp), %eax
        movl    %eax, -4(%ebp)
.L4:
        subl    $1, 8(%ebp)
.L2:
        cmpl    $0, 8(%ebp)
        jne     .L3
        movl    -4(%ebp), %eax
        leave
        ret
        .size   wipe, .-wipe
        .ident  ‘‘GCC: (GNU) 4.1.2 20071124 (Red Hat 4.1.2-42)’’
        .section        .note.GNU-stack,’’’’,@progbits
```

Here are some hints to help understand this code:

1. register “al” is just the low-order byte of the A register
2. “testb” performs a bitwise AND of the two operands, sets the flags, and discards the results.
3. “andl” is the bitwise AND operation
4. “leave” releases the stack frame, copying EBP into ESP

For the following questions, give answers in plain English, in brief, high-level descriptions. Each question is worth 3 points.

- What is stored in 8(%ebp)?

- What is stored in -4(%ebp)?

- The two instructions right after L2 perform one task. What is it?

- Describe the conditions necessary for the "je" after L3 to fail.

- When "je" fails and the three instructions after it are executed, what is happening?

- Give a high-level summary of what this code does. You should be able to state it in one or two sentences.

4. Bug hunt! (20pts)

The following program is supposed to execute three commands, wait for a while, and repeat the process. However, it's full of bugs.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static void
SigChildHandler(int sig)
{
  /* hint: this usage of waitpid is correct */
  while (waitpid(-1, NULL, WNOHANG) > 0)
    ;
}

int
main(int argc, char *argv[])
{
  int i;
  static char *commands[] = {
    "ls",
    "df",
    "quota"
  };

  while (1) {
    for (i = 0; commands[i] != NULL; i++) {
      if (fork(commands[i]) == 0) {
        char args[2] = {commands[i], NULL};
        if (execvp(commands[i], args) == 0)
          printf("success\n");
      }
    }
    for (i = 0; i < 10; i++)
      sleep(1);
  }
  return 0;  /* hint: just to shut up the compiler warning */
}
```

On the following pages, identify **and briefly explain** five bugs, possible bugs, and design flaws. Do **NOT** try to identify style flaws. Fix the program so that it works. Only the first five bugs you report/fix will be considered.

this page left intentionally blank to answer the previous question

this page left intentionally blank to answer the previous question

5. Putting It Together (21pts)

Given what you know about the system as a whole, give well-rounded answers to the following questions:

- (7 points) Modern memory allocators typically avoid using brk/sbrk, and instead use mapped files. Describe a scenario (explaining each step) in which brk/sbrk interact poorly with demand-paged virtual memory.

- (7 points) Some modern processors and operating systems disable the ability to execute instructions on the stack. How does this affect attacks on the system, and what kinds of workarounds are possible for the attacker?

- (7 points) Describe **efficient** changes to a K&R-style memory allocator to detect underflow and overflow in allocated buffers. How much additional processing is required? Will all overflows and underflows be caught? If not, describe what will be missed.