Princeton University
COS 217 Spring 2017 Final Exam

Name: Login Id:

Precept (circle one): P01 (MW 1:30) Bob Dondero P02 (MW 3:30) Bob Dondero
P03 (TTh 12:30) Ananda Gunawardena P04 (TTh 1:30) Ananda Gunawardena
P06 (TTh 7:30) Huilian (Sophie) Qiu

The exam is covered by the Honor Code. Write and sign the pledge after you finish your exam.
The pledge is "I pledge my honor that I have not violated the honor code during this examination."

Pledge and Signature:

The exam is closed-book, closed-notes, and no cheat sheet is allowed. Use of any electronic device is
prohibited.

There are seven questions – budget your time wisely. Show your work where possible.
If you use extra sheets of paper, indicate this clearly and write your name on every extra page.

All questions are in the context of the CourseLab/Linux/C/gcc217 environment unless otherwise
stated.

For instructor use only:

<table>
<thead>
<tr>
<th>Question</th>
<th>Max Points</th>
<th>Points Earned</th>
<th>Question</th>
<th>Max Points</th>
<th>Points Earned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td></td>
<td>5</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td></td>
<td>6</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td></td>
<td>7</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td></td>
<td>TOTAL</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
**Question 1: Who Dunnit? (15 Points)**

For each operation shown in the table (second column), indicate the component that implements the operation (in the first column). The choices for components are:

- Preprocessor (P)
- Compiler (C)
- Assembler (A)
- Linker (L)
- Operating System (OS)
- Hardware (HW)

Notes:
- More than one component may apply.
- Although every software operation is ultimately executed on hardware, you should indicate HW only where it *directly* performs that operation.

<table>
<thead>
<tr>
<th></th>
<th>Operation Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Inserts code from contents of other files when building a program</td>
</tr>
<tr>
<td>2.</td>
<td>Resolves external references</td>
</tr>
<tr>
<td>3.</td>
<td>Performs macro substitutions</td>
</tr>
<tr>
<td>4.</td>
<td>Generates relocation records</td>
</tr>
<tr>
<td>5.</td>
<td>Determines which variables go into which registers</td>
</tr>
<tr>
<td>6.</td>
<td>Performs read from a register</td>
</tr>
<tr>
<td>7.</td>
<td>Performs context switches between processes</td>
</tr>
<tr>
<td>8.</td>
<td>Generates assembly code</td>
</tr>
<tr>
<td>9.</td>
<td>Invokes an exception handler when an exception occurs</td>
</tr>
<tr>
<td>10.</td>
<td>Checks L1/L2 cache when there is a memory access</td>
</tr>
<tr>
<td>11.</td>
<td>Patches object code with memory addresses</td>
</tr>
<tr>
<td>12.</td>
<td>Manages the process control block for each process</td>
</tr>
<tr>
<td>13.</td>
<td>Removes comments from program</td>
</tr>
<tr>
<td>14.</td>
<td>Combines multiple .o files into an a.out</td>
</tr>
<tr>
<td>15.</td>
<td>Consults page mapping table to check if virtual page is in physical memory</td>
</tr>
</tbody>
</table>
Question 2: ~140 characters (15 Points)

2a) (4 points) Suppose we have an initialization `char *p = "goTigers";`

What does `putc(*(p+3), stdout)` print? ______________________

What does `putc(*(p)+3, stdout)` print? ______________________

2b) (2 points) What does `printf("%d", 6^7^8)` print? (* is bit-wise xor.)

2c) (3 points) In 8-bit two’s complement, what decimal number does 11111101 represent?

2d) (1 point) In which class of exceptions (interrupt, trap, fault, abort) could control return back to the same instruction in a process?

2e) (2 points) Why is calling a `system` function (such as `sbrk`) more expensive than calling a function within an application program?

2f) (2 points) Give one example of a C standard I/O library function and the corresponding Unix system function that it calls, where some calls to the system function are avoided.

C I/O function: _________________________________

Unix system function: _______________________________

2g) (1 point) What kind of code vulnerability was exploited by the Heartbleed bug?
**Question 3: Going Virtual (14 points)**

Consider a system with 64-bit virtual addresses, a 16 KB page size, and 8 GB of physical memory. (Recall that \( K = 2^{10}, \ G = 2^{30} \).)

3a) (1 point) How many bits are there in the physical address? (Give answer in decimal.)

3b) (1 point) How many (maximum) physical pages can be resident in physical memory? (Give answer as a power of 2.)

3c) (4 points) Consider a virtual address \( 0x0000 \ 0000 \ 0011 \ FF11 \). Write the virtual page number and the offset in the virtual page (in hexadecimal)

| Virtual page number: | ________________________________ |
| Offset in page:       | ________________________________ |

3d) (3 points) List three distinct cases that can arise when a process tries to access memory at some virtual address, depending on its mapping in the page table. State in each case (in one sentence) what happens afterward.
3e) (3 points) For this part, assume that only 4 pages can fit into physical memory at one time. Suppose there is only one process in the system, and it accesses virtual page numbers (with valid addresses) in the sequence shown below:

1, 5, 2, 3, 5, 2, 4, 1, 2

Assume that initially none of these pages is mapped to physical memory. Also assume that an LRU (least recently used) policy is used, i.e., the physical page that was the least recently used is evicted to bring in a new page from disk into memory. (Don’t worry about the specific physical page that a virtual page is mapped to – it is enough to know that some mapping exists when the page resides in physical memory.)

For each access in the sequence, as shown below, state whether there will be a page fault or not.

Virtual page  Page fault: Yes/No?

1
5
2
3
5
2
4
1
2

3f) (2 points) When a parent process forks a child process, the entire address space is duplicated (text, bss, data, rodata, stack, and heap sections). Explain briefly (in two sentences or less) how use of virtual memory makes this efficient.
Question 4: Powering through Assembly (20 points)

The following assembly code (using x86-64 architecture and assembly language) shows two functions \( g \) and \( f \) (line numbers are provided for reference).

```assembly
section ".text"
 g:
    movl $1, %eax
    movl $1, %r10d
    movl $2, %r11d
    .L1:
    cmpl %edi, %r10d
    jbe .L2
    jmp .L3
    .L2:
    mull %r11d
    addl $1, %r10d
    jmp .L1
    .L3:
    ret
 f:
    movl $0, %r10d
    movl %edi, %r11d
    .L4:
    cmpl %esi, %r11d
    jbe .L5
    jmp .L6
    .L5:
    movl %r11d, %edi
    pushq %r10
    pushq %r11
    call g
    popq %r11
    popq %r10
    addl %eax, %r10d
    addl $1, %r11d
    jmp .L4
    .L6:
    movl %r10d, %eax
    ret
... 
```

Important:
- You can assume that all integer variables and return values in the program have type `unsigned int`.

Other useful notes:
- Arguments are passed in registers: rdi, rsi, rdx, rcx, r8, r9
- Callee-saved registers are: rbx, rbp, r12, r13, r14, r15
- Caller-saved registers are: rdi, rsi, rdx, rcx, r8, r9, rax, r10, r11
- The instruction `mull src` multiplies %eax and src and puts the result back in %eax.

Answer the following questions *concisely* after studying the assembly code.
(You can use blank pages at the end of the exam as scratch paper, but copy your answers here.)
4a) (1 point) How many parameters does function $g$ have?

4b) (5 points) Translate the function $g$, first into flattened C, and then into idiomatic C. The C code must be such that our C compiler might reasonably have generated the given assembly language code. (Give meaningful names to variables and parameters.)

Flattened C for $g$

Idiomatic C for $g$

4c) (2 points) What is the result computed by function $g$, in terms of its input parameter(s)?
4d) (2 points) Note that functions g and f use registers r10 and r11. Why does f need to save them on the stack (and restore them) whereas g does not?

4c) (5 points) Now translate the function f into flattened and idiomatic C; it has two parameters and calls function g. (It is shown here again, for your convenience. Follow the earlier directions and give meaningful names to variables and parameters.)

```c
f:
    movl $0, %r10d
    movl %edi, %r11d
.L4:
    cmp %esi, %r11d
    jbe .L5
    jmp .L6
.L5:
    movl %r11d, %edi
    pushq %r10
    pushq %r11
    call g
    popq %r11
    popq %r10
    addl %eax, %r10d
    addl $1, %r11d
    jmp .L4
.L6:
    movl %r10d, %eax
    ret
```

Flattened C for f
4f) (2 points) Suppose `main` (not shown here) calls \( f(2,3) \). What is the result returned by \( f \)?

4g) (2 points) Rewrite the function \( f \) in C (not in assembly) that gets rid of the loops completely. 

*(Hint: This does not use loop unrolling! Don’t spend too much time on this part, return to it later.)*
**Question 5: Process this, Child! (8 points)**

Consider the code shown below. The goal here is to apply the Linux command `wc` (which prints number of lines, words, and characters in a given file) to multiple files. Each application of `wc` should be in a new child process. The multiple files are specified as arguments of the given program. For example, after the program is compiled, `.a.out f1 f2 f3` should result in applying `wc` to each of the files `f1`, `f2`, and `f3`. (No other arguments to `wc` are needed.)

```c
1 int main (int argc, char *argv[]) {
2    int i;
3
4    for (i=1; i < argc; i++) {
5        pid_t pid;
6        char *newArgs[3];
7
8        newArgs[0] = "wc";
9        newArgs[1] = ________________;
10       newArgs[2] = ________________;
11
12        pid = fork();
13        if (pid != 0) {
14            wait(NULL);
15            continue;
16        }
17        execvp(newArgs[0], newArgs);
18        exit(EXIT_FAILURE);
19    }
20    exit(EXIT_SUCCESS);
21 }
```

Assume that all needed header files are included and there are no bugs in the shown code.

5a) (4 points) Fill in the blanks at lines 9 and 10, so that the program works as required.

Line 9: ________________________________

Line 10: ________________________________

5b) (2 points) What is the purpose of the `wait` statement at line 14, i.e., how does it affect the output of the program? (*Hint: think about when `wc` might complete on different files.*)

5c) (2 points) Suppose we comment out the `continue` statement at line 15. How would the program behave then, i.e., describe what output would be written in terms of the argument files?
Question 6: Map the Stars (14 points)

Recall the interface for the SymTable ADT you used in Assignment 3, shown below:

typedef struct SymTable *SymTable_T;

SymTable_T SymTable_new(void);
void SymTable_free(SymTable_T oSymTable);
int SymTable_getLength(SymTable_T oSymTable);
int SymTable_put(SymTable_T oSymTable, const char *pcKey, const void *pvValue);
void *SymTable_replace(SymTable_T oSymTable, const char *pcKey, const void *pvValue);
int SymTable_contains(SymTable_T oSymTable, const char *pcKey);
void *SymTable_get(SymTable_T oSymTable, const char *pcKey);
void *SymTable_remove(SymTable_T oSymTable, const char *pcKey);
void SymTable_map(SymTable_T oSymTable, void (*pfApply)(const char *pcKey, void *pvValue, void *pvExtra), const void *pvExtra);

Consider a client baz.c of this ADT, where some parts of the code are shown on the next page.

• This client is similar to buzz.c (discussed in the lecture). It reads the input and creates a symbol table with bindings, where a given word is the key, and the value is the word’s count, i.e., the number of times the word appears in the input. Then, instead of printing the top 25 words by count (as done by buzz), baz prints statistics about “star” words, where a word is considered a star if its count >= 1000.

• The function readInput(table) reads words from stdin until end-of-file, and stores the word (key) and its count (value) in a binding in table. (This is the same function used in client buzz.c.) You do not have to compose readInput(table).

• You have to compose the callback function updateStars and answer some questions.
# client baz.c */

#include "symtable.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>

enum {CUTOFF = 1000}; /* a word is a star if its count >= CUTOFF */

struct starStats { /* struct for star statistics */
    int numw; /* number of star words */
    int sumc; /* sum of counts for star words */
    int maxc; /* max count among star words */
    const char *max_word; /* word with max count among star words */
} /* function declarations */
static void readInput (SymTable_T table);
static void updateStars(const char *key, void *value, void *extra);

/* main function
    Read words from stdin until end-of-file.
    Print Star words and their counts to stdout, print statistics to stdout.
    Return 0 if successful and EXIT_FAILURE otherwise. */
int main(void) {
    struct starStats *ps;
    /* create a symbol table and read input to add (word, count) pairs */
    SymTable_T table = SymTable_new();
    if (table == NULL) exit(EXIT_FAILURE);
    readInput(table);

    /* create a starStats structure and initialize it */
    ps = (struct starStats *) malloc(sizeof(struct starStats));
    if (ps == NULL) exit(EXIT_FAILURE);
    ps->numw = 0;
    ps->sumc = 0;
    ps->maxc = 0;
    ps->max_word = NULL;

    /* apply callback function updateStars on each binding in the table */
    SymTable_map(table, updateStars, (void*)ps);

    /* print statistics for star words */
    printf("Star words: total number %d, average count %d\n",
            ps->numw, ps->sumc/(ps->numw));
    printf("Star word %s has maximum count %d\n", ps->max_word, ps->maxc);

    /* clean up */
    SymTable_free(table);
    free(ps);
    return 0;
}

...
6a) (10 points) Compose the callback function `updateStars` according to the specification given in its comment below. Your code should be well-styled, except that no additional comments are needed. Use the blank pages at the end if you need to, and put a box around your answer.

```c
/* callback function in SymTable_map to apply on each binding in the table. The extra parameter points to a starStats structure. The key parameter points to a word and the value parameter points to the word’s count. If the word is a star, i.e., its count >= CUTOFF, then update the starStats structure. */

static void updateStars(const char *key, void *value, void *extra) {
    /* your code goes here */
}
```
6b) (2 points) Indicate a runtime bug that could occur in the given client code, and state the conditions under which the bug would manifest itself.

6c) (2 points) What changes would be needed in your callback function updateStars if the implementation of the symbol table is changed from using a linked list to using a hash function?
**Question 7: Exit through the Course (14 points)**

Consider the code shown below (assume all needed header files are included). For the purpose this question, assume that buffers are not flushed at arbitrary times, but only at well-defined times as described in lectures.

```c
1 void hello(void) {
2    int x = 1;
3    if (fork() == 0) {
4        x++;
5        printf("%d", x);
6        /* fflush(stdout); */
7        if (fork() == 0) {
8            x--;
9            printf("%d", x);
10       }
11    } else {
12        wait(NULL);
13        printf("%d", x);
14        printf("%d", x);
15    }
16 } else {
17    wait(NULL);
18    printf("%d", x-1);
19 } }
20 }
21 int main(void) {
22    printf("COS ");
23    fflush(stdout);
24    hello();
25    printf("\n");
26    return 0;
27 }
```

7a) (1 point) How many times is newline ("\n") printed (line 26)?

7b) (6 points) What is the output of the program? (Put a box around it.)
7c) (2 points) Suppose line 6 is uncommented, i.e., the statement `fflush(stdout);` is in the program. Again, show the output of the program (put a box around it).

7d) (5 points) Keeping the uncommented line 6 in the program, modify the program by adding `exit(0)` statement(s), such that the program prints the following output (where `\n` indicates that it ends with a newline).

```
COS 217 ;0\n```

You are allowed to add `exit(0)` multiple times.
Indicate the line numbers clearly, e.g., add `exit(0)` after line XX, after line YY, after line ZZ etc.
(Blank extra page)