University of Maryland College Park

Dept of Computer Science

CMSC216 Fall 2013

Midterm III Key

Last Name (PRINT): ____

First Name (PRINT): ____

University Directory ID (e.g., umcpturtle)______________________________________

I pledge on my honor that I have not given or received any unauthorized assistance on this examination.

Your signature: ____

Lab TA (Circle One):

Andrew(0103(10am) /0201(11am)) Brendan (0301(2pm)) Eric (0302(3pm)) Kevin (0304(1pm))

Luke (0303(4pm)) Steven (0101 (8am)/0102(9am)) Xu (0202(12pm)/0203(1pm))

Instructions

- If the answer to a question depends on the architecture involved, assume the question applies to behavior on linux.grace.umd.edu.
- This exam is a closed-book and closed-notes exam.
- Total point value is 200 points.
- The exam is a 50 minutes exam.
- Please use a pencil to complete the exam.
- WRITE NEATLY.
- You don't need to use meaningful variable names; however, we expect good indentation.

Grader Use Only

#1	Problem 1 (Process Control)	(90)	
#2	Problem 2 (Assembly)	(110)	
Total	Total (200)	(200)	

Problem #1, Process Control (90 pts)

Write a program that computes the average of integer values in a file. For this problem:

- You will use only two processes: the parent and a child.
- The parent process will:
  - Read the name of the file with integer values to average. Use the message “Enter filename:”
  - Create a child
  - Receive the average value computed by the child using a pipe
  - Display the message “Average value for file ***FILENAME*** is ***AVERAGE_VALUE*** where ***FILENAME*** and ***AVERAGE_VALUE*** represent the filename and average value, respectively
- The child process will:
  - Compute the average by calling the **compute_average()** function. **NOTICE THAT YOU MAY NOT MODIFY THIS FUNCTION.** If you do, you will not receive credit for this problem
  - Return to the parent the average by using a pipe
- **The parent will never open any file.**
- The child will open the file.
- You can assume the pipe can handle any amount of data we sent.
- You may not add any functions beside main().
- You can assume all system calls are successful (no need to print error messages).

Below you will find an example of running the program you are expected to write. The executable’s name is average, the file data.txt has the values to average, and the unix prompt is represented by a %.

```
% cat data.txt
2
4
12
% average
Enter filename: data.txt
Average value for file data.txt is 6
%
```

```
#define OPEN_FLAGS O_RDONLY
#define MAX_LENGTH 80

/* YOU MAY NOT MODIFY THIS FUNCTION */
int compute_average() {
   int count = 0, value, sum = 0;
   while(scanf("%d", &value) != EOF) {
      sum += value;
      count++;
   }
   return sum / count;
}

int main() {
   /* FUNCTION YOU NEED TO IMPLEMENT */
   return 0;
}
```

One Possible Answer:

```
int main() {
   pid_t child_pid;
   int pipe_fd[2], average;
   char filename[MAX_LENGTH + 1];

   printf("Enter filename: ");
   scanf("%s", filename);

   if (pipe(pipe_fd) < 0) { err(EX_OSERR, "pipe error"); }
   if ((child_pid = fork()) < 0) { err(EX_OSERR, "fork error"); }

   if (child_pid != 0) {  /* parent's code */
      close(pipe_fd[1]);  /* closing pipe's write end */
      read(pipe_fd[0], &average, sizeof(int));
      wait(NULL);  /* reaping */
      printf("Average value for file %s is %d\n", filename, average);
   } else {               /* child's code */
      int file_fd;
      if ((file_fd = open(filename, OPEN_FLAGS)) < 0) {
         err(EX_OSERR, "open file error");
      }
      if (dup2(file_fd, STDIN_FILENO) == -1) {
         err(EX_OSERR, "dup2 failed");
      }
      close(pipe_fd[0]);  /* closing pipe's read end */
      average = compute_average();
      write(pipe_fd[1], &average, sizeof(int));
      exit(0);
   }

   return 0;
}
```

Problem #2, Assembly Programming (110 pts)

Write assembly code that corresponds to the **sum_of_difference** function below. The function computes the sum of the difference of elements from two arrays. For example, the main() function below displays **16** ((4 -1) + (11 - 4) + (8 – 2)).

- You only need to provide assembly code for the **sum_of_difference** function.
- Do not provide assembly code for main(). We have provided the main function to illustrate how the function can be used.
- On the next page you will see assembly representing the arrays.
- Your assembly code must work for any two arrays of at least one element.
- You can assume the **elements** parameter will not exceed the length of the smaller array.
- **Provide comments with your assembly code. You will receive at least 10 pts for comments.**
- **Your code must be efficient.**
- The **sum_of_difference** function must save and restore the base pointer.
- Parameters must be passed on the stack.
- You may not modify the provided function and implement the modified version.
- **You need to define and use the local variable (difference).** That is, at the beginning of **sum_of_difference** you need to reserve space on the stack for this local variable.
- **Your solution must be recursive otherwise you will receive no credit (-110 pts).**

```
#include <stdio.h>

int sum_of_difference(int *a, int *b, int elements) {
   int difference;

   if (elements) {
      difference = *a - *b;
      return difference + sum_of_difference(++a, ++b, --elements);
   }
   return 0;
}

int main() {
   int a[] = {4, 11, 8, 12};
   int b[] = {1, 4, 2};

   printf("Difference: %d\n", sum_of_difference(a, b, 3));

   return 0;
}
```

Assembly Cheat Sheet

- Registers: %eax, %ecx, %edx, %ebx, %esi, %edi, %esp, %ebp
- Assembler Directives: .align, .long, .pos
- Data movement: irmovl, rrmovl, rmmovl, mrmovl
- Integer instructions: addl, subl, multl, divl, modl
- Branch instructions: jmp, jle, jl, je, jne, jge, jg
- Reading/Writing instructions: rdch, rdint, wrch, wrint
- Other: pushl, popl, call, ret, halt
- Ascii code for newline character: 0x0a
- Ascii code for space: 0x20

Answer:

```
sum_of_diff:    pushl %ebp
                rrmovl %esp, %ebp

                # Local variable space
                irmovl $4, %esi
                subl %esi, %esp

                # Retrieve parameters
                mrmovl 8(%ebp),  %ecx
                mrmovl 12(%ebp), %ebx
                mrmovl 16(%ebp), %edx

                # Checking elements equals 0
                irmovl 0, %esi
                subl %edx, %esi
                je base_case

                # Computing and storing difference
                mrmovl (%ecx), %edi     # *a
                mrmovl (%ebx), %esi     # *b
                subl %esi, %edi         # *a - *b
                rmmovl %edi, -4(%ebp)   # Storing *a - *b in difference loc var
                # ++a,++b,--elements
                irmovl 4, %esi          # Setting increment for ++a and ++b
                addl %esi, %ecx         # ++a
                addl %esi, %ebx         # ++b
                irmovl 1, %esi          # Decreasing number of elements
                subl %esi, %edx         # --elements

                # Pushing parameters for rec call
                pushl %edx
                pushl %ebx
                pushl %ecx

                # Recursive call
                call sum_of_diff

                # Popping parameters
                popl %ecx
                popl %ebx
                popl %edx

                # Computing (difference + result rec call) and leaving result in %eax
                mrmovl -4(%ebp), %edi
                addl %edi, %eax

                jmp func_end

                # Setting %eax to 0 for base case
base_case:      irmovl 0, %eax

                # Resetting stack ptr, %eb and returning
func_end:       rrmovl %ebp, %esp       # Reset stack ptr
                popl %ebp               # Restore callers frame ptr
                ret                     # Return
```