University of Maryland College Park
Dept of Computer Science
CMSC216 Spring 2015
Midterm II

Last Name (PRINT): ____

First Name (PRINT): ____

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

Lab TA (Circle One):

8am 0101 Jonathan Gluck	11am 0104 Baris Gokpinar	3pm 0203 Lee Williams	2pm 0302 Charles Parker
9am 0102 Jonathan Gluck	11am 0201 Ben Gmurczyk	4pm 0204 Charles Parker	3pm 0303 Andrew Liu
10am 0103 Ben Gmurczyk	12pm 0202 Halley Weitzman	1pm 0301 Halley Weitzman	4pm 0304 Eric Jeney

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

Your signature: ____

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 75 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 (C/Unix Concepts)	(50)	
#2	Problem #2 (Assembly Concepts)	(22)	
#3	Problem #3 (Linked Lists/**Coding**)	(80)	
#4	Problem #4 (Assembly/**Coding**)	(48)	
Total	Total	(200)	

Problem #1, C/Unix Concepts

1. (4 pts) Write a Unix command that will create a tar file named **myfile.tar** that includes the files **data1.txt** and **results.txt.** Both files are in the current directory.

2. (4 pts) How many dangling pointer variables do we have when execution has reached the point identified by /* HERE */ (just before the return 0).

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   char *p, *m, *k;

   p = calloc(2, sizeof(int));
   free(p);
   m = calloc(2, sizeof(float));
   p = m;
   k = m;
   free(k);
   /* HERE */

   return 0;
}
```

   Number of dangling pointer variables: _____

3. (4 pts) For the following code select all that apply.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   char *p = NULL;

   free(p);
   printf("Done\n");

   return 0;
}
```

   a. The code does not compile.
   b. The code compiles, but generates a segmentation fault.
   c. The code compiles and always prints "Done".
   d. The code may generate "Done" or a segmentation fault.
   e. None of the above.

4. (4 pts) What will happen to the memory associated with p when a free(p) operation takes place? Select all that apply.

   a. Memory may change.
   b. Memory gets initialized to 0 so it can be recycled.
   c. Memory will never be modified by free.
   d. Memory will be moved to the bottom of the heap.

5. (4 pts) Which integer value we can provide that will take care of dynamic memory correctly?

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main() {
   double *p = calloc(10, sizeof(double));
   int value;

   p++;
   scanf("%d", &value);
   free(p + value);

   return 0;
}
```

a. Any positive integer value.
b. Only 0.
c. Only -1.
d. Only -10.
e. Only 10.
f. There is no correct value we can provide.

6. (4 pts) In class we saw the address space of a program has the following sections: stack, data, text, and heap. In which of those areas are functions stored?

7. (4 pts) Identify any bugs/problems associated with the following code. If there are no bugs write NONE and the expected output. If the code does not compile write DOES NOT COMPILE and why.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int x, *d = &x, value;

   scanf("%d", &value);
   if (value >= 0) {
      d = malloc(sizeof(int));
      *d = 1;
   } else {
      *d = 0;
   }
   printf("%d\n", *d);

   free(d);

   return 0;
}
```

8. (4 pts) Identify any bugs/problems associated with the following code. If there are no bugs write NONE and the expected output. If the code does not compile write DOES NOT COMPILE and why.

```
#include <stdio.h>
#include <string.h>

int main() {
   int a[4] = {10, 20};

   memcpy(a, a + 1, sizeof(int));
   printf("%d\n", a[0]);
   return 0;
}
```

9. (4 pts) Identify any bugs/problems associated with the following code. If there are no bugs write NONE and the expected output. If the code does not compile write DOES NOT COMPILE and why.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct machine {
   double capacity;
   Machine *backup;
} Machine;

int main() {
   Machine *m = malloc(sizeof(Machine *));

   free(m);
   return 0;
}
```

10. (4 pts) The file radio.h defines the structure you will find below. Modify the radio.h file (feel free to edit/cross out the code provided), so stereo_system.c compiles. You may not change stereo_system.c.

```
/* radio.h */


typedef struct {
   double max_freq;
   int serial_no;
} Radio;
```

```
/* stereo_system.c */


#include <stdio.h>
#include "radio.h"
#include "radio.h"

int main() {
   Radio r = {102.34, 9};

   return 0;
}
```

11. (10 pts) The file **tv.c** includes the files **r.h** and **s.h**. For this problem:

   a. Define a makefile rule called **tv** that creates the executable **tv**. Use the CC macro to represent gcc and assume we have initialized the macro for you. You don't need to specify any gcc flags.

   b. Define a makefile rule called **clean** that will remove the **tv** executable.

Problem #2, Assembly Concepts

For the following assembly questions assume the stack frame organization discussed in class (unless otherwise indicated) and that the stack pointer has been initialized correctly (%esp initialized to 0x1000).

1. (4 pts) Are there any problems with the following instructions? If there are no problems write NONE, otherwise briefly describe the problem.

```
irmovl $20, %ebx
rmmovl %ebx, (%esp)
halt
```

2. (4 pts) Complete the following statement so we can copy the value of **%eax** to the second local variable (assuming we have two or more local variables).

```
rmmovl %eax,
```

3. (4 pts) Which of the following instructions changes condition codes (CC values)? Select all that apply.

   a. irmovl
   b. addl
   c. mrmovl
   d. subl
   e. rrmovl

4. (6 pts) Instead of using %eax to return a value we want to use the stack. Before a function is called, parameters are pushed (as usual), the stack pointer is decreased by 4 bytes (space for the return value) and the function is called.

   a. Write a **single** assembly instruction that will be executed by the function before it finishes that will place the return value in the area we have reserved for the return value in the stack. You can assume the value to return is in the %ecx register.

   b. Write a **single** assembly instruction that will allow the function to retrieve the first parameter.

5. (4 pts) Suppose there is an assembly function named **process** that takes a function as a parameter. Write at most three assembly instructions that will call the function **process** with the function **to_pass** as a parameter.

Problem #3, Linked Lists

```
typedef struct task {
   const char *name;
   const char *description;
   struct task *next_task;
} Task;

typedef struct {
   Task **priority_array;
   int max_priority;
} Calendar;
```

Information about the data structure:

- A Calendar structure keeps an array of linked lists (list of tasks).
- The size of the priority_array corresponds to the max_priority value. For example, for a max_priority value of 2, the array will have a size of 3 (0 is a valid priority value).
- Each list of tasks is kept sorted by name (in increasing order).

For this problem you can assume memory allocations will not fail (no need to check). You may not add an auxiliary function unless indicated otherwise.

1. Implement the **init_calendar** function. The function will dynamically-allocate memory for a Calendar structure, a priority_array (initializing each entry of the array to NULL), and initialize the max_priority field. The function returns a Calendar structure pointer.

   `Calendar *init_calendar(int max_priority);`

2. Implement an **insert_sorted** function. This function is used by the **add_task** function below. The insert_sorted function will insert a task (with the specified name and description) keeping the list sorted by name. Notice the list keeps track of the name and description, but it does not make copies of the strings. You **may not** modify the add_task function.

```
void add_task(Calendar *cal_ptr, int task_priority,
             const char *name, const char *description) {
   cal_ptr->priority_array[task_priority] =
         insert_sorted(cal_ptr->priority_array[task_priority], name, description);
}
```

   `Task *insert_sorted(Task *head, const char *name, const char *description);`

3. Implement the **destroy_calendar** function. It deallocates all the dynamically-allocated memory associated with the calendar. It addition, it sets the original Calendar pointer variable to NULL. For example, destroy_calendar(&cal_ptr) will leave cal_ptr set to NULL. You may add an auxiliary function if you understand it helps you.

   `void destroy_calendar(Calendar **cal_ptr);`

PAGE FOR YOUR CODE

PAGE FOR YOUR CODE

PAGE FOR YOUR CODE

Problem #4, Assembly

Write assembly code that corresponds to the **sum_of_products** function below. The function computes the sum of the product of elements of an array and a factor value. For example, the main() function below displays **30** ((1 * 2) + (3 * 2) + (5 * 2) + (6 * 2)).

- You only need to provide assembly code for the **sum_of_products** 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 array.
- Your assembly code must work for any array (not just the one in the example).
- **Provide comments with your assembly code. You will receive at least 5 pts for comments.**
- **Your code must be efficient.**
- The **sum_of_products** 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 (product).** That is, at the beginning of **sum_of_products** you need to reserve space on the stack for this local variable.
- **Your solution must be recursive otherwise you will receive no credit.**
- Use %ebx to represent the array **a** parameter, %ecx the **factor** parameter, and %edx the **elements** parameter. This will help during grading.

```
#include <stdio.h>

int sum_of_products(int *a, int factor, int elements) {
   /* You need to allocate and use stack space for this local */
   int product;

   if (elements) {
      product = *a * factor;
      return product + sum_of_products(++a, factor, --elements);
   }

   return 0;
}

int main() {
   int a[] = { 1, 3, 5, 6 }, factor = 2, elements = 4;

   printf("Sum: %d\n", sum_of_products(a, factor, elements));

   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

PAGE FOR YOUR ASSEMBLY CODE

```
.align 4
a: .long 1
   .long 3
   .long 5
   .long 6
```