Spring Semester 2015

KAIST EE209

Programming Structures for Electrical Engineering

Final Exam

Name: ______________________________

Student ID: ______________________________

This exam is closed book and notes. Read the questions carefully and focus your answers on what has been asked. You are allowed to ask the instructor/TAs for help only in understanding the questions, in case you find then not completely clear. Be concise and precise in your answers and state clearly any assumption you may have made. You have 135 minutes to complete your exam. Be wise in managing your time. Good luck.

Question 1	/ 15
Question 2	/ 20
Question 3	/ 15
Question 4	/ 25
Question 5	/ 25
Total	/ 100

Name: Student ID:

1. Basic Concepts (15 points)

A. Describe external and internal testing briefly. (5 points)

B. List four external testing and explain each testing briefly. (5 points)

C. List four exceptions and explain each exception briefly. (5 points)

Name: Student ID:

2. Assembly Language: Return and Argument Passing of Structures (20 points)

In the class, we discussed how we return the primitive type values or pass them as arguments. This problem is about how to do it for "structures", which you should find the answers by conjecturing the assembly codes of the given C code.

The following C code has a function `word_sum` having ***structures*** as argument and return values, and a function `prod` that calls `word_sum`:

```
typedef struct {
	int a;
	int *p;
} str1;

typedef struct {
	int sum;
	int diff;
} str2;

str2 word_sum(str1 s1) {
	str2 result;
	result.sum = s1.a + *s1.p;
	result.diff = s1.a - *s1.p;
	return result;
}

int prod(int x, int y) {
	str1 s1;
	str2 s2;
	s1.a = x;
	s1.p = &y;
	s2 = word_sum(s1);
	return s2.sum * s2.diff;
}
```

GCC generates the following code for these two functions:

```
1   word_sum:
2     pushl %ebp
3     movl  %esp, %ebp
4     pushl %ebx
5     movl  8(%ebp), %eax
6     movl  12(%ebp), %ebx
7     movl  16(%ebp), %edx
8     movl  (%edx), %edx
9     movl  %ebx, %ecx
10    subl  %edx, %ecx
11    movl  %ecx, 4(%eax)
12    addl  %ebx, %edx
13    movl  %edx, (%eax)
14    popl  %ebx
15    popl  %ebp
16    ret   $4
```

```
1   prod:
2     pushl %ebp
3     movl  %esp, %ebp
4     subl  $20, %esp
5     leal  12(%ebp), %edx
6     leal  -8(%ebp), %ecx
7     movl  8(%ebp), %eax
8     movl  %eax, 4(%esp)
9     movl  %edx, 8(%esp)
10    movl  %ecx, (%esp)
11    call  word_sum
12    subl  $4, %esp
13    movl  -4(%ebp), %eax
14    imull -8(%ebp), %eax
15    leave
16    ret
```

The optional numeric parameter to `ret` specifies the number of stack bytes to be released after the return address is popped from the stack. Thus the instruction `ret $4` is like a normal return instruction, but it increments the stack pointer by 8 (4 for the return address plus 4 additional), rather than 4. The `lea` (load effective address) instruction calculates the address of the first operand and loads it into the second operand while `mov` instruction copies the first operand into the second operand. The `imul` instruction is signed multiply. It multiplies two operands and stores the result to the second operand. The `leave` instruction is high level procedure exit. It sets `esp` to `ebp`, then pops `ebp`.

A. We can see in lines 5-7 of the code for `word_sum` that it appears as if three values are being retrieved from the stack, even though the function has only a single argument. Compare the C and assembly code to infer how to use the memory space and describe what these three values are. (5 points)

Name: Student ID:

B. We can see in line 4 of the code for `prod` that 20 bytes are allocated in the stack frame. These get used as five fields of 4 bytes each. Compare the C and assembly code to infer how to use the memory space and describe how each of these fields gets used. (5 points)

C. How would you describe the general strategy for passing ***structures*** as arguments to a function? (5 points)

D. How would you describe the general strategy for handling a ***structure*** as a return value from a function? (5 points)

Name: Student ID:

3. Modularization (15 points)

The following C code is an implementation of a list data structure.

```
/* list.c */
struct Node {
   const char *item;
   struct Node *next;
};

struct List {
   struct Node *first;
};

struct List *List_new(void) { … }
void List_free(struct List *l) { … }
void List_insert(struct List *l, const char *item) { … }
char *List_remove(struct List *l, const char *item) { … }
int List_search(struct List *l, const char *item) { … }
int List_isEmpty(struct List *l) { … }
```

```
/* client.c */
#include "stack.c"
/* use the functions defined in stack.c */
```

A. Describe the problems of this implementation (list up the answers by an itemized list, e.g., Prob1: xxxxx, Prob2: yyyy, Prob3: zzzz) (5 points)

Name: Student ID:

B. Improve this code in terms of modularity by satisfying the philosophies of (a) separation of interface and implementation and (b) encapsulation. (10 points)

```
/* list.h */
```

Name: Student ID:

```
/* list.c */
```

4. Data Structure (25 points)

Following C code is an implementation of a hash table module and a test program. `HashTable_insert()` inserts `node` to `ht` using the hash function `ht->hashfunc` and the size of the hash table `ht->size`. The new node is inserted as the first node of the slot. The macro `offsetof(type, member)` returns the offset of the field `member` from the start of the structure `type`.

A. Fill the blanks in the following code. Each blank is a single line. (20 points)

```
/* hashtable.h */
#ifndef __HASHTABLE_H__
#define __HASHTABLE_H__

#include <stddef.h>

struct Node {
  struct Node *prev;
  struct Node *next;
};

struct HashTable {
  struct Node **hashtable;
  int (*hashfunc)(void*);
  int size;
};

typedef struct Node *Node_T;
typedef struct HashTable *HashTable_T;

HashTable_T HashTable_new(size_t size, int (*hashfunc)(void*));
void HashTable_free(HashTable_T ht);
void HashTable_insert(HashTable_T ht, Node_T node);
void HashTable_remove(HashTable_T ht, Node_T node);
void HashTable_map(HashTable_T ht, void (*map)(void*));

#endif
```

```
/* hashtable.c */
#include <stdio.h>
#include <stdlib.h>
#include "hashtable.h"

HashTable_T HashTable_new(size_t size, int (*hashfunc)(void*))
{
  HashTable_T ht = (HashTable_T)malloc(sizeof(struct HashTable));
  if (ht == NULL) return NULL;

  ht->hashtable = (struct Node**)calloc(size, sizeof(struct Node*));
  if (ht->hashtable == NULL) {
    free(ht);
    return NULL;
  }

  ht->hashfunc = hashfunc;
  ht->size = size;

  return ht;
}

void HashTable_free(HashTable_T ht)
{
  free(ht->hashtable);
  free(ht);
}

void HashTable_insert(HashTable_T ht, Node_T node)
{
  int hash = ___________________________________________________;

  if (ht->hashtable[hash] == NULL) {
    node->next = NULL;
  }
  else {

    ___________________________________________________;

    ___________________________________________________;
  }
  node->prev = NULL;

  ___________________________________________________;
}

void HashTable_remove(HashTable_T ht, Node_T node)
{
  int hash = ___________________________________________________;
  Node_T p;

  for (_____________________________________________________){
    if (p == node)
      break;
  }

  if (p == NULL)
    return;

  if (p->prev)
    p->prev->next = p->next;
  else
    ht->hashtable[hash] = p->next;
  if (p->next)
    p->next->prev = p->prev;
}

void HashTable_map(HashTable_T ht, void (*map)(void *in))
{
  Node_T p;
  int i;

  for (i = 0; i < ht->size; i++)

    for (__________________________________________________)
      map((void *)p);
}
```

```
/* testhashtable.c */
#include <stdio.h>
#include <stddef.h>
#include <assert.h>
#include "hashtable.h"

#define HT_SIZE 5
#define NUM_ELEM 10

struct myNode {
  struct Node node;
  int value;
};

int myHashFunc(void *in)
{
  Node_T node = (Node_T)in;
  struct myNode *p = (struct myNode*)((char*)node -
                                      offsetof(struct myNode, node));
  return p->value;
}

void myMap(void *in)
{
  Node_T node = (Node_T)in;
  struct myNode *p = (struct myNode*)((char*)node -
                                      offsetof(struct myNode, node));
  printf("%d\n", p->value);
}

int main(int argc, char *argv[])
{
  HashTable_T ht;
  struct myNode elem[NUM_ELEM];
  int i;
```

Name: Student ID:

```
  ht = HashTable_new(HT_SIZE, myHashFunc);
  assert(ht);

  for (i = 0; i < NUM_ELEM; i++) {
    elem[i].value = i;
    HashTable_insert(ht, &elem[i].node);
  }

  HashTable_map(ht, myMap);
  printf("\n");

  for (i = 0; i < NUM_ELEM; i = i + 2) {
    HashTable_remove(ht, &elem[i].node);
  }

  HashTable_map(ht, myMap);
  HashTable_free(ht);

  return 0;
}
```

B. What is the output printed out? (5 points)

Name: Student ID:

5. Dynamic Memory (25 points)

Following C code is an implementation of a dynamic memory manager module, named heap manager K&R, which is given in the assignment 6. In this implementation, the heap memory is divided into several units. A chunk is a set of the contiguous units in memory. Each chunk's header unit contains a length and, if the chunk is free, a pointer to the next chunk in the free list. The free list is a singly-linked list and chunks in the free list are in increasing order of memory address. When a user calls `HeapMgr_malloc()` to allocate memory, this module finds a large enough chunk in the free list and return it. If the user calls `HeapMgr_free()` to deallocate memory, this module traverses the free list to find the correct spot for the given chunk to insert it into the free list considering the memory address of the chunk. Fill the blanks in the following code. Each blank is a single line.

```
/* heapmgr.h */
#ifndef HEAPMGR_INCLUDED
#define HEAPMGR_INCLUDED

#include <stddef.h>

void *HeapMgr_malloc(size_t uiSize);
void HeapMgr_free(void *pv);

#endif
```

```
/* heapmgrkr.c */
#include "heapmgr.h"

struct header {        /* block header */
   struct header *ptr; /* next block if on free list */
   unsigned size;      /* size of this block */
};

typedef struct header Header;

static Header base;       /* empty list to get started */
static Header *freep = NULL;     /* start of free list */

static Header *morecore(unsigned);

/* malloc:  general-purpose storage allocator */
void *HeapMgr_malloc(unsigned nbytes)
{
    Header *p, *prevp;
    unsigned nunits;

    nunits =__________________________________________________;
    if ((prevp = freep) == NULL) { /* no free list yet */
        base.ptr = freep = prevp = &base;
```

Name: Student ID:

```
        base.size = 0;
    }
    for (p = prevp->ptr; ; prevp = p, p = p->ptr) {
        if (p->size >= nunits) {    /* big enough */
            if (p->size == nunits)     /* exactly */
                prevp->ptr = p->ptr;
            else {            /* allocate tail end */

                _____________________________;

                _____________________________;

                _____________________________;
            }
            freep = prevp;
            return (void*)(p+1);
        }
        if (p == freep)  /* wrapped around free list */
            if ((p = morecore(nunits)) == NULL)
                return NULL;   /* none left */
    }
}

#define NALLOC 1024

/* morecore:  ask system for more memory */
static Header *morecore(unsigned nu)
{
    char *cp, *sbrk(int);
    Header *up;

    if (nu < NALLOC)
        nu = NALLOC;
    cp = sbrk(nu * sizeof(Header));
    if (cp == (char *) -1)  /* no space at all */
        return NULL;
    up = (Header *) cp;
    up->size = nu;
    HeapMgr_free((void *)(up+1));
    return freep;
}

/* free:  put block ap in free list */
void HeapMgr_free(void *ap)
{
    Header *bp, *p;

    bp = (Header *)ap - 1;    /* point to block header */
    for (p = freep; !(bp > p && bp < p->ptr); p = p->ptr)
        if (p >= p->ptr && (bp > p || bp < p->ptr))
```

Name: Student ID:

```
            break;  /* freed block at start or end of arena */

    if (bp + bp->size == p->ptr) {    /* join to upper nbr */

        _________________________________;

        _________________________________;
    } else

        _________________________________;
    if (p + p->size == bp) {           /* join to lower nbr */

        _________________________________;

        _________________________________;
    } else

        _________________________________;
    freep = p;
}
```