CS 310H **Computer Organization and Programming** **Spring 2008**
Professor Fussell **Due: May. 1, 2008**

Assignment #9

Instructions: The assignment is due on the date shown above. Tips to remember: give the assignments to your TA in section, remember your name, section number, TA name, and assignment number (5 points). Also, make sure your assignment is neat, stapled, and is entirely **your own work**.

1. In the procedure calling mechanism presented in class, we assumed that every procedure has a return value of exactly one word (one memory location), and that parameters are passed by value (i.e. copied to the stack). However, one may often want to pass arguments by reference rather than by value so that the callee can modify the data passed to it by the caller. It may also be desirable to return more than a single word to the caller. Consider the C linked list management code below.

```
#include <stdio.h>
#define LISTSIZE 3
typedef struct Record record;
struct Record {
    int value;
    record *next;
};
void insert(record **head, record *new_node) {
    record *head_ptr;

    head_ptr = *head;
    if(head_ptr == NULL|| new_node->value <= head_ptr->value) {
        new_node->next = head_ptr;
        *head = new_node;
        return;
    }
    else {
        while(head_ptr->next != NULL) {
            if(new_node->value <= head_ptr->next->value) {
                new_node->next = head_ptr->next;
                head_ptr->next = new_node;
                return;
            }
            head_ptr = head_ptr->next;
        }
        new_node->next = NULL;
        head_ptr->next = new_node;
        return;
    }
}
```

```
record *delete(record **list, int value) {
    record *this_record, *next_record;

    this_record = *list;
    if(this_record == NULL || this_record->value == value) {
        *list = this_record->next;
        this_record->next = NULL;
        return this_record;
    }
    else {
        while(this_record->next != NULL) {
            if(this_record->next->value == value) {
                next_record = this_record->next;
                this_record->next = next_record->next;
                next_record->next = NULL;
                return next_record;
            }
            this_record = this_record->next;
        }
        return NULL;
    }
}
int main (int argc, const char * argv[]) {
    int value[LISTSIZE] = {5, 2, 10};
    int i;
    record *new_node = NULL, *list = NULL;
    for (i = 0; i < LISTSIZE; i++) {
        new_node = (record *)malloc(sizeof(record));
        new_node->value = value[i];
        insert(&list, new_node);
    }
    new_node = delete(&list, 5);
    free(new_node);
    return 0;
}
```

Assume that on the LC-3 the heap starts at x9000 and the stack starts at xd000. Draw the heap and the stack just before `delete` returns to the main program.

2. P&P 14.15

3. P&P 14.19

4. In this programming assignment, you will implement a simple memory manager with two functions:

   - `address new(int size)` - which is passed a parameter indicating how many LC-3 memory locations are desired. The function returns an address to the allocated memory.
   - `void free(address addr)` - which tells the memory manager that it can deallocate the memory pointed to by addr and reallocate to a later call to new.

   You must maintain a free list that is a linked list containing all of the free memory blocks, where a block is specified in C as:

```
struct block {
    int size;
    struct block *next;
}
```

   Use the template in hw9-1.asm as a starting point. This code includes a function `void init()` which initializes the heap and the free list to point to the entire heap.

   Make sure that newly-freed blocks are always merged with adjacent free blocks.

   Use the trick of allocating one more memory location than requested so you can store the size of the allocated object. This will help you deallocate the right amount of memory in free.

   Your free list should be embedded in the free blocks of memory, rather than using some auxiliary data structure.

   You need not maintain a list of allocated objects - you may assume that the calling program will keep track of these.

   You should use a first-fit strategy for finding a new block.

   You should also generate a test program that allocates and deallocates memory locations. See hw9-main.asm as an example. When we test your code, we will use our own version of hw9-main.asm, so please ensure that your functions follow the appropriate procedure calling convention.

   Because you are writing a program that spans multiple files, you need to fill in the function address table in hw9-main.asm for `NEW_ADDR` and `FREE_ADDR`. The current values will be incorrect and must be fixed.

   Please turn in the following:

   A hardcopy of your code (in hw9-1.asm) and test code (in hw9-main.asm). Electronic copies of both files using the linux turnin-script.

   A diagram of your heap after the following sequence of allocations and deallocations:

```
A = new(x8)
B = new(x16)
C = new(x3)
D = new(x40)
free(C)
free(A)
E = new(x2)
F = new(x7) free(D)
```

5. Suppose you have been asked to write a memory manager (I know, it's very unlikely) and you are choosing between a first-fit and a best-fit strategy. The API for your manager consists of the functions `void *malloc(int size)` and `void free(void *ptr)` as in C, where `malloc` returns a pointer to the allocated block of memory (which of course you must cast to the proper type in C), and `free` takes as input a pointer to the block of memory to be returned to the free list. You are maintaining the free list as a singly-linked list in sorted order by starting address of the blocks. A free list block is declared as follows:

```
struct block {
    int size;
    struct block *next;
}
```

   (a) How many words are in the smallest block possible on the LC-3?

   (b) Assuming applications will request blocks of storage with sizes that are highly variable, for which strategy would `malloc` be expected to run faster? Why?

   (c) Using the same assumptions, for which strategy would `free` be expected to run faster? Why?

   (d) If your application requests only blocks of the same size, say because it' s managing a linked list of records, all of which are the same type, which strategy will cause less memory fragmentation? Why?

   (e) Suppose your application requests blocks of two different sizes, with these requests being interleaved in some random order. For which strategy would we expect `malloc` to run faster? Why?

   (f) Suppose you have the same application allocating blocks of two sizes. Its behavior is that it first allocates enough blocks to nearly fill up the heap, and then after that it only requests a block of a given size after it has freed one or more such blocks. Is the best-fit strategy guaranteed to satisfy all the `malloc` requests? Why?

   (g) Is first-fit guaranteed to satisfy all the `malloc` requests under the same assumptions as in (f)? Why? (Hint: Consider cases in which after we've allocated almost all the heap, we then free a pair of adjacent blocks of one size, and then another pair of adjacent blocks of the other size. What can happen then?)