CSE 374 Final Exam 12/14/17 Sample Solution

Question 1. (13 points) Preprocessor. Consider the following program:

```
#include <stdio.h>
#define SQUARE(x) x * x
#define DOUBLE(x) 2 * x
int main() {
  int a = 2;
  int b = 3;
  int x = DOUBLE(a);
  int y = SQUARE(a+b);
  int z = DOUBLE(1+a+b);
  printf("x = %d, y = %d, z = %d\n", x, y, z);
  return 0;
}
```

(a) (10 points) Show the output produced by the preprocessor (`cpp -P`) when it reads and processes this C program, which is the first step that happens before the compiler can translate the code to generate a `.o` file. Ignore the `#include <stdio.h>` line – it will insert the declarations from `stdio.h` and do nothing further. Otherwise, your answer should show all of the output from the preprocessor. There are no preprocessor errors in this program, and the resulting program compiles and executes without errors.

```
int main() {
  int a = 2;
  int b = 3;
  int x = 2 * a;
  int y = a+b * a+b;
  int z = 2 * 1+a+b;
  printf("x = %d, y = %d, z = %d\n", x, y, z);
  return 0;
}
```

Notes: None of the preprocessor directives (`#define`, etc.) are copied to the output. The preprocessor only does textual substitution and does not simplify, rewrite, or add parentheses to expressions. When grading the question, we did not worry about the exact number of spaces included before or after the expansion of a `#define` macro.

(b) (3 points) Fill in the blanks to show the numbers printed when this program is executed:

x = ___**4**___, y = ___**11**___, z = ___**7**___

Question 2. (16 points) Making things. Suppose we have a project with multiple source files that have the `#include` dependencies shown below.

```
***************
* foo.h       *
***************
#ifndef _FOO_H_
#define _FOO_H_
...
#endif
```

```
***************
* foo.c       *
***************
#include "foo.h"
...
```

```
***************
* main.c      *
***************
#include "foo.h"
#include "bar.h"

int main() {
  ...
}
```

```
***************
* bar.h       *
***************
#ifndef _BAR_H_
#define _BAR_H_

#include "foo.h"
...
#endif
```

```
***************
* bar.c       *
***************
#include "bar.h"
...
```

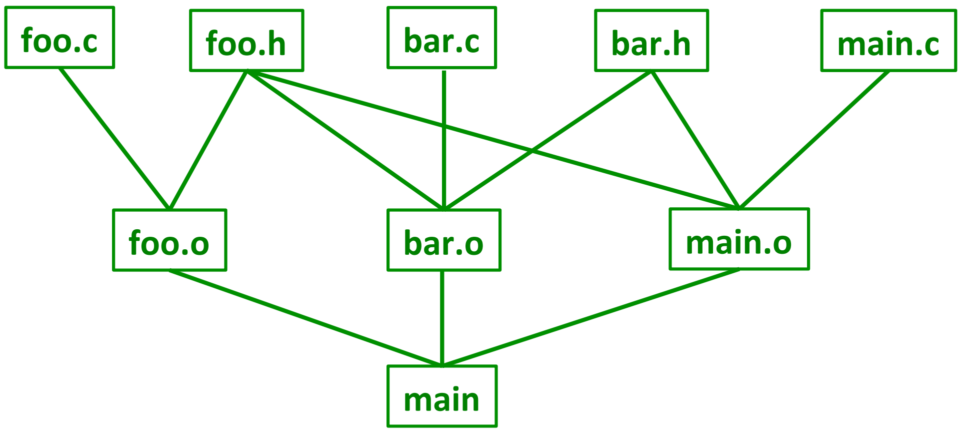
If we're being lazy, we could create an executable program from these files by executing the command `gcc -Wall -g -std=c11 -o main *.c`.

On the next page, create a `Makefile` that builds an executable program named `main` from these files as done by the `gcc` command above, but only recompiles and relinks the minimum number of files needed after any changes are made to the source files. Your answer should be done in two steps:

- First, construct the dependency graph that shows the dependencies between the source files, the compiled `.o` files, and the final executable file `main` that is created by linking the `.o` files.
- After drawing the dependency graph, write the final `Makefile`.

Write your answers on the next page. You should remove this page from the exam. **Do not write on this page.** It will not be scanned for grading.

Question 2. (cont.) (a) (8 points) Draw the dependency graph (diagram) showing dependencies between the files on the previous page, the `.o` files created by compiling the `.c` files, and the final executable program `main`.



Notes: `bar.h` `#includes` `foo.h`, so any `.o` file that depends on `bar.h` also depends on `foo.h`. Also, note that there is no direct dependency between `.h` files and the `.c` files that `#include` them. The connection is between each `.o` file and the `.h` and `.c` files that are read by the compiler to produce the .o file.

(b) (8 points) Give the contents of a `Makefile` that will build the program as described by the dependency diagram above. The default target that is built if we just type `make` with no arguments should be the executable program `main`.

```
#default target:
main: main.o foo.o bar.o
   gcc -Wall -g -std=c11 -o main main.o foo.o bar.o

# individual .c/.o files
main.o: main.c foo.h bar.h
   gcc -Wall -g -std=c11 -c main.c

foo.o: foo.c foo.h
   gcc -Wall -g -std=c11 -c foo.c

bar.o: bar.c foo.h bar.h
   gcc -Wall -g -std=c11 -c bar.c
```

There are, of course, other solutions. However, the first target must be the one that builds `main`, which needs to be the default target for this `Makefile`.

Question 3. (10 points) git. Two of our programmers, Bart and Lisa, are using git to manage a project. Both of them have been working on various parts of the code. Bart has made some changes, tested them, and now wants to commit and push his changes to the shared repository. A transcript of the terminal session follows. Answer questions about it below.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
        modified:   main.c
no changes added to commit (use "git add" and/or "git commit -a")
$ git add main.c
$ git commit -m "change output message"
[master 2413774] change output message
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push
To gitlab.cs.washington.edu:cse374-17au-students/cse374-17au-xa.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'git@gitlab.cs.washington.edu:cse374-17au-
students/cse374-17au-xa.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
$
```

(a) (2 points) Something went wrong when Bart tried to do the `git push`. What is the explanation of the problem? (Don't just repeat the message printed above – explain what we can actually conclude from the message about the underlying cause of the failure or what the likely conflict is, etc., but be brief.)

The only thing we can say for sure is that someone pushed changes to the shared `git` repository between the last time Bart did a `git pull` and the current attempt to do a `git push`. There's a fair chance it's because the other person is editing the same file, but that is not guaranteed – the conflict could be due to something else that was changed in the repository.

(continued on next page)

Question 3. (cont.) Not quite sure what to do next, Bart tries entering the `git pull` command suggested in the original error message. Here is the output from that:

```
$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From gitlab.cs.washington.edu:cse374-17au-students/cse374-17au-xa
   7809b75..226da1e  master     -> origin/master
Auto-merging main.c
CONFLICT (content): Merge conflict in main.c
Automatic merge failed; fix conflicts and then commit the result.
$
```

(b) (8 points) Well, that didn't quite work, apparently. Explain specifically what the problem is now and then give a complete list of the specific steps that Bart needs to do to clear up the problem(s) and successfully push changes to the git repository.

The problem is that the current repository version of `main.c` has changes that conflict with the current copy of the file in Bart's local repository.

To fix this, Bart needs to do the following steps:

1. **Edit `main.c` and make whatever changes are needed to resolve the conflict. (Note that `main.c` will contain markers inserted by `git` to indicate where the conflicts occur in the file.)**
2. **Run the following `git` commands in this order:**
   a. `git add main.c`
   b. `git commit -m "message describing changes"`
   c. `git push`

There are other `git` commands and options that combine `git add` and `git commit`. It's fine to use those to answer this question as long as they have the necessary effect.

Note that it is not reasonable for Bart to save a copy of his `main.c` file somewhere, delete his copy of the repository, re-clone it, and then replace `main.c` with his copy. That will either lose whatever changes were in the repository version of the file, or will replace `main.c` with exactly the same copy that was just downloaded and still contains the conflict(s) that caused the problem in the first place.

Question 4. (20 points) Another trieing question. In HW5 we used a trie to store the words in a dictionary based on the digit sequences that encoded the words. There are several possible ways to represent trie nodes, but for this problem, assume that a node is defined as follows:

```
struct tnode {              // one node in the trie:
  char * word;              //  C-string if this node has a
                            //  word attached, otherwise NULL
  struct tnode *next[10];   //  Subtrees.  next[2] to next[9]
                            //  are subtrees for digits 2-9;
                            //  next[0] is the synonym ('#') link.
};                          //  For 0<=i<10, next[i]==NULL if
                            //  next[i] is an empty subtree.
```

For this problem, write a function that produces an exact copy of a trie. In other words, `clone(r)` should return a pointer to a new trie that is an exact copy of the original one, including copies of all of the nodes and strings in the original.

You should assume that all necessary header files have already been `#include`d and you do not need to add any `#includes`. You may assume that `malloc` always succeeds and you do not need to check the result for `NULL`.

Hints: recursion really, *really*, **really** is your friend. The clone of an empty trie is empty.

A bit of (maybe) useful reference information about strings and memory:

Some string library functions:

- `char* strncpy`(*dest*, *src*, *n*), copies exactly *n* characters from *src* to *dst*, adding '\0's at end if fewer than *n* characters in *src* so that *n* chars. are copied.
- `char* strcpy`(*dest*, *src*)
- `char* strncat`(*dest*, *src, n*), append up to *n* characters from *src* to the end of *dest*, put '\0' at end, either copy from *src* or added if no '\0' in copied part of *src*.
- `char* strcat`(*dest*, *src*)
- `int   strncmp`(*string1*, *string2*, *n*), <0, =0, >0 if compare <, =, >
- `int   strcmp`(*string1*, *string2*)
- `char* strstr`(*string, search_string*)
- `int   strnlen`(*s, max_length*)
- `int   strlen`(*s*)

Basic C memory management functions:

- void * **malloc**(size_t size)
- void **free**(void *ptr)
- void * **calloc**(size_t number, size_t size)
- void * **realloc**(void *ptr, size_t size)

Write your answer on the next page. You should remove this page from the exam. **Do not write on this page.** It will not be scanned for grading.

Question 4. Write your implementation of the trie `clone` function below. Additional space is provided on the next page if you need it.

```
// Return a pointer to the root of a new trie that is an
// exact copy of the trie with root r.  If r is NULL
// (i.e., empty), return NULL.
struct tnode *clone(struct tnode *r) {
  // Clone of an empty tree is an empty tree
  if (r == NULL)
    return NULL;

  // Create a new root node for the cloned tree
  struct tnode *new_root =
          (struct tnode *)malloc(sizeof(struct tnode));

  // If node r contains a string, make a copy of
  // the string and include it in the new root node.
  // Otherwise set word to NULL in new root node.
  if (r->word != NULL) {
    new_root->word = (char *)malloc(strlen(r->word)+1);
    strcpy(new_root->word, r->word);
  } else {
    new_root->word = NULL;
  }

  // clone subtrees (clone of a NULL pointer is NULL)
  for (int i = 0; i < 10; i++)
    new_root->next[i] = clone(r->next[i]);

  return new_root;
}
```

Notes: We have to make a copy of the string if it is present so the two trees don't wind up sharing pointers to a single copy of the string on the heap. In production code it is probably safer to use functions like `strncpy` instead of `strcpy`, but either was fine for this problem. Also, it is necessary to explicitly store `NULL` in `new_root->word` if node `r` does not contain a string. We cannot assume that memory allocated by `malloc` will be initialized to `0`, `NULL`, or any other specific value. If `calloc` were used instead of `malloc`, the explicit `NULL` initialization would not be needed since storage allocated by `calloc` is initialized to binary zeros.

Question 5. (20 points) A bit of memory management. Recall from HW6 that we can represent the free list for the getmem/freemem storage allocator as a linked list of blocks. The beginning of each block is described by the following C struct:

```
struct free_node {          // node on free list:
  uintptr_t size;           //   number of bytes in this
                            //     block, not including the
                            //     size of this header
  struct free_node *next;   //   next block on free list or
                            //     NULL if none
};
```

(We will assume that the `size` in the `free_node` struct includes only the number of bytes of data following the header and does not include the header, although this won't really matter for this problem.)

One of the operations needed in the memory manager is to insert a new block in the free list in the correct position so that the blocks on the free list have ascending memory addresses. On the next page, implement the function `insert_free_node` that does this. You should assume that there is a global pointer variable that stores the address of the first block on the free list, and your code can access that variable without further declaration:

```
struct free_node *free_list;  // free list blocks; NULL
                              //   if free_list is empty
```

For full credit, your solution should not need to traverse the free list more than once (i.e., it should work in a single pass, not multiple passes over the list).

Simplifying assumptions: your code only needs to insert the block in the list. It does not need to detect adjacent blocks or combine blocks.

You can assume that the new block being inserted is not currently on the free list and does not overlap any blocks already on the free list.

You should assume that all necessary header files have already been `#include`d and you do not need to add any `#includes`.

Write your answer on the next page. You should remove this page from the exam. **Do not write on this page.** It will not be scanned for grading.

(continued on next page)

Question 5. (cont.) Write your implementation of `insert_free_node` here. The heading of the function is written for you.

```
// Insert node b in correct place on the global free_list.
// Nodes are stored in order based on node addresses.
void insert_free_node(struct free_node *b) {

  // If node b belongs at the beginning of the list,
  // insert it there and return.
  if (free_list == NULL || b < free_list) {
    b->next = free_list;
    free_list = b;
    return;
  }

  // Free list is not empty and contains at least one node
  // prior to b. Advance p to the last node on the free
  // list whose address is less than b.
  struct free_node *p = free_list;
  while (p->next != NULL && p->next < b) {
    p = p->next;
  }

  // Either p->next is NULL or p->next >= b. In either
  // case, insert b between p and p->next (or at the end
  // of the list if p->next is NULL).
  b->next = p->next;
  p->next = b;
}
```

Note: The conditions in the `if` and `while` statements both depend on short-circuit evaluation of the `||` and `&&` operators so that the right operand is not evaluated if the left operand (the `NULL` check) is sufficient to decide whether the whole expression is true or false.

Question 6. (12 points) Testing. Describe three different *black-box* tests for the `insert_free_node` function from the previous problem. For full credit, the three tests must verify different things about the implementation, and must describe the specific input or setup for the test and the expected results. In your description of test inputs and outputs you can specify that a particular block is located at a particular address or that the free list contains blocks located at specific addresses, without having to explain how those blocks would be created at those specific addresses. Also, to simplify the question, you do not need to worry about whether block addresses are multiples of 16.

There are many, many possible tests. Here are a few:

Setup: Empty free list (`free_list == NULL`). Insert a block at address 100 in the list (insert into empty free list).

Expected result: free list contains a single block at address 100.

Setup: Free list contains blocks at addresses 200, 400, 600. Insert a block at address 100 in the free list (insert at beginning).

Expected result: Free list contains blocks at addresses 100, 200, 400, 600.

Setup: Free list contains blocks at addresses 200, 400, 600. Insert a block at address 500 in the free list (insert in middle).

Expected result: Free list contains blocks at addresses 200, 400, 500, 600.

Setup: Free list contains blocks at addresses 200, 400, 600. Insert a block at address 800 in the free list (insert at end).

Expected result: Free list contains blocks at addresses 200, 400, 600, 800.

Notes: Black-box tests are those that test software only against its specification, without any knowledge of or reference to implementation details. A good test needs a precise description of the input(s) and expected output(s). Without both of these it is not possible to determine whether the test succeeded.

Question 7. (14 points) C++. To explore a bit more C++, we've designed a simple class to represent complex numbers. A complex number is represented by a pair of doubles, holding the real and imaginary parts of the number respectively. (Don't worry if you aren't familiar with complex numbers – this is a programming problem and all the details needed are described below.)

The class declaration in file `Complex.h` looks like this:

```
#ifndef _COMPLEX_H_
#define _COMPLEX_H_

class Complex {
public:
  // Construct Complex x+y*i
  Complex(double x, double y);

  // accessors - return parts of this Complex number
  double real() const;
  double imag() const;

  // addition: return the value this + other
  Complex plus(Complex other) const;

private:
  // Representation of a Complex number: re+im*i
  double re;  // real part
  double im;  // imaginary part
};

#endif   // _COMPLEX_H_
```

On the next page, give implementations for the four functions (constructor, `real` and `imag` accessor functions, and `plus`) that are declared but not implemented above.

Reminder: The sum of two complex numbers (a+bi)+(c+di) is the complex number (a+c)+(b+d)i – in other words, simply add the real and imaginary parts together to get the real and imaginary parts of the result.

You should remove this page from the exam. **Do not write on this page.** It will not be scanned for grading.

Question 7. (cont.) Provide your implementation of the Complex class declared on the previous page as it would be written in the implementation file Complex.cc.

```
#include "Complex.h"

// write your implementation of class Complex below.

// Construct Complex x+yi
Complex::Complex(double x, double y) {
  re = x;
  im = y;
}

// accessors
double Complex::real() const { return re; }
double Complex::imag() const { return im; }

// addition: return the value this + other
Complex Complex::plus(Complex other) const {
  return Complex(re+other.re, im+other.im);
}
```

Question 8. (9 points) Concurrency. Consider the following code, which maintains a list of integer values.

```
struct int_node {         // nodes for an integer list:
  int val;                //   value in this node
  struct int_node *next;  //   next node or NULL if none
};
struct int_node *list = NULL;  // list of nodes; NULL if empty
// Insert new_node on the front of the list
void add_node(struct int_node *new_node) {
 atomic {
  new_node->next = list;
  list = new_node;
 }
}
// remove first list node and return it; NULL if list empty
struct int_node *delete_node() {
 atomic {
  struct int_node *temp = list;
  if (list == NULL) return NULL;
  list = list->next;
 }  // the '}' could also appear after return
  return temp;
}
```

Suppose we use this code in a program with two threads t1 and t2, both of which add and delete nodes from the (shared) `list` by calling `add_node` and `delete_node`.

(a) (5 points) This code is not thread-safe. Give a brief explanation of what can go wrong if the two threads are using these functions concurrently. (If it matters, assume that no other code modifies the list nodes or otherwise changes the data contained in the list.)

There are many reasonable answers, but the key point is that if two threads are manipulating the shared variable `list` without locking, the different threads can overwrite and lose updates to `list`, or see inconsistent values, or update `list` and accidentally remove a block that was inserted by the other thread while the first thread was in the process of removing a different block, etc.

(b) (4 points) Good news! It turns out that the C standards committee has added an atomic statement to C, as described in lecture. A sequence of statements s1; …; sn; will be executed atomically if we surround them with atomic as follows: atomic{s1; …; sn}.

Add atomic{ … } statements to the above code so the resulting code is thread-safe.

See boldface additions in the code above.

Question 9. (6 points) Concurrency and locks. We have a program with three concurrent threads and three shared objects X, Y, and Z. Here are two possible locking strategies for handling access to the shared objects:

Strategy 1

Thread 1
```
Acquire lock for X
Acquire lock for Y
Modify X and Y
Release lock on Y
Release lock on X
```

Thread 2
```
Acquire lock for Z
Acquire lock for X
Modify Z and X
Release lock on X
Release lock on Z
```

Thread 3
```
Acquire lock for Y
Acquire lock for Z
Modify Y and Z
Release lock on Z
Release lock on Y
```

Strategy 2

Thread 1
```
Acquire lock for X
Acquire lock for Y
Modify X and Y
Release lock on Y
Release lock on X
```

Thread 2
```
Acquire lock for X
Acquire lock for Z
Modify Z and X
Release lock on Z
Release lock on X
```

Thread 3
```
Acquire lock for Y
Acquire lock for Z
Modify Y and Z
Release lock on Z
Release lock on Y
```

Which one of these strategies can cause a deadlock? Show a sequence of operations that lead to the deadlock, and explain why the other strategy will never cause a deadlock.

Strategy 1 can lead to a deadlock. If each thread acquires its first lock and is then interrupted before it can acquire its second one, then each thread will hold one lock for either X, Y, or Z, and will wait attempting to acquire the lock for its other resource, which is already locked by a different thread.

Strategy 2 will never deadlock because all threads acquire the locks they need in the same order (X then Y then Z). At any time at least one of the threads will be able to acquire its remaining lock and run to completion. Once that thread finishes, it will release its locks and, if any other threads are waiting to acquire a lock, at least one of those threads will be able to acquire the lock(s) it needs to proceed.

Have a great winter break! The CSE 374 Staff