HW2 Solutions

1.1.

```
node_t *
find_insert_pos(node_t *head, node_t *node)
{
    if (head == NULL) return NULL;

    node_t *ret = NULL;

    // 2.1 your code here
    while (head != NULL) {
        if (head->id >= node->id) break;
        else {
            ret = head;
            head = head->next;
        }
    }

    return ret;
}
```

1.2.

```
node_t *
insert(node_t *head, node_t *node)
{
    if (head == NULL) return node;

    // find the proper position to insert
    // this node pair.
    node_t *pos = find_insert_pos(head, node);

    // 2.2 your code here
    if (pos == NULL) { // insert before head
        node->next = head;
        head = node;
    } else { // insert after pos
        node->next = pos->next;
        pos->next = node;
    }

    return head;
}
```

2.1.

i. `echo echo hello $world` Output: `echo hello` Explanation: The command is the first `echo`, and `$world` is a variable whose name is 'world'. Since we have not set the value for `$world` yet, `$world` will be expanded to an empty string.

ii. `echo 'echo hello $world'` Output: `echo hello $world` Explanation: Within single quotes, `$world` is not treated as a variable but rather as part of the string.

iii. `echo "echo hello $world"` Output: `echo hello` Explanation: The command is the first `echo`. Double quotes will expand the variables inside, and we have not set `$world` yet

iv. `echo \`echo hello $world\`` Output: `hello` Explanation: The symbol ` will evaluate the command inside, and the output of `echo hello $world` is "hello". Then the command becomes: `echo hello`.

v. `echo (echo hello $world)` Output: `syntax error` Explanation: Cannot use subshell as part of another command

2.2.

i. `echo 'hello world' | cat` Output: `hello world` Explanation: First `echo` prints out "hello world", and it's passed to `cat` as the input. `cat` will print its input to screen

ii. `echo 'hello world' > cat` Output: No output; create a file called 'cat', and the content is "hello world" Explanation: `echo` prints "hello world", and got redirected into a file called cat

iii. `echo 'hello world' 2> cat` Output: `hello world` (printed on the screen); and create an empty file called cat Explanation: `echo` prints "hello world" to the standard output, which is the screen by default. Standard error was redirected to a file called cat. Since nothing was printed to stderr, the file is empty

2.3.

i. `echo a && echo b` Output:

```
a
b
```

Explanation: `&&` means first run the command prior to it, and if the exit status is 0, run the command after it; if the exit status is not 0, stop.

ii. `echo a ; echo b` Output:

```
a
b
```

Explanation: `;` means run the command prior to it, and no matter what is the exit status, always run the command after it.

iii. `echo a & echo b` Output:

```
a
b
```

or

```
b
a
```

Explanation: `&` will put `echo a` in background and run `echo b` in foreground. So the output may come out in any order

2.4.

To output the first 100 names in this file:

```
$ cat members.txt | grep "^Name:[a-zA-Z']\+$" | head -n100 | cut -d':' -f2
```

To identify the first 100 names by alphabetical order, and then output them and write them to a file called 'names.txt'

```
$ cat members.txt | grep "^Name:[a-zA-Z']\+$" | head -n100 | cut -d':' -f2 | sort | tee names.txt
```

3.1.

The differences between `thread_create(func)` and `fork()` include:

- `thread_create` doesn't need to make a copy of the invoking thread's memory. It will make both the original thread and created thread share the invoking thread's memory. By contrast, `fork` will make a copy of the parent process's memory, and the child will run using this copy.
- Both `thread_create` and `fork` will give the created thread/process a new stack. `fork` creates this stack from the parent process's stack, while `thread_create` creates a brand new stack. In other words, `fork` will set `%esp` to a copy of the current bottom of the stack, `thread_create` will set `%esp` to the top of a new stack within the same address space.
- As indicated in the interface, `thread_create` takes a function pointer as the parameter. thread_create will set the created thread's `%eip` to `func`, while `fork` will not touch `%eip`, since parent and child are running the "same" code located at the "same" address.

3.2.

There aren't many essential differences between these data structures. Each "process" has its own view of memory (we will later study how this works), and multiple "threads" share a view of memory. But the actual OS data structures used for each can be essentially the same.

4.1.

```
foo: 1                  foo: 0
boo: 0          or      boo: 1
main: 2                 main: 2

foo: 0                  foo: 0
boo: 0          or      boo: 0
main: 1                 main: 2
```

Note that the first two lines may come out in any order (`boo` before `foo`)

Note that even if we don't have sequential consistency, we can't get more outcomes than this. The reason is that we assume `join_thread()` (which is a synchronization call) is implemented correctly. As mentioned in the lecture notes, a correct implementation of a synchronization call includes an internal memory barrier. Concretely, in order for the main thread to "see" the threads as complete, the main thread also has to see all of the updates to memory by those threads.

4.2.

Use `mutex` to protect the shared state (integer `i` in this case), in both `foo` and `boo`

4.3.

If we modify our `foo` and `boo` to be:

```
mutex_t m;
void
foo(void *)
{
    lock(&m);
    int n = i;
    i = i + 1;
    printf("foo: %d\n", n);
    unlock(&m);
}

void
boo(void *)
{
```

```
    lock(&m);
    int n = i;
    i = i + 1;
    printf("boo: %d\n", n);
    unlock(&m);
}
```

Then the output should be:

```
foo: 0              boo: 0
boo: 1      or      foo: 1
main: 2             main: 2
```

Note: you can also put `unlock` before `printf`, then the possible output will be the above two, and also the first two lines come in a different order:

```
boo: 1              foo: 1
foo: 0      or      boo: 0
main: 2             main: 2
```