UCLA Computer Science 97 Practice Final – Spring 2020

120 points total. Open notes, open computer. Forewarning: **this practice final is very hard** (like the real one will probably be)!! You are **far** from expected to get perfect on it. Don't stress if you don't know everything – use it as a guide to see what topics you need to spend more time on. Answer what you know first before questions you aren't as sure about.

1. (8 mins.) Consider the following pseudocode for an event-based shell program written in JavaScript:

```
class Event { handle() { ... } }
class KeyboardEvent extends Event { handle() { ... } }
class OutputEvent extends Event { handle() { ... } }

function wait_for_event() {
  while (true) {
    if (keyboard_input_waiting) {
      // A keyboard button was pressed by the user
      keyboard_input_waiting = false;
      return KeyboardEvent();
    } else if (output_waiting) {
      // We ran a shell command and have output that needs
      // to be displayed back to the user
      output_waiting = false;
      return OutputEvent();
    }
  }
}

// Main
while (true) {
  const e = wait_for_event();
  e.handle();
}
```

a. (1 min.) What code corresponds to the event loop?

```
while (true) {
  const e = wait_for_event();
  e.handle();
}
```

b. (1 min.) What code corresponds to the event handler(s)?

```
e.handle();
```

c. (6 min.) Discuss the performance of this particular implementation. Is it a good implementation of event-based programming? Why or why not?

This is not a good implementation. One of the key benefits of event-based programming is that the CPU is able to do other work while the process is waiting for events to happen. However, in this implementation, that does not happen. In `wait_for_event`, we simply have a while loop that spins until an event is ready. The process does not use any special low-level function to release control of the CPU. While the OS will probably interrupt the process itself, it could get much more work done if our program explicitly slept while waiting.

2. (12 min.) In lecture, we talked at a very high level about threads. A thread is a single sequential flow of instructions within a program. Their main purpose is to allow for concurrency – executing multiple instructions at the same time – in a multithreaded process. Threads share memory, and they can even be running on different CPUs (again, all at the same time).

   a. (2 min.) List **one** main difference between a process and a thread.

   A few differences include:
   — Threads share memory, processes do not (by default).
   — A process can execute on multiple threads, but not vice versa.

b. (10 mins.) The CPython Global Interpreter Lock (GIL) is a mechanism that allows only one thread in a Python program to have control of the interpreter at a time. This means that for any Python program, only **one** of its threads can be in execution at a time. Why, then, would you ever want to multithread a Python process (assuming you're using the CPython interpreter)?

   Just because there can only be one Python **process running doesn't mean that we can't invoke other processes. For example, suppose we need to perform some I/O work like writing to a file. We can create another thread to handle the I/O work (which will involve system calls and the like, i.e. the OS will be doing work) while our Python process does other things. In addition, there may be some time where we want it to *seem* like two Python functions are running concurrently. For example, if we have two functions A and B, we can use threads to have the CPU do a little bit of A, then a little bit of B, then back to a little bit of A, back to B, etc. Although it is only running one function at a time, it will seem like they are being done in parallel.**

3. (6 min.) Interprocess communication (IPC) refers to the passing of data between different processes. Give **two** examples of IPC that you have used in class, and briefly explain how they work.

   One example is piping output in the shell, such as when we run the command:

   ```
   cat file | grep 'a'
   ```

   The pipe character allows us to pass the output of the process '`cat`' as the input to the process '`grep`'. The shell does this by manipulating the standard input and output of each process to get the necessary data from one to the next.

   Another example is through HTTP messages (i.e. networking). In your project, the client and server processes communicate through HTTP requests and responses. This works using the Internet and its associated protocols.

4. (10 mins.) In agile development, a *user story* is a simple description of one or more features of a software system, written from the perspective of the end user. Altogether, they describe the complete desired functionality of a software product. Ideally, multiple engineers simultaneously work on implementing different stories in code, and it is easy to figure out which commits correspond to which story. What features of Git would you use to facilitate this?

   The best way to do this (and indeed the way it is done in industry) is to rely on branches. To implement a story, an engineer should branch off master and use that branch specifically for that story. Branching allows multiple people to be working simultaneously; each can commit and push to their own branches without disturbing the work of others. When they are done, they can merge into the master branch using a pull request, or locally using '`git merge`'. Now, in the project's history there is a merge commit that details and associates previous commits with a given story.

5. (10 mins.) Most of your final projects used an interpreted language like JavaScript or Python on the back-end. Suppose instead you had used a compiled language like C. What benefits or drawbacks would you encounter?

   (If you *did* use a compiled language on the back-end, suppose you used an interpreted language instead. If you used a serverless database like Firebase, discuss the implications of implementing it using a compiled language like C.)

   Benefits:
   — The server will perform better. A compiled binary is usually optimized for the hardware that it is running on, so the server will be faster. We get rid of the cost of interpreting lines of Python/JavaScript.
   — C is a statically typed language. This means that errors in code can be caught at compile-time, whereas you typically only learn of many JavaScript/Python errors midway through execution of your program. This may make development easier.

   Drawbacks:
   — Every time we want to redeploy the server (i.e. we make changes to the source code) we need to recompile it, which takes more time.
   — Generally, Python and JavaScript are considered "easier" languages to

develop in – for example, they also manage memory automatically for you. C is not as forgiving, and may introduce different complications depending on developer experience.
— Python and JavaScript both have built-in support for asynchronous code execution. Doing so in C requires additional setup code and is an altogether unsavory experience.

6. (8 mins.) When the following JavaScript code is run, what does it print out?

```
function callbackHell(a, b) {
  setTimeout(() => {
    console.log("1");
    a(b);
  }, 100000);
  console.log("2");
}

function c(d) {
  setTimeout(() => {
    d();
    console.log("3");
  }, 100000);
  console.log("4");
}

function e() {
  setTimeout(() => {
    console.log("5");
  }, 100000);
  console.log("6");
}

callbackHell(c, e);
```

It will print out:
2
1
4
6

3
5

7. (22 mins.) Consider the following C program:

```c
#include <stdlib.h>
int main(void)
{
  char *buffer = NULL;
  int length = 0;

  for (int i = 0; i < 10; i++) {
    if (buffer)
      free(buffer);

    buffer = malloc(20);
    length++;
  }

  return 0;
}
```

... with the following Makefile:

```
all: main.c
	gcc -o main -O3 -g main.c
```

   a. (2 min.) List **two** different make commands that can generate the executable `main`.

      make
      make all

b. (10 mins.) Consider the following GDB output:

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000400490 <+0>:     push   %rbx
   0x0000000000400491 <+1>:     mov    $0xa,%ebx
   0x0000000000400496 <+6>:     mov    $0x14,%edi
   0x000000000040049b <+11>:    callq  0x400480 <malloc@plt>
   0x00000000004004a0 <+16>:    sub    $0x1,%ebx
   0x00000000004004a3 <+19>:    je     0x4004c1 <main+49>
   0x00000000004004a5 <+21>:    test   %rax,%rax
   0x00000000004004a8 <+24>:    je     0x400496 <main+6>
   0x00000000004004aa <+26>:    mov    %rax,%rdi
   0x00000000004004ad <+29>:    callq  0x400450 <free@plt>
   0x00000000004004b2 <+34>:    mov    $0x14,%edi
   0x00000000004004b7 <+39>:    callq  0x400480 <malloc@plt>
   0x00000000004004bc <+44>:    sub    $0x1,%ebx
   0x00000000004004bf <+47>:    jne    0x4004a5 <main+21>
   0x00000000004004c1 <+49>:    xor    %eax,%eax
   0x00000000004004c3 <+51>:    pop    %rbx
   0x00000000004004c4 <+52>:    retq
End of assembler dump.
```

In the generated assembly code for `main`, there isn't a single add instruction (sub means subtract) corresponding to the code '`length++`'. How can this be the case?

The Makefile specifies `main` to be compiled with optimization level 3 ('-03'). Since the `length` variable is never used, the compiler has completely optimized it away, along with all of its corresponding code.

(Not required) It can also be noted that the instructions for the for loop have been modified (there are no add instructions despite `i++` being in the source code). You can see that there are instructions to move the value "10" (0xA) into the register ebx at <+6>. Instead of counting up from 0 until 10, the program counts down from 10 until 0 (see the sub `$0x1 , %ebx` instruction which subtracts 1 from ebx).

c. (10 mins.) Consider this truncated output from valgrind:

```
$ valgrind --tool=memcheck ./main
==16102== HEAP SUMMARY:
==16102==     in use at exit: 10 bytes in 1 blocks
==16102==   total heap usage: 10 allocs, 9 frees, 100 bytes allocated
==16102==
==16102== LEAK SUMMARY:
==16102==    definitely lost: 10 bytes in 1 blocks
==16102==    indirectly lost: 0 bytes in 0 blocks
==16102==      possibly lost: 0 bytes in 0 blocks
==16102==    still reachable: 0 bytes in 0 blocks
==16102==         suppressed: 0 bytes in 0 blocks
```

What is the reason for this output? How should the source code be fixed?

This output occurs because there is a memory leak in our code. Namely, on the last iteration of the for loop, memory for the buffer is allocated but it is never freed. There needs to be one more additional `free(buffer)` line before the return statement in main.

8. (6 mins.) Write a `grep` command that would output all preprocessor directives given a C file as input.

   Preprocessor directives are lines that start with a '#':

   ```
   grep '^#'
   ```

9. (8 mins.) Most online Git repositories containing C code contain just that: the code. To run the program, someone that clones the repository needs to compile it themselves. Why is this the case? Why don't developers push the executable as well so that others can skip the compilation step?

   Compilers typically generate binaries that are specifically for the hardware and operating system that user has. Different computers have different ISAs (Instruction Set Architectures, the set of assembly instructions available), so if you compiled a program on one computer it may actually not work at all on a different one. Different operating systems have different calling conventions and even executable formats. Therefore, the only real way to guarantee that a binary works on any arbitrary computer is to force the developer to compile it themselves.

   Note that many companies actually do pre-compile binaries when they distribute their products (think Google Chrome, League of Legends, etc.). They compile binaries for multiple platforms and host all of them somewhere (like on their website). That's why you will see links such as "Download this if you have MacOS" vs. "Windows" vs. "Linux", etc.

10. (8 mins.) Why can't two servers be running on the same port on a computer? What bad things would happen if you could do this?

    Transport layer protocols like TCP and UDP rely on the port number to figure out which process to forward a received packet to. In other words, the port number is a way to uniquely identify processes. If you could have two servers running on the same port, it would become significantly harder (and less performant) to figure out which server to forward an incoming packet to.

11. (12 mins.) Suppose you're developing a new language called Viper. Viper is just like Python, except every object now has a free() method that the developer is responsible for calling to deallocate its memory. How is Viper better than Python? How is it worse?

    The most important thing to recognize is that *everything* in Python is an object, even ints and strings.

    Viper is better because it gives the developer more control over when objects are freed. You explicitly know exactly when an object's lifecycle ends and when you can no longer use it. This could arguably be more space-efficient and performant as there is no longer a need for a garbage collector.

    Viper is actually way worse in many ways. First of all, it introduces the chance for error in the form of memory leaks. Now that it is the developer's responsibility to manage memory, every object must be properly freed. This is actually very difficult to do in practice, especially as the program becomes very complex.

    Secondly, it now becomes possible to refer to memory that is already freed. Freeing an object does not necessarily entail getting rid of references to it. If you free a variable and try to perform an operation on it again, this can lead to undefined or incorrect behavior (use after free).

    Finally, having a "free" method on objects is arguably unpythonic. "Beautiful is better than ugly" and "Simple is better than complex." are two tenets of the Python philosophy. Littering code with calls to free will probably detract from the actual purpose of the code.

12. (10 min.) Consider the following two C files and their headers:

```c
// parent.c
#include "child.h"
Parent *parent() {
  Parent *p = malloc(sizeof(Parent));
  p->c = child();
  p->c->p = p;
  return p;
}

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

```c
// parent.h
#include "child.h"
struct Parent {
  Child *c;
};
Parent *parent();
```

```c
// child.c
#include "parent.h"
Child *child() {
  Child *c = malloc(sizeof(Child));
  c->p = NULL;
  return c;
}

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

```c
// child.h
#include "parent.h"
struct Child {
  Parent *p;
};
Child *child();
```

Assuming all memory is freed properly, what's wrong with the above code? How can it be fixed?

One issue with this code is that it has an include cycle. During the preprocessing stage of compilation, all '`#include`'s are replaced with that file's contents. That means inside both parent.h and child.h, a never-ending cycle occurs – including parent.h causes the line '`#include "child.h"`' to be introduced which in turn introduces the line '`#include "parent.h"`'.

There are three ways to solve this. One could use forward declarations in each header file, since the full definitions of the Parent and Child structures are not needed: each header file only refers to the other structure by pointer.

```c
// parent.h
struct Child; // forward-declare Child
```

```
struct Parent {
  Child *c;
};
Parent *parent();
```

```
// child.h
struct Parent; // forward-declare Parent
struct Child {
  Parent *p;
};
Child *child();
```

Additionally, as done in CS 31 and 32 assignments, we can use *include guards* in both header files (we'll still need forward declarations):

```
#ifndef PARENT_H
#define PARENT_H

#include "child.h"
struct Child;
struct Parent {
  Child *c;
};
Parent *parent();
#endif // PARENT_H
```

(There are also other ways of adding include guards, such as the nonstandard extension '#pragma once'.)

Another alternative is to consolidate child.h and parent.h into one file, which is probably the best in this kind of situation in general. Note that we still need a forward declaration for one of the classes.

```
struct Parent;
struct Child {
  Parent *p;
};
struct Parent {
```

```
  Child *c;
};
Parent *parent();
Child *child();
```

A second issue is that both child.c and parent.c define their own `main()` function. When the compiled object files child.o and parent.o are linked together, the linker will complain that the same symbol (function) is defined by two object files: a symbol clash.