COMPSCI 3SH3 Winter, 2021
Student Name: Jatin Chowdhary
Mac ID: Chowdhaj
Student #: 400033011
Date: February 14th, 2021

Assignment #1

Ethics

1. a)

The act of (straight up) copying code from any source is unethical in all settings; professional, educational, personal, etc. However, there are certain exceptions to this rule. For this question, I will discuss the process of using somebody else's code in a professional setting. In order to use somebody else's code, the first thing you need to do is seek out the permission of the author(s), get their approval, and a license to use their code. You should explicitly state why you need their code and where it will be used. In addition, if they have feedback to offer, then you should listen to it. After you have the necessary permission(s) and license(s), the next step would be to inform your team that you obtained code from an alternate source. Getting their permission, and making them aware is also important. The next step is to thoroughly review the code. You need to make sure that:

=> There are no bugs in the code

=> You completely understand the code (i.e. It's best/worse/average case, pitfalls, etc.)

=> The code passes all test cases

Once you have thoroughly vetted the code, the last step is to include appropriate attribution to the original author(s). In layman's terms, give full credits to the author(s). This can take place in the form of a (multi-line) comment that sits above the code, or in a README file that is visible to everyone else working on the system. A link to the original code should also be included. If, and only if, all these steps are followed, then it is ethical to copy code from an alternate source.

1. b)

The practice of straight up copying and pasting code from alternate sources into your software system is bad engineering practice. In fact, in some cases it can even be considered malpractice (i.e. When working on a (medical) software system that deals with life and death). However, as an engineer, there are certain steps you can take to ensure that good engineering practices are influencing the design and production of your system; most notably, following the IEEE Code of Ethics, and good software engineering practices.

Before you incorporate the code, you should take the time to thoroughly understand it, how it works, its pitfalls, best case, worst case, average case, and other relevant information that will help you make informed decisions. After thoroughly reviewing the code, you should modify it, so it adheres to good software engineering practices such as, but not limited to:

=> Modularity

=> Information Hiding

=> Separation of Concerns

Furthermore, if your development team has agreed to a specific format, then you need to modify the code so it follows the pre-determined format. For example, the team has agreed on using spaces, not tabs, and wants to follow GNU's formatting standards. If there is a (formal) design specification, then the code needs to comply with the specification. For example, the specification limits the use of libraries and prioritizes code that is written in-house. Finally, you need to run your new and improved code on several test cases to ensure that it works as expected.

You should also keep in mind the IEEE Code of Ethics (rules) and follow them when possible. The IEEE rules you need to follow are, but not limited to:

=> Rule 1: To hold paramount the safety, health, and welfare of the public

=> Rule 2: To avoid real or perceived conflicts of interest

=> Rule 3: To be honest and realistic in stating claims or estimates based on available data

=> Rule 7: To seek, accept, and offer honest criticism... to acknowledge and correct errors... to credit properly the contribution of others

=> Rule 9: To avoid injuring others, their property, reputation, or employment by false or malicious action

(*IEEE Code of Ethics,* 2020)

Note: If you follow good software engineering practices, then most of these IEEE rules will be indirectly achieved.

Introduction

2.1)

An interrupt transfers control to the interrupt service routine (ISR). The ISR processes the request and sends it to the CPU. Generally, this is done through the interrupt vector (Galvin, Gagne, & Silberschatz, 2018). When an interrupt occurs, the flow of execution is transferred to another process, typically the one that caused the interrupt. For example, you send a 10 page document to your printer for printing. Printers are I/O devices, and are really slow compared to a CPU. Having the CPU wait for the printer to finish is inefficient, time consuming, and a poor management of hardware resources. The solution to this problem is interrupts. The printer can take its sweet time to print, and the CPU can process information for other programs. Once the printer finishes, it signals to the operating system (OS) that the printing job has completed and the printer needs some CPU time for the next event. This is the purpose of interrupts; to signal to the operating system that an action needs to be taken on an event, and CPU time is needed. This event isn't limited to printers, it can also be an exception generated by a process, a process requesting I/O data, a process signalling its completion, etc. Operating systems are interrupt driven (Galvin, Gagne, & Silberschatz, 2018).

Traps, not to be confused with Mach traps, are software generated interrupts caused by an error or a user request (Galvin, Gagne, & Silberschatz, 2018). For example, if you try to divide by zero in Java, the program will throw an "ArithmeticException". If you try to access the 11^{th} element in an array with a size of 2, Java will throw an "ArrayIndexOutOfBoundsException". If you try to store a String into an int, Java will throw an "InputMismatchException".

A trap is typically caused by software, for all the reasons stated above. An interrupt, on the other hand, is typically caused by hardware (i.e. I/O devices). However, both are used to signal to the OS that it needs CPU time to process the next thing. From an abstract point of view, they accomplish the same thing; to get CPU time for further processing.

Traps can be intentionally generated by a user-space program (Galvin, Gagne, & Silberschatz, 2018). For example, when a program says, "Press any key to continue", "Please insert the target device", or "Copy complete, remove the device". A timer going off is also an intentionally generated trap. When the timer hits zero, it requests CPU time from the OS, and then plays a sound. All of these scenarios have one thing in common – they all want CPU time.

2.2 a)

When a high-speed I/O device uses direct memory access (DMA) to copy data into main memory, the CPU is only involved at the beginning and end of the transfer. Everything in between is carried out by the DMA controller. The initial step is carried out by the CPU, with the help of the DMA registers. These registers contain important information relevant to the memory transfer (i.e. Size of data, Location in memory, etc). Once the DMA registers are setup, the data is copied from the I/O device into main memory without going through the CPU. This process is carried out by the DMA controller (Galvin, Gagne, & Silberschatz, 2018).

2.2 b)

Operating systems are interrupt driven. Interrupts are used by the DMA controller to signal the CPU that the operation has completed, and it needs CPU time to execute the next set of actions. There are two ways that the CPU can figure out when the memory operations have been completed. The first is interrupts. The DMA controller can send an interrupt, alerting the CPU that the memory transfer has completed, and it needs CPU time to execute the next set of instructions. Another way that the CPU can be alerted is by periodically checking the values of the DMA register to see if they contain a special value (Galvin, Gagne, & Silberschatz, 2018). This value indicates the completion or failure of the memory transfer. However, this approach is not recommended and can degrade the performance of the entire system. The best way for the CPU to know when the memory transfer is complete is for the DMA controller to interrupt the CPU (Galvin, Gagne, & Silberschatz, 2018).

2.2 c)

Normally, this process does not interfere with the execution of other user programs, because the CPU and DMA controller are processing their own (unique) information. However, the CPU executes more slowly during a transfer when its access to the bus is required (Galvin, Gagne, & Silberschatz, 2018). Another interference/problem can be caused if the CPU and DMA controller try to access the same bit of memory (Galvin, Gagne, & Silberschatz, 2018). This is the classic concurrency problem of two entities trying to access the same piece of information.

OS Structures

3.1)

Android (Differences)	Similarities	iOS (Differences)
=> The base version of Android (AOSP) is fully open source, and other people can use it to create their own version of Android (i.e. LinageOS, AOKP, GrapheneOS/CopperHeadOS, OmniRom, Pixel Experience, etc). However, it is important to note that Google's core apps are all proprietary and closed source. Although, there are initiatives (i.e. MicroG) that are alternatives to Google's Android libraries. => Can be installed on just about any device (i.e. Phones, computers, tablets, thermostats, cars, fridges, watches, etc). This is why multiple companies (i.e. Samsung, LG, Sony, etc.) ship their phones with Android. => Allows users to install 3rd party apps quite effortlessly; this allows you to run unsigned code. => Most vendors allow users to root their Android devices. => Downgrading to previous versions of Android is allowed => Developed by the Open Handset Alliance, and is now maintained by Google. => Based on Linux Kernel. => Apps are (mostly) written in Java plus the Android API. => Updates are slow to roll out, and often times are dependent on the vendors (i.e. Samsung). => Voice assistant is Google Assistant. => Android phones vastly vary in prices; some are $50 and others are $2000. => Has a tremendous amount of customization options.	=> Have a similar stack => Were built for mobile phones, and that is their main use => Have an app store where developers can upload their programs for others to download. Plus, Apple and Google take a gargantuan 30% cut of your profits => Well crafted exploits for each OS fetch a high price in the exploit black market (i.e. Zerodium) => Contains power management, and battery monitoring, software (i.e. Deep sleep & Low power mode (LPM)) => Have spin-offs to work on other devices (i.e. iPadOS for iPads, watchOS for smart watches, etc.) => Main functionalities are largely the same (i.e. Call, text, take pictures, record video, listen to music, browse the web, download apps, play games, watch movies, etc.) => New features being added are largely the same (i.e. Dark mode, privacy options, etc.) => Disavowed by Richard Stallman => Apple and Google are getting sued for Anti-Trust violations, because their software is too restrictive, and their business model hurts competition.	=> Mix of open source components, and proprietary software; you cannot modify it and create your own version of iOS. => Strictly for iPhones. The only phone that you can buy with iOS on it is an iPhone. Other than Apple's iDevices, no other device can run iOS natively. However, you can emulate iOS using Corellium. => Is a walled-garden, and you cannot (easily) install apps without Apple's approval; side-loading does not count because it is tedious and a hassle. => Obtaining root privileges, to run unsigned code, requires the user to jailbreak the iPhone (i.e. unc0ver, checkra1n, Taig, Pangu, evasi0n, limera1n, etc.) => Cannot downgrade iPhones to previous versions, unless they are signed by Apple. So, once an iPhone is updated to the latest version of iOS, it cannot be downgraded to previous versions. => Developed and maintained by Apple. => Based on Darwin (BSD), and the kernel is a hybrid (XNU). => Apps are (mostly) written in Objective-C or Swift. => Has better support with constant updates for security, performance, and bugs. => Voice assistant is Siri. => iPhones are very expensive, and overpriced. => Limited access to file-system. => Simple, and easy to use with limited customization options.

3.2 a)

System calls provide an interface, to processes/threads, to the services made available by an operating system (Galvin, Gagne, & Silberschatz, 2018). For example, a process that requires concurrency or parallelism might use the *fork()* system call to create a child process, split the computation across two processes, and multiple cores, if present. The *read()* and *write()* system calls are used in pipes, an interprocess communication method, which allow a parent and child process to communicate. System calls are mostly accessed by processes, or threads, via a high-level application programming interface (API) (Galvin, Gagne, & Silberschatz, 2018). This API is essential because it follows the practice of information hiding. For example, if a user program needs more RAM, it can request it from the kernel/OS. The intricacies of memory management are handled by the kernel/OS, and the user program gets the resources it needs without having to worry about the finer details of memory management, or knowing how the system call is implemented.

3.2 b)

System programs are essential functions in every operating system. They sit ideally until invoked (Galvin, Gagne, & Silberschatz, 2018). These programs are used to carry out common tasks that the user and other processes need. For example, deleting files is very common and useful in an operating system. In Linux, the system program *rm* accomplishes this. System programs vastly vary in their functionality. System programs are used for file management, communication, status information, managing programs, helping other processes, etc. Examples of other system programs include, but is not limited to: *cd*, *ls*, *rm*, *format*, *mkdir*, *touch*, etc.

3.2 c)

When text is entered into a terminal, the command interpreter takes the input, and converts it into an executable command with arguments, if any. In most operating systems, the command interpreter does not understand commands in any way, and merely uses the command to locate an executable file that contains the command's code (Galvin, Gagne, & Silberschatz, 2018). The main function of the command interpreter is to get and execute the next user-specified command (Galvin, Gagne, & Silberschatz, 2018). It can also help users complete command names and arguments (i.e. Pressing *tab* to auto-complete the query, or pressing the *arrow keys* to cycle through previously executed commands).

The command interpreter is usually separated from the kernel because the command interpreter only needs to be programmed once, and does not (really*) require subsequent updates. Its only job is to be able to parse text, find the corresponding command, and execute it. Separating the kernel from the command interpreter is good software engineering practice; design for change and separation of concerns. Adding, or updating, commands, or system programs, does not require the command interpreter to be modified. On the other hand, system programs, kernel, etc. need to be constantly updated due to bugs, new features, etc. Modifying these will not break the command interpreter. Even if there is a new update to the operating system that is radically different (i.e. New file-system), the command interpreter will continue to function, or it will require a minor update.

*Note: The command interpreter may need to be updated if adding support for a new language, or changing the functionality of something.

Processes

4. a)

A = 0

B = 2603

C = 2603

D = 2600

4. b)

Introducing concurrency to a system causes the following complications:

=> Deadlock; a major problem introduced by concurrent processing is deadlock. Deadlock occurs when tasks are waiting on each other. For example, we have 3 tasks: A, B, and C. If A is waiting on B, and B is waiting on C, and C is waiting on A, then there is deadlock and nothing gets done. The program freezes/crashes. Deadlock can also occur if a concurrent system is badly programmed. For example, the programmer forgets to release the mutex lock, preventing other threads from accessing the data it needs.

=> Starvation; another problem introduced by concurrency is starvation. Starvation occurs when a task does not get the resources it needs to complete its job because other processes are hogging CPU time, or because its priority is too low. If the task cannot be completed, then it may affect the performance of the entire system, especially if other tasks are dependent on it. This can also lead to deadlock.

=> Complex; the added complexity of concurrency, makes it much (much) harder to (properly) design, program, test, and debug a concurrent system; when compared to non-concurrent systems. It takes more time, and money, to create a successful concurrent system. Maintaining a concurrent system takes a tremendous amount of work, and requires collaboration across multiple people.

=> Security; concurrency introduces new security holes to a system that can be maliciously exploited by an attacker. If concurrency isn't properly implemented it can lead to race condition bugs, memory corruption, buffer/stack overflow, etc. A race condition occurs when task 'B' depends on task 'A', but finishes before 'A'. Race conditions are common bugs in concurrent systems.

References

Galvin, P., Gagne, G., & Silberschatz, A. (2018). *Operating System Concepts* (10th ed.). Hoboken, NJ: John Wiley.

This is the course textbook, and is the primary source of information.

IEEE Code of Ethics. (2020, June). Retrieved February 10, 2021, from https://www.ieee.org/about/corporate/governance/p7-8.html