Question 1

1.1.

Interrupt

An interrupt is an asynchronous, hardware-generated change of flow within an operating system. Hardware will trigger an interrupt by sending a signal to the CPU. In some cases, software may trigger an interrupt by executing a special operation called a system/monitor call.

Trap

A trap is a synchronous, software-generated interrupt. A user program can intentionally generate a trap as an exception. Traps can be used to call operating system/kernel routines or be an indicator of errors (division by zero or invalid memory access in user processes).

1.2.

Single-processor system

There is typically one cache and only one process being access sequentially by the CPU. The CPU will update a file in the cache and later update it in memory reliably and consistently.

Coherence: Issues will only occur in a multitasking environment where the CPU has to switch between various processes. When trying to access a copy of a file in each process concurrently, the system needs to ensure that each process is accessing the most recently updated copy of the file.

Multi-processor system

Each CPU could contain its own local cache. Each CPU will update a file in the cache and later update it in memory. Copies of the same file could be stored in different caches in a multi-processor system.

Coherence: When trying to update a copy of the file in each CPUT concurrently, the entire system needs to communicate with all CPU's that contains a copy of the file to ensure that all copies of the file are updated simultaneously.

Distributed system

No cache or memory is typically shared; however, several copies of the same file could be stored on different computers/nodes in a distributed system. Each node will update a file in its local cache and later update it in its local memory.

Coherence: When trying to update these files concurrently, the entire system needs to needs to communicate with all nodes that contain a copy of the file to ensure that all copies of the file are updated simultaneously.

Question 2

2.1.
In an operating system, system calls can be used for both file and device manipulation (through a file interface) to allow for devices to be access though they are a file in the file system.
Advantages
- Simplifies the development of program code.
- Simplifies the development of device-driver code (rewritten to support API calls). This provides several benefits such as support made for multiple versions of the operating system, additional functionality and reduction of the lines of code needed to perform the same function of many lines of code for a system call.
Disadvantages
- When using the aforementioned APIs as a simplification of system calls, it may be difficult to access the full functionality of a device. With this reduced functionality, may also lead to performance loss.

2.2.
Shared-memory model
Advantages
- Allows maximum speed when the processes are on the same machine.
- Offers better convenience of communication
Disadvantages
- Different processes need to ensure that they are not writing to the same location simultaneously.
- Processes that communicate using shared memory need to address problems of memory protection and synchronization.
- Potential for security issues

Message-passing model.
Advantages
- Much easier to implement
- Useful for exchanging smaller amounts of data
- No conflicts to avoid
Disadvantages
- Communication is slower than shared memory because of the time it takes for the connection setup.

2.3.
Similarities
- both designed to run on smartphones and tablets to provide functionality such as calling, messaging, web browsing, video chat, maps voice commands etc.
- both based on existing kernels. Android is based on the Linux kernel and iOS based on Mac OS X.
- both have architecture that uses software stacks
- both provide frameworks for developers to write software for the respective systems

Differences

Android	iOS
Customizable: 3rd party manufacturers often provide custom version of Android on their devices and Android is available on the devices they may produce	Available on devices Apple manufactures.
Uses an open-source model	Closed, with open-source components
Written in Java. It uses a virtual machine (Darkvil virtual machine)	Written in C, C++, Objective-C, Swift and assembly language.
Google provides an Android API for Java development of applications	Applications are developed in Objective-C

Question 3

3.1.
Every time `fork()` is used, create a new level in a binary tree, with the initial process spawning two child nodes. The formula for calculating the number of processes is:

Total Number of Processes = $2^n - 1$
Where n is the total number of fork system calls and 1 represents the initial parent process. So, including the parent process for figure 3.32 we have:
$\therefore 2^4 - 1 + 1 = 16$

3.2.

line X:
CHILD: 0 CHILD: -1 CHILD: -4 CHILD: -9 CHILD: -16
line Y:
PARENT: 0 PARENT: 1 PARENT: 2 PARENT: 3 PARENT: 4

Question 4

4.1.
A multithreaded solution would perform better single-processor system when a kernel thread suffers a page fault; another kernel thread can be switched in to use the interleaving time in a useful manner.

On the contrary, a single-threaded process, will not be capable of performing useful work when a page fault takes place.

4.2.
Amdahl's law states that the overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used. It is best described by the speedup equation:

$$t = \frac{1}{(S+(1-S)/N)}$$

Where S is the portion of the application that must be performed serially and N is the number of processing cores.

40 percent parallel with:

(a) eight processing cores ($S = 1 - 0.4 = 0.6$)

$$t = \frac{1}{(0.6+(1-0.6)/8)} = \frac{1}{(0.6+0.05)} = \frac{1}{0.65} = 1.53846153846$$

(b) sixteen processing cores ($S = 1 - 0.4 = 0.6$)

$$t = \frac{1}{(0.6+(1-0.6)/16)} = \frac{1}{(0.6+0.025)} = \frac{1}{0.625} = 1.6$$

67 percent parallel with:

(a) two processing cores ($S = 1 - 0.67 = 0.33$)

$$t = \frac{1}{(0.33+(1-0.33)/2)} = \frac{1}{(0.33+0.335)} = \frac{1}{0.665} = 1.5037593985$$

(a) four processing cores ($S = 1 - 0.67 = 0.33$)

$$t = \frac{1}{(0.33+(1-0.33)/4)} = \frac{1}{0.33+0.1675} = \frac{1}{0.4975} = 2.010050251$$

90 percent parallel with:

(a) four processing cores ($S = 1 - 0.9 = 0.1$)

$$t = \frac{1}{(0.1+(1-0.1)/4)} = \frac{1}{0.1+0.225} = \frac{1}{0.325} = 3.07692307692$$

(b) eight processing cores ($S = 1 - 0.9 = 0.1$)

$$t = \frac{1}{(0.1+(1-0.1)/8)} = \frac{1}{0.1+0.1125} = \frac{1}{0.2125} = 4.70588235294$$

4.3.
For this question, we will assume that the child process will execute.
First `fork()` is called:

```
[p0] initial process
       // new subprocesss
       [p1] child of p0
```

Second `fork()` is called:

```
[p0] initial process
       [p1] child of p0
       // new subprocesss
              [p2] child of p1
```

$\therefore 2^2 - 1 = 3$ (1 parent, 2 child processes)

Afterwards, `thread_create(...)` is called, which creates a new thread (2 operations, thread 0 and thread 1) within a parent task. We need to create a thread for each unique child:

```
[p0] initial process
       [p1] child of p0
              thread 0
              thread 1
              [p2] child of p1
                     thread 0
                     thread 1
```

Third `fork()` is called.
This creates a new subprocess in p0, p1 and p2 respectively.
Each new subprocess spawned by a parent with a thread will also contain a thread:

```
[p0] initial process
       // new subprocess
       [p3] child of p0
       [p1] child of p0
              thread 0
              thread 1
              // new subprocess
              [p4] child of p1
                     thread 0
                     thread 1
              [p2] child of p1
                     thread 0
                     thread 1
                     // new subprocess
                     [p5] child of p2
                            thread 0
                            thread 1
```

Therefore, there are:
(a) 6 unique processes
(b) 8 unique threads

Question 5

5.1.
Shortest Job First (SJF) is one of several CPU Scheduling strategies that aims to predict the length of the next CPU burst. The dynamic method describes two formulas for calculating SJF, mainly the simple average and exponential average methods. Using the exponential average method:

Exponential average (aging)

$$T_{n+1} = \alpha t_n (1 - \alpha) T_n$$

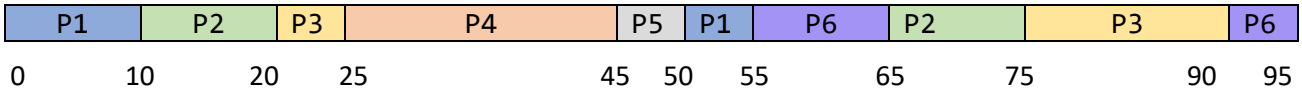
Where:
$t_n =$ actual length of n^{th} CPU burst
$T_{n+1} =$ predicted value for the next CPU burst
$\alpha, 0 \leq \alpha \leq 1$

(a) $\alpha = 0$ and $T_0 = 100\, milliseconds$
Where $\alpha = 0$, the condition of the formula $\alpha, 0 \leq \alpha \leq 1$ essentially does not apply. This being said, the most recent history which is used to give an estimate of the next CPU burst will have no quantifiable effect. Thus, the predicted value will remain the same, $100\, milliseconds$.

(b) $\alpha = 0.99$ and $T_0 = 10\, milliseconds$
Where $\alpha = 0$, the condition of the formula $\alpha, 0 \leq \alpha \leq 1$ has a much higher weight than the past history associated with the process. The issue in this case is that only the recent history will be used, replacing the prediction entirely.

5.2.
(a)
Gantt chart

Process	Priority	Burst Time	Arrival
P1	8	15	0
P2	3	20	0
P3	4	20	20
P4	4	20	25
P5	5	5	45
P6	5	15	55



(b) Turnaround = end time – arrival time
P1: 55-0 = 55
P2: 75-10 = 65
P3: 90-20 = 70
P4: 45-25 = 40
P5: 45-45 = 0
P6: 95-55 = 40

(c) Wait time = Turnaround time - burst time
P1: 55-15= 40
P2: 65-20 = 45
P3: 70-20 = 50
P4: 40-20 = 20
P5: 0-5 = 0
P6: 40-15 = 25

5.3.
(a)
The time quantum is 1 millisecond, irrespective of which task is running.
Time required for ALL processes: 1.1 ms
(tasks execute for 1ms then incur overhead cost)

CPU utilization: 0.1 millisecond overhead

$$CPU\ utilization = \frac{time\ quantam\ for\ tasks}{time\ require\ for\ tasks}$$

$$CPU\ utilization = \frac{1ms}{1.1ms} * 100 = 91\%$$

(b)
The time quantum is 10 milliseconds. The I/O-bound tasks incur a context switch after using up only 1 millisecond of the time quantum.
Time required for I/O processes: 10*1.1 ms
(tasks execute for 1ms then incur overhead cost)
Time required for CPU processes: 10.1 ms
(tasks execute for 10ms then incur overhead cost)

CPU utilization: 0.1 millisecond overhead

$$CPU\ utilization = \frac{time\ quantam\ for\ tasks}{time\ require\ for\ tasks}$$

$$CPU\ utilization = \frac{10ms+10ms}{(10*1.1)+10.1ms}$$

$$CPU\ utilization = \frac{20}{21.1} * 100 = 94\%.$$

Question 6
6.1.
6.2.
6.3.

Question 7
7.1.
7.2.

Question 8

8.1.
Type 0 hypervisors
Implemented by Firmware. Low overhead but generally fewer features. These VMMs, which are commonly found in mainframe and large to midsized servers.

Type 1 hypervisors.
Special purpose software or general-purpose operating systems that are built to provide virtualization.

Type 2 hypervisors
Applications that run on standard operating systems but provide VMM features to guest operating systems. More overhead and fewer features than type 1

8.2.
A CPU is able to execute instructions at two levels: user mode and kernel mode (elevated privileges). A VMM or hypervisor will have two separate modes: virtual user mode and virtual kernel mode.

If the guest attempts a privileged instruction, the hypervisor will gain control, analyse the error, execute the operation and return to the guest in user mode. This is trap-and-emulate.

The issue occurs when some CPU's do not separate between privileged and nonprivileged instructions. Early intel x86 CPU's are among these. In these cases, trap-and-emulate cannot be utilized. Instead, binary translation is utilized.

Binary translation executes with the following logic:

- If guest VCPU is in user mode, guest can run instructions natively
- If guest VCPU in kernel mode (guest believes it is in kernel mode)