1.14 (10 points) Under what circumstances would a user be better off using a timesharing system rather than a PC or a single-user workstation.

Timesharing can be used where a single-user workstation is not financially or practically (i.e. available space) possible. As a consequence of those issues, a facility may only have one or two computers available, but a large number of users who need to perform some sort of work on the machine. Rather than taking turns (single-user) on those machines, time sharing can be used to allow all users to use the one system simultaneously.
The system can periodically switch between working each users jobs, thus creating the illusion that each user is using their own personal system, when in fact there is only one machine.

1.20 (10 points) Direct memory access is used for high-speed I/O devices in order to avoid increasing the CPU's execution load.

(a) How does the CPU interface with the device to coordinate the transfer?

(b) How does the CPU know when the memory operations are complete?

(c) The CPU is allowed to execute other programs while the DMA controller is transferring data. Does this process interfere with the execution of the user programs? If so, describe what forms of interference are caused.

(a) The operating system uses device drivers (which run on the kernel) to communicate with the I/O devices. In order to allow the CPU to continuing executing programs during I/O operations, a DMA can be used, this can load device drivers, wait for device operations to complete, and then send an interrupt to the CPU once the operation is complete.

(b) The I/O device can send an interrupt to the CPU, letting it know that the memory operations are complete. The CPU now knows that that memory location is once again available, and can resume the process requiring the completion of that I/O operation at the CPU's leisure.

(c) In general, no, the CPU can run any other process freely while the DMA controller is transferring data. There may be small situations where for example if the DMA is busy another program may have to wait to make memory requests. Further when the DMA

sends an interrupt, this can cause the current user program being executed by the CPU to be temporarily suspended.

1.25 (10 points) Describe a mechanism enforcing memory protection in order to prevent a program from modifying the memory associated with other programs.

User programs are given their own local 'virtual' memory spaces which are independent of physical device locations. When memory operations go through the CPU, the CPU can check that addresses given are in fact within the local 'virtual' memory space assigned to the program (base and limit registers), if so the location can be converted to a physical address and memory operations performed without ever telling the program the physical address. However, if the address is not within the local memory space of the user program, an exception can be generated and sent to the user program (segmentation fault) causing the operation to fail, thus protecting memory used by other programs. This is implemented via a physical trap on the CPU.

2.18 (10 points) What are the two models of inter-process communication? What are the strengths and weaknesses of the two approaches.

Message Passing Model: With a message passing model, messages can be sent in either a blocking or non-blocking way (synchronous or asynchronous). This method is much easier to implement, but isn't very suitable for large amounts of data as message passing requires more overhead (as message passing is performed via system calls and must pass through the kernel) and thus results in slower speeds.

Memory Sharing Model: Memory is allocated which both processes have access to. Thus read/write operations are direct, and thus much faster than message passing, making memory sharing much more suitable for large blocks of data. However, memory sharing can cause synchronization issues when multiple processes read/write to the same location. A simple example is if both processes get a value and then increment it, depending on the order of the 2 sets of get/set operations, the value may be incremented by 1 or 2 ({get1,get2,set1,set2}$\Rightarrow +1$ vs {get1,set1,get2,set2}$\Rightarrow +2$)!

2.23 (10 points) How are iOS and Android similar? How are they different?

At a high level, Android and iOS are very similar in their goals to be mobile friendly, relatively lightweight (compared to a desktop os) operating systems. Where iOS was initially largely based off the core of the Mac OS X operating system and thus uses the XNU kernel of Darwin, Android was built off of the Linux kernel. Though different in many regards (XNU is hybrrid, Linux is monolithic), both are unix-like kernels, and aim to be POSIX compatible.

Overall, iOS is built explicitly to be run on Apple devices (iPhone & iPad) where as Android, being completely open source (with exception of some drivers) in nature can run on nearly any device meeting minimum resource requirements.

On the subject of open vs closed source, iOS is largely closed source, with small bits of open source components at its core (XNU for example is open source) in contrast to Android.

6. (25 points) Review the memory management for a C program (read the recitation notes if you haven't got the chance to attend recitation), and then study the following C code and answer questions:

```
char a;
int main() {
        foo();
        // ...
}

int foo() {
        char * b, * c;
        char d;

        b = (char *)malloc(10);
        c = (char *)&a;
        d = 10;
        return d;
}
```

Where (data segment, stack, or heap) are (a) variable a, (b) variable b, (c) the space pointed by b, (d) variable c, (e) the space pointed by c, and (f) variable d stored respectively? After foo finishes its execution, which of the above variable/space are/is reclaimed by the OS?

(a) Global variable ⇒ **data segment**. This value will not be reclaimed after foo finishes executing and will exist for the entire duration of the programs execution.

(b) Local variable ⇒ **stack**. Exists only in the scope of foo, once foo is done executing, the variable will be popped off the stack (reclaimed by OS).

(c) Malloc'd data ⇒ **heap**. Variable 'b' points to malloc'd data in the heap, this will not be reclaimed by the OS until 'free' is called on that address. As this never happens in our program, we actually have a memory leak here.

(d) Local variable ⇒ **stack**. Exists only in the scope of foo, once foo is done executing, the variable will be popped off the stack (reclaimed by OS).

(e) Variable 'c' points to the address of 'a'. This means it is stored in the **data segment**. And, just like 'a' (because it is a), it will exist for the entire duration of the programs execution. Because no 'malloc' was used, no corresponding 'free' is necessary.

(f) Local variable ⇒ **stack**. Exists only in the scope of foo, once foo is done executing, the variable will be popped off the stack (reclaimed by OS).

7. (25 points) When the following C program is run in a Linux system, the execution of which line(s) must trigger invocation(s) of system call and/or exception? When will the program terminate?

```
int main() {
        double w = 1234.789;
        double x = 0;
        double y;
        double * z;

        /* (1) */ z = (double *)&w;
        /* (2) */ scanf("%f", &y);
        /* (3) */ *z = sqrt(w);
```

```
        /* (4) */ y = y * 2.0;
        /* (5) */ printf("y=%f, *z=%f\n", y, *z);
        /* (6) */ y = y/x;
        /* (7) */ w = y * z;
        return 0;
}
```

Lines (2) and (5) are both the obvious examples of system calls as the pertain to file I/O (through the file descriptors stdin and stdout). Line (3) is NOT a system call, sqrt is only a function defined by the C standard library, and is not actually a wrapper around any system call.

Although z is a pointer, we don't actually malloc anything, it's only a reference to w, so no system call is needed there, line 3 is taking the square root of w and assigning it back to w via the reference. z, nothing all that interesting.

Line 7 is a syntax error, as you cannot multiply a pointer (z) with a value (y). Therefore you will not actually be able to run this program, so it is thus impossible for it to terminate. If you included a typecast, to convert (z) to an integer (as that is all an address is), it will not be particularly disastrous, you will just end up with an arbitrary value stored in (w), dependant on whatever happens to be the address of (w). Alternatively, you could also dereference z (to the value of w), this would also be fine.

The program will return 0 following either of these corrections.