1 Class #1

The *operating system* can be thought of as the main body of code that allows the hardware to provide services to users. At the same time, the operating system has the responsibility of providing a certain amount of isolation from the hardware to the application programmer. Essentially then, the operating system is a service provider. Consequently, there is a close relationship between hardware evolution and the characteristics of an operating system. The rest of this section will consider hardware evolution and its effect on the operating system evolution.

1.1 OS and hardware evolution

The earliest machines (circa 1949) were extremely different from the machines available today. The machines had to be programmed at the level of individual bits (so called "machine language"). The environment at such a place was like in an open shop. One could basically sign out the machine for a specific amount of time. One had to sit at the console, debug and load one's own tapes. About the only support the operating system provided was the ability to provide a loader which would load the user's program into the computer's memory. The user's program was typically in the form of a punched card or a magnetic tape drive. The operating system provided a "loader" which would load this program into memory and execute it. This was the **first generation** operating systems. These machines subsequently evolved into the stage where the operating system came provided with an assembler. An assembler enabled a little bit more separation between the programmer and the hardware. The programmer did not need to be physically present near the hardware anymore. A program could be typed up in a semi-readable form (assembly language), then assembled by the built in assembler into machine language, loaded into the system's memory and executed. This was essentially, what is known as a *batch* mode of operation. The (fast) system was basically halted every time the (slow) operator had to enter a new deck of cards corresponding to a new job. In these preliminary systems, I/O devices such as tape drives were as slow as the CPU and I/O was done synchronously.

The **second generation** of operating systems were necessitated by the second generation of computer hardware. Valves were replaced by much faster circuit elements consisting of transistors. This increased the processing power of the CPU. At the same time, newer output devices such as disks appeared. While disks were faster than tape drives, they were still not as fast as the newer CPUs. So, in these newer systems, I/O was asynchronous and *Interrupts* were used to notify the CPU to indicate completion of I/O operations. An interrupt mechanism enabled the CPU to stop and save the status of the current instruction stream and jump to a new (usually predefined) program address to *handle* the interrupt. After the interrupt was handled, the control of program execution returned to the initial instruction stream. Having an interrupt mechanism made I/O more efficient. For example, when the CPU wrote some data out to disk, it did not have to wait for the data transfer to the slow disk to complete. Instead, it could just initiate the write operation and the disk would interrupt the processor when the data transfer operation completed. The notion of I/O interrupt was generalized to include *clocks*, *timers* and *supervisor interrupts*. Also, interrupts could be used to increase the throughput of a system by enabling multiple programs to execute simultaneously. For example, imagine a process that needs to wait for some data it is reading from disk before it can perform any computation. By having interrupts, it is possible to execute some other process while the data is being read. After the data is read completely, the device can interrupt the CPU to let it know that the data transfer is complete. This information can be used to restart the process that needed the data.

The **third generation** of operating systems was a natural evolution of the third generation hardware consisting of integrated chips. ICs led to the CPU being much much faster than before, whereas the nature of I/O did not change significantly. So, it became *necessary* to be able to run multiple programs *concurrently*. *Timers* provided a convenient means to provide the ability to *schedule* a large number of processes. *Interrupts* were used for I/O. However, once multiple processes running concurrently was possible, it also became necessary to prevent the various processes from clobbering each other. This led to *Protection* being a really important issue. Mechanisms such as *base* and *bounds* registers could be used to provide protection to the *address spaces* of individual processes. However, since such a mechanism was useless if users could modify these protection mechanisms, it became necessary to provide two modes of execution to the processor. A normal *user* mode would be used to run regular applications. A *supervisor* mode would be used to execute very specific instructions related to matters of protection. This gave rise to *dual-state* systems. A *user* mode could not execute specific types of instructions such as I/O and state-change operations. Another contribution of this generation of operating systems was the introduction of *virtual storage*. Hardware support for virtualization and relocation of memory made it easier to run multiple applications concurrently. Special *translation* interrupts were introduced to convert virtual memory addresses to physical addresses and to seamlessly bring data into physical memory if it was not there. Finally, *system calls* interfaces were provided to user processes to request the kernel to perform services (such as I/O) that the user process could not perform itself.

The **fourth generation** of operating systems that we currently see can be broadly classified into three classes depending on the kind of hardware that they run on and the classes of services they provide for users. The first in this class is *personal computers*. The typical operating system (win95) for such machines provides ease of use and plug-and-play facilities. Special features such as *menus*, *windows* and *mouse* are provided to make life easy for the user. The second class of fourth generation operating systems are operating systems for *multiprocessors*. Here the emphasis is on operating systems to provide primitives for scheduling of processes, enhanced support for synchronization, communication and global collective operations. The final class of modern operating systems is the class of *distributed operating systems* typical of the class of *client-server* computing. The issues here are fault tolerance, heterogeneity and high performance in the presence of large amounts of latency.

A modern day machine has the typical appearance in figure 1. The machine consists of a processor, (typically a RISC processor), main memory and I/O devices. The devices usually seen are disks, tape drives, keyboards and network intefaces. These devices are connected by a high-speed bus. Processor speed is typically an order of magnitude faster than the memory which is in turn an order of magnitude faster than the I/O devices. I/O devices use interrupts to intimate the processor of their state. Since the types of interrupts are predefined and the interrupt must be handled very fast, it is common to maintain tables which store the interrupt handler addresses for each type of interrupt. When an interrupt arrives, the operating system quickly (by means of a table look-up) determines the address of the interrupt service routine and handles the interrupt.

All I/O operations occur through device controllers. For example, to read a word of data from disk, the CPU writes some control and data words to the disk controller. This information can be the block of the disk from where to read the data and the number of bytes to be read, for example. I/O can be either *synchronous* or *asynchronous*. In a *synchronous* mode, the processor simply waits for the operation to be completed by using a wait instruction or a delay loop

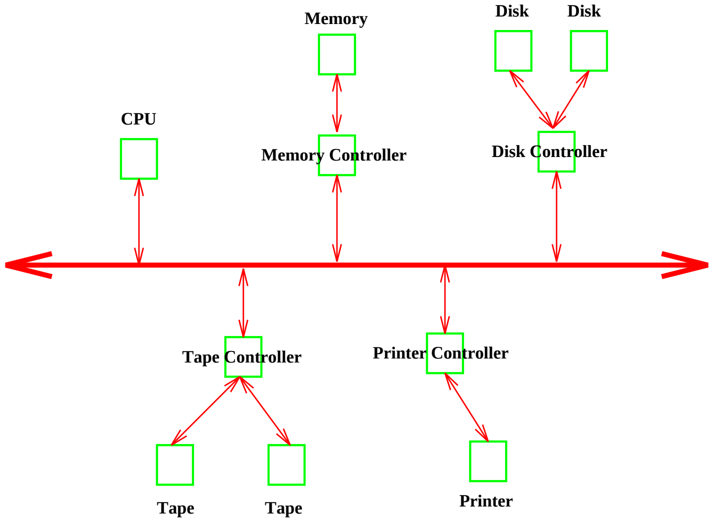

Figure 1: Typical modern architecture of a processor

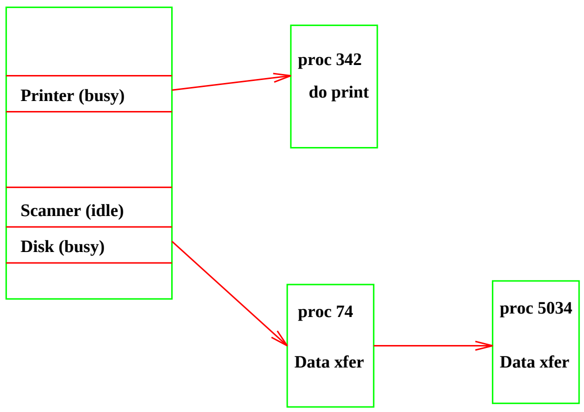

Figure 2: Device status tables in the case of asynchronous I/O

```
loop: jmp loop
```

When the I/O controller interrupts to inform the CPU that the I/O operation is completed, the *interrupt service routine* corresponding to the interrupt is executed, doing the appropriate data transfer from the controller buffers to the main memory. The operating system returns from the interrupt service routine and simply waits for another interrupt to happen. Synchronous mode has the advantage that since every I/O operation is waited till completion, there is at most one outstanding I/O operation at any time. The disadvantage is that a user process does not always need to wait for the result of an I/O operation as soon as it is initiated. By simply returning from the initiation of the I/O operation, and doing some other work before checking to see if the operation is complete allows more work to be done by the user process while the slow I/O completes. In *asynchronous* I/O, the operating system does not wait until the completion of the I/O. This means that any time, there can be several outstanding I/O operations. So, the operating system usually maintains a device status table (Figure 2) that stores information about all currently outstanding I/O operations. Each entry in the table has information about the devive as well as the (many) requests currently pending on the device. When an interrupt arrives, the operating system first has to determine the interrupting device and then has to determine which particular request to that device has succeeded. Then the particular interrupt can be serviced.

To sumamrize, the following are services typically provided by a modern day operating systems.

- Program Execution (synchronization)
- I/O (File systems, devices)
- Communication (networks, files, shared memory)
- resource allocation
- accounting
- protection

The modern day operating system is a complex beast. It is not unusual for a typical modern day unix source code to run into millions of lines of code. To make the task more manageable, an operating system is typically structured in terms of the following subsystems:

- Process Management
- Main memory management
- Secondary storage management
- I/O device management
- File system subsystems
- Protection subsystem
- networking, interrupts etc.