Computer Science 010
Introduction to Computer Applications
Siena College
Fall 2010

Topic Notes: Computer Systems

Our next topics all involve looking at *computer systems*. We talked briefly about hardware and a bit more extensively about networks. We will look in more detail at hardware, with a special focus on file storage and file systems, and look at the function of an operating system and some useful system utilities.

Hardware

Earlier in the semester, we looked at the basic parts of a computer.

If we open up a computer, we should find (and be able to identify):

- A *power supply*
- The *motherboard*, which has:
  - the *CPU (central processing unit)*
  - *memory*
  - *card slots*
  - a *battery*
  - *ROM (read-only memory)* to guide initial booting process (the *POST (power-on self test)*) and with a *BIOS (basic input/output system)* that facilitates low-level interaction between the hardware and operating system
- *storage devices* such as
  - *hard drive*
  - *optical drive*

The CPU chip will include the *ALU (arithmetic logic unit)* and *control* as we saw in our unit on von Neumann architecture.

The components are built from circuits using the *transistor* as the basic building block. We will not be concerned about the details of these circuits, but it is worth considering the number and size of the transistors.

See: `http://en.wikipedia.org/wiki/Transistor_count`

Early processors we built using thousands of transistors. Today's processors can have over a billion! They're obviously much much smaller, and today's transistors can be as small as 32 nm!

A long-standing rule of thumb, known as *Moore's Law*, states that the number of transistors that can be put on a chip will double every 18 to 24 months.

For a a long time, Moore's Law generally meant that our computers would become twice as fast within two years. For example from the early 1990's, when an Intel 80486 would be in the range of 66 MHz, to the mid 2000's, where Intel and AMD processors ran over 3 GHz, Moore's Law also predicted processor speeds (*clock rates*).

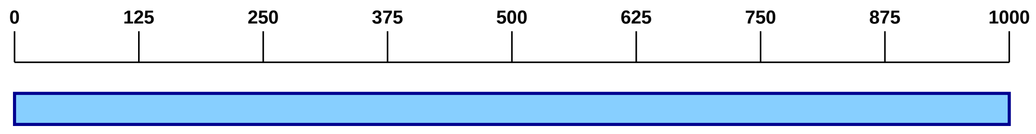
But in the mid 2000's, physical limitations were being reached on the clock rates (basically, the chips would be too hot and melt if the clock went any faster). Intel and AMD started to introduce *multicore* processor chips. This means that the chip contains more than one CPU!

The idea is that two CPUs running at a lower clock rate can do the same work as a single faster CPU running at a higher rate, but without as much heat generated.

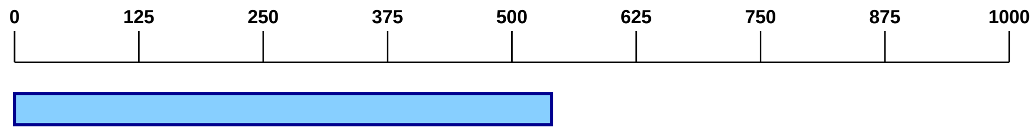
The problem is that now *parallel programming* is needed to take advantage of the processing power on your desk or in your lap.

Consider a computer user who gets a new computer every three years. For a long time, he or she would just expect all of his programs to run faster every time he got a new computer. Until multicore.

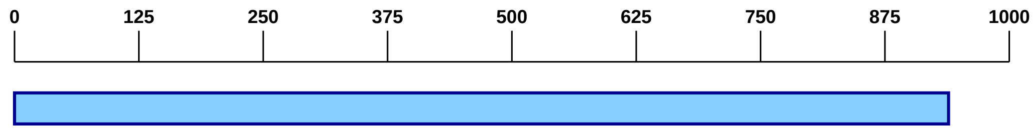
- 2001: 2 GHz Intel Pentium 4



- 2004: 3.7 GHz Intel Pentium 4



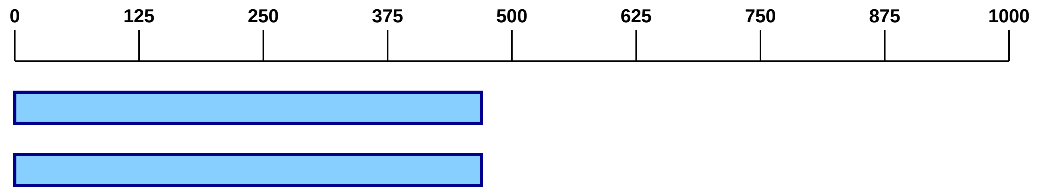
- 2007: 2.13 GHz Intel Core 2 Duo, using just one core



This poor user bought the latest and greatest 2007 model, and his program runs barely faster than it did in 2001!

He needs to be able to split the work done by his program to be able to use both processors.

- 2007: 2.13 GHz Intel Core 2 Duo, using both cores



Unfortunately, this is a very difficult thing to do in many situations. Even now, several years after the first multicore chips became available, the majority of programs can make use of only one core at a time.

Other components of a typical computer also change over time:

- Main memory
  - size: growing steadily but not keeping up with CPU advances. Typical today: 2-4 GB.
  - speed: memory speed (the amount of time it takes for the memory to retrieve or store values) has increased, but again has not kept pace with CPU advances. Typical today: about 1 GHz.
- Persistent storage: hard disks
  - incredible growth in size: early hard disks were dozens of megabytes, today's are measured in terabytes!
- Display
  - *CRT (cathode ray tube)* screens have largely been replaced by *LCD (liquid crystal display)* flat panels.
- Cost: decreasing! A new decent new PC 10-15 years ago would be $1500 or more, today probably more like $500.

Cache Memory

The growing differences in speed between the CPU and memory have introduced an important problem.

Recall that programs are stored in memory. Every time the CPU executes an instruction, it must obtain that instruction from memory. That instruction may also have operands that need to be read from or written to memory.

Consider my Mac, which has memory capable of providing data at 1 GHz, but CPUs running at around 2.5 GHz. That means that each CPU (it has two) needs at least 2.5 billion memory accesses per second, but the memory can only be accessed about 1 billion times per second.

How can the CPUs be doing any useful work the majority of the time if they're always waiting on the (slower) memory system to provide instructions and data?

The answer is: they don't. There is some special memory called *cache memory* that is much faster but also much smaller and much more expensive than main memory that is placed "in between" the CPU and main memory. It is used as a repository for recently-used data and instructions from the main memory. If we access the same instruction or data again within a short time after we accessed it the last time, it will be in the cache, and the CPU will not have to wait for the (slow) main memory.

Caches work because data accesses by computer programs naturally exhibit a property called *locality*. Two types of locality help make caches work efficiently: *spatial locality* and *temporal locality*. Spatial locality results from the fact that we are likely to access data that is nearby in memory to recently-accessed data. Temporal locality results from the fact that we are likely to access the same data repeatedly before we are finished using it. Locality means that a much larger percentage of memory access will be found in the cache than we might expect, given the fact that cache memory is only a tiny fraction of the size of main memory.

A large enough and fast enough cache is often more important than raw CPU speed in the overall performance of a computer. What good is a faster CPU if you can't keep it busy because your memory is too slow?

Other Types of Computers

Much of our discussion has been of the typical home or office desktop or notebook computer. Of course, computers are used in many other settings and it stands to reason that different situations call for different types of computers.

There are other classes of machines we should consider:

- *Mainframes*
  - May have many very fast CPUs with a capability to perform intense calculations.
  - Input/Output (I/O) *bandwidth* is very high – lots of ability to transfer data among I/O devices, memory, and the CPUs.
  - Often will have specialized I/O CPUs.
  - Used for databases, for very large volumes of work.
  - Banking, airline reservations, etc.
- *Servers*
  - May have multiple CPUs, Error-Correcting (ECC) memory, large network bandwidth.
  - Many servers may be placed in a *rack*, to form a *server farm*.
  - Often used to support Web applications.
  - May also be used as *compute farms* for computationally-intensive problems.

- *Embedded Systems* and *Controllers*
  - Found in devices ranging from washing machines to automobiles (probably have many of them in one car, in fact), to toys
  - Computer entirely on one board.
    - Simple and slow CPU, small amount of memory.
    - ROM with instructions (run just one program forever).
    - I/O connections to sensors and controls.
  - The majority of all computers in the world today fall into this class.

Persistent Storage

Recall that main memory is volatile – it loses its contents when the computer is shut off. It is also limited in size. Larger *persistent* storage devices are used for longer term storage and to hold data that is too large to fit entirely in main memory.

Persistent storage has taken many forms over the years:

- Paper: cards and tape with holes punched
- Magnetic tape
- Magnetic film on plastic (*floppy* disks)
- Magnetic film on metal (*hard* disks)
- Holes burned in metal film (CD/DVD *optical* disks)
- Quantum tunneling (*flash* memory)

Let's look a bit more closely at the last three, which are the most common today.

Magnetic Hard Disks

Hard disks have been the standard persistent storage device in most computers for many years.

We looked at the internals of a hard disk earlier in the semester, but we revisit it here:

We'll concentrate on magnetic disks (floppy disk, hard disk). A hard disk may have multiple surfaces, or *platters*. For simplicity, assume there is only one disk, or platter, involved.

A read/write head (an *armature*) is needed for each platter. The platter has a magnetic film that holds the data. Each bit of data requires a magnetic particle that can be oriented by the read/write head when the bit is written, and whose orientation can be detected by the read/write head when the data is read.

The data on a disk is arranged in concentric rings called *cylinders* or *tracks*.

Each cylinder of the disk is divided into chunks called *sectors* that contain *blocks*, the minimum allocatable and addressable unit on the disk. Since there is more space on the outside of the disk, there may be more blocks in outer cylinders than there are on inner cylinders.

The particular configuration of cylinders, sectors and the number of platters is the *drive geometry*.

So to read or write data on the disk, a cylinder and sector must be specified. The read/write head must be positioned over the desired cylinder and sector. The read/write heads are typically connected to the end of a moveable arm. This arm is moved to position the head at the correct cylinder. When the disk rotates and the desired sector reaches the read-write head, the read or write operation can proceed.

The speed of this operation depends on two major factors:

- *seek time* – the time it takes to move the read/write head to the correct cylinder
- *rotational latency* – the time it takes for the correct sector to rotate under the read/write head

Typical platter rotational speeds are 5400, 7200, or 10,000 RPM.

Access times are measured in milliseconds. Given that CPU operations are measured in picoseconds, disk access is incredibly slow by comparison.

Typical capacities are now measured in hundreds of gigabytes to a few terabytes.

Optical Data Storage

Compact Disc (CD) and Digital Video Disc (DVD) storage is also very common in modern computers.

An optical storage media like a CD or a DVD is simply a rotating plastic disc with metal film on it.

Unlike hard disks, where the data is arranged in concentric cylinders on the disk, optical storage organizes the data in a continuous spiral (like a vinyl record).

The bits in this case are stored as *flats* (or *lands*) and *holes* (or *pits*) which are burned into the metal film. A laser (780 nm wavelength for a CD) on a armature bounces off of these lands and pits to read the bits.

A CD has a data capacity of 650 to 900 MB.

A DVD uses a 650 nm laser and a Blu Ray a 405 nm laser, which means the pits can be smaller. Data capacities for a DVD are 8+ GB. A dual-layer Blu Ray data disc can hold 50 GB of data.

Access times are in the range of hundreds of milliseconds.

Flash Storage

Flash storage is a non-volatile memory typically found in small external drives and memory cards, but is sometimes used for internal storage in *solid state* drives.

This type of storage uses a *quantum tunneling* to “trap” an electronic bit of information between two insulators. This bit retains its value even when the device is not powered.

Capacities of flash storage now range from around 64 MB to 64 GB. Access times are typically in the 1 to 10 millisecond range.

An important long-term consideration when using flash memory is that it can “wear out” from repeated writing. Many flash devices are guaranteed to withstand 100,000 write cycles, some up to one million.

Files and File Systems

With any of these storage technologies, we need to have a way to organize the data on the device (we’ll focus on hard disks).

On a hard disk, the storage space is organized into blocks, usually of some fixed size, perhaps 4 KB. However, when we store data on the disk, we don’t think of it as blocks. We think of our data as *files* and *folders* (or *directories*).

So if a file is to be stored on the disk, it must be broken down into block-sized chunks, and the right number of blocks on the disk need to be allocated for use by the file.

For example, if we have a file of size 34 KB and a block size of 4 KB, we would need 9 blocks (the first 8 blocks would each store a 4 KB chunk of the file, and the 9th would store the remaining 2 KB).

These blocks must not be already in use by some other file in the system, and once they are used for this file, they cannot be used by any other files.

On a system with many thousands of files, this becomes a complicated task. This is one of many functions of the *operating system*. We will look at other functions of the operating system soon.

File System Requirements

In the particular case of file management, the operating system will need to:

- keep track of which disk blocks belong to which files,
- keep track of available disk blocks,
- allocate disk blocks to newly-created or modified files,
- write the contents of a file to that file’s disk blocks,
- read a file from the file’s disk blocks.

A file system also needs to have some organization for the files: this is usually the familiar *folder* or *directory hierarchy*. This structure is also managed by the operating system.

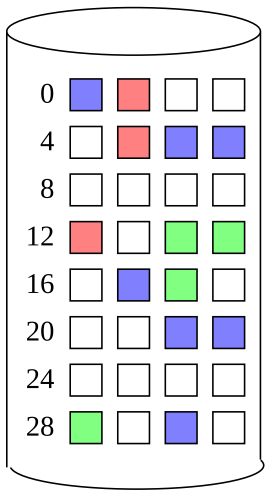
Allocating Disk Space for Files

There are many choices for directories are organized and how files are stored on a disk. We will consider just one possibility: the *File Allocation Table* or *FAT* filesystem used by many versions of Windows.

To implement a FAT filesystem, a table is created in a special location on the disk (known to the operating system). This table has one entry for each disk block (those equal-sized chunks we talked about earlier). The blocks are numbered sequentially starting at 0, so the first entry in the table is associated with disk block 0, the next entry is associated with disk block 1, and so on.

For each file, we need to know the first block of the file (more on this soon - this is where directory structures come into play). That block will contain the first part of the file. The corresponding file allocation table entry for that block will contain the block number of the next portion of the file. Then that block's table entry contains the block number of the next, and so on. Eventually, we will have visited enough blocks to store the entire file. The table entry corresponding to that last entry will contain a special "end of file" marker, so we know not to look for another block.

Here is an example with 32 disk blocks and 3 files:



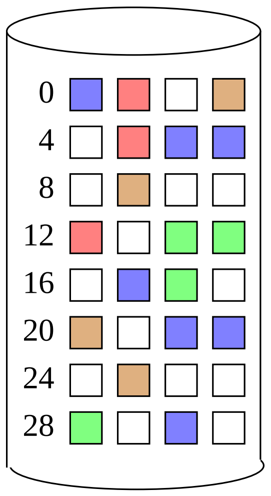
Directory

File	Start
`moo`	5
`snow`	30
`fall`	14

0	EOF	EOF		
4		12	0	17
8				
12	1		18	EOF
16		22	28	
20			6	7
24				
28	15		23	

In this case, the file `moo` starts at block 5, then uses blocks 12, and 1. `snow` uses 30, 23, 7, 17, 22, 6, and 0. Finally, `fall` uses 14, 18, 28, and 15.

If we wanted to add a new file called `dog`, that requires 4 blocks, to this disk, the operating system would locate four available disk blocks (by finding empty entries in the FAT). Those blocks would then be filled with the contents of our file, and the FAT would be updated accordingly:

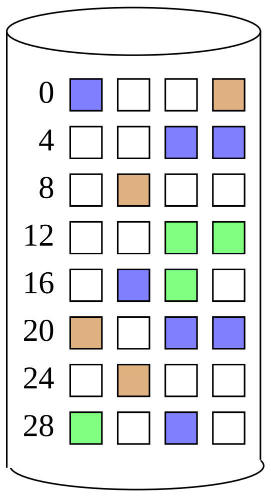


Directory

File	Start
moo	5
snow	30
fall	14
dog	9

0	EOF	EOF		EOF
4		12	0	17
8		25		
12	1		18	EOF
16		22	28	
20	3		6	7
24		20		
28	15		23	

Then if the file `moo` is deleted, the disk blocks and file allocation entries are simply made available for use by other files.



Directory

File	Start
snow	30
fall	14
dog	9

0	EOF			EOF
4			0	17
8		25		
12			18	EOF
16		22	28	
20	3		6	7
24		20		
28	15		23	

Easy enough.

Fragmentation

Next, let's remember how these disk blocks are physically layed out on the disk platters.

Now also think about what the disk drive needs to do to read the blocks of a file.

To read a block, the read/write head must be positioned over the correct cylinder and then needs to wait for the platter to spin the desired block into position under that read/write head. Then to read the next block, the read/write head will need to move to position above the next block's cylinder then wait for the platter to spin the block into position.

If the blocks that make up consecutive parts of the file are in very different places on the disk, it will take a relatively long time for the disk to be able to read that next block. However, if the next block is in the same cylinder and just happens to be the next block that will be spinning under the read/write head anyway, the drive will be all set to read it immediately.

So it makes sense when allocating blocks for a file, to allocate successive blocks in the file to adjacent blocks in the same cylinder on the disk whenever possible.

But even if the system does its best to allocate contiguous chunks of disk blocks to files (and it does), this will not always be possible. Think of the situation where we have a files of different sizes stored and deleted over time. For a while, we can always find contiguous space, but eventually all of the large spaces will be filled up. The remaining space adds up to enough for additional files, but no contiguous chunks are available. In this circumstance, the system will do the best it can, but the files will necessarily be spread across the disk.

Once this happens, we say that the files, and in fact, the whole disk, has become *fragmented*. This will slow the overall performance of the computer since the fragmented files will take longer to read and write. Generally, the closer a disk is to being full, the greater the chance of fragmentation.

To help remedy this situation when it occurs, a user (or, possibly an operating system) can run a *defragmentation* (often simply *defrag*) process to reorder the files on the disk so the files occupy contiguous blocks, and the remaining free space is in one big chunk.

Deleting Files (or NOT)

Let's consider a bit more carefully what happens when we delete a file from the a FAT file system. We said that the disk blocks are made available for use by other files and the corresponding FAT entries are marked as available. However, there is usually nothing done to the actual data in the blocks on the disk. It's still there! But the files are considered to be deleted since no directory or FAT entry is referring to that block.

For most purposes, this is not a problem at all. The data is there but since it cannot be accessed as part of a file, it is as good as deleted. And the next time some file is given those disk blocks to use, it will be gone for good.

This approach has one major disadvantage and one major advantage:

- On the positive side, this can be very useful in the case of an accidental deletion of a file. As long as no further changes are made to the filesystem after the file is deleted, we know that the data is still there even if the links to that data have been removed. System utilities can usually "undelete" a mistakenly deleted file in this case.
- But on the down side, when you delete a file it probably means you don't want it on your system any longer. It can be a security or privacy concern that someone might be able to come along after you've deleted a file and reconstruct it.

Folders/Directories

So far in our discussion of FAT filesystems, we have shown the directory as a single list of files. In real systems, we know that there is a more complex folder or directory structure.

Most modern filesystems offer a *hierarchical* structure for folders and directories. We have used this in our lab work this semester: You have your own home folder. Within that, you have a folder for work for this course. Within that, you have folders for each lab's work. Each folder can contain any combination of other folders or actual files.

It is the entries in these directories that contain a set of information including:

- the name of the file
- the information to find the file on the disk (for FAT, the block number of the block that contains the first part of the file)
- time stamps, such as file creation time, last modification time, last access time
- access controls: who is allowed to access the file

But for most of us, this structure mainly allows a user to keep his or her files organized.

Partitions and Devices

There is one more level we often see in our file hierarchy, and that is the name of the *partition* or *device*. Particularly in Windows systems, this usually takes the form of a *drive letter*.

Usually, these refer to physical devices in the system and appear at the start of the *path* to any file.

For example, you might have a file:

```
Z:\CSIS010\Lab5\Lab5-Workbook-Solution.xlsm
```

Here, the device or partition is `Z:\`, `CSIS010` and `Lab5` are folder names that we follow from the "top" of the `Z` drive, and `Lab5-Workbook-Solution.xlsm` is the name of the file contained in that `Lab5` folder.

Each drive letter on a system is typically associated with a particular physical device. A few of the letters have very common uses (or at least formerly common uses):

Drive Letter	Typical Usage
`A`	3.5" floppy disk
`B`	5.25" floppy disk
`C`	Primary Internal Hard Disk
`D` and beyond	Additional Hard Disks, CD/DVD Drives, Network Drives

In our lab, we also have a *network* drive assigned to the letter `Z`. Each time you log into one of the systems, it automatically assigns the `Z:\` path to your home folder.

Operating Systems

We have just looked at one of the main functions of an operating system: managing disk devices and making them look like the files and folders we wish to see as users.

Disks are just one of many *system resources* that are managed by an operating system. The operating system controls which users (or more accurately, which processes) in the system get to access these resources.

These resources include

- CPU(s)
- Main memory
- I/O devices like the keyboard, mouse, monitor, microphone, camera, speakers
- Storage devices (fixed and removable)

The operating system performs many functions, but we will gather its functions into these five categories:

1. System start up (booting)
2. Managing memory and CPU usage for executing processes
3. Managing devices
4. Providing user interfaces
5. Managing file systems (which we already considered)

Booting

Let's consider a bit more carefully what needs to happen when a computer starts up ("boots").

At that point, our main memory (the *DRAM – dynamic random access memory*) is empty – whatever was there the last time the computer was on has been lost.

But a CPU only knows how to get instructions from memory.

The startup instructions are located in a special bank of *ROM*, or *read-only memory*. The CPU will start by executing instructions in this special block of non-volatile memory.

The block of memory is often called the *BIOS* (*Basic Input/Output System*).

The booting sequence in the BIOS gets the CPU going, and figures out how to load the operating system kernel. The *kernel* is the core piece of the operating system that is in control of all of the other functions of the system. The kernel is really just a program sitting on some persistent storage device. It is up to the BIOS to select a device from which to load an operating system kernel.

Many BIOS programs allow different devices to be selected when you start up. If you don't specify anything, it will likely search on the internal hard disk. Other possibilities are to look for an operating system kernel on a floppy disk, a CD or DVD, on an external device like a removable hard drive or flash drive, or somewhere on the network.

However, the BIOS generally does not have detailed information about the location of files, in particular, the operating system kernel, on the hard disk. But it does know where to look. Each disk will have a *Master Boot Record (MBR)* that is essentially a map to tell the BIOS where to look for the kernel to get things started.

The above describes the process from a powered-down state (a *cold boot*). We might also sometimes "restart" the computer without turning off the power (a *warm boot*). For a warm boot, some information may still be in memory, so a faster startup process can sometimes be used.

CPU and Memory Management

Once booted, the operating system's kernel is responsible for allocating the CPU and memory resources of the computer to the processes that are running on the system. The kernel itself does no useful work in the sense that it is not computing answers to problems or composing documents or browsing the web. Other programs (*applications* and *utilities*) allow us to perform those specific functions.

Modern operating systems will allow many such programs to be running at the same time. Think about your own usage - you don't need to shut down your web browser to start Excel or Word. You can switch back and forth between them. The operating system allows this to happen.

First, think about the CPU. Assume for the moment that our computer has just one CPU. That CPU can only be doing one thing at a time. But we're going to ask it to perform operating system kernel functions as well as to run several programs, seemingly "at the same time".

It turns out that the programs are not really running at the same time. The operating system allows each program that needs to execute some instructions a small amount of time to have exclusive access to the CPU. This is the process' *time slice* or *quantum* on the CPU, and typically is around $\frac{1}{10}$ second. After that time is up, the operating system kernel removes that process from the CPU and gives access to another process. Fortunately, this happens quickly enough that as users, we usually do not notice the fact that some of our programs are sitting there waiting their turn.

This would become a problem if there are too many processes trying to execute in the system at the same time. The time between each process getting a "turn" on the CPU depends on the number of processes competing for access to the CPU. Too many processes would mean each gets a smaller fraction of the CPU and will each run slower.

If we are going to have many processes in the system at the same time, each of those processes will also need some main memory. Remember, the instructions to be executed and the data on which they operate will be located in main memory.

Each process will need its own private chunk of memory with which to work. So the operating system needs to allocate chunks of the memory among the executing processes. But that's not all - we need to make sure that each process is only allowed to access its own allocated chunk of memory. It would be a big problem if one program could unexpectedly change values in memory being used by another. It would be very disturbing if by clicking on a button in your web browser that the document you are editing in Word changes somehow! So the operating system need to *allocate* memory to processes and provide a *protection* scheme to make sure a process can only access the memory that has been allocated to it.

Operating System Kernel

We mentioned the idea of the operating system kernel - the program that is in charge of managing resources and running all other programs.

The kernel also includes the software that performs many other functions, such as

- Networking (TCP/IP software)
- Login/logout of users
- Interact with user processes

This interaction between user processes and the kernel is what allows regular (non-kernel) processes to do things that would normally only be done by the kernel (allocating memory, accessing devices, accessing the network). The user processes will interact with *system calls* – special interfaces that allow a controlled access of kernel resources by user programs.

Hardware Devices

The applications we run will need access to a variety of devices such as mice, keyboards, printers, scanners, sound cards, *etc.*. These programs simply cannot be aware of all possible such devices and the specifics of how to interact with them correctly.

The operating system also needs to determine which application program gets to interact with the devices at any given time. If I type on the keyboard, or click the mouse, which program, if any, should receive that input?

Another function of the operating system is to provide a set of standard *interfaces* to these devices – some way of talking to, for example, printers, that works with any printer. Then the operating system translates the generic commands (*e.g.*“print pages 1 and 2”) into the exact commands that the specific brand and model of printer expects.

But, the operating system also cannot possibly know the details of every device that might exist on every computer. In fact, some devices may not even exist at all when the operating system is created.

Device Drivers

This problem is addressed with *device drivers*, which are small programs provided by the *hardware vendors* (*i.e.*, the manufacturer) that essentially extend the operating system to allow it to interact with the device, acting as an intermediary between the device and the user programs that will access it.

In addition to providing the operating system the specifics of the device that it might not otherwise have, it allows the system to load only those drivers for devices in use by the computer.

Many common modern devices are *plug and play* devices that can use device drivers that are a standard part of the operating system. These devices usually can be “plugged in” and then you can “play with them” right away without having to install any drivers or do any configuration.

If not, you may see the “Windows has detected new hardware” and you may be prompted to find the appropriate device driver. Ideally, the driver was packaged with a CD containing that driver. Drivers may be available from Internet sites as well, but beware – if you install a malicious device driver, you have given that driver access to your operating system kernel. And once it has that, it has anything it wants.

A bad or misconfigured device driver can interfere with your operating system’s ability to function correctly or even to function at all. It can be difficult or impossible to correct the situation while the system is running the bad driver.

Windows provides a *Safe Mode* boot option that installs only the most basic and known-to-be-safe device drivers in the kernel. This usually will allow the computer to be booted and the bad driver to be identified and removed.

See the drivers in the Windows Device Manager (Start/Control Panel/System/Hardware).

User Interfaces

It turns out that these system call interfaces to the kernel are very much **not** user friendly. In fact, they are usually incredibly cryptic and difficult to understand. For this reason, the operating system will also provide an easier-to-use *user interface*.

The user interface will translate between you, the user, and the kernel.

The simplest and oldest user interfaces were (and still are) text based and are often referred to as *shells*.

We opened the Windows command shell during our discussion of networking. This window allows us to type commands (sort of a direct telephone line to the kernel) and have the operating system execute them on our behalf and report back the results.

However, most users today rarely if ever interact with a command shell. Modern computers provide *graphical user interfaces (GUIs)* – the windowing systems we all know and use on a regular basis. Where in the command line you would type the name of a program or file to access it, with a graphical user interface, we would click on *icons* in *windows*, using a *pointer* (usually controlled by a mouse or trackpad). Our files are displayed organized into *folders*. Most programs include *menus* that can also be selected with the pointer device to interact with the programs.

Underneath the hood, the effect is the same. Double-clicking on a program’s icon in the GUI translates to the exact same system calls as does typing the name at the command shell, but it much more intuitive for most users.

GUIs provide a set of window operations that help users interact with multiple programs:

- minimize a window (put it into the taskbar)
- maximize a window (make it fill the screen)
- drag windows to move
- drag window corners to resize
- select a window that is *in focus* – the one that will receive the input from the mouse and keyboard

Modern operating systems also provide common tools that all applications may use to perform similar operations.

Perhaps the best example is for *file selection*. The majority of applications running on modern computer systems will deal with files on the file system.

- copy, rename, delete, save, open, *etc.*.

You probably have noticed that most Windows applications have a similar-looking file dialog, even those written by different companies.

Try it... open a variety of programs and choose File..Open... and you should seem very similar dialogs presented.

That's because Windows provides an *Application Programming Interface (API)* that all applications can use when they need to open or save or print a file. Similarly, Mac OS X and other systems will provide such interfaces.

This has advantages for both software developers and software users:

- Software developers do not have to "reinvent the wheel" for common operations. If the system already provides a file open dialog, no need to write one. It also means that the application can access files on all types of devices that the operating system knows about. The individual application does not need to know or care if the file chosen is on a local hard disk, a CDROM, or a network drive.
- This also provides a *look and feel* that is consistent across all applications that run under that system. Software users will have a built-in familiarity with new programs – they'll have a File...Open menu option and it will bring up a file selection dialog that looks like the ones they use in other programs.

- Since these APIs are implemented by the operating system kernel, they can be updated when the operating system is updated. A program that was written before Windows 7 existed (maybe it was developed during the Windows XP era) will gain a Windows 7 look and feel as soon as it is run on Windows 7.

Other commonly-provided APIs include:

- a Print dialog that selects among available printers and other print setting.
- a Properties dialog that will bring up appropriate user-modifiable settings for that program.

Key Ideas of Operating Systems

Overall, the operating system is there to make it **easier** and **safer** to use the computer.

- easier by hiding complex details
- safer by ensuring that programs can only interact with the hardware and with each other in prescribed ways

It acts as a “benevolent dictator” as it manages all of the resources of the computer.

Common Desktop/Laptop OSs Today

There are three primary categories of operating systems in widespread usage today:

1. Microsoft Windows
   - runs on PCs manufactured by anyone (including individuals) with Intel processors
   - Historically, started as DOS (the command line), then Windows 3.1 $\longrightarrow$ 95 $\longrightarrow$ 98 $\longrightarrow$ ME $\longrightarrow$ NT $\longrightarrow$ 2K $\longrightarrow$ XP $\longrightarrow$ Vista $\longrightarrow$ 7
2. Apple Mac OS X
   - runs only on Apple-manufactured hardware, now also with Intel processors
   - Historically, first GUI-based OS to gain wide usage, modern versions are built on top of a Unix foundation
3. Linux/FreeBSD
   - Based on Unix, which was and is the primary operating system for many business, academic, and research computer systems
   - Run on a wide variety of modern and historical hardware

- Usually *free and open source software (FOSS)* – it's really free
- Many *distributions* available for Linux, each with a different target audience
- Many GUIs available, and are highly configurable

All of these perform similar functions, as we discussed here.

We will have more to say later about free and open source software both in terms of operating systems and other programs.

System Utilities

Many operating system functions are packaged into convenient *system utilities*. Some of these would be provided as part of the operating system, while others might be tools installed separately.

Some common examples:

- Operating system-provided
  - Add/remove programs
  - search for files
  - disk defragmenter
  - disk cleanup
  - task manager
  - command shell
- User-installed
  - WinZip – package and compress files
  - backup/restore tools