G53OPS Operating Systems

Handout Introduction

These notes were originally written using (Tanenbaum, 1992) but later editions (Tanenbaum, 2001; 2008) contain the same information.

What is an Operating System?

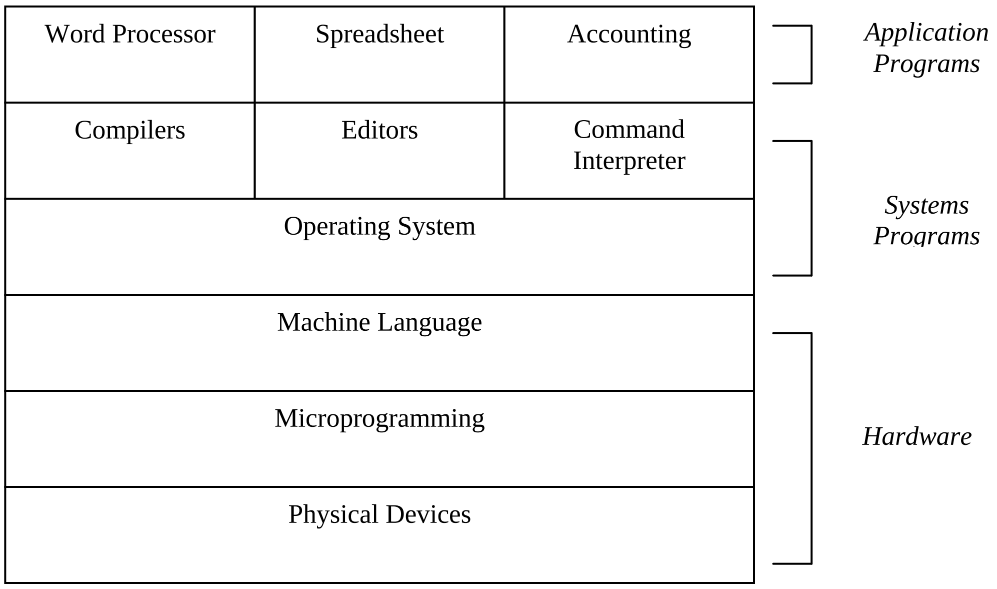
Introduction

Before we look at what an operating system ***does*** we ought to know what an operating system ***is***.

If we just build a computer, using physical components, we end up with a lot of assembled metal, plastic and silicon. In this state the computer can do nothing useful. To turn it into one of the most versatile tools known to man we need software. We need applications that allow us to write letters, develop (other) programs and calculate cash flow forecasts.

But, if all we have are the applications, then each programmer has to deal with the complexities of the hardware. If a program requires data from a disc, the programmer would need to know how *every* type of disc worked and then be able to program at a low level in order to extract the data. In addition, the programmer would have to deal with all the error conditions that could arise.
As an example, it is a lot easier for a programmer to say READ NEXT RECORD than have to worry about moving the read/write heads, waiting for the correct sector to come around and then reading the data (and this is very simplified view of what actually happens).

It was clear, from an early stage in the development of computers, that there needed to be a layer of software that sat between the hardware and the software.
This had led to this view of a computer.



The bottom layer of the hardware consists of the integrated circuits, the cathode ray tubes, the wire and everything else that electrical engineers use to build physical devices.

The next layer of hardware (microprogramming) is actually software (sometimes called firmware) that provides basic operations that allow communication with the physical devices. This software is normally in Read Only Memory (and that is why it is called firmware).
The machine language layer defines the instruction set that are available to the computer. Again, this layer is really software but it is considered as hardware as the machine language will be part and parcel of the hardware package supplied by the hardware supplier.
The operating system is a layer between this hardware and the software that we use. It allows us (as programmers) to use the hardware in a user friendly way. Furthermore, it allows a layer of abstraction from the hardware. For example, we can issue a print command without having to worry about what printer is physically connected to the computer. In fact, we can even have different operating systems running on the same hardware (e.g. DOS, Windows and UNIX) so that we can use the hardware using an operating system that suits us.

On top of the operating system we have the system software. This is the software that allows us to do useful things with the computer, but does not directly allow us to use the computer for anything that is useful in the real world (this is a broad statement which we could argue about but system software really only allows us to use the computer more effectively).

It is important to realise that system software is ***not*** part of the operating system. However, much system software is supplied by the computer manufacturer that supplied the operating system. But, system software can be written by programmers and many large companies have teams of programmers (often called system programmers) who's main aim is to make the operating system easier to use for other people (typically programmers) in the organisation.
The main difference between the operating system and system software is that the operating system runs in ***kernel*** or ***supervisor*** mode (system software and applications run in ***user mode***). This means that the hardware stops the user programs directly accessing the hardware. And, due to this restriction, you cannot, for example, write your own disc interrupt handler and replace the one in the operating system. However, you can write your own (say) command shell and replace the one supplied with the computer.

Finally, at the top level we have the application programs that, at last, allow us to do something useful.

Two views of an operating system

In the section above, we viewed the operating system in the context of how it fits in within the overall structure of the computer. Now we are going to look at another two views of an operating system.

One view considers the operating system as a ***resource manager***. In this view the operating system is seen as a way of providing the users of the computer with the resources they need at any given time.
Some of these resource requests may not be able to be met (memory, CPU usage etc.) but, as we shall see later in the course, the operating system is able to deal with scheduling problems such as these.
Other resources have a layer of abstraction placed between them and the physical resource. An example, of this is a printer. If your program wants to write to a printer, in this day and age, it is unlikely that the program will be directly connected to a physical printer. The operating system will step in and take the print requests and spool the data to disc. It will then schedule the prints, making the best use of the printer as it can. During all of this it will appear to the user and the program as if their prints requests are going to a physical printer.

Another view of an operating system sees it as a way of not having to deal with the complexity of the hardware. In (Tanenbaum, 1992) the example is given of a floppy disc controller (using an NEC PDP765 controller chip).
This chip has sixteen commands which allow the programmer to read and write data, move the disc heads, format tracks etc. Just carrying out a simple READ or WRITE command requires thirteen parameters, which are packed into nine bytes. The operation, when complete, returns 23 status and error fields packed into seven bytes.

And, if you think that is complicated, we have barely scratched the surface (for example, concerning ourselves with if the floppy disc is spinning, what type of recording method we should use and the fact that hard discs are just as complicated but work differently).
It is all these complexities that the operating system hides and in this view of the machine the operating system can be seen as an ***extended machine*** or a ***virtual machine***.

History of Operating Systems

In this section we take a brief look at the history of operating systems – which is almost the same as looking at the history of computers.
If you are particularly interested in the history of computers you might like to read (Levy, 1994). Although the title of the book suggests activities of an illegal nature, in fact, hacking, used to refer to people who had intimate knowledge of computers and were addicted to using computers and extending their knowledge.

You are probably aware that Charles Babbage is attributed with designing the first digital computer, which he called the ***Analytical Engine***. It is a matter of regret that he never managed to build the computer as, being of a mechanical design, the technology of the day could not produce the components to the needed precision. Of course, Babbage's machine did not have an operating system.

First Generation (1945-1955)

Like many developments, the first digital computer was developed due to the motivation of war. During the second world war many people were developing automatic calculating machines. For example

- By 1941 a German engineer (Konrad Zuse) had developed a computer (the Z3) that designed airplanes and missiles.
- In 1943, the British had built a code breaking computer called Colossus which decoded German messages (in fact, Colossus only had a limited effect on the development of computers as it was not a general purpose computer – it could only break codes – and its existence was kept secret until long after the war ended).
- By 1944, Howard H. Aiken, an engineer with IBM, had built an all-electronic calculator that created ballistic charts for the US Navy. This computer contained about 500 miles of wiring and was about half as long as a football field. Called The Harvard IBM Automatic Sequence Controlled Calculator (Mark I, for short) it took between three and five seconds to do a calculation and was inflexible as the sequence of calculations could not change. But it could carry out basic arithmetic as well as more complex equations.
- ENIAC (Electronic Numerical Integrator and Computer) was developed by John Presper Eckert and John Mauchly. It consisted of 18,000 vacuum tubes, 70,000 soldered resisters and five million soldered joints. It consumed so much electricity (160kw) that an entire section of Philadelphia had their lights dim whilst it was running. ENIAC was a general purpose computer that ran about 1000 faster than the Mark I.
- In 1945 John von Neumann designed the Electronic Discrete Variable Automatic Computer (EDVAC) which had a memory which held a program as well as data. In addition the CPU, allowed all computer functions to be coordinated through a single source. The UNIVAC I (Universal Automatic Computer), built by Remington Rand in 1951 was one of the first commercial computers to make use of these advances.

These first computers filled entire rooms with thousands of vacuum tubes. Like the analytical engine they did not have an operating system, they did not even have programming languages and programmers had to physically wire the computer to carry out their intended instructions. The programmers also had to book time on the computer as a programmer had to have dedicated use of the machine.

Second Generation (1955-1965)

Vacuum tubes proved very unreliable and a programmer, wishing to run his program, could quite easily spend all his/her time searching for and replacing tubes that had blown. The mid fifties saw the development of the transistor which, as well as being smaller than vacuum tubes, were much more reliable.

It now became feasible to manufacture computers that could be sold to customers willing to part with their money. Of course, the only people who could afford computers were large organisations who needed large air conditioned rooms in which to place them.
Now, instead of programmers booking time on the machine, the computers were under the control of computer operators. Programs were submitted on punched cards that were placed onto a magnetic tape. This tape was given to the operators who ran the job through the computer and delivered the output to the expectant programmer.

As computers were so expensive methods were developed that allowed the computer to be as productive as possible. One method of doing this (which is still in use today) is the concept of ***batch jobs***. Instead of submitting one job at a time, many jobs were placed onto a single tape and these were processed one after another by the computer. The ability to do this can be seen as the first real operating system (although, as we said above, depending on your view of an operating system, much of the complexity of the hardware had been abstracted away by this time).

Third Generation (1965-1980)

The third generation of computers is characterised by the use of Integrated Circuits as a replacement for transistors. This allowed computer manufacturers to build systems that users could upgrade as necessary. IBM, at this time introduced its System/360 range and ICL introduced its 1900 range (this would later be updated to the 2900 range, the 3900 range and the SX range, which is still in use today).

Up until this time, computers were single tasking. The third generation saw the start of ***multiprogramming***. That is, the computer could *give the illusion* of running more than one task at a time. Being able to do this allowed the CPU to be used much more effectively. When one job had to wait for an I/O request, another program could use the CPU.
The concept of multiprogramming led to a need for a more complex operating system. One was now needed that could schedule tasks and deal with all the problems that this brings (which we will be looking at in some detail later in the course).
In implementing multiprogramming, the system was confined by the amount of physical memory that was available (unlike today where we have the concept of virtual memory).

Another feature of third generation machines was that they implemented ***spooling***. This allowed reading of punch cards onto disc as soon as they were brought into the computer room. This eliminated the need to store the jobs on tape, with all the problems this brings.
Similarly, the output from jobs could also be stored to disc, thus allowing programs that produced output to run at the speed of the disc, and not the printer.

Although, compared to first and second generation machines, third generation machines were far superior but they did have a downside. Up until this point programmers were used to giving their job to an operator (in the case of second generation machines) and watching it run (often through the computer room door – which the operator kept closed but allowed the programmers to press their nose up against the glass). The turnaround of the jobs was fairly fast.
Now, this changed. With the introduction of batch processing the turnaround could be hours if not days.
This problem led to the concept of ***time sharing***. This allowed programmers to access the computer from a terminal and work in an interactive manner.

Obviously, with the advent of multiprogramming, spooling and time sharing, operating systems had to become a lot more complex in order to deal with all these issues.

Fourth Generation (1980-present)

The late seventies saw the development of Large Scale Integration (LSI). This led directly to the development of the personal computer (PC). These computers were (originally) designed to be single user, highly interactive and provide graphics capability.

One of the requirements for the original PC produced by IBM was an operating system and, in what is probably regarded as the deal of the century, Bill Gates supplied MS-DOS on which he built his fortune.
In addition, mainly on non-Intel processors, the UNIX operating system was being used.

It is still (largely) true today that there are mainframe operating systems (such as VME which runs on ICL mainframes) and PC operating systems (such as MS-Windows and UNIX), although the edges are starting to blur. For example, you can run a version of UNIX on ICL's mainframes and, similarly, ICL were planning to make a version of VME that could be run on a PC.

Fifth Generation (Sometime in the future)

If you look through the descriptions of the computer generations you will notice that each have been influenced by new hardware that was developed (vacuum tubes, transistors, integrated circuits and LSI).
The fifth generation of computers may be the first that breaks with this tradition and the advances in software will be as important as advances in hardware.
One view of what will define a fifth generation computer is one that is able to interact with humans in a way that is natural to us. No longer will we use mice and keyboards but we will be able to talk to computers in the same way that we communicate with each other. In addition, we will be able to talk in any language and the computer will have the ability to convert to any other language.
Computers will also be able to reason in a way that imitates humans.

Just being able to accept (and understand!) the spoken word and carry out reasoning on that data requires many things to come together before we have a fifth generation computer. For example, advances need to be made in AI (Artificial Intelligence) so that the computer can mimic human reasoning. It is also likely that computers will need to be more powerful. Maybe parallel processing will be required. Maybe a computer based on a non-silicon substance may be needed to fulfill that requirement (as silicon has a theoretical limit as to how fast it can go).

This is one view of what will make a fifth generation computer. At the moment, as we do not have any, it is difficult to provide a reliable definition.

Another View

The view of how computers have developed, with regard to where the generation gaps lie, is slightly different, depending who you ask. Ask somebody else and they might agree with the slightly amended model below.

Most commentators agree on what is the first generation and the fact they are characterised by the fact that they were developed during the war and used vacuum tubes.
Similarly, most people agree that the transistor heralded the second generation.
And, the third generation came about because of the development of the IC and operating systems that allowed multiprogramming. But, in the model above, we stated that the third generation ran from 1965 to 1980.
Some people would argue that the fourth generation actually started in 1971 with the introduction of LSI, then VLSI (Very Large Scale Integration) and then ULSI (Ultra Large Scale Integration).
Really, all we are arguing about is when the PC revolution started. Was it in the early 70's when LSI first became available? Or was it in 1980, when the IBM PC was launched?

Case Study

To show, via an example, how an operating system developed, we give a brief history of ICL's mainframe operating systems.

One of ICL's first operating systems was known as ***manual exec*** (short for executive). It ran on its 1900 mainframe computers and provided a level of abstraction between the hardware and also allowed multi-programming.

However, it was very much a manual operating system. The operators had to load and run each program. Commands such as these were used.

```
LO#RA15#REP3
GO#RA15 21
```

The first instruction told the computer to load the program called RA15 from a program library called REP3. This loaded the program from disc into memory.
The “GO 21” instruction told the program to start running, using entry point 21. This (typically) told the program to read a punched card(s) from the card reader, which held information to control the program.

The important point is that the computer operator had control over every program in the computer. It had to be manually loaded into memory, initiated and finally deleted from the memory of the computer (which was typically 32K). In between, any prompts had to be dealt with. This might mean allowing the computer to use tape decks, allowing the program to print special stationery or dealing with events that were unusual.

ICL then brought out an operating system they called GEORGE (***GE***neral ***ORG***anisational ***E***nvironment). The first version was called George 1 (G1). G2 and G2+ quickly followed.
The idea behind G1/2/2+ was that it ran on top of the operating system. So it was not an operating system as such (in the same way that Windows 3.1 is not a true operating system as it is only a GUI that runs on top of DOS).
What G2+ (we’ll ignore the previous versions for now) allowed you to do was submit jobs to the machine and then G2+ would schedule those jobs and process them accordingly. Some of the features of G2+ included.

- It allowed you to batch many programs into a single job. For example, you could run a program that extracted data from a masterfile, run a sort and then run a print program to print the results. Under manual exec you would need to run each program manually.
  It was not unusual to have a typical job process twenty or thirty separate programs.
- You could write parameterised macros (or JCL – Job Control Language) so that you could automate tasks.
  For example, you could capture prompts that would normally be sent to the operator and have the macro answer those prompts.
- You could provide parameters at the time you submitted the job so that the jobs could run without user intervention.
- You could submit many jobs at the same time so that G2+ would run them one after another.
- You could adjust the scheduling algorithm (via the operators console) so that an important job could be run next – rather than waiting for all the jobs in the input queue to complete.
- You could inform G2+ of the requirements of each job so that it would not run (say) two jobs which both required four tape decks when the computer only had six tape decks.

Under G2+, the operators still looked after individual jobs (albeit, they now consisted of several programs).
When ICL released George 3 (G3) and later G4, all this changed. The operators no longer looked after individual jobs. Instead they looked after the system as a whole.
Jobs could now be submitted via interactive terminals. Whereas the operators used to submit the jobs, this role was now typically carried out by a dedicated scheduling team who would set up the workload that had to be run over night, and would set up dependencies between the jobs.
In addition, development staff would be able to issue their own batch jobs and also run jobs in an interactive environment.
If there were any problems with any of the jobs, the output would either go to the development staff or to the technical support staff where the problem would be resolved and the job resubmitted.

Operators, under this type of operating system were, in some peoples opinion, little more than “tape monkeys”, although the amount of technical knowledge held by the operators varied greatly from site to site.

In addition to G3 being an operating system in its own right G3 also had the following features

- To use the machine you had to run the job in a *user*. This is a widely used concept today but was not a requirement of G2+.
- The Job Control Language (JCL) was much more extensive than that of G2+.
- It allowed interactive sessions
- It had a concept of ***filestore***. When you created a file you had no idea where it was stored. G3 simply placed it in filestore. This was a vast amount of disc space used to store files. In fact the filestore was virtual in that some of it was on tape. What files were placed on tape was controlled by G3. For example, you could set the parameters so that files over a certain size or files that had not been used for a certain length of time were more likely to be placed onto tape. If your job requested a file that was in filestore but had been copied to tape the operator would be asked to load that tape. The operator had no idea what file was being requested or who it was for (although they could find out). G3 simply asked for a TSN (Tape Serial Number) to be loaded.
- The operators ran the system, rather than individual jobs.

After G3/G4, ICL released their VME (Virtual Machine Environment) operating system. This is still the operating system used on ICL mainframes today.
VME, as its name suggests, creates virtual machines that jobs run in. If you log onto (or run a job on) VME, you will be created a virtual machine for your session.
In addition, VME is written to cater for the many different workloads that mainframes have to perform. It supports databases (using ICL's DBMS – Database Management System and, more recently relational databases such as Ingress and Oracle), TP (Transaction Processing) systems as well as batch and interactive working.
The job control language which, under VME is called SCL – System Control Language is a lot more sophisticated and you can often carry out tasks without having to use another language for operations such as file I/O.

There is still the concept of filestore but, due the to the (relatively) low cost of disc space and the problems associated with having to wait for tapes, all filestore is now on disc. In addition, the amount of filestore available to users or group of users is under the control of the operating system (and thus the technical support teams).
Like G3, the operators control the entire system and are not normally concerned with individual jobs. In fact, there is a move towards having ***lights out*** working. This removes the need for operators entirely and if there are any problems, VME, will telephone a pager.

Operating System Concepts

As you might expect we use an operating system (in terms of using the OS within a program) via a well defined interface. That is, we make calls to the operating system so that it will perform some task for us. These "entries" into the OS are called ***system calls***. Through these system calls we can manipulate various types of objects. During this course we will be looking at some of these objects in detail (e.g. processes) but we briefly introduce them here.

Processes

One of the key tasks for an operating system is to run ***processes***. For now, we can consider a process as a running program with all the other information that is needed to control its execution (e.g. program counter, stack, registers, file pointers etc.).
If the CPU is running one process and, for whatever reason, another process now needs to run, the operating system must save the details of the currently running process and start the new process from ***exactly*** the same point as it was left.
All the data for a process is normally held in a ***process table***. This is a data structure that contains a list of active processes which the operating system uses to decide which process to run next and to restore a process to the state it was in before it was stopped from running.

A process may create a ***child process***. For example, if we issue a command from the command shell (one process), this will create another process to execute the command. This can lead to a tree like structure of processes. When that command finishes the child process will issue a system call to destroy itself.

One of the main tasks of an operating system is to schedule all the processes which are currently competing for the CPU.

Processes may also communicate with other processes. This might sound simple but, as we shall see, later in the course, this leads to all sorts of complications which the operating system must handle.

Files

Another broad class of system calls relate to the file system.
We said above that one of the tasks of an operating system is to hide the complexities of the hardware. In order to do this system calls must be provided to (for example) create files, delete files, move files, rename files, copy files, open files, close files, read files, write files etc. etc.

As an example of this abstraction, consider that some operating systems provide you with different types of file. You may be able to open a file in ***text*** mode or in ***binary*** mode. The way you open the file determines how the data is delivered to your program. In fact, the underlying mechanisms which position the read/write head, access the data and return it to the operating system are the same no matter what type of file you are reading. It is only when it is delivered to your program does the data get interpreted in the way the program is expecting.
To take this one stage further, many operating systems provide a special file called ***standard input*** and ***standard output***. These default to data typed at the terminal and data sent to the terminal. We consider them as files but there is obviously a lot more going on in the operating system that allows us to visualise a terminal as a file.

There is even the concept of a process being an input file for another process, which acts as if it is writing to a file. This is normally presented to us as a ***pipe***. For example, in MS-DOS if you type

```
DIR | SORT
```

It pipes the output from the DIR command to the SORT command, so that the display comes out sorted (in fact, it sorts the lines in alphabetical order – and might not be what you expect).

Similar to pipes is ***redirection***. This allows the output from a program to be redirected to a file. For example

```
DIR > dir.txt
```

Will redirect the output of the DIR command from the standard output (the screen) to the file called dir.txt.
(If you have never used MS-DOS (or UNIX) you might like to experiment with redirection – also try “>>” and “<”).

The whole point of mentioning standard input, standard output, pipes and redirection is to demonstrate that the operating system is hiding a lot of complexity from us as well as providing us with many features which we might find useful.

Many file systems also support the concept of directory (or folder) hierarchies. That is, there is a top level view of the file system and by creating folders you can build a tree (conceptually) which represents your view of the data. Therefore, the file system must provide system calls to maintain these directory structures.

When discussing processes we said that it is possible to build a tree of processes, in the same way we can build a directory hierarchy. But there are many differences between these trees.
- Process trees are normally short lived. Directory trees can last for years.

- Files (assuming access rights are granted) can be maintained by any user of the system. A process can only be controlled by its parent.
- Directories (and files) can have access rights associated with them. Processes have no such information.
- Directories (and files) can be accessed in a number of ways (e.g. relative or absolute pathnames). Process tress have no such concept.

System Calls

Access to the operating system is done through system calls. Each system call has a procedure associated with it so that calls to the operating system can be done in a familiar way.
When you call one of these procedures it places the parameters into registers and informs the operating system that there is work to be done. Notifying the operating system is done via a TRAP instruction (sometimes known as a ***kernel call*** or a ***supervisor call***). This instruction switches the CPU from ***user mode*** to ***kernel*** (or ***supervisor***) mode. In user mode certain instructions are unavailable to the programs. In kernel mode all the CPU instructions are available.
The operating system now carries out the work, which includes validating the parameters. Eventually, the operating system will have completed the work and will return a result in the same way as a user written function written in a high level language.

An example of an operating system call (via a procedure) is

```
count = read(file, buffer, nbytes);
```

You can see that it is just the same as calling a user written function in a language such as C.

The Shell

The operating system is the mechanism that carries out the system calls requested by the various parts of the system. Tools such as compilers, editors etc. are not part of the operating.
Similarly, the ***shell*** is not part of the operating system. The shell is the part of (for example) UNIX and MS-DOS where you can type commands to the operating system and receive a response. You may also hear the shell get called the ***Command Line Interpreter*** (CLI) or the "C" prompt. However, it is worth mentioning the shell as it makes heavy use of operating system features and is a good way to experiment.

We have already seen one example of a command line (`DIR > dir.txt`).

A more complicated command (in UNIX this time) could be

```
cat file1 file2 file3 | sort > /dev/lp &
```

This command concatenates three files and pipes them to the sort program. It then redirects the sorted file to a line printer.
The ampersand at the end of the command instructs UNIX to issue the command as a background job. This results in the command prompt being returned immediately, whilst another process carries out the requested work.

You can appreciate, by looking at the above command that there will be a series of system calls to the operating system in order to satisfy the whole request.

Operating System Structure

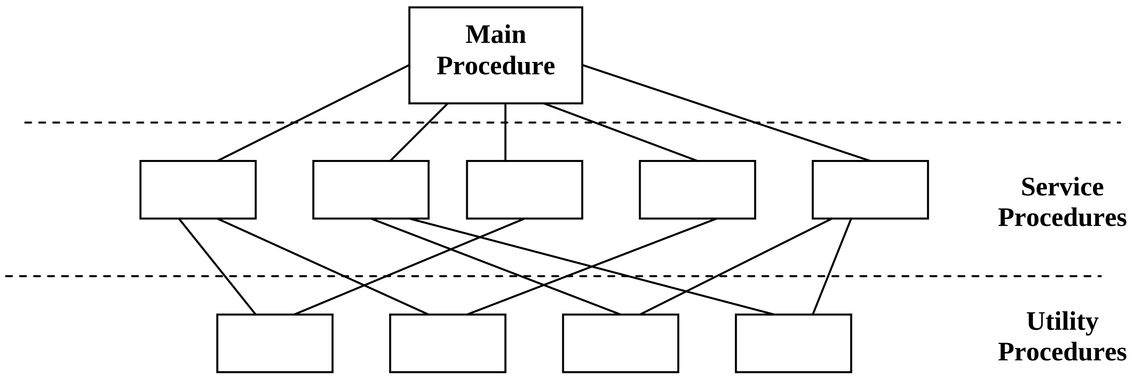
In all the discussions above we have mainly looked at the operating system from an outside view. In this section we look inside the operating system to see the various ways they can be structured.

Monolithic Systems

Operating Systems

One way an operating system can be structured is not to have a structure at all. That is, the operating system is simply a collection of procedures. Each procedure has a well defined interface and any procedure is able to call any other procedure.
The operating system is constructed by compiling all the procedures into one huge monolithic system. There is no concept of encapsulation, data hiding or structure amongst the procedures.

However, you find that the way the system procedures are written they naturally fall into a structure whereby some procedures will be high level procedures and these will call on other utility procedures.
This diagram shows this structure. The main procedure is called by the user programs. These call the service procedures which, in turn call on utility procedures.



Layered Systems

In 1968 E. W. Dijkstra and his students built an operating system that was structured into layers.
It can be viewed as a generalisation of the model shown above, but this model had six layers.

Layer 0	was responsible for the multiprogramming aspects of the operating system. It decided which process was allocated to the CPU. It dealt with interrupts and performed the context switches when a process change was required.
Layer 1	was concerned with allocating memory to processes.
Layer 2	deals with inter-process communication and communication between the operating system and the console.
Layer 3	managed all I/O between the devices attached to the computer. This included buffering information from the various devices.
Layer 4	was where the user programs were stored.
Layer 5	was the overall control of the system (called the system operator)

As you move through this hierarchy (from 0 to 5) you do not need to worry about the aspects you have “left behind.” For example, user programs (level 4) do not have to worry about where they are stored in memory or if they are currently allocated to the processor or not, as these are handled in level 0 and level 1.

Virtual Machines

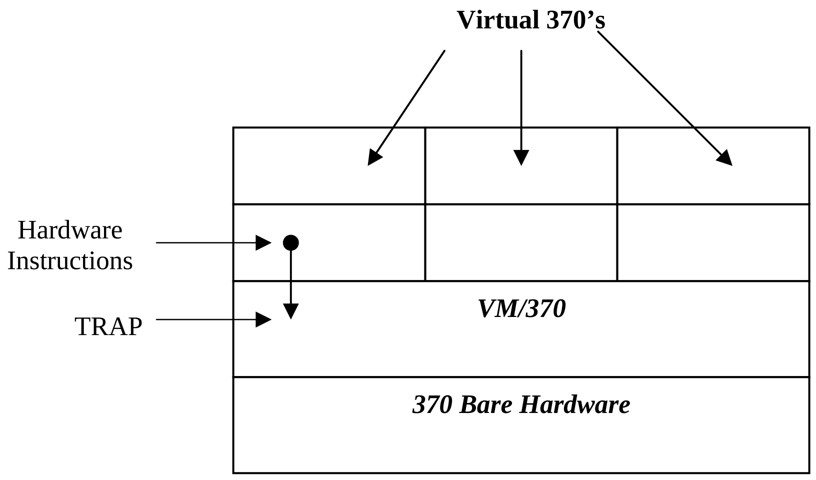
Virtual machines mean different things to different people. For example, if you run an MS-DOS prompt from with Windows 95/98/NT you are running, what Microsoft call, a virtual machine. It is given this name as the MS-DOS program is fooled into thinking that it is running on a machine that it has sole use of.

ICL’s mainframe operating system is called VME (Virtual Machine Environment). The idea is that when you log onto the machine a VM (Virtual Machine) is built and it looks as if you have the computer all to yourself (in an abstract sense – nobody really expects to have an entire mainframe to themselves).

Both of these (Windows 95/98/NT and VME) are fairly recent developments but one of the first operating systems (VM/370) was able to provide a virtual machine to each user. In addition, each user was able to run different operating systems if they so desired. This is a major achievement, if you think about it, as different operating systems will access the hardware in different ways (to name just one problem).

The way the system operated was that the bare hardware was "protected" by VM/370 (called a ***virtual machine monitor***). This provided access to the hardware when needed by the various processes running on the computer.
In addition, VM/370 created virtual machines when a user required one. But, instead of simply providing an extension of the hardware that abstracted away the complexities of the hardware, VM/370 provided an exact copy of the hardware, which included I/O, interrupts and user/kernel mode.
Any instructions to the hardware are trapped by VM/370, which carried out the instructions on the physical hardware and the results returned to the calling process.

The diagram below shows a model of the VM/370 computer.



Client-Server Model

One of the recent advances in computing is the idea of a client/server model. A server provides services to any client that requests it. This model is heavily used in distributed systems where a central computer acts as a server to many other computers.
The server may be something as simple as a print server, which handles print requests from clients. Or, it could be relatively complex, and the server could provide access to a database which only it is allowed to access directly.

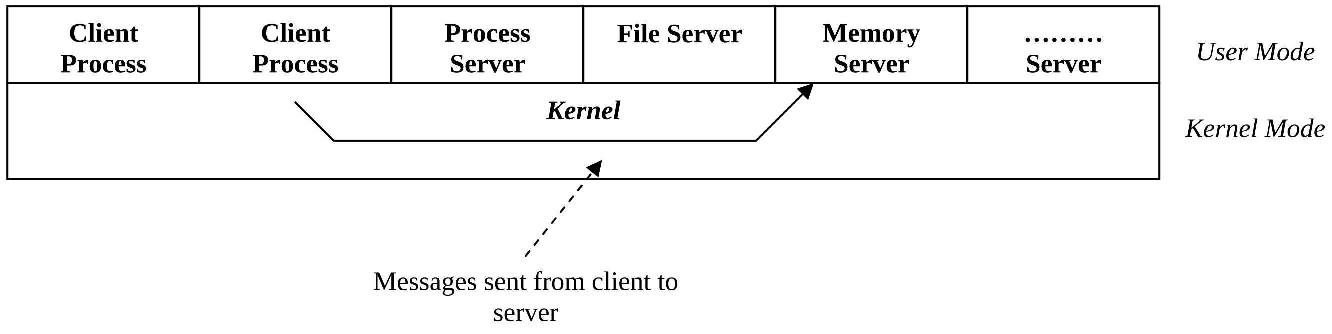
Operating systems can be designed along similar lines. Take, for example, the part of the operating system that deals with file management. This could be written as a server so that any process which requires access to any part of the filing system asks the file management server to carry out a request, which presents the calling client with the results.
Similarly, there could be servers which deal with memory management, process scheduling etc.

The benefits of this approach include

- It can result in a minimal kernel. This results in easier maintenance as not so many processes are running in kernel mode. All the kernel does is provide the communication between the clients and the servers.
- As each server is managing one part of the operating system, the procedures can be better structured and more easily maintained.

- If a server crashes it is less likely to bring the entire machine down as it won't be running in kernel mode. Only the service that has crashed will be affected.

The client-server model can be represented as follows



References

- Levy, S. 1994. Hackers.
- Tanenbaum, A., S. 1992. Modern Operating Systems (1st ed.). Prentice Hall.
- Tanenbaum, A., S. 2001. Modern Operating Systems (2nd ed.). Prentice Hall.
- Tanenbaum, A., S. 2008. Modern Operating Systems (3rd ed.). Prentice Hall.