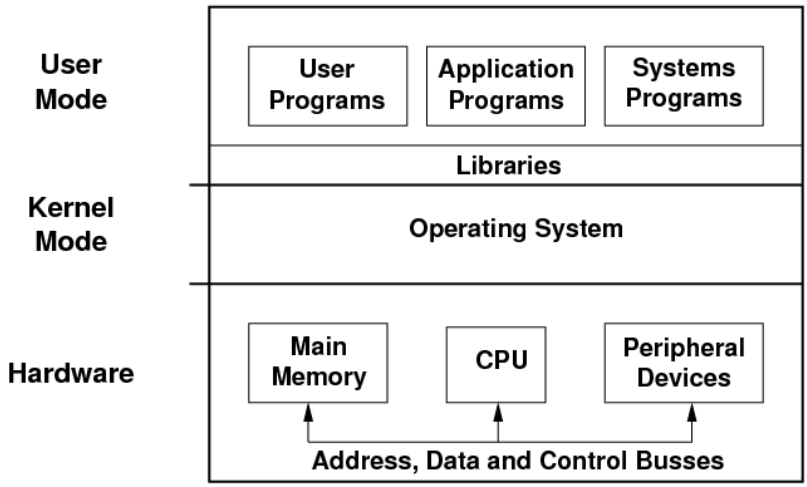
Introduction to Operating Systems

1 Introduction to Operating Systems

1.1 What is an Operating System?

- Without software, a computer is effectively useless. Computer software controls the use of the hardware (CPU, memory, disks etc.), and makes the computer into a useful tool for its users.
- As we saw in the first lecture, the software side of a computer can be divided into programs, libraries and the operating system.
- Each layer hides much of the complexity of the layer below, and provides a set of abstract services and concepts to the layer above.



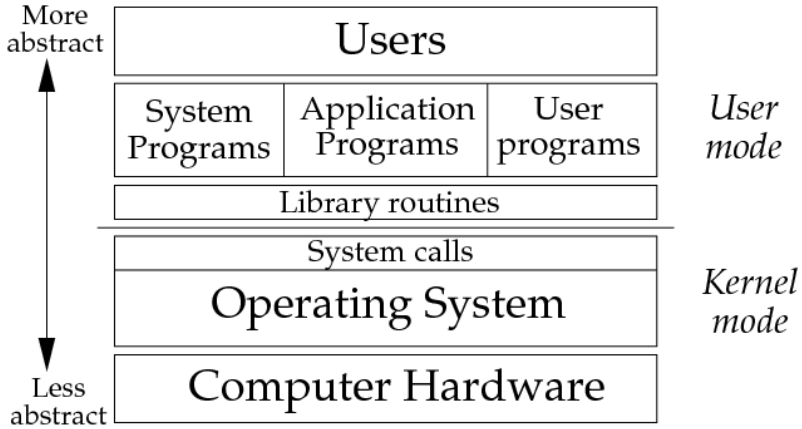
- For example, the computer's hard disk allows data to be stored on it in a set of fixed–sized blocks. The operating system hides this complexity, and provides the concept of files to the application software. In turn, an application program such as a word processor hides the idea of a file, and allows the user to work with documents instead.
- The most fundamental of all system software is the **operating system**. It has three main tasks to perform.
  1. The operating system must shield the details of the hardware from the application programs, and thus from the user.
  2. The operating system has to *substitute* a set of abstract services to the application programs, to replace the physical hardware services. When applications use these abstract services, the operations must be translated into real hardware operations.
  3. Finally, the resources in a computer (CPU, memory, disk space) are limited. The operating system must act as a resource manager, optimising the use of the resources, and protecting them against misuse and abuse. When a system provides multiuser or multitasking capabilities, resources must be allocated fairly and equitably amongst a number of competing requests.

1.2 Kernel Mode and User Mode

- Because an operating system must hide the computer's hardware, and manage the hardware resources, it needs to prevent the application software from accessing the hardware directly. Without this sort of protection, the operating system would not be able to do its job.
- The computer's CPU provides two modes of operation which enforce this protection. The operating system runs in **kernel mode**, also known as supervisor mode or privileged mode. In kernel mode, the software has complete access to all of the computer's hardware, and can control the switching between the CPU modes. Interrupts are also received in the kernel mode software.
- The rest of the software runs in **user mode**. In this mode, direct access to the hardware is prohibited, and so is any arbitrary switching to kernel mode. Any attempts to violate these restrictions are reported to the kernel mode software: in other words, to the operating system itself. Programs running in user mode are given an **address space**, visible only to themselves, which contains enough memory for them to do their job.
- By having two modes of operation which are enforced by the computer's own hardware, the operating system can force application programs to use the operating system's abstract services, instead of circumventing any resource allocations by direct hardware access.

1.3 APIs: Interfaces to Programs

- If the software layers hide complexity and replace it with simpler abstractions, how do programs access these abstractions?
- Each layer provides an **API** (Application Program Interface), which is the set of functions and commands that it provides.



- At the top of the operating system are the **system calls**. These are the set of abstract operations that the operating system provides to the applications programs. This interface is generally constant: users cannot change what is in the operating system while it is running. When executed, a system call provides a controlled transition from user mode to kernel mode.
- Above the system calls are a set of **library routines** which come with the operating system. These are functions and subroutines which are useful for many programs.
- Programs do the work for the user. Regardless of their type, all programs can use the library routines and the system calls that come with an operating system. In fact, the only way to ask the kernel to do anything is to perform a system call.

1.4 Types of Operating Systems

Every operating system is different, and each is designed to meet a set of goals. However, we can generally classify operating systems into the following categories.

- A **mainframe operating system** runs on mainframes, which provide immense I/O and availability. They may provide a **batch** environment: jobs are segregated into batches with similar requirements. Each batch is given to the computer to run. When jobs with similar system requirements are batched together, this helps to streamline their processing.

  Alternatively, the system may provide **transaction processing**, where hundred or thousands of small requests must be performed every second.

  User interaction is generally lacking in these systems, as the emphasis is on the computer's utilisation. An example mainframe system is IBM's OS/390.
- A **server operating system** runs on servers. The aim here is to provide services to many users simultaneously. Services may include applications, file storage, print and other networking operations.

  The emphasis here is on user response time as well as computer utilisation. Server systems are usually more general–purpose than the other types of operating systems. Example multiprogramming systems are Linux, Solaris and Windows 2K.
- A **personal computer operating system** is designed to provide a good environment to a single user at a time. User interface elements (desktops, GUIs etc.) are important here. Response time and the user experience is often more important that efficient computer utilisation. Windows 7 and Mac OS X are examples. Linux also fits in here, although it can deal with multiple users at the same time.
- A **real–time system** is designed to respond to input within certain time constraints. This input usually comes from external sensors, and not from humans. Thus, there is usually little or no user interaction. Many embedded systems are also real–time systems. An example real–time system is the QNX operating system.
- An **embedded operating system** usually has the operating system built into the computer, and is used to control external hardware. There is little or no application software in an embedded system. Examples here are the PDAs that everybody seems to have, and of course the computers built into DVDs, microwaves, and into most cars. Symbian is an example of an embedded operating system.

In this course, we will concentrate on server operating systems: these are much more sophisticated and complex then the other operating system types, and will give us a lot more to look at. In particular, we will also concentrate on *multi–user* systems: these are systems which support multiple users at the same time.

2 Design Principles & Concepts

- The services provided by an operating system depends on the **concepts** around which the operating system was created; this gives each operating system a certain `feel' to the programmers who write programs for it.
- We are talking here not about the `look & feel' of the user interface, but the `look & feel' of the *programmer's interface*, i.e the services provided by the API (i.e. the set of system calls).
- Although each operating system provides its own unique set of services, most operating systems share a few common concepts. Let's briefly take at look at each now. We will examine most of these concepts in detail in later topics.

2.1 The Process

- Most operating systems provide the concept of a process. Here, we need to distinguish between a **program** and a **process**.

- A **program** is a collection of computer instructions plus some data that resides on a storage medium (e.g. a disk), waiting to be called into action.
- A **process** is a program during execution. It has been loaded into the computer's main memory, and is taking input, manipulating the input, and producing output.
- Specifically, a process is an enviroment for a program to run in. This includes:
  - An **address space**, which is a set of memory locations which stores the machine–code instructions required by process, and the process' variables. The address space is protected, so that other processes cannot interfere.
  - A set of **CPU registers**, so that the process can perform some operations quickly (e.g additions, subtractions) without having to access memory, which can be slow.
  - The ability to invoke system calls, so that the process can obtain services from the operating system.
- This environment is often known as a process' **context**.

2.2 Memory

- Part of every computer's hardware is its main memory. This is a set of temporary storage locations which can hold machine code instructions and data. Memory is volatile: when the power is turned off, the contents of main memory are lost.
- In current computers, there are usually several gigabytes of memory (i.e billions of 8–bit storage areas). Memory contents can be accessed by reading or writing a **memory location**, which has an integer **address**, just like the numbers on the letter boxes in a street.
- Memory locations often have a hardware protection, allowing or preventing read and writes. Usually, a process can only read or write to a specific set of locations that have been given to it by the operating system: its *address space*.
- The operating system allocates memory to processes as they are created, and reclaims the memory once they finish. As well, processes can usually request more memory, and also relinquish this extra memory if they no longer require it. The operating system must allocate memory on a fair and equitable basis to all the processes on the system.

2.3 Files

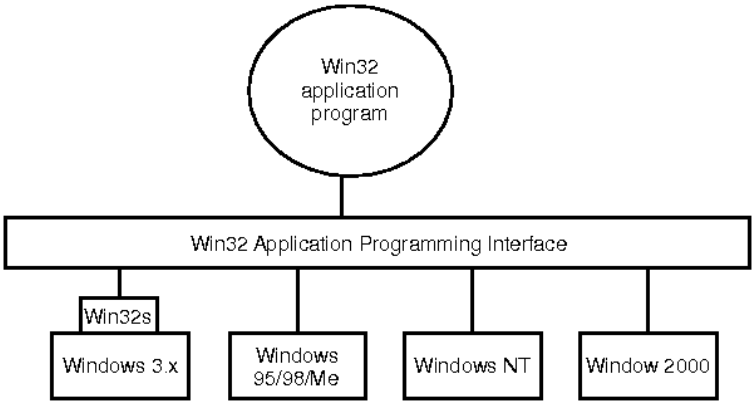
- Files are storage areas for programs, source code, data, documents etc. They can be accessed by processes, but don't disappear when processes die, or when the machine is turned off. They are thus *persistent* objects.
- Operating systems provide mechanisms for file manipulation, such as open, close, create, read and write.
- As part of the job of hiding the hardware and providing abstract services, the operating system must map files onto areas on disks, USB and flash devices, and tapes. The operating system must also deal with files that grow or shrink in size.
- Some operating systems don't enforce any structure to files, or enforce particular file types types. Others distinguish between file types and structures, e.g. Java source files, text documents, executable files, data files etc.
- Most operating systems allow files to have **permissions**, allowing certain types of file access to authorised users only.
- **Directories** (i.e folders) may exist to allow related files to be collected. The *main* reason for the existence of directories is to make file organisation easier and more flexible for the user.

2.4 Windows

- Nearly all operating systems these days provide some form of graphical user interface, although in many cases a command–line interface is also available.
- In these operating systems, there are services available to allow processes to do graphical work. Although there are primitive services such as line and rectangle drawing, most GUI interfaces provide an abstract concept known as the **window**.
- The window is a logical, rectangular, drawing area. Processes can create one or more windows, of any size. The operating system may decorate each window with borders, and these may include *icons* which allow the window to be destroyed, resized, or hidden.
- The operating system must map these logical windows onto the physical display area provided by the video card and computer monitor. As well, the operating system must direct the input from the user (in the form of keyboard input, and mouse operations) to the appropriate window: this is known as changing the input *focus*.

2.5 Different Operating System, Different APIs

- From a programmer's point of view, an operating system is defined mainly by the Application Program Interface (API) that it provides, and to a lesser extent what library routines are available.
- It follows, therefore, that a number of different operating system products may provide exactly the same Application Program Interface, and thus appear to be the same operating system to the programmer. The most obvious example of this is Unix.
- Unix is really not a single operating system, but rather a collection of operating systems that share a common API. This API has now been standardised, and is known as the POSIX standard. Solaris, Linux, Minix, Mac OS X and FreeBSD are all examples of Unix operating systems.
- What this means is that a program written to run on one Unix platform can be recompiled and will run on another Unix system. As long as the set of systems calls are the same on both systems, the program will run on both systems.
- Another group of operating systems which share a common API are the Windows systems from Microsoft: Windows CE, Windows 2K, Windows XP and Windows 7. Although each one is structurally different, a program can be written to run on all three.



(from Tanenbaum)

2.6 The OS vs The User

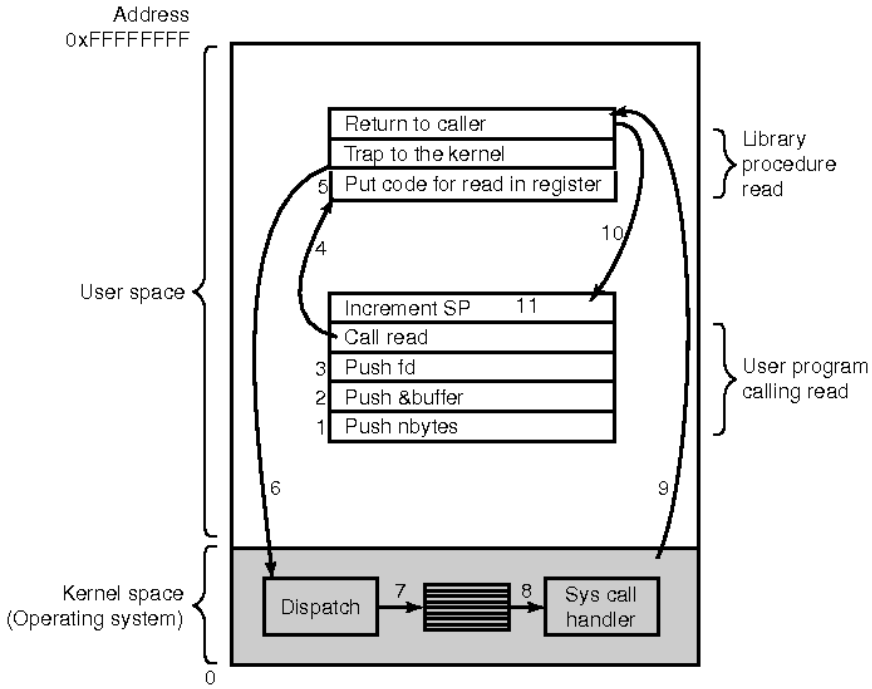
- The operating system must hide the actual computer from the users and their programs, and present an abstract interface to the user instead. The operating system must also ensure fair resource allocation to users and programs. The operating system must shield each user and her programs from all other users and programs.
- Therefore, the operating system must prevent all access to devices by user programs. It must also limit each program's access to main memory, to only that program's memory locations.
- These restrictions are typically built into the CPU (i.e into unchangeable hardware) as two operating modes: user and kernel mode. In **kernel mode**, all memory is visible, all devices are visible, all instructions can be executed. The operating system must run in kernel mode, why? In **user mode**, all devices are hidden, and most of main memory is hidden. This is performed by the *Memory Management Unit*, of which we will learn more later. Instructions relating to device access, interrupt handling and mode changing cannot be executed either.
- When a user program run as a process, the operating system forces the process to run in user mode. Any attempt to violate the user mode will cause an *exception*, which is caught by the operating system. Thus, the operating system can thus determine when user mode violations have been attempted.
- **Note:** Every interrupt or exception causes the CPU to switch from its current mode into kernel mode. Why? This ensures that the operating system, and never a user process, will catch the event. This ensures that user processes cannot hijack the operation of the system.
- Finally, because a process runs in user mode and can only see its own memory, it cannot see the operating system's instructions or data. This prevents nosy user programs from subverting the working of the operating system.

2.7 System Calls and Traps

- If the operating system is protected, how does a process ask for services from the OS? User programs can't call functions within the operating system's memory, because it can't see those areas of memory.
- Even worse, the operating system executes in **kernel mode**, which the user mode process cannot reach.
- To solve the problem, all computers with a user mode/kernel mode split have a special user–mode machine instruction, known as a **TRAP** instruction.
- When the TRAP instruction is executed in user mode, an **exception** occurs. The CPU switches to kernel mode, and sets the **Program Counter** to a specific location. The CPU continues executing instructions starting at this specific location, which is always somewhere in the operating system.
- The TRAP instruction merely provides the mechanism to get to the operating system from user mode. However, the operating system has to be told exactly what services a user mode process wants.
- Consider the following C command to read `nbytes` of data from the file `fd` into the `buffer`. This requires the services of the operating system to get the data from the disk and place it into the buffer.

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

- To perform the *read()* operation, the process first pushes the arguments to the operation onto the stack (steps 1–3 below). The process then calls the *read()* system call, which is normally written in assembly code (step 4 below).



(from Tanenbaum)

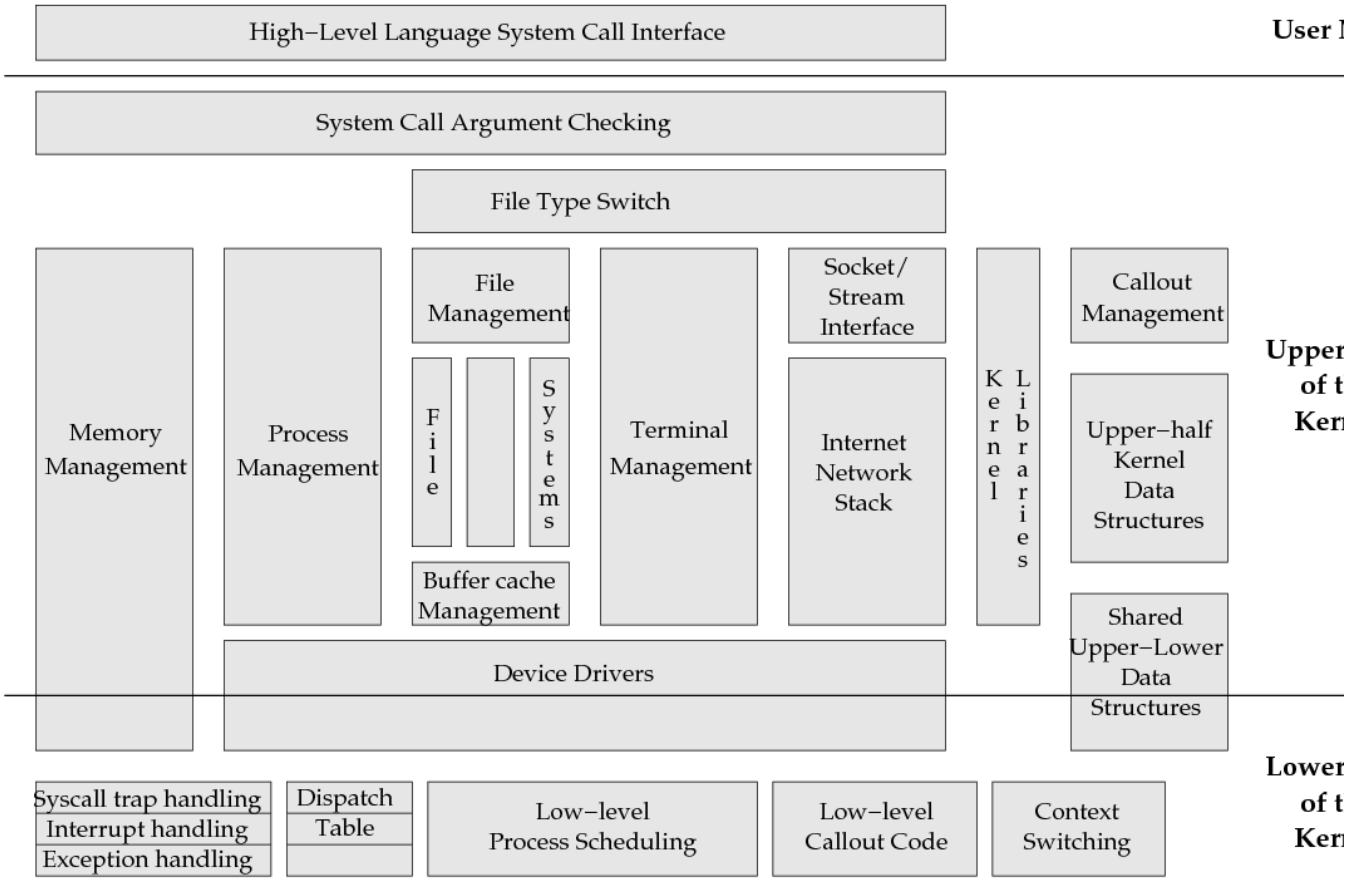
- The system call places a number identifying the *read()* operation into a register (step 5), and then executes the TRAP instruction (step 6). The CPU jumps to a fixed location within the kernel. Once in the kernel, the *dispatch handler* uses the register value as an index into a table of system calls (step 7), and calls the correct kernel function (step 8).
- The kernel is now able to perform the *read()* request, and as it is running in kernel mode, it can look at the arguments to *read()* which are on the stack, and it can also command the hard drive to fetch the appropriate data.
- When the data has been read in and copied into the buffer, the kernel needs to return to user mode, and resume the execution of the process. A new machine instruction called ReTurn from Interrupt (RTI) does this (step 9). Back in user mode, the CPU is now executing the next instruction after the TRAP, but is still inside the *read()* system call. Any return value from the kernel is collected, and the system call returns to the original calling function (step 10).
- Remember, the TRAP instruction ensures that the only access to the privileged operating system is via a single, well–protected entry point. It is up to the kernel to inspect the arguments provided by the user process, and to ensure that they are valid.

2.8 Operating System Structure

- The implementation of an operating system is completely up to its designers, and throughout the course we will look at some of the design decisions that must be made when creating an operating system.
- In general, none of the implementation details of an operating system are visible to the programmer or user: these details are hidden behind the operating system's Application Program Interface. The API fixes the "look" of the operating system, as seen by the programmer.
- This API, however, can be implemented by very different operating system designs. So, for example, Solaris, Linux and FreeBSD all provide a POSIX API, but all three systems have a very different operating system architecture.
- We will examine the two most common operating system designs, the monolithic model and the client–server model.

2.9 The Monolithic Model

- In the monolithic model, the operating system is written as a collection of routines, each of which can call any of the other routines as required. At build–time, each routine is compiled, and then they are all linked together to create a single program called the operating system *kernel*.
- When the operating system is started, this kernel is loaded into the computer's memory, and runs in *kernel mode*. Most versions of Unix, including Linux, use the monolithic design model.
- The monolithic design model suffers from the fact that every part of the operating system can see all the other parts; thus, a bug in one part may destroy the data that another part is using. Recompilation of the operating system can also be slow and painful.
- To reduce this shortcoming, most designers place some overriding structure on their operating system design. Many of the routines and data structures are `hidden' in some way, and are visible only to the other routines that need them.
- An abstract map of Unix's architecture is shown in the diagram below. As you can see, the functionality provided by the kernel is broken up into a number of sections. Each section provides a small number of interface routines, and it is these routines which can be used by the other sections.
- Because Unix is monolithic, nothing stops one section of the operating system from calling another with *function calls*, or using another section's data. Each box is a set of C source files.

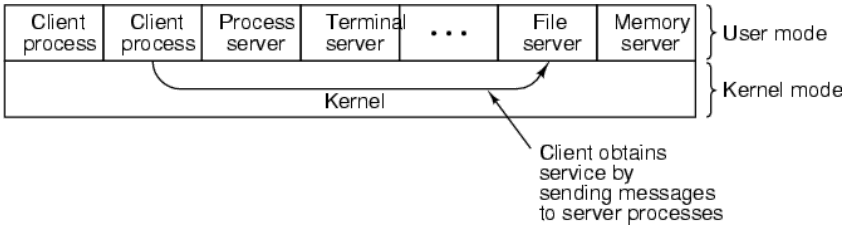


Overview of the Unix Kernel Architecture

- The **upper-half** of the kernel is the portion that runs in kernel mode but which is invoked by a system call.
- The **lower-half** of the kernel is the portion that reacts to exceptions and interrupts.
- This is an important distinction which we will return to.

2.10 Client-Server Model

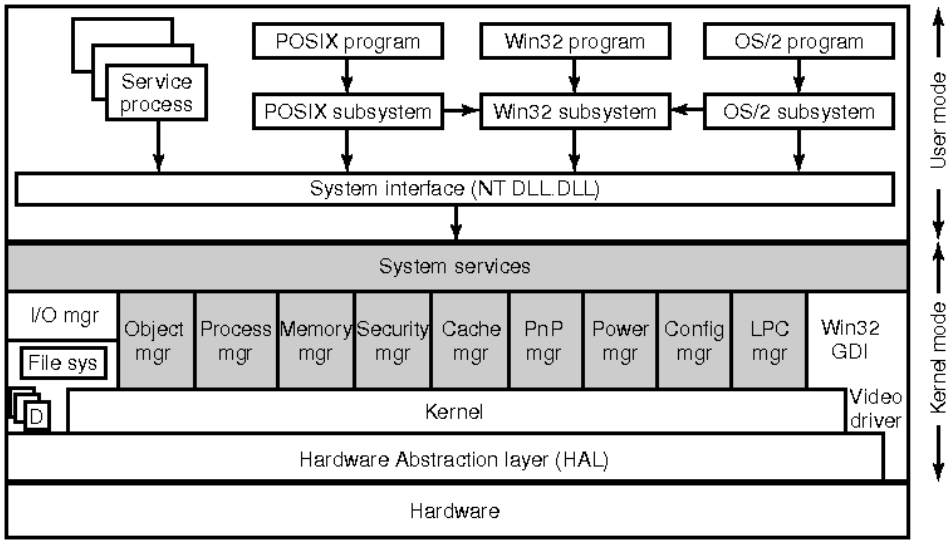
- An alternative method of operating system design, called the client-server model, tries to minimise the chance of a bug in one part of the operating system from corrupting another part.
- In this model, most of the operating system services are implemented as privileged processes called **servers**. Remember, each process is protected against interference by other processes. These servers have some ability to access the computer's hardware, which ordinary processes cannot.
- Ordinary processes are known as **clients**. These send requests in the form of **messages** to the servers, which do the work on their behalf and return a reply.
- The set of services that the servers provide to the user processes thus form the operating system's Application Program Interface.



(from Tanenbaum)

- The messages sent between the clients and servers are well-defined `lumps' of data. These must be copied between the client and the server. This copying can slow the overall system down, when compared to a monolithic system where no such copying is required. The servers themselves also may need to intercommunicate.
- There must be a layer in the operating system that does message passing. This model can be implemented on top of a single machine, where messages are copied from a client's memory are into the server's memory area. The client-server model can also be adapted to work over a network or distributed system where the processes run on several machines.

- Windows 2K uses the client–server model, as shown in the diagram below. Most of the subsystems are privileged processes. Other client–server based operating systems are Minix and Plan 9.



(from Tanenbaum)

2.11 Microkernels

- You will have noticed that most, if not all, of an operating system appears to run in kernel mode.
- This is not exactly necessary. The only code which has to run in kernel mode is that which:
  - has to deal with hardware directly: I/O, interrupts, exceptions, syscall catching, memory permissions.
  - enforces security restrictions at the hardware level.
- There is a lot of operating system software which deals with the OS abstractions, e.g. files, directories, windows, and which does not really need to run in kernel mode.
- Where a kernel does little more than process scheduling, basic memory management, interrupt handling and message passing, it is called a **microkernel**.
- The code which deals with higher–level abstractions such as files, virtual memory, windows etc. can run in user mode. Message passing is used to pass requests from the user–mode programs to the user–mode parts of the operating system.
- Minix and QNX are examples of microkernels.

File translated from TEX by TTH, version 3.85.
On 13 Jan 2012, 15:26.