# 1 What is an operating system?

At first sight, an operating system is 'just a program' that supports the use of some hardware. It emulates an ideal machine – one with a uniformity and plenty of resources – that would be expensive or infeasible to implement directly in hardware.

Although the processor(s) of the underlying hardware execute most of the instructions of a program directly (for speed) the operating system catches some operations and emulates the unimplemented parts of the ideal machine. These may be simplified, standard peripherals; or they may be memory bigger than that provided by the hardware; or they may be numbers of processors in excess of those in the real hardware.

The idea is to make the machine seen by the user one that is easy to program, while keeping the hardware as cheap and simple as possible to build.

Having provided plentiful resources, the operating system also has to manage them: managing resources means sharing them between programs, managing co-operation and conflict, and protecting programs both from each other and from themselves. These resources might be peripherals, memory, processors, time.

Most of a modern operating system consists of server processes which are just other 'user programs'. Some privileged code necessarily lives in a small *kernel*. Typically, non-kernel code is prevented from doing anything to other programs, or to parts of the real hardware which might affect the behaviour of other programs. Any interaction between programs, or I/O, therefore necessarily has to be implemented by the kernel. In particular, starting and stopping programs has to be done by the kernel.

Some critical functions are provided with hardware support because of timing constraints. For example, many modern operating systems provide a virtual memory abstraction. Virtual memory involves doing some calculations on every memory access; since there are several memory accesses per instruction this means that those calculations cannot be implemented by instructions. There has to be some hardware support for virtual memory.

Operating systems are interesting partly because they were the first parallel programs, and it was their design which drove the development of parallel programming techniques. One theme which will recur is the distinction between *mechanism* which ensures safety, and *policy* which adjusts performance.

## 1.1 Interrupts

Interrupts provide the ability rapidly to resume a suspended process, providing a small amount of processor time at very short notice when an urgent request is made. There is nothing special about interrupts, they are just polling, but implemented in the hardware. The fetch/execute cycle of the process might be described something like

```
$pc := pc_0$;
**while not** $halted$ **do**
    $ir := fetch(pc)$;
    $pc := pc + 1$;
    $execute(ir)$
**end**
```

and with a facility for interrupts it becomes

```
$pc := pc_0$;
**while not** $halted$ **do**
    **if** $interrupt$ **and not** $kernel$ **then**
        $kernel :=$ **true**;
        $pc_{save} := pc$;
        $pc := pc_{int}$
    **end**;
    $ir := fetch(pc)$;
    $pc := pc + 1$;
    $execute(ir)$
**end**
```

The effect of the (first) interrupt is to transfer control to another piece of code within the time it takes to execute one instruction. In general there are other things to do at this point: preserving the state of the program that was being executed, and so on, but the point is usually to deliver computing power at very short notice. In this description, the flag *kernel* is set to prevent repeated interrupts. More generally, there might be different priorities of interrupt, with interrupts at one level preventing interrupts at lower levels.

Interrupts might be caused by

**I/O** : completion of long tasks by slow devices, error conditions in peripheral devices.

**Hardware errors** : memory faults, power failure.

**Timers** : regularly scheduled events, or timeouts.

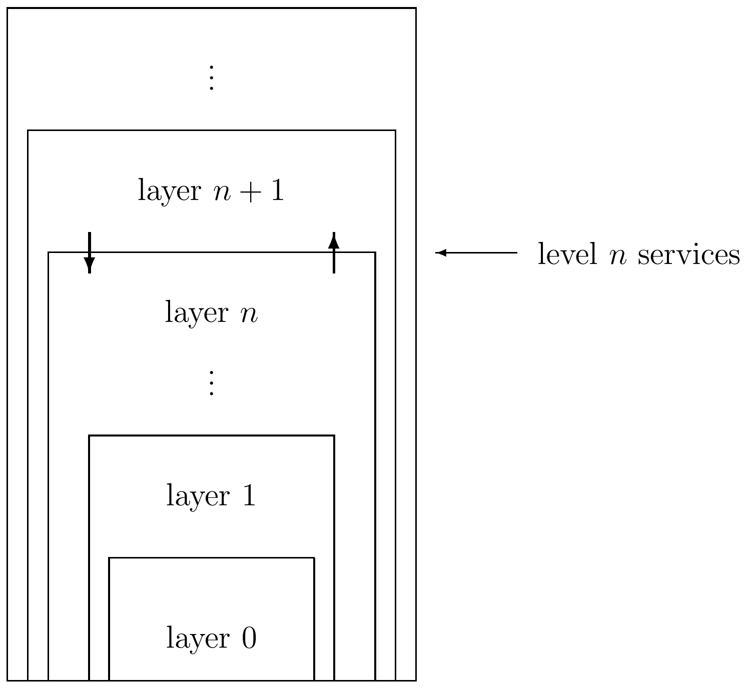
**Program errors** : breach of precondition for operations, addressing errors, deliberate calls to the operating system (*system calls*).

When an interrupt happens, most processors will switch into a different state, known as *kernel mode* or *supervisor mode.* As well as suppressing interrupts (or reducing the range of interrupts possible) instructions executed in kernel mode might have different effects. For example, some instructions modifying the states of parts of the machine might be invalid in normal (‘*user*’) mode. Instructions in kernel mode might have unrestricted access to the address space, when in user mode they would be restricted to those parts of memory allocated to the current process. In this way processes can be protected against each other’s failures, but a trusted *kernel* of the operating system would be able to do those things that process need to do to communicate with each other, or with the outside world.

## 1.2 Structure of a kernel

The most common structure for operation systems is still a complete lack of structure. because it is hard to disentangle all the parts of the services provided by an operating system it is all to easy to write a single monolithic mess. While *system calls* to the kernel from user code are obliged to enter through a relatively small number of well-defined interfaces, procedure calls from kernel code to kernel code have to be trusted to be right.

Kernels with loadable *modules* provide the ability to omit bits of the code which are not going to be needed in particular configurations, but do not of themselves make for any greater logical structure.



A *layered* operating system tries to impose a hierarchy on the tasks of the kernel. Each layer in the operating system provides services to the layer above it, with the user's programs being the outer layer. This idea dates back to the 1960s where it was used by Dijkstra as a design principle in building the THE operating system at *Technische Hogeschool Eindhoven*.

There was no protection provided by the hardware for the THE layers to be insulated from each other, but a processor with several layers of kernel status could provide protection so that layer $n + 1$ running its code at protection level $n + 1$ would be unable to access the code and data of layers 0 to $n$ except by system calls to layer $n$. Such hardware protection is provided by much modern processor hardware, but rarely used by popular operating systems.

*Client/server* operating systems move as much as possible of the functions of the operating system out of the kernel into user-level server processes. To a large extent, the function of the kernel is simply to pass messages from user programs (client processes) to server processes and back again. However there remains the question of how a server communicates with the hardware. This can be done either by having some trusted server processes run in kernel mode (effectively, a kernel with internal multitasking), or by keeping all the *mechanism* in the kernel and putting only the *policy* in the servers.

One of the advantages of client/server structures is that it becomes easy to implement distributed operating system services: a client need not know that services are provided on the same processor, they may be provided by another computer to which the kernel sends messages over a network.

## 1.3 The input/output system

A substantial part of an operating system is often devoted to handling input and output. This is partly for *efficiency*, because the peripheral devices (and the world with which they interact) can be slow and complicated to use. It is for *device independence*, so that user programs do not need to be modified to work with different models of printer or disk, or the substitution of one form of backing store (say on a network) for another (such as a local disk). It also makes it possible to handle different devices *uniformly*: an abstraction such as a stream of data is just a stream of data to the user program, whether it comes from a keyboard or from a data logger.

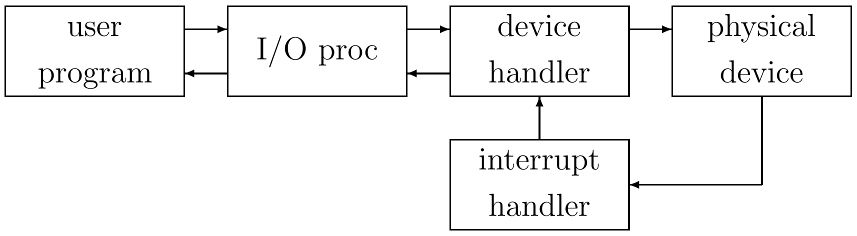
Some characteristics of devices can be hidden behind such abstraction, others cannot. The relevant characteristics are speed of transfer – both the *latency* between a request and a response, and the sustainable *throughput* once a transfer is started. It is hard to hide hide these. A device may naturally transfer a bit at a time, or a byte at a time, or some other word size, or it may always transfer large fixed size blocks, or even variable sized blocks.

UNIX traditionally tries to hide all devices behind one of two abstractions: devices (like a user terminal) which implement a byte-by-byte transfer, and devices which implement fixed size blocks (like disks). Even quite low-level UNIX program are written to operate only on byte devices and block devices, without needing to be modified for new kinds of byte device or block device.

Other characteristics of the real device that are relevant will include such things as which operation are permitted, which error conditions can arise, and what the data representation is (ranging from what the character set is to which video encoding is used).

Some devices are difficult to package in this way, such as a timer or a graphics card.

The flow of control might well be structured like this:



Each user program need know only about *streams* of data; it calls library routines which run as user code (but do not have to be rewritten in every program). The I/O procedure associates the stream with a device, checks parameters for sanity, and constructs a request to the device handler to transfer a given quantity of data. In UNIX this request is a call to one of two procedures to *read* or *write* to a logical device.

The device handler typically handles a pool of requests. It selects (by some policy) a request from this pool, and causes the physical device to perform the operation. This is likely to require privileged access to the hardware, and so will happen in kernel mode, but in general parts of the device handler (for example, policy implementation and data translation) can be written as co-operating user processes. When the operation is complete, this is reported back to the I/O procedure and ultimately to the user. Being device-specific, the device driver usually comes packaged with the hardware as a component module which can be added to the operating system.

Many responses from I/O devices will take so long that it is not feasible to wait for them. The device handler is often written as a co-routine or process which can be repeatedly resumed by interrupts from the device.

At least for non-interactive traffic, the overhead of I/O activity can often be reduced by aggregating a number of consecutive requests. This smooths out bursts of I/O activity, and reduces the cost of latency. Double buffering uses two buffers on each stream: one being filled while the other is being emptied. This allows the I/O activity to overlap with computation.

## 1.4 Disk-like devices

Traditionally large quantities of cheap storage involve mechanical devices with moving magnetic oxide and some number of (often movable) read/write heads. Most commonly these days this is a constantly spinning disk with a head that can be moved radially. They record data in blocks around concentric tracks.

(Traditionally, several physical disks on one spindle have one head per platter, with the heads moving together; the group of tracks being read at any one time are a cylinder. The physical blocks transferred by the underlying device are called *sectors* and are often not the same size as the logical blocks of the operating system.)

These devices have quite long latency: both to move the head to the right track, and to wait for the right block of the track to come round to the head. However they also have high data rates: once a block (or sequence of consecutive blocks from one track) starts to come, it comes at a fixed rate determined by the rotational speed of the disk. Moreover, there are occasional urgent requests for action such as deciding whether there is an opportunity to carry on reading (or writing) at the end of a block.

The high data rates, and in particular the need for guaranteed throughput, mean that a device controller will very likely have its own dedicated memory buffer into which it perform a read, so that the data is not delayed by memory contention. The controller can then perform validity and consistency checks before anything is transferred into the main memory.

One way of dealing with the urgency of what to do next at the end of a transfer is to interleave the sectors of a track so that there is more time to handle the question before the numerically next sector has to be handled. Using separate heads on separate platters for adjacent transfers allows them to operate in parallel, but the extra cost of the hardware means that this is now rare.

The disk handler is likely to gather a pool of requests for a disk (from many concurrent processes) and to re-order them for efficiency. The policy for selecting the next request to be serviced has to be chosen carefully. Since the cost of moving the head is roughly proportional to the distance between tracks there is a premium on serving requests on the current track first, and then perhaps moving to the nearest track with outstanding requests.

However, this can cause starvation: if requests continue to arrive for a small number of adjacent tracks then outstanding requests on other tracks may never be serviced. The solution is to keep a pool of requests for the current track that were outstanding before the head arrived here, and service these; request that arrive later are added to a second pool which is left until later.

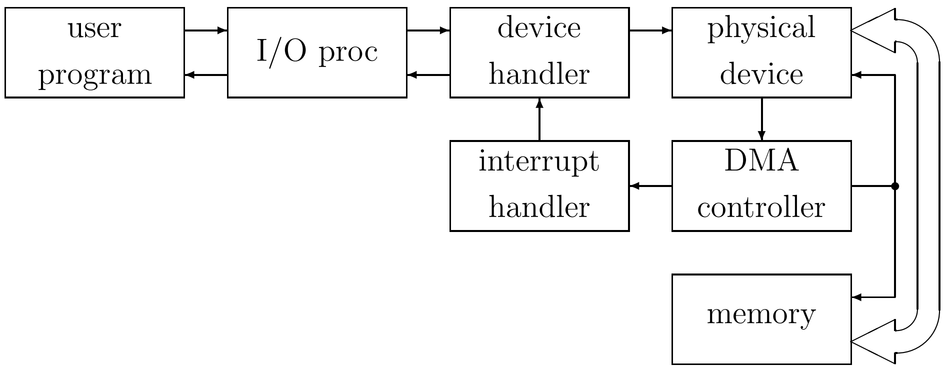
The *lift algorithm* (or elevator algorithm) avoids starvation from small groups of busy tracks sweeping alternately up and down the tracks, servicing all the requests from tracks on one side of the head in turn, and then moving back in the other direction. This is the algorithm used by the lifts in buildings.

## 1.5 Direct memory access

A high data-rate device with a controller which uses an internal buffer still has to transfer the contents of that buffer into the main memory in the processor's memory (or vice versa). This might done by code in the device handler, at the expense of tying up the processor. In particular it might well require several times more instructions than memory transfers.

A faster data transfer might be arranged by a *direct memory access* controller. This would appear as an additional device with registers defining the scope of a transfer. It interacts with the memory as if it were an additional processor sharing the same memory and address space as the processor(s).

When the device reports that the data are ready in its buffer, the dma runs through the relevant addresses, simultaneously requesting that the device read from its buffer and the memory accepts a write (or vice versa).



Although the actual transfers contend with the processor(s) for access to the memory, the processor need not be troubled until the transfer is complete.

In this way the device controller can provide for its device uncontended access to memory in its buffer, without making apparent the penalty of having to copy between that buffer and the processor-accessible address space.

## Exercises

1.1 Suppose that some user process takes time $p$ to produce each output, that the total overhead of calling the output routine is $w$ and that the output device takes a time $c$ to consume each output.

How much greater a throughput would the output system have if the producer buffered $n$ items of output before each call of the output routine.

How much greater throughput would the output system have if it were double-buffered? (Double buffering allows the producer to write to one buffer at the same time as the consumer is emptying another.)

What would happen to the throughput with a third buffer? Is there any purpose to triple buffering?

1.2 Why is it usual if the output of a program is being sent to a printer, for the output to be written in its entirely to a disk file before it is subsequently copied to the printer? (This is called *spooling*, because fifty years ago the intermediate file would have been a spool of magnetic tape.)

1.3 Show that serving disk requests in order of *shortest seek first* (that is, serving the requests on the track nearest to the position of the head) can be faster than serving them in the order in which they are made.

Show that shortest seek first can cause a pending request to be indefinitely delayed.

1.4 What should the lift algorithm do if new requests are received for the cylinder on which the heads are currently placed?

1.5 You know that your operating system implements the lift algorithm to schedule disk accesses to its single physical disk, and that it sorts the requests for a single track into sector order for even greater efficiency. As an experiment you write a small program which reads a randomly determined sequence of a million disk blocks spread over the whole disk, and run it on an otherwise idle machine.

You find that it performs much as you would have expected it to do if the disk requests were served in the order in which they are requested. Should you be surprised?