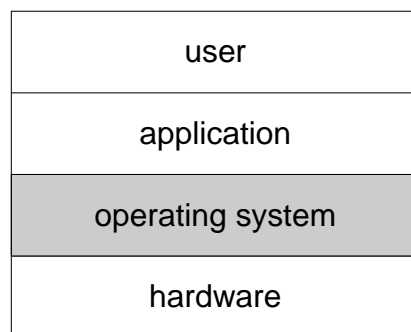

Introduction

In the simplest scenario, the operating system is the first piece of software to run on a computer when it is booted. Its job is to coordinate the execution of all other software, mainly user applications. It also provides various common services that are needed by users and applications.

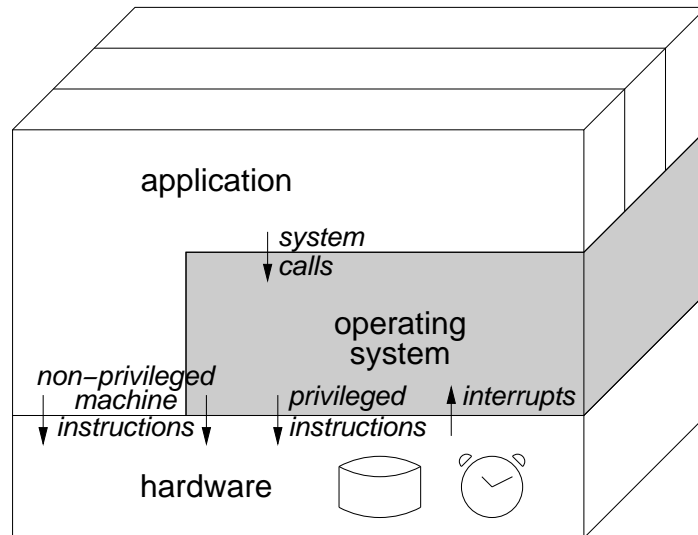
1.1 Operating System Functionality

The operating system controls the machine

It is common to draw the following picture to show the place of the operating system:



This is a misleading picture, because applications mostly execute machine instructions that do not go through the operating system. A better picture is:



where we have used a 3-D perspective to show that there is one hardware base, one operating system, but many applications. It also shows the important interfaces: applications can execute only non-privileged machine instructions, and they may also call upon the operating system to perform some service for them. The operating system may use privileged instructions that are not available to applications. And in addition, various hardware devices may generate interrupts that lead to the execution of operating system code.

A possible sequence of actions in such a system is the following:

1. The operating system executes, and *schedules* an application (makes it run).
2. The chosen application runs: the CPU executes its (non-privileged) instructions, and the operating system is not involved at all.
3. The system clock *interrupts* the CPU, causing it to switch to the clock's interrupt handler, which is an operating system function.
4. The clock interrupt handler updates the operating system's notion of time, and calls the scheduler to decide what to do next.
5. The operating system scheduler chooses another application to run in place of the previous one, thus performing a *context switch*.
6. The chosen application runs directly on the hardware; again, the operating system is not involved. After some time, the application performs a *system call* to read from a file.
7. The system call causes a *trap* into the operating system. The operating system sets things up for the I/O operation (using some privileged instructions). It then puts the calling application to sleep, to await the I/O completion, and chooses another application to run in its place.
8. The third application runs.

The important thing to notice is that at any given time, only one program is running¹. Sometimes this is the operating system, and at other times it is a user application. When a user application is running, the operating system loses its control over the machine. It regains control if the user application performs a system call, or if there is a hardware interrupt.

Exercise 1 *How can the operating system guarantee that there will be a system call or interrupt, so that it will regain control?*

The operating system is a reactive program

Another important thing to notice is that the operating system is a *reactive program*. It does not get an input, do some processing, and produce an output. Instead, it is constantly waiting for some *event* to happen. When the event happens, the operating system reacts. This usually involves some administration to handle whatever it is that happened. Then the operating system schedules another application, and waits for the next event.

Because it is a reactive system, the logical flow of control is also different. “Normal” programs, which accept an input and compute an output, have a `main` function that is the program’s entry point. `main` typically calls other functions, and when it returns the program terminates. An operating system, in contradistinction, has many different entry points, one for each event type. And it is not supposed to terminate — when it finishes handling one event, it just waits for the next event.

Events can be classified into two types: interrupts and system calls. These are described in more detail below. The goal of the operating system is to run as little as possible, handle the events quickly, and let applications run most of the time.

Exercise 2 *Make a list of applications you use in everyday activities. Which of them are reactive? Are reactive programs common or rare?*

The operating system performs resource management

One of the main features of operating systems is support for multiprogramming. This means that multiple programs may execute “at the same time”. But given that there is only one processor, this concurrent execution is actually a fiction. In reality, the operating system juggles the system’s resources between the competing programs, trying to make it look as if each one has the computer for itself.

At the heart of multiprogramming lies resource management — deciding which running program will get what resources. Resource management is akin to the short blanket problem: everyone wants to be covered, but the blanket is too short to cover everyone at once.

¹This is not strictly true on modern microprocessors with hyper-threading or multiple cores, but we’ll assume a simple single-CPU system for now.

The resources in a computer system include the obvious pieces of hardware needed by programs:

- The CPU itself.
- Memory to store programs and their data.
- Disk space for files.

But there are also internal resources needed by the operating system:

- Disk space for paging memory.
- Entries in system tables, such as the process table and open files table.

All the applications want to run on the CPU, but only one can run at a time. Therefore the operating system lets each one run for a short while, and then *preempts* it and gives the CPU to another. This is called *time slicing*. The decision about which application to run is *scheduling* (discussed in Chapter 2).

As for memory, each application gets some memory frames to store its code and data. If the sum of the requirements of all the applications is more than the available physical memory, *paging* is used: memory pages that are not currently used are temporarily stored on disk (we'll get to this in Chapter 4).

With disk space (and possibly also with entries in system tables) there is usually a hard limit. The system makes allocations as long as they are possible. When the resource runs out, additional requests are failed. However, they can try again later, when some resources have hopefully been released by their users.

Exercise 3 As system tables are part of the operating system, they can be made as big as we want. Why is this a bad idea? What sizes should be chosen?

The operating system provides services

In addition, the operating system provides various services to the applications running on the system. These services typically have two aspects: abstraction and isolation.

Abstraction means that the services provide a more convenient working environment for applications, by hiding some of the details of the hardware, and allowing the applications to operate at a higher level of abstraction. For example, the operating system provides the abstraction of a file system, and applications don't need to handle raw disk interfaces directly.

Isolation means that many applications can co-exist at the same time, using the same hardware devices, without falling over each other's feet. These two issues are discussed next. For example, if several applications send and receive data over a network, the operating system keeps the data streams separated from each other.

1.2 Abstraction and Virtualization

The operating system presents an abstract machine

The dynamics of a multiprogrammed computer system are rather complex: each application runs for some time, then it is preempted, runs again, and so on. One of the roles of the operating system is to present the applications with an environment in which these complexities are hidden. Rather than seeing all the complexities of the real system, each application sees a simpler *abstract machine*, which seems to be dedicated to itself. It is blissfully unaware of the other applications and of operating system activity.

As part of the abstract machine, the operating system also supports some abstractions that do not exist at the hardware level. The chief one is files: persistent repositories of data with names. The hardware (in this case, the disks) only supports persistent storage of data blocks. The operating system builds the file system above this support, and creates named sequences of blocks (as explained in Chapter 5). Thus applications are spared the need to interact directly with disks.

Exercise 4 *What features exist in the hardware but are not available in the abstract machine presented to applications?*

Exercise 5 *Can the abstraction include new instructions too?*

The abstract machines are isolated

An important aspect of multiprogrammed systems is that there is not one abstract machine, but many abstract machines. Each running application gets its own abstract machine.

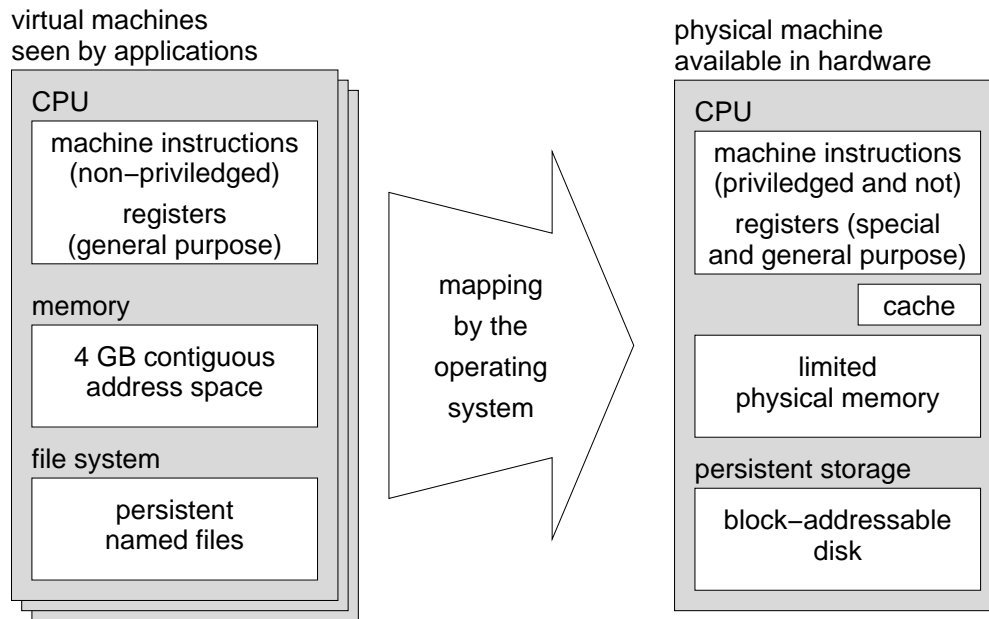
A very important feature of the abstract machines presented to applications is that they are isolated from each other. Part of the abstraction is to hide all those resources that are being used to support other running applications. Each running application therefore sees the system as if it were dedicated to itself. The operating system juggles resources among these competing abstract machines in order to support this illusion. One example of this is scheduling: allocating the CPU to each application in its turn.

Exercise 6 *Can an application nevertheless find out that it is actually sharing the machine with other applications?*

Virtualization allows for decoupling from physical restrictions

The abstract machine presented by the operating system is “better” than the hardware by virtue of supporting more convenient abstractions. Another important improvement is that it is also not limited by the physical resource limitations of the underlying hardware: it is a *virtual* machine. This means that the application does

not access the physical resources directly. Instead, there is a level of indirection, managed by the operating system.



The main reason for using virtualization is to make up for limited resources. If the physical hardware machine at our disposal has only 1GB of memory, and each abstract machine presents its application with a 4GB address space, then obviously a direct mapping from these address spaces to the available memory is impossible. The operating system solves this problem by coupling its resource management functionality with the support for the abstract machines. In effect, it juggles the available resources among the competing virtual machines, trying to hide the deficiency. The specific case of virtual memory is described in Chapter 4.

Virtualization does not necessarily imply abstraction

Virtualization does not necessarily involve abstraction. In recent years there is a growing trend of using virtualization to create multiple copies of the same hardware base. This allows one to run a different operating system on each one. As each operating system provides different abstractions, this decouples the issue of creating abstractions within a virtual machine from the provisioning of resources to the different virtual machines.

The idea of virtual machines is not new. It originated with MVS, the operating system for the IBM mainframes. In this system, time slicing and abstractions are completely decoupled. MVS actually only does the time slicing, and creates multiple *exact copies* of the original physical machine. Then, a single-user operating system called CMS is executed in each virtual machine. CMS provides the abstractions of the user environment, such as a file system.

As each virtual machine is an exact copy of the physical machine, it was also possible to run MVS itself on such a virtual machine. This was useful to debug new versions of the operating system on a running system. If the new version is buggy, only its virtual machine will crash, but the parent MVS will not. This practice continues today, and VMware has been used as a platform for allowing students to experiment with operating systems. We will discuss virtual machine support in Section 9.5.

To read more: History buffs can read more about MVS in the book by Johnson [7].

Things can get complicated

The structure of virtual machines running different operating systems may lead to a confusion in terminology. In particular, the allocation of resources to competing virtual machines may be done by a very thin layer of software that does not really qualify as a full-fledged operating system. Such software is usually called a *hypervisor*.

On the other hand, virtualization can also be done at the application level. A remarkable example is given by VMware. This is actually a user-level application, that runs on top of a conventional operating system such as Linux or Windows. It creates a set of virtual machines that mimic the underlying hardware. Each of these virtual machines can boot an independent operating system, and run different applications. Thus the issue of what exactly constitutes the operating system can be murky. In particular, several layers of virtualization and operating systems may be involved with the execution of a single application.

In these notes we'll ignore such complexities, at least initially. We'll take the (somewhat outdated) view that all the operating system is a monolithic piece of code, which is called the kernel. But in later chapters we'll consider some deviations from this viewpoint.

1.3 Hardware Support for the Operating System

The operating system doesn't need to do everything itself — it gets some help from the hardware. There are even quite a few hardware features that are included specifically for the operating system, and do not serve user applications directly.

The operating system enjoys a privileged execution mode

CPUs typically have (at least) two execution modes: *user* mode and *kernel* mode. User applications run in user mode. The heart of the operating system is called the *kernel*. This is the collection of functions that perform the basic services such as scheduling applications. The kernel runs in kernel mode. Kernel mode is also called *supervisor* mode or *privileged* mode.

The execution mode is indicated by a bit in a special register called the *processor status word* (PSW). Various CPU instructions are only available to software running

in kernel mode, i.e., when the bit is set. Hence these privileged instructions can only be executed by the operating system, and not by user applications. Examples include:

- Instructions to set the interrupt priority level (IPL). This can be used to block certain classes of interrupts from occurring, thus guaranteeing undisturbed execution.
- Instructions to set the hardware clock to generate an interrupt at a certain time in the future.
- Instructions to activate I/O devices. These are used to implement I/O operations on files.
- Instructions to load and store special CPU registers, such as those used to define the accessible memory addresses, and the mapping from each application's virtual addresses to the appropriate addresses in the physical memory.
- Instructions to load and store values from memory directly, without going through the usual mapping. This allows the operating system to access all the memory.

Exercise 7 *Which of the following instructions should be privileged?*

1. *Change the program counter*
2. *Halt the machine*
3. *Divide by zero*
4. *Change the execution mode*

Exercise 8 *You can write a program in assembler that includes privileged instructions. What will happen if you attempt to execute this program?*

Example: levels of protection on Intel processors

At the hardware level, Intel processors provide not two but four levels of protection.

Level 0 is the most protected and intended for use by the kernel.

Level 1 is intended for other, non-kernel parts of the operating system.

Level 2 is offered for device drivers: needy of protection from user applications, but not trusted as much as the operating system proper².

Level 3 is the least protected and intended for use by user applications.

Each data segment in memory is also tagged by a level. A program running in a certain level can only access data that is in the same level or (numerically) higher, that is, has the same or lesser protection. For example, this could be used to protect kernel data structures from being manipulated directly by untrusted device drivers; instead, drivers would be forced to use pre-defined interfaces to request the service they need from the kernel. Programs running in numerically higher levels are also restricted from issuing certain instructions, such as that for halting the machine.

Despite this support, most operating systems (including Unix, Linux, and Windows) only use two of the four levels, corresponding to kernel and user modes.

²Indeed, device drivers are typically buggier than the rest of the kernel [5].

Only predefined software can run in kernel mode

Obviously, software running in kernel mode can control the computer. If a user application was to run in kernel mode, it could prevent other applications from running, destroy their data, etc. It is therefore important to guarantee that user code will never run in kernel mode.

The trick is that when the CPU switches to kernel mode, it also changes the program counter³ (PC) to point at operating system code. Thus user code will never get to run in kernel mode.

Note: kernel mode and superuser

Unix has a special privileged user called the “superuser”. The superuser can override various protection mechanisms imposed by the operating system; for example, he can access other users’ private files. However, this does not imply running in kernel mode. The difference is between restrictions imposed by the operating system software, as part of the operating system services, and restrictions imposed by the hardware.

There are two ways to enter kernel mode: interrupts and system calls.

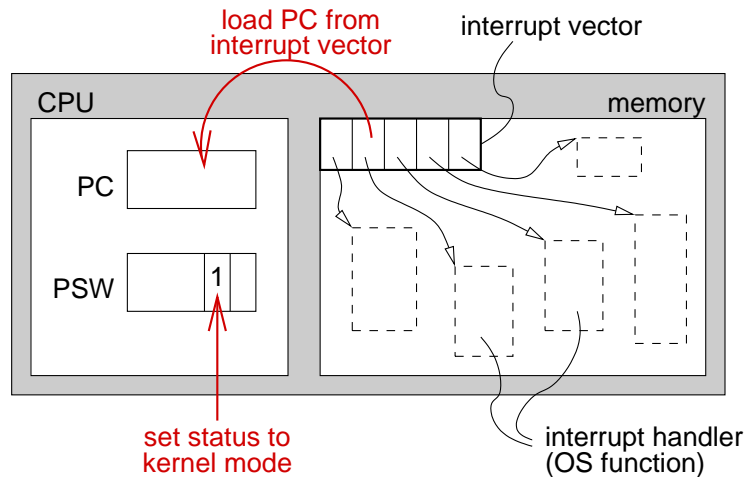
Interrupts cause a switch to kernel mode

Interrupts are special conditions that cause the CPU not to execute the next instruction. Instead, it enters kernel mode and executes an operating system interrupt handler.

But how does the CPU (hardware) know the address of the appropriate kernel function? This depends on what operating system is running, and the operating system might not have been written yet when the CPU was manufactured! The answer to this problem is to use an agreement between the hardware and the software. This agreement is asymmetric, as the hardware was there first. Thus, part of the hardware architecture is the definition of certain features and how the operating system is expected to use them. All operating systems written for this architecture must comply with these specifications.

Two particular details of the specification are the numbering of interrupts, and the designation of a certain physical memory address that will serve as an *interrupt vector*. When the system is booted, the operating system stores the addresses of the interrupt handling functions in the interrupt vector. When an interrupt occurs, the hardware stores the current PSW and PC, and loads the appropriate PSW and PC values for the interrupt handler. The PSW indicates execution in kernel mode. The PC is obtained by using the interrupt number as an index into the interrupt vector, and using the address found there.

³The PC is a special register that holds the address of the next instruction to be executed. This isn’t a very good name. For an overview of this and other special registers see Appendix A.



Note that the hardware does this blindly, using the predefined address of the interrupt vector as a base. It is up to the operating system to actually store the correct addresses in the correct places. If it does not, this is a bug in the operating system.

Exercise 9 *And what happens if such a bug occurs?*

There are two main types of interrupts: asynchronous and internal. *Asynchronous (external) interrupts* are generated by external devices at unpredictable times. Examples include:

- Clock interrupt. This tells the operating system that a certain amount of time has passed. It's handler is the operating system function that keeps track of time. Sometimes, this function also calls the scheduler which might preempt the current application and run another in its place. Without clock interrupts, the application might run forever and monopolize the computer.

Exercise 10 *A typical value for clock interrupt resolution is once every 10 milliseconds. How does this affect the resolution of timing various things?*

- I/O device interrupt. This tells the operating system that an I/O operation has completed. The operating system then wakes up the application that requested the I/O operation.

Internal (synchronous) interrupts occur as a result of an exception condition when executing the current instruction (as this is a result of what the software did, this is sometimes also called a "software interrupt"). This means that the processor cannot complete the current instruction for some reason, so it transfers responsibility to the operating system. There are two main types of exceptions:

- An error condition: this tells the operating system that the current application did something illegal (divide by zero, try to issue a privileged instruction, etc.). The handler is the operating system function that deals with misbehaved applications; usually, it kills them.

- A temporary problem: for example, the process tried to access a page of memory that is not allocated at the moment. This is an error condition that the operating system can handle, and it does so by bringing the required page into memory. We will discuss this in Chapter 4.

Exercise 11 *Can another interrupt occur when the system is still in the interrupt handler for a previous interrupt? What happens then?*

When the handler finishes its execution, the execution of the interrupted application continues where it left off — except if the operating system killed the application or decided to schedule another one.

To read more: Stallings [18, Sect. 1.4] provides a detailed discussion of interrupts, and how they are integrated with the instruction execution cycle.

System calls explicitly ask for the operating system

An application can also explicitly transfer control to the operating system by performing a *system call*. This is implemented by issuing the *trap* instruction. This instruction causes the CPU to enter kernel mode, and set the program counter to a special operating system entry point. The operating system then performs some service on behalf of the application. Technically, this is actually just another (internal) interrupt — but a desirable one that was generated by an explicit request.

As an operating system can have more than a hundred system calls, the hardware cannot be expected to know about all of them (as opposed to interrupts, which are a hardware thing to begin with). The sequence of events leading to the execution of a system call is therefore slightly more involved:

1. The application calls a library function that serves as a wrapper for the system call.
2. The library function (still running in user mode) stores the system call identifier and the provided arguments in a designated place in memory.
3. It then issues the trap instruction.
4. The hardware switches to privileged mode and loads the PC with the address of the operating system function that serves as an entry point for system calls.
5. The entry point function starts running (in kernel mode). It looks in the designated place to find which system call is requested.
6. The system call identifier is used in a big switch statement to find and call the appropriate operating system function to actually perform the desired service. This function starts by retrieving its arguments from where they were stored by the wrapper library function.

When the function completes the requested service a similar sequence happens in reverse:

1. The function that implements the system call stores its return value in a designated place.
2. It then returns to the function implementing the system calls entry point (the big switch).
3. This function calls the instruction that is the opposite of a trap: it returns to user mode and loads the PC with the address of the next instruction in the library function.
4. The library function (running in user mode again) retrieves the system call's return value, and returns it to the application.

Exercise 12 *Should the library of system-call wrappers be part of the distribution of the compiler or of the operating system?*

Typical system calls include:

- Open, close, read, or write to a file.
- Create a new process (that is, start running another application).
- Get some information from the system, e.g. the time of day.
- Request to change the status of the application, e.g. to reduce its priority or to allow it to use more memory.

When the system call finishes, it simply returns to its caller like any other function. Of course, the CPU must return to normal execution mode.

The hardware has special features to help the operating system

In addition to kernel mode and the interrupt vector, computers have various features that are specifically designed to help the operating system.

The most common are features used to help with memory management. Examples include:

- Hardware to translate each virtual memory address to a physical address. This allows the operating system to allocate various scattered memory pages to an application, rather than having to allocate one long continuous stretch of memory.
- “Used” bits on memory pages, which are set automatically whenever any address in the page is accessed. This allows the operating system to see which pages were accessed (bit is 1) and which were not (bit is 0).

We'll review specific hardware features used by the operating system as we need them.

1.4 Roadmap

There are different views of operating systems

An operating system can be viewed in three ways:

- According to the services it provides to users, such as
 - Time slicing.
 - A file system.
- By its programming interface, i.e. its system calls.
- According to its internal structure, algorithms, and data structures.

An operating system is *defined* by its interface — different implementations of the same interface are equivalent as far as users and programs are concerned. However, these notes are organized according to services, and for each one we will detail the internal structures and algorithms used. Occasionally, we will also provide examples of interfaces, mainly from Unix.

To read more: To actually use the services provided by a system, you need to read a book that describes that system's system calls. Good books for Unix programming are Rochkind [15] and Stevens [19]. A good book for Windows programming is Richter [14]. Note that these books teach you about how the operating system looks “from the outside”; in contrast, we will focus on how it is built internally.

Operating system components can be studied in isolation

The main components that we will focus on are

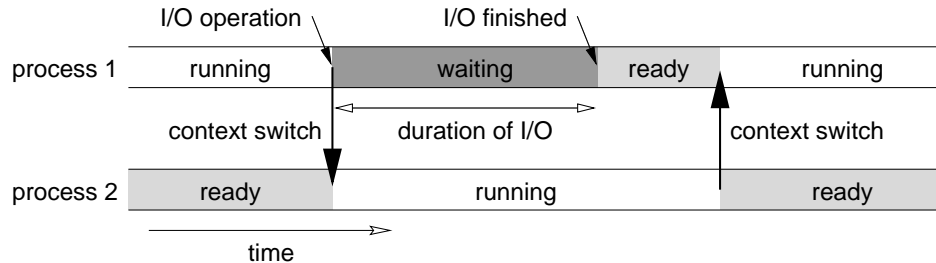
- Process handling. Processes are the agents of processing. The operating system creates them, schedules them, and coordinates their interactions. In particular, multiple processes may co-exist in the system (this is called *multiprogramming*).
- Memory management. Memory is allocated to processes as needed, but there typically is not enough for all, so paging is used.
- File system. Files are an abstraction providing named data repositories based on disks that store individual blocks. The operating system does the bookkeeping.

In addition there are a host of other issues, such as security, protection, accounting, error handling, etc. These will be discussed later or in the context of the larger issues.

But in a living system, the components interact

It is important to understand that in a real system the different components interact all the time. For example,

- When a process performs an I/O operation on a file, it is descheduled until the operation completes, and another process is scheduled in its place. This improves system utilization by overlapping the computation of one process with the I/O of another:



Thus both the CPU and the I/O subsystem are busy at the same time, instead of idling the CPU to wait for the I/O to complete.

- If a process does not have a memory page it requires, it suffers a page fault (this is a type of interrupt). Again, this results in an I/O operation, and another process is run in the meanwhile.
- Memory availability may determine if a new process is started or made to wait.

We will initially ignore such interactions to keep things simple. They will be mentioned later on.

Then there's the interaction among multiple systems

The above paragraphs relate to a single system with a single processor. The first part of these notes is restricted to such systems. The second part of the notes is about distributed systems, where multiple independent systems interact.

Distributed systems are based on networking and communication. We therefore discuss these issues, even though they belong in a separate course on computer communications. We'll then go on to discuss the services provided by the operating system in order to manage and use a distributed environment. Finally, we'll discuss the construction of heterogeneous systems using middleware. While this is not strictly part of the operating system curriculum, it makes sense to mention it here.

And we'll leave a few advanced topics to the end

Finally, there are a few advanced topics that are best discussed in isolation after we already have a solid background in the basics. These topics include

- The structuring of operating systems, the concept of microkernels, and the possibility of extensible systems

- Operating systems and mobile computing, such as disconnected operation of laptops
- Operating systems for parallel processing, and how things change when each user application is composed of multiple interacting processes or threads.

1.5 Scope and Limitations

The kernel is a small part of a distribution

All the things we mentioned so far relate to the operating system kernel. This will indeed be our focus. But it should be noted that in general, when one talks of a certain operating system, one is actually referring to a distribution. For example, a typical Unix distribution contains the following elements:

- The Unix kernel itself. Strictly speaking, this is “the operating system”.
- The `libc` library. This provides the runtime environment for programs written in C. For example, it contains `printf`, the function to format printed output, and `strncpy`, the function to copy strings⁴.
- Various tools, such as `gcc`, the GNU C compiler.
- Many utilities, which are useful programs you may need. Examples include a windowing system, desktop, and shell.

As noted above, we will focus exclusively on the kernel — what it is supposed to do, and how it does it.

You can (and should!) read more elsewhere

These notes should not be considered to be the full story. For example, most operating system textbooks contain historical information on the development of operating systems, which is an interesting story and is not included here. They also contain more details and examples for many of the topics that are covered here.

The main recommended textbooks are Stallings [18], Silberschatz et al. [17], and Tanenbaum [21]. These are general books covering the principles of both theoretical work and the practice in various systems. In general, Stallings is more detailed, and gives extensive examples and descriptions of real systems; Tanenbaum has a somewhat broader scope.

Of course it is also possible to use other operating system textbooks. For example, one approach is to use an educational system to provide students with hands-on experience of operating systems. The best known is Tanenbaum [22], who wrote the

⁴Always use `strncpy`, not `strcpy`!

Minix system specifically for this purpose; the book contains extensive descriptions of Minix as well as full source code (This is the same Tanenbaum as above, but a different book). Nutt [13] uses Linux as his main example. Another approach is to emphasize principles rather than actual examples. Good (though somewhat dated) books in this category include Krakowiak [8] and Finkel [6]. Finally, some books concentrate on a certain class of systems rather than the full scope, such as Tanenbaum's book on distributed operating systems [20] (the same Tanenbaum again; indeed, one of the problems in the field is that a few prolific authors have each written a number of books on related issues; try not to get confused).

In addition, there are a number of books on specific (real) systems. The first and most detailed description of Unix system V is by Bach [1]. A similar description of 4.4BSD was written by McKusick and friends [12]. The most recent is a book on Solaris [10]. Vahalia is another very good book, with focus on advanced issues in different Unix versions [23]. Linux has been described in detail by Card and friends [4], by Beck and other friends [2], and by Bovet and Cesati [3]; of these, the first gives a very detailed low-level description, including all the fields in all major data structures. Alternatively, source code with extensive commentary is available for Unix version 6 (old but a classic) [9] and for Linux [11]. It is hard to find anything with technical details about Windows. The best available is Russinovich and Solomon [16].

While these notes attempt to represent the lectures, and therefore have considerable overlap with textbooks (or, rather, are subsumed by the textbooks), they do have some unique parts that are not commonly found in textbooks. These include an emphasis on understanding system behavior and dynamics. Specifically, we focus on the complementary roles of hardware and software, and on the importance of knowing the expected workload in order to be able to make design decisions and perform reliable evaluations.

Bibliography

- [1] M. J. Bach, *The Design of the UNIX Operating System*. Prentice-Hall, 1986.
- [2] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner, *Linux Kernel Internals*. Addison-Wesley, 2nd ed., 1998.
- [3] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly, 2001.
- [4] R. Card, E. Dumas, and F. Mével, *The Linux Kernel Book*. Wiley, 1998.
- [5] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating system errors". In *18th Symp. Operating Systems Principles*, pp. 73–88, Oct 2001.

- [6] R. A. Finkel, *An Operating Systems Vade Mecum*. Prentice-Hall Inc., 2nd ed., 1988.
- [7] R. H. Johnson, *MVS: Concepts and Facilities*. McGraw-Hill, 1989.
- [8] S. Krakowiak, *Principles of Operating Systems*. MIT Press, 1988.
- [9] J. Lions, *Lions' Commentary on UNIX 6th Edition, with Source Code*. Annabooks, 1996.
- [10] J. Mauro and R. McDougall, *Solaris Internals*. Prentice Hall, 2001.
- [11] S. Maxwell, *Linux Core Kernel Commentary*. Coriolis Open Press, 1999.
- [12] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.
- [13] G. J. Nutt, *Operating Systems: A Modern Perspective*. Addison-Wesley, 1997.
- [14] J. Richter, *Programming Applications for Microsoft Windows*. Microsoft Press, 4th ed., 1999.
- [15] M. J. Rochkind, *Advanced Unix Programming*. Prentice-Hall, 1985.
- [16] M. E. Russinovic and D. A. Solomon, *Microsoft Windows Internals*. Microsoft Press, 4th ed., 2005.
- [17] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. John Wiley & Sons, 7th ed., 2005.
- [18] W. Stallings, *Operating Systems: Internals and Design Principles*. Prentice-Hall, 5th ed., 2005.
- [19] W. R. Stevens, *Advanced Programming in the Unix Environment*. Addison Wesley, 1993.
- [20] A. S. Tanenbaum, *Distributed Operating Systems*. Prentice Hall, 1995.
- [21] A. S. Tanenbaum, *Modern Operating Systems*. Pearson Prentice Hall, 3rd ed., 2008.
- [22] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems: Design and Implementation*. Prentice-Hall, 2nd ed., 1997.
- [23] U. Vahalia, *Unix Internals: The New Frontiers*. Prentice Hall, 1996.