Operating Systems

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
⟨Daniel.Page@bristol.ac.uk⟩

January 25, 2016

Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
   - the associated notes page may be pre-populated with extra, written explaination of material covered in lecture(s), plus
   - anything with a "grey'ed out" header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

Notes:

- Imagine life *without* an operating system, e.g.,

  https://www.youtube.com/watch?v=6v4Juzn10gM

  which details use of EDSAC circa 1951.

Notes:

- Although it depends on your definition of what an operating system is, the so-called initial orders [7] for EDSAC are typically cited as the first instance of one. They were somewhat like a modern BIOS, in the sense of them being a hard-wired sequence of instructions invoked when EDSAC was powered-on: their goal was somewhere between a modern linker and loader.

COMS20001 lecture: week #13

- Imagine life *without* an operating system, e.g.,

  https://www.youtube.com/watch?v=6v4Juzn10gM

  which details use of EDSAC circa 1951.

- Among other things, note that
  1. there was only 1 computer
     - ⇒ crucial to maximise utilisation
  2. computer time was expensive, humans time was inexpensive
     - ⇒ assembly, linking, scheduling *all* done by hand
  3. operator $\neq$ user
     - ⇒ no interactive use
     - ⇒ uni-programming, batch processing
  4. few, fairly simple peripherals
     - ⇒ no third-party vendors
     - ⇒ limited need for protection
  5. *does* have a debugging (cf. core dump) facility
  6. *does* have sub-routine library $\simeq$ system calls

  the latter of which hint at the genesis [7] of modern operating systems.

Notes:

- Although it depends on your definition of what an operating system is, the so-called initial orders [7] for EDSAC are typically cited as the first instance of one. They were somewhat like a modern BIOS, in the sense of them being a hard-wired sequence of instructions invoked when EDSAC was powered-on: their goal was somewhere between a modern linker and loader.

Concepts – Function (1)

- Attempt #1: a (fairly) clean definition might be

> **Definition (operating system)**
>
> **operating system**, *n. the low-level software that supports a computer's basic functions, such as scheduling tasks, controlling peripherals, and allocating storage.*
>
> – *OED*, http://www.oed.com/

but, in practice, we often find that

$$\text{operating system } \textit{distribution} = \{\text{kernel}, \text{system software}, \text{application software}, \ldots\}$$

so, from here on, we'll make the strict assumption that

$$\textbf{operating system} \;\equiv\; \textbf{kernel}.$$

Notes:

- In some contexts, no operating system *could* be the correct choice. For example, on an embedded platform it simply might not make sense; or, it could make sense to have one with selected functionality (e.g., with process, but without memory or file management). In fact, what might be termed a run-time system could, arguably, be classified as an operating system with such selected functionality.
- This definition implicitly assumes the basic functions are provided by a hardware platform (i.e., the computer itself). That is *not* a limit, however. Although a virtual machine such as the JVM satisfies the definition, the platform here is more often software, i.e., another operating system; in a sense, the JVM is arguably more like application software to the host operating system, but an operating system to the user programs it executes.
- Maybe it's obvious, or maybe not, but the term kernel suggests it acts as the core or nucleus of the operating system.
- A reasonable way to define systems software is by saying it is intended to support *other* (typically application) software, rather than to directly support (or provide a service to) a user. Being precise about this can be hard, however. For example, in certain versions of Windows the web-browser (i.e., Internet Explorer) was fundamentally integrated with the operating system; it could not be removed, and offered a range of support to both the user (as a web-browser) and other applications (as a form of shell). In addition, some system software provides a way for the user to configure the operating system: this clearly *does* support the user, but equally is not an application in the same way a word processor, for instance, is.

Concepts – Function (2)

- Attempt #2: the basic idea is that

software {	software	
	- - - - - - - - - - - -	⟵ ***virtual*** **machine interface**
	kernel	
	- - - - - - - - - - - -	⟵ ***physical*** **machine interface**
hardware {	hardware	

so, in practice, the kernel tames complexity via layer of software that delivers

1. **virtualisation** : make it look like resource has features you want
2. **abstraction** : offer appropriate interface to resource
3. **management** : maximise utilisation of and protect resource etc.

plus various standard services.

Notes:

- Hopefully the complexity alluded to is already obvious, but, if not, keep in mind that it stems from above *and* below the kernel. From above, it must deal with software that demands (backward-)compatibility, competes for resources (e.g., space on a disk), has differing (static and dynamic) demands wrt. quality-of-service, and has bugs in (i.e., can fail). From below, it must deal with hardware devices that are diverse wrt. their interface and implementation, operate asynchronously, and has bugs in (i.e., can fail, sometimes in unexpected and uncontrollable ways). Crucially, it is not feasible to test every combination of software and hardware: this is enormously challenging from the perspective of stability and robustness.
- In essence, the kernel is offering a "nice" virtual machine interface to compensate for some form of deficiency in the "not nice" physical machine interface, i.e., the physical hardware. For example, the kernel will normally attempt to optimise for convenience (i.e., via an interface that supports operations closer to what applications want: a good example is the concept of a file system, which avoids details of a given disk), or reliability (it might either prevent errors, e.g., via protection, or allow recovery from them).
- In terms of virtualisation and abstraction of a processor-based hardware platform, the processor ISA *could* be thought of as forming part of the physical machine interface: it provides software (i.e., instructions) with an abstract interface to the underlying micro-architecture, which could be thought of as (conceptually) similar to the kernel providing an interface to the hardware platform more generally.

Concepts – Function (3)

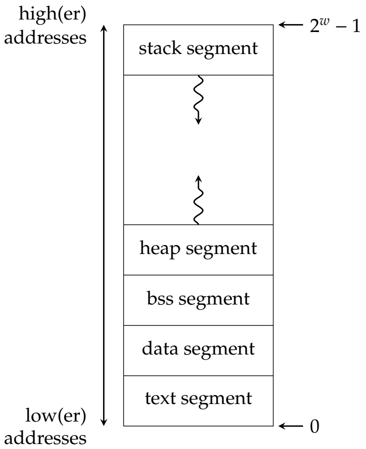
Definition (**program**, **process** and **address space**)

- A **process** is an executing instance of some **program**.
- Each process has an **address space**, i.e., set of addresses it can access.
- Traditionally, such an address space is organised into
  - a text segment (i.e., instructions),
  - a data segment (i.e., initialised, static data), and
  - a bss segment (i.e., uninitialised, static data),

  plus

  - a stack segment, and
  - a heap segment

  whose size can change dynamically.



Notes:

- It is important to stress that this textbook description is of *a typically* not *the definitive* address space. For example, Linux uses the space betweeen heap and stack segments to house an additional segment relating to `mmap`.

Concepts – Function (4)

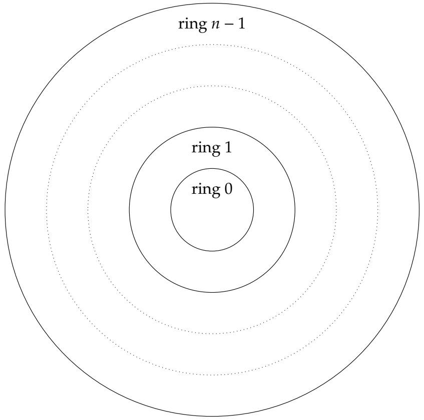
Definition (**user mode**, **kernel mode** and **system calls**)

- To allow the kernel to exert control, there *must* be (hardware enforced) execution privileges.
- In the simplest case there are two modes: the processor can be in
  - **kernel mode** (i.e., a privileged mode), or
  - **user mode** (i.e., a non-privileged mode),

  with

  - **kernel space**, and
  - **user space**

  often used to describe the set of resources kernel and user mode can access.



Notes:

- The terms kernel and user space are fairly loose; quite often they are used to describe the address space associated with the kernel or given user process, but equally it is common to say some X "is in" kernel or user space (implying it can be accessed in kernel or user mode).
- Perhaps a better word that ring would be layer; one can then think of hardware itself *being* the lowest layer. Either way, a concrete example would be classic x86 protection which has 3 rings, with user mode application software executing in ring 3 and the kernel in ring 0.

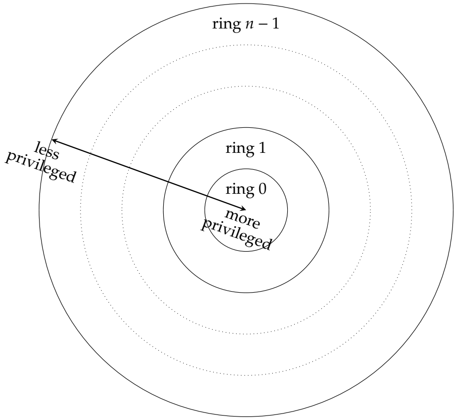
Concepts – Function (4)

Definition (**user mode**, **kernel mode** and **system calls**)

- To allow the kernel to exert control, there *must* be (hardware enforced) execution privileges.
- In the simplest case there are two modes: the processor can be in
  - **kernel mode** (i.e., a privileged mode), or
  - **user mode** (i.e., a non-privileged mode),

  with
  - **kernel space**, and
  - **user space**

  often used to describe the set of resources kernel and user mode can access.



Notes:

- The terms kernel and user space are fairly loose; quite often they are used to describe the address space associated with the kernel or given user process, but equally it is common to say some X "is in" kernel or user space (implying it can be accessed in kernel or user mode).
- Perhaps a better word that ring would be layer; one can then think of hardware itself *being* the lowest layer. Either way, a concrete example would be classic x86 protection which has 3 rings, with user mode application software executing in ring 3 and the kernel in ring 0.

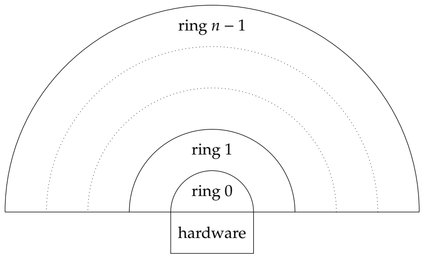
Concepts – Function (4)

Definition (**user mode**, **kernel mode** and **system calls**)

- To allow the kernel to exert control, there *must* be (hardware enforced) execution privileges.
- In the simplest case there are two modes: the processor can be in
  - **kernel mode** (i.e., a privileged mode), or
  - **user mode** (i.e., a non-privileged mode),

  with
  - **kernel space**, and
  - **user space**

  often used to describe the set of resources kernel and user mode can access.



Notes:

- The terms kernel and user space are fairly loose; quite often they are used to describe the address space associated with the kernel or given user process, but equally it is common to say some X "is in" kernel or user space (implying it can be accessed in kernel or user mode).
- Perhaps a better word that ring would be layer; one can then think of hardware itself *being* the lowest layer. Either way, a concrete example would be classic x86 protection which has 3 rings, with user mode application software executing in ring 3 and the kernel in ring 0.

Daniel Page ⟨@bristol.ac.uk⟩
Operating Systems
git # 3ca1bf1 @ 2016-01-22
University of BRISTOL

Concepts – Function (5)

- In summary, the kernel delivers (at least)
  - processor virtualisation
    - $\Rightarrow$ each process among n appears to have dedicated access to the processor
  - memory virtualisation
    - $\Rightarrow$ each process among n appears to have dedicated access to the (entire) memory
    - $\Rightarrow$ the (actual) memory allocated to one process is protected from the $n - 1$ others
  - uniform interface to underlying hardware devices
    - $\Rightarrow$ can shared 1 device among n processes
    - $\Rightarrow$ can enhance raw hardware functionality, e.g., Ethernet NIC to get TCP/IP
    - $\Rightarrow$ can `write` to an on-disk file, irrespective of the disk type

  plus some
  - externalised services, e.g.,
    - program execution,
    - I/O operations,
    - file system manipulation,
    - Inter-Process Communication (IPC),

    and
  - internalised services, e.g.,
    - error handling,
    - resource allocation,
    - protection

Notes:

Concepts – Form (6)

- A kernel is a complex software artefact: this complexity is managed via
  - a layered, modular design, wrt.
    - compartmentalisation of the sub-systems, and
    - the ability to replace modules (at run-time)

    and
  - careful separation between

    1. **mechanism** : concrete implementation of functionality
    2. **policy** : decision re. how functionality should work
    3. **permission** : decision re. when functionality is accessible

  - adherence to standards, namely POSIX [1]

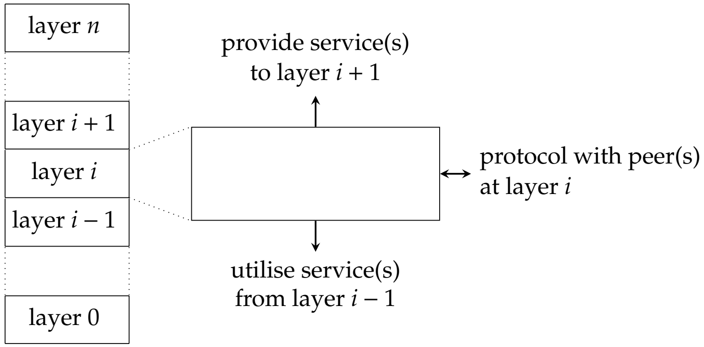
  but *beyond* these principles, various architectures are viable.

Notes:

- Strictly speaking, you could clearly view the permission to access some functionality as a policy; it basically describes enforcement of access rights. This requirement is so ubiquitous, however, it is often placed in a separate category to more general policies for configuration etc.

Concepts – Form (7)

- An idealised architecture will decompose the functionality



but idealised separation of layers is often

- too restrictive, and/or
- too inefficient

so various more concrete compromises are made ...

Notes:

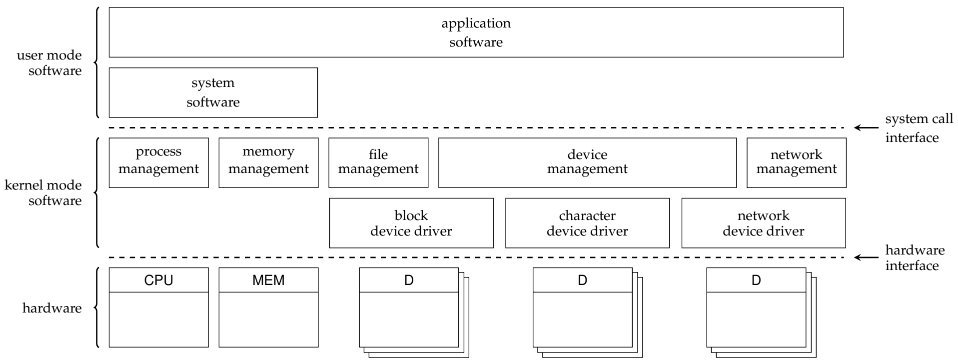
Concepts – Form (8)

Definition (**monolithic kernel** vs. **micro-kernel**)

The two (currently) dominant architectures are

1. a **monolithic kernel** is st. the *entire* kernel executes in kernel space,
2. a **micro-kernel** is st. the *kernel* of the kernel executes in kernel space.

That is, following a somewhat traditional structure:



Notes:

- The concept of a monolithic kernel relates more obviously to the goal of providing a high-level abstraction over all the underlying hardware. In contrast, you could think of micro-kernel as providing a less complete abstraction: by including a minimal set of services in kernel mode, many of the traditional sub-systems can execute in user space (given there is protection between user space processes *anyway*).
- To so-called Tanenbaum-Torvalds debate (very) publicly documents many of the arguments wrt. advantages of monolithic kernel vs. micro-kernels. A good overview is here:

  http://en.wikipedia.org/wiki/Tanenbaum-Torvalds_debate

  A (very) limited comparison would be

  – A monolithic kernel implements services like a function call, via system calls. Bar interrupts, switches between user and kernel mode are mostly limited to the point where the former requires the latter to do something. There is little compartmentalisation, in the sense that anything in the kernel executes in kernel mode so has total access to the platform as a whole; any bug in any part of the kernel could be catastrophic.
  – A micro-kernel implements services more like a client-server system: the user mode software *plus* user mode kernel sub-systems need to communicate with each other via the kernel using some form of IPC. This can represent a significant overhead. There is more compartmentalisation, in the sense that each of the user mode kernel subsystems is protected from the other.

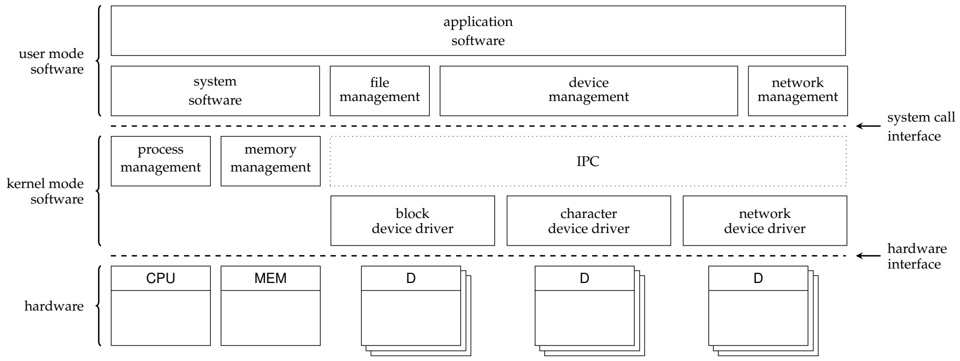
- To illustrate the difficulty of being definitive about what a kernel (or an operating system mode generally) is and is not, consider the concept of an **exokernel** [2, 3]: this specifically refutes the idea a kernel should be a layer of abstraction over underlying hardware. Instead, the such a design attempts to a) provide few(er), low(er)-level abstractions, and so b) focus on efficient management (e.g., protection or multiplexing) of all the associated hardware. Clearly trade-offs exist with this approach versus another, but the fact such an approach is at all viable illustrates the diversity, and hence large design space of options.
- In both architectures, the concept of a Virtual File System (VFS) layer is common. The issue here is that there could be *many* underlying file system *types* mounted on different devices; the VFS layer is an abstraction layer over these differences, allowing the diverse set of file systems to be presented in a uniform way to user space.

Definition (monolithic kernel vs. micro-kernel)

The two (currently) dominant architectures are

1. a **monolithic kernel** is st. the *entire* kernel executes in kernel space,
2. a **micro-kernel** is st. the *kernel* of the kernel executes in kernel space.

That is, following a somewhat traditional structure:



Notes:

- The concept of a monolithic kernel relates more obviously to the goal of providing a high-level abstraction over all the underlying hardware. In contrast, you could think of micro-kernel as providing a less complete abstraction: by including a minimal set of services in kernel mode, many of the traditional sub-systems can execute in user space (given there is protection between user space processes *anyway*).
- To so-called Tanenbaum-Torvalds debate (very) publicly documents many of the arguments wrt. advantages of monolithic kernel vs. micro-kernels. A good overview is here:

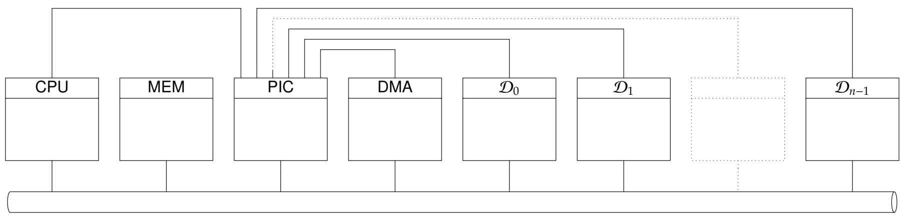
  `http://en.wikipedia.org/wiki/Tanenbaum-Torvalds_debate`

  A (very) limited comparison would be

  - A monolithic kernel implements services like a function call, via system calls. Bar interrupts, switches between user and kernel mode are mostly limited to the point where the former requires the latter to do something. There is little compartmentalisation, in the sense that anything in the kernel executes in kernel mode so has total access to the platform as a whole; any bug in any part of the kernel could be catastrophic.
  - A micro-kernel implements services more like a client-server system: the user mode software *plus* user mode kernel sub-systems need to communicate with each other via the kernel using some form of IPC. This can represent a significant overhead. There is more compartmentalisation, in the sense that each of the user mode kernel subsystems is protected from the other.
- To illustrate the difficulty of being definitive about what a kernel (or an operating system mode generally) is and is not, consider the concept of an **exokernel** [2, 3]: this specifically refutes the idea a kernel should be a layer of abstraction over underlying hardware. Instead, the such a design attempts to a) provide few(er), low(er)-level abstractions, and so b) focus on efficient management (e.g., protection or multiplexing) of all the associated hardware. Clearly trade-offs exist with this approach versus another, but the fact such an approach is at all viable illustrates the diversity, and hence large design space of options.
- In both architectures, the concept of a Virtual File System (VFS) layer is common. The issue here is that there could be *many* underlying file system *types* mounted on different devices; the VFS layer is an abstraction layer over these differences, allowing the diverse set of file systems to be presented in a uniform way to user space.

Conclusions

- Remit:
  - understand a simple(ish) computer system



    wrt. the "life-cycle" of a user mode process under control of an operating system kernel, *but*
  - limit the detail and volume of coverage to fit allocated time.
- *Why*?!
  1. *some* of you may
     - develop an operating system, or
     - work in systems administration,
  2. *most* of you will develop hardware or software that *depends* on an operating system, and
  3. many of the concepts and techniques *generalise*.

Notes:

- An underlying point here is that the topics of computer architecture can *overly* focus on computer processes; all the other components such a processor is attached to are equally or even *more* important wrt. using it. A central challenge within this context is, of course, the fact such components will, inherently operate in a concurrent manner.

References

[1] Standard for information technology - portable operating system interface (POSIX).
Institute of Electrical and Electronics Engineers (IEEE) 1003.1, 2008.
`http://standards.ieee.org/findstds/standard/1003.1-2008.html`.

[2] D.R. Engler and M.F. Kaashoek.
Exterminate all operating system abstractions.
In *Hot Topics in Operating Systems (HotOS-V)*, pages 78–83, 1995.

[3] D.R. Engler, M.F. Kaashoek, and J. O'Toole Jr.
Exokernel: an operating system architecture for application-level resource management.
In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, 1995.

[4] A.S. Tanenbaum and H. Bos.
Chapter 1.5: Operating system concepts.
In *Modern Operating Systems* [6].

[5] A.S. Tanenbaum and H. Bos.
Chapter 1.7: Operating system structure.
In *Modern Operating Systems* [6].

[6] A.S. Tanenbaum and H. Bos.
Modern Operating Systems.
Pearson, 4th edition, 2015.

Notes:

References

[7] D.J. Wheeler.
Programme organization and initial orders for the EDSAC.
Proceedings of the Royal Society A, 202(1071):573–589, 1950.

Daniel Page (bristol.ac.uk)
Operating Systems git # 3ca1bf1 @ 2016-01-22 University of BRISTOL

Notes: