Concurrent Computing (Operating Systems)

Daniel Page

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

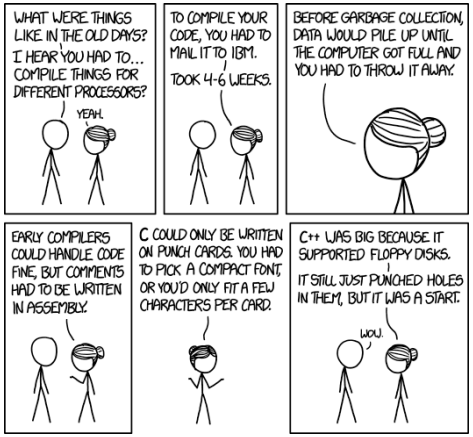
February 9, 2018

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:

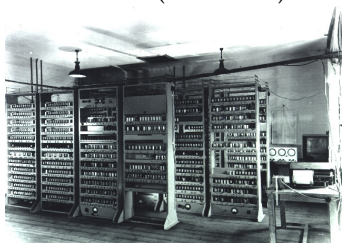


http://xkcd.com/1775/

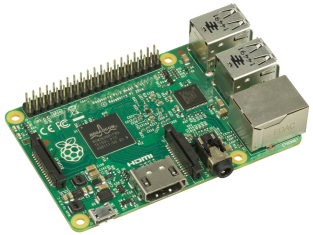
Notes:

COMS20001 lecture: week #13

EDSAC (circa 1949)



RaspberryPi2 (circa 2015)

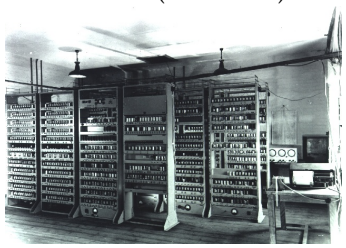


http://www.cl.cam.ac.uk/Relics/jpegs/edsac99-9.jpg
http://en.wikipedia.org/wiki/File:Raspberry-Pi-2-Bare-BR.jpg

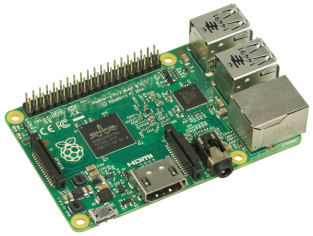
Notes:

- The film at
  http://www.youtube.com/watch?v=6v4Juzn10gM
  should help to bridge the gap between what you already know about the right-hand computer (or some equivalent) and the left-hand computer, i.e., EDSAC.
  One reason to focus on EDSAC in particular, is that it (arguably) had the first, embryonic instance of what we now term an operating system. The EDSAC initial orders [8] were implemented somewhat like a modern BIOS (in the sense of them being a hard-wired sequence of instructions invoked at power-on), and tasked with initialising or bootstrapping the execution of a program, so, more precisely, acting as a mixture of modern linker and loader.
- Rather than technology and societal trends, you might as well describe the changes as relating to a) what a computer *is*, and b) how we *use* computers.
  In more descriptive terms, consider the EDSAC case: computers of this era were characterised by their supporting 1 user at a time and 1 program at a time, with relatively few, relatively simple peripheral devices attached to the computer (that were typically developed and maintained in-house). The user (often an *operator*, working on behalf of a programmer) *manually* undertook tasks such as loading programs from punched card or paper tape; although progress could be monitored (e.g., via a display of memory content), execution was typically non-interactive. There was typically a lot of under-utilisation (i.e., idle time), as a result of a) the relative high latency of I/O between computer and peripherals, and b) the need for manual intervention to manage execution. Within that era computer time was expensive but *human* time was inexpensive, with under-utilisation therefore a significant problem.
  Now considering a modern alternative to EDSAC, almost every aspect of the description above has changed: each change tends to add motivation for operating systems to exist. Take the switch from supporting the execution of 1 program to n (concurrent) programs as an example. The technique supporting this feature is multi-programming, which in turn stemmed multi-tasking as a result of the change in emphasis to those programs being interactive; the *operating system* is what allows this to happen in a largely transparent manner.

EDSAC (circa 1949)



RaspberryPi2 (circa 2015)



- Question: what's changed?

http://www.cl.cam.ac.uk/Relics/jpegs/edsac99-9.jpg

http://en.wikipedia.org/wiki/File:Raspberry-Pi-2-Bare-BR.jpg

Notes:

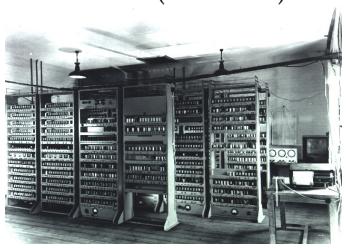
- The film at

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

  should help to bridge the gap between what you already know about the right-hand computer (or some equivalent) and the left-hand computer, i.e., EDSAC.
  One reason to focus on EDSAC in particular, is that it (arguably) had the first, embryonic instance of what we now term an operating system. The EDSAC initial orders [8] were implemented somewhat like a modern BIOS (in the sense of them being a hard-wired sequence of instructions invoked at power-on), and tasked with initialising or bootstrapping the execution of a program, so, more precisely, acting as a mixture of modern linker and loader.
- Rather than technology and societal trends, you might as well describe the changes as relating to a) what a computer *is*, and b) how we *use* computers.
  In more descriptive terms, consider the EDSAC case: computers of this era were characterised by their supporting 1 user at a time and 1 program at a time, with relatively few, relatively simple peripheral devices attached to the computer (that were typically developed and maintained in-house). The user (often an *operator*, working on behalf of a programmer) *manually* undertook tasks such as loading programs from punched card or paper tape; although progress could be monitored (e.g., via a display of memory content), execution was typically non-interactive. There was typically a lot of under-utilisation (i.e., idle time), as a result of a) the relative high latency of I/O between computer and peripherals, and b) the need for manual intervention to manage execution. Within that era computer time was expensive but *human* time was inexpensive, with under-utilisation therefore a significant problem.
  Now considering a modern alternative to EDSAC, almost every aspect of the description above has changed: each change tends to add motivation for operating systems to exist. Take the switch from supporting the execution of 1 program to n (concurrent) programs as an example. The technique supporting this feature is multi-programming, which in turn stemmed multi-tasking as a result of the change in emphasis to those programs being interactive; the *operating system* is what allows this to happen in a largely transparent manner.

COMS20001 lecture: week #13

EDSAC (circa 1949)



RaspberryPi2 (circa 2015)

- Question: what's changed?
- Answer:
  1. *technology* trends:
     - Moore's Law, Joy's Law, Wirth's Law, Koomey's Law, Nielsen's Law, Metcalfe's Law,
     - volume and diversity of (peripheral) devices,
     - ...
  2. *societal* trends:
     - use-cases, e.g., mobile vs. not, or interactive vs. not,
     - volume and diversity of data,
     - users-per-computer, users-per-resource (and so on) ratios,
     - ...

http://www.cl.cam.ac.uk/Relics/jpegs/edsac99-9.jpg

http://en.wikipedia.org/wiki/File:Raspberry-Pi-2-Bare-BR.jpg

© Daniel Page (<dsp@bristol.ac.uk>)
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08 University of BRISTOL

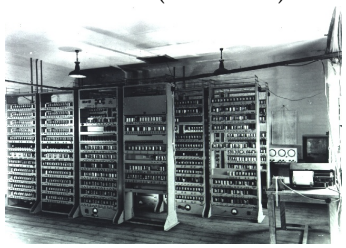
Notes:

- The film at

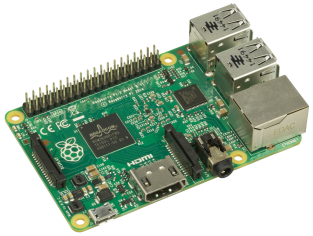
  http://www.youtube.com/watch?v=6v4Juzn10gM

  should help to bridge the gap between what you already know about the right-hand computer (or some equivalent) and the left-hand computer, i.e., EDSAC.
  One reason to focus on EDSAC in particular, is that it (arguably) had the first, embryonic instance of what we now term an operating system. The EDSAC initial orders [8] were implemented somewhat like a modern BIOS (in the sense of them being a hard-wired sequence of instructions invoked at power-on), and tasked with initialising or bootstrapping the execution of a program, so, more precisely, acting as a mixture of modern linker and loader.
- Rather than technology and societal trends, you might as well describe the changes as relating to a) what a computer *is*, and b) how we *use* computers.
  In more descriptive terms, consider the EDSAC case: computers of this era were characterised by their supporting 1 user at a time and 1 program at a time, with relatively few, relatively simple peripheral devices attached to the computer (that were typically developed and maintained in-house). The user (often an *operator*, working on behalf of a programmer) *manually* undertook tasks such as loading programs from punched card or paper tape; although progress could be monitored (e.g., via a display of memory content), execution was typically non-interactive. There was typically a lot of under-utilisation (i.e., idle time), as a result of a) the relative high latency of I/O between computer and peripherals, and b) the need for manual intervention to manage execution. Within that era computer time was expensive but *human* time was inexpensive, with under-utilisation therefore a significant problem.
  Now considering a modern alternative to EDSAC, almost every aspect of the description above has changed: each change tends to add motivation for operating systems to exist. Take the switch from supporting the execution of 1 program to n (concurrent) programs as an example. The technique supporting this feature is multi-programming, which in turn stemmed multi-tasking as a result of the change in emphasis to those programs being interactive; the *operating system* is what allows this to happen in a largely transparent manner.

EDSAC (circa 1949)



RaspberryPi2 (circa 2015)



- Question: what's changed?
- Answer: *massively* increased **complexity**
  - complex use-cases,
  - complex quality metrics and requirements,
  - complex software,
  - complex hardware,
  - complex interactions and failure modes,
  - ...

  which form motivation for utilising an **operating system**.

http://www.cl.cam.ac.uk/Relics/jpegs/edsac99-9.jpg

http://en.wikipedia.org/wiki/File:Raspberry-Pi-2-Bare-BR.jpg

Notes:

- The film at

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

  should help to bridge the gap between what you already know about the right-hand computer (or some equivalent) and the left-hand computer, i.e., EDSAC.
  One reason to focus on EDSAC in particular, is that it (arguably) had the first, embryonic instance of what we now term an operating system. The EDSAC initial orders [8] were implemented somewhat like a modern BIOS (in the sense of them being a hard-wired sequence of instructions invoked at power-on), and tasked with initialising or bootstrapping the execution of a program, so, more precisely, acting as a mixture of modern linker and loader.
- Rather than technology and societal trends, you might as well describe the changes as relating to a) what a computer *is*, and b) how we *use* computers.
  In more descriptive terms, consider the EDSAC case: computers of this era were characterised by their supporting 1 user at a time and 1 program at a time, with relatively few, relatively simple peripheral devices attached to the computer (that were typically developed and maintained in-house). The user (often an *operator*, working on behalf of a programmer) *manually* undertook tasks such as loading programs from punched card or paper tape; although progress could be monitored (e.g., via a display of memory content), execution was typically non-interactive. There was typically a lot of under-utilisation (i.e., idle time), as a result of a) the relative high latency of I/O between computer and peripherals, and b) the need for manual intervention to manage execution. Within that era computer time was expensive but *human* time was inexpensive, with under-utilisation therefore a significant problem.
  Now considering a modern alternative to EDSAC, almost every aspect of the description above has changed: each change tends to add motivation for operating systems to exist. Take the switch from supporting the execution of 1 program to n (concurrent) programs as an example. The technique supporting this feature is multi-programming, which in turn stemmed multi-tasking as a result of the change in emphasis to those programs being interactive; the *operating system* is what allows this to happen in a largely transparent manner.

Concepts (1)

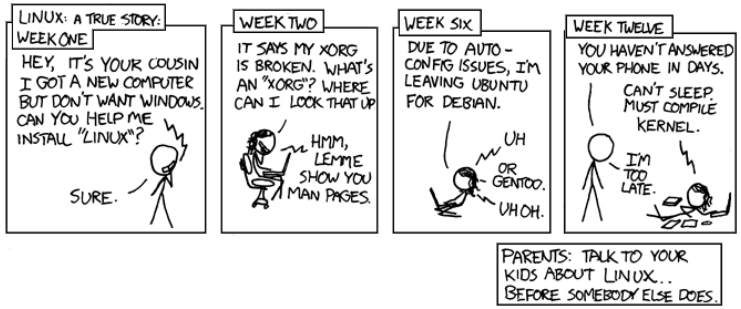
What *is* an operating system?

- Question: what *is* an operating system?

Notes:

- Maybe it's obvious, or maybe not, but the term kernel suggests it acts as the core or nucleus of the operating system.
- 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.
- A reasonable way to define system 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.
  However, apart from specific cases where a distinction is useful and/or necessary, from here on we focus on application software: we assume any program executed in user mode can be generically described as an application.
- In essence, the kernel is providing a "nice" virtual machine interface to compensate for some form of deficiency in the "not so nice" physical machine interface; it could be viewed as providing an *extension* of the native, physical hardware. Overall, it is reasonable to define the kernel differently depending on whether the perspective is top-down or bottom-up: from the former it is essentially abstraction of hardware, whereas from the latter it acts as a manager for the hardware resources available.
  However, it is crucial to remember that, as the diagram suggests, the kernel is fundamentally just software. Although it has requirements and responsibilities that differ from application and system software, it is still, fundamentally, a program that executes on the processor.
- It is attractive to consider a general case wrt. resources: pre-emptible resources as those the kernel can revoke access to (i.e., take away from) having previously allocated them, with non pre-emptible resources being those it cannot (i.e., those which must be voluntarily relinquished). As such, including allocation alone as part of the resource management role is a little misleading: it is more accurate to say it includes allocation (i.e., which process *obtains* access to the resource) and scheduling (i.e., how long a process *retains* access to resource for, resp. if/when access is revoked). Keep in mind that while the latter is frequently interpreted as relating to processes, and hence the processor as a resource, it actually relates to *any* resource under this general definition.

- Question: what *is* an operating system?
- Answer: maybe you define it via *experience*



but that's not *so* useful.

http://xkcd.com/456/

Notes:

- Maybe it's obvious, or maybe not, but the term kernel suggests it acts as the core or nucleus of the operating system.
- 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.
- A reasonable way to define system 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.
  However, apart from specific cases where a distinction is useful and/or necessary, from here on we focus on application software: we assume any program executed in user mode can be generically described as an application.
- In essence, the kernel is providing a "nice" virtual machine interface to compensate for some form of deficiency in the "not so nice" physical machine interface; it could be viewed as providing an *extension* of the native, physical hardware. Overall, it is reasonable to define the kernel differently depending on whether the perspective is top-down or bottom-up: from the former it is essentially abstraction of hardware, whereas from the latter it acts as a manager for the hardware resources available.
  However, it is crucial to remember that, as the diagram suggests, the kernel is fundamentally just software. Although it has requirements and responsibilities that differ from application and system software, it is still, fundamentally, a program that executes on the processor.
- It is attractive to consider a general case wrt. resources: pre-emptible resources as those the kernel can revoke access to (i.e., take away from) having previously allocated them, with non pre-emptible resources being those it cannot (i.e., those which must be voluntarily relinquished). As such, including allocation alone as part of the resource management role is a little misleading: it is more accurate to say it includes allocation (i.e., which process *obtains* access to the resource) and scheduling (i.e., how long a process *retains* access to resource for, resp. if/when access is revoked). Keep in mind that while the latter is frequently interpreted as relating to processes, and hence the processor as a resource, it actually relates to *any* resource under this general definition.

Concepts (1)
What *is* an operating system?

- Question: what *is* an operating system?
- Answer: a more technical definition might be

Definition

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

operating system *distribution* = {kernel, system software, application software, ...}

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

operating system ≡ **kernel**.

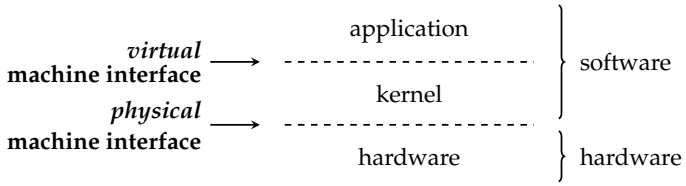
© Daniel Page (<redacted>)
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08 University of BRISTOL

Notes:

- Maybe it's obvious, or maybe not, but the term kernel suggests it acts as the core or nucleus of the operating system.
- 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.
- A reasonable way to define system 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.
  However, apart from specific cases where a distinction is useful and/or necessary, from here on we focus on application software: we assume any program executed in user mode can be generically described as an application.
- In essence, the kernel is providing a "nice" virtual machine interface to compensate for some form of deficiency in the "not so nice" physical machine interface; it could be viewed as providing an *extension* of the native, physical hardware. Overall, it is reasonable to define the kernel differently depending on whether the perspective is top-down or bottom-up: from the former it is essentially abstraction of hardware, whereas from the latter it acts as a manager for the hardware resources available.
  However, it is crucial to remember that, as the diagram suggests, the kernel is fundamentally just software. Although it has requirements and responsibilities that differ from application and system software, it is still, fundamentally, a program that executes on the processor.
- It is attractive to consider a general case wrt. resources: pre-emptible resources as those the kernel can revoke access to (i.e., take away from) having previously allocated them, with non pre-emptible resources being those it cannot (i.e., those which must be voluntarily relinquished). As such, including allocation alone as part of the resource management role is a little misleading: it is more accurate to say it includes allocation (i.e., which process *obtains* access to the resource) and scheduling (i.e., how long a process *retains* access to resource for, resp. if/when access is revoked). Keep in mind that while the latter is frequently interpreted as relating to processes, and hence the processor as a resource, it actually relates to *any* resource under this general definition.

Concepts (1)
What *is* an operating system?

- Question: what *is* ~~an operating system~~ a kernel?
- Answer: a more technical definition might be



so the kernel is a layer of software that delivers

1. **management** : allocate, multiplex, and protect access to resource
2. **abstraction** : offer appropriate interface to resource
3. **virtualisation** : make it look like resource has features you want

plus various standard services.

Notes:

- Maybe it's obvious, or maybe not, but the term kernel suggests it acts as the core or nucleus of the operating system.
- 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.
- A reasonable way to define system 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.
  However, apart from specific cases where a distinction is useful and/or necessary, from here on we focus on application software: we assume any program executed in user mode can be generically described as an application.
- In essence, the kernel is providing a "nice" virtual machine interface to compensate for some form of deficiency in the "not so nice" physical machine interface; it could be viewed as providing an *extension* of the native, physical hardware. Overall, it is reasonable to define the kernel differently depending on whether the perspective is top-down or bottom-up: from the former it is essentially abstraction of hardware, whereas from the latter it acts as a manager for the hardware resources available.
  However, it is crucial to remember that, as the diagram suggests, the kernel is fundamentally just software. Although it has requirements and responsibilities that differ from application and system software, it is still, fundamentally, a program that executes on the processor.
- It is attractive to consider a general case wrt. resources: pre-emptible resources as those the kernel can revoke access to (i.e., take away from) having previously allocated them, with non pre-emptible resources being those it cannot (i.e., those which must be voluntarily relinquished). As such, including allocation alone as part of the resource management role is a little misleading: it is more accurate to say it includes allocation (i.e., which process *obtains* access to the resource) and scheduling (i.e., how long a process *retains* access to resource for, resp. if/when access is revoked). Keep in mind that while the latter is frequently interpreted as relating to processes, and hence the processor as a resource, it actually relates to *any* resource under this general definition.

Concepts (2)
Fundamental abstractions

- Question: what is an abstraction?

Notes:

- A policy could be viewed as a commitment to there being a mechanism for some functionality. Separating the two therefore allows the mechanism to be *transparently* changed, for example to a) meet design goals and/or b) cope with emergent behaviour (e.g., alter the way a resource is managed based on high demand). Although it overloads the term a little, it is also reasonable to think of policy as dictating how underlying mechanisms work. For example, we might allow the user to select a scheduling policy for processes which favours those of type X vs. those of type Y; this does not necessarily specify how the scheduler operates concretely, but allows control over what the intended behaviour is abstractly. Again, separating the two is attractive because it allows selection of the former to suit; this also implies not hard-coding such policies in the kernel, which clearly disallows such selection.
- Among various examples of the separation between policy and mechanism, consider a function for sorting integers: the interface is captured by the function prototype

```
void sort( int* x, int n );
```

  The policy or semantics of `sort` may be st. having called it, the n-element array `x` is sorted in ascending order. This says *nothing* about the mechanism used, however. Put another way, the same semantics can be realised using the quick-sort algorithm in one implementation and bubble-sort in another.

- Question: what is an abstraction?
- Answer: a method of managing (e.g., hiding) complexity.

Notes:

- A policy could be viewed as a commitment to there being a mechanism for some functionality. Separating the two therefore allows the mechanism to be *transparently* changed, for example to a) meet design goals and/or b) cope with emergent behaviour (e.g., alter the way a resource is managed based on high demand). Although it overloads the term a little, it is also reasonable to think of policy as dictating how underlying mechanisms work. For example, we might allow the user to select a scheduling policy for processes which favours those of type X vs. those of type Y; this does not necessarily specify how the scheduler operates concretely, but allows control over what the intended behaviour is abstractly. Again, separating the two is attractive because it allows selection of the former to suit; this also implies not hard-coding such policies in the kernel, which clearly disallows such selection.
- Among various examples of the separation between policy and mechanism, consider a function for sorting integers: the interface is captured by the function prototype

```
void sort( int* x, int n );
```

The policy or semantics of `sort` may be st. having called it, the `n`-element array `x` is sorted in ascending order. This says *nothing* about the mechanism used, however. Put another way, the same semantics can be realised using the quick-sort algorithm in one implementation and bubble-sort in another.

Concepts (2)
Fundamental abstractions

- Question: what is an abstraction?
- Answer: a method of managing (e.g., hiding) complexity.
- Question: what constitutes a *good* abstraction?

Notes:

- A policy could be viewed as a commitment to there being a mechanism for some functionality. Separating the two therefore allows the mechanism to be *transparently* changed, for example to a) meet design goals and/or b) cope with emergent behaviour (e.g., alter the way a resource is managed based on high demand). Although it overloads the term a little, it is also reasonable to think of policy as dictating how underlying mechanisms work. For example, we might allow the user to select a scheduling policy for processes which favours those of type X vs. those of type Y; this does not necessarily specify how the scheduler operates concretely, but allows control over what the intended behaviour is abstractly. Again, separating the two is attractive because it allows selection of the former to suit; this also implies not hard-coding such policies in the kernel, which clearly disallows such selection.
- Among various examples of the separation between policy and mechanism, consider a function for sorting integers: the interface is captured by the function prototype

```
void sort( int* x, int n );
```

The policy or semantics of `sort` may be st. having called it, the `n`-element array `x` is sorted in ascending order. This says *nothing* about the mechanism used, however. Put another way, the same semantics can be realised using the quick-sort algorithm in one implementation and bubble-sort in another.

© Daniel Page ⟨csdsp@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08 University of BRISTOL

- Question: what is an abstraction?
- Answer: a method of managing (e.g., hiding) complexity.
- Question: what constitutes a *good* abstraction?
- Answer: one that affords

1. simplification by
   - hiding unattractive properties,
   - adding new functionality, and/or
   - organising information
2. an separation [9] between (or decoupling of)

$$\begin{aligned} \text{policy} &\equiv \text{how the interface works abstractly (i.e., } \textit{semantics}\text{)} \\ \text{mechanism} &\equiv \text{how the interface works concretely (i.e., } \textit{implementation}\text{)} \end{aligned}$$

Notes:

- A policy could be viewed as a commitment to there being a mechanism for some functionality. Separating the two therefore allows the mechanism to be *transparently* changed, for example to a) meet design goals and/or b) cope with emergent behaviour (e.g., alter the way a resource is managed based on high demand). Although it overloads the term a little, it is also reasonable to think of policy as dictating how underlying mechanisms work. For example, we might allow the user to select a scheduling policy for processes which favours those of type X vs. those of type Y; this does not necessarily specify how the scheduler operates concretely, but allows control over what the intended behaviour is abstractly. Again, separating the two is attractive because it allows selection of the former to suit; this also implies not hard-coding such policies in the kernel, which clearly disallows such selection.
- Among various examples of the separation between policy and mechanism, consider a function for sorting integers: the interface is captured by the function prototype

  ```
  void sort( int* x, int n );
  ```

  The policy or semantics of `sort` may be st. having called it, the `n`-element array `x` is sorted in ascending order. This says *nothing* about the mechanism used, however. Put another way, the same semantics can be realised using the quick-sort algorithm in one implementation and bubble-sort in another.

Definition

An **address space** is abstraction of memory.

Notes:

- It is important to stress that this textbook description is of *a typically* not *the definitive* address space. Within it, we can identify
  - 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.

Definition

An **address space** is abstraction of memory.

- Question:
  - unattractive properties : **?**
  - new capabilities : **?**
  - organise information : **?**

Notes:

- It is important to stress that this textbook description is of *a typically* not *the definitive* address space. Within it, we can identify
  - 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.

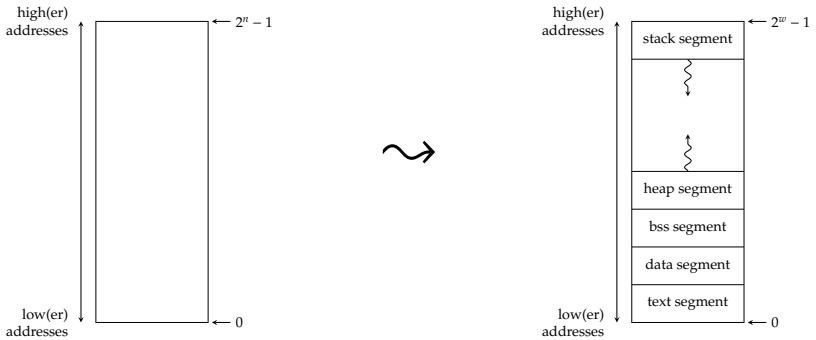
Concepts (3)
Fundamental abstractions

Definition

An **address space** is abstraction of memory.

- Question:
  - unattractive properties : 1 memory vs. n processes, fixed size, sparse, ...
  - new capabilities : virtualisation, protection, ...
  - organise information : ...

  st. it represents a (structured) set of accessible addresses, e.g.,



Notes:

- It is important to stress that this textbook description is of *a typically* not *the definitive* address space. Within it, we can identify
  - 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.

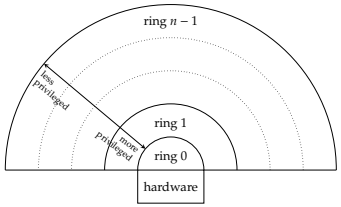
© Daniel Page ⟨dan@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08 University of BRISTOL

Definition

A **process** is abstraction of the processor.

Notes:

- It is common to term *and* illustrate the various privilege levels as (a set of) privilege rings



  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; ARM uses a name- rather than number-based scheme to identify the layers, e.g., user and supervisor mode vs. ring 3 and 0.

- 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).

Concepts (4)
Fundamental abstractions

Definition

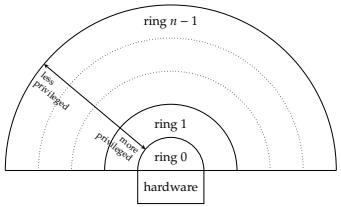
A **process** is abstraction of the processor.

- Question:
  - unattractive properties : **?**
  - new capabilities : **?**
  - organise information : **?**

© Daniel Page ⟨cdp@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08 University of BRISTOL

Notes:

- It is common to term *and* illustrate the various privilege levels as (a set of) privilege rings



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; ARM uses a name- rather than number-based scheme to identify the layers, e.g., user and supervisor mode vs. ring 3 and 0.

- 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).

Definition

A **process** is abstraction of the processor.

- Question:
  - unattractive properties : 1 processor vs. n processes, ...
  - new capabilities : virtualisation, protection, communication, ...
  - organise information : execution context(s), address space, resources, ...

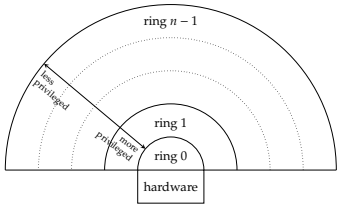
st. it represents an executing instance of some program, acting as

1. a "container" to organise other abstractions, and
2. a boundary for privilege or protection, which demands hardware support:
   - in the simplest case, two **processor modes** exist
     a. **kernel mode** (i.e., a privileged mode), *or*
     b. **user mode** (i.e., a non-privileged mode),
   - the terms
     a. **kernel space**, and
     b. **user space**

   describe the associated set of accessible resources, and
   - switching *between* modes is carefully controlled.

Notes:

- It is common to term *and* illustrate the various privilege levels as (a set of) privilege rings



  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; ARM uses a name- rather than number-based scheme to identify the layers, e.g., user and supervisor mode vs. ring 3 and 0.
- 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).

Definition

A **file** is abstraction of the disk.

Notes:

Definition

A **file** is abstraction of the disk.

- Question:
  - unattractive properties : **?**
  - new capabilities : **?**
  - organise information : **?**

Notes:

Concepts (5)
Fundamental abstractions

Definition

A **file** is abstraction of the disk.

- Question:
  - unattractive properties : reliability, latency, fragmentation, ...
  - new capabilities : identifiers, hierarchy, dynamic size, ...
  - organise information : access control, ...

or, actually, ... only *sort of*:

- UNIX uses what is often termed an "everything is a file" philosophy [1],
- *any* stream of bytes has a file-like interface, e.g.,
  - persistent storage,
  - pseudo-files (e.g., `/dev/random`),
  - I/O with devices (e.g., `/dev/sda`),
  - kernel configuration (e.g., `/proc`),
  - ...

Notes:

Definition

A **socket** is abstraction of the network.

Notes:

Concepts (6)
Fundamental abstractions

Definition

A **socket** is abstraction of the network.

- Question:
  - unattractive properties : **?**
  - new capabilities : **?**
  - organise information : **?**

Notes:

Definition

A **socket** is abstraction of the network.

- Question:
  - unattractive properties : reliability, latency, topology, ...
  - new capabilities : name look-up, packet filtering, ...
  - organise information : ...

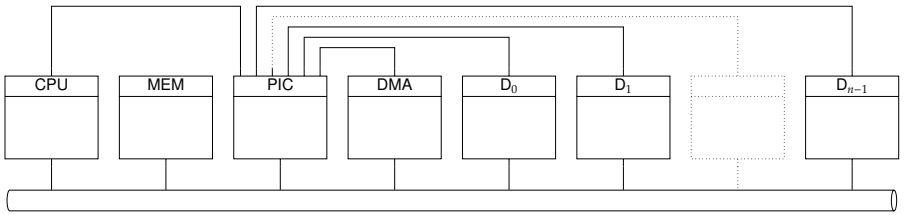
  or, actually, ... only *sort of*:
  - in reality, there are several *types* of socket
    - domain sockets $\rightsquigarrow$ local communication
    - internet domain sockets $\rightsquigarrow$ remote communication

    which are more like a generalisation of file abstraction,
  - abstraction of the network exists via *multiple* interfaces
    - OSI layers $4+5 \rightsquigarrow$ TCP $\Rightarrow$ sockets
    - OSI layer $3 \rightsquigarrow$ IP $\Rightarrow$ kernel configuration
    - OSI layer $2 \rightsquigarrow$ NIC $\Rightarrow$ kernel network interface
    - OSI layer $1 \rightsquigarrow$ NIC $\Rightarrow$ kernel device drivers

Notes:

Conclusions

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



  and how an operating system kernel supports processes executing on it, *but*
  - limit the detail and volume of coverage to fit allocated time.

Notes:

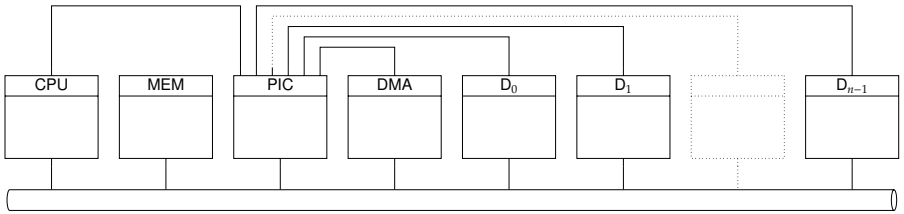
- An underlying point here is that the topics of computer architecture can *overly* focus on computer processors; all the other components such a processor is attached to, are equally important wrt. actually using it to do something useful! You don't need to simply take *my* word for it: numerous other people have tried to articulate the value of studying operating systems, an example of which is

  http://blog.regehr.org/archives/164

- The point about learning from experience is an important one: operating system kernel designs and implementations a) represent some of the largest investments in human effort (and thus are among the largest development projects undertaken), a) are among the most long-lived of *all* those we use (i.e., they have endured over lengthy periods of use and development), and b) deal with some of the most complex and demanding challenges. As such, there is a vast amount one can learn from instances where they have succeeded *and* failed. By analogy, you could think of this as the same motivation as studying "masters" of art or literature.
- There are, roughly speaking, two ways to deliver this sort of material: we might take either a) an "architectural" approach, focusing on high-level concepts and structure in the abstract, with few, if any, concrete examples and little technical detail, or b) an "implementation" approach, focusing on low-level, concrete examples and technical detail, and using them to identify common concepts and structure. In reality a mixture of the two is also viable, but we adopt an approach closer to the latter.

Conclusions

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



  and how an operating system kernel supports processes executing on it, *but*
  - limit the detail and volume of coverage to fit allocated time.
- *Why*?!

1. technical curiosity:
   - to explain how things work,
   - to extract *general* principles from experience.
2. practical utility:
   - *some* of you will develop an operating system,
   - *some* of you will work in system administration,
   - *most* of you will develop hardware or software that *depends* on an operating system.

Notes:

- An underlying point here is that the topics of computer architecture can *overly* focus on computer processors; all the other components such a processor is attached to, are equally important wrt. actually using it to do something useful! You don't need to simply take *my* word for it: numerous other people have tried to articulate the value of studying operating systems, an example of which is

  http://blog.regehr.org/archives/164

- The point about learning from experience is an important one: operating system kernel designs and implementations a) represent some of the largest investments in human effort (and thus are among the largest development projects undertaken), a) are among the most long-lived of *all* those we use (i.e., they have endured over lengthy periods of use and development), and b) deal with some of the most complex and demanding challenges. As such, there is a vast amount one can learn from instances where they have succeeded *and* failed. By analogy, you could think of this as the same motivation as studying "masters" of art or literature.
- There are, roughly speaking, two ways to deliver this sort of material: we might take either a) an "architectural" approach, focusing on high-level concepts and structure in the abstract, with few, if any, concrete examples and little technical detail, or b) an "implementation" approach, focusing on low-level, concrete examples and technical detail, and using them to identify common concepts and structure. In reality a mixture of the two is also viable, but we adopt an approach closer to the latter.

Additional Reading

- *Wikipedia: Operating system*. URL: http://en.wikipedia.org/wiki/Operating_system.
- *Wikipedia: Kernel*. URL: http://en.wikipedia.org/wiki/Kernel_(operating_system).
- A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 1: Introduction". In: *Operating System Concepts*. 9th ed. Wiley, 2014.
- A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 2: System structures". In: *Operating System Concepts*. 9th ed. Wiley, 2014.
- A.S. Tanenbaum and H. Bos. "Chapter 1.1: What is an operating system". In: *Modern Operating Systems*. 4th ed. Pearson, 2015.
- A.S. Tanenbaum and H. Bos. "Chapter 1.5: Operating system concepts". In: *Modern Operating Systems*. 4th ed. Pearson, 2015.

Notes:

References

[1] *Wikipedia: Everything is a file*. URL: http://en.wikipedia.org/wiki/Everything_is_a_file (see pp. 43, 45, 47).

[2] *Wikipedia: Kernel*. URL: http://en.wikipedia.org/wiki/Kernel_(operating_system) (see p. 59).

[3] *Wikipedia: Operating system*. URL: http://en.wikipedia.org/wiki/Operating_system (see p. 59).

[4] A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 1: Introduction". In: *Operating System Concepts*. 9th ed. Wiley, 2014 (see p. 59).

[5] A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 2: System structures". In: *Operating System Concepts*. 9th ed. Wiley, 2014 (see p. 59).

[6] A.S. Tanenbaum and H. Bos. "Chapter 1.1: What is an operating system". In: *Modern Operating Systems*. 4th ed. Pearson, 2015 (see p. 59).

[7] A.S. Tanenbaum and H. Bos. "Chapter 1.5: Operating system concepts". In: *Modern Operating Systems*. 4th ed. Pearson, 2015 (see p. 59).

[8] D.J. Wheeler. "Programme Organization and Initial Orders for the EDSAC". In: *Proceedings of the Royal Society A* 202.1071 (1950), pp. 573–589 (see pp. 8, 10, 12, 14).

[9] W. Wulf et al. "HYDRA: The Kernel of a Multiprocessor Operating System". In: *Communications of the ACM (CACM)* 17.6 (1974), pp. 337–345 (see pp. 23, 25, 27, 29).

Notes: