Concurrent Computing (Operating Systems)

Daniel Page

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

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:

- Recap:

	application	software
virtual **machine interface** →	- - - - - - - - - - - -	
	kernel	
physical **machine interface** →	- - - - - - - - - - - -	
	hardware	hardware

Notes:

COMS20001 additional content

- Recap:

	application	software
virtual **machine interface** →	- - - - - - - - - - - -	
	kernel	
physical **machine interface** →	- - - - - - - - - - - -	
	hardware	hardware

which suggests we need a kernel *design*, wrt. the

1. interface *and*
2. implementation.

Notes:

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

Definition

metric, *n. a system or standard of measurement; a criterion or set of criteria stated in quantifiable terms.*

– *OED* (`http://www.oed.com`)

Notes:

- Although POSIX is a standard, and arguably *the* standard, it is also fairly old wrt. the general pace of change in technology: various studies such [26] tried to evaluate to what extent the abstractions it offers are still useful (resp. whether or not it carries any historical "baggage" we could now remove), and whether any *new* abstractions are missing (e.g., wrt. modern platforms, such as Android).
- *Whatever* the set of design metrics, keep in mind they a) can be dependent or even mutually exclusive, st. an improvement wrt. one degrades another, or they simply cannot be satisfied at the same time, so b) require a set of priorities, which allow the selection of a suitable trade-off between them. The fact one kernel is unlikely to satisfy *every* design metric is not a problem per se; it leads to domain-specific designs, which target *specific* contexts (e.g., a real-time kernel for embedded systems, which is not suitable for desktop computers). Note also that one metric may be by-products of, or at least strongly influenced by, another: for example, it seems reasonably to posit (however imperfectly) that simplicity + modularity $\Rightarrow$ maintainability.
- Developing a complete set of design metrics is near impossible; it even remains difficult to give a precise, agreed definition for some. That said, here is a (somewhat subjective) overview of those listed:
  - completeness: does or supports everything required wrt. the target domain.
  - efficiency, responsiveness: kernel allocates and manages resources efficiently in time and space; user mode programs observe low-latency and/or high-throughput wrt. interactions with the kernel; user mode programs execute with negligible overhead vs. bare-metal case (i.e., without kernel). Note these stem from a combination of implementation *and* interface (so design as a whole).
  - simplicity, elegance: interfaces between sub-systems are focused, and easy to understand and use.
  - correctness (cf. seL4 [32]), predictability: allows formally proved statements about behaviour; makes it easy to reason (accurately) about current and predict future behaviour.
  - uniformity, consistency, modularity, orthogonality: the kernel is organised into functionally separate sub-systems; there is a no unnecessary dependencies between sub-systems; sub-systems can be combined or composed; interaction with similar (enough) resources is via the same interface.
  - agility, extensability: the behaviour of a sub-system (e.g., policy used) can be altered at run-time; the sub-systems themselves can be added or removed at run-time; reflecting the addition or removal of associated resources.
  - robustness, reliability, resilience (or fault tolerance), availability: hard to cause errors (st. few errors occur); degrades gracefully, and can recover when errors do occur; permits continuous operation (vs. a need to restart in order to cope with an error, for example); prevents any local action by a user mode program causing a global error (e.g., crashing the kernel). Note that user mode programs are borderline adversarial, and that over an extended period of time "exceptions" are not exceptional and mistakes in allocate or management of resources "add up".
  - maintainability, administrability development of the kernel should be easy (implying a need for documentation, and potentially stemming from simplicity and modularity); kernel should be easy to administer (i.e., configure, and manage at run-time).
  - security, isolation, protection supports effective access control for and protection of resources. Note that, in general, modularity $\rightsquigarrow$ isolation $\equiv$ protection $\Rightarrow$ reliability which is the same argument used to motivate the protection of one process from another. A good example of this is the motivation for moving device drivers into user space within a micro-kernel: Swift et al. [37] report that in Windows XP, device drivers account for 85% of all failures; Chou et al. [27] report that programming errors in Linux device drivers are upto ten times more common than the norm.
- A common feature of any assumptions is that efficiency will depend on their being correct. That is, the efficiency of some abnormal program which does *not* exhibit assumed behaviour will be degraded: this is obvious to see wrt. caches, for example, where lack of locality in the address stream will incur a greater number of cache-misses and so longer latency for memory accesses. Normally we prefer the failure of assumptions to be graceful rather than absolute. Following the example above, we would prefer the number of cache misses (and hence access latency) to degrade gradually as the amount of locality decreases, rather than rapidly at some threshold.

COMS20001 additional content

Definition

metric, *n. a system or standard of measurement; a criterion or set of criteria stated in quantifiable terms.*

– *OED* (`http://www.oed.com`)

- Question: what dictates the design itself and/or any design metrics?

Notes:

- Although POSIX is a standard, and arguably *the* standard, it is also fairly old wrt. the general pace of change in technology: various studies such [26] tried to evaluate to what extent the abstractions it offers are still useful (resp. whether or not it carries any historical "baggage" we could now remove), and whether any *new* abstractions are missing (e.g., wrt. modern platforms, such as Android).
- *Whatever* the set of design metrics, keep in mind they a) can be dependent or even mutually exclusive, st. an improvement wrt. one degrades another, or they simply cannot be satisfied at the same time, so b) require a set of priorities, which allow the selection of a suitable trade-off between them. The fact one kernel is unlikely to satisfy *every* design metric is not a problem per se; it leads to domain-specific designs, which target *specific* contexts (e.g., a real-time kernel for embedded systems, which is not suitable for desktop computers). Note also that one metric may be by-products of, or at least strongly influenced by, another: for example, it seems reasonably to posit (however imperfectly) that simplicity + modularity $\Rightarrow$ maintainability.
- Developing a complete set of design metrics is near impossible; it even remains difficult to give a precise, agreed definition for some. That said, here is a (somewhat subjective) overview of those listed:
  - completeness: does or supports everything required wrt. the target domain.
  - efficiency, responsiveness: kernel allocates and manages resources efficiently in time and space; user mode programs observe low-latency and/or high-throughput wrt. interactions with the kernel; user mode programs execute with negligible overhead vs. bare-metal case (i.e., without kernel). Note these stem from a combination of implementation *and* interface (so design as a whole).
  - simplicity, elegance: interfaces between sub-systems are focused, and easy to understand and use.
  - correctness (cf. seL4 [32]), predictability: allows formally proved statements about behaviour; makes it easy to reason (accurately) about current and predict future behaviour.
  - uniformity, consistency, modularity, orthogonality: the kernel is organised into functionally separate sub-systems; there is a no unnecessary dependencies between sub-systems; sub-systems can be combined or composed; interaction with similar (enough) resources is via the same interface.
  - agility, extensability: the behaviour of a sub-system (e.g., policy used) can be altered at run-time; the sub-systems themselves can be added or removed at run-time; reflecting the addition or removal of associated resources.
  - robustness, reliability, resilience (or fault tolerance), availability: hard to cause errors (st. few errors occur); degrades gracefully, and can recover when errors do occur; permits continuous operation (vs. a need to restart in order to cope with an error, for example); prevents any local action by a user mode program causing a global error (e.g., crashing the kernel). Note that user mode programs are borderline adversarial, and that over an extended period of time "exceptions" are not exceptional and mistakes in allocate or management of resources "add up".
  - maintainability, administrability development of the kernel should be easy (implying a need for documentation, and potentially stemming from simplicity and modularity); kernel should be easy to administer (i.e., configure, and manage at run-time).
  - security, isolation, protection supports effective access control for and protection of resources. Note that, in general, modularity $\rightsquigarrow$ isolation $\equiv$ protection $\Rightarrow$ reliability which is the same argument used to motivate the protection of one process from another. A good example of this is the motivation for moving device drivers into user space within a micro-kernel: Swift et al. [37] report that in Windows XP, device drivers account for 85% of all failures; Chou et al. [27] report that programming errors in Linux device drivers are upto ten times more common than the norm.
- A common feature of any assumptions is that efficiency will depend on their being correct. That is, the efficiency of some abnormal program which does *not* exhibit assumed behaviour will be degraded: this is obvious to see wrt. caches, for example, where lack of locality in the address stream will incur a greater number of cache-misses and so longer latency for memory accesses. Normally we prefer the failure of assumptions to be graceful rather than absolute. Following the example above, we would prefer the number of cache misses (and hence access latency) to degrade gradually as the amount of locality decreases, rather than rapidly at some threshold.

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

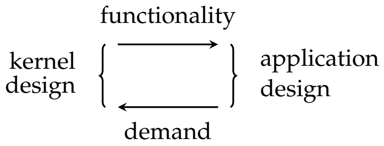
Definition

metric, *n. a system or standard of measurement; a criterion or set of criteria stated in quantifiable terms.*

– OED (`http://www.oed.com`)

- Question: what dictates the design itself and/or any design metrics?
- Answer (?):
  1. user mode programs, i.e.,
     - application software,
     - systems software

     because clearly



  2. human users,
  3. standards,
  4. market forces,
  5. ...

Notes:

- Although POSIX is a standard, and arguably *the* standard, it is also fairly old wrt. the general pace of change in technology: various studies such [26] tried to evaluate to what extent the abstractions it offers are still useful (resp. whether or not it carries any historical "baggage" we could now remove), and whether any *new* abstractions are missing (e.g., wrt. modern platforms, such as Android).
- *Whatever* the set of design metrics, keep in mind they a) can be dependent or even mutually exclusive, st. an improvement wrt. one degrades another, or they simply cannot be satisfied at the same time, so b) require a set of priorities, which allow the selection of a suitable trade-off between them. The fact one kernel is unlikely to satisfy *every* design metric is not a problem per se; it leads to domain-specific designs, which target *specific* contexts (e.g., a real-time kernel for embedded systems, which is not suitable for desktop computers). Note also that one metric may be by-products of, or at least strongly influenced by, another: for example, it seems reasonably to posit (however imperfectly) that simplicity + modularity $\Rightarrow$ maintainability.
- Developing a complete set of design metrics is near impossible; it even remains difficult to give a precise, agreed definition for some. That said, here is a (somewhat subjective) overview of those listed:
  - completeness: does or supports everything required wrt. the target domain.
  - efficiency, responsiveness: kernel allocates and manages resources efficiently in time and space; user mode programs observe low-latency and/or high-throughput wrt. interactions with the kernel; user mode programs execute with negligible overhead vs. bare-metal case (i.e., without kernel). Note these stem from a combination of implementation *and* interface (so design as a whole).
  - simplicity, elegance: interfaces between sub-systems are focused, and easy to understand and use.
  - correctness (cf. seL4 [32]), predictability: allows formally proved statements about behaviour; makes it easy to reason (accurately) about current and predict future behaviour.
  - uniformity, consistency, modularity, orthogonality: the kernel is organised into functionally separate sub-systems; there is a no unnecessary dependencies between sub-systems; sub-systems can be combined or composed; interaction with similar (enough) resources is via the same interface.
  - agility, extensability: the behaviour of a sub-system (e.g., policy used) can be altered at run-time; the sub-systems themselves can be added or removed at run-time; reflecting the addition or removal of associated resources.
  - robustness, reliability, resilience (or fault tolerance), availability: hard to cause errors (st. few errors occur); degrades gracefully, and can recover when errors do occur; permits continuous operation (vs. a need to restart in order to cope with an error, for example); prevents any local action by a user mode program causing a global error (e.g., crashing the kernel). Note that user mode programs are borderline adversarial, and that over an extended period of time "exceptions" are not exceptional and mistakes in allocate or management of resources "add up".
  - maintainability, administrability development of the kernel should be easy (implying a need for documentation, and potentially stemming from simplicity and modularity); kernel should be easy to administer (i.e., configure, and manage at run-time).
  - security, isolation, protection supports effective access control for and protection of resources. Note that, in general, modularity $\rightsquigarrow$ isolation $\equiv$ protection $\Rightarrow$ reliability which is the same argument used to motivate the protection of one process from another. A good example of this is the motivation for moving device drivers into user space within a micro-kernel: Swift et al. [37] report that in Windows XP, device drivers account for 85% of all failures; Chou et al. [27] report that programming errors in Linux device drivers are upto ten times more common than the norm.
- A common feature of any assumptions is that efficiency will depend on their being correct. That is, the efficiency of some abnormal program which does *not* exhibit assumed behaviour will be degraded: this is obvious to see wrt. caches, for example, where lack of locality in the address stream will incur a greater number of cache-misses and so longer latency for memory accesses. Normally we prefer the failure of assumptions to be graceful rather than absolute. Following the example above, we would prefer the number of cache misses (and hence access latency) to degrade gradually as the amount of locality decreases, rather than rapidly at some threshold.

Definition

metric, *n. a system or standard of measurement; a criterion or set of criteria stated in quantifiable terms.*

– OED (`http://www.oed.com`)

- Question: what specific design metrics might exist?

Notes:

- Although POSIX is a standard, and arguably *the* standard, it is also fairly old wrt. the general pace of change in technology: various studies such [26] tried to evaluate to what extent the abstractions it offers are still useful (resp. whether or not it carries any historical "baggage" we could now remove), and whether any *new* abstractions are missing (e.g., wrt. modern platforms, such as Android).
- *Whatever* the set of design metrics, keep in mind they a) can be dependent or even mutually exclusive, st. an improvement wrt. one degrades another, or they simply cannot be satisfied at the same time, so b) require a set of priorities, which allow the selection of a suitable trade-off between them. The fact one kernel is unlikely to satisfy *every* design metric is not a problem per se; it leads to domain-specific designs, which target *specific* contexts (e.g., a real-time kernel for embedded systems, which is not suitable for desktop computers). Note also that one metric may be by-products of, or at least strongly influenced by, another: for example, it seems reasonably to posit (however imperfectly) that simplicity + modularity $\Rightarrow$ maintainability.
- Developing a complete set of design metrics is near impossible; it even remains difficult to give a precise, agreed definition for some. That said, here is a (somewhat subjective) overview of those listed:
  - completeness: does or supports everything required wrt. the target domain.
  - efficiency, responsiveness: kernel allocates and manages resources efficiently in time and space; user mode programs observe low-latency and/or high-throughput wrt. interactions with the kernel; user mode programs execute with negligible overhead vs. bare-metal case (i.e., without kernel). Note these stem from a combination of implementation *and* interface (so design as a whole).
  - simplicity, elegance: interfaces between sub-systems are focused, and easy to understand and use.
  - correctness (cf. seL4 [32]), predictability: allows formally proved statements about behaviour; makes it easy to reason (accurately) about current and predict future behaviour.
  - uniformity, consistency, modularity, orthogonality: the kernel is organised into functionally separate sub-systems; there is a no unnecessary dependencies between sub-systems; sub-systems can be combined or composed; interaction with similar (enough) resources is via the same interface.
  - agility, extensability: the behaviour of a sub-system (e.g., policy used) can be altered at run-time; the sub-systems themselves can be added or removed at run-time; reflecting the addition or removal of associated resources.
  - robustness, reliability, resilience (or fault tolerance), availability: hard to cause errors (st. few errors occur); degrades gracefully, and can recover when errors do occur; permits continuous operation (vs. a need to restart in order to cope with an error, for example); prevents any local action by a user mode program causing a global error (e.g., crashing the kernel). Note that user mode programs are borderline adversarial, and that over an extended period of time "exceptions" are not exceptional and mistakes in allocate or management of resources "add up".
  - maintainability, administrability development of the kernel should be easy (implying a need for documentation, and potentially stemming from simplicity and modularity); kernel should be easy to administer (i.e., configure, and manage at run-time).
  - security, isolation, protection supports effective access control for and protection of resources. Note that, in general, modularity $\rightsquigarrow$ isolation $\equiv$ protection $\Rightarrow$ reliability which is the same argument used to motivate the protection of one process from another. A good example of this is the motivation for moving device drivers into user space within a micro-kernel: Swift et al. [37] report that in Windows XP, device drivers account for 85% of all failures; Chou et al. [27] report that programming errors in Linux device drivers are upto ten times more common than the norm.
- A common feature of any assumptions is that efficiency will depend on their being correct. That is, the efficiency of some abnormal program which does *not* exhibit assumed behaviour will be degraded: this is obvious to see wrt. caches, for example, where lack of locality in the address stream will incur a greater number of cache-misses and so longer latency for memory accesses. Normally we prefer the failure of assumptions to be graceful rather than absolute. Following the example above, we would prefer the number of cache misses (and hence access latency) to degrade gradually as the amount of locality decreases, rather than rapidly at some threshold.

Definition

metric, *n. a system or standard of measurement; a criterion or set of criteria stated in quantifiable terms.*

– *OED* (`http://www.oed.com`)

- Question: what specific design metrics might exist?
- Answer (?):

1. completeness,
2. efficiency, responsiveness,
3. simplicity, elegance,
4. correctness, predictability,
5. uniformity, consistency, modularity, orthogonality,
6. agility, extensibility,
7. robustness, reliability, resilience (or fault tolerance), availability,
8. maintainability, administrability,
9. security, isolation, protection,
10. ...

Notes:

- Although POSIX is a standard, and arguably *the* standard, it is also fairly old wrt. the general pace of change in technology: various studies such [26] tried to evaluate to what extent the abstractions it offers are still useful (resp. whether or not it carries any historical "baggage" we could now remove), and whether any *new* abstractions are missing (e.g., wrt. modern platforms, such as Android).
- *Whatever* the set of design metrics, keep in mind they a) can be dependent or even mutually exclusive, st. an improvement wrt. one degrades another, or they simply cannot be satisfied at the same time, so b) require a set of priorities, which allow the selection of a suitable trade-off between them. The fact one kernel is unlikely to satisfy *every* design metric is not a problem per se; it leads to domain-specific designs, which target *specific* contexts (e.g., a real-time kernel for embedded systems, which is not suitable for desktop computers). Note also that one metric may be by-products of, or at least strongly influenced by, another: for example, it seems reasonably to posit (however imperfectly) that simplicity + modularity $\Rightarrow$ maintainability.
- Developing a complete set of design metrics is near impossible; it even remains difficult to give a precise, agreed definition for some. That said, here is a (somewhat subjective) overview of those listed:
  - completeness: does or supports everything required wrt. the target domain.
  - efficiency, responsiveness: kernel allocates and manages resources efficiently in time and space; user mode programs observe low-latency and/or high-throughput wrt. interactions with the kernel; user mode programs execute with negligible overhead vs. bare-metal case (i.e., without kernel). Note these stem from a combination of implementation *and* interface (so design as a whole).
  - simplicity, elegance: interfaces between sub-systems are focused, and easy to understand and use.
  - correctness (cf. seL4 [32]), predictability: allows formally proved statements about behaviour; makes it easy to reason (accurately) about current and predict future behaviour.
  - uniformity, consistency, modularity, orthogonality: the kernel is organised into functionally separate sub-systems; there is a no unnecessary dependencies between sub-systems; sub-systems can be combined or composed; interaction with similar (enough) resources is via the same interface.
  - agility, extensability: the behaviour of a sub-system (e.g., policy used) can be altered at run-time; the sub-systems themselves can be added or removed at run-time; reflecting the addition or removal of associated resources.
  - robustness, reliability, resilience (or fault tolerance), availability: hard to cause errors (st. few errors occur); degrades gracefully, and can recover when errors do occur; permits continuous operation (vs. a need to restart in order to cope with an error, for example); prevents any local action by a user mode program causing a global error (e.g., crashing the kernel). Note that user mode programs are borderline adversarial, and that over an extended period of time "exceptions" are not exceptional and mistakes in allocate or management of resources "add up".
  - maintainability, administrability development of the kernel should be easy (implying a need for documentation, and potentially stemming from simplicity and modularity); kernel should be easy to administer (i.e., configure, and manage at run-time).
  - security, isolation, protection supports effective access control for and protection of resources. Note that, in general, modularity $\rightsquigarrow$ isolation $\equiv$ protection $\Rightarrow$ reliability which is the same argument used to motivate the protection of one process from another. A good example of this is the motivation for moving device drivers into user space within a micro-kernel: Swift et al. [37] report that in Windows XP, device drivers account for 85% of all failures; Chou et al. [27] report that programming errors in Linux device drivers are upto ten times more common than the norm.
- A common feature of any assumptions is that efficiency will depend on their being correct. That is, the efficiency of some abnormal program which does *not* exhibit assumed behaviour will be degraded: this is obvious to see wrt. caches, for example, where lack of locality in the address stream will incur a greater number of cache-misses and so longer latency for memory accesses. Normally we prefer the failure of assumptions to be graceful rather than absolute. Following the example above, we would prefer the number of cache misses (and hence access latency) to degrade gradually as the amount of locality decreases, rather than rapidly at some threshold.

COMS20001 additional content

Definition

metric, *n. a system or standard of measurement; a criterion or set of criteria stated in quantifiable terms.*

– *OED* (`http://www.oed.com`)

- Question: what specific design metrics might exist?
- Answer (?):

1. completeness,
2. efficiency, responsiveness,
3. simplicity, elegance,
4. correctness, predictability,
5. uniformity, consistency, modularity, orthogonality,
6. agility, extensibility,
7. robustness, reliability, resilience (or fault tolerance), availability,
8. maintainability, administrability,
9. security, isolation, protection,
10. ...

- Implication: need a select a trade-off, often leading to

$$\begin{array}{rcl} \text{general-purpose design} & \Rightarrow & \text{stronger assumptions} \\ \text{special-purpose design} & \Rightarrow & \text{weaker assumptions} \end{array}$$

Notes:

- Although POSIX is a standard, and arguably *the* standard, it is also fairly old wrt. the general pace of change in technology: various studies such [26] tried to evaluate to what extent the abstractions it offers are still useful (resp. whether or not it carries any historical "baggage" we could now remove), and whether any *new* abstractions are missing (e.g., wrt. modern platforms, such as Android).
- *Whatever* the set of design metrics, keep in mind they a) can be dependent or even mutually exclusive, st. an improvement wrt. one degrades another, or they simply cannot be satisfied at the same time, so b) require a set of priorities, which allow the selection of a suitable trade-off between them. The fact one kernel is unlikely to satisfy *every* design metric is not a problem per se; it leads to domain-specific designs, which target *specific* contexts (e.g., a real-time kernel for embedded systems, which is not suitable for desktop computers). Note also that one metric may be by-products of, or at least strongly influenced by, another: for example, it seems reasonably to posit (however imperfectly) that simplicity + modularity $\Rightarrow$ maintainability.
- Developing a complete set of design metrics is near impossible; it even remains difficult to give a precise, agreed definition for some. That said, here is a (somewhat subjective) overview of those listed:
  - completeness: does or supports everything required wrt. the target domain.
  - efficiency, responsiveness: kernel allocates and manages resources efficiently in time and space; user mode programs observe low-latency and/or high-throughput wrt. interactions with the kernel; user mode programs execute with negligible overhead vs. bare-metal case (i.e., without kernel). Note these stem from a combination of implementation *and* interface (so design as a whole).
  - simplicity, elegance: interfaces between sub-systems are focused, and easy to understand and use.
  - correctness (cf. seL4 [32]), predictability: allows formally proved statements about behaviour; makes it easy to reason (accurately) about current and predict future behaviour.
  - uniformity, consistency, modularity, orthogonality: the kernel is organised into functionally separate sub-systems; there is a no unnecessary dependencies between sub-systems; sub-systems can be combined or composed; interaction with similar (enough) resources is via the same interface.
  - agility, extensability: the behaviour of a sub-system (e.g., policy used) can be altered at run-time; the sub-systems themselves can be added or removed at run-time; reflecting the addition or removal of associated resources.
  - robustness, reliability, resilience (or fault tolerance), availability: hard to cause errors (st. few errors occur); degrades gracefully, and can recover when errors do occur; permits continuous operation (vs. a need to restart in order to cope with an error, for example); prevents any local action by a user mode program causing a global error (e.g., crashing the kernel). Note that user mode programs are borderline adversarial, and that over an extended period of time "exceptions" are not exceptional and mistakes in allocate or management of resources "add up".
  - maintainability, administrability development of the kernel should be easy (implying a need for documentation, and potentially stemming from simplicity and modularity); kernel should be easy to administer (i.e., configure, and manage at run-time).
  - security, isolation, protection supports effective access control for and protection of resources. Note that, in general, modularity $\rightsquigarrow$ isolation $\equiv$ protection $\Rightarrow$ reliability which is the same argument used to motivate the protection of one process from another. A good example of this is the motivation for moving device drivers into user space within a micro-kernel: Swift et al. [37] report that in Windows XP, device drivers account for 85% of all failures; Chou et al. [27] report that programming errors in Linux device drivers are upto ten times more common than the norm.
- A common feature of any assumptions is that efficiency will depend on their being correct. That is, the efficiency of some abnormal program which does *not* exhibit assumed behaviour will be degraded: this is obvious to see wrt. caches, for example, where lack of locality in the address stream will incur a greater number of cache-misses and so longer latency for memory accesses. Normally we prefer the failure of assumptions to be graceful rather than absolute. Following the example above, we would prefer the number of cache misses (and hence access latency) to degrade gradually as the amount of locality decreases, rather than rapidly at some threshold.

- Beware: this can be a difficult topic to give precise answers about, with a famous essay [9] summarising the issue (roughly) as

Metric	do-the-right-thing (or, MIT philosophy)	worse-is-better (or, New Jersey philosophy)
Simplicity	interface > implementation	implementation > interface
Correctness	totally	mostly
Consistency	totally	mostly
Completeness	mostly	as given
Example		

Notes:

- The difficulty described is *potentially* the reason that high(er)-level design principles are often paid less attention than low(er)-level technical detail. That said, some places to start include
  - [11, Section 2.7] on operating system structure (i.e., architecture),
  - [12, Section 1.7] on operating system structure (i.e., architecture), and
  - [12, Chapter 12] on operating system design (e.g., of interfaces and implementations)

  plus various historical retrospectives [34, 33].

- Beware: this can be a difficult topic to give precise answers about, with a famous essay [9] summarising the issue (roughly) as

Metric	do-the-right-thing (or, MIT philosophy)	worse-is-better (or, New Jersey philosophy)
Simplicity	interface > implementation	implementation > interface
Correctness	totally	mostly
Consistency	totally	mostly
Completeness	mostly	as given
Example	MULTICS [29, 28] kernel OSI [6] network model	UNIX [36, 35, 38] kernel Internet [4] network model

where, perhaps surprisingly, the right-hand case “wins”.

Notes:

- The difficulty described is *potentially* the reason that high(er)-level design principles are often paid less attention than low(er)-level technical detail. That said, some places to start include
  - [11, Section 2.7] on operating system structure (i.e., architecture),
  - [12, Section 1.7] on operating system structure (i.e., architecture), and
  - [12, Chapter 12] on operating system design (e.g., of interfaces and implementations)

  plus various historical retrospectives [34, 33].

- Beware: this can be a difficult topic to give precise answers about, with a famous essay [9] summarising the issue (roughly) as

Metric	do-the-right-thing (or, MIT philosophy)	worse-is-better (or, New Jersey philosophy)
Simplicity	interface > implementation	implementation > interface
Correctness	totally	mostly
Consistency	totally	mostly
Completeness	mostly	as given
Example	MULTICS [29, 28] kernel OSI [6] network model	UNIX [36, 35, 38] kernel Internet [4] network model

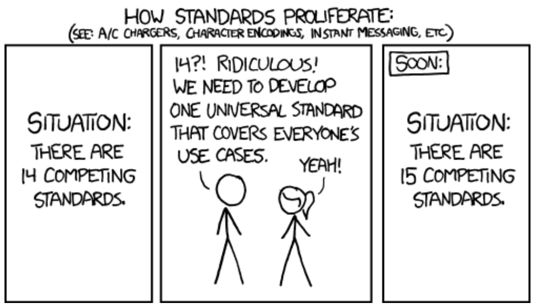
  where, perhaps surprisingly, the right-hand case "wins".
- Moral #1: *over*-design (cf. *over*-engineering) is often due to bad design trade-off(s).
- Moral #2: purely academic study of this topic *might* not be so worthwhile!

Notes:

- The difficulty described is *potentially* the reason that high(er)-level design principles are often paid less attention than low(er)-level technical detail. That said, some places to start include
  - [11, Section 2.7] on operating system structure (i.e., architecture),
  - [12, Section 1.7] on operating system structure (i.e., architecture), and
  - [12, Chapter 12] on operating system design (e.g., of interfaces and implementations)

  plus various historical retrospectives [34, 33].

An Aside: (UNIX-like) standardisation



http://xkcd.com/927/

Notes:

An Aside: (UNIX-like) standardisation

- Bad news: at face value, this issue looks complex, e.g., see
  - **System V Interface Definition (SVID)** [17, 18, 19, 20, 21],
  - **X/Open Portability Guide (XPG)** [23, 24, 25]
  - **Portable Operating System Interface (POSIX)** [14, 13],
  - **Single UNIX Specification (SUS)** [22],
  - **Linux Standard Base (LSB)** [15].

Notes:

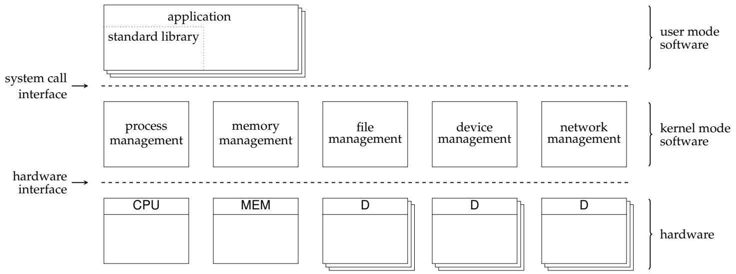
- Good news:
  - there's a *lot* of overlap between UNIX-like standards,
  - we'll focus on POSIX [16] only,
    - – [16] is older than (so replaced by) [14, 13],
    - + it's more likely you can actually get a (free) copy of [16] than [14, 13]

  which simplifies matters considerably.

Notes:

Definition

A **monolithic kernel** is st.



where the central idea is:

- the *entire* kernel executes in kernel space:
  + less inter-mode interaction, so less overhead wrt. context switching
  – larger sub-systems, so harder to maintain and verify
  – less isolation, so worse wrt. reliability
  – must make assumptions wrt. general-purpose applications

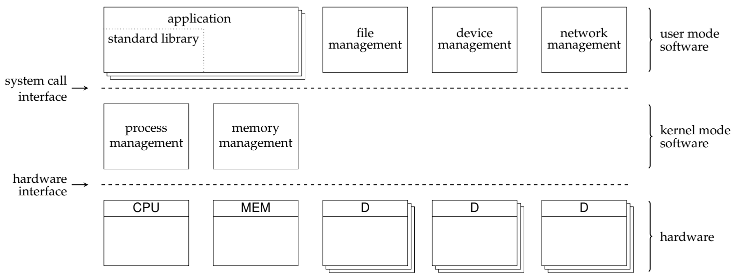
Notes:

- Remember: these diagrams are for illustration only. Put another way, you may infer from a given diagram that the kernel is "in between" hardware and applications executing on it. This is only true conceptually: when a user mode process is executing, it does so without involvement from the kernel *until* need be. For example, the kernel is *not* an interpreter; it does not process each instruction the user mode software attempts to execute before the hardware does. So *either* the kernel is executing (e.g., performing a management task on behalf of a process) in kernel mode *or* a process is executing in user mode.
- 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 [8] publicly documents many of the arguments wrt. advantages of monolithic kernel vs. micro-kernels. 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, which therefore represents a crucial aspect of the kernel. There is more compartmentalisation, in the sense that each of the user mode kernel sub-systems is protected from the other.
- The term hybrid kernel (or macro-kernel, sometimes used in the context of Windows NT) describes a mix of monolithic and micro-kernel designs. A simple definition is that it relaxes the micro-kernel client-server model. That is, a pure micro-kernel is characterised by a) modular organisation of kernel sub-systems, b) execution of kernel sub-systems in kernel mode iff. they need to be, with user mode execution preferred otherwise. A hybrid kernel retains the former, while relaxing the latter: if, for example, efficiency (rather than function) dictates executing a given sub-system in kernel mode (so absorbing it into a "partly monolithic") kernel, this is deemed reasonable and allowed. So, in a sense, it is a monolithic kernel wrt. processor mode but a micro-kernel wrt. the architecture. However, the value of this point in the design space is contentious:
  – proponents of monolithic kernels would argue, justifiably, they make effective use of modularity in the same way *any* complex software artefact will, whereas
  – proponents of micro-kernels might argue the need to execute a sub-system in kernel mode as a result of efficiency is due to poor design or implementation (e.g., of the IPC mechanism, if this were the bottleneck) rather than the micro-kernel concept per se.
- The term **library kernel** is sometimes used to mean a uni-kernel; either way, the idea is basically that the "kernel" is more like a run-time support system.
- The concept of an exo-kernel [30, 31] *specifically* refutes the idea that a kernel should be a layer of abstraction over the underlying hardware. Instead, 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.
  The **end-to-end principle** forms part of computer networks design. In short, it states that all application-specific functionality should be implemented in the source and destination, rather than intermediate nodes; this is demonstrated, for example, by IP which is agnostic to any particular application (or higher-level protocol). The exo-kernel concept follows basically the same idea, in the sense it forces every application to implement application-specific functionality: the kernel is simply there to facilitate this, rather than force

Kernel design (1)
High(er)-level

Definition

A **micro-kernel** is st.



where the central idea is:

- akin to a client-server model, st.
  - the *kernel* of the kernel executes in kernel space, but
  - the rest of the kernel executes in user space

  implying a strong requirement for efficient mechanisms for IPC:
  – more inter-mode interaction, so more overhead wrt. context switching
  + smaller sub-systems, so easier to maintain and verify
  + more isolation, so better wrt. reliability
  – must make assumptions wrt. general-purpose applications

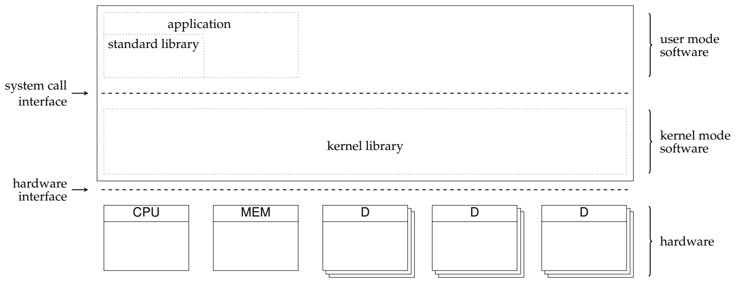
© Daniel Page ⟨Daniel.Page⟩
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08 University of BRISTOL

Notes:

- Remember: these diagrams are for illustration only. Put another way, you may infer from a given diagram that the kernel is "in between" hardware and applications executing on it. This is only true conceptually: when a user mode process is executing, it does so without involvement from the kernel *until* need be. For example, the kernel is *not* an interpreter; it does not process each instruction the user mode software attempts to execute before the hardware does. So *either* the kernel is executing (e.g., performing a management task on behalf of a process) in kernel mode *or* a process is executing in user mode.
- 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 [8] publicly documents many of the arguments wrt. advantages of monolithic kernel vs. micro-kernels. 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, which therefore represents a crucial aspect of the kernel. There is more compartmentalisation, in the sense that each of the user mode kernel sub-systems is protected from the other.
- The term hybrid kernel (or macro-kernel, sometimes used in the context of Windows NT) describes a mix of monolithic and micro-kernel designs. A simple definition is that it relaxes the micro-kernel client-server model. That is, a pure micro-kernel is characterised by a) modular organisation of kernel sub-systems, b) execution of kernel sub-systems in kernel mode iff. they need to be, with user mode execution preferred otherwise. A hybrid kernel retains the former, while relaxing the latter: if, for example, efficiency (rather than function) dictates executing a given sub-system in kernel mode (so absorbing it into a "partly monolithic") kernel, this is deemed reasonable and allowed. So, in a sense, it is a monolithic kernel wrt. processor mode but a micro-kernel wrt. the architecture. However, the value of this point in the design space is contentious:
  – proponents of monolithic kernels would argue, justifiably, they make effective use of modularity in the same way *any* complex software artefact will, whereas
  – proponents of micro-kernels might argue the need to execute a sub-system in kernel mode as a result of efficiency is due to poor design or implementation (e.g., of the IPC mechanism, if this were the bottleneck) rather than the micro-kernel concept per se.
- The term **library kernel** is sometimes used to mean a uni-kernel; either way, the idea is basically that the "kernel" is more like a run-time support system.
- The concept of an exo-kernel [30, 31] *specifically* refutes the idea that a kernel should be a layer of abstraction over the underlying hardware. Instead, 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.
  The **end-to-end principle** forms part of computer networks design. In short, it states that all application-specific functionality should be implemented in the source and destination, rather than intermediate nodes; this is demonstrated, for example, by IP which is agnostic to any particular application (or higher-level protocol). The exo-kernel concept follows basically the same idea, in the sense it forces every application to implement application-specific functionality: the kernel is simply there to facilitate this, rather than force

Definition

A **uni-kernel** is st.



where the central idea is:

- the *entire* kernel and application are linked together, so
- kernel effectively becomes a run-time library:

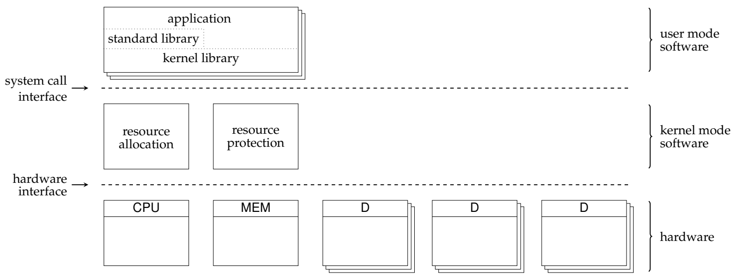
  + less inter-mode interaction, so less overhead wrt. context switching
  – larger sub-systems, so harder to maintain and verify
  – less isolation, so worse wrt. reliability
  + can make assumptions wrt. special-purpose application
  – disallows multi-programming!

Notes:

- Remember: these diagrams are for illustration only. Put another way, you may infer from a given diagram that the kernel is "in between" hardware and applications executing on it. This is only true conceptually: when a user mode process is executing, it does so without involvement from the kernel *until* need be. For example, the kernel is *not* an interpreter; it does not process each instruction the user mode software attempts to execute before the hardware does. So *either* the kernel is executing (e.g., performing a management task on behalf of a process) in kernel mode *or* a process is executing in user mode.
- 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 [8] publicly documents many of the arguments wrt. advantages of monolithic kernel vs. micro-kernels. 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, which therefore represents a crucial aspect of the kernel. There is more compartmentalisation, in the sense that each of the user mode kernel sub-systems is protected from the other.
- The term hybrid kernel (or macro-kernel, sometimes used in the context of Windows NT) describes a mix of monolithic and micro-kernel designs. A simple definition is that it relaxes the micro-kernel client-server model. That is, a pure micro-kernel is characterised by a) modular organisation of kernel sub-systems, b) execution of kernel sub-systems in kernel mode iff. they need to be, with user mode execution preferred otherwise. A hybrid kernel retains the former, while relaxing the latter: if, for example, efficiency (rather than function) dictates executing a given sub-system in kernel mode (so absorbing it into a "partly monolithic") kernel, this is deemed reasonable and allowed. So, in a sense, it is a monolithic kernel wrt. processor mode but a micro-kernel wrt. the architecture. However, the value of this point in the design space is contentious:
  – proponents of monolithic kernels would argue, justifiably, they make effective use of modularity in the same way *any* complex software artefact will, whereas
  – proponents of micro-kernels might argue the need to execute a sub-system in kernel mode as a result of efficiency is due to poor design or implementation (e.g., of the IPC mechanism, if this were the bottleneck) rather than the micro-kernel concept per se.
- The term **library kernel** is sometimes used to mean a uni-kernel; either way, the idea is basically that the "kernel" is more like a run-time support system.
- The concept of an exo-kernel [30, 31] *specifically* refutes the idea that a kernel should be a layer of abstraction over the underlying hardware. Instead, 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.
  The **end-to-end principle** forms part of computer networks design. In short, it states that all application-specific functionality should be implemented in the source and destination, rather than intermediate nodes; this is demonstrated, for example, by IP which is agnostic to any particular application (or higher-level protocol). The exo-kernel concept follows basically the same idea, in the sense it forces every application to implement application-specific functionality: the kernel is simply there to facilitate this, rather than force

Definition

An **exo-kernel** is st.



where the central idea is:

- make micro-kernel concept even *more* minimalist:
  - kernel *only* deals with allocation and protection of resources,
  - applications access resources *directly*, potentially via a kernel library.

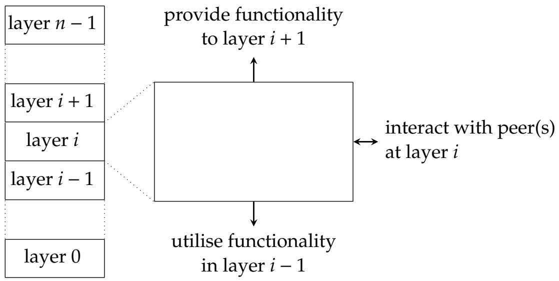
  + less inter-mode interaction, so less overhead wrt. context switching
  + smaller sub-systems, so easier to maintain and verify
  + more isolation, so better wrt. reliability
  + can make assumptions wrt. special-purpose application

Notes:

- Remember: these diagrams are for illustration only. Put another way, you may infer from a given diagram that the kernel is "in between" hardware and applications executing on it. This is only true conceptually: when a user mode process is executing, it does so without involvement from the kernel *until* need be. For example, the kernel is *not* an interpreter; it does not process each instruction the user mode software attempts to execute before the hardware does. So *either* the kernel is executing (e.g., performing a management task on behalf of a process) in kernel mode *or* a process is executing in user mode.
- 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 [8] publicly documents many of the arguments wrt. advantages of monolithic kernel vs. micro-kernels. 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, which therefore represents a crucial aspect of the kernel. There is more compartmentalisation, in the sense that each of the user mode kernel sub-systems is protected from the other.
- The term hybrid kernel (or macro-kernel, sometimes used in the context of Windows NT) describes a mix of monolithic and micro-kernel designs. A simple definition is that it relaxes the micro-kernel client-server model. That is, a pure micro-kernel is characterised by a) modular organisation of kernel sub-systems, b) execution of kernel sub-systems in kernel mode iff. they need to be, with user mode execution preferred otherwise. A hybrid kernel retains the former, while relaxing the latter: if, for example, efficiency (rather than function) dictates executing a given sub-system in kernel mode (so absorbing it into a "partly monolithic") kernel, this is deemed reasonable and allowed. So, in a sense, it is a monolithic kernel wrt. processor mode but a micro-kernel wrt. the architecture. However, the value of this point in the design space is contentious:
  – proponents of monolithic kernels would argue, justifiably, they make effective use of modularity in the same way *any* complex software artefact will, whereas
  – proponents of micro-kernels might argue the need to execute a sub-system in kernel mode as a result of efficiency is due to poor design or implementation (e.g., of the IPC mechanism, if this were the bottleneck) rather than the micro-kernel concept per se.
- The term **library kernel** is sometimes used to mean a uni-kernel; either way, the idea is basically that the "kernel" is more like a run-time support system.
- The concept of an exo-kernel [30, 31] *specifically* refutes the idea that a kernel should be a layer of abstraction over the underlying hardware. Instead, 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.
  The **end-to-end principle** forms part of computer networks design. In short, it states that all application-specific functionality should be implemented in the source and destination, rather than intermediate nodes; this is demonstrated, for example, by IP which is agnostic to any particular application (or higher-level protocol). The exo-kernel concept follows basically the same idea, in the sense it forces every application to implement application-specific functionality: the kernel is simply there to facilitate this, rather than force

Kernel design (2)
Low(er)-level

- At a lower level, an idealised architecture might resemble



but

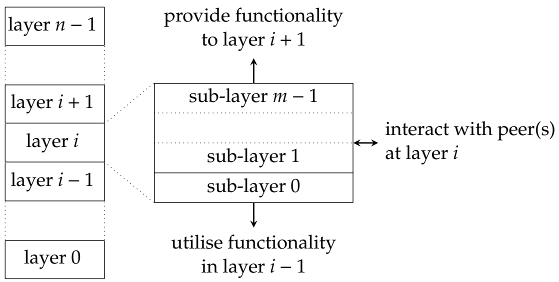
1. strict separation of layers is often
   - too restrictive, and/or
   - too inefficient.

Notes:

- An example of relaxing the layered design relates to the implication that access to the device driver, from user space, must be performed via other layers. In reality, there is often a way to bypass these layers: Linux has the `ioctl` [5] system call, and Windows has `DeviceIoControl` which basically represent instances of the same concept.
- The design of Linux includes numerous examples of the sub-layering concept. As with most kernels, the treatment of USB devices is a case in point. USB devices are similar in the sense they all communicate via the USB protocol: at a high level, for example, a USB keyboard is no different from a USB disk. However, the USB device driver stack is organised as a set sub-layers that attempt to a) deal with high-level commonalities *and* b) allow low-level differences between devices. Note that the term stack hints at the similarity of this organisation to a network stack.
- The design of Linux includes numerous examples of the unified interface concept, the description of which (wrt. the metaphor of shapes representing the interface) stems from [12, Figure 5-14]:
  1. The collection of devices into block, character and network classes clearly represents an attempt to present a unified interface to each class. The fact there are three classes highlights the challenge of devising a single "perfect" interface, and/or recognition that advantages stem from the presentation of a more specific (albeit general within that class) interface.
  2. Treating "everything as a file" [2] is a central philosophy in UNIX, wherein a different classes of resource are exposed (via the system call interface) as part of the file system. For example, a process can access a diverse set of devices (e.g., a keyboard) via a uniform set of system calls (e.g., `read`) then translated into the appropriate device-specific semantics (e.g., read a key press); such devices appear as **device nodes** (so act as pseudo-files) in the file system (e.g., within the `/dev` directory).

Kernel design (2)
Low(er)-level

- At a lower level, an idealised architecture might resemble



but

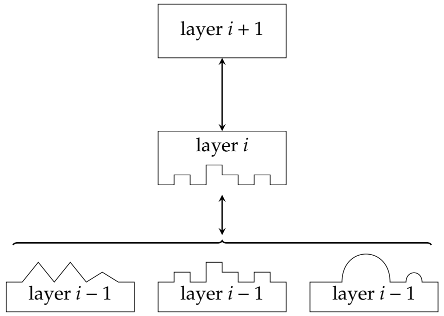
1. strict separation of layers is often
   - too restrictive, and/or
   - too inefficient,
2. a given layer might comprise numerous sub-layers.

Notes:

- An example of relaxing the layered design relates to the implication that access to the device driver, from user space, must be performed via other layers. In reality, there is often a way to bypass these layers: Linux has the `ioctl` [5] system call, and Windows has `DeviceIoControl` which basically represent instances of the same concept.
- The design of Linux includes numerous examples of the sub-layering concept. As with most kernels, the treatment of USB devices is a case in point. USB devices are similar in the sense they all communicate via the USB protocol: at a high level, for example, a USB keyboard is no different from a USB disk. However, the USB device driver stack is organised as a set sub-layers that attempt to a) deal with high-level commonalities *and* b) allow low-level differences between devices. Note that the term stack hints at the similarity of this organisation to a network stack.
- The design of Linux includes numerous examples of the unified interface concept, the description of which (wrt. the metaphor of shapes representing the interface) stems from [12, Figure 5-14]:
  1. The collection of devices into block, character and network classes clearly represents an attempt to present a unified interface to each class. The fact there are three classes highlights the challenge of devising a single "perfect" interface, and/or recognition that advantages stem from the presentation of a more specific (albeit general within that class) interface.
  2. Treating "everything as a file" [2] is a central philosophy in UNIX, wherein a different classes of resource are exposed (via the system call interface) as part of the file system. For example, a process can access a diverse set of devices (e.g., a keyboard) via a uniform set of system calls (e.g., `read`) then translated into the appropriate device-specific semantics (e.g., read a key press); such devices appear as **device nodes** (so act as pseudo-files) in the file system (e.g., within the `/dev` directory).

Kernel design (2)
Low(er)-level

- At a lower level, an idealised architecture might resemble



but

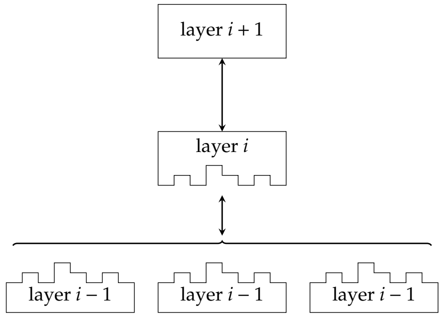
1. strict separation of layers is often
   - too restrictive, and/or
   - too inefficient,
2. a given layer might comprise numerous sub-layers,
3. we prefer unified (vs. natural) interfaces, to permit plug-and-play of concrete instances.

Notes:

- An example of relaxing the layered design relates to the implication that access to the device driver, from user space, must be performed via other layers. In reality, there is often a way to bypass these layers: Linux has the `ioctl` [5] system call, and Windows has `DeviceIoControl` which basically represent instances of the same concept.
- The design of Linux includes numerous examples of the sub-layering concept. As with most kernels, the treatment of USB devices is a case in point. USB devices are similar in the sense they all communicate via the USB protocol: at a high level, for example, a USB keyboard is no different from a USB disk. However, the USB device driver stack is organised as a set sub-layers that attempt to a) deal with high-level commonalities *and* b) allow low-level differences between devices. Note that the term stack hints at the similarity of this organisation to a network stack.
- The design of Linux includes numerous examples of the unified interface concept, the description of which (wrt. the metaphor of shapes representing the interface) stems from [12, Figure 5-14]:
  1. The collection of devices into block, character and network classes clearly represents an attempt to present a unified interface to each class. The fact there are three classes highlights the challenge of devising a single "perfect" interface, and/or recognition that advantages stem from the presentation of a more specific (albeit general within that class) interface.
  2. Treating "everything as a file" [2] is a central philosophy in UNIX, wherein a different classes of resource are exposed (via the system call interface) as part of the file system. For example, a process can access a diverse set of devices (e.g., a keyboard) via a uniform set of system calls (e.g., `read`) then translated into the appropriate device-specific semantics (e.g., read a key press); such devices appear as **device nodes** (so act as pseudo-files) in the file system (e.g., within the `/dev` directory).

Kernel design (2)
Low(er)-level

- At a lower level, an idealised architecture might resemble



but

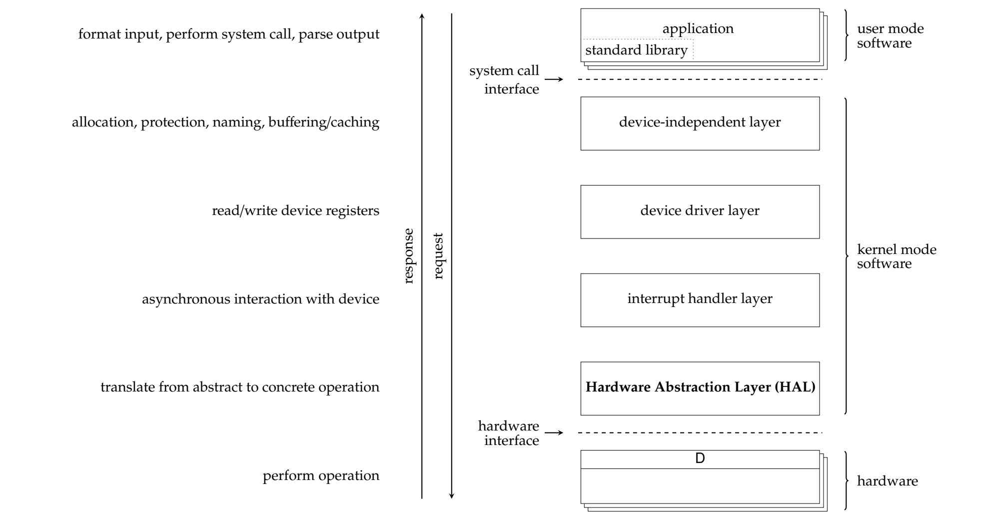
1. strict separation of layers is often
   - too restrictive, and/or
   - too inefficient,
2. a given layer might comprise numerous sub-layers,
3. we prefer unified (vs. natural) interfaces, to permit plug-and-play of concrete instances.

Notes:

- An example of relaxing the layered design relates to the implication that access to the device driver, from user space, must be performed via other layers. In reality, there is often a way to bypass these layers: Linux has the `ioctl` [5] system call, and Windows has `DeviceIoControl` which basically represent instances of the same concept.
- The design of Linux includes numerous examples of the sub-layering concept. As with most kernels, the treatment of USB devices is a case in point. USB devices are similar in the sense they all communicate via the USB protocol: at a high level, for example, a USB keyboard is no different from a USB disk. However, the USB device driver stack is organised as a set sub-layers that attempt to a) deal with high-level commonalities *and* b) allow low-level differences between devices. Note that the term stack hints at the similarity of this organisation to a network stack.
- The design of Linux includes numerous examples of the unified interface concept, the description of which (wrt. the metaphor of shapes representing the interface) stems from [12, Figure 5-14]:
  1. The collection of devices into block, character and network classes clearly represents an attempt to present a unified interface to each class. The fact there are three classes highlights the challenge of devising a single "perfect" interface, and/or recognition that advantages stem from the presentation of a more specific (albeit general within that class) interface.
  2. Treating "everything as a file" [2] is a central philosophy in UNIX, wherein a different classes of resource are exposed (via the system call interface) as part of the file system. For example, a process can access a diverse set of devices (e.g., a keyboard) via a uniform set of system calls (e.g., `read`) then translated into the appropriate device-specific semantics (e.g., read a key press); such devices appear as **device nodes** (so act as pseudo-files) in the file system (e.g., within the `/dev` directory).

Kernel design (3)
Low(er)-level

Example

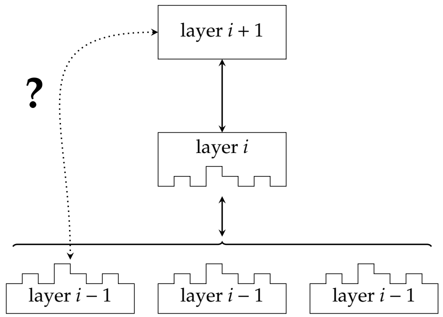


Notes:

- The acronym HAL needs to be understood in context. For example, from the perspective of a user mode program the entire kernel is a form of HAL! Rather, we mean a layer that, as far as possible, encapsulates and abstracts specificities of underlying hardware (e.g., how to perform a context switch); a typical motivation for this is to permit easier porting of the kernel between platforms. This approach was used in Windows NT [1], and thus permitted Intel- and ARM-based versions for example.

Kernel design (4)
Complications and asides

- Problem: we need end-to-end connectivity across interfaces, e.g.,



implying we need a way to *identify* resources and/or sub-system.

Notes:

- A descriptor is commonly used to bridge the interface between kernel and user space. Consider a user space process that makes a request of the kernel to allocate some resource, such as opening a file; it does so using the system call `open`. The kernel allocates the resource (i.e., and entry in the file descriptor table in the PCB for that process), and passes the descriptor (i.e., the index in said table) back, as a return value, to the process. The descriptor is not *meaningful* to the process in any real sense: it simply acts as a way to specify a particular, open file from then on. For example, when the process needs to read data from the file using `read`, it passes the descriptor to the kernel as a way of specifying the file.
- Whether considering kernel-facing (or internalised) or user-facing (or externalised) names, identifying examples is fairly simple: Process IDentifiers (PIDs), User IDentifiers (UIDs), Group IDentifiers (GIDs), file descriptors are just a few.
- There are various challenges related to allocation of names, but two are that a) the set of names may be static or dynamically, so change over time (e.g., as resources such as hardware devices are added or removed) and be short- or long-lived, and b) there needs to be a way to prevent name clashes (and so ambiguity), which typically demands some mechanism for namespaces
- [10, Chapter 3] offers an explanation of how this issue is dealt with in Linux, focusing wlog. on how character devices are identified using a pair of major/minor device numbers: the former suggests the device driver being used, whereas the latter identifies the specific device. You can *see* these identifiers in the output of

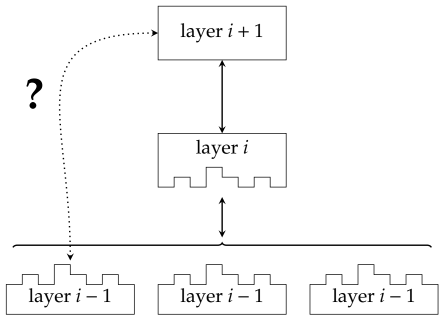
```
ls -l /dev
```

which will include various devices nodes (i.e., pseudo-files) which allow interaction with specific devices.

Kernel design (4)

Complications and asides

- Problem: we need end-to-end connectivity across interfaces, e.g.,



  implying we need a way to *identify* resources and/or sub-system.
- Solution(s):
  - via a **descriptor** (or **handle**) (e.g., a file descriptor of type `int`),
  - via the file system (e.g., `/dev/ttyUSB0`),
  - via allocated, abstract identifier (e.g., `eth0`)
  - via fixed, concrete identifier (e.g., USB device `045e:0039`)
  - ...

Notes:

- A descriptor is commonly used to bridge the interface between kernel and user space. Consider a user space process that makes a request of the kernel to allocate some resource, such as opening a file; it does so using the system call `open`. The kernel allocates the resource (i.e., and entry in the file descriptor table in the PCB for that process), and passes the descriptor (i.e., the index in said table) back, as a return value, to the process. The descriptor is not *meaningful* to the process in any real sense: it simply acts as a way to specify a particular, open file from then on. For example, when the process needs to read data from the file using `read`, it passes the descriptor to the kernel as a way of specifying the file.
- Whether considering kernel-facing (or internalised) or user-facing (or externalised) names, identifying examples is fairly simple: Process IDentifiers (PIDs), User IDentifiers (UIDs), Group IDentifiers (GIDs), file descriptors are just a few.
- There are various challenges related to allocation of names, but two are that a) the set of names may be static or dynamically, so change over time (e.g., as resources such as hardware devices are added or removed) and be short- or long-lived, and b) there needs to be a way to prevent name clashes (and so ambiguity), which typically demands some mechanism for namespaces
- [10, Chapter 3] offers an explanation of how this issue is dealt with in Linux, focusing wlog. on how character devices are identified using a pair of major/minor device numbers: the former suggests the device driver being used, whereas the latter identifies the specific device. You can *see* these identifiers in the output of

  ```
  ls -l /dev
  ```

  which will include various devices nodes (i.e., pseudo-files) which allow interaction with specific devices.

Kernel design (5)

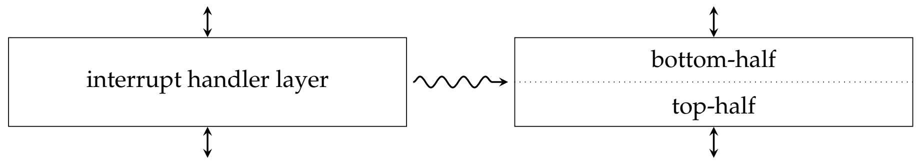
Complications and asides

- Problem:
  - we demand low interrupt latency to maximise responsiveness, *but*
  - some interrupt handling might involve high-latency tasks.

Notes:

- [10, Chapter 10] offers an explanation of how this issue is dealt with in Linux.

- Problem:
  - we demand low interrupt latency to maximise responsiveness, *but*
  - some interrupt handling might involve high-latency tasks.
- Solution: split the interrupt handler into



where

- the low(er)-latency top-half responds to interrupts,
- the high(er)-latency bottom-half is scheduled (by the top-half) to execute later

st. interrupts can be enabled in the latter but not the former.

Notes:

- [10, Chapter 10] offers an explanation of how this issue is dealt with in Linux.

- Problem: the kernel is a highly concurrent system, e.g.,
  - interrupts occur asynchronously, at *any* time, and/or
  - the kernel might itself be executing concurrently on different processor cores.

Notes:

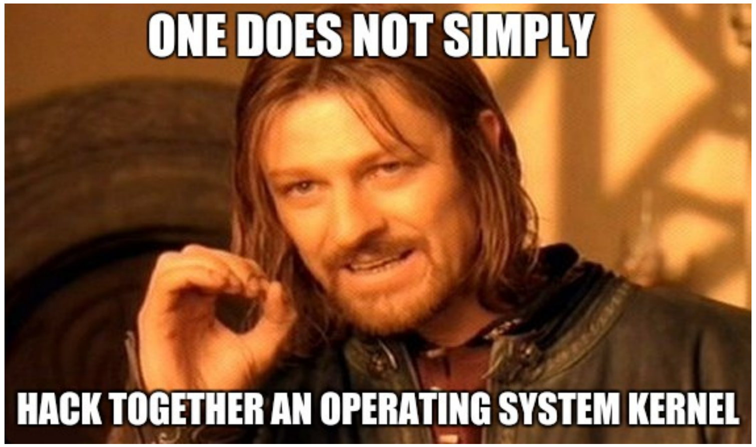
- [10, Chapter 2] and [10, Chapter 5] offer an explanation of how this issue is dealt with in Linux; note that the coarse-grained case is often called a **big kernel lock** [3], implying the *entire* kernel is a critical section.
- If we describe a function as reentrant [7], we mean that it can (correctly) execute in more than one concurrent execution context; equivalently, *during* execution of the function within one execution context it can be called again within another (i.e., it is "reentered" or "entered again" before returning). For example, this would imply the function can be called within separate execution contexts associated with execution on separate (and so *parallel*) processor cores.

- Problem: the kernel is a highly concurrent system, e.g.,
  - interrupts occur asynchronously, at *any* time, and/or
  - the kernel might itself be executing concurrently on different processor cores.
- Solution:
  - take care to allow kernel functionality (e.g., an interrupt handler) to be **reentrant**, and
  - use fine- or coarse-grain locking around critical regions to hence consistency.

Notes:

- [10, Chapter 2] and [10, Chapter 5] offer an explanation of how this issue is dealt with in Linux; note that the coarse-grained case is often called a **big kernel lock** [3], implying the *entire* kernel is a critical section.
- If we describe a function as reentrant [7], we mean that it can (correctly) execute in more than one concurrent execution context; equivalently, *during* execution of the function within one execution context it can be called again within another (i.e., it is "reentered" or "entered again" before returning). For example, this would imply the function can be called within separate execution contexts associated with execution on separate (and so *parallel*) processor cores.

Conclusions



http://memegenerator.net

Notes:

Conclusions

- Take away points:
  1. A kernel is a highly complex software artefact; we cannot hope to simple "hack together" an implementation!
     - the design process is usually iterative,
     - clear assumptions and design goals are crucial,
     - design metrics allow quantitative decisions to navigate a massive design space.
  2. The high- *and* low-level kernel design has a strong influence on the
     2.1 interface *and*
     2.2 implementation

     rather than just the former.
  3. In a sense, high-level kernel design is a philosophical decision about what a kernel *is*.

Notes:

References

[1] *Wikipedia: Architecture of Windows NT*. URL: http://en.wikipedia.org/wiki/Architecture_of_Windows_NT (see p. 50).

[2] *Wikipedia: Everything is a file*. URL: http://en.wikipedia.org/wiki/Everything_is_a_file (see pp. 42, 44, 46, 48).

[3] *Wikipedia: Giant lock*. URL: http://en.wikipedia.org/wiki/Giant_lock (see pp. 60, 62).

[4] *Wikipedia: Internet protocol suite*. URL: http://en.wikipedia.org/wiki/Internet_protocol_suite (see pp. 21, 23, 25).

[5] *Wikipedia: ioctl*. URL: https://en.wikipedia.org/wiki/Ioctl (see pp. 42, 44, 46, 48).

[6] *Wikipedia: OSI model*. URL: http://en.wikipedia.org/wiki/OSI_model (see pp. 21, 23, 25).

[7] *Wikipedia: Reentrancy*. URL: http://en.wikipedia.org/wiki/Reentrancy_(computing) (see pp. 60, 62).

[8] *Wikipedia: Tanenbaum-Torvalds debate*. URL: http://en.wikipedia.org/wiki/Tanenbaum-Torvalds_debate (see pp. 34, 36, 38, 40).

[9] *Wikipedia: Worse is better*. URL: http://en.wikipedia.org/wiki/Worse_is_better (see pp. 21, 23, 25).

[10] J. Corbet, G. Kroah-Hartman, and A. Rubini. *Linux Device Drivers*. 3rd ed. http://www.makelinux.net/ldd3/. O'Reilly, 2005 (see pp. 52, 54, 56, 58, 60, 62).

[11] A. Silberschatz, P.B. Galvin, and G. Gagne. *Operating System Concepts*. 9th ed. Wiley, 2014 (see pp. 22, 24, 26).

[12] A.S. Tanenbaum and H. Bos. *Modern Operating Systems*. 4th ed. Pearson, 2015 (see pp. 22, 24, 26, 42, 44, 46, 48).

[13] *Information technology – Portable Operating System Interface (POSIX) Base Specifications, Issue 7*. Internation Organization for Standardization (ISO) and International Electrotechnical Commission (IET) Standard 9945:2009. 2009. URL: http://www.iso.org (see pp. 29, 31).

[14] *International Standard - Information technology Portable Operating System Interface (POSIX) Base Specifications, Issue 7*. Institute of Electrical and Electronics Engineers (IEEE) 9945-2009. 2009. URL: http://standards.ieee.org (see pp. 29, 31).

[15] *Linux Standard Base (LSB) core specification 3.1 – Part 1: Generic specification*. Internation Organization for Standardization (ISO) and International Electrotechnical Commission (IET) Standard 23360-6:2006. 2006. URL: http://www.iso.org (see pp. 29, 31).

Notes:

References

[16] *Standard for Information Technology - Portable Operating System Interface (POSIX)*. Institute of Electrical and Electronics Engineers (IEEE) 1003.1-2008. 2008. URL: http://standards.ieee.org (see pp. 29, 31).

[17] *System V Application Binary Interface*. Tech. rep. 1995. URL: http://www.sco.com/developers/devspecs (see pp. 29, 31).

[18] *System V Interface Definition*. Tech. rep. Volume 1a. 1995. URL: http://www.sco.com/developers/devspecs (see pp. 29, 31).

[19] *System V Interface Definition*. Tech. rep. Volume 1b. 1995. URL: http://www.sco.com/developers/devspecs (see pp. 29, 31).

[20] *System V Interface Definition*. Tech. rep. Volume 2. 1995. URL: http://www.sco.com/developers/devspecs (see pp. 29, 31).

[21] *System V Interface Definition*. Tech. rep. Volume 3. 1995. URL: http://www.sco.com/developers/devspecs (see pp. 29, 31).

[22] ISO/IEC The Austin Group (IEEE and Open Group). *Single UNIX Specification (SUS)*. Tech. rep. http://www.unix.org/version4 (see pp. 29, 31).

[23] *X/Open Portability Guide (XPG)*. Tech. rep. Volume C204: System Interface Definitions (XBD). 1992. URL: http://www.opengroup.org (see pp. 29, 31).

[24] *X/Open Portability Guide (XPG)*. Tech. rep. Volume C202: System Interfaces and Headers (XSH). 1992. URL: http://www.opengroup.org (see pp. 29, 31).

[25] *X/Open Portability Guide (XPG)*. Tech. rep. Volume C203: Commands and Utilities (XCU). 1992. URL: http://www.opengroup.org (see pp. 29, 31).

[26] V. Atlidakis et al. "POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing". In: *European Conference on Computer Systems (EuroSys)*. 19. 2016 (see pp. 10, 12, 14, 16, 18, 20).

[27] A. Chou et al. "An empirical study of operating systems errors". In: *ACM Symposium on Operating System Principles (SOSP)*. 2001, pp. 73–88 (see pp. 10, 12, 14, 16, 18, 20).

[28] F.J. Corbató, J.H. Saltzer, and C. T. Clingen. "Multics: The first seven years". In: 1972, pp. 571–582 (see pp. 21, 23, 25).

[29] F.J. Corbató and V. A. Vyssotsky. "Introduction and overview of the Multics system". In: 1965, pp. 185–196 (see pp. 21, 23, 25).

[30] D.R. Engler and M.F. Kaashoek. "Exterminate all operating system abstractions". In: *Hot Topics in Operating Systems (HotOS-V)*. 1995, pp. 78–83 (see pp. 34, 36, 38, 40).

Notes:

References

[31] 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)*. 1995, pp. 251–266 (see pp. 34, 36, 38, 40).

[32] G. Klein et al. "seL4: formal verification of an OS kernel". In: *ACM Symposium on Operating System Principles (SOSP)*. 2009, pp. 207–220 (see pp. 10, 12, 14, 16, 18, 20).

[33] B.W. Lampson. "Hints for computer system design". In: *ACM Symposium on Operating System Principles (SOSP)*. 1983, pp. 33–48 (see pp. 22, 24, 26).

[34] B.W. Lampson and H.E. Sturgis. "Reflections on Operating System Design". In: *Communications of the ACM (CACM)* 19.5 (1976), pp. 251–265 (see pp. 22, 24, 26).

[35] D.M. Ritchie. "The UNIX System: Evolution of the UNIX Time-sharing System". In: *Bell Laboratories Technical Journal* 63.8 (1984), pp. 1577–1593 (see pp. 21, 23, 25).

[36] D.M. Ritchie and K. Thompson. "The UNIX Time-Sharing System". In: *Communications of the ACM (CACM)* 17.7 (1974), pp. 365–375 (see pp. 21, 23, 25).

[37] M.M. Swift, B.N. Bershad, and H.M. Levy. "Improving the reliability of commodity operating systems". In: *ACM Symposium on Operating System Principles (SOSP)*. 2003, pp. 207–222 (see pp. 10, 12, 14, 16, 18, 20).

[38] K. Thompson. "UNIX Implementation". In: *Bell System Technical Journal* 57.6 (1978), pp. 1931–1946 (see pp. 21, 23, 25).

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

Notes: