Systemsoftware

1. Einführung

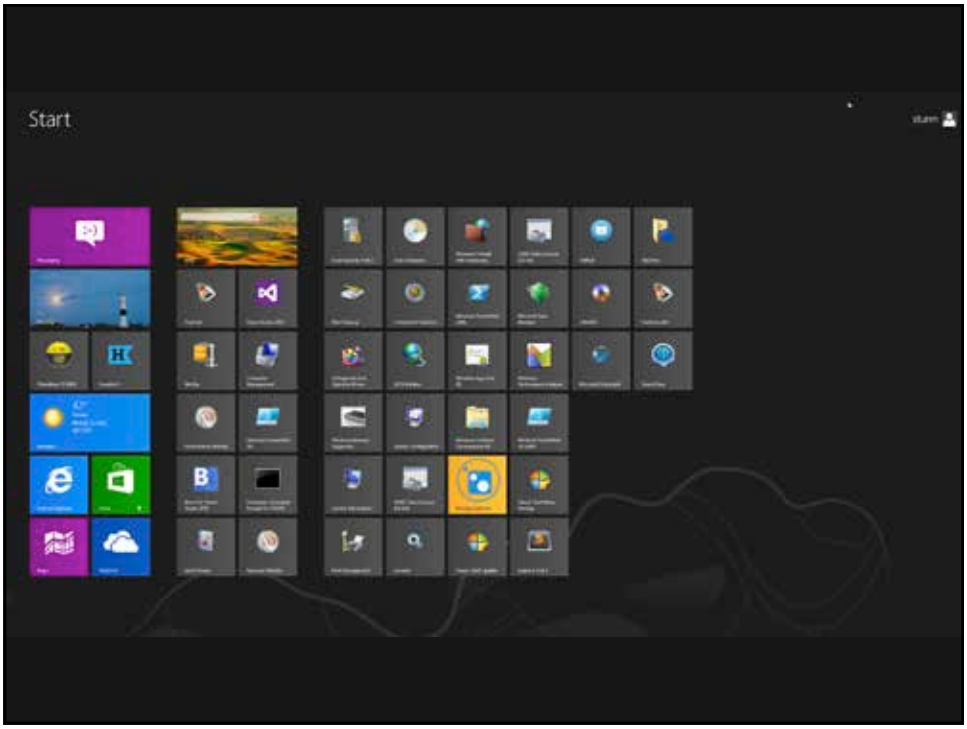
Peter Sturm
sturm@uni-trier.de
+49 651 201 3311

Organisatorisches

- Mittwochs, 10 – 11.30 Uhr
  - Grundlagenvorlesung
  - HS11
- Es gibt Übungen
- Klausur: Februar 2016
- Folienkopien und mehr
  - Asysob-Blog
  - studip


Bekannte Vertreter


Recycle Bin
Feedback Button
Default Programs
Internet Explorer
TS Workspace
Windows Defender
Windows DVD Maker
Windows Fax and Scan
Windows Live Messenger Download
Windows Mail
Windows Media Center
Windows Media Player
Windows Meeting Space
Windows Movie Maker
Windows Photo Gallery
Windows Update
XPS Viewer EP
Accessories
Administrative Tools
Extras and Upgrades
Games
Maintenance
Programs
Startup
Back
Connect to a TS Workspace
Paul
Documents
Pictures
Music
Games
Recent Items
Computer
Network
Connect To
Control Panel
Default Programs
Help and Support
Windows 7
Rate your Experience
View headlines
Personalization
11:14 AM


Start

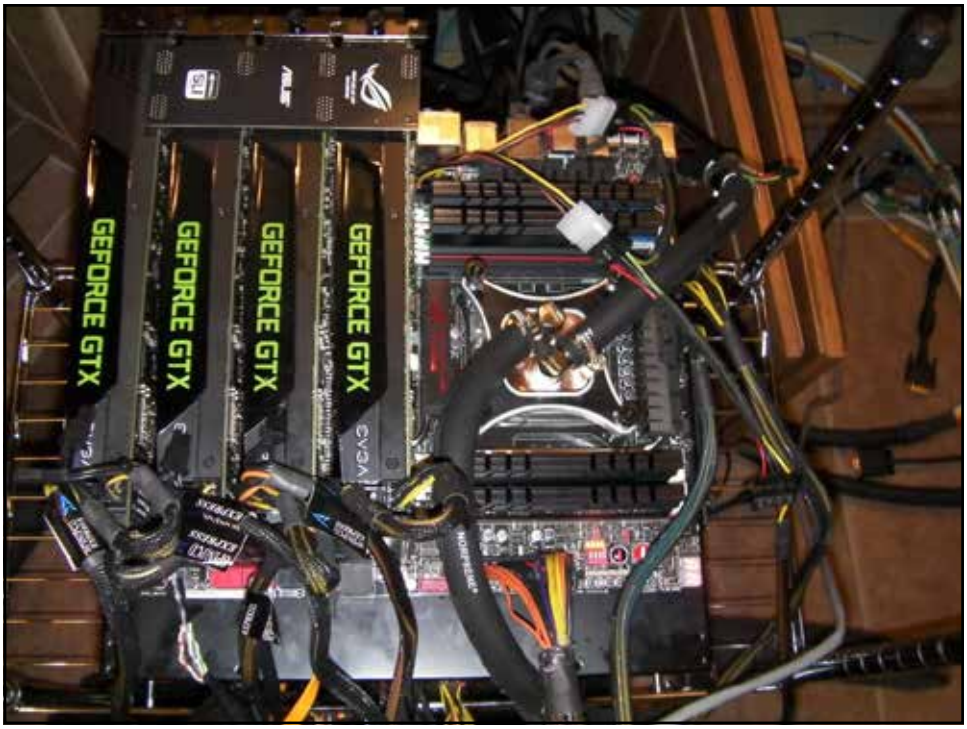

UHU-Linux

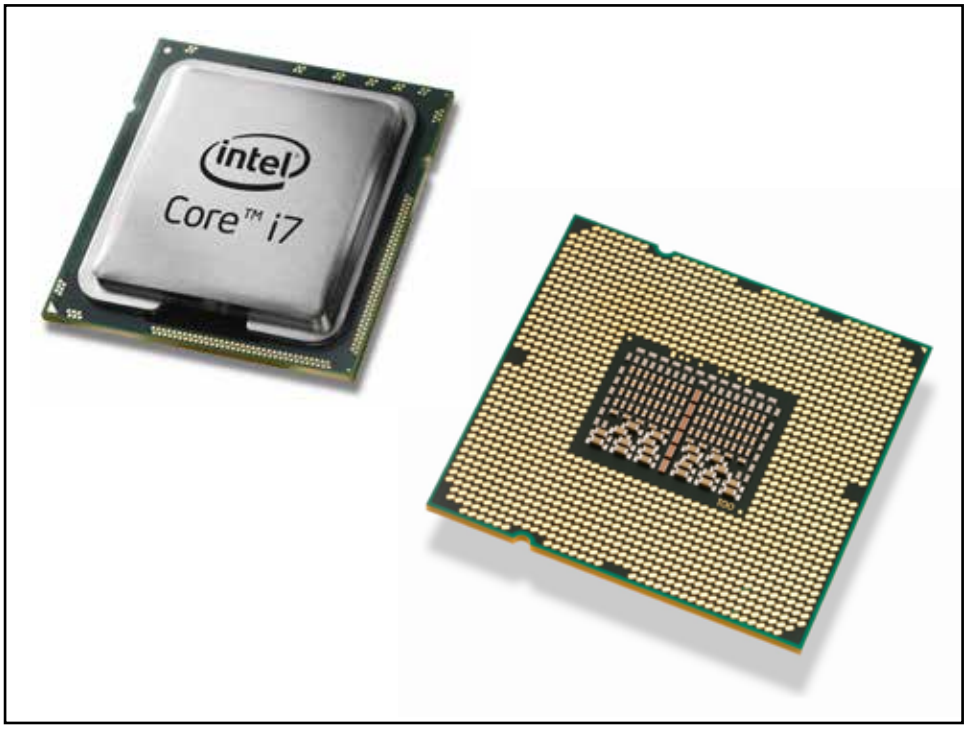

Ein gutes Betriebssystem ist unsichtbar?!

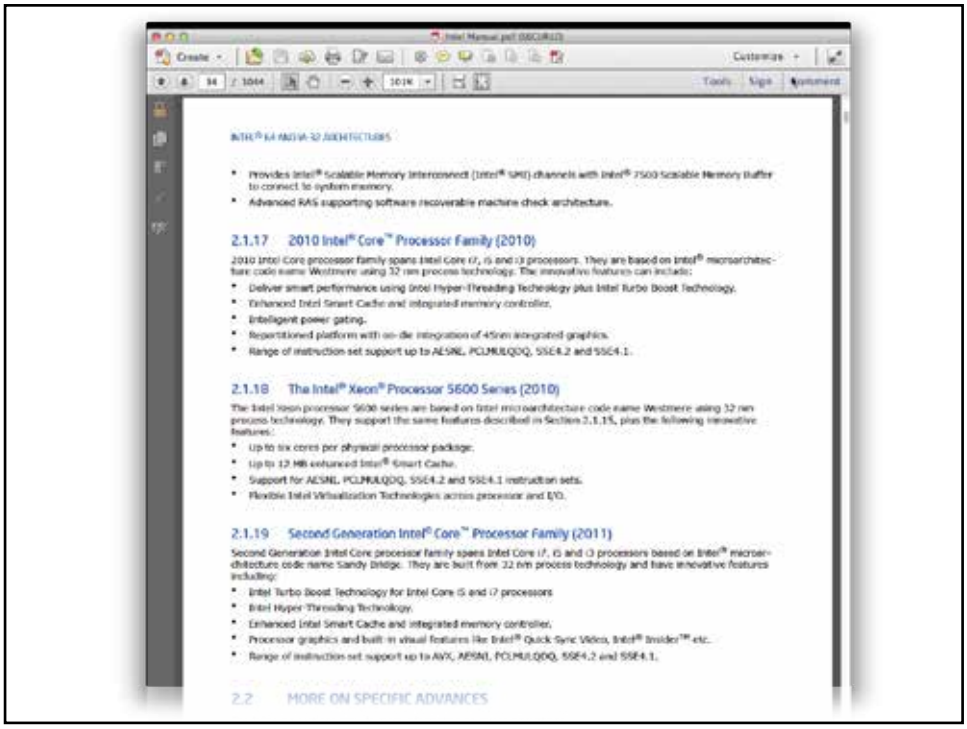
Hardware?

- Datenspeicherung (nicht-flüchtig, langsam)
  - Festplatten, SSD, CD/DVD/BD
- Datenspeicherung (flüchtig, schnell)
  - Hauptspeicher (RAM)
- Datenverarbeitung
  - Prozessor(en): CPU
  - Graphikkarte: GPU
  - Soundkarte usw.
- Datenkommunikation


ASUS
RAMPAGE IV EXTREME


GEFORCE GTX
GEFORCE GTX
GEFORCE GTX
GEFORCE GTX
EVGA





- Provides Intel® Scalable Memory Interconnect (Intel® SMI) channels with Intel® 7500 Scalable Memory Buffer to connect to system memory.
- Advanced RAS supporting software recoverable machine check architecture.

2.1.17 2010 Intel® Core™ Processor Family (2010)

2010 Intel Core processor family spans Intel Core i7, i5 and i3 processors. They are based on Intel® microarchitecture code name Westmere using 32 nm process technology. The innovative features can include:

- Deliver smart performance using Intel Hyper-Threading Technology plus Intel Turbo Boost Technology.
- Enhanced Intel Smart Cache and integrated memory controller.
- Intelligent power gating.
- Repartitioned platform with on-die integration of 45nm integrated graphics.
- Range of instruction set support up to AESNI, PCLMULQDQ, SSE4.2 and SSE4.1.

2.1.18 The Intel® Xeon® Processor 5600 Series (2010)

The Intel Xeon processor 5600 series are based on Intel microarchitecture code name Westmere using 32 nm process technology. They support the same features described in Section 2.1.15, plus the following innovative features:

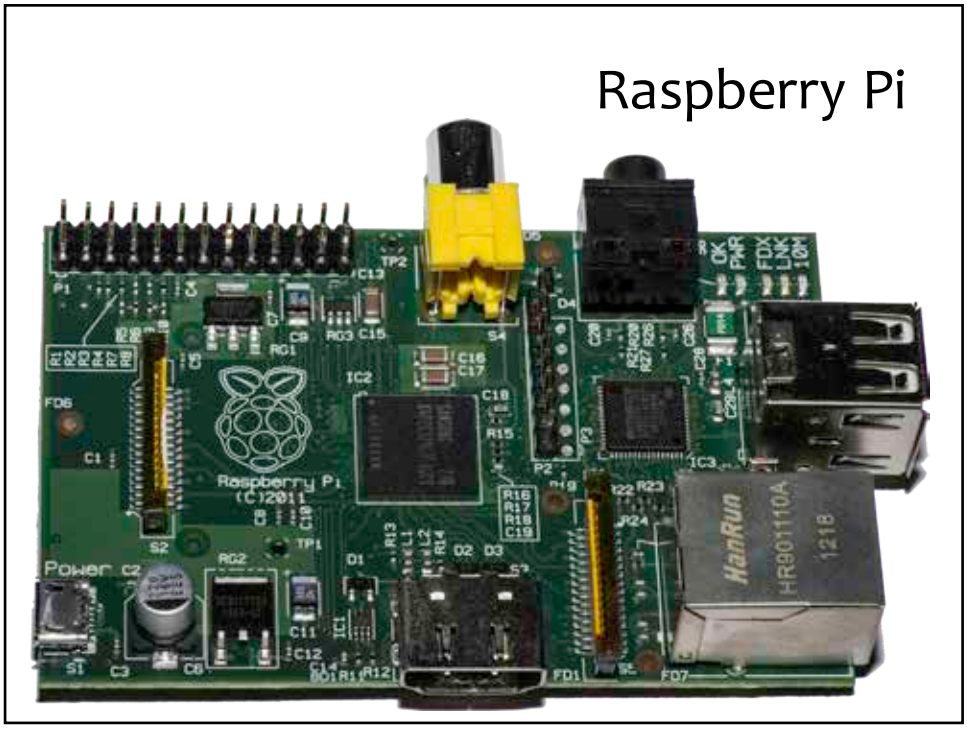
- Up to six cores per physical processor package.
- Up to 12 MB enhanced Intel® Smart Cache.
- Support for AESNI, PCLMULQDQ, SSE4.2 and SSE4.1 instruction sets.
- Flexible Intel Virtualization Technologies across processor and I/O.

2.1.19 Second Generation Intel® Core™ Processor Family (2011)

Second Generation Intel Core processor family spans Intel Core i7, i5 and i3 processors based on Intel® microarchitecture code name Sandy Bridge. They are built from 32 nm process technology and have innovative features including:

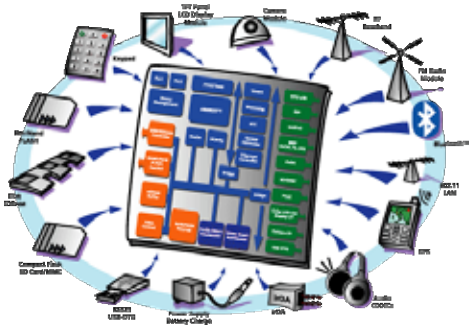
- Intel Turbo Boost Technology for Intel Core i5 and i7 processors
- Intel Hyper-Threading Technology.
- Enhanced Intel Smart Cache and integrated memory controller.
- Processor graphics and built-in visual features like Intel® Quick Sync Video, Intel® Insider™ etc.
- Range of instruction set support up to AVX, AESNI, PCLMULQDQ, SSE4.2 and SSE4.1.

2.2 MORE ON SPECIFIC ADVANCES


Raspberry Pi
Raspberry Pi
(C)2011
Power
HanRun
HR901110A
1218

SOC

- System on a Chip
- Chip enthält
  - CPU
  - ggf. GPU
  - I/O-Controller
  - AD/DA-Wandler
  - Speicher
- Rest sind Leitungen und Anschlüsse



Pi again

GPIO (General Purpose Input Output)

SPI (Serial Peripheral Interface Bus)

UART (Universal Asynchronous Receiver/Transmitter)

I2C (inter-Integrated Circuit)

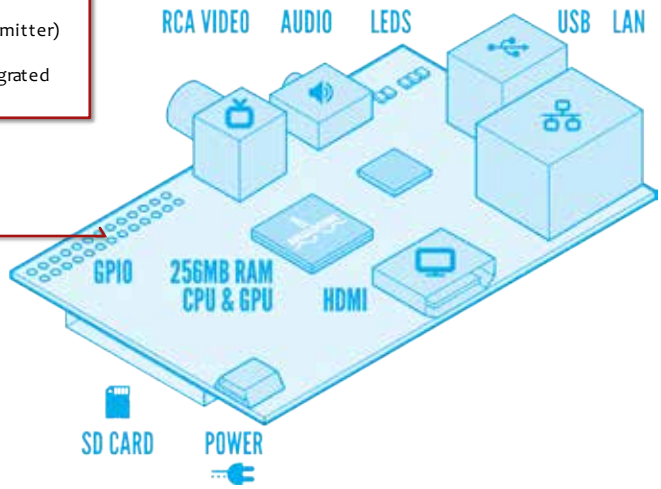
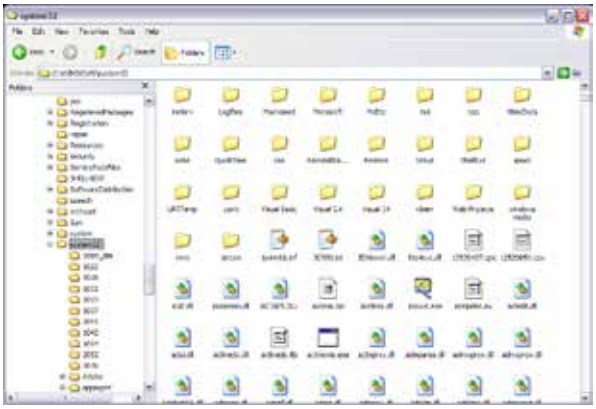
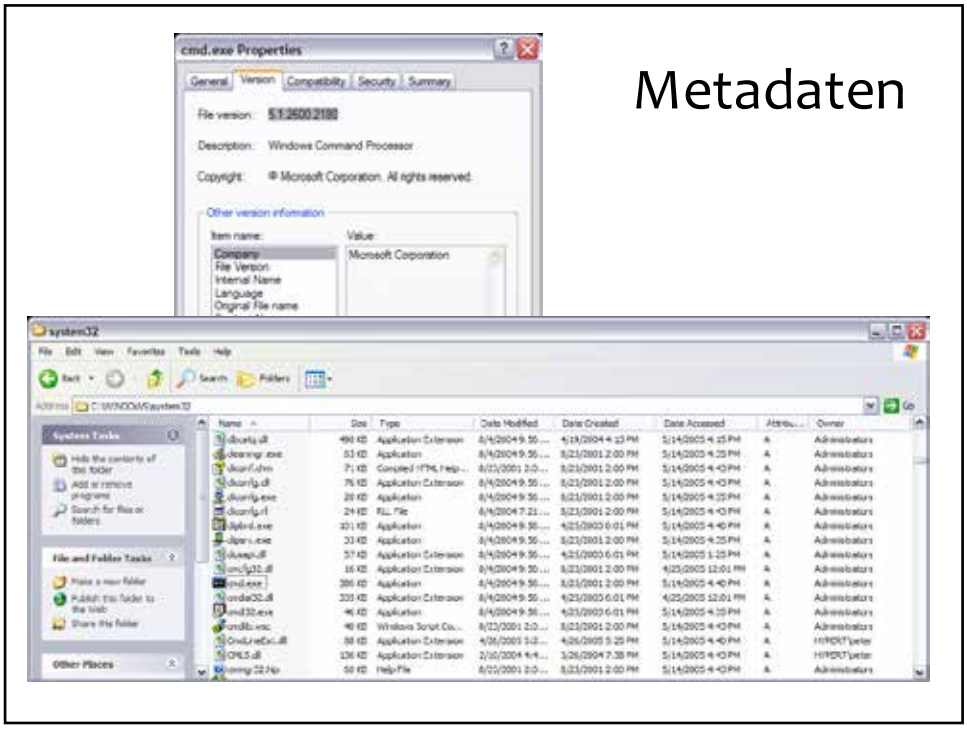


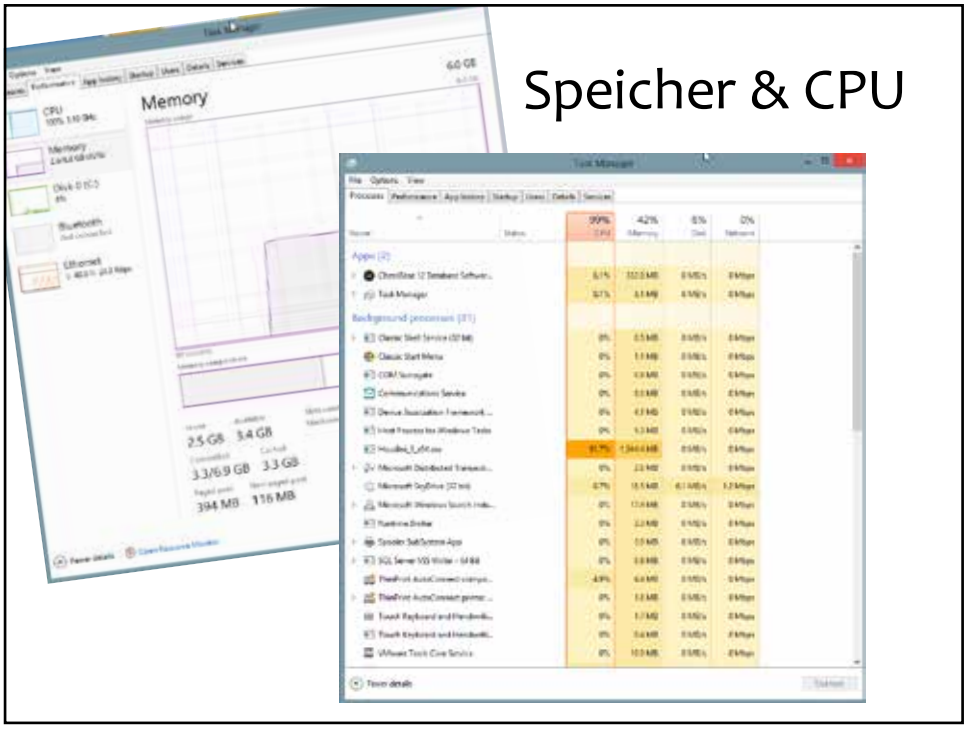
Diagramm von Paul Beech (www.raspberrypi.org)

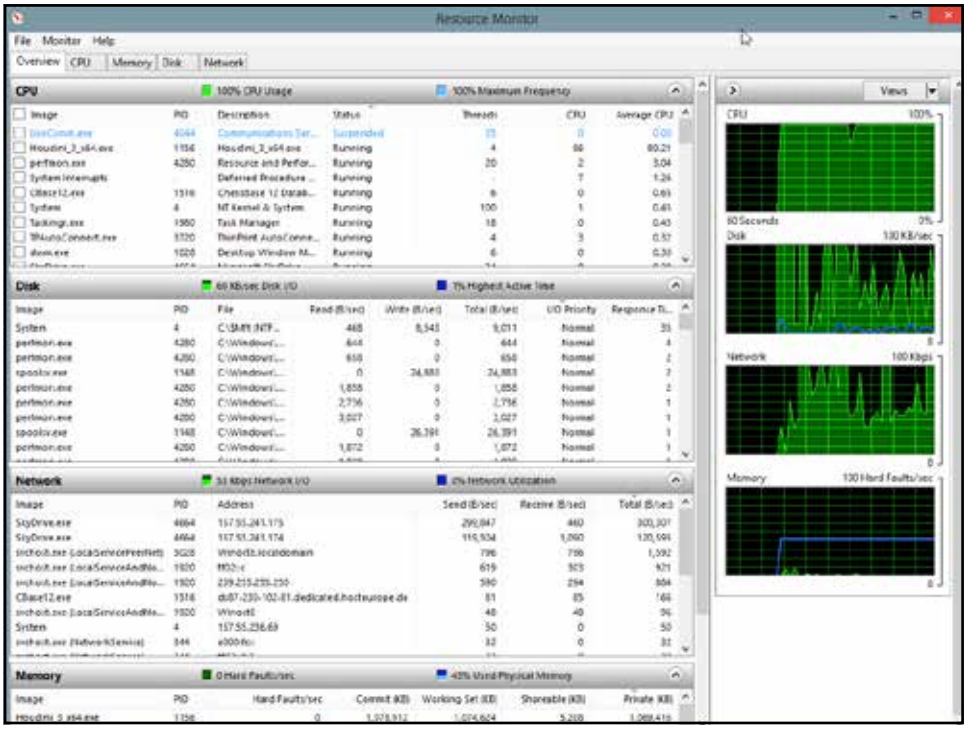
Dateien und Verzeichnisse



- Graphischer und textueller Zugang


Metadaten
cmd.exe Properties
General
Version
Compatibility
Security
Summary
File version:
Description:
Windows Command Processor
Copyright:
© Microsoft Corporation. All rights reserved.
Other version information
Item name:
Value:
Company
File Version
Internal Name
Language
Original File name
Microsoft Corporation
system32


Speicher & CPU
Memory
6.0 GB
CPU
Memory
Disk 0 (C:)
Bluetooth
Ethernet
2.5 GB
3.4 GB
3.3/6.9 GB
3.3 GB
394 MB
116 MB



Mediatorfunktion

User

Programme

- Vermitteln zwischen
  - Hardware
  - Programmen und Benutzern
- Ziele
  - Von technischen Details abstrahieren
  - Einfache Benutzbarkeit
  - Keine Monopolisierung
  - Vermeidung von Engpässen (Virtualisierung)
- Schnittstellen
  - User Interface (Shell)
  - Programmierung (API)

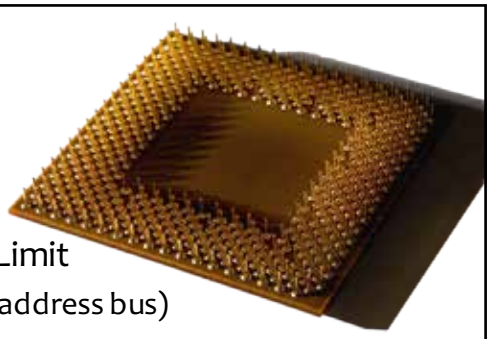
Hardware

CPUs
Bus
MMU
Speicher Controller
Disks
Graphics
3D
Audio
Video
CD Writer
...
Netzwerk

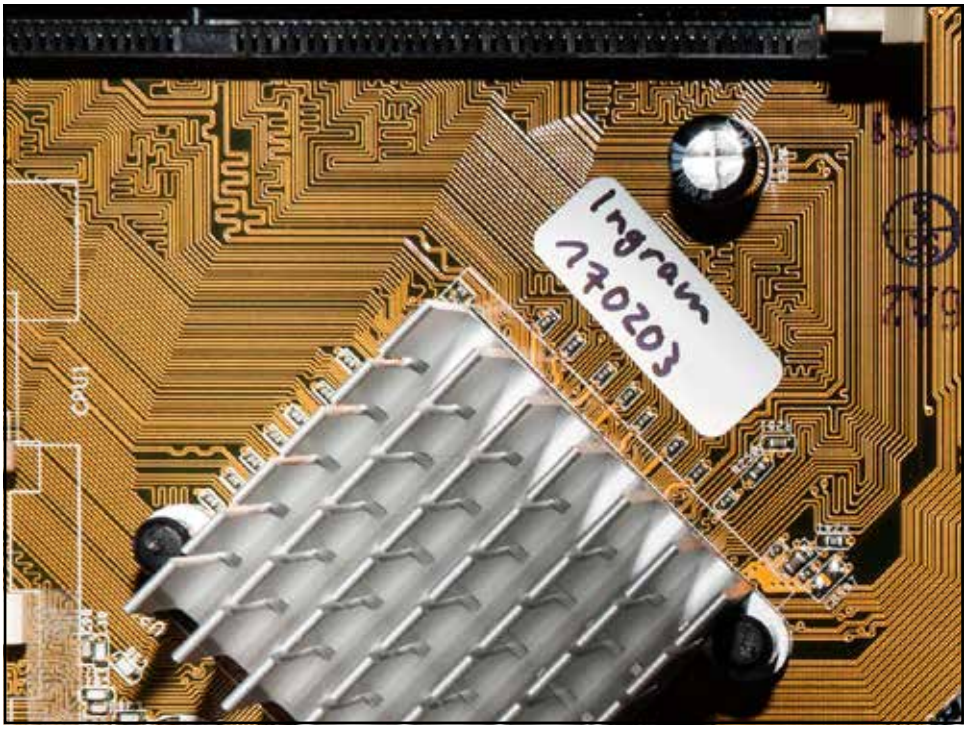
Das Kontinuum

- Aktivität (Zeit)
  - Abfolge ausgeführter Instruktionen
  - Kontrollfluß oder Thread
  - Unterstützung für mehrere Threads
  - Concurreny Control
- Raum
  - Menge der adressierbaren Speicherzellen
  - Adreßraum oder "Adress space"
  - Unterstützung für mehrere Adreßräume

Adreßraum



- CPU definiert ein theoretisches, oberes Limit
  - Anzahl der Adreß-Pins (address bus)
- Beispiele
  - 32 bit Architektur (Intel Pentium, AMD Athlon XP, …)
    - 32 Adressleitungen = 4 GByte Adreßraum
  - 64 bit Architektur (AMD Athlon 64, Intel Itanium, …)
    - 64 Leitungen = 4194304 TByte theoretischer Adreßraum
    - Real limitiert auf ~40 Pins (Server 48 Pins)

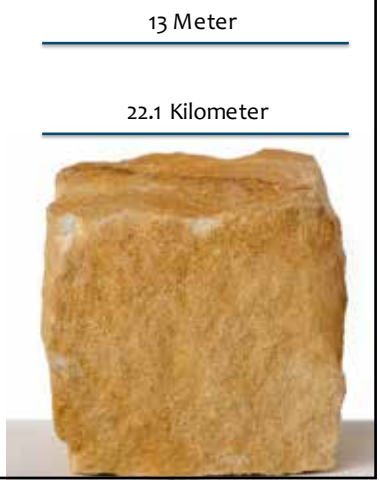


Vorstellbar?

- 1 Adresse = 1 Smartie = 1 Gramm
- Wie schwer sind 2^{32} Smarties?
- Wie schwer sind 2^{64} Smarties?

Lösung

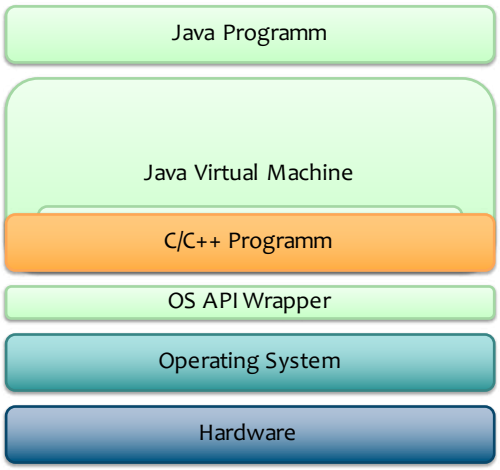
- Gewicht von 2^{32} Smarties
  - 4294 Tonnen
- Gewicht von 2^{64} Smarties
  - 18,446,744,073,709 Tonnen



Was ist im Adreßraum?

- Instruktionen (Maschinenbefehle)
- Programmdaten
  - Verschiedene Bereiche

Back to the roots

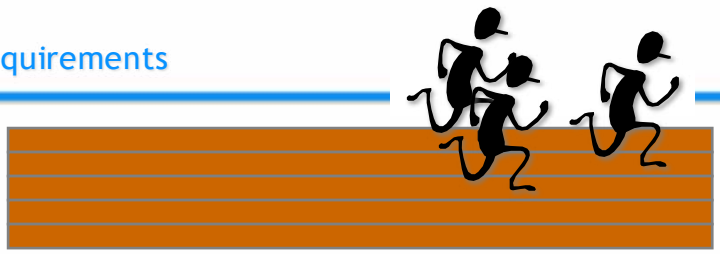


Operating Systems

Introduction
Space-Time Continuum

System Software Group
University of Trier
D-54286 Trier, Germany
Contact: Peter Sturm
sturm@uni-trier.de

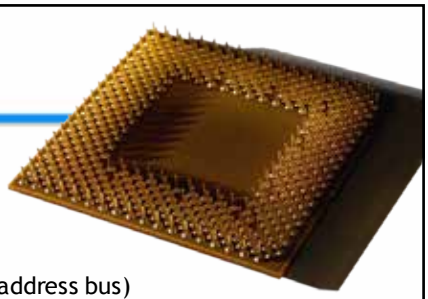
Basic Requirements



- Activity
  - Sequence of instructions executed in time = Thread of control
  - Support for multiple threads
  - Cooperation or concurrency among threads
- Space
  - Set of accessible memory cells = Address space
  - Support for multiple address spaces
  - Communication among shared memory

Operating Systems

Adress Space



- Theoretical limit defined by the CPU architecture
  - Specific number of address pins (address bus)
- Examples
  - 32 bit architecture (Intel Pentium, AMD Athlon XP, …)
    - 32 address lines = 4 GByte address space
  - 64 bit architecture (AMD Athlon 64, Intel Itanium, …)
    - 64 addresses = 4194304 TByte theoretical address space
    - Address bus of any 64 bit CPU limited to ~40 pins

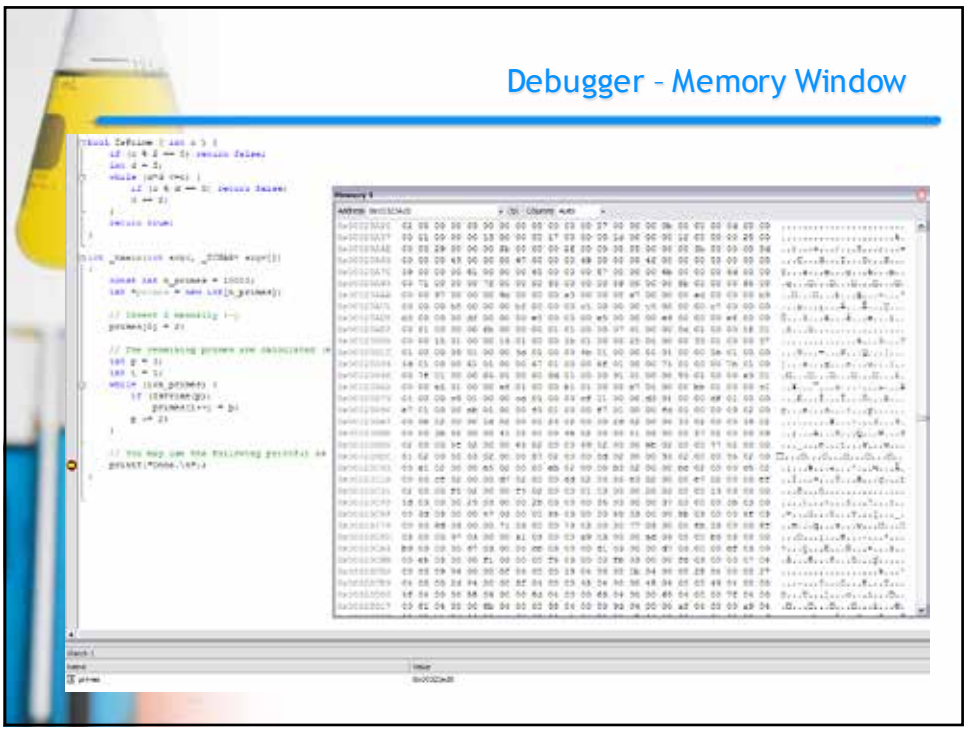
Operating Systems

What's inside the Address Space?

- Program instructions
  - Machine instructions executed by CPU directly
  - Intermediate instructions executed by some virtual machine
  - Application code, runtime, libraries, operating system
- Program data
  - Static and dynamic data structures (the latter are so-called heaps)
  - program state reflecting the nested call of methods resp. procedures (stack)

Operating Systems

Debugger – Memory Window

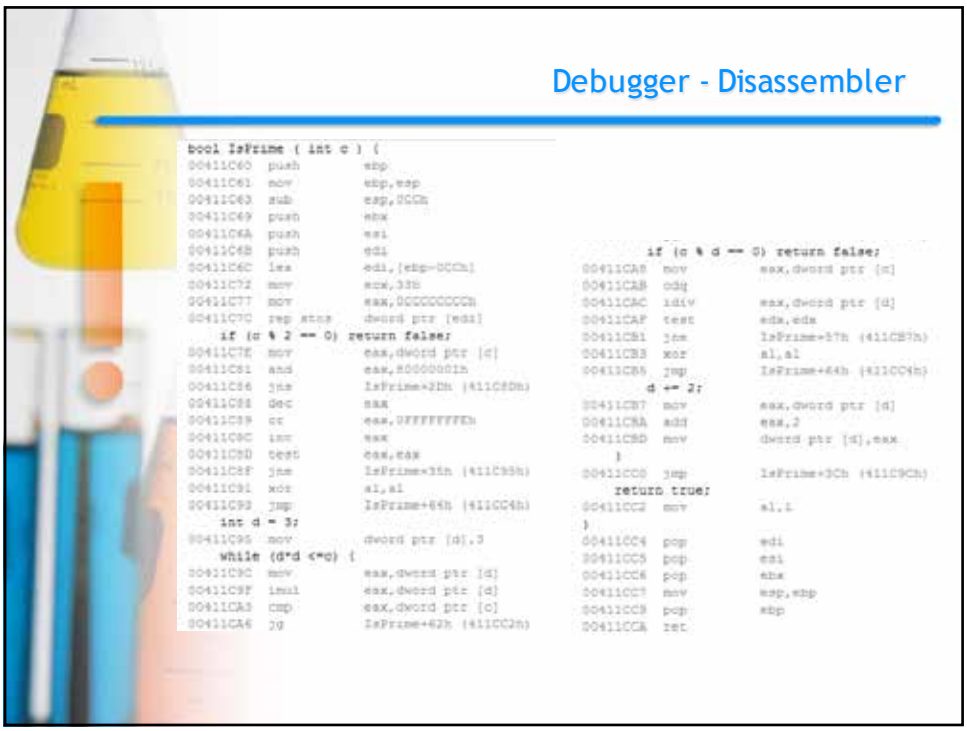


Time

- Sequence of instructions executed by CPU over time
  - Machine instruction
  - Virtual instructions (e.g. IL) ⇒ Even more machine instructions

  = **Thread of Control**
- Instructions are tied to memory addresses
  - Execution starts with a well-defined address
  - Next instruction to be executed:
    - Sequential next address (default)
    - Address defined in instruction itself
    - Next Address will be calculated during instruction execution (conditional instruction/branch)
- Content of all memory cells inside address space storing instructions defines the concrete application

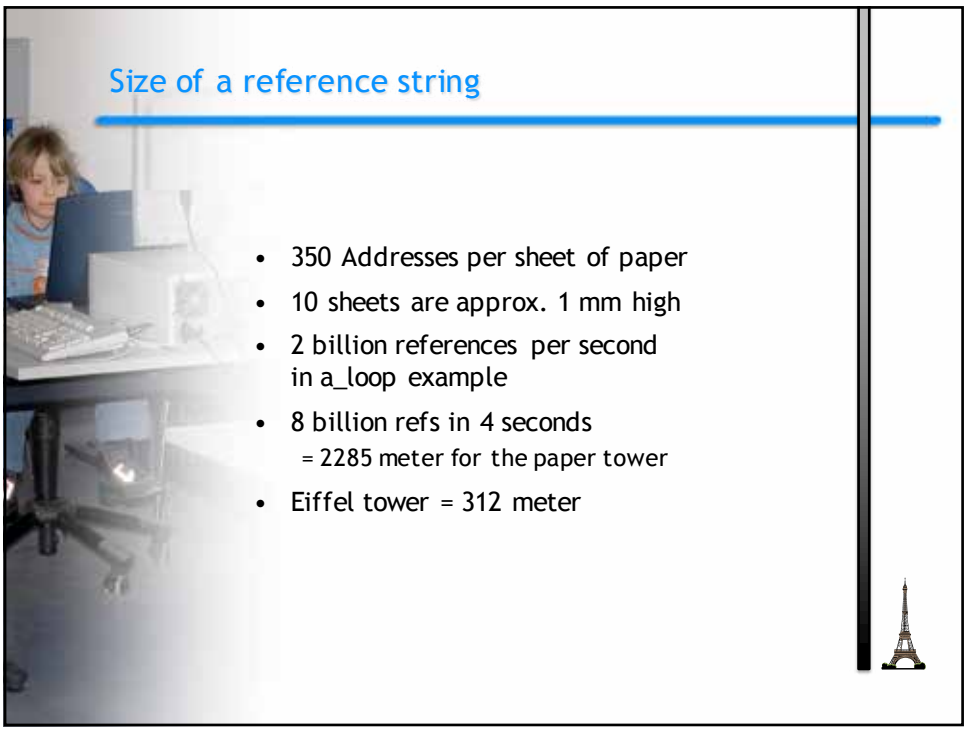
Operating Systems



Reference String

- From the “outside”, the execution of instructions is recognized as a series of memory references
  - References to instructions
  - References to data stored in main memory
  - References to data stored in stack
- Reference string = series of addresses the CPU is referencing while executing a program
- Reference string are huge
  - Caches must be disabled to observe any memory reference
  - With a modern CPU some billion references per second possible
  - Several GByte collected addresses per second
  - Dedicated hardware required to catch the string during program execution

Operating Systems



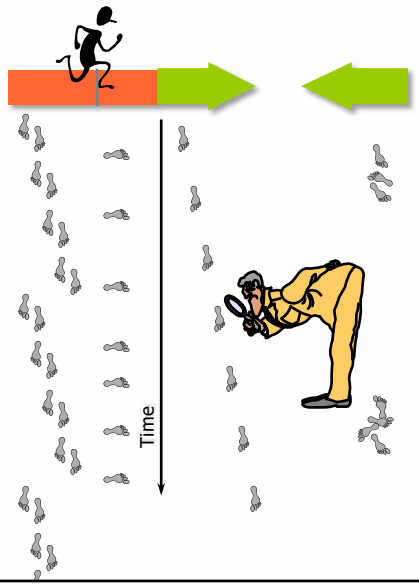
Reference Locality

- Most programs running on classical "von Neumann" architectures exhibit some "regular" access pattern to memory
  - Repeated updates to counting variables inside loops
  - Repeated execution of instructions forming a loop
  - Access to neighboring elements in an array
- Consequences
  - If some address *a* is referenced, there is a high probability that *a* will be referenced in the near future again
  - If some address *a* is referenced, there is a high probability that addresses around *a* will be referenced in the near future
- Many hardware and software mechanisms depend on this behavior
  - Primary reason that caches have hit rates close to 1
  - Virtual memory can be realized efficiently

Operating Systems

Reference Locality

- Instructions are repeated
- Variables are referenced repeatedly
- If one address is referenced, nearby addresses will be referenced with high probability
- Reasons:
  - Sequential program execution
  - Procedure nesting is modest
  - Short loops
  - Frequent processing of arrays, list, and records



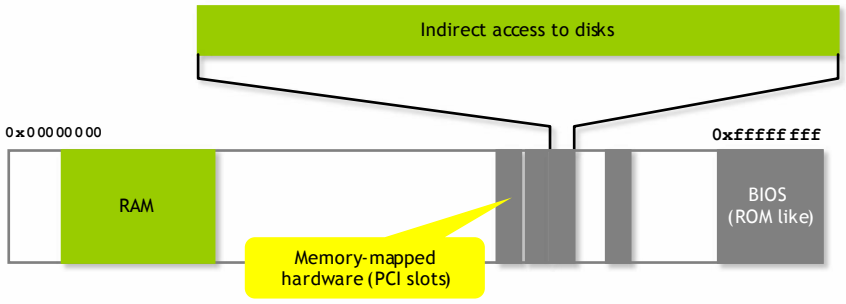
Operating Systems

The Ideal Translation Table

MEMORY

Operating Systems

The Physical Address Space

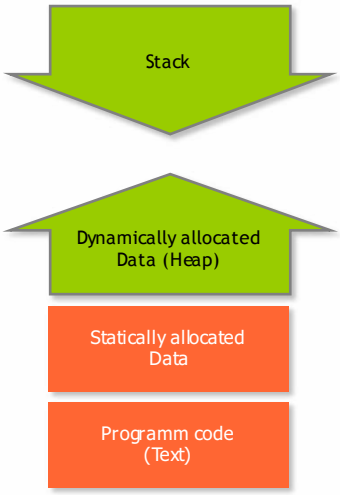


- Physical address space is chaotic and contains many gaps
  - Interrupt vectors
  - Some RAM
  - ROM to boot system (BIOS in case of PC)
  - Memory-mapped I/O

Operating Systems

Ideal Address Space Usage

- Text is relatively small
  - Typical code sizes?
  - Maximum code sizes?
- Stack in most cases small, but
  - Sometimes more space required
- Separate heap and stack
- Gaps in physical memory invisible
  - Linear address space
- Maxmium address space usable for some applications
  - No matter how much RAM installed?
- Exclusive usage by application
  - Isolation from other applications



Operating Systems

Program Size

- Determine the program size of common executables
- Identify the biggest executable on your machine

Program Sizes

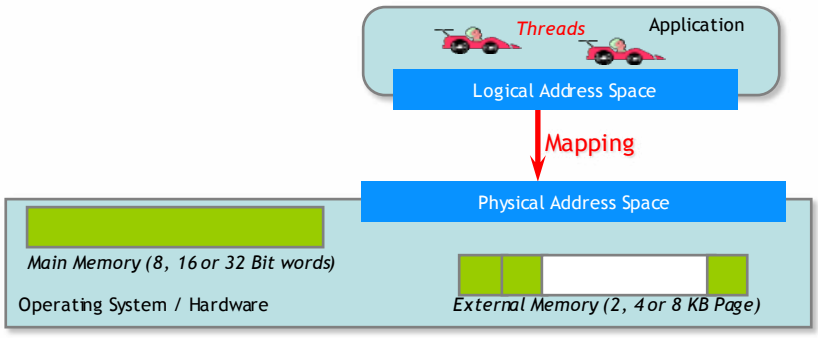
GdUve.exe	C:\Progra...	24,552 KB	Application	1/15/2004 7:08 PM	12/3/2... A
ImageReady.exe	C:\Progra...	19,188 KB	Application	10/15/2003 2:10 AM	12/5/2... A
ActOfWar.exe	C:\Progra...	18,013 KB	Application	3/14/2005 12:01 PM	4/26/2... A
Photoshop.exe	C:\Progra...	17,532 KB	Application	10/17/2003 1:59 AM	12/5/2... A
XMLSpy.exe	C:\Progra...	17,460 KB	Application	3/9/2005 4:55 AM	3/9/20... A
Uru.exe	C:\Progra...	15,267 KB	Application	3/12/2004 9:56 AM	3/12/2...
nero.exe	C:\Progra...	14,989 KB	Application	1/5/2005 2:47 PM	2/19/2... A
iPod Updater 2004-04-28.exe	C:\Progra...	14,288 KB	Application	4/24/2004 5:20 PM	4/24/2... A
SBA2_EAX4DRV_031031.exe	C:\Progra...	13,896 KB	Application	1/21/2004 5:32 PM	1/21/2...
solce20sd2bp2.exe	C:\Progra...	13,329 KB	Application	8/29/2002 11:36 PM	8/29/2...
solce20sd2bp1.exe	C:\Progra...	13,032 KB	Application	8/29/2002 11:18 PM	8/29/2...
icace.exe	C:\WINCE...	12,877 KB	Application	3/21/2003 12:00 AM	3/21/2...
StyleVision.exe	C:\Progra...	12,368 KB	Application	3/9/2005 4:43 AM	3/9/20...
icace.exe	C:\WINCE...	11,972 KB	Application	3/21/2003 12:00 AM	3/21/2...
WINWORD.EXE	C:\Progra...	11,766 KB	Application	6/10/2004 10:29 PM	6/10/2... A
WINWORD.EXE	C:\WINDO...	11,756 KB	Application	8/6/2003 2:24 PM	8/6/20... R
Illustrator.exe	C:\Progra...	11,473 KB	Application	10/14/2003 6:23 AM	12/5/2... A
61.76_winxp2k_english_whql.exe	C:\Drivers\...	11,324 KB	Application	7/23/2004 2:47 PM	7/23/2...
Recode.exe	C:\Progra...	10,889 KB	Application	12/8/2004 5:15 PM	7/12/2... A
56.72_winxp2k_english_whql.exe	C:\Drivers\...	10,481 KB	Application	7/10/2004 6:19 PM	7/10/2...
Acrobat.exe	C:\Progra...	9,981 KB	Application	11/4/2003 12:58 AM	11/4/2... A
imperiumxp.exe	C:\Progra...	9,899 KB	Application	4/25/2003 6:19 PM	4/25/2... A
EXCEL.EXE	C:\Progra...	9,845 KB	Application	5/19/2004 2:58 AM	5/19/2... A
EXCEL.EXE	C:\WINDO...	9,838 KB	Application	8/13/2003 3:34 AM	8/13/2... R
WorldBuilder.exe	C:\Progra...	9,513 KB	Application	8/30/2003 9:55 PM	8/30/2... A
DMSP021216.exe	C:\Progra...	9,465 KB	Application	7/22/2002 4:31 PM	7/22/2...
43.45_win2xp_english.exe	C:\Drivers\...	8,964 KB	Application	4/25/2003 5:44 PM	4/25/2...
iTunes.exe	C:\Progra...	8,823 KB	Application	5/4/2005 5:10 PM	5/4/20... A
patchsdk.exe	C:\Progra...	8,780 KB	Application	1/15/2005 1:50 PM	1/15/2...
UruExplorer.exe	C:\Progra...	8,456 KB	Application	3/12/2004 10:05 AM	3/12/2...
ChessProgram8.exe	C:\Progra...	7,784 KB	Application	3/5/2003 8:11 PM	3/5/20... A
Thumbs.exe	C:\Progra...	7,688 KB	Application	11/24/2003 7:38 PM	2/6/20... A

Supporting Multiple Address Spaces

- Each virtual address space as big as the physical one (e.g. 32 Bit)
- Any solutions in mind?
- Hints:
  - Typically, most of the address space is unused
  - Programs exhibit a specific behavior (Reference locality)

Operating Systems

Motivation



- Physical Addressing
  Logical and physical address space are identical
- Logical (Virtual) Addressing
  Logical and physical address space may and mostly will differ

Operating Systems

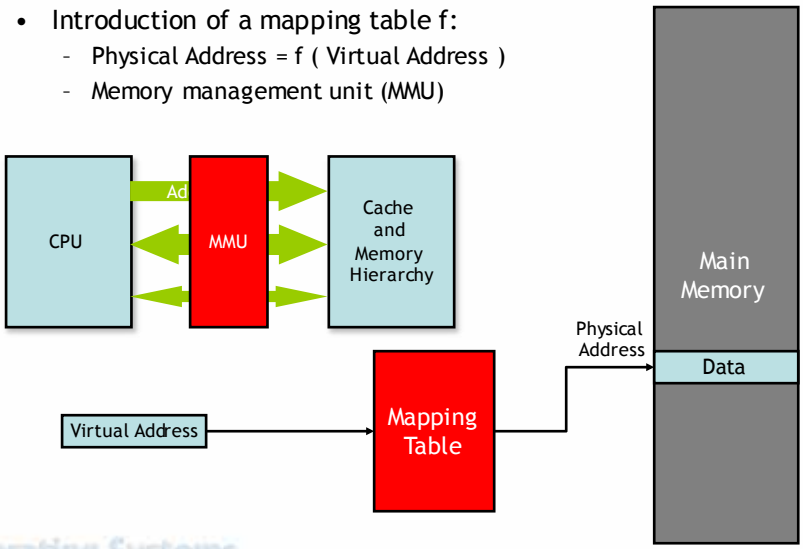
Goal

- Support for multiple address spaces
  - With multiple threads (multi-threaded programming)
- Internal protection (inside the address space)
  - Protection against stack overflows and other attacks
  - Protection against program flaws
- External protection (between address spaces)
  - Protect OS from malicious or faulty applications
  - Protect application from other applications
- Increase virtual memory beyond installed main memory
  - Integrate external persistent memory (disks)
  - Paging /Swaping
- Support for hardware-related servers
  - Memory-mapped I/O accessible in some virtual address spaces

Operating Systems

Basic Idea

- Introduction of a mapping table f:
  - Physical Address = f (Virtual Address)
  - Memory management unit (MMU)



Operating Systems

Emerging Problems

- Size of table
  - Mapping 32 bit virtual address to 32 bit physical address:
    - 2^{32} entries in table required
    - 4 Byte for each entry (32 bit physical address)

$$2^{32} \cdot 4 = 2^{34}$$

    - 16 Gbyte memory consumption
    - Sparse array, so compression strategies are possible
- Performance
  - 1 additional memory access (index-based access)
  - Complex access algoritm in case of compressed sparse array
- Conclusion: Good idea, but too expensive!

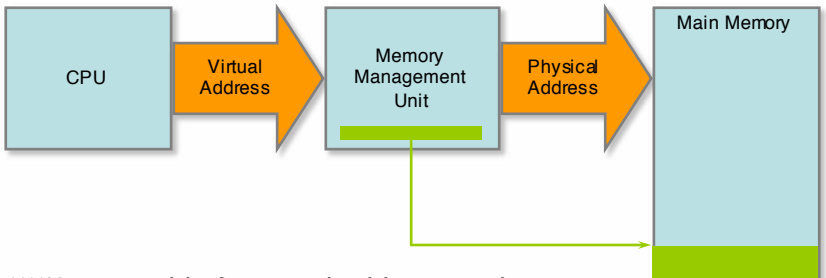
Operating Systems

Operating Systems

Memory
Pages

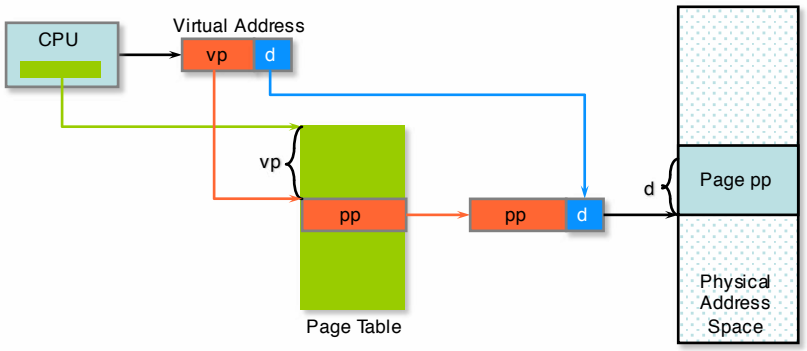
System Software Group
University of Trier
D-54286 Trier, Germany
Contact: Peter Sturm
sturm@uni-trier.de

MMU



- MMU responsible for virtual address translation
- Mapping table also stored in main memory
- Size of table must be minimized
  - Each table entry is valid for a contiguous set of addresses called page

Page-based Virtual Addressing



- Single-step mapping
- Typical page sizes 2, 4, or 8 KB

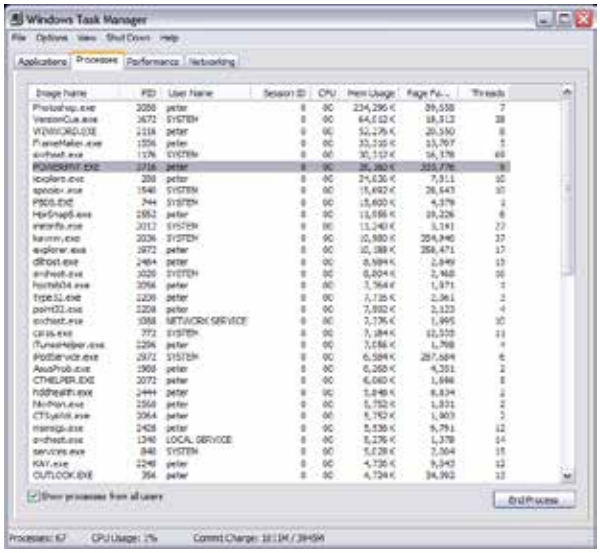
Table Size

- Mapping 32 bit virtual address to 32 bit physical address
  - Page size = 2 kbyte (2^{11} bytes)
    - 2^{32} / 2^{11} = 2^{21} entries, 4 byte per entry
      ⇒ 2^{23} byte table size = 8 mbyte
  - Page size = 4 kbyte (2^{12} bytes)
    - 2^{32} / 2^{12} = 2^{20} entries, 4 byte per entry
      ⇒ 2^{22} byte table size = 4 mbyte
  - Page size = 8 kbyte (2^{13} bytes)
    - 2^{32} / 2^{13} = 2^{19} entries, 4 byte per entry
      ⇒ 2^{21} byte table size = 2 mbyte
- Again, it is a sparse table since most apps don't require so much address space, but
  - Complete table must be allocated to guarantee constant time for address translation

Memory Sizes of Applications

- Determine the memory consumption of common applications
- Hint
  - Start some applications and check the Windows task manager

Memory consumption for Selected Applications

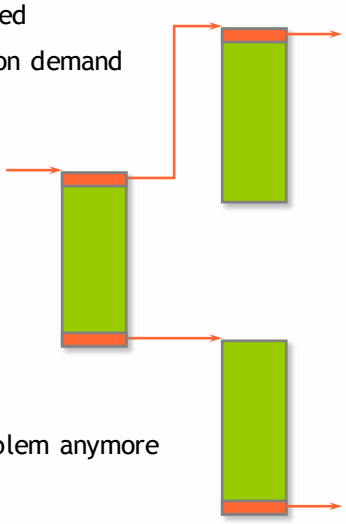


Another Reduction in Size

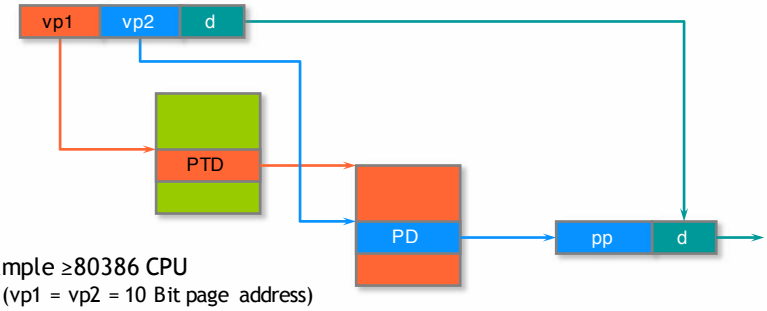
- Most applications require only a fraction of the possible address space
- With the exception of Photoshop – and of course Photoshop is different ☺ - most applications require less than 64 mbyte
  - 64 mbyte is only 1/64 of 4 gbyte
- Observation
  - Several contiguous memory areas are used by most apps
    - Code
    - Static data
    - Dynamic data = heap
    - Stack
- Possible improvement
  - Introduction of a multi-stage, tree-like mapping table

Gain

- Root page table MUST be allocated
- Page tables of later stages only on demand
  - Example: First and last byte in address space used:
    - vp1=vp2=10
    - $3 * 2^{10} * 4$ Byte = 12 Kbyte
  - Above config also works with
    - 1024 * 4 KB = 4 MB for text, data, and heap
    - 1024 * 4 KB = 4 MB stack
- 12 kbyte are sufficient for many small executables
- Conclusion: Size of table no problem anymore



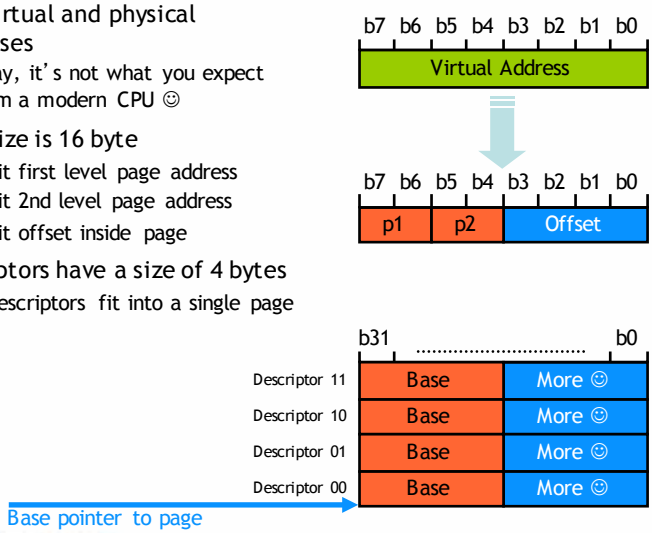
Multiple Stages



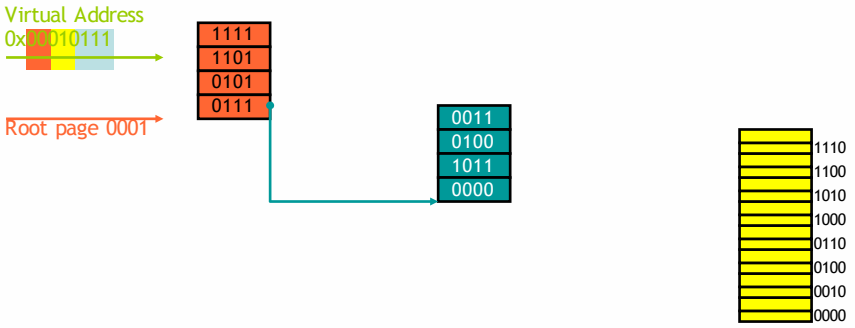
- Example ≥80386 CPU
  - (vp1 = vp2 = 10 Bit page address)
- PTD = Page table descriptor
  - Points to the next lower page table
- PD = Page descriptor
  - Points to the physical location of virtual page

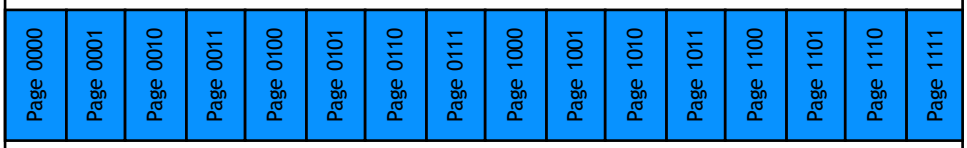
Example

- 8 bit virtual and physical addresses
  - Okay, it's not what you expect from a modern CPU ☺
- Page size is 16 byte
  - 2 bit first level page address
  - 2 bit 2nd level page address
  - 4 bit offset inside page
- Descriptors have a size of 4 bytes
  - 4 descriptors fit into a single page



Example (contd.)





Operating Systems

Memory - Pages 13

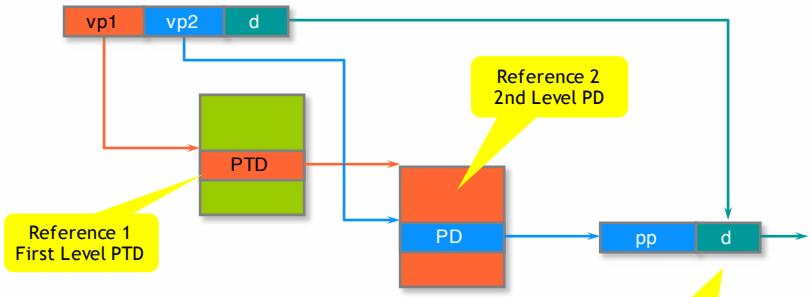
Loss

- Each stage requires an additional memory access for table lookup
  - 1 access when addressing phyiscal memory
  - 2 accesses with single-stage page table
  - 3 accesses with dual-stage page table
  - …
  - n+1 accesses with n-stage page table
- Slow-down not accetable
  - Memory is already too slow compared to CPU speed

Operating Systems

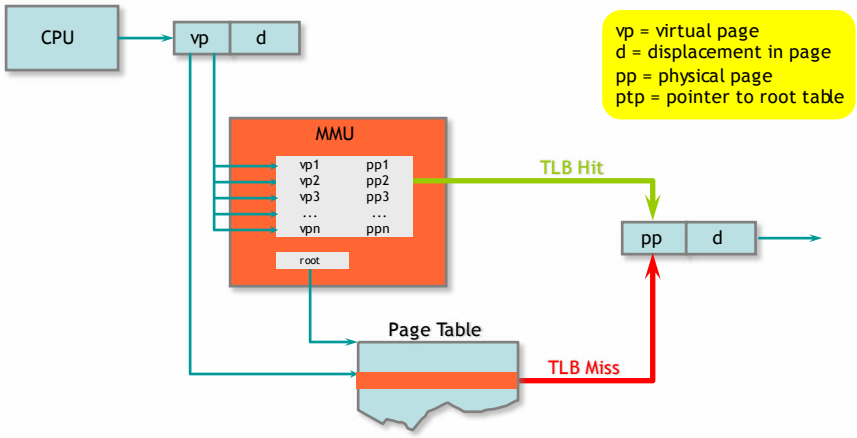
Memory - Pages 14

Performance Issues

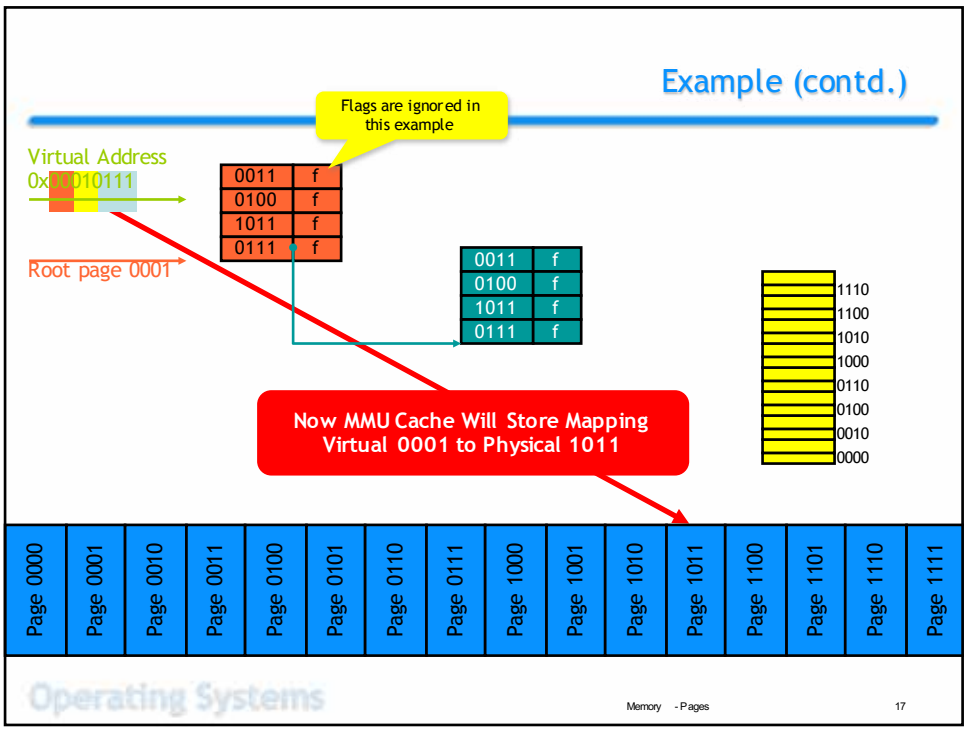


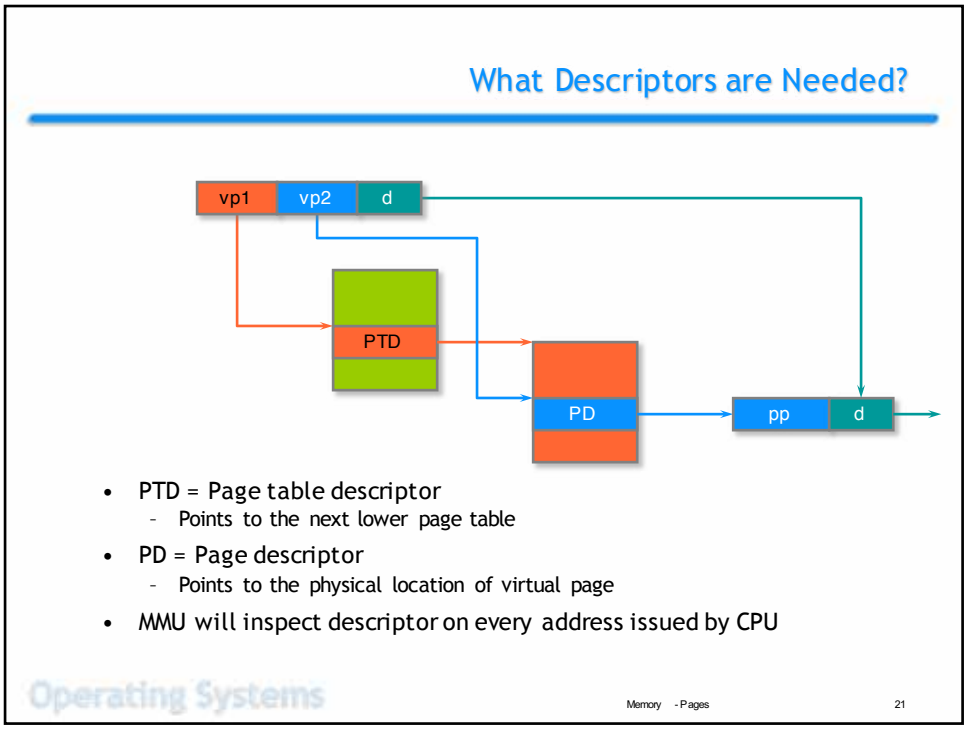
- Multiple memory accesses for a single address translation
  - 1 access for each stage
- Most main memory already too slow
- Idea: Result of address translation should be cached
  - Subsequent translations can be solved without delay (just lookup)
  - Large page + reference locality = good hit rate

TLB (Translation Lookaside Buffer)



- TLB = Associative search for virtual address vp
- Reference locality of programs yields very high hit rate (≥95%)


Example (contd.)
Flags are ignored in this example
Virtual Address
0x0010111
Root page 0001
0011 f
0100 f
1011 f
0111 f
0011 f
0100 f
1011 f
0111 f
1110
1100
1010
1000
0110
0100
0010
0000
Now MMU Cache Will Store Mapping
Virtual 0001 to Physical 1011
Page 0000
Page 0001
Page 0010
Page 0011
Page 0100
Page 0101
Page 0110
Page 0111
Page 1000
Page 1001
Page 1010
Page 1011
Page 1100
Page 1101
Page 1110
Page 1111
Operating Systems
Memory - Pages
17

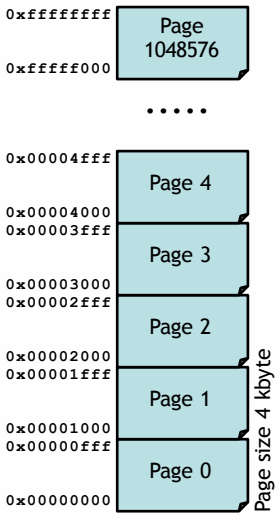

What Descriptors are Needed?
vp1
vp2
d
PTD
PD
pp
d
• PTD = Page table descriptor
– Points to the next lower page table
• PD = Page descriptor
– Points to the physical location of virtual page
• MMU will inspect descriptor on every address issued by CPU
Operating Systems
Memory - Pages
21

What Must/Can be Stored in Descriptor?

- Must be stored
  - Page location
    - Base address of virtual page in physical memory
  - Type flag to distinguish page descriptors from page table descriptors (could also be definied implicitly by translation stage)
- Can be stored
  - Access permissions
    - Read, Write, Execute for user-mode access
    - Read, Write, Execute for privileged access
  - Statistics
    - Has some address inside page been referenced?
    - Has some address inside page been modified?
  - Page location again
    - Page is currently not in main memory but on disk

Size of Descriptor

- Address of page in physical memory is 32 bit, but
  - by limiting the position of a page to multiples of page size k, the lower k bits are always 0 and can be ignored
- Assuming a page size of 4 kbyte, 12 bit are free in each descriptor
  - Space for permissions, statistics, etc.
  - Additional bits can be used to implemented advanced memory techniques
- 32 bit descriptor size sufficient with all extras ☺



Page Table Descriptor

- T-Bit required to identify descriptor type:
  - Page table descriptor (PTD)
  - Page descriptor (PD)
- P-Bit:
  - Present bit:
    - P=1: Page accessible in main memory
      - Base address points to main memory
    - P=0: Page stored on external memory (e.g. disk)
      - Base address identifies external memory block
- Some bits in descriptor free for OS usage

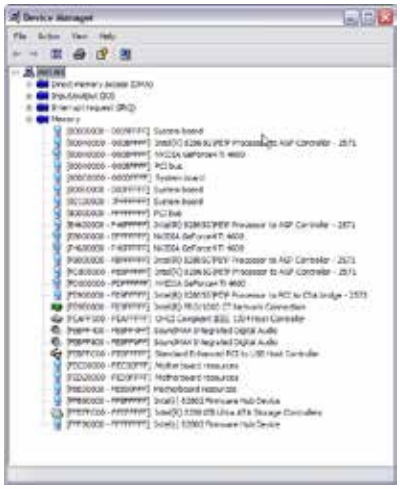
Operating Systems Memory - Pages 24

Page Descriptor

- Access control bits
  - Define the allowed access modes (read/write/exec, user/supervisor)
  - Denying any access sometimes useful
  - Access violation leads to interrupt and OS intervention
- C-Bit (Cache Disable Bit)
- R-Bit (Referenced Bit)
  - R=1 ⇔ Page has been referenced since last reset of bit
  - Must be reset explicitly by OS
- D-Bit (Dirty Bit)
  - D=1 ⇔ Page has been written since last reset of bit (⇒ R=1)
  - Must be reset explicitly by OS
- Again a few bits freely available to OS

Operating Systems Memory - Pages 25

Cache Disable Bit



- Some virtual pages map to physical pages with memory-mapped IO
  - Make PCI memory areas available to privileged services
  - Access to command and status registers for some peripherals
- Changes to these memory positions happen as side-effects
- Caches shouldn't store the content they've seen the last time
  - Otherwise caches would signal a hit after the first access (hit 'a' once on the keyboard, means 'a' forever ☺)

Conclusions

- Virtual memory can be implemented efficiently
  - Space efficiency
    - tree of page tables
  - Time efficiency
    - TLB cache in MMU
- The advantages outweight the space and time requirements by far
  - Homogeneous address space
  - Internal and external protection
- Many technical details still to be solved
  - Modification in some page table requires partial or sometimes complete flush of TLB

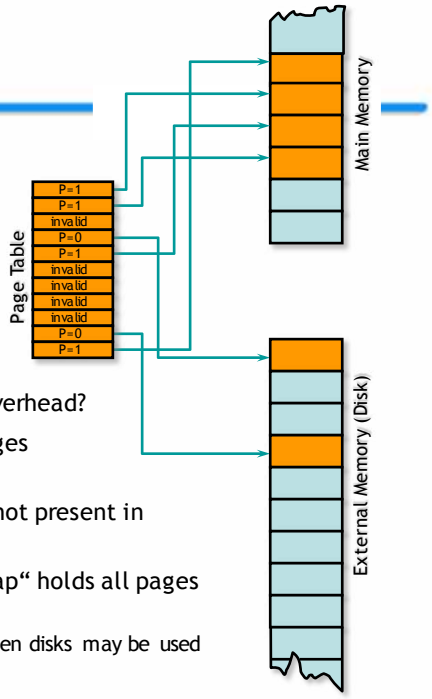
Operating Systems

Memory
Paging: Pages are present or not

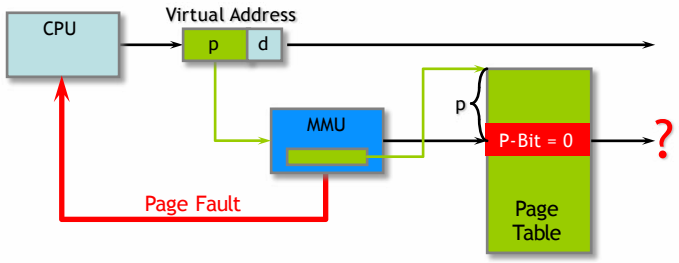
System Software Group
University of Trier
D-54286 Trier, Germany
Contact: Peter Sturm
sturm@uni-trier.de

Paging

- Advantages
  - Increasing virtual address space
  - Virtual address space limited by capacity of disk only
  - Enough space to support multiple address spaces in parallel
- Reference locality guarantees low overhead?
- Reading and writing of individual pages
  - Present flag in MMU descriptor
- Page fault in case of access to page not present in memory
- Special file called „pagefile“ or „swap“ holds all pages not in main memory
  - In some OS dedicated partitions or even disks may be used



Page Fault



- Page stored on external memory
  - Present flag = 0
  - Address in descriptor points to position on external memory
  - External page should be store in memory and instruction should be repeated
- Memory area not allocated
  - Program error, terminate the application

Operating Systems

OS Reaction on Page Fault

- Address not allowed (invalid reference)?
  - Terminate application resp. thread
- Page is stored on external memory
  - Searching for a free page in main memory
    - High probability that no free page available
    - Discard or write back another page (page replacement algorithm)
  - Schedule page transfer from disk
  - *Context switch*
  - Update page table
    - Set present flag in descriptor
    - Assign correct base address of page in descriptor
  - Repeat instruction
    - Support by CPU required

Operating Systems

Efficiency of Page Replacement

- Effective access time depends on the probability of a page fault

$$t_{effective} = (1-p) * t_{Memory} + p * t_{PageFault}$$

  - p = Page fault probability
  - t_{Memory} = Access time main memory (few ns)
  - $t_{PageFault}$ = Time to handle the page fault

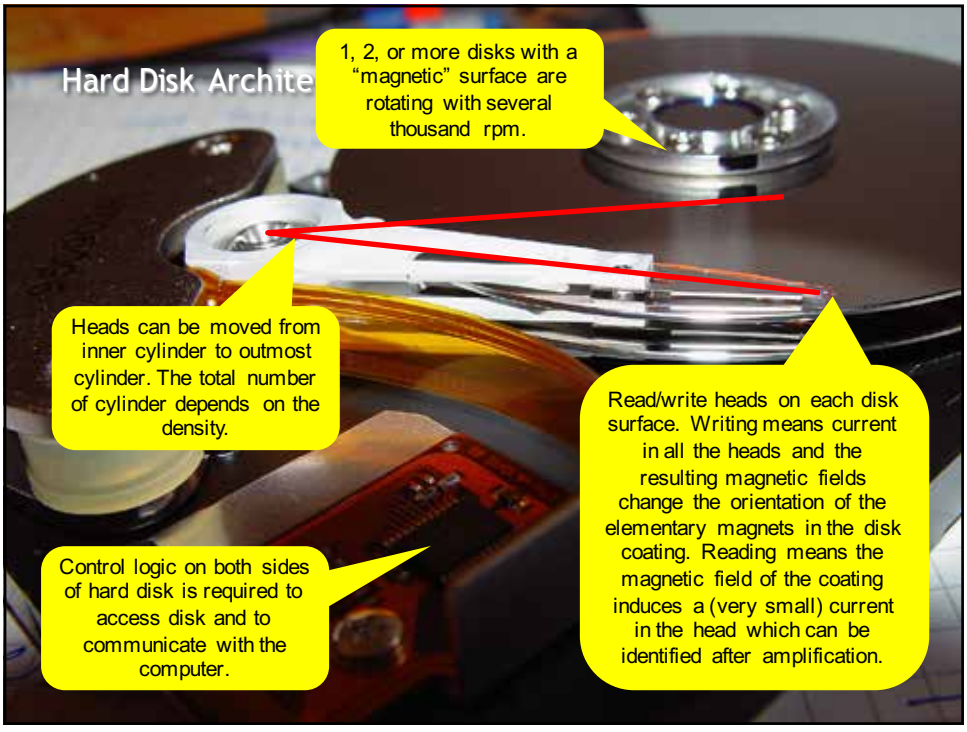
- $t_{PageFault}$

Interrupt latency and handling | Search for a free page in main memory | Write back candidate page if dirty | Read required page from disk | Repeat instruction

$$t_{Page\ Fault} = t_{Interrupt} + t_{Search} + t_{Write} + t_{Read} + t_{Instr}$$

$$\approx 2 * t_{Read/Write}$$

Operating Systems



Example

- Assumption
  - Memory access time: 100 nsec
  - Disk access time: 10 msec

$$t_{effective} = (1-p)*10^{-7} + 2p*10^{-2}$$
$$\approx 2p*10^{-2}$$

- Required: Less than 10% overhead

$$1.1*10^{-7} > 10^{-7} + 2p*10^{-2}$$
$$p < 5*10^{-7}$$

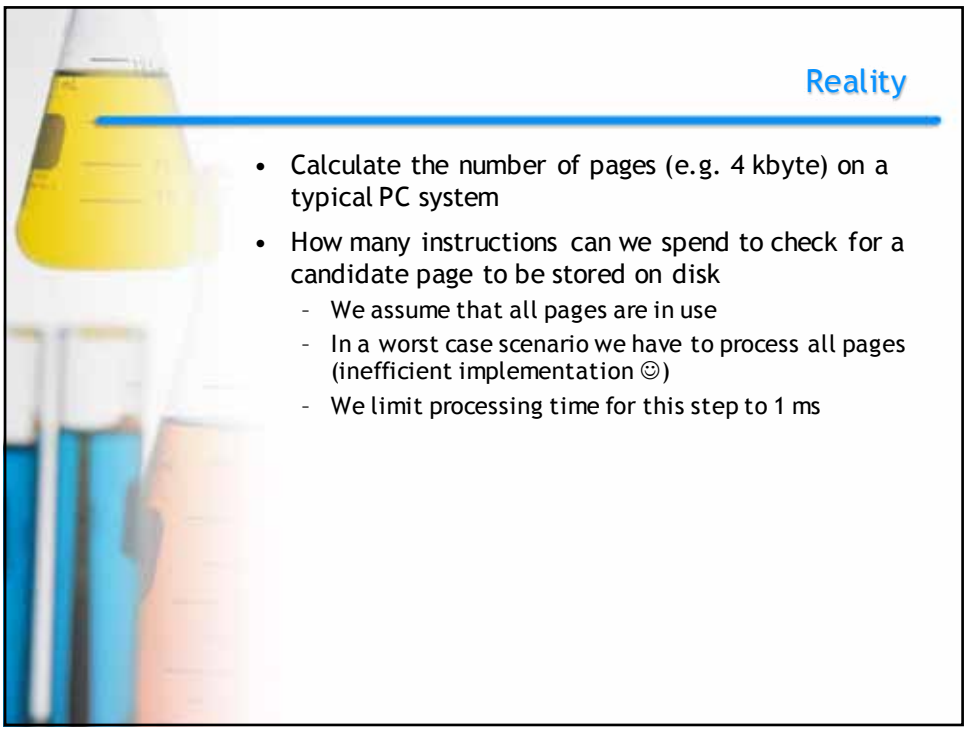
  - At most 1 page fault for each 2 million instructions

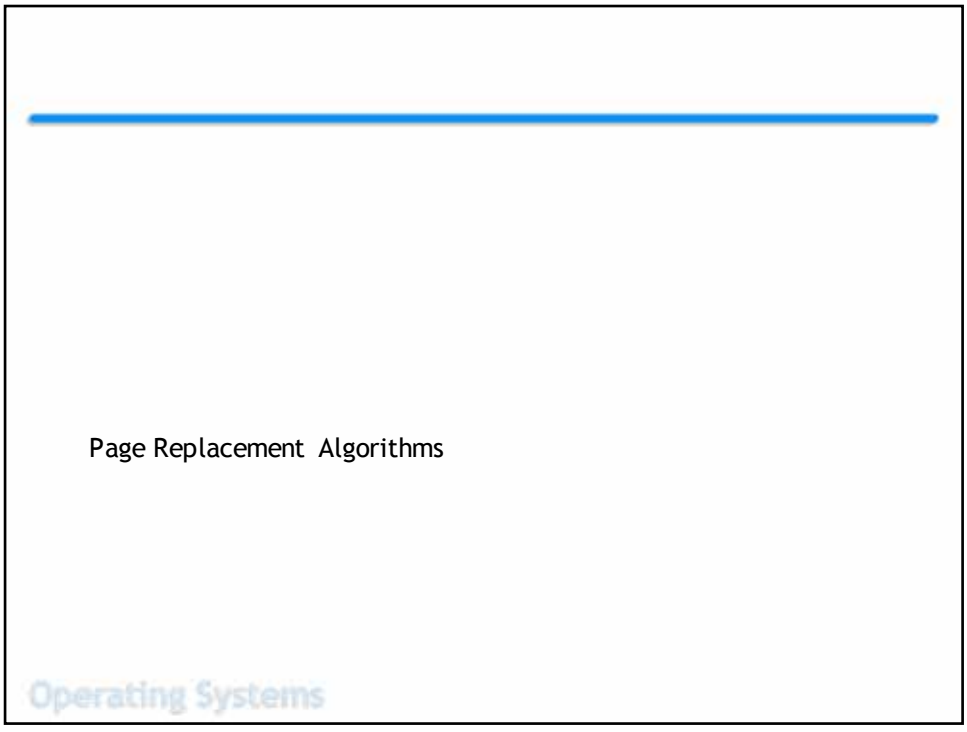
Operating Systems

Time of Page Fault

- Page fault handling dominated by at most 2 accesses to disk
  - Access time for disks in the order of 3-5 ms
- How can we improve the effective access time
  - Keep page faults at minimum
    - Always have the required pages in main memory ☺
  - Not very much possible with respect to interrupt handling
    - Faster CPU, better OS, but the gain very limited
  - Find the candidate page to leave memory very fast
    - Crucial since we have a lot of pages
  - Read and write page
    - Buy bigger and/or faster disk
      (bigger also means faster in most cases, since data are stored more densely)
  - No substantial improvement possible when repeating instructions

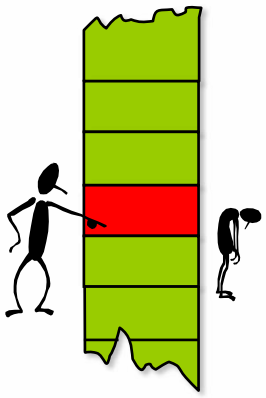
Operating Systems


Reality
• Calculate the number of pages (e.g. 4 kbyte) on a typical PC system
• How many instructions can we spend to check for a candidate page to be stored on disk
– We assume that all pages are in use
– In a worst case scenario we have to process all pages (inefficient implementation ☺)
– We limit processing time for this step to 1 ms


Page Replacement Algorithms
Operating Systems

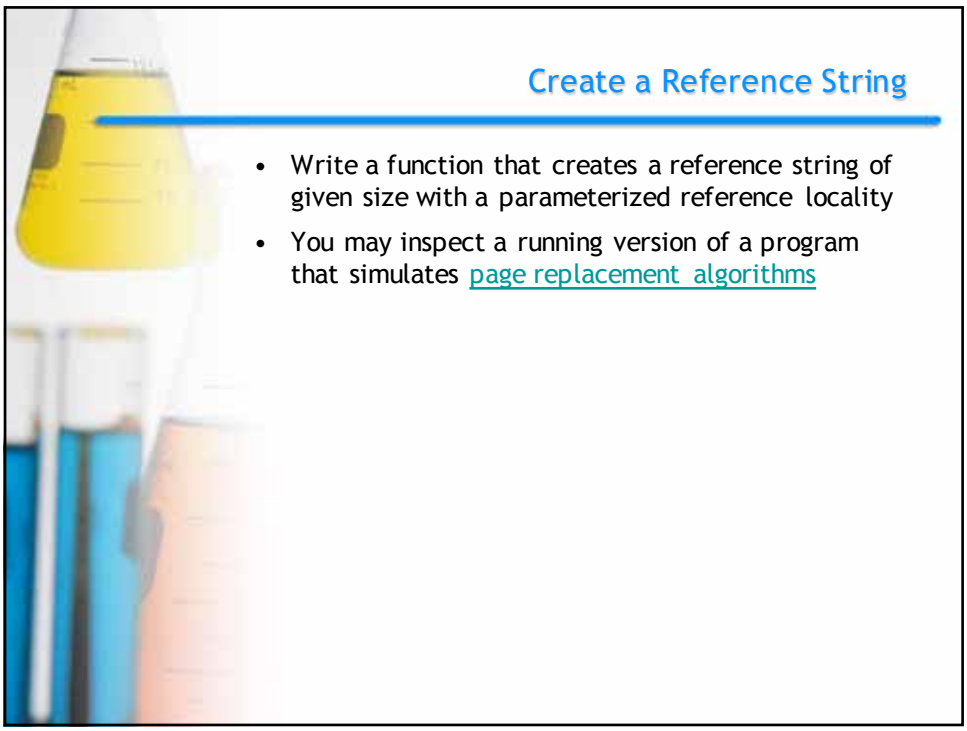
Page Replacement Algorithm

- Assumption
  No free page available when page fault occurs
- Possible consequences
  - Terminate application
  - Swap another application completely
  - Replace a memory page used otherwise
- But which page to replace?
  - Lots of pages available
    - 512 MB main memory, page size 4 kb
      ⇒ 131072 pages
  - Goal is to minimize the number of future page faults



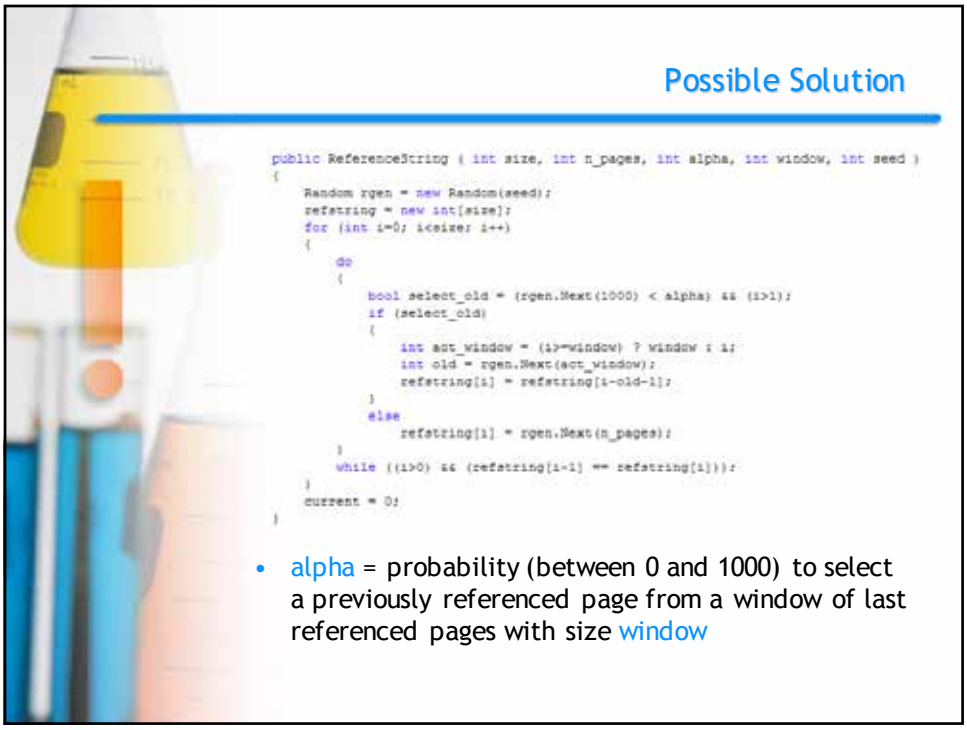
Reference Strings

- Used to evaluate resp. benchmark replacement algorithms
- Calculated from the sequence of memory references executed by the CPU
  - Hard to obtain! Why?
  - Only page information of address required
  - Only first reference to page represented
    - Subsequent references inside same page can be omitted
- Applying reference string to a replacement algorithm with a fixed set of memory pages
  - Count the number of page faults



Create a Reference String

- Write a function that creates a reference string of given size with a parameterized reference locality
- You may inspect a running version of a program that simulates page replacement algorithms



Possible Solution

```
public ReferenceString ( int size, int n_pages, int alpha, int window, int seed )
{
    Random rgen = new Random(seed);
    refstring = new int[size];
    for (int i=0; i<size; i++)
    {
        do
        {
            bool select_old = (rgen.Next(1000) < alpha) && (i>1);
            if (select_old)
            {
                int act_window = (i>=window) ? window : i;
                int old = rgen.Next(act_window);
                refstring[i] = refstring[i-old-1];
            }
            else
                refstring[i] = rgen.Next(n_pages);
        }
        while ((i>0) && (refstring[i-1] == refstring[i]));
    }
    current = 0;
}
```

- alpha = probability (between 0 and 1000) to select a previously referenced page from a window of last referenced pages with size window

Optimal Replacement by Belady

- Replace the page which will not be used for the longest time in future
  - Quite hard to implement
  - Used to compare with other strategies
- Why is Belady optimal?

Operating Systems

Implement Belady

- Extend your program creating a reference string with an implementation of Belady
  - By using the reference string, we know the future
- Inspect the complete program

Belady Example

```
Algorithm Belady
  0 ref=    1 |    -1   -1   -1   -1 |  page fault, replace empty page 0
  1 ref=    5 |     1   -1   -1   -1 |  page fault, replace empty page 1
  2 ref=    1 |     1    5   -1   -1 |  page found at 0
  3 ref=    5 |     1    5   -1   -1 |  page found at 1
  4 ref=    1 |     1    5   -1   -1 |  page found at 0
  5 ref=    5 |     1    5   -1   -1 |  page found at 1
  6 ref=    0 |     1    5   -1   -1 |  page fault, replace empty page 2
  7 ref=    1 |     1    5    0   -1 |  page found at 0
  8 ref=    3 |     1    5    0   -1 |  page fault, replace empty page 3
  9 ref=    5 |     1    5    0    3 |  page found at 1
 10 ref=    1 |     1    5    0    3 |  page found at 0
 11 ref=    5 |     1    5    0    3 |  page found at 1
 12 ref=    6 |     1    5    0    3 |  page fault, replace used page 3
 13 ref=    5 |     1    5    0    6 |  page found at 1
 14 ref=    1 |     1    5    0    6 |  page found at 0
 15 ref=    6 |     1    5    0    6 |  page found at 3
 16 ref=    5 |     1    5    0    6 |  page found at 1
 17 ref=    0 |     1    5    0    6 |  page found at 2
 18 ref=    9 |     1    5    0    6 |  page fault, replace used page 3
 19 ref=    0 |     1    5    0    9 |  page found at 2
 20 ref=    5 |     1    5    0    9 |  page found at 1
 21 ref=    9 |     1    5    0    9 |  page found at 3
 22 ref=    0 |     1    5    0    9 |  page found at 2
 23 ref=    9 |     1    5    0    9 |  page found at 3
 24 ref=    3 |     1    5    0    9 |  page fault, replace used page 1
 25 ref=    1 |     1    3    0    9 |  page found at 0
 26 ref=    9 |     1    3    0    9 |  page found at 3
 27 ref=    3 |     1    3    0    9 |  page found at 1
 28 ref=    9 |     1    3    0    9 |  page found at 3
 29 ref=    6 |     1    3    0    9 |  page fault, replace used page 1
 30 ref=    2 |     1    6    0    9 |  page fault, replace used page 1
 31 ref=    9 |     1    2    0    9 |  page found at 3
 32 ref=    0 |                     |  found at 2
 33 ref=    9 |                     |  found at 3
 34 ref=    0 |                     |
 35 ref=    5 |                     |
 36 ref=    9 |                     |
 37 ref=    0 |                     |
 38 ref=    0 |     1    2    0    9 |  page found at 2
 39 ref=    2 |     1    2    0    9 |  page found at 2
 40 ref=    8 |     1    2    0    9 |  page fault, repla
 41 ref=    0 |     1    2    0    8 |  page found at 2
 42 ref=    6 |     1    2    0    8 |  page fault, replace used page 3
 43 ref=    0 |     1    2    0    6 |  page found at 2
 44 ref=    2 |     1    2    0    6 |  page found at 1
 45 ref=    7 |     1    2    0    6 |  page fault, replace used page 1
 46 ref=    0 |     1    7    0    6 |  page found at 2
 47 ref=    7 |     1    7    0    6 |  page found at 1
 48 ref=    1 |     1    7    0    6 |  page found at 0
 49 ref=    7 |     1    7    0    6 |  page found at 1
Number of hits 38
Number of page faults 12
```

This and all subsequent examples: a set of 50 references are executed on main memory consisting of 4 pages in total. The virtual address space has a size of 10 pages (0..9)

Referenced page

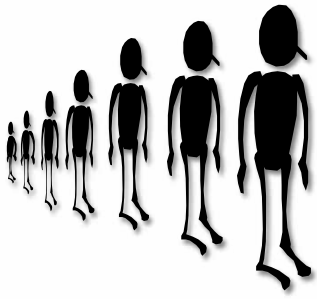
Number of reference

Virtual pages currently in main memory (-1 means free)

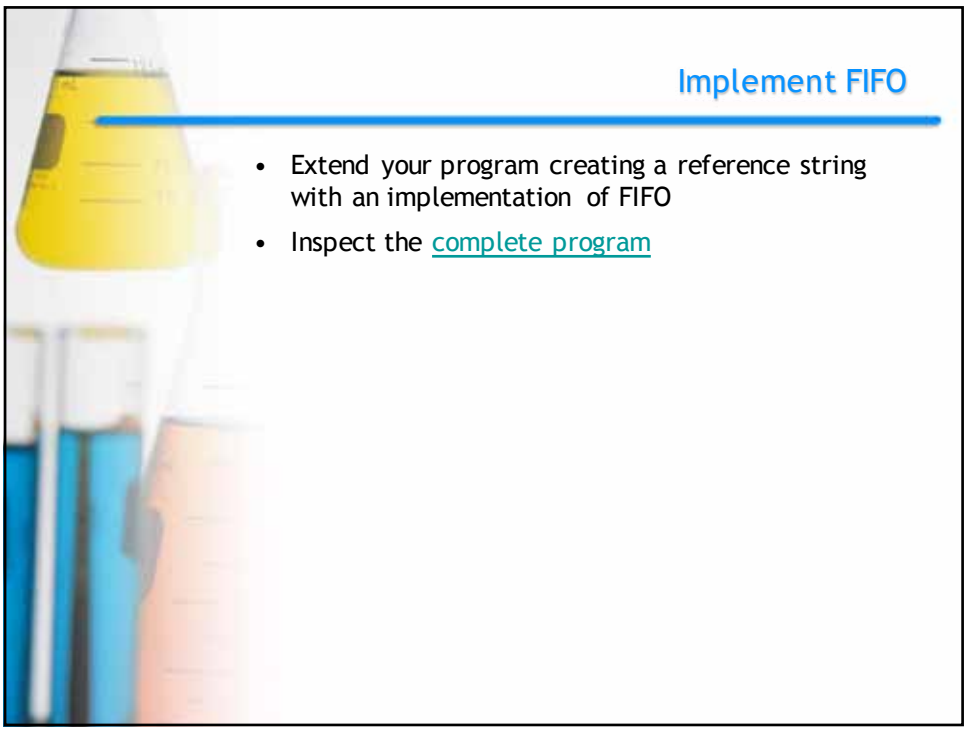
What happens?

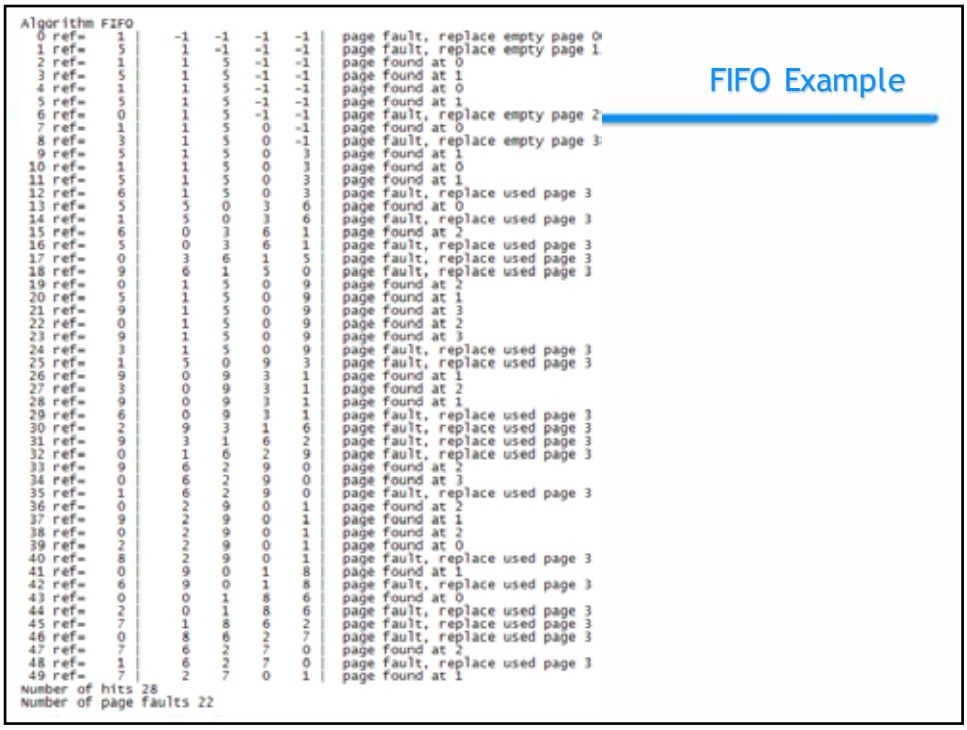
FIFO

- Replace the page which is in main memory for the longest time
- Does not take reference locality into account
  - Frequently used pages (hot spots) are paged out
  - Rarely referenced pages stay in main memory
    - Start up code of some application and others
- Easy to implement via list

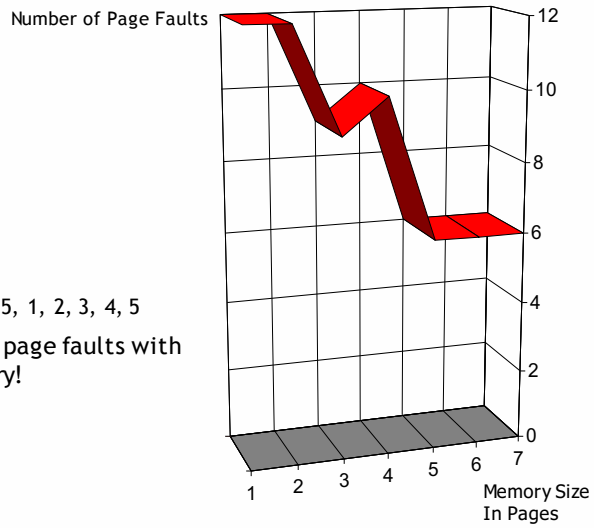


Operating Systems


Implement FIFO
• Extend your program creating a reference string with an implementation of FIFO
• Inspect the complete program


FIFO Example
Algorithm FIFO
0 ref= 1 | -1 -1 -1 -1 | page fault, replace empty page 0
1 ref= 5 | 1 -1 -1 -1 | page fault, replace empty page 1
2 ref= 1 | 1 5 -1 -1 | page found at 0
3 ref= 5 | 1 5 -1 -1 | page found at 1
4 ref= 1 | 1 5 -1 -1 | page found at 0
5 ref= 5 | 1 5 -1 -1 | page found at 1
6 ref= 0 | 1 5 -1 -1 | page fault, replace empty page 2
7 ref= 1 | 1 5 0 -1 | page found at 0
8 ref= 3 | 1 5 0 -1 | page fault, replace empty page 3
9 ref= 5 | 1 5 0 3 | page found at 1
10 ref= 1 | 1 5 0 3 | page found at 0
11 ref= 5 | 1 5 0 3 | page found at 1
12 ref= 6 | 1 5 0 3 | page fault, replace used page 3
13 ref= 5 | 5 0 3 6 | page found at 0
14 ref= 1 | 5 0 3 6 | page fault, replace used page 3
15 ref= 6 | 0 3 6 1 | page found at 2
16 ref= 5 | 0 3 6 1 | page fault, replace used page 3
17 ref= 0 | 3 6 1 5 | page fault, replace used page 3
18 ref= 9 | 6 1 5 0 | page fault, replace used page 3
19 ref= 0 | 1 5 0 9 | page found at 2
20 ref= 5 | 1 5 0 9 | page found at 1
21 ref= 9 | 1 5 0 9 | page found at 3
22 ref= 0 | 1 5 0 9 | page found at 2
23 ref= 9 | 1 5 0 9 | page found at 3
24 ref= 3 | 1 5 0 9 | page fault, replace used page 3
25 ref= 1 | 5 0 9 3 | page fault, replace used page 3
26 ref= 9 | 0 9 3 1 | page found at 1
27 ref= 3 | 0 9 3 1 | page found at 2
28 ref= 9 | 0 9 3 1 | page found at 1
29 ref= 6 | 0 9 3 1 | page fault, replace used page 3
30 ref= 2 | 9 3 1 6 | page fault, replace used page 3
31 ref= 9 | 3 1 6 2 | page fault, replace used page 3
32 ref= 0 | 1 6 2 9 | page fault, replace used page 3
33 ref= 9 | 6 2 9 0 | page found at 2
34 ref= 0 | 6 2 9 0 | page found at 3
35 ref= 1 | 6 2 9 0 | page fault, replace used page 3
36 ref= 0 | 2 9 0 1 | page found at 2
37 ref= 9 | 2 9 0 1 | page found at 1
38 ref= 0 | 2 9 0 1 | page found at 2
39 ref= 2 | 2 9 0 1 | page found at 0
40 ref= 8 | 2 9 0 1 | page fault, replace used page 3
41 ref= 0 | 9 0 1 8 | page found at 1
42 ref= 6 | 9 0 1 8 | page fault, replace used page 3
43 ref= 0 | 0 1 8 6 | page found at 0
44 ref= 2 | 0 1 8 6 | page fault, replace used page 3
45 ref= 7 | 1 8 6 2 | page fault, replace used page 3
46 ref= 0 | 8 6 2 7 | page fault, replace used page 3
47 ref= 7 | 6 2 7 0 | page found at 2
48 ref= 1 | 6 2 7 0 | page fault, replace used page 3
49 ref= 7 | 2 7 0 1 | page found at 1
Number of hits 28
Number of page faults 22

Belady's Anomaly with FIFO



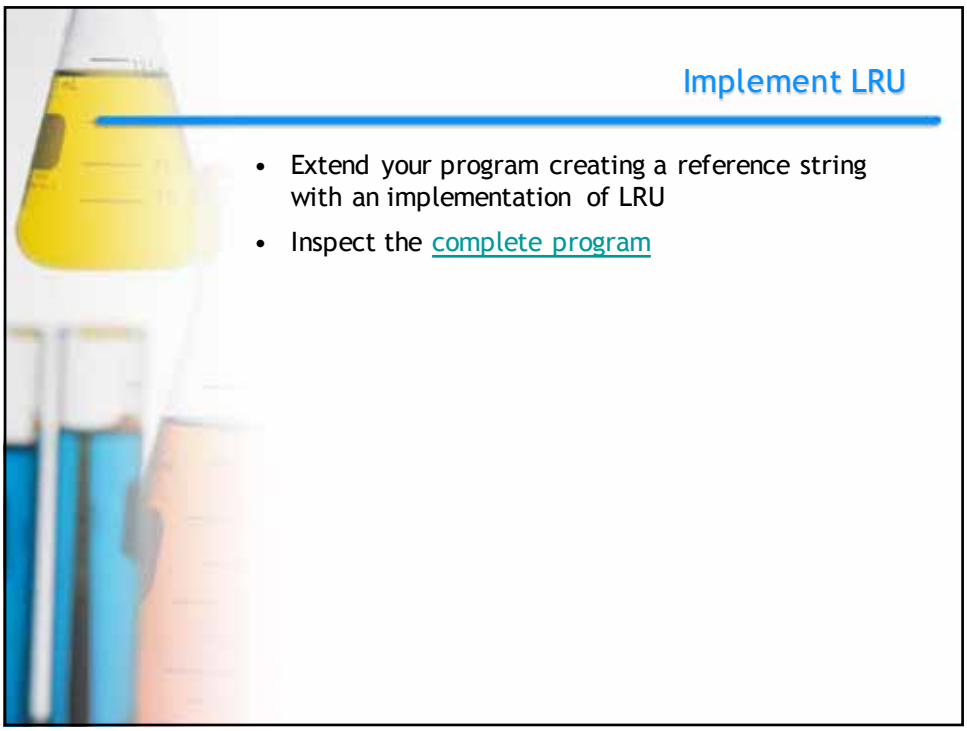
- Reference string
  - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Higher number of page faults with more main memory!

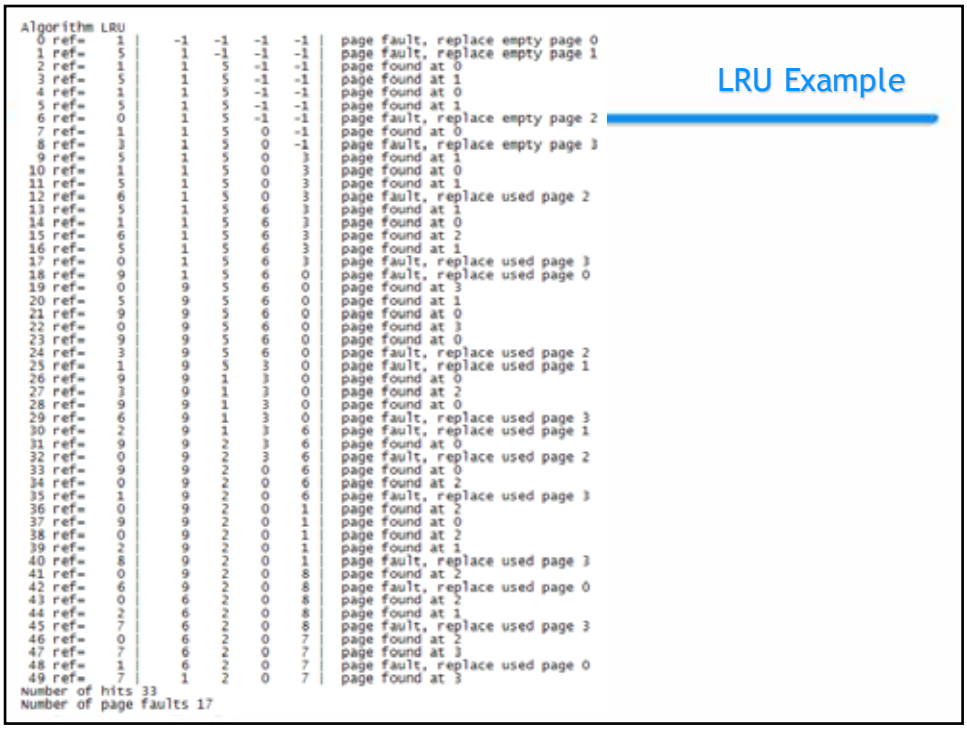
Operating Systems

Least Recently Used (LRU)

- Replace the page which has not been referenced for the longest time in the past
  - Again reference locality
    - What you don´t reference in the past for a long time you won´t reference in the near future
    - Same with your CDs, cloths, etc.
  - Project the past onto the future
- LRU is yet the best approximation to Belady
- Hard to implement

Operating Systems


Implement LRU
• Extend your program creating a reference string with an implementation of LRU
• Inspect the complete program

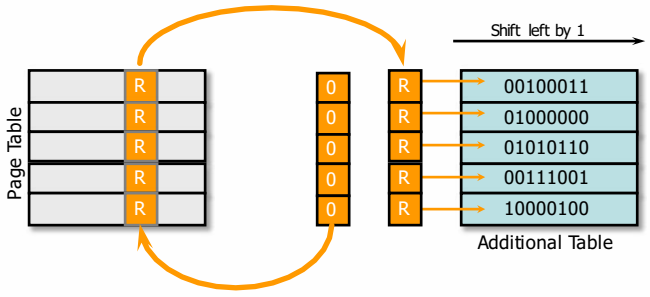

LRU Example
Algorithm LRU
0 ref= 1 | -1 -1 -1 -1 | page fault, replace empty page 0
1 ref= 5 | 1 -1 -1 -1 | page fault, replace empty page 1
2 ref= 1 | 1 5 -1 -1 | page found at 0
3 ref= 5 | 1 5 -1 -1 | page found at 1
4 ref= 1 | 1 5 -1 -1 | page found at 0
5 ref= 5 | 1 5 -1 -1 | page found at 1
6 ref= 0 | 1 5 -1 -1 | page fault, replace empty page 2
7 ref= 1 | 1 5 0 -1 | page found at 0
8 ref= 3 | 1 5 0 -1 | page fault, replace empty page 3
9 ref= 5 | 1 5 0 3 | page found at 1
10 ref= 1 | 1 5 0 3 | page found at 0
11 ref= 5 | 1 5 0 3 | page found at 1
12 ref= 6 | 1 5 0 3 | page fault, replace used page 2
13 ref= 5 | 1 5 6 3 | page found at 1
14 ref= 1 | 1 5 6 3 | page found at 0
15 ref= 6 | 1 5 6 3 | page found at 2
16 ref= 5 | 1 5 6 3 | page found at 1
17 ref= 0 | 1 5 6 3 | page fault, replace used page 3
18 ref= 9 | 1 5 6 0 | page fault, replace used page 0
19 ref= 0 | 9 5 6 0 | page found at 3
20 ref= 5 | 9 5 6 0 | page found at 1
21 ref= 9 | 9 5 6 0 | page found at 0
22 ref= 0 | 9 5 6 0 | page found at 3
23 ref= 9 | 9 5 6 0 | page found at 0
24 ref= 3 | 9 5 6 0 | page fault, replace used page 2
25 ref= 1 | 9 5 3 0 | page fault, replace used page 1
26 ref= 9 | 9 1 3 0 | page found at 0
27 ref= 3 | 9 1 3 0 | page found at 2
28 ref= 9 | 9 1 3 0 | page found at 0
29 ref= 6 | 9 1 3 0 | page fault, replace used page 3
30 ref= 2 | 9 1 3 6 | page fault, replace used page 1
31 ref= 9 | 9 2 3 6 | page found at 0
32 ref= 0 | 9 2 3 6 | page fault, replace used page 2
33 ref= 9 | 9 2 0 6 | page found at 0
34 ref= 0 | 9 2 0 6 | page found at 2
35 ref= 1 | 9 2 0 6 | page fault, replace used page 3
36 ref= 0 | 9 2 0 1 | page found at 2
37 ref= 9 | 9 2 0 1 | page found at 0
38 ref= 0 | 9 2 0 1 | page found at 2
39 ref= 2 | 9 2 0 1 | page found at 1
40 ref= 8 | 9 2 0 1 | page fault, replace used page 3
41 ref= 0 | 9 2 0 8 | page found at 2
42 ref= 6 | 9 2 0 8 | page fault, replace used page 0
43 ref= 0 | 6 2 0 8 | page found at 2
44 ref= 2 | 6 2 0 8 | page found at 1
45 ref= 7 | 6 2 0 8 | page fault, replace used page 3
46 ref= 0 | 6 2 0 7 | page found at 2
47 ref= 7 | 6 2 0 7 | page found at 3
48 ref= 1 | 6 2 0 7 | page fault, replace used page 0
49 ref= 7 | 1 2 0 7 | page found at 3
Number of hits 33
Number of page faults 17

Implementing LRU

- LRU is hard too implement
- Problem
  - Sort all pages by last access time
  - Every reference (write and read) has to be considered
- Counter-based implementation
  - Add counter field to page descriptor
  - Add logical clock to CPU (Increment with every reference)
  - MMU updates descriptor with CPU clock for every reference
  - Page with the smallest clock value is the victim
- Stack-based implementation
  - Stack of referenced pages
  - Referenced pages are pushed on top of stack
  - Page at the stacks´ tail is victim

Operating Systems

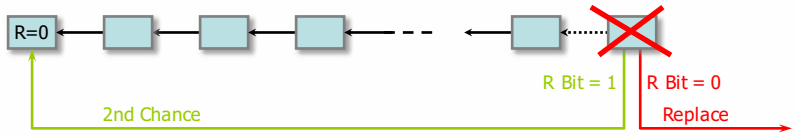
Approaching LRU (1)



- Additional table
  - Utilizing the referenced bit in each descriptor
  - Periodically (e.g. 100 msec), the referenced flag updates the MSBs in the additional table
  - Table entry reflects the last n references to each page
- Page with the smallest numerical value must leave
- Solution by A. Tanenbaum

Operating Systems

Approaching LRU (2): 2nd Chance



- Based on FIFO
- Page selected by FIFO
  - Referenced Bit = 0: Page must leave
  - Referenced Bit = 1: Second chance for page
    - Page is moved to the other end of FIFO again with
    - Referenced bit reset
- If all referenced bits set, 2nd chance degenerates into FIFO

Operating Systems

Implement 2nd Chance

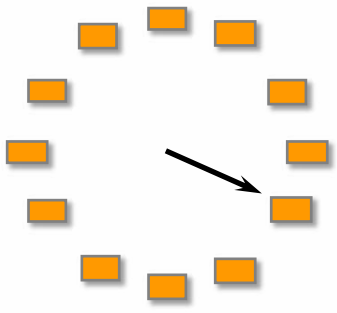
- Extend your program creating a reference string with an implementation of 2nd Chance
- Inspect the complete program

2nd Chance Example

```
Algorithm SecondChance
  0 ref=    1 |    -1-  -1-  -1-  -1- |  page fault, replace empty page 0
  1 ref=    5 |     1R  -1-  -1-  -1- |  page fault, replace empty page 1
  2 ref=    1 |     1R   5R  -1-  -1- |  page found at 0
  3 ref=    5 |     1R   5R  -1-  -1- |  page found at 1
  4 ref=    1 |     1R   5R  -1-  -1- |  page found at 0
  5 ref=    5 |     1R   5R  -1-  -1- |  page found at 1
  6 ref=    0 |     1R   5R  -1-  -1- |  page fault, replace empty page 2
  7 ref=    1 |     1R   5R   0R  -1- |  page found at 0
  8 ref=    3 |     1R   5R   0R  -1- |  page fault, replace empty page 3
  9 ref=    5 |     1R   5R   0R   3R |  page found at 1
 10 ref=    1 |     1R   5R   0R   3R |  page found at 0
 11 ref=    5 |     1R   5R   0R   3R |  page found at 1
 12 ref=    6 |     1R   5R   0R   3R |  page fault, replace used page 0
 13 ref=    5 |     6R   5-   0-   3- |  page found at 1
 14 ref=    1 |     6R   5R   0-   3- |  page fault, replace used page 0
 15 ref=    6 |     1R   3-   6-   5- |  page found at 2
 16 ref=    5 |     1R   3-   6R   5- |  page found at 3
 17 ref=    0 |     1R   3-   6R   5R |  page fault, replace used page 0
 18 ref=    9 |     0R   6R   5R   1- |  page fault, replace used page 0
 19 ref=    0 |     9R   0-   6-   5- |  page found at 1
 20 ref=    5 |     9R   0R   6-   5- |  page found at 3
 21 ref=    9 |     9R   0R   6-   5R |  page found at 0
 22 ref=    0 |     9R   0R   6-   5R |  page found at 1
 23 ref=    9 |     9R   0R   6-   5R |  page found at 0
 24 ref=    3 |     9R   0R   6-   5R |  page fault, replace used page 0
 25 ref=    1 |     3R   5R   9-   0- |  page fault, replace used page 0
 26 ref=    9 |     1R   0-   3-   5- |  page fault, replace used page 0
 27 ref=    3 |     9R   3-   5-   1- |  page found at 1
 28 ref=    9 |     9R   3R   5-   1- |  page found at 0
 29 ref=    6 |     9R   3R   5-   1- |  page fault, replace used page 0
 30 ref=    2 |     6R   1-   9-   3- |  page fault, replace used page 0
 31 ref=    9 |     2R   9-   3-   6- |  page found at 1
 32 ref=    0 |     2R   9R   3-   6- |  page fault, replace used page 0
 33 ref=    9 |     0R   6-   2-   9- |  page found at 3
 34 ref=    0 |     0R   6-   2-   9R |  page found at 0
 35 ref=    1 |     0R   6-   2-   9R |  page fault, replace used page 0
 36 ref=    0 |     1R   2-   9R   0- |  page found at 3
 37 ref=    9 |     1R   2-   9R   0R |  page found at 2
 38 ref=    0 |     1R   2-   9R   0R |  page found at 3
 39 ref=    2 |     1R   2-   9R   0R |  page found at 1
 40 ref=    8 |     1R   2R   9R   0R |  page fault, replace used page 0
 41 ref=    0 |     8R   2-   9-   0- |  page found at 3
 42 ref=    6 |     8R   2-   9-   0R |  page fault, replace used page 0
 43 ref=    0 |     6R   9-   0R   8- |  page found at 2
 44 ref=    2 |     6R   9-   0R   8- |  page fault, replace used page 0
 45 ref=    7 |     2R   0R   8-   6- |  page fault, replace used page 0
 46 ref=    0 |     7R   6-   2-   0- |  page found at 3
 47 ref=    7 |     7R   6-   2-   0R |  page found at 0
 48 ref=    1 |     7R   6-   2-   0R |  page fault, replace used page 0
 49 ref=    7 |     1R   2-   0R   7- |  page found at 3
Number of hits 30
Number of page faults 20
```

Clock Algorithm

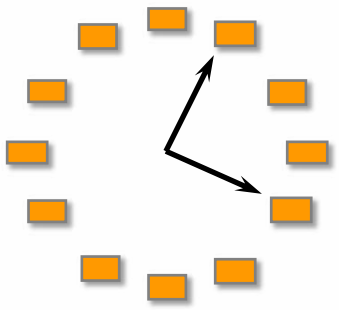
- Minor variation of 2nd chance
  - Updating the FIFO queue is too expensive
  - Using a circular list instead
- Hand of clock points to „oldest“ page
  - Referenced bit = 0
    - Replace page
  - Referenced bit = 1
    - Reset flag
    - Move hand



Operating Systems

2 Hand Version ☺

- Single hand version to clumsy for todays memory sizes
- Two Hands
  - Hand 1 resets flag
  - Hand 2 checks flag
  - Both hands „tick“ synchronously
- Distance between hands determines time period for second chance
  - Small: Only frequently used pages get a second chance
  - Maxiumum: Identical to single hand version



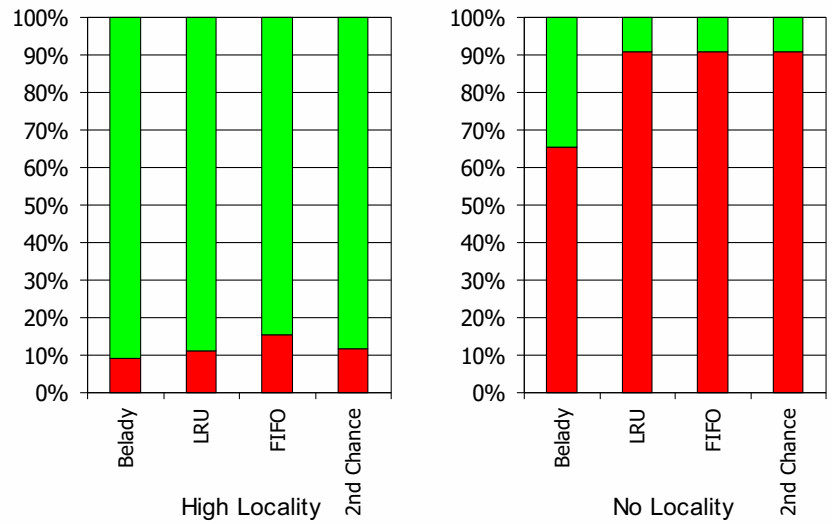
Operating Systems

Comparison Belady, LRU, FIFO, 2nd Chance

- High reference locality
  - 0.9 probability to reference one of the past 20 pages
- No reference locality
  - Next page is chosen randomly
- Configuration
  - 1000000 references
  - References within pages 0 and 100
  - Size of main memory 10 pages

Operating Systems

Comparison (contd.)



Operating Systems

Other Replacement Algorihtms

- Random
  - A randomly selected page is the victim
- Least Frequently Use (LFU)
  - Increment counter with each page reference
  - Page with the smallest value will be discarded
  - Problem with out-of-date hot spots
  - Shift counter periodically to the left
    - Exponential attenuation
- ...

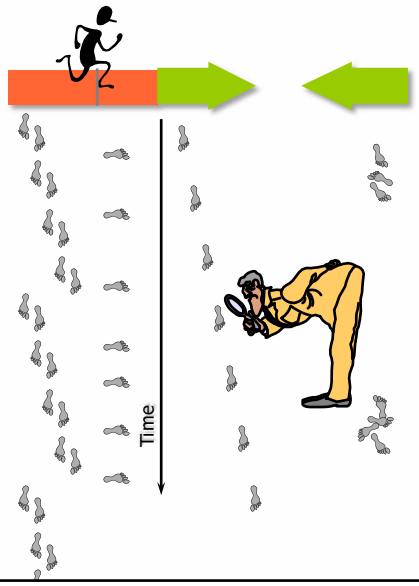
Operating Systems

Isn‘t main memory nothing but a huge cache?

Operating Systems

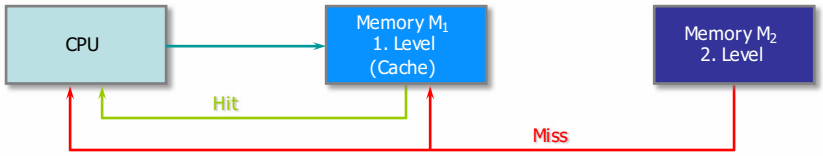
Reference Locality

- Instructions are repeated
- Variables are referenced repeatedly
- If one address is referenced, nearby addresses will be referenced with high probability
- Reasons:
  - Sequential program execution
  - Procedure nesting is modest
  - Short loops
  - Frequent processing of arrays, list, and records



Operating Systems

Caching: Utilizing Reference Locality



- 2-Level-Storage
- Characteristic parameters of M_k:
  - T_k = Average access time
  - P_k = Price per byte
  - S_k = Size of memory in bytes
- Hit rate H
  - Probability, that a referenced address is in cache

Operating Systems

Average Access Time and Costs

- Average access time:

$$T = HT_1 + (1-H)(T_1 + T_2) = T_1 + (1-H)T_2$$

- Costs:

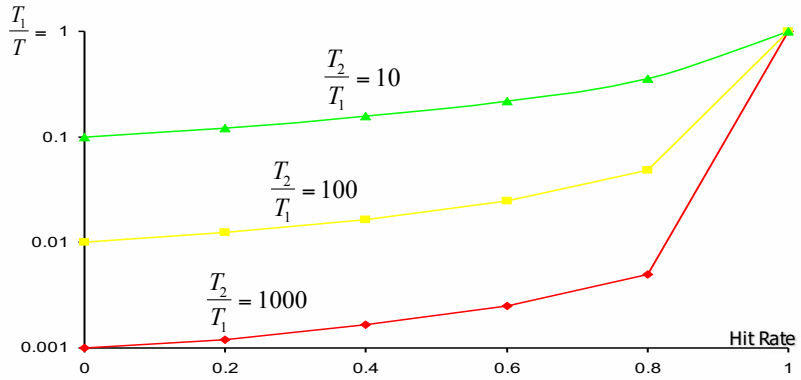
$$P = \frac{P_1 S_1 + P_2 S_2}{S_1 + S_2}$$

- Access efficiency:

$$\frac{T_1}{T} = \frac{1}{H + (1-H)\frac{T_2}{T_1}}$$

Operating Systems

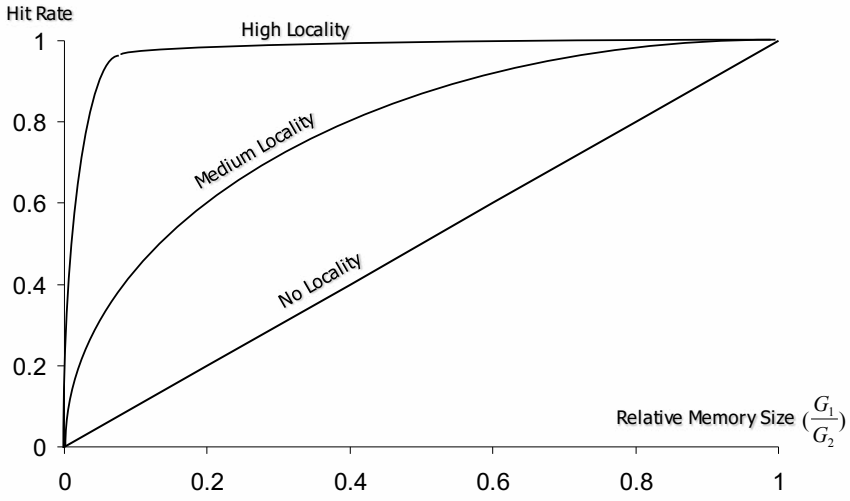
Access Efficiency



- Typical ratio for L2 cache to main memory: $5 \leq \frac{T_2}{T_1} \leq 10$

Operating Systems

Relative Cache Size



Operating Systems

Optimization Goal

- Strive for achieve

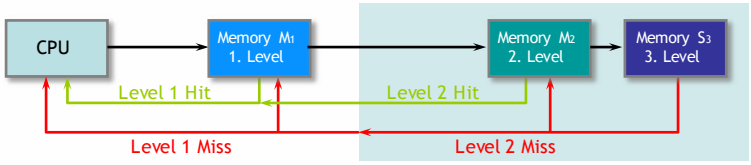
$$T_1 << T_2 : T \approx T_1$$

$$S_1 << S_2 : S \approx S_2$$

- Required hit rate H between 0.8 and 0.9
- Very small caches ($S_1 << S_2$) yield high hit rates already

Operating Systems

Multi-Level Caching



- Take advantage of memory hierarchies
  - L1 cache and MMU TLB to L2 cache
  - L2 cache to main memory (sometimes also L3 cache)
  - Main memory to external storage (disk)
- Caching between main memory and external storage promising:
  - Access time ratio: $1:10^5$
  - Size ratio: $1:10^2$-10^3
  - Cost ratio: 1:200
    - June 2005: 1 GB RAM 120 Dollar, 200 GB Disk 120 Dollar
    - November 2008: 1 GB RAM 20 Dollar, 1000 GB Disk 120 Dollar

Operating Systems

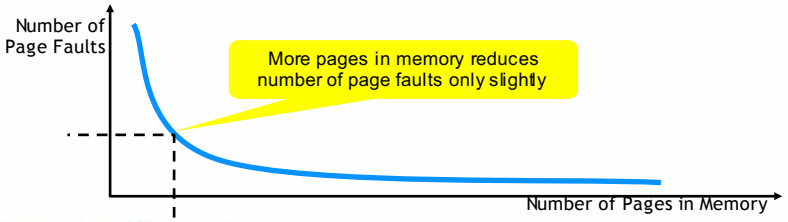
Operating Systems

Paging
Working Set Theory

System Software Group
University of Trier
D-54286 Trier, Germany
Contact: Peter Sturm
sturm@uni-trier.de

Motivation

- Most programs don't exploit the complete virtual address space
  - Memory footprint in many cases a few mbyte only
  - Of course, huge server applications such as databases and heavy web servers are different
- Due to reference locality, at a given time interval Δt they require even less pages
- Can we limit the number of pages in main memory for every given application?



Operating Systems

Locality Set and Working Set

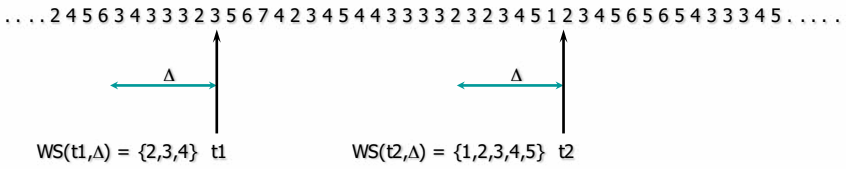
- Operating point k
  - Less than k papes in memory ⇒ Many page faults
  - More than k pages in memory ⇒ No substantial change in number of faults
  - k depends on application
- Optimal number of k pages = Locality Set
- Working set (WS) of a thread

$$WS(t,\Delta) = \text{last } \Delta \text{ referenced pages before } t$$

- Selection of Δ critical:
  - Too small: Locality set not part of working set
  - Too large: Comprises too many locality sets
    - waste of resources
  - Sensible values were around 10^4 (in 1968)

Operating Systems

Example



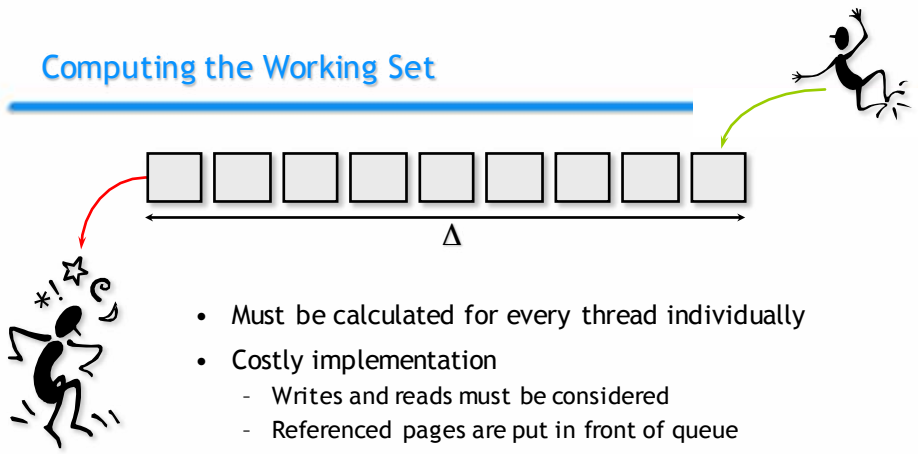
- Size of WS relevant for page assignment:

$$D = \sum_p \left| WS_p(t,\Delta) \right|$$

  - D > Available number of pages: severe memory congestion
    - Swap whole applications (address space)
    - Stop executing some applications (reduced number of faults)
  - D < Available number of pages: Resume/Start other threads

Operating Systems

Computing the Working Set



- Must be calculated for every thread individually
- Costly implementation
  - Writes and reads must be considered
  - Referenced pages are put in front of queue
  - Last page may be dropped
  - Every page in queue is part of working set
- Hard to recognize size changes in WS
- Approximations possible (just like LRU)

Operating Systems

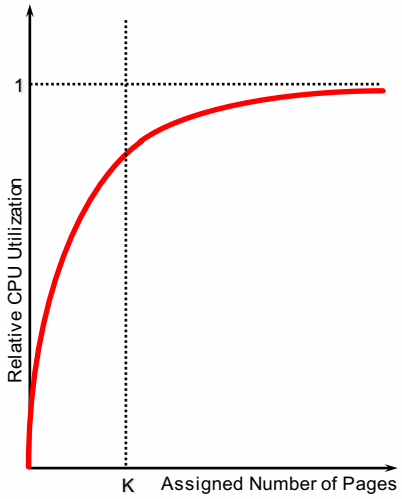
Number of Assigned Pages

- As little as possible
  - Supports more address spaces in parallel
  - Free pages are available just in case
  - Disadvantage: Page faults, but free main memory
- As much as possible
  - Very low page fault rate
  - Applications started first are preferred
  - Disadvantage: Maybe waste of resources
- Same number of pages for every application?

Operating Systems

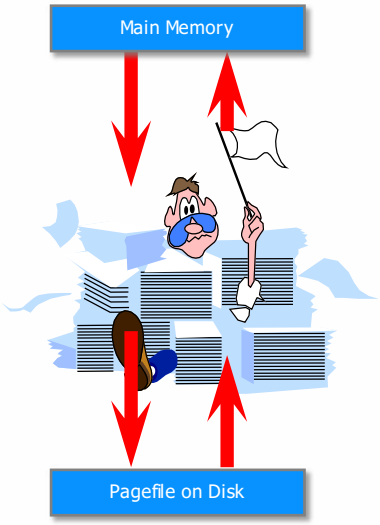
Thrashing

- Threshold k
- More than k pages allocated
  - Acceptable number of page faults
  - Some pages referenced rarely
  - Surplus on pages better used by other applications
- Exactly k pages allocated
  - Optimum
- Less than k pages allocated
  - Thread exhibits high page fault rate
  - OS loads pages continuously
  - Neglectable progress in thread
- How do we determine k?
  - Application dependent



Operating Systems

Thrashing is a Global Effect

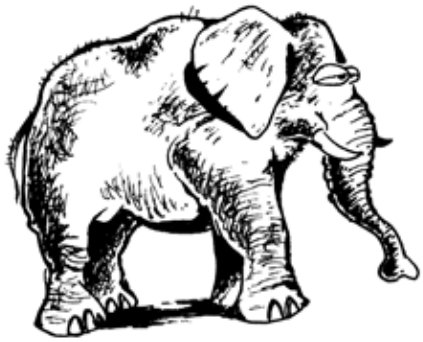


- Thrashing starts with one or more address spaces
  - Not enough memory assigned to application
- Increasing number of page ins and page outs on disk
- Average access time to disk increases
- Without precautions, the complete system stalls

Operating Systems

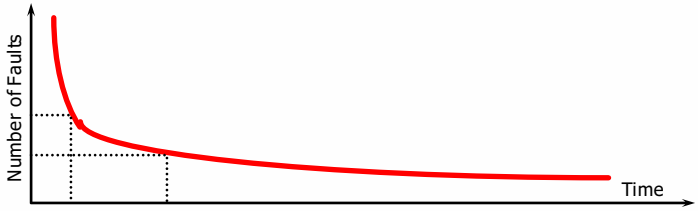
Lets Thrash ☺

- Write a small program that occupies a given number of memory pages very strongly?
- How does the operating system behave with an increasing number of program instances?



Operating Systems

Avoid Thrashing



- Tracking page fault rate
- Exceeding a threshold frequence: Assume thrashing
  - Assign additional pages (if possible)
  - Stop applications (write their pages to disk)
- Falling below a minium threshold frequency
  - Take away pages
  - Continue stopped applications

Operating Systems

Global and Local Replacement

- Are other address spaces involved in page fault?
- Local
  - No, another page of same address space must leave
  - Number of assigned pages remains constant
  - Observing working set useful
- Global
  - Yes, any other page might be replaced
  - Number of assigned pages varies
  - Operating system adapts to specific needs (e.g. priorities)
- Again, hybrid solutions are possible

Operating Systems

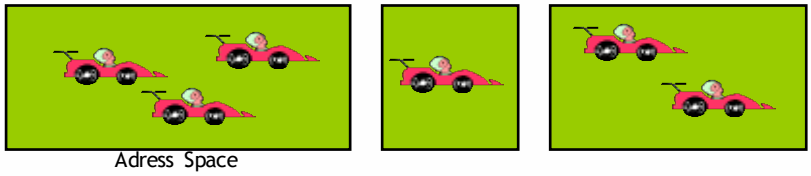
Operating Systems

Threads
Motivation

System Software Group
University of Trier
D-54286 Trier, Germany
Contact: Peter Sturm
sturm@uni-trier.de

Vision

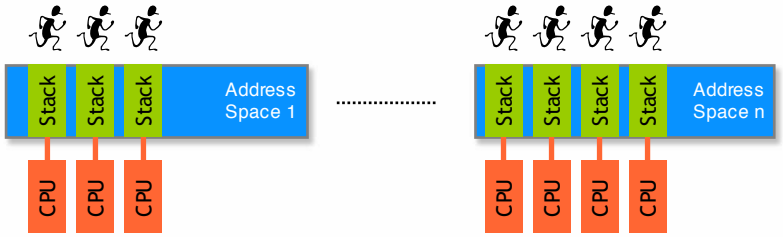
- Assume that there is an unlimited number of processors available
- Within an application
  - The required number of processors primarily depends on the inherent parallelism
- Between application
  - Every application may have its own processors



Adress Space

Operating Systems

Why Its Useful to Have "Unlimited" CPUs!

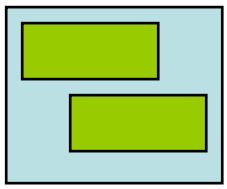


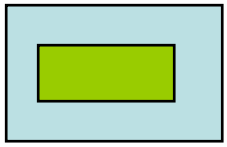
- Abstract from the number of CPUs on a given computer
  - Exploit concurrency inherent to an application
  - Utilize possible parallelism of multiprocessor systems
- Other reasons for multiple threads
  - Improve response time of interactive applications
  - Utilize parallelism available in most modern computer hardware

Operating Systems

... But 1 Physical CPU Available?

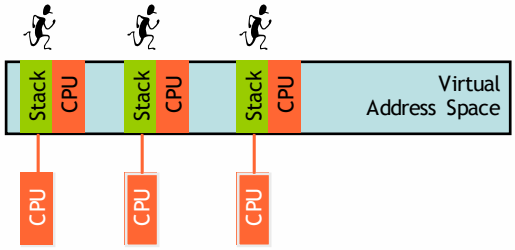
- Easy to map many virtual CPUs to a physical CPU
- A "Thread of Control" consists of
  - Address space
  - State of registers
  - Stack (already part of address space)
- Context Switch
  - Save current thread state
  - Restore another thread state
- When to switch
  - Calls that might block for a longer period of time
  - Implicitly, e.g. in case of page fault or memory congestion





Operating Systems

CPU Multiplexing



- Address space already maintained by operating system
- Each thread will have its runtime stack as part of address space
- Time-Multiplexing the CPU (Context Switch)
  - Storing the register content in memory
  - Restoring the register content of a previously stored thread
  - Switch address space if required (change pointer to root page table)

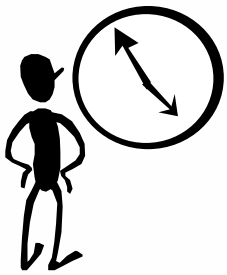
Operating Systems

Context Switches Per Second

- Sleep(0) yields the CPU
- By measuring the time a given number of Sleep(0) requires on an otherwise idle system, the average time for a context switch can be estimated
- Result depends on the CPU speed and other factors
- On a Intel Pentium 4 HT with 3.06 GHz clock speed, we measure approx. 630 ns for a context switch
- The hyper-threading feature leads to 50% system utilization, since only 1 CPU can execute the program

Program Behavior

- Most application programs can't execute continuously for a longer period of time
  - Waiting for user input
  - Waiting for a message
  - Waiting for a free buffer to send a message
  - Waiting for a block from disk
  - waiting for virtual memory to be paged in after page fault



- Some can
  - Number crunching, cryptography, simulations, …
  - Complex operations on large images (e.g. Photoshop)
  - Realtime games, chess, …

Operating Systems

Bursts

- Different phases of program behavior are called bursts
- CPU burst
  - Thread is running on a CPU
  - Burst length inherent to thread depends on type of application
- IO burst
  - Thread is waiting for some io operation to complete
  - Thread is not running
  - Most io bursts last at least for several milliseconds or longer
- Burst alternation
- Can OS exploit this typical program behavior?

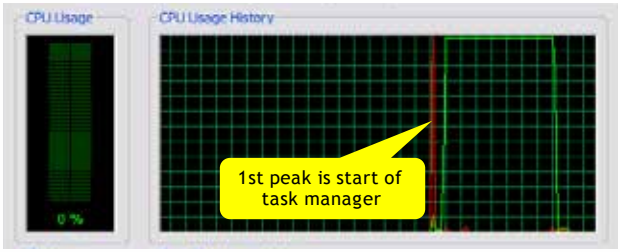
Operating Systems

Neverending CPU Burst

- Write a program that exhibits a very long CPU burst
  - The program should do some number crunching
  - It should have a small memory footprint to avoid paging
  - It should have no i/o
- Possible number crunching problems
  - Identify prime number in a given range
  - Check the cross product of values to be divisible by 7
- Determine the execution time for a given problem size when executing 1, 2, or more instances of the program in parallel
  - On a CPU without hyperthreading
  - On a CPU with hyperthreading
  - On a Dual CPU system (if available)
- Check the system state with the performance tab in the Windows Task Manager

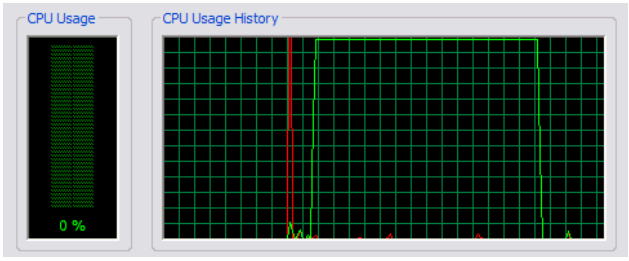
Count Primes

- Count the number of primes
- Execution time on a AMD 64 4000+ running Windows XP (32 Bit)
  - 44.02s (primes between 2 and 20000000)
  - 100% load



2 "Count Primes" in Parallel

- Executing 2 of these programs in parallel requires 88 s for both programs to end
- As expected: 2 * 44 s = 88 s
- Again 100% load

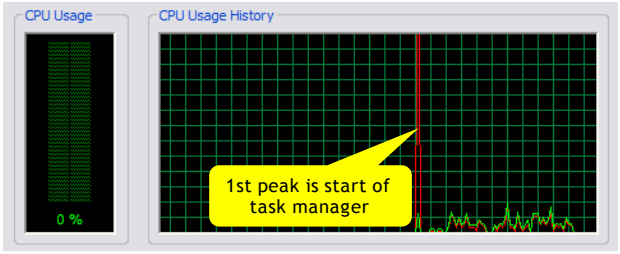


Mostly IO Burst

- Write a program that exhibits IO bursts only
  - Don't do number crunching at all
  - Write some easy to calculate data to disk
  - Write in a very fine granularity
- Determine the execution time for a given problem size when executing 1, 2, or more instances of the program in parallel
  - On a CPU without hyperthreading
  - On a CPU with hyperthreading
  - On a Dual CPU system (if available)
- Check the system state with the performance tab in the Windows Task Manager

Talk Nonsense

- Create a file containing nonsense
- Execution time on a AMD 64 4000+ running Windows XP (32 Bit)
  - 48.1s (file size 50000)
  - Between 0% and 4% load



2 "Talk Nonsense" in Parallel

- Writing two files on a single disk
  - Both programs finished after 88.7 s

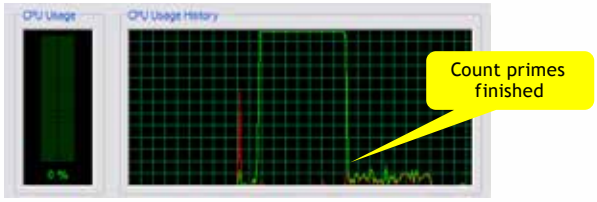
- Writing two files on different disks
  - Both programs finished after 62.2 s

Count Primes While Talking Nonsense ☺

- Start both programs simultaneously
- Determine the execution time
  - On a CPU without hyperthreading
  - On a CPU with hyperthreading
  - On a Dual CPU system (if available)
- Check the system state with the performance tab in the Windows Task Manager

Application Mix

- Total execution time for both applications shorter than expected
  - Count Primes alone needs 44 seconds
  - Talking Nonsense alone needs 48 seconds
  - Both executed in parallel requires 85 seconds instead of 92 seconds
- Depending on the balance in CPU bursts and IO bursts, speedup can still improve



Conclusion

- Parallelism inherent to most modern hardware can be exploited with a mixture of applications
- Speedup possible even on a monoprocessor system
  - Two apps with IO burst, if independent IO hardware is used
  - Applications with CPU bursts and IO bursts may coexist
- In many cases, hyperthreading can be very similar to a dual processor
- The beginning of an IO burst enables an operating system to transfer CPU time to other threads that are willing to execute

Operating Systems

Operating Systems

Threads
Multi-Threading

System Software Group
University of Trier
D-54286 Trier, Germany
Contact: Peter Sturm
sturm@uni-trier.de

Single-Threaded Process

- Traditional notion of process
  - An address space
  - Initial thread starting with the execution of `main()`
- Process terminates when initial thread leaves `main()`

Multiple Threads

- Additional threads must be instantiated explicitly
  - In .NET, additional instances of class Thread must be created
- Reasons to create multiple threads
  - Exploit hardware parallelism
  - Continue computation while another thread blocks
  - React immediately on user input

Operating Systems

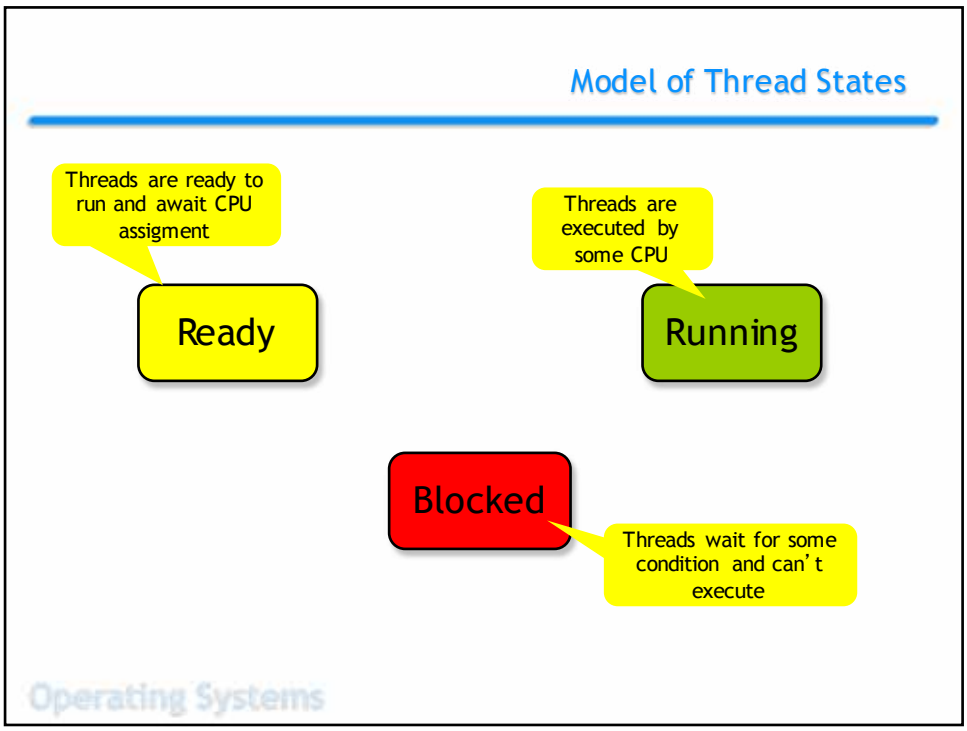
Threads
State Model

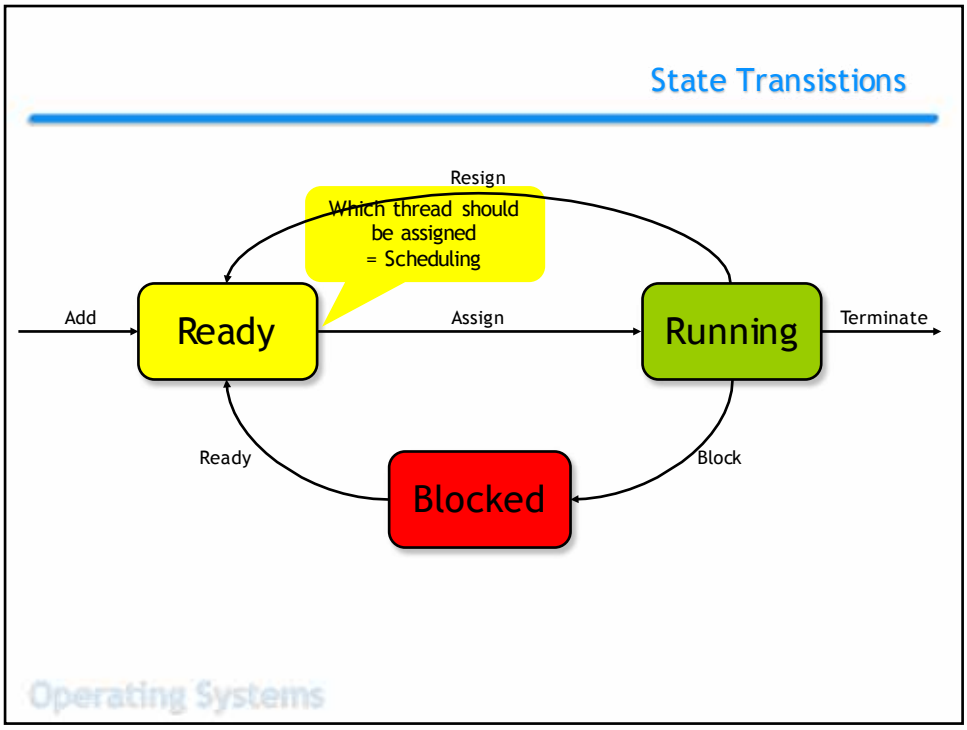
System Software Group
University of Trier
D-54286 Trier, Germany
Contact: Peter Sturm
sturm@uni-trier.de

States of a Thread

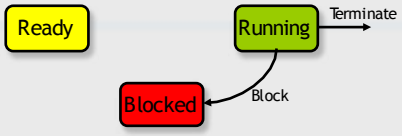
- Threads can be executed by some CPU
- Threads may be blocked
  - Waiting for a memory page to be read from disk
  - Waiting for a message to arrive
  - Waiting for some user input
- Threads are willing to execute and await CPU assignment

Operating Systems


Model of Thread States
Threads are ready to run and await CPU assigment
Threads are executed by some CPU
Ready
Running
Blocked
Threads wait for some condition and can't execute
Operating Systems


State Transistions
Resign
Which thread should be assigned = Scheduling
Add
Ready
Assign
Running
Terminate
Ready
Block
Blocked
Operating Systems

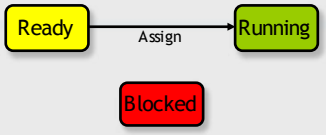
Block & Terminate



- Terminate
  - Thread terminates itself
    - Returns from top-level function executed (main, …)
    - Calls specific OS function (exit)
  - Thread is aborted by somebody else (permission required)
    - By user via job control
    - Other thread is calling an abort function
- Block
  - Page fault - Need to wait until page has been transferred
  - Waiting for some IO function to complete
  - Waiting for user input
  - Waiting for some synchronization condition to hold

Operating Systems

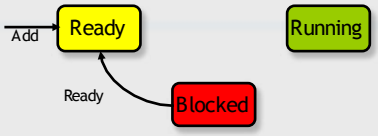
Assign



- Size of running queue is equal to number of CPUs in the system
- Each time a slot in running is free, system has to select another thread from the ready queue
- Normally, several (many) threads compete
- Scheduling strategy defines an execution order
  - Not necessarily fair

Operating Systems

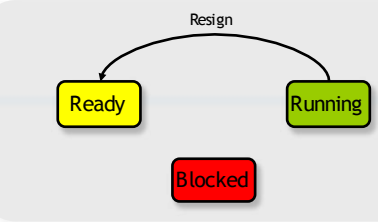
Ready & Add



- Add
  - New threads are prepared for execution and are put into the ready queue
- Ready
  - Page fault: The page is in main memory and machine instruction can be repeated
  - IO operation is completed
  - User input is available
  - Synchronization condition is fulfilled

Operating Systems

Resign



- Thread is yielding the CPU voluntarily
  - In Windows this is done by calling some sleep function but not sleeping at all (argument 0)
- Thread is exceeding a given time quantum
  - Important state transition to prevent monopolization of CPU
  - Enforced by many operating systems
- Depending on some scheduling strategy, a CPU must be assigned to a another more privileged thread immediately

Operating Systems

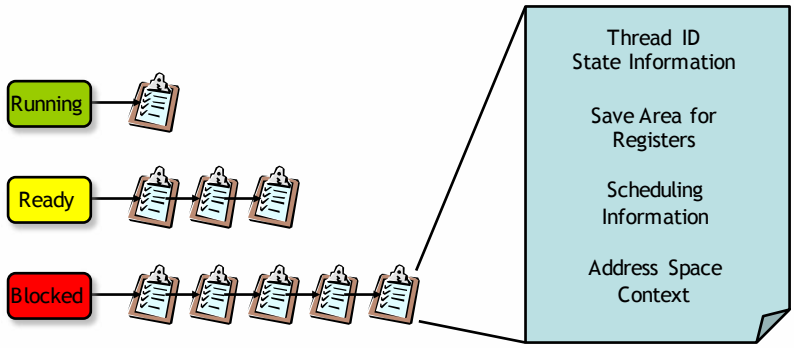
Dispatcher

- Responsible for performing all state transitions
- Several hardware-related issues have to be addressed
  - Saving and restoring registers in an orchestrated way
  - Information about address spaces may change
  - Sensible to asynchronous input such as interrupt
- As a consequence, the dispatcher consists of several assembler routines
- During the assign transition, dispatcher will access scheduler information to select appropriate thread
  - Short-term scheduler selects candidate thread fast (preferably in constant time)
  - Long-term scheduler keeps track of a threads in ready queue

Operating Systems

Implementation Issues

- At least from an conceptual point of view, the three lists are implemented as such
  - Due to efficiency constraints this may be different in practice
- All required state information is stored in a thread-specific data structure (tcb = thread control block)



Operating Systems

Operating Systems

Threads
Monoprocessor Scheduling

System Software Group
University of Trier
D-54286 Trier, Germany
Contact: Peter Sturm
sturm@uni-trier.de

Motivation

- Scheduling strategie depends on operational field
  - Batch processing
    - Non interactive, long execution times
  - Real-time processing
    - Guaranties for execution deadlines
  - Interactive processing
    - Quick responses to users, short CPU burts
- Hardware platform
  - Monoprocessor - Single CPU is a scarce resource
  - Multiprocessor
- Minimal number of context switches
  - Cheap but not for free (major problem earlier)
  - Beware of cold caches and TLBs

Structural Consequence

- Two-level approach
- Short-term scheduler
  - Called by dispatcher during state transition "assign"
  - Scheduling decision must be made very fast
- Long-term scheduler
  - Continuous update of state information relevant for all scheduling decisions
  - Preprocessing of expensive scheduling strategies
  - Information preparation for short-term scheduler

Operating Systems

Scheduling Criteria

- CPU utilization
  - Use maximum of CPU cycles for applications
- Throughput
  - Maximize the number of completed jobs per time interval
- Turnaround time
  - Minimize time between two consecutive job activations
- Waiting time
  - Minimize the time a thread stays in ready queue
- Response time
  - Favor interactive applications that react on user input
- Real time
  - Meet application-specific deadlines

Operating Systems

Preemptive and Non-preemptive

- Preemptive Scheduling
  - Thread will be forced to withdraw CPU
  - Reason may be exceeded time quantum or another more privileged thread must run
- Pro
  - Only way to prevent monopolization
  - Does improve response time
- Contra
  - Additional context switches
  - May add substantial overhead to total execution time of a thread
- Non-preemptive Scheduling
  - Thread yields CPU voluntarily
    - Implicitly in case of blocking operation
    - Explicitly
- Pro
  - Total execution time can be assessed (realtime systems)
- Contra
  - CPU monopolization possible
  - Fast response times can't be guaranteed in all cases

Operating Systems

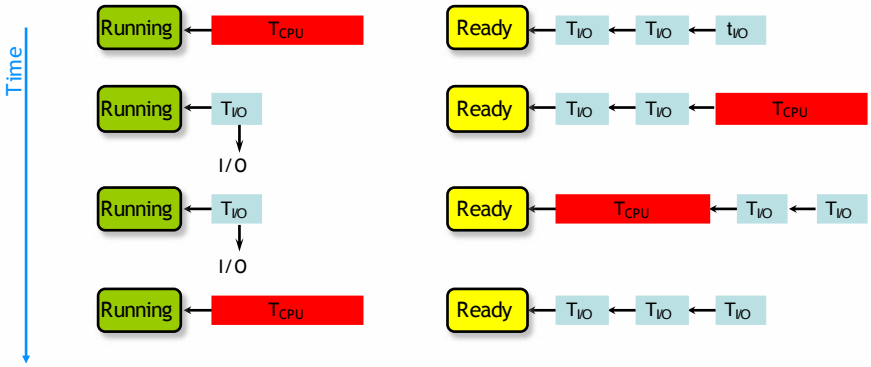
First-Come, First-Served (FCFS)

- CPU assignment in the order of arrival
- Non-preemptive scheduling strategy
- Simple implementation via queue
  - Thread at queue head will be assigned next
  - Threads entering the ready queue will be appended to tail
- Average waiting time may vary

Operating Systems

The Convoi Effect of FCFS

- A single thread with a very long CPU burst may slow down the whole system
- Something you may recognize from crowded highways ☺

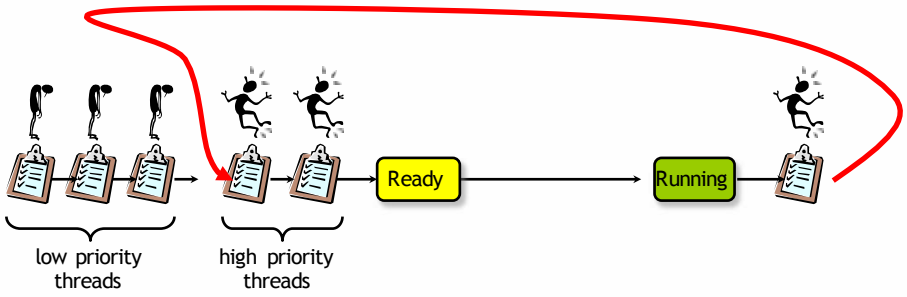


Priority-based Scheduling

- Priorities are assigned to threads
  - Explicitly: Defined by application itself, by user, or by some administrator
  - Implicitly: Defined by operating system
- Thread with highest priority will be assigned to next free CPU
- Preemptive version (common)
  - A running thread will be preempted if a thread with higher priority gets ready
    - New thread with higher priority created (add transition)
    - Previously blocked thread with higher priority is ready again (ready transition)
- Non-preemptive version
  - Thread with highest priority will be scheduled on next assign transition

Starvation

- Priority-based scheduling doesn't guarantee fariness
- Example
  - Threads with high priorities are always ready to run
  - Threads with low priorities are always ready to run too



Operating Systems

Round-Robin (RR)

- RR = FCFS with preemption
  - Prevalent scheduling strategy for multitasking systems
  - Exceeding a given time quantum lead to preemption
  - Typical lengt of time quantum between 10 ms and 20 ms
- Size of time quantum critical for overall system performance
  - too large: FCFS (most threads block before their quantum exceeds)
  - too small: too many additional context switches
- Average waiting time
  - Depends on number of threads
    - n threads and time quantum tq

$$Waittime_{Average} \leq tq * (n-1)$$

  - Length of CPU burst has no influence

Operating Systems

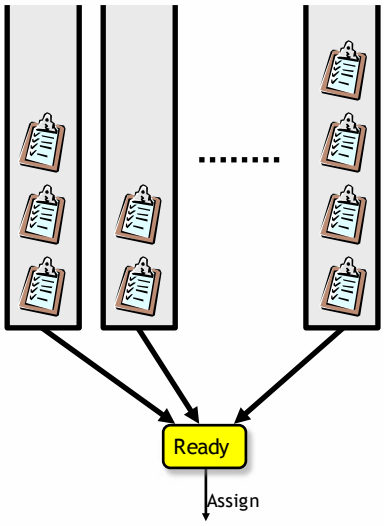
Threads Doing IO are Penalized

- RR favors threads with long CPU burst
  - exhaust their time quantum
- Threads doing IO yield CPU before time quantum exceeded
- Possible solution
  - Multi-level scheduling
    - Additional queue for threads not exhausting their quantum
  - Hierarchy of two scheduling strategies
    - 1st Level: Which queue to select next?
    - 2nd Level: Order within queue?

Operating Systems

Multi-Level Scheduling

- Ready queue partitioned into several sub-queues
  - Address different scheduling goals
- 1st level strategy
  - Which sub-queue to select?
  - Priorities are very common
- 2nd level strategy
  - Which thread within queue will get next CPU
  - Depending on scheduling goal, different strategies are used

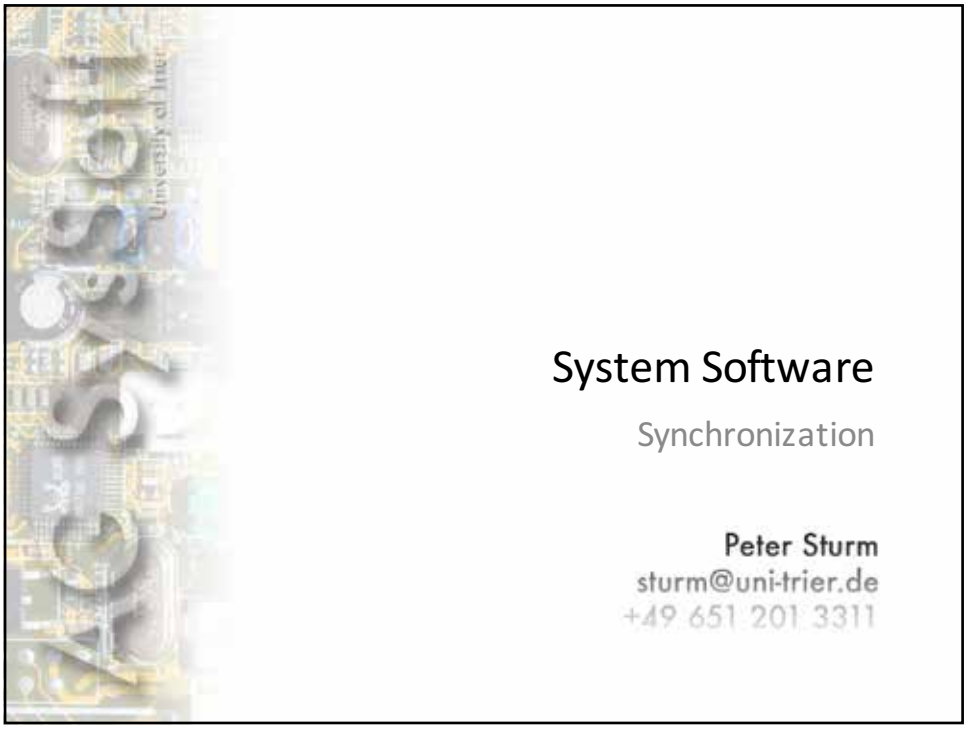


Operating Systems

Feedback Scheduling

- Many scheduling strategies take the past into account
  - SJF approximation makes scheduling decision on the length of previous CPU bursts
  - In interactive environments, threads exhausting their time quantum should be penalized
  - The application currently holding the UI focus should be favorized (priority boost done by Windows operating systems)
  - Aging of priorities in order to minimize risk of starvation
  - …
- Requires additional tracking of all relevant state information
  - Stored within thread control block
- Feedback may lead to complex and unpredictable behavior

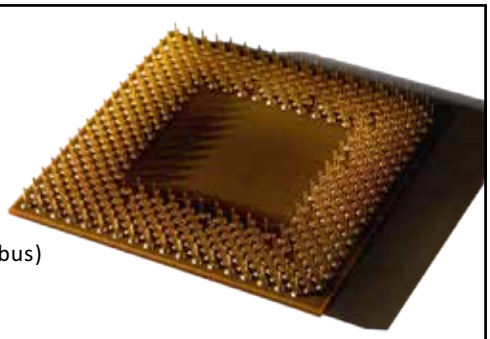
Operating Systems



Das Kontinuum

- Aktivität (Zeit)
  - Abfolge ausgeführter Instruktionen
  - Kontrollfluß oder Thread
  - Unterstützung für mehrere Threads
  - Concurreny Control
- Raum
  - Menge der adressierbaren Speicherzellen
  - Adressraum oder “Adress space”
  - Unterstützung für mehrere Adreßräume

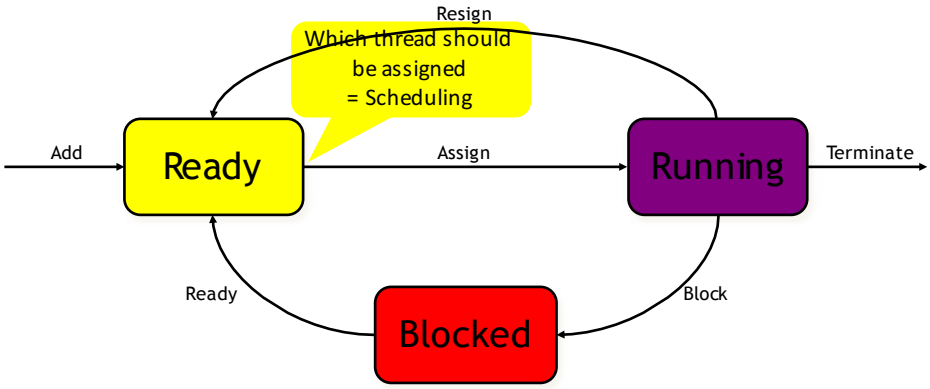
Adreßraum



- CPU definiert ein theoretisches, oberes Limit
  - Anzahl der Adreß-Pins (address bus)
- Beispiele
  - 32 bit Architektur (Intel Pentium, AMD Athlon XP, ...)
    - 32 Adressleitungen = 4 GByte Adreßraum
  - 64 bit Architektur (AMD Athlon 64, Intel Itanium, ...)
    - 64 Leitungen = 4194304 TByte theoretischer Adreßraum
    - Real limitiert auf ~40 Pins (Server 48 Pins)

Virtuelle Prozessoren

State Transistions

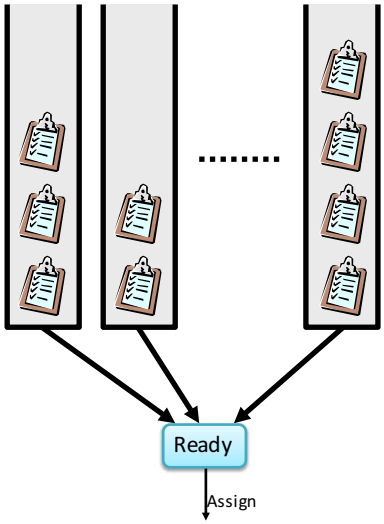


Round-Robin (RR)

- RR = FCFS mit Preemption
  - Weit verbreiteter Ansatz
  - Überschreiten eines Zeitquantums führt zur Preemption
  - Gängige Werte: 10 ms and 20 ms

Multi-Level Scheduling

- Bereit-Liste wird unterteilt
- 1st Level
  - Welche Teilliste?
  - Meist Prioritäten
- 2nd Level
  - Welcher Thread in gewählter Liste?
  - Häufig RR

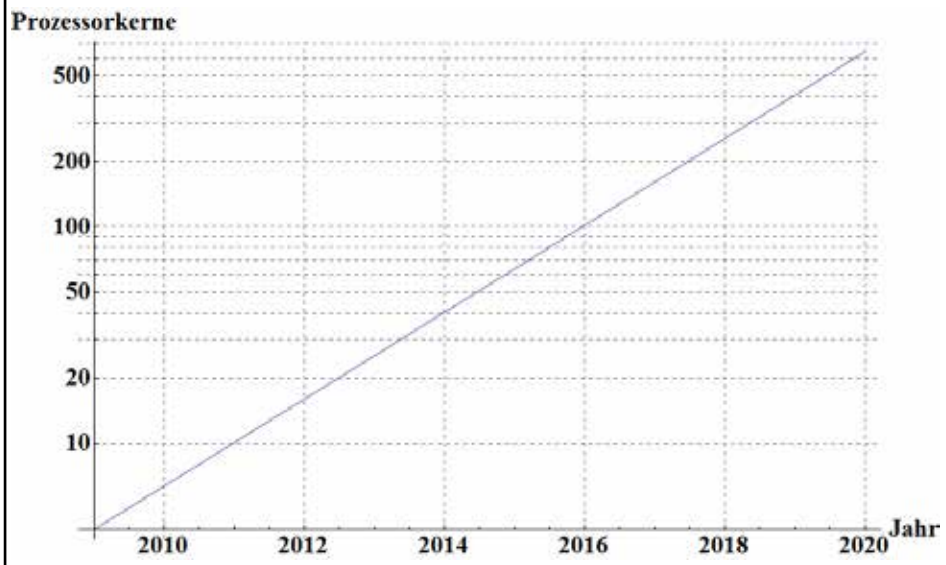


Gordon Moore

Gene Amdahl

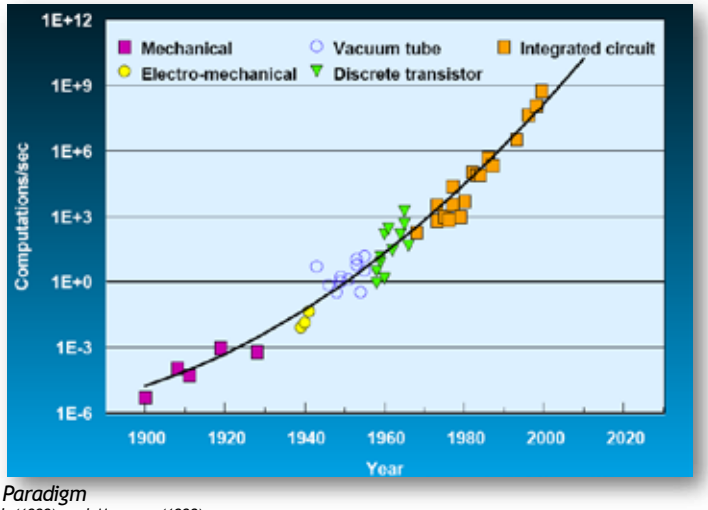
AMDAHL UND MOORE …

Moores Gesetz



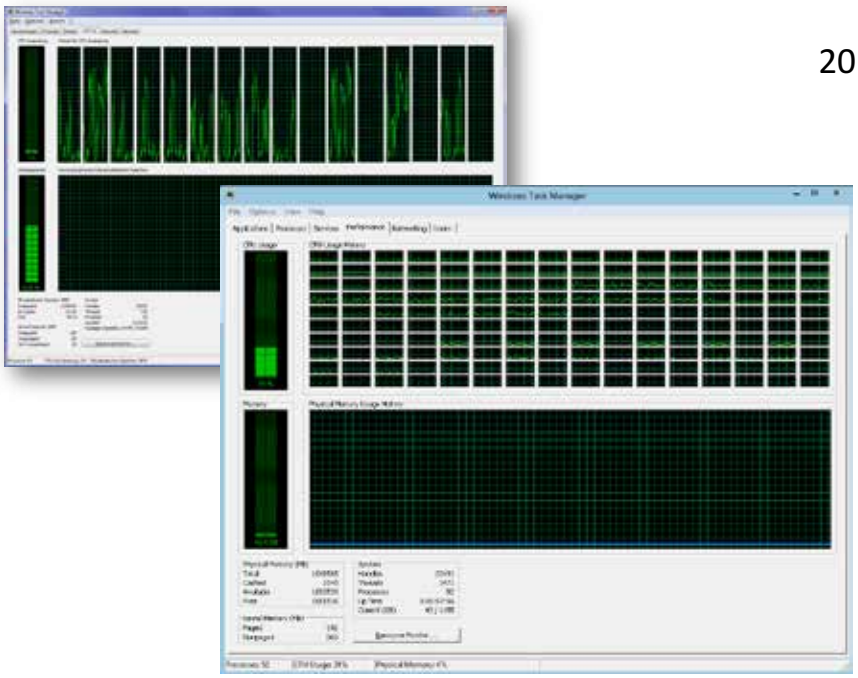
Annahmen: *Verdopplung Kernzahl alle 18 Monate*
4-Kern-Prozessor in 2009

Moores Gesetz …



The Fifth Paradigm
Nach Kurzweil (1999) und Moravec (1998)

2014?

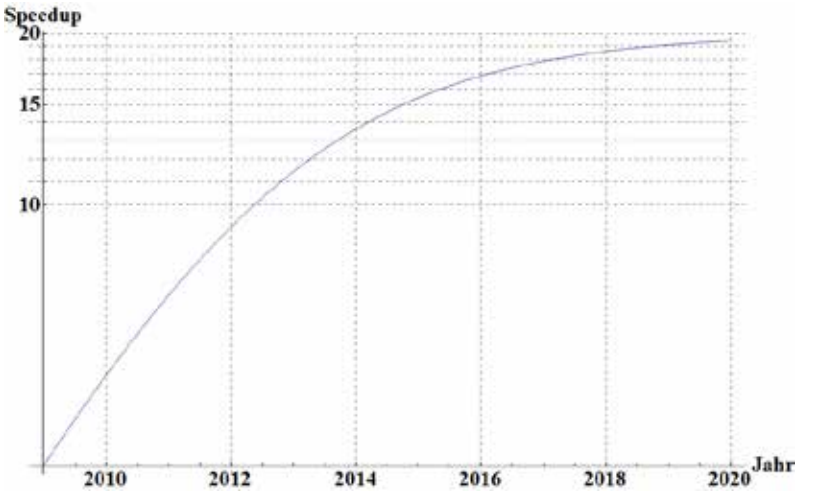


Weiter?

- 2014
  - 64 Cores
- 2018
  - 1024 Cores
- 2028
  - 8192 Cores
  - Many Cores
- …
  - Enough Cores ☺

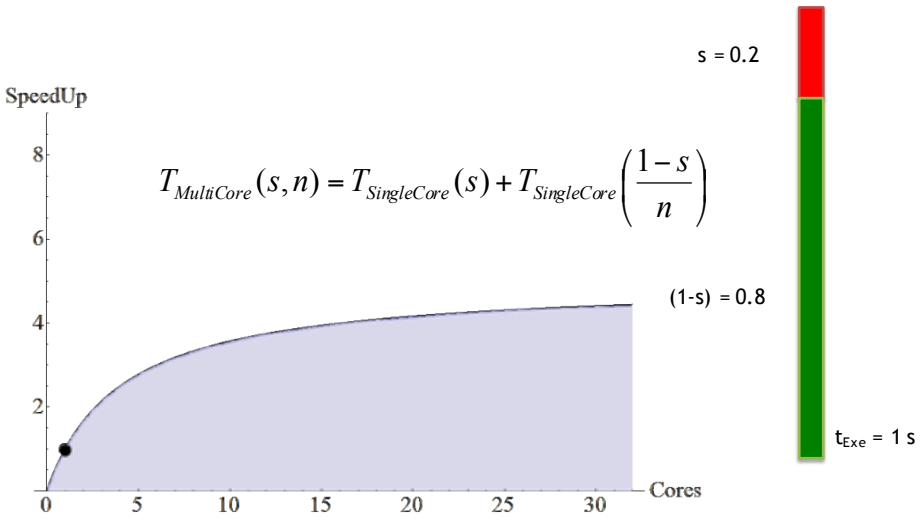
Nicht-parallele Software wird nie mehr schneller laufen als heute!

Amdahls Gesetz



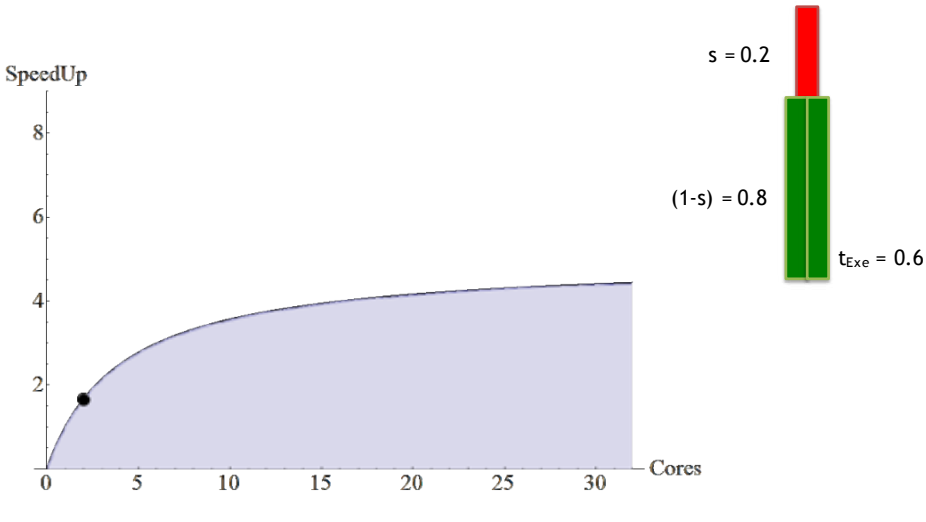
Annahmen: *Zu 95% parallelisierbare Software*
Gleichbleibende Taktfrequenz
Speedup gegenüber SingleCore

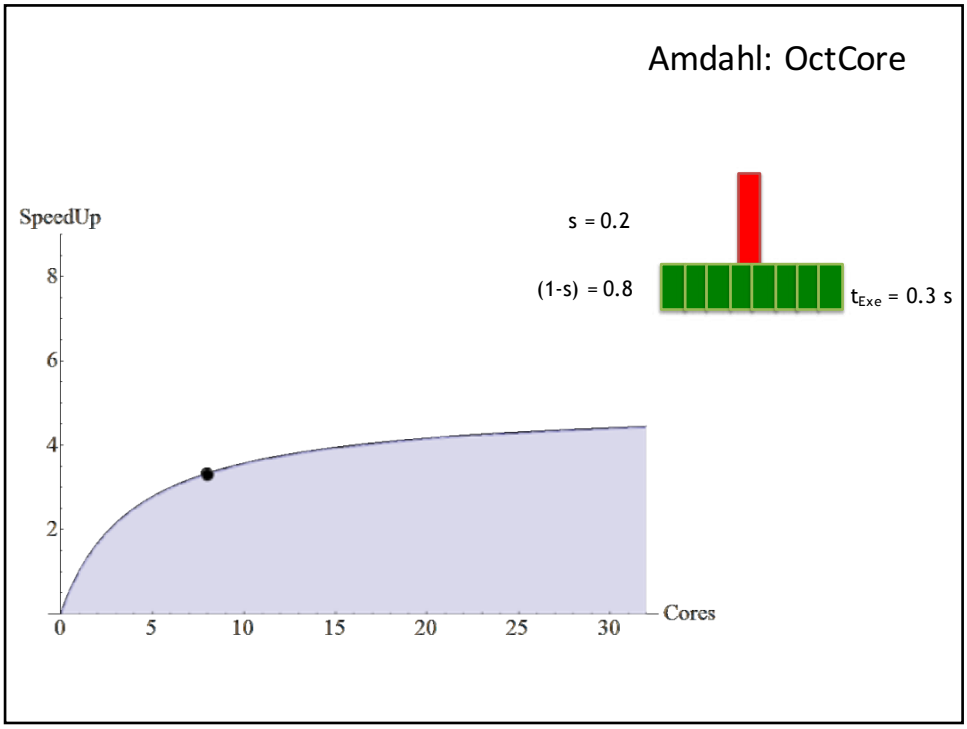
Amdahl: SingleCore

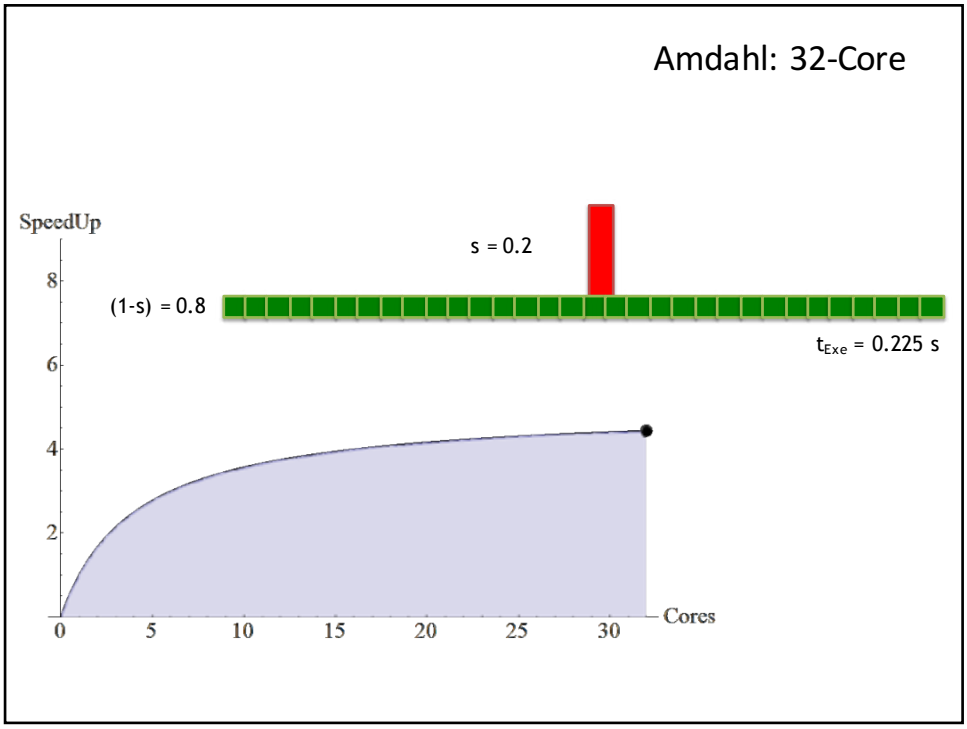


$$T_{MultiCore}(s,n) = T_{SingleCore}(s) + T_{SingleCore}\left(\frac{1-s}{n}\right)$$

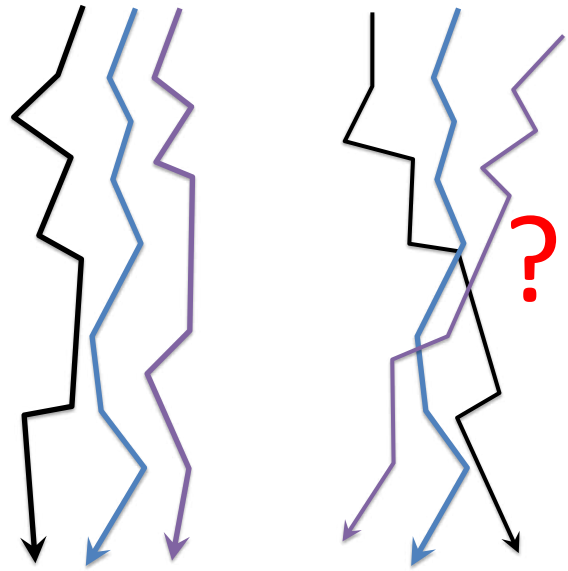
Amdahl: DualCore




Amdahl: OctCore
SpeedUp
s = 0.2
(1-s) = 0.8
t_{Exe} = 0.3 s
8
6
4
2
0
5
10
15
20
25
30
Cores

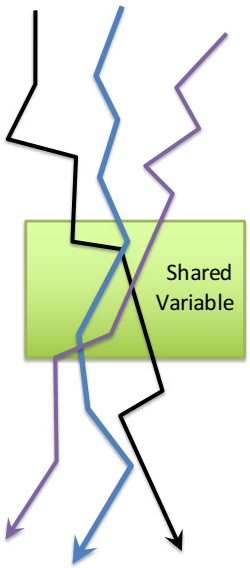

Amdahl: 32-Core
SpeedUp
s = 0.2
(1-s) = 0.8
t_{Exe} = 0.225 s
8
6
4
2
0
5
10
15
20
25
30
Cores

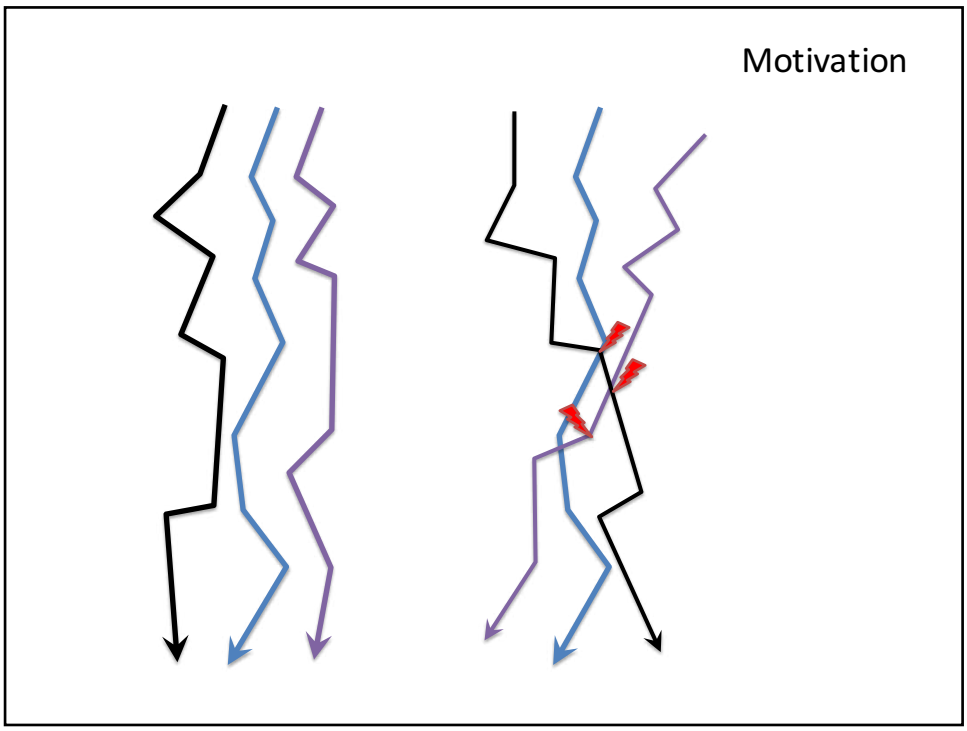
Motivation



Let‘s try!

- Example program
  - Multiple threads
  - Accessing a single integer variable

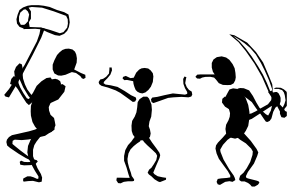



Motivation

Konkurrenz und Kooperation

Konkurrenz

- Kampf um Ressourcen
  - Ziel: Alles für mich
- Implizite Synchronisation
  - Virtuelle Ressourcen
  - OS serialisiert
- Wechselseitiger Ausschluß



Kooperation

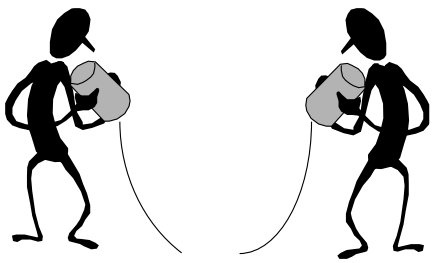
- Kooperierende Threads
- Teilt sich Ressourcen & Aufgaben
- Ziele
  - Speedup
  - Reaktionsgeschwindigkeit
- Beliebig komplexe Synchronisationsprobleme

Mutual Exclusion

- Wechselseitiger Ausschluß
- Atomarität für eine Gruppe von Befehlen
- Höhere Synchronisationslösungen ableitbar
- Paßt zum Scheduling
  - Blockiert-Liste wenn belegt

IPC

- Interprozeßkommunikation
  - Synchronisation
    - A wartet auf Bedingung
    - B erfüllt Bedingung und informiert A
    - 1-Bit-Nachricht
  - Kommunikation
    - Austausch von Daten
- Vielfältige Verfahren

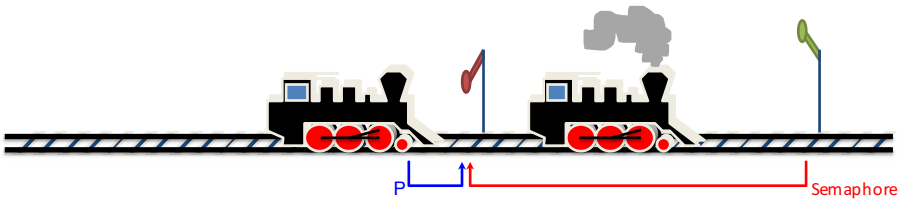


Abstand

- In einem Prozeß (Adreßraum)
  - Zugriff auf gemeinsamen Speicher
- Zwischen Prozessen auf einem Rechner
- Zwischen Prozessen auf verschiedenen Rechnern
  - Verteilte Systeme
  - Internet, IP, TCP, UDP

SEMAPHORE

Semaphore



- Fundamental synchronization primitive
- Two basic functions (on a semaphore s)
  - s.P()
    - May block in case semaphore already "occupied"
  - s.V()
    - Never blocks; releases semaphore and may free a blocked thread
- Dijkstra (1968), "THE" Multiprogramming System
  - P = passeren (request entry)
  - V = vrygeven (release)

Semantic

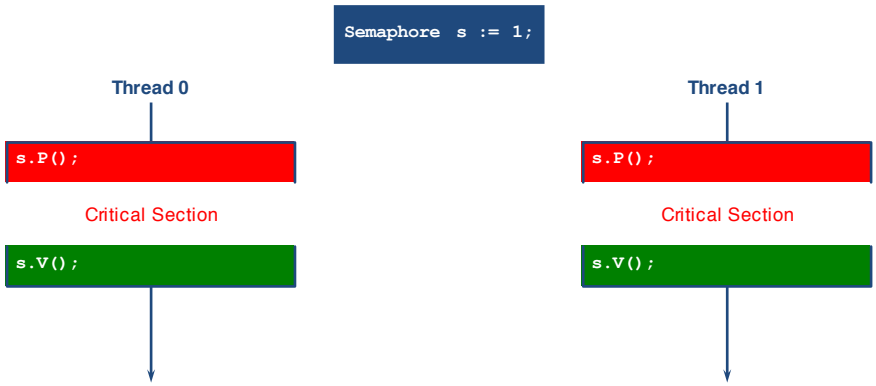
- Semaphores are counting signals
  - s.P(): Thread waits for some signal s
  - s.V(): Thread sends a signal s
- Semaphor = Integer (s.value)
  - Initialized with a s.value ≥ 0

```
void P () {
  s.value--;
  if (s.value < 0)
    Block calling thread;
}
```

Of course, P and V themselves are critical sections and must be protected!

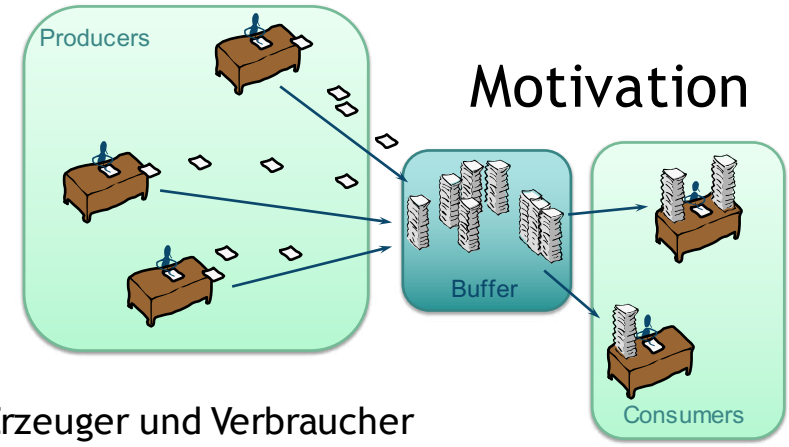
```
void V () {
  s.value++;
  if (s.value < 1)
    Put a thread waiting in s into ready queue;
}
```

Mutual Exclusion With Semaphore



PRODUCER/CONSUMER

Motivation



- Erzeuger und Verbraucher
- Puffer fester Größe
  - Geschwindigkeitsunterschiede
- Wie löst man das Problem?

Outline of Buffer

```
public class Buffer
{
    public Buffer ( int n )
    {
    }

    /// Inserts another good into the buffer.
    /// May block until free space is available.
    public void Produce ( int good )
    {
    }

    /// Consumes a good stored inside the buffer.
    /// May signal blocked producer threads.
    public int Consume ()
    {
        return 42;
    }
}
```

Producer (C#)

```
public class Producer
    {
        public Producer ( Buffer b )
        {
            buffer = b;
            my_id = this.GetHashCode();
            ThreadStart ts = new ThreadStart(Run);
            my_thread = new Thread(ts);
            my_thread.Start();
        }

        private void Run ()
        {
            Console.WriteLine("Producer {0}: started ...",my_id);
            int good = this.GetHashCode() * 1000000;
            while (true)
            {
                buffer.Produce(good);
                Console.WriteLine("Producer {0}: good {1} stored",my_id,good);
                good++;
            }
        }

        private Buffer buffer;
        private Thread my_thread;
        private int my_id;
    }
```

Consumer (C#)

```
public class Consumer
    {
        public Consumer ( Buffer b )
        {
            buffer = b;
            my_id = this.GetHashCode();
            ThreadStart ts = new ThreadStart(Run);
            my_thread = new Thread(ts);
            my_thread.Start();
        }

        private void Run ()
        {
            Console.WriteLine("Consumer {0}: started ...",my_id);
            while (true)
            {
                int good = buffer.Consume();
                Console.WriteLine("Consumer {0}: good {1} retrieved",my_id,good);
            }
        }

        private Buffer buffer;
        private Thread my_thread;
        private int my_id;
    }
```

The Buffer (C#)

```
public class Buffer
    {
        public Buffer ( int n )
        {
            this.n = n;
            slots = new int[n];
            mutex = new Semaphore(1);
            slots_available = new Semaphore(n);
            goods_available = new Semaphore(0);
        }

        …

        private int n;
        private int [] slots;
        private int free = 0;
        private int used = 0;
        private Semaphore mutex;
        private Semaphore slots_available;
        private Semaphore goods_available;
    }
```

```
public void Produce ( int good )
{
    slots_available.P();
    mutex.P();
    slots[free] = good;
    free = (free+1) % n;
    mutex.V();
    goods_available.V();
}

public int Consume ()
{
    goods_available.P();
    mutex.P();
    int good = slots[used];
    used = (used+1) % n;
    mutex.V();
    slots_available.V();
    return good;
}
```

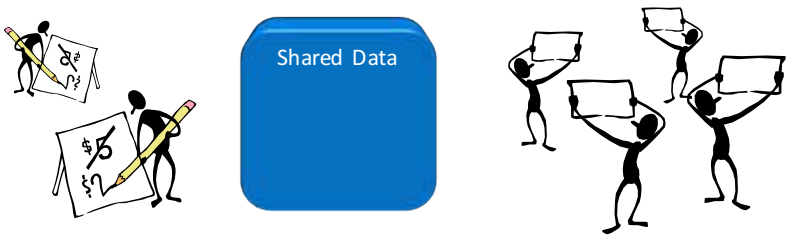
The Optimal Buffer (C#)

```
public class Buffer
    {
        public Buffer ( int n )
        {
            this.n = n;
            slots = new int[n];
            mutex_p = new Semaphore(1);
            mutex_c = new Semaphore(1);
            slots_available = new Semaphore(n);
            goods_available = new Semaphore(0);
        }

        …

        private int n;
        private int [] slots;
        private int free = 0;
        private int used = 0;
        private Semaphore mutex_p, mutex_c;
        private Semaphore slots_available;
        private Semaphore goods_available;
    }
```

```
public void Produce ( int good )
{
    slots_available.P();
    mutex_p.P();
    slots[free] = good;
    free = (free+1) % n;
    mutex_p.V();
    goods_available.V();
}

public int Consume ()
{
    goods_available.P();
    mutex_c.P();
    int good = slots[used];
    used = (used+1) % n;
    mutex_c.V();
    slots_available.V();
    return good;
}
```

READER/WRITER

The Problem

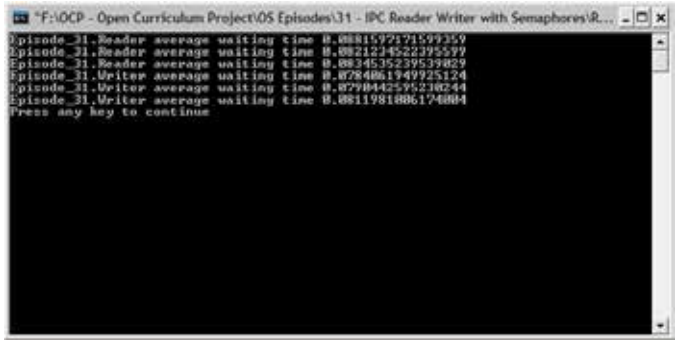


- A shared data structure is used by multiple threads
- Writer threads
  - These threads modify the shared data and require mutual exclusion
- Reader threads
  - Since readers don't modify data, they can access the shared data structure simultaneously

1. Mutual Exclusion

- As a starting point, implement a simple solution based on mutual exclusion for readers and writers. Of course, this version doesn't exploit the potential for parallelism by allowing multiple readers to access the shared data simultaneously.
- Start from scratch (C#)
- Inspect the complete example program (C#)

Works as Expected



- But only one reader inside sanctum at any given time

Starting Point: Mutual Exclusion

```
Semaphore Sanctum = new Semaphore(1);
```

Shared Data

```
while (true) {
  Sanctum.P();
  // Change data
  Sanctum.V();
}
```

Writer

```
while (true) {
  Sanctum.P();
  // Read data
  Sanctum.V();
}
```

Reader

2. Reader Preference

- Improve the previous program to enable multiple readers to access shared datas concurrently
- Hint
  - Only the first reader must acquire semaphore
  - Subsequent readers just enter
  - Only the last reader leaving must release semaphore
- We accept the risk of starvation for writers
- This is a still faulty version of the program ☺

Reader Preference (Faulty)

Shared Data

```
Semaphore Sanctum = new Semaphore(1);
int readers_inside = 0;
```

```
while (true) {
  Sanctum.P();
  // Change data
  Sanctum.V();
}
```

Writer

```
while (true) {
  if (readers_inside == 0)
    Sanctum.P();
  readers_inside++;
  // Read data
  readers_inside--;
  if (readers_inside == 0)
    Sanctum.V();
}
```

Reader

Why Does It Fail?

```
while (true) {
  if (readers_inside == 0)
    Sanctum.P();
  readers_inside++;
  // Read data
  readers_inside--;
  if (readers_inside == 0)
    Sanctum.V();
}
```

Reader

- Multiple readers may access the shared variable readers_inside concurrently
- Testing and setting variable must be atomic among readers
- Mutual exclusion required

2. Reader Preference (Correct)

Shared Data

```
Semaphore Sanctum = new Semaphore(1);
Semaphore RMutex = new Semaphore(1);
int readers_inside = 0;
```

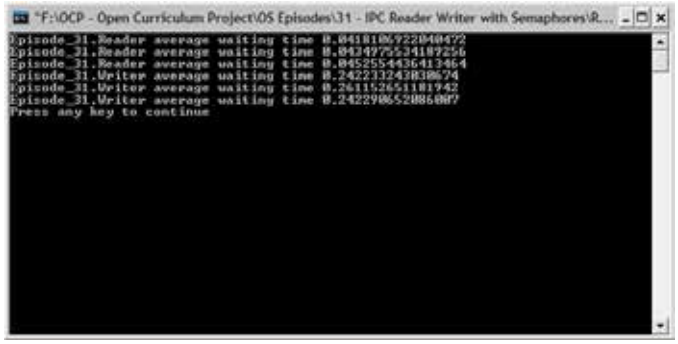
Writer

```
while (true) {
  Sanctum.P();
  // Change data
  Sanctum.V();
}
```

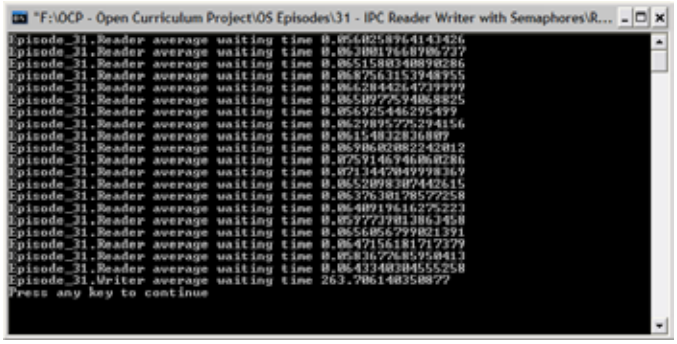
Reader

```
while (true) {
  RMutex.P();
  if (readers_inside == 0)
    Sanctum.P();
  readers_inside++;
  RMutex.V();
  // Read data
  RMutex.P();
  readers_inside--;
  if (readers_inside == 0)
    Sanctum.V();
  RMutex.V();
}
```

Obviously Reader Preference



But Writer May Starve



- It takes a long time for a single writer, if there are 20 readers continuously accessing shared data
- Sometimes forever ☹

3. Writer Preference

- The moment, a single writer is requesting to enter the shared data section, no other reader might enter anymore
- The writer has to wait, until all readers inside left the sanctum
- Subsequent writers are preferred
- We accept a possible starvation among readers
- Hint
  - Similar to reader preference, we count the number of writers willing to enter
  - Another semaphore locks readers in case of writer interest

3. Writer Preference

Shared Data

```
Semaphore Sanctum = new Semaphore(1);
Semaphore RMutex = new Semaphore(1);
Semaphore WMutex = new Semaphore(1);
Semaphore PreferWriter = new Semaphore(1);
int readers_inside = 0;
int writers_interested = 0;
```

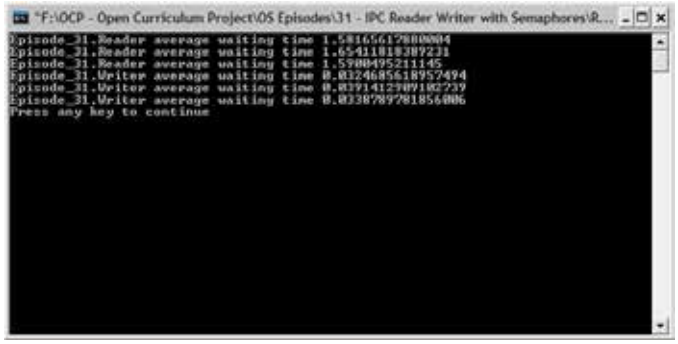
```
while (true) {
  WMutex.P();
  if (writers_interested  == 0)
    PreferWriter.P();
  writers_interested++;
  WMutex.V();
  Sanctum.P();
  // Change data
  Sanctum.V();
  WMutex.P();
  writers_interested--;
  if (writers_interested  == 0)
    PreferWriter.V();
  WMutex.V();
}
```

Writer

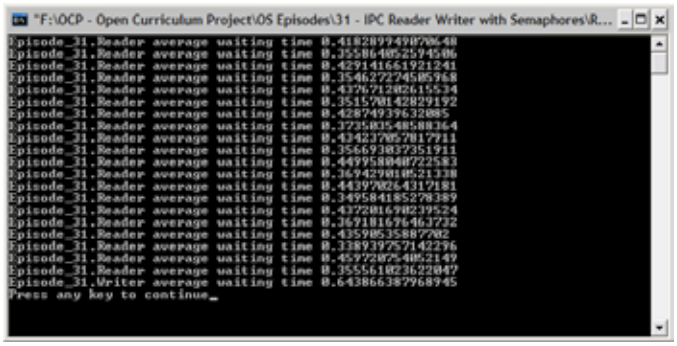
```
while (true) {
  PreferWriter.P();
  RMutex.P();
  if (readers_inside  == 0)
    Sanctum.P();
  readers_inside++;
  RMutex.V();
  PreferWriter.V();
  // Read data
  RMutex.P();
  readers_inside--;
  if (readers_inside  == 0)
    Sanctum.V();
  RMutex.V();
}
```

Reader

Works Great



… and Writer Doesn't Starve



4. Writer Preference (2nd Edition)

- Any ideas to improve the program with writer preference even further?
- Hint
  - A single writer may compete with multiple readers when calling **`PreferWriter.P()`**
  - Is it possible to limit that to at most one reader?

4. Writer Preference (Improved)

Shared Data

```
Semaphore Sanctum = new Semaphore(1);
Semaphore RMutex = new Semaphore(1);
Semaphore WMutex = new Semaphore(1);
Semaphore PreferWriter = new Semaphore(1);
Semaphore ReaderQueue = new Semaphore(1);
int readers_inside = 0;
int writers_interested = 0;
```

```
while (true) {
  WMutex.P();
  if (writers_interested == 0)
    PreferWriter.P();
  writers_interested++;
  WMutex.V();
  Sanctum.P();
  // Change data
  Sanctum.V();
  WMutex.P();
  writers_interested--;
  if (writers_interested == 0)
    PreferWriter.V();
  WMutex.V();
}
```

Writer

```
while (true) {
  ReaderQueue.P();
  PreferWriter.P();
  ReaderQueue.V();
  RMutex.P();
  if (readers_inside == 0)
    Sanctum.P();
  readers_inside++;
  RMutex.V();
  PreferWriter.V();

  // Read data
  RMutex.P();
  readers_inside--;
  if (readers_inside == 0)
    Sanctum.V();
  RMutex.V();
}
```

Reader

4. Writer Preference (Improved but faulty)

Shared Data

```
Semaphore Sanctum = new Semaphore(1);
Semaphore RMutex = new Semaphore(1);
Semaphore WMutex = new Semaphore(1);
Semaphore PreferWriter = new Semaphore(1);
Semaphore ReaderQueue = new Semaphore(1);
int readers_inside = 0;
int writers_interested = 0;
```

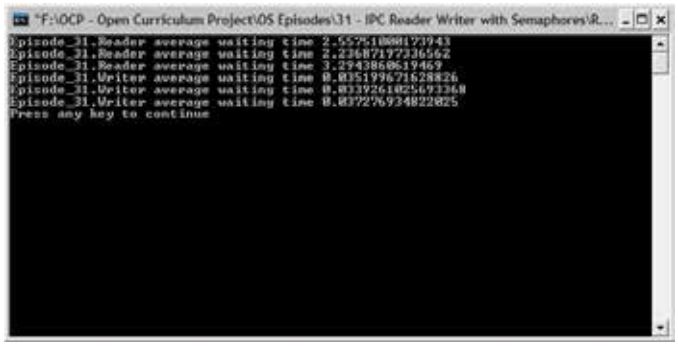
```
while (true) {
  WMutex.P();
  if (writers_interested == 0)
    PreferWriter.P();
  writers_interested++;
  WMutex.V();
  Sanctum.P();
  // Change data
  Sanctum.V();
  WMutex.P();
  writers_interested--;
  if (writers_interested == 0)
    PreferWriter.V();
  WMutex.V();
}
```

Writer

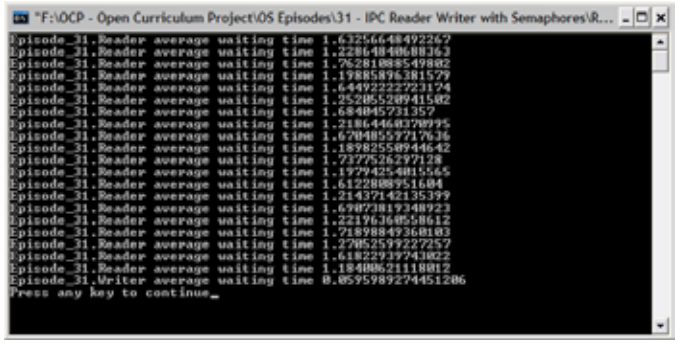
```
while (true) {
  ReaderQueue.P();
  PreferWriter.P();
  RMutex.P();
  if (readers_inside == 0)
    Sanctum.P();
  readers_inside++;
  RMutex.V();
  ReaderQueue.V();
  PreferWriter.V();

  // Read data
  RMutex.P();
  readers_inside--;
  if (readers_inside == 0)
    Sanctum.V();
  RMutex.V();
}
```

Reader

Works Even Better



… and Writer Still Doesn't Starve



One Semaphore too much?

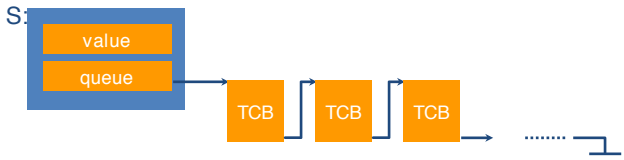
- Why not throw semaphore ReaderQueue away and move RMutex before PreferWriter? Wouldn't that serialize all readers too?

```
while (true) {
  ReaderQueue.P();
  PreferWriter.P();
  RMutex.P();
  if (readers_inside == 0)
    Sanctum.P();
  readers_inside++;
  RMutex.V();
  PreferWriter.V();
  ReaderQueue.V();
  // Read data
  RMutex.P();
  readers_inside--;
  if (readers_inside == 0)
    Sanctum.V();
  RMutex.V();
}
```

Reader

IMPLEMENTATION DETAILS

Implementation



P

- Execution of P must be atomic
- Thread k calls P

```
s.value--;
if (s.value < 0) {
   tail(s.queue) := k;
   block(k);
   }
```

V

- Execution of V must be atomic
- Calling thread normally doesn't block

```
s.value++;
if (s.value <= 0) {
   ready(head(s.queue))
   delhead(s.queue)
   }
```

Binary Semaphores

- Value must be 0 or 1
- Easy to implement with general semaphores
  - Beware of V operation: Simple increment invalidates the semaphore value
- P

```
if (s.value = 0) {
   tail(s.qqueue) := k;
   block(k);
   }
s.value := 0;
```

- V

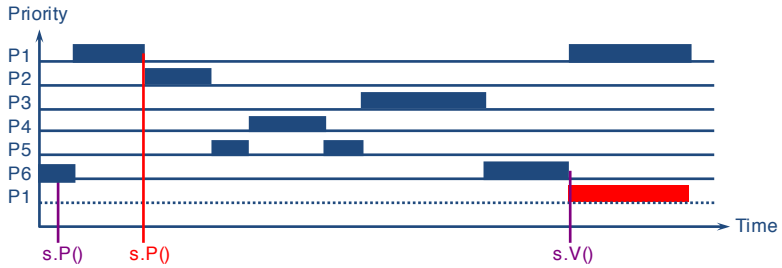
```
if (s.qqueue != 0) {
   ready(head(s.queue));
   delhead(s.queue)
   }
else
   s.value := 1;
```

Nested (binary) Semaphores

```
Bsemaphor  Mutex(1);

f1 (...) {          f2 (...) {                    fn (...) {
   Mutex.P();          Mutex.P();                    Mutex.P();
   ...                 f1();                         ...
   Mutex.V();          Mutex.V();                    Mutex.V();
   }                   }                             }
```

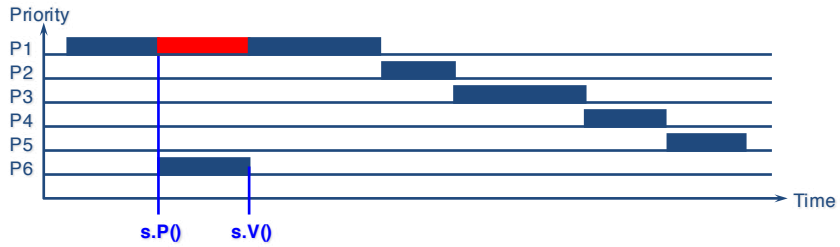
- Make multiple critical sections (procedures) thread-safe
  - Mutual exclusion among library functions
  - Reentrance
- Most semaphore implementations are smart or friendly

Semaphores and Scheduling Priorities



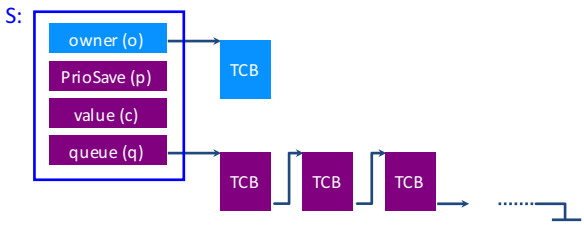
- P1 and P6 are synchronized using semaphore s
- Priority inversion
  - Apparent priority of P1 is less than priority of P6
- Dangerous for real-time systems

Priority Inheritance



- Low priority thread inherits higher priority until calling V
- Any modern OS supports inheritance
- Not necessarily dequeue head of blocked list (FCFS)

Implementing Priority Inheritance



- Limited to binary semaphores
- P operation
  - No blocking: Store ownership and current priority
  - Blocking: Transfer higher priority to owner thread if needed
- V operation
  - Restore original priority
  - Put blocked thread with highest priority in ready queue

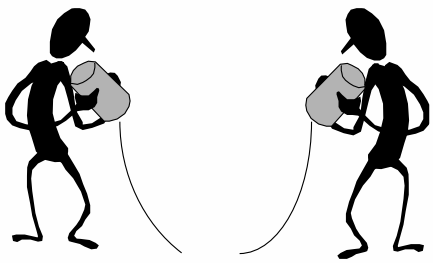
Operating Systems

Communication
Motivation

System Software Group
University of Trier
D-54286 Trier, Germany
Contact: Peter Sturm
sturm@uni-trier.de

Synchronization and Communication

- Interprocess Communcation (IPC) typically comprises synchronization as well as communication issues
  - Synchronization
    - A waits until a specific condition holds
    - B fulfills this specific condition and "informs" A
    - Only a single bit is the information entity
  - Communication
    - True exchange of data
- Various communication concepts exist



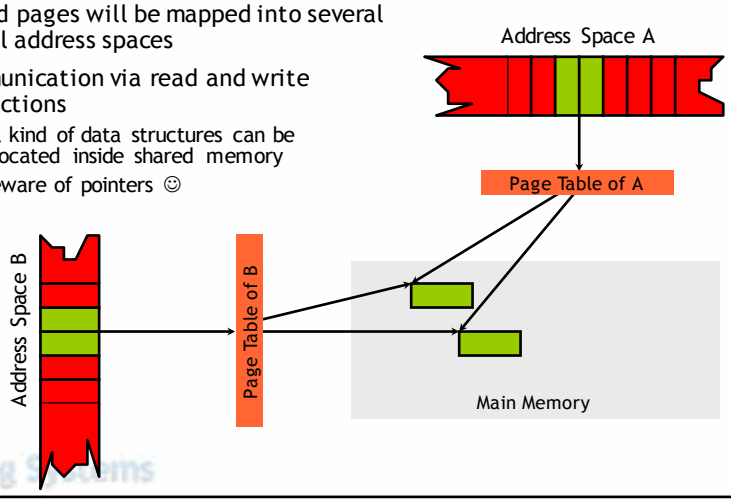
Operating Systems

3 Classes of Communication

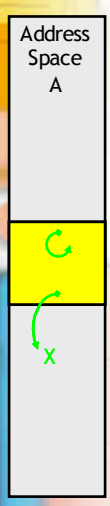
- Within a single process
  - A threads inside a process share the complete address space
  - Communication through any data structures provided by the programming language
- Between processes on a single machine
  - Explicit use of memory portions shared between different address spaces
  - Fast but complicated way to communicate
- Between processes on different machines
  - Distributed systems
  - Message passing is the primary form of communication
  - Some people prefer to use various kinds of distributed shared memory

Sharing Memory Between Processes

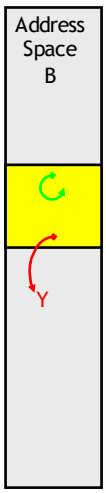
- Multiple processes share a given number of pages explicitly
- Shared pages will be mapped into several virtual address spaces
- Communication via read and write instructions
  - All kind of data structures can be allocated inside shared memory
  - Beware of pointers ☺



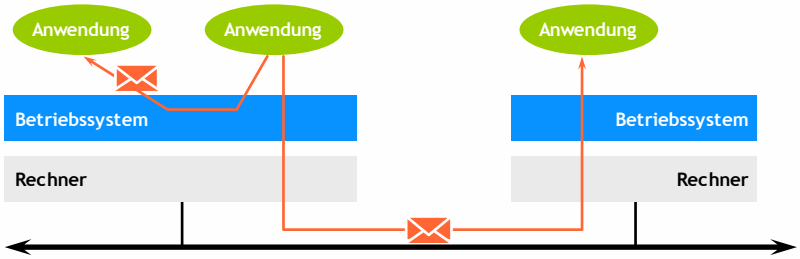
Beware of Pointers!



- Dangerous to use pointers inside shared memory
- Pointers within shared memory itself are okay if mapped to the same address in all processes
  - Therefore, the address to be mapped to can be fixed
- Pointers stored inside but pointing outside of shared memory
  - are deadly since they point to something different in other address spaces



Message-based Communication



- Message = Sequence of bytes
- 2 fundamental operations
  - send
  - receive
- For many message-based techniques, it is easy to cross computer boundaries

Operating Systems

Comparison

Memory-based Communication

- Access schared memory via read and write operations
- Pro
  - Performance
  - Transparency (for some)
- Contra
  - Shared memory needed (requires monoprocessor or multiprocessor)
  - Transparency (for others)
  - You don't see when you communicate
  - Need for synchronization to preserve data consistency

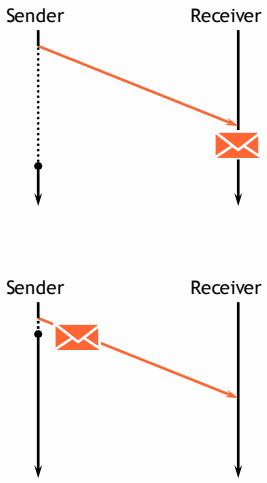
Message-based Communication

- Send and receive messages
- Pro
  - More general concept (applicable to monoprocessors, multiprocessors, and even distributed systems)
  - No transparency (for some)
- Contra
  - Efficiency
    - In case sender and receiver are on the same host
    - Networks are slow compared to memory
  - No transparency (for others)

Operating Systems

Synchronicity

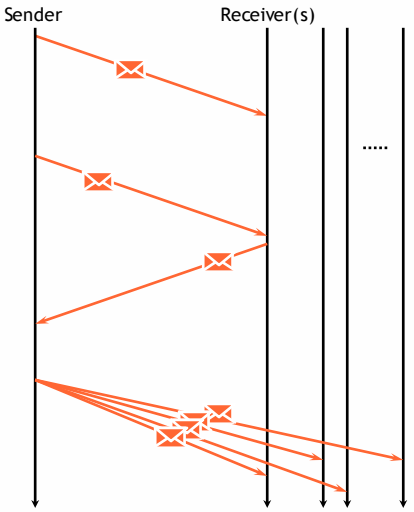
- Synchronous Communication
  - Sender is blocked until message received
  - Less buffer space required
  - Limits concurrency
  - Also called "blocking send"
- Asynchronous Communication
  - Sender only blocked until message has been copied
  - Sender may issue many messages in short sequence (risk of congestion)
  - Buffering of messages required
  - Also called "Non-blocking Send"



Operating Systems

Communication Pattern

- Smallest unit of communication
  - Notification
    - Single message from sender to receiver
  - Service Invocation
    - Client request to server
    - Reply back to sender
- Additional patterns (mostly used in distributed systems)
  - Multicast
    - Send a message to a group of receivers
  - Broadcast
    - Send a message to any potential receiver on the network



Operating Systems

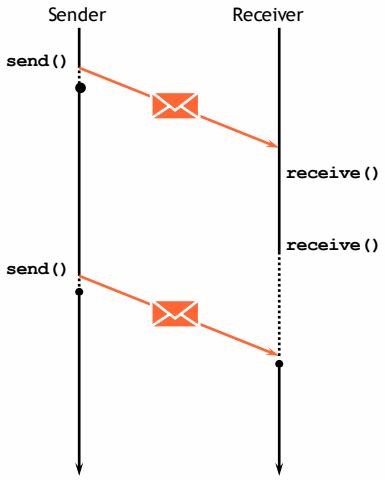
4 Basic Ways to Communicate

	Asynchronous	Synchronous
Notification	Datagram	Rendezvous
Service Invocation	Asynchronous Service Invocation	Synchronous Service Invocation

Operating Systems

Datagram

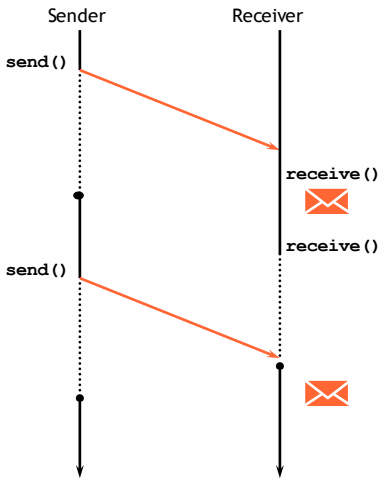
- Sender and receiver are decoupled
  - Concurrency
- Operating system has to buffer messages
  - Complex (overflow, ...)
  - Sometimes still need to block sender in case of resource limitations
- Examples
  - UDP (User Datagramm Protocol)
    - Part of TCP/IP
    - Best Effort
    - max. 64 kbyte data
  - Signals
    - Software interrupts in UNIX
    - No data or a maximum of 4 Byte
- Sender knows ...
  - Nothing



Operating Systems

Rendezvous

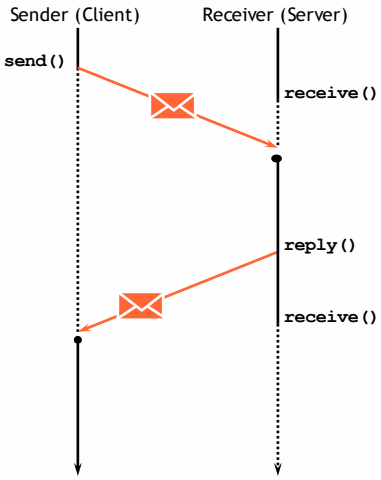
- Sender and receiver are coupled for the duration of message transfer
  - Limited concurrency
- No buffers required
  - Direct copy of data from sender to receiver
  - Very attractive when crossing address space boundaries
  - Simple to implement
- Examples
  - Ada, QNX, ...
- Sender knows ...
  - that message arrived safely



Operating Systems

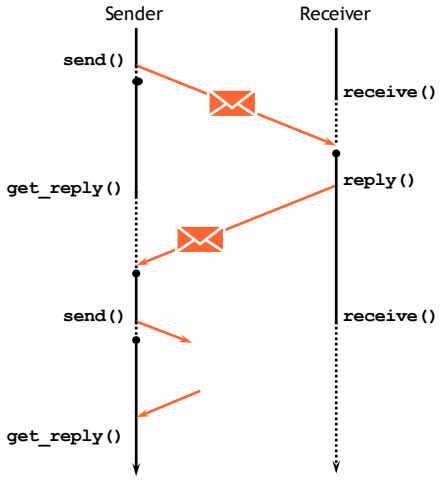
Synchronous Service Invocation

- Even less concurrency
- Bi-directional communication possible
- Example
  - Remote Procedure Call (RPC)
    - DCE, Corba, COM+, Java RMI, ...
  - QNX
- Sender knows ...
  - Message has been received
  - Request has been processed
  - The result

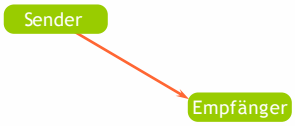


Asynchronous Service Invocation

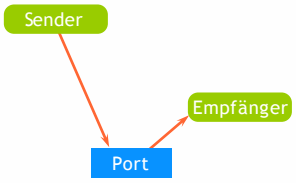
- More concurrency again
- Sometimes called "asynchronous RPC"
- Examples
  - V-Kernel
- Sender knows ...
  - Message has been received
  - Request has been processed
  - The result



Direct and Indirect Communication



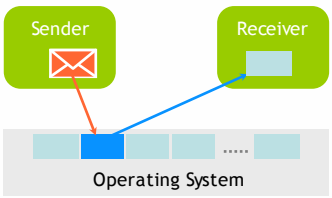
- Direct
  - The receiver is addressed, e.g. process id
- Pro
  - Efficiency
- Contra
  - Changing the receiver can be complicated



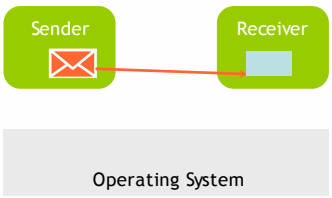
- Indirect
  - The receiver is addressed via port or mailbox
- Pro
  - Flexibility
    - Change of receiver
    - Multiple receivers
- Contra
  - Overhead induced by additional indirection

Operating Systems

Messages Across Address Spaces



- Asynchronous
- 2 copies
  - From sender to buffer
  - From buffer to receiver
- Problems
  - Buffer management
  - Overflow handling



- Synchronous
- Sender and receiver are blocked
  - Direct copy from sender to receiver possible
  - 1 copy
- Problems
  - Limited concurrency
    - Solved with multiple threads

Operating Systems

Beispiele

- Kommunikation zwischen zwei Adreßräumen
  - Signale
    - Asynchron, Mitteilung, Direkt, Paket
  - Pipes und Named Pipes
    - Synchron/Asynchron, Mitteilung, Direkt/Indirekt, Strom
  - Message Queues
    - Synchron/Asynchron, Mitteilung, Indirekt, Nachricht
- Client/Server-Modell
  - Remote Procedure Call (RPC)
    - Synchron/Asynchron, Auftrag, Direkt/Indirekt, Paket
  - RPC auf einem Rechner = Local Procedure Call (LPC)

Operating Systems

Signale

- Übermittlung asynchroner Ereignisse zwischen Prozessen (keine zusätzliche Übertragung von Daten möglich)
- Ursprung
  - Software-Interrupts in Fehlerfällen und für "Job Control"
    - Arithmetikfehler
    - Adreß- und Busfehler
    - Unerlaubter Zugriff
    - Timer
    - ...
- Reaktion auf Signal
  - Blockieren: Zeitliche Empfangsbeschränkung (vgl. Disable Interrupts)
  - Ignorieren
  - Verarbeiten: Anmeldung eines Signal Handlers
  - Prozeß terminieren

Operating Systems

Beispiel

- Prozeß gibt Zustandsinformationen bei Erhalt eines Signals aus

```
int signal_arrived  = 0;

void signal_handler  ( int signo ) {
    signal_arrived  = 1;
    }

int main () {
    signal(SIGUSR1,signal_handler);
    while (1) {
        // Tue etwas sinnvolles  ...
        if (signal_arrived)  {
            // Reagiere auf Signal ...
            signal_arrived  = 0;
            }
        }
    }
```

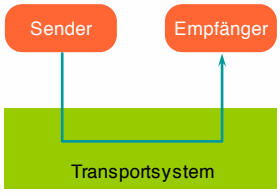
- Absenden des Signals, z.B.

```
kill -USR1 pid
```


Pipes

- “Pipes” und “Named Pipes” (auch FIFO genannt) für Prozeßkommunikation auf einem Rechner
- Pipes zwischen Vater- und Kindprozessen:
  - Verbindung über vererbte Datei-Deskriptoren
- Beispiel

```
main () {
    int pipe_ends[2];

pipe(pipe_ends);
if (fork() != 0) {
    // Vaterprozeß
    write(pipe_ends[1],Daten,...);
    }
else {
    // Kindprozeß
    read(pipe_ends[0],Daten,...);
    }
}
```



Exkurs: UNIX-Shell und Pipes

- Exemplarische Realisierung von: “a | b”

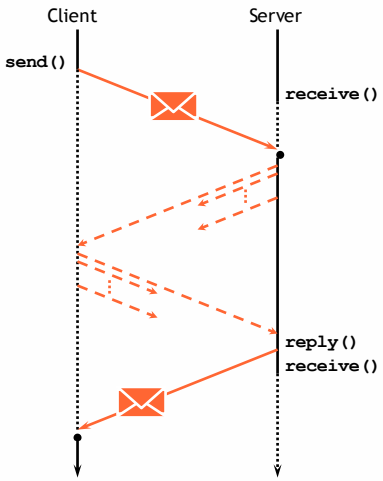
```
...
int pfd[2];

pipe(pfd);
if (fork()==0) {
    // Kind - Wird Prozeß a
    dup(pfd[1],1);  /* stdout auf Pipe-Eingang */
    execve(“a”,...);
    }

if (fork()==0) {
    // Kind - Wird Prozeß b
    dup(pfd[0],0),  /* stdin auf Pipe/Ausgang */
    execve(“b”,...);

if (Vordergrund-Bearbeitung) {
    wait(...);
    }

... zurück zur Eingabeschleife
```


Realisierung

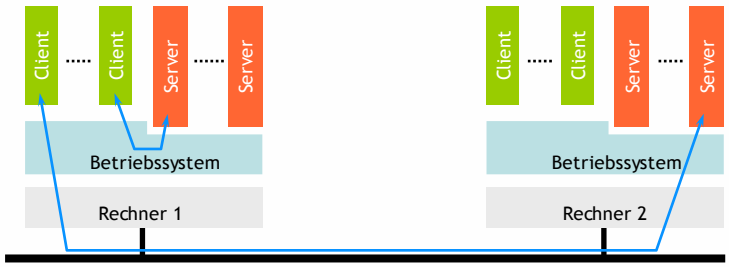
- Klassisches Erzeuger/Verbraucher-System
  - Puffer bestimmter Länge für jede Pipe
  - Empfänger wird blockiert, wenn keine Daten vorliegen
  - Sender wird blockiert, solange Puffer voll ist
- Synchrones Verfahren bei zu kleinem Puffer
  - Gefahr der Verklemmung?
- “Named Pipes”
  - Pipe-Deskriptoren nur über fork vererbbar
  - Pipe-Namen im Dateisystem aufbauen, damit unabhängige Prozesse miteinander kommunizieren können
  - Kommunizierte Daten werden nicht über Dateisystem geleitet!

Client/Server Systems

- Based on synchronous service invocation
- Forms the platform for most distributed systems today
  - Worldwide availability of servers
  - Clients send requests
  - RPC dominant communication technique
  - Addresses open and heterogeneous systems
- But also a possible architecture for modern operating systems
  - Mikrokernel with minimal functionality
  - Lifting of operating system functionality into user-mode servers
  - Basis for distributed operating systems
  - Efficient communication among address spaces required



Client/Server Architecture

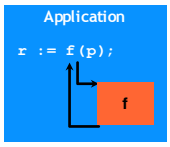


- Lokal servers
  - File system
  - Input and Output
    - Graphics subsystem
  - Process management
  - Paging
  - ...
- Remote servers
  - Network file systems (e.g. shares or NFS)
  - Distributed file systems
  - Rlogin, Telnet, ...
  - WWW server (IIS, apache, ...)
  - Clock synchronization (ntp)

Remote Procedure Call

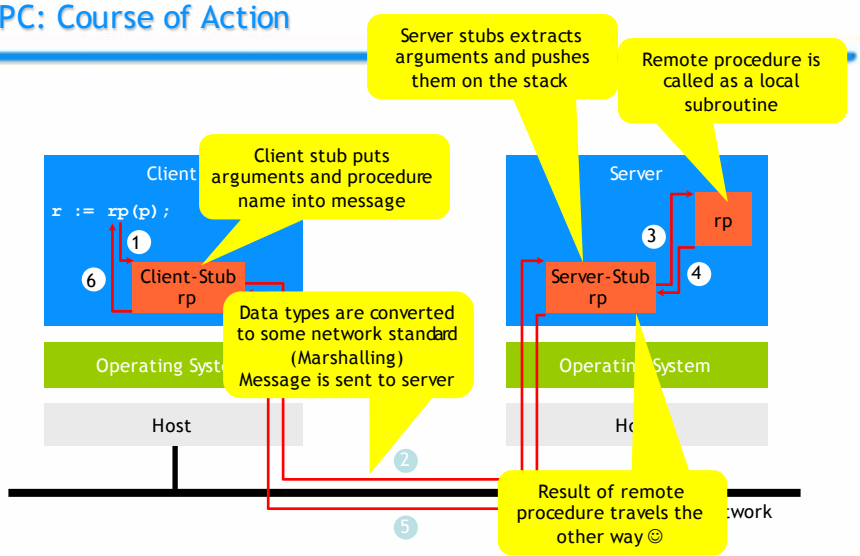
- Tries to resemble semantics of local procedure call
  - Push arguments on stack
  - Jump subroutine
    - Push return address on stack
  - Execute procedure
  - Store result in well-known register
  - Return from procedure
- Local procedure are part of address space of caller
- Now, procedure may be remote
  - In some other address space
  - Maybe on another machine

Local Procedure Call



Operating Systems

RPC: Course of Action



Operating Systems

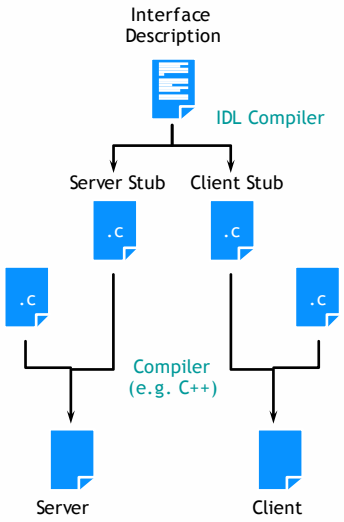
Marshalling

- Imported for distributed systems
  - Encoding of basic data types may differ
    - Little endian vs. big endian
    - …
- Define a generic network standard
  - Sender converts data from local representation to network standard
  - Receiver converts data from network standard to local representation
- Not needed if we just want to cross address spaces not hosts

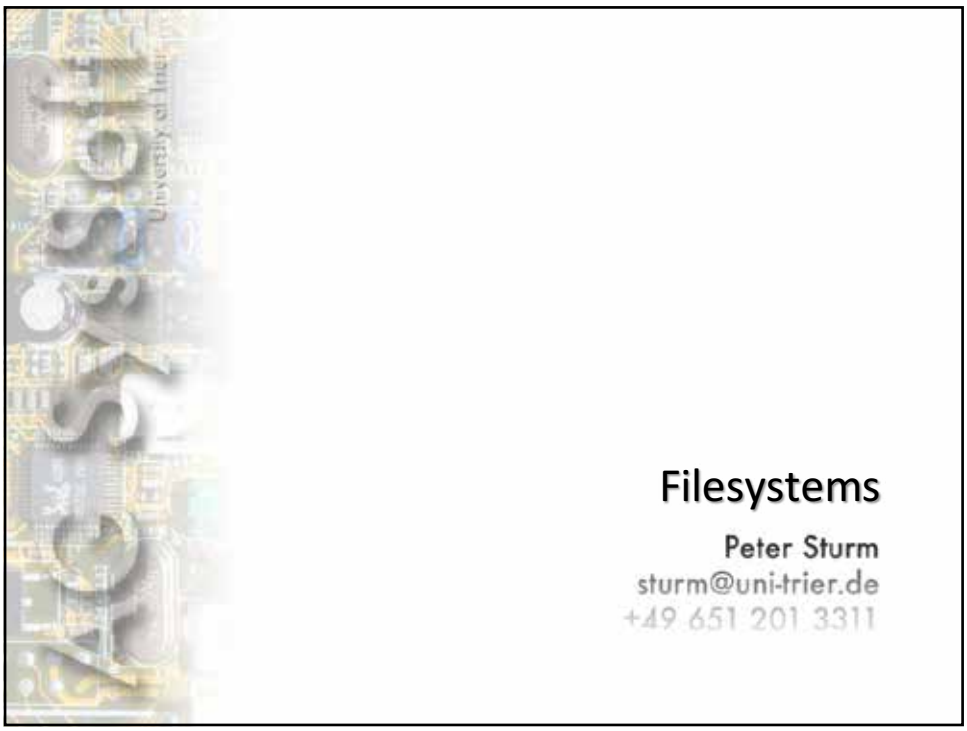
Operating Systems

RPC Compiler

- Server
  - Defines a set of remote procedures
- Service or interface description
  - Define signature for each method
    - Type and sequence of arguments
    - Name of procedure
    - Return type
- Interface definition language (IDL)
- IDL compiler automates most functionality to call a remote procedure
  - Client stubs
  - Server stubs
  - Additional server functionality

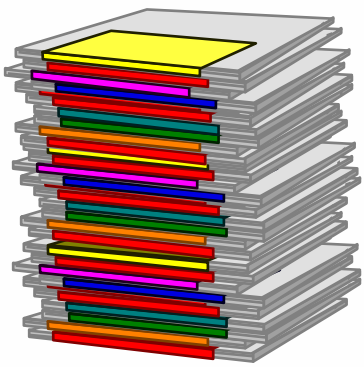


Operating Systems



File Systems

- Most functionality of OS invisible for the user
  - Virtual address spaces
  - Threads
  - Synchronization
  - Communication
  - ...
- File system is the most visibile OS component
  - Stores ALL persistent data
  - Data loss can be bothersome or disastrous



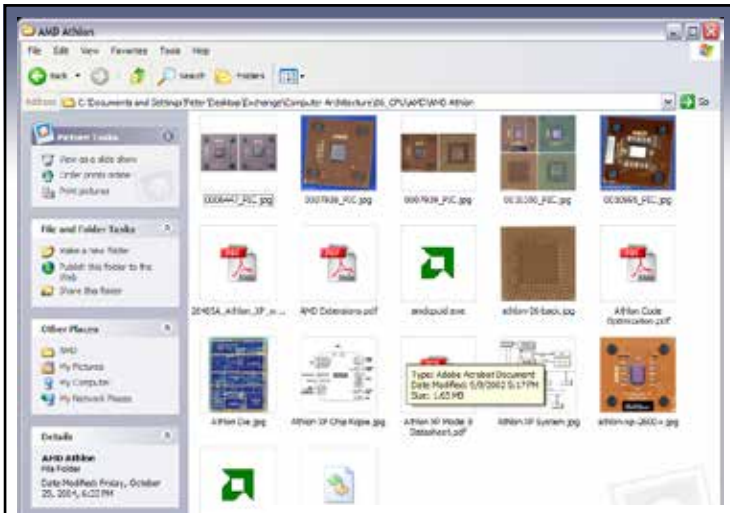
Basics

- Persistent storage of programs and data
- Basic abstraction = File
  - Sequence of data with a name
- File attributes (meta data)
  - Name
  - Type (in some file systems)
  - Physical position
  - Size
  - Access permissions
  - Time stamps
  - owner
- Directories
  - Hierarchical organization of files

Typical Usage of File

- Characteristics (identified empirically)
  - Most files are small (some KB) or very large (audio, video)
  - Reads are frequent, writes are less frequent
  - (Files are deleted rarely or never)
  - Sequential file access is dominant
  - Some files are shared by many users
  - Most files are not shared at all
- Characteristics determine functionality and structure of an efficient file system
  - Make the common case fast
  - Sequential read access for (small) files
  - Utilize caching
    - Frequent reading, 1 user, no sharing

File

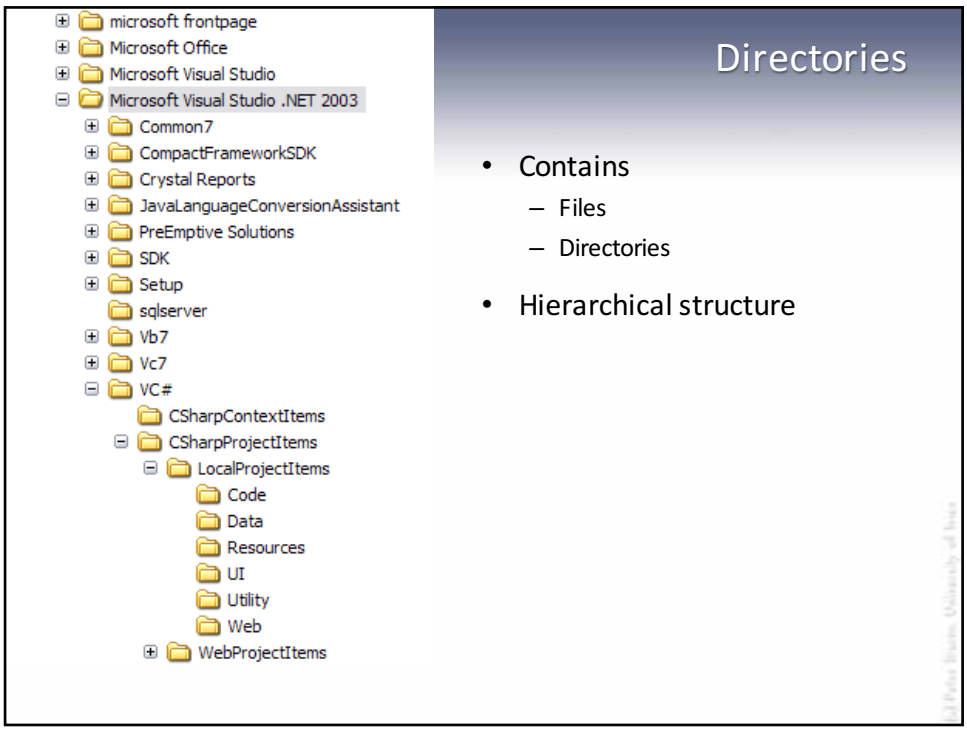


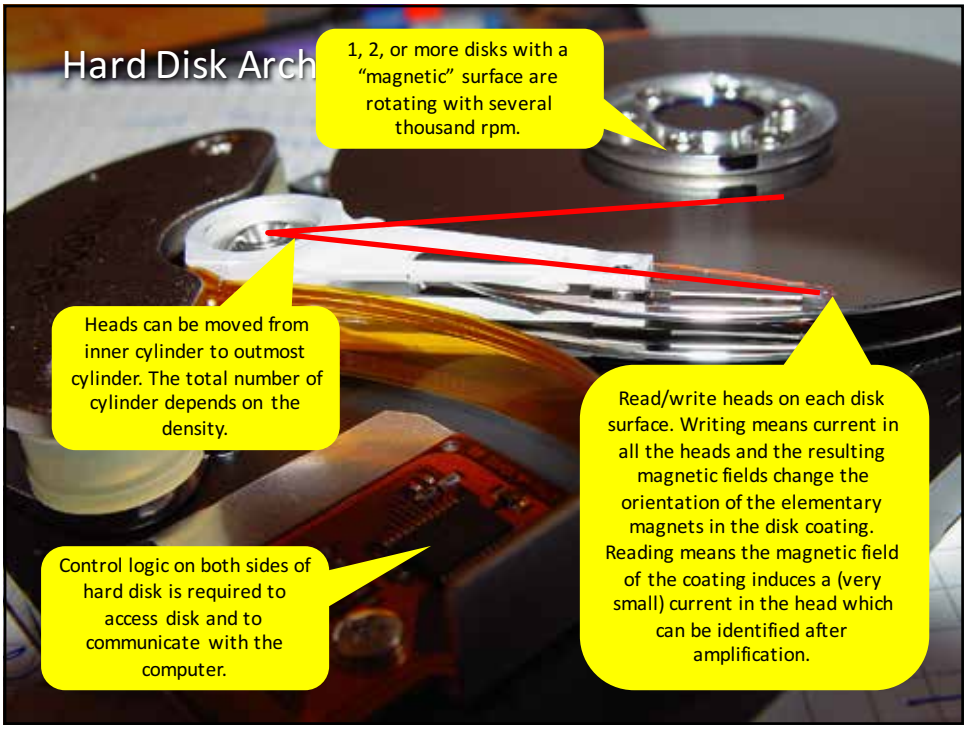
- Fundamental abstraction
- File = Stream of byte
- Structured file types available in some file systems

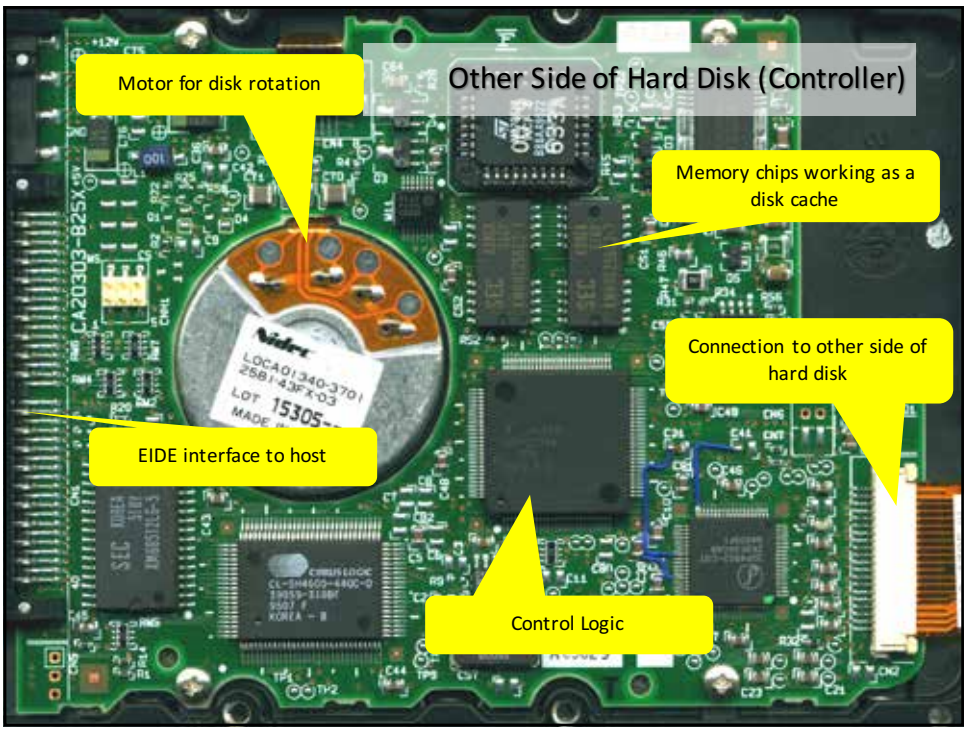
ORWC

```
int main ( int ac, char **av ) {
     if (ac != 2) {
          fprintf(stderr,"usage: %s <filename>\n",av[0]);
          exit(-1);
     }
     char *filename = av[1];

     int fd = open(filename,O_RDONLY);
     assert(fd != -1);

     const int bufsize = 4096;
     char *buffer = malloc(bufsize);
     ssize_t readsize;
     do {
          readsize = read(fd,buffer,bufsize);
          ssize_t writesize = readsize;
          while (writesize > 0)
               writesize -= write(STDOUT_FILENO,buffer,writesize);
     } while (readsize > 0);
     close(fd);
}
```

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <assert.h>
```


microsoft frontpage
Microsoft Office
Microsoft Visual Studio
Microsoft Visual Studio .NET 2003
Common7
CompactFrameworkSDK
Crystal Reports
JavaLanguageConversionAssistant
PreEmptive Solutions
SDK
Setup
sqlserver
Vb7
Vc7
VC#
CSharpContextItems
CSharpProjectItems
LocalProjectItems
Code
Data
Resources
UI
Utility
Web
WebProjectItems
Directories
• Contains
– Files
– Directories
• Hierarchical structure


Hard Disk Arch
1, 2, or more disks with a "magnetic" surface are rotating with several thousand rpm.
Heads can be moved from inner cylinder to outmost cylinder. The total number of cylinder depends on the density.
Control logic on both sides of hard disk is required to access disk and to communicate with the computer.
Read/write heads on each disk surface. Writing means current in all the heads and the resulting magnetic fields change the orientation of the elementary magnets in the disk coating. Reading means the magnetic field of the coating induces a (very small) current in the head which can be identified after amplification.



Comparison: Main Memory and Disk Memory (of July 2005)

Internal Memory
- Volatile
- Byte and word based access
- Quite fast (≤10ns)
- Several GB/s throughput

- Expensive
  - 1 GB = 110 Euro
  - 1 MB = 11 Cent
- Common capacities
  - 512 MB .. 4 GB on PCs

External Memory
- Persistent
- Block based access (512 Byte … 8 KByte)
- Very slow (4-8 ms)
- 50 MB/s and more

- Cheap
  - 200 GB = 100 Dollar
  - 1 MB ca. 0.05 Cent
- Common capacities
  - 160 GB .. 400 GB and more

These values change very fast, so they may differ substantially from your findings.

Comparison: Main Memory and Disk Memory (of January 2009)

Internal Memory
- Volatile
- Byte and word based access
- Quite fast (≤10ns)
- Many GB/s throughput

- Expensive
  - 2 GB = 30 Euro
  - 1 MB = 1.46 Cent
- Common capacities
  - 1 GB .. 16 GB on PCs

External Memory
- Persistent
- Block based access (512 Byte … 8 KByte)
- Very slow (4-8 ms)
- 100 MB/s and more

- Cheap
  - 1000 GB = 90 Euro
  - 1 MB ca. 0.009 Cent
- Common capacities
  - 200 GB .. 1500 GB and more

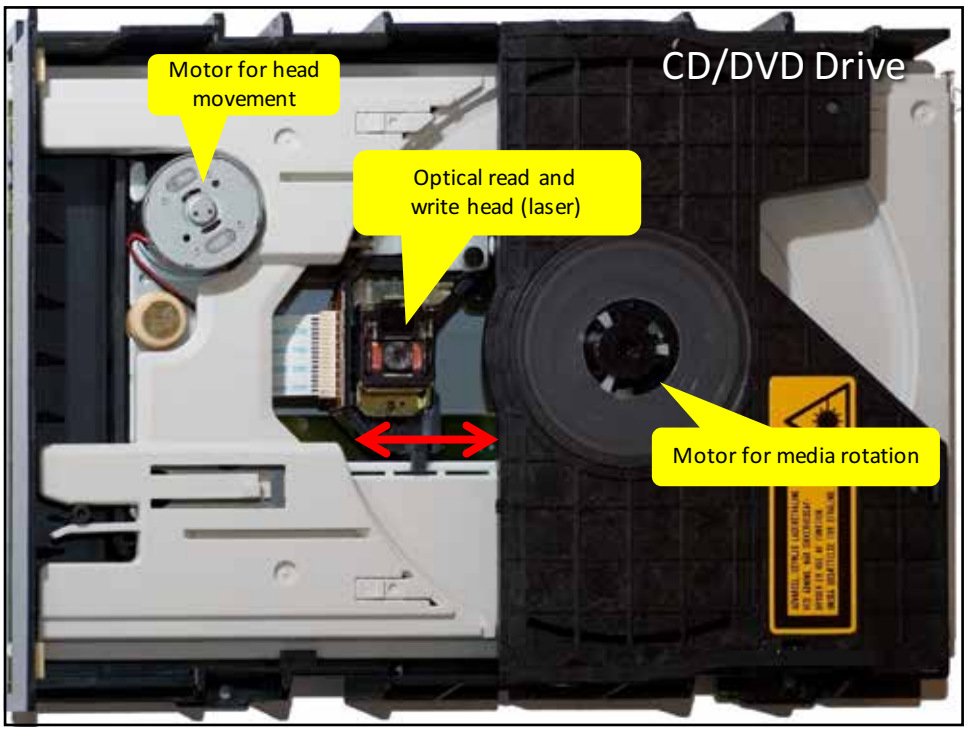
Comparison: Main Memory and Disk Memory (of January 2012)

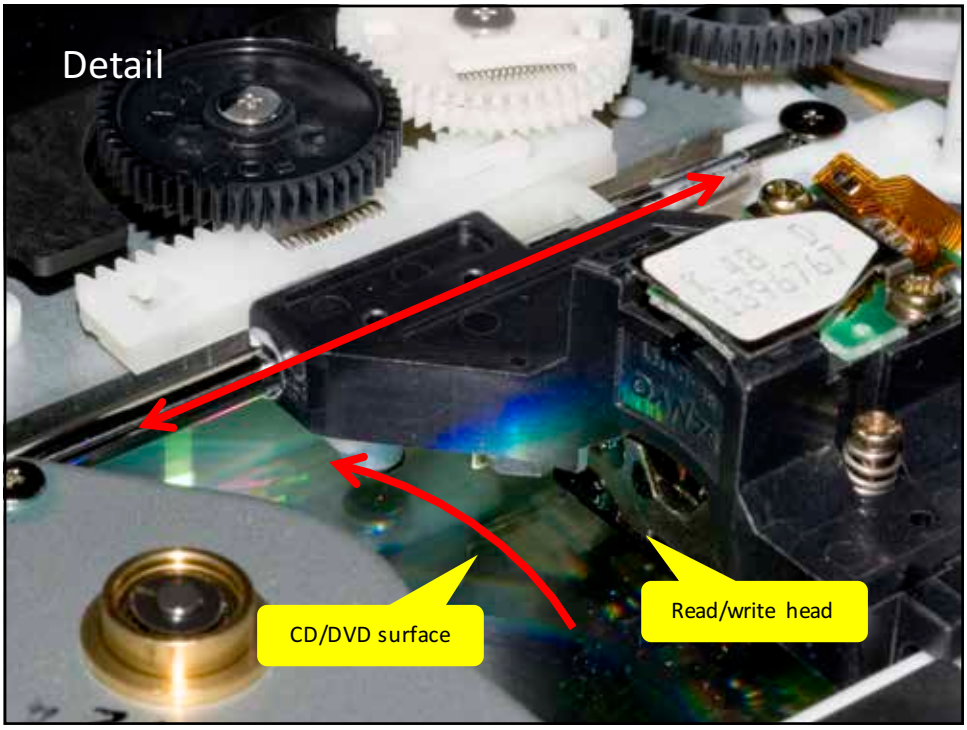
Internal Memory
- Volatile
- Byte and word based access
- Quite fast (≤10ns)
- Many GB/s throughput

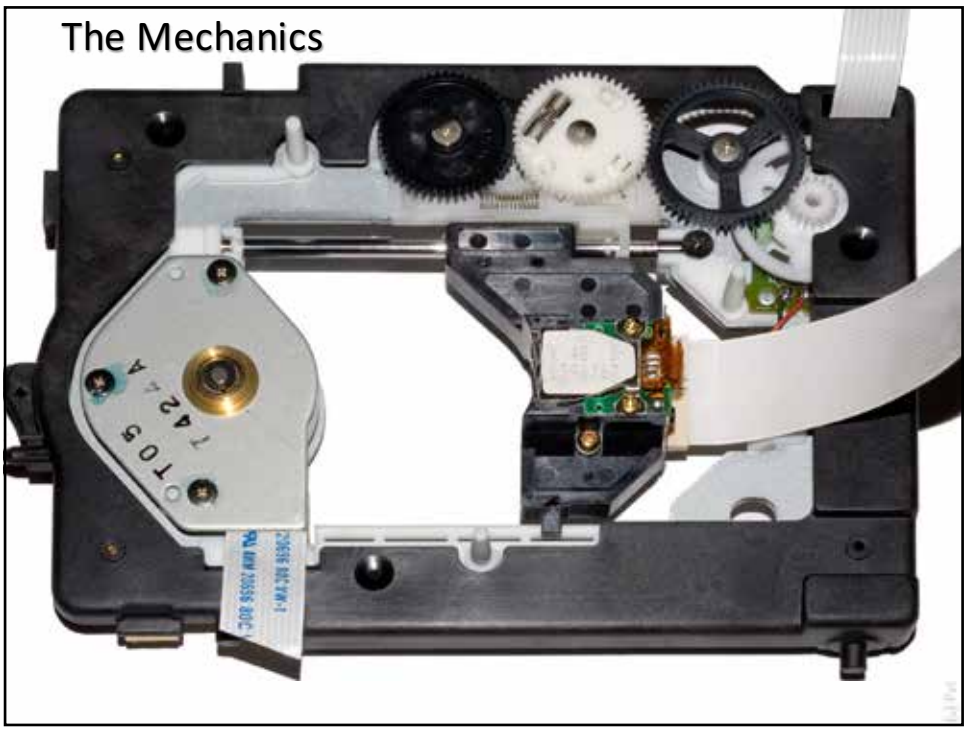
- Expensive
  - 4 GB = 20 Euro
  - 1 MB = 0.48 Cent
- Common capacities
  - 4 GB .. 16 GB on PCs

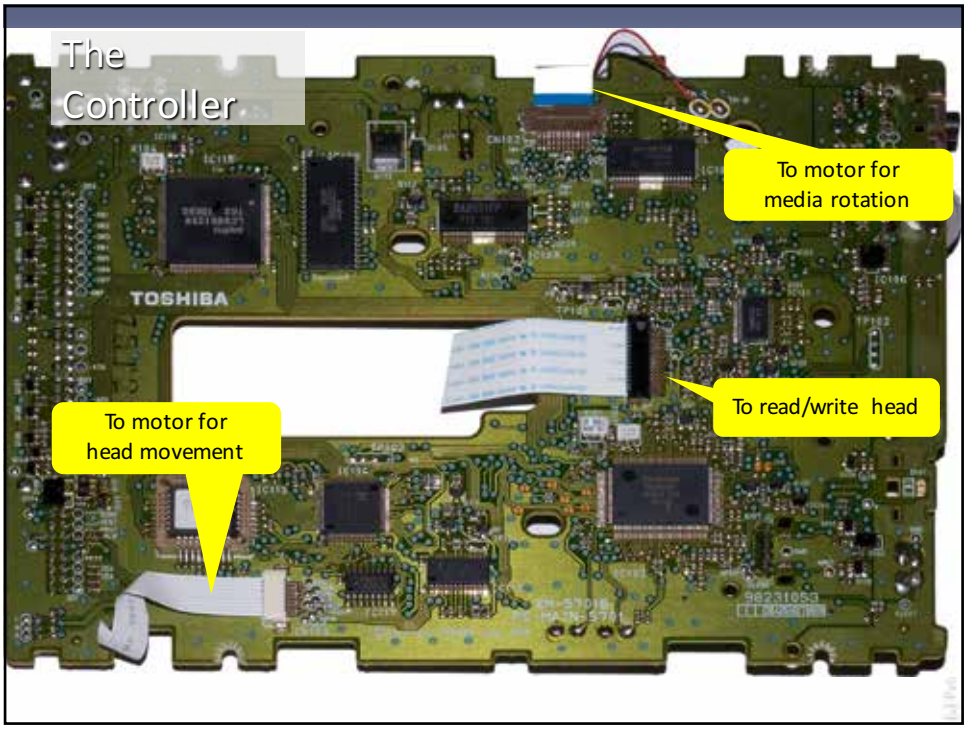
External Memory
- Persistent
- Block based access (512 Byte … 8 KByte)
- Very slow (4-8 ms)
- 100 MB/s and more

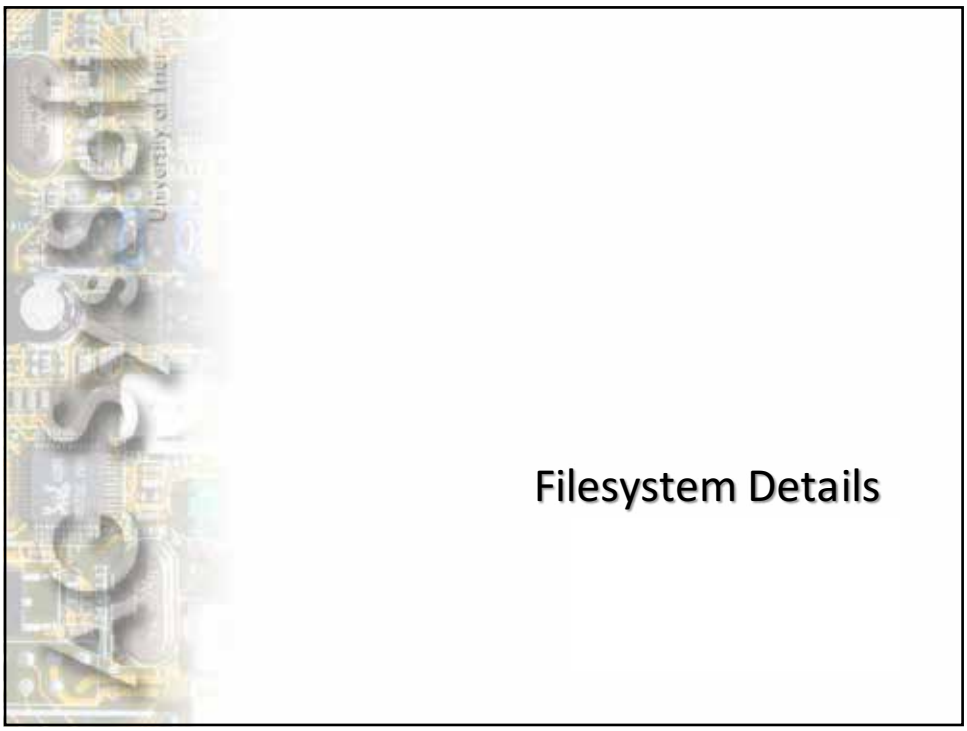
- Cheap
  - 2000 GB = 120 Euro
  - 1 MB ca. 0.005 Cent
- Common capacities
  - 500 GB .. 3000 GB and more


CD/DVD Drive
Motor for head movement
Optical read and write head (laser)
Motor for media rotation


Detail
CD/DVD surface
Read/write head


The Mechanics


The Controller
To motor for media rotation
To read/write head
To motor for head movement
TOSHIBA



Introduction

- Persistent storage of applications, user files, configuration data, operating system
- Basic abstraction
  - File = Sequence of bytes
- Charateristics of hard drives require block-based access
  - Head movement limitating factor
  - Typical size is 4 kbytes
- Make the common case fast
  - Reading files sequentially
  - Read ahead, Caching

Requirements for Modern Filesystems

- Minimize time-consuming head movement
  - Store file data and all accompanying meta-data in close neighborhood
  - Collect mutiple disk requests and optimize movement
  - Cluster blocks consecutive blocks of a file phyiscally
- Improve read/write performance
  - Utilize caching agressively (within hard disk itself and operating system)
  - Exploit striping and mirroring to improve performance and/or fault tolerance
- Keep pace with increasing disk capacity
  - "Since disk usage is like an ideal gas, which takes up every available space", file systems must deal with more and more files
  - Recovery in case of system failure must be fast

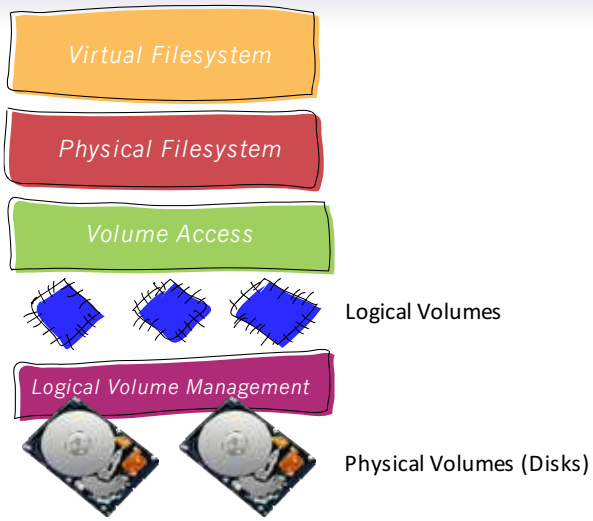
Functions

- Formatting and initializing media
  - Create initial content to access media
- Keeping track of bad blocks
  - Recognize bad blocks and exclude them from future use
- Managing of meta-data
  - Small set of data but high update frequency
- Storing and retrieval of files
  - Sequential reading and writing
  - Positioning of file pointer
  - Mapping files into virtual address space
- Security
  - Who's allowed to access a file?

Meta Data

- File information (inode)
  - Size, owner, access rights, timestamps
  - Where are the blocks on disk
- Directory information
  - Content, owner, access rights, timestamps
- Device information
  - root directory
  - Free blocks
  - Bad blocks

Hierarchical Structure



Classes of filesystems

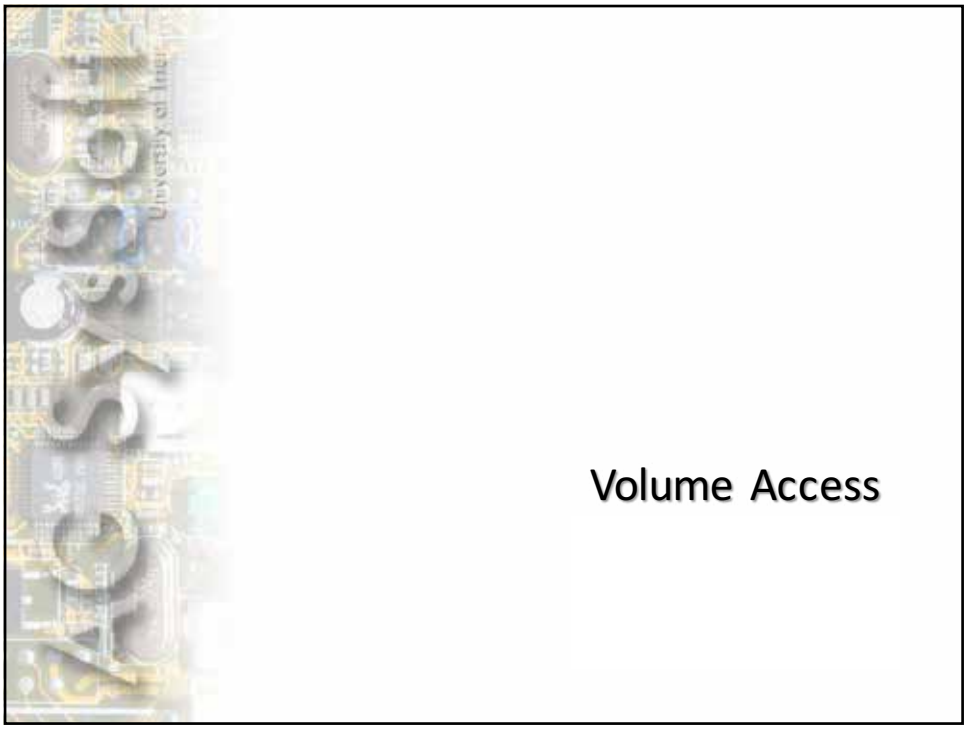
- Media characteristics
  - Read/Write
    - Hard disk filesystems (most common)
    - FATx, NTFS, HFS, ext2, ext3, reiserfs, XFS, ZFS, …
  - Limited write
    - Filesystems for flash media
  - Readonly
    - CD/DVD
    - ISO9660, UDF, …
- Networking capabilities

In Place vs. Log

- Sometimes high update frequency
  - e.g. every access changes some timestamps
- Small updates
  - 4 bytes timestamp
- Cache changes might lead to inconsistencies
- How to update
  - In place
  - Log-based

Classification

Meta Data \ Data	Update in Place	Log
Update in Place	Traditional filesystems ext2, FAT, …	unknown
Log	Journaling filesystems ext4, NTFS, HFS, reiserfs, XFS, …	Log-based filesystems ZFS, btrfs, …

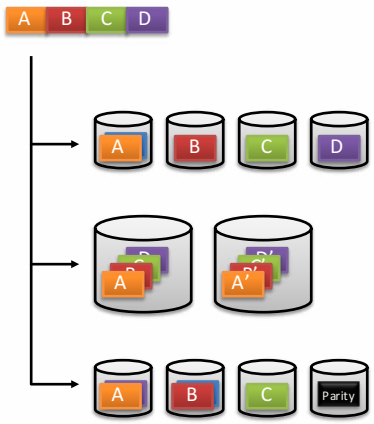


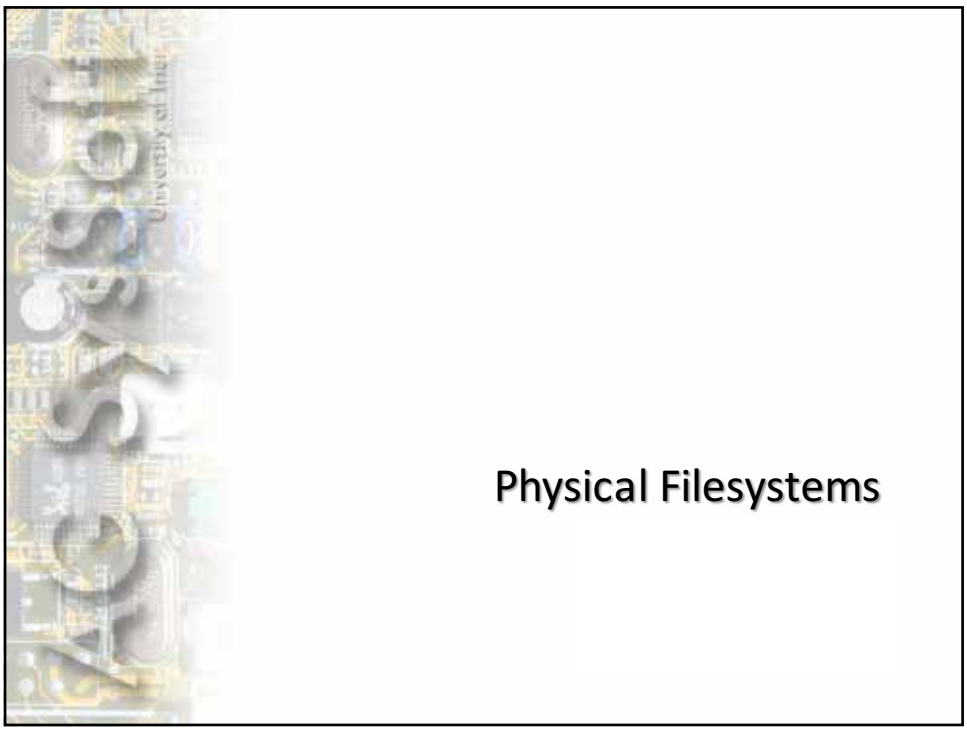
Basic access

- Defines unique address for each block
  - Outdated is Cylinder/Head/Sector (CHS)
  - Logical block address (LBA)
- Size of address determines maximum size of filesystem
  - Huge numbers for today's disks
- Basic functionality
  - Access block by address
- Maintaining bad block and free block lists
  - Typically stored in bitlists
- Implemented by BIOS and hardware

Redundant Arrays of Independent Disks (RAID)

- 6 Levels
  - RAID 0: Striped
  - RAID 1: Mirrored
  - RAID 2: Striped+ ECC
  - RAID 3: Striped+ Parity
  - RAID 4: Huge stripes + Parity
  - RAID 5: Huge stripes + Rotating Parity
  - RAID 6: RAID 5 + additional parity
- Most common are RAID 0,1,5 and combinations (RAID 10 = 1 + 0)
- Favor huge writes
  - Overhead 1/(n-1) for ndisks
- Small writes are expensive
  - Read data
  - Write data
  - Read parity
  - Write parity





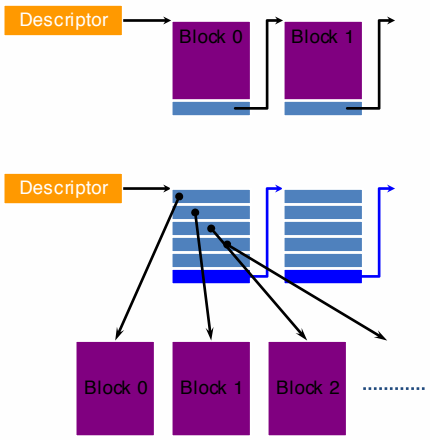
File Organization

- Contiguous allocation (clustering)
- Advantages
  - Fast sequential and random access
  - Easy to manage
- Disadvantages
  - External Fragmentation
  - Shrinking of files creates holes
  - File growth can be expensive
  - Degfragmentation (space)

- Distributed allocation
- Advantages
  - Internal fragmentation only
  - Size change of file simple
- Disadvantages
  - No uniform access time
  - Complex to manage
- Hybrid approaches are favored
  - Clustering as far as possible
  - But distributed allocation always possible
  - Defragmentation (performance)

Distributed Allocation

- Dedicated index structure to keep track of block order
- Internal
  - Index part of data blocks
  - Exotic block sizes (no multiple of 2)
  - Random access expensive
- External
  - Dedicated index blocks
  - Single-linked and double-linked lists
  - Trees, balanced trees
  - Hybrid approaches



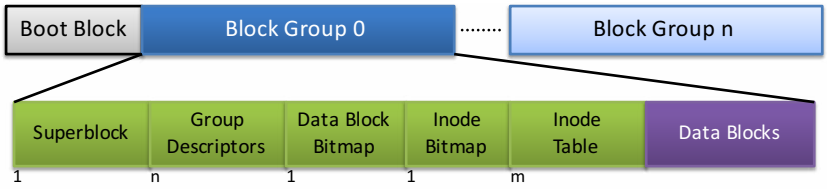
File Allocation Table (FAT)

- Developed by Microsoft for MS-DOS (1977)
- Best supported filesystem
- Disk space is allocated to files in contigous groups (clusters)
  - FAT (FAT12) = 2^{12} clusters ⇒ max. 32 MBytes disk size
  - FAT16 ⇒ 2 GByte
  - FAT32 ⇒ 8 TByte
- Linked list of clusters for bigger files
- Newly created files tend to be fragmented
  - Defragmentation needed

ext2

- The original Linux filesystem
  - Origin was Minix filesystem (just as in case of Linux itself)
  - Extended filesystem (inefficient extension)
  - 2nd Extended filesystem (sice1994)
- Properties
  - Selectable block size between 1024 and 4096 bytes
  - Selectable number of inodes per partition
  - Clustering of blocks
  - Preallocation of file blocks (enlargement)

Structure



- Reason for block groups?
- Maximum size of block group:
  - Data Block Bitmap = 1 Block
  - 8 MB (1 KB blocks) up to 128 MB (4 KB blocks)
- Superblock and group descriptors are replicated in each group
  - Kernel will access block group 0 only

Superblock

- Total number of inodes
- Block size
- Size of overall filesystem in blocks
- Number of reserved blocks
  - Only accessible by root processes
  - Accessing an otherwise full filesystem
- Number of free blocks and free inodes
- Characteristics of block groups
- Statistics
  - Timestamps (Last mount operation, Last write, Last filesystem check)
  - Mount statistics (Number of mount operations, Maximum mounts before check)
- Status flag
  - 0 mounted or missed unmount, 1 cleanly unmounted, 2 Filesystem has errors

Trends

- Reinvent the file system
  - Apply the state of the art
  - Copy-on-Write, Transactional, Checksums, Ease of use
- Precursor
  - ZFS (Sun Microsystems)
  - Available in Solaris, FreeBSD, (Linux), (Mac OS)
  - Pooling, Software RAID
- Imitators ☺
  - btrfs (GPL, Oracle)
  - ReFS (Microsoft, Windows 8 Server)