UNIVERSITY OF COPENHAGEN

Principles of Computer System Design (PCSD)

Marcos Vaz Salles
Assistant Professor, DIKU

Annual satisfaction and well-being assessment

UCPH is investigating the study environment.

Tell us how we can improve your study environment!

Remember to answer the survey by 3 December.

You will recieve a link to the survey via your UCPH e-mail (KU-mail), read more about the assessment at KUnet.

Why study computer systems?

- The IBM/Microsoft/Oracle question
  - How can I program large systems with clean interfaces and high performance?
- The Amazon/Facebook/Google question
  - How can I understand the guarantees and reliability of scalable services offered to me on the cloud?
- The Cloudera/Greenplum/ Teradata question
  - How do I build systems to process TBs to PBs of data?

What is the scale of our computer systems?

- Source: Michael Brodie, Computer Science 2.0, presented at VLDB 2007, Vienna, Austria

- **Databases**
  - AT&T has 11 exabytes (10^7 TB) of wireline, wireless, Internet data; 2+ trillion calls
  - Google's BigTable (US): 1-2 petabytes,
  - Wal-Mart (US): 500 TB, 10^7 transactions / day
  - All in 2004

What is the scale of our computer systems?

- Source: Michael Brodie, Computer Science 2.0, presented at VLDB 2007, Vienna, Austria
- **Internet**
  - 1/2 billion hosts (IP addresses)
  - 1.17 billion users
  - or 17.8% of the world's population
- **Web**
  - 109 million distinct web sites
  - 29.7 billion web pages
  - ~5 pages for every man, woman, and child on the planet
  - 7.2 billion Web searches/month (3.9 billion by Google) far exceed the world population

What is the scale of our computer systems?

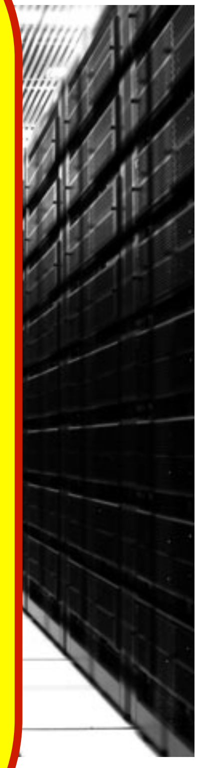
- Source: Michael Brodie, Computer Science 2.0, presented at VLDB 2007, Vienna, Austria

- **Facebook**
  - 1.8 billion photos
  - 41 million active users
  - 10^5 new users / day
  - 1,800 applications

- **YouTube**
  - 1.7 billion served / month
  - 1 million streams / day

What is the scale of our computer systems?

How can we think about and architect large-scale computer systems?



What should we learn in this course?

- Knowledge
  - Describe the design of transactional and distributed systems
  - Explain how to enforce modularity through a client-service abstraction
  - Explain techniques for large-scale data processing
- Skills
  - Implement systems that include mechanisms for modularity, atomicity, and fault tolerance
  - Structure and conduct experiments to evaluate a system's performance
- Competences
  - Discuss design alternatives for a computer system, identifying system properties as well as mechanisms for improving performance
  - Analyze protocols for concurrency control and recovery, as well as for distribution and replication.
  - Apply principles of large-scale data processing to concrete problems.

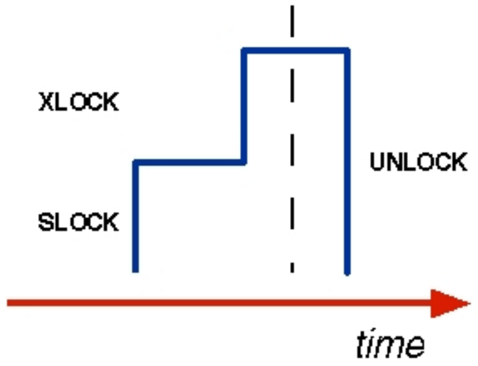
PCSD: What will we study?

- Fundamentals
  - Abstractions: interpreters, memory, communication links
  - Modularity with clients and services, RPC
  - Techniques for performance, e.g., concurrency, fast paths, dallying, batching, speculation

Property: Strong Modularity

PCSD: What will we study?

- Concurrency Control and Recovery
  - Two-phase locking
  - Serializability, schedules
  - Optimistic and multi-version approaches to concurrency control
  - Recovery concepts
  - ARIES recovery algorithm



Properties: Atomicity and Durability

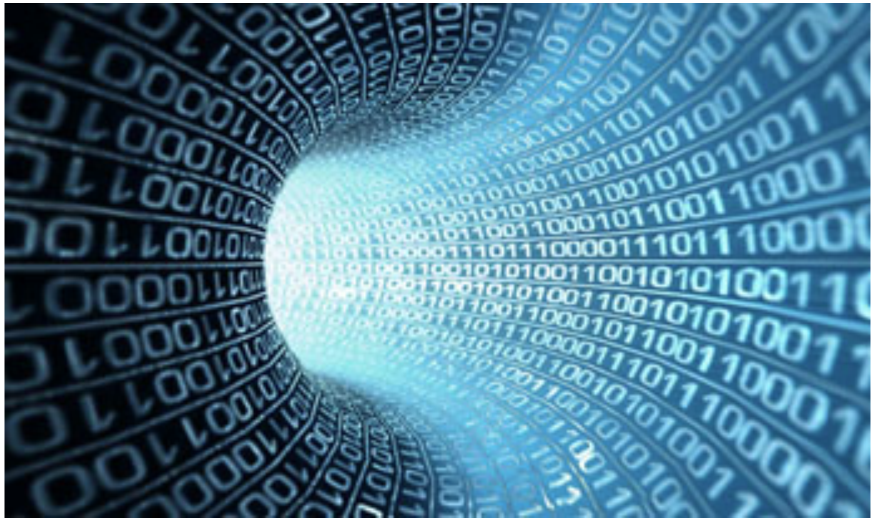
PCSD: What will we study?

- Communication
  - Message queues, streams, multicast, BASE
  - End-to-end argument
- Reliability & Distribution
  - Reliability concepts
  - Replication techniques
  - Topics in coordination and distributed transactions

Property: High Availability

PCSD: What will we study?

- Data Processing
  - Operators
  - External sorting
  - Hash- and sort-based techniques for multiple operations (e.g., set operations, joins)
  - Parallelism



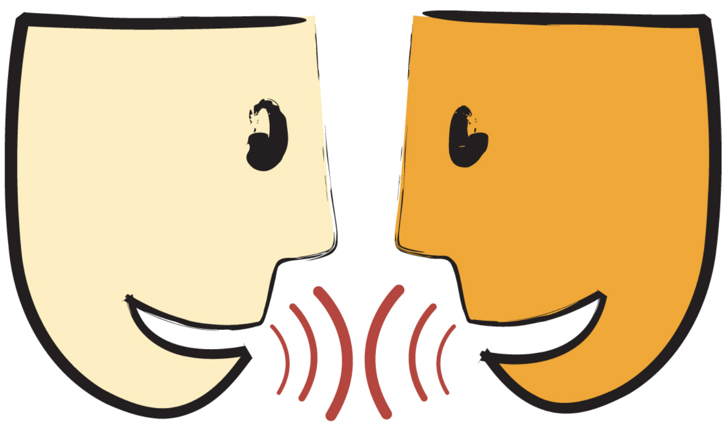
Property: Scalability with Data Size

PCSD: What will we study?

- Experimental Design
  - Performance metrics, workloads
  - Structuring and conducting simple experiments

PCSD: What will we study?

- Guest Lectures
  - **Date: December 16**
  - **Service-Oriented Architecture (SOA) in the Real World**, by Morten Steffensen of **Netcompany**
  - **Recovery in Practice,** by Paz Padilla Thygesen of **IBM**



- **Extra Talk Annoucement**
  - **Date & Time:** December 3, 3pm
  - **Autogenerating data warehouses**, by Kennie Nybo Pontoppidan of **Rehfeld**

References & Course Materials

- Course webpage
  - Kurser: http://kurser.ku.dk/course/ndaa09004u/2014-2015
- Course materials in Absalon
  - Tentative course syllabus
    - Includes readings for after each class
  - Slides before each class
  - Assignments & Feedback
  - Message forums
- Please always post your questions in Absalon
  - Your colleagues can profit too!

References & Course Materials

- Book
  - Principles of Computer System Design (PCSD): DIKU Course Compendium. Collected references from sources cited therein, organized by Marcos Vaz Salles and Michael Kirkedal Carøe.
  - IMPORTANT: Same compendium as 2013/2014!

- Papers & other references
  - Vast majority listed in the syllabus
  - A few more will come as we go
  - Optional references for more depth

Team

- Lecturers
  - Marcos Vaz Salles
    - vmarcos@diku.dk
  - Vivek Shah
    - bonii@diku.dk
  - Office hours:
    - By email appointment

- TAs
  - Vivek Shah
  - Abigail Lowe
  - Håkan Lane
  - Meet them in the TA sessions on Thursdays!

Weekly Schedule

- Lectures Tuesdays and Thursdays, 10am-12pm
  - Two 45 min sessions, 15 min break, with lecturer
  - Participation will be encouraged ☺

- TA sessions Thursdays 1pm-3pm
  - Session length according to need
  - TAs will guide most of those
    - Exercises
    - Assignment work time and Q&A
  - I will come over most Thursdays for the second hour

Learning is the main goal!

First Steps

- **Java Warm-up Exercise**
  - If you passed Advanced Java, you do not need it ☺
  - **BUT FILL THE EVALUATION FOR ADVANCED JAVA! ☺**
  - If you did not take Advanced Java, the warm-up assignment tells you the level of Java you **need** for this course
- **First TA Session**
  - This Thursday, **November 20**
  - **Exercises + Q&A on Assignment 1**
  - Setup of **optional Windows Azure** cloud service
    - Generous gift by Microsoft
    - Allows you to learn while using a cloud service
    - We have been awarded one pass per student
    - Roughly two small instances for 5 months
    - Beware of crackers!

Assignments

- **4 + 1 take-home assignments**
  - **Groups: 2 people** strongly recommended
  - 4/5 must be passed to qualify for exam
  - **Exactly one** resubmission allowed by **January 8**
- **First 4 assignments**
  - Build **specific** skills and concepts on **weekly** basis
  - Include both theory exercises and programming
  - **Due dates: November 25, December 2,
    December 9, December 16**
- **Final 5th assignment**
  - **Integrates** multiple skills and concepts into a single assignment
  - **Exam-style**: Based on a previous year's exam
  - Mostly focused on programming + report
  - **Due date: January 6**

One potential way of succeeding in PCSD

- Course workload: 206 hours over 7+1 weeks
- Normal week: 26 hours / Exam week: 24 hours

Wed	Thu	Fri	Mon	Tue
Assignment out, start working (5h) → Target: work on programming	Lecture (2h) TA session (2h) → Target: warm-up on theory + Q&A and work on programming! Reading (1.5h)	Work on assignment (5h) → Target: tests and code done + first look at theory	Work on assignment (5h) → Target: theory done + write on programming questions / report	Lecture (2h) Reading (1.5h) Review assignment, finish report, hand in (2h) CELEBRATE!

Exam

- **Exam format**
  - 5-day take-home assignment with external grading on 7-point scale, between **January 16** and **January 22**
  - Must be solved individually, no groups allowed
  - Includes both theory exercises and programming
  - Programming similar in structure to Assignment 5
  - Submission in Absalon

Academic Integrity
taken very seriously

Acknowledgements

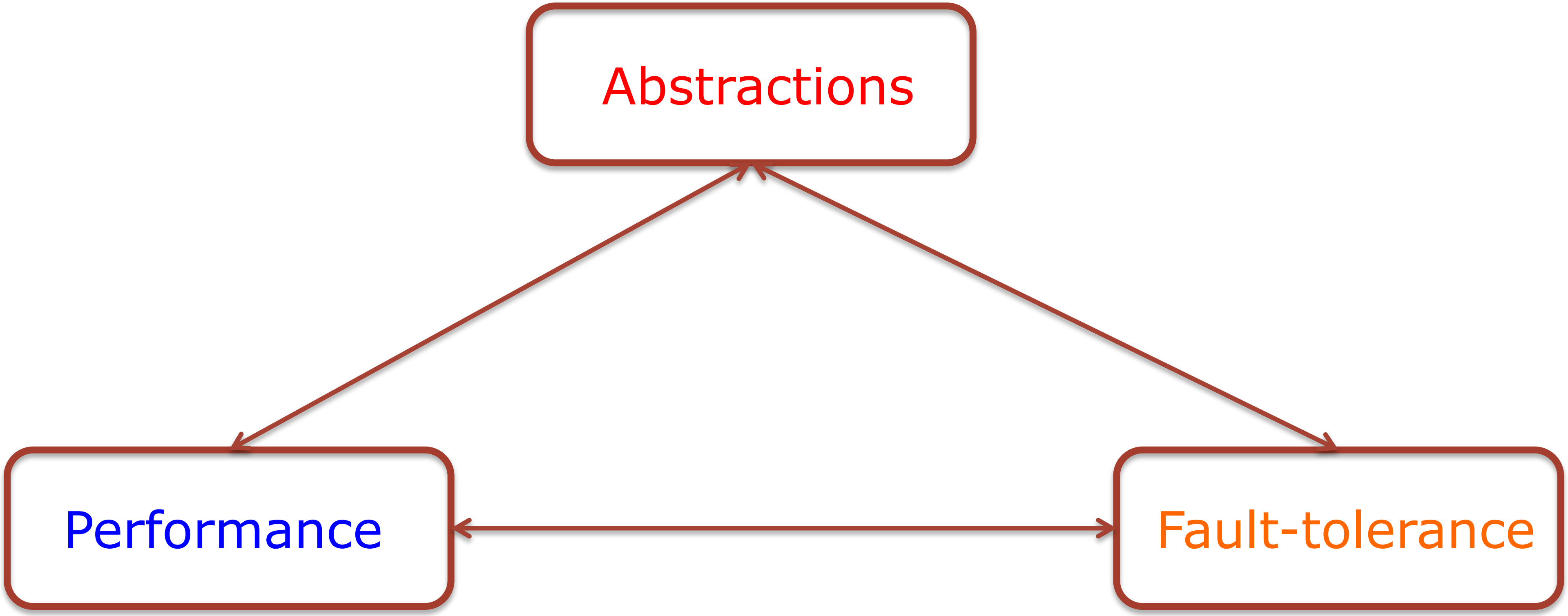
- Many of the slides in this course are based on or reproduce material kindly made available by Jerome Saltzer & M. Frans Kaashoek & Robert Morris (MIT, PCSD textbook material), Johannes Gehrke (Cornell, Ramakrishnan & Gehrke textbook), Gustavo Alonso (ETH Zurich, EAI course), Michael Freedman (Princeton), Nesime Tatbul (ETH Zurich), James Kurose & Keith Ross (U Mass Amherst & NYU, networking textbook), Jens Dittrich (Saarland University)

Questions so far?

What should we learn today?

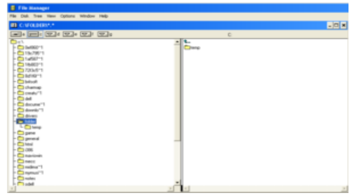
- Identify the fundamental abstractions in computer systems and their APIs, including memory, interpreters, communication links
- Explain how names are used in the fundamental abstractions
- Design a top-level abstraction, respecting its correspondent API, based on lower-level abstractions
- Discuss performance and fault-tolerance aspects of such a design

The Central Trade-off: Abstractions, Performance, Fault-Tolerance



Fundamental abstractions

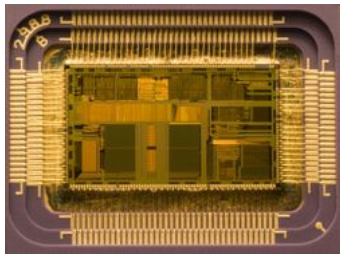
- ***Memory***
  - Read/write

- ***Interpreters***
  - Instruction repertoire
  - Environment
  - Instruction pointer

(loop (print (eval (read))))

Names
make
connections



- ***Communication links***
  - Send/receive

Source: Saltzer & Kaashoek & Morris (partial)

Examples of Names

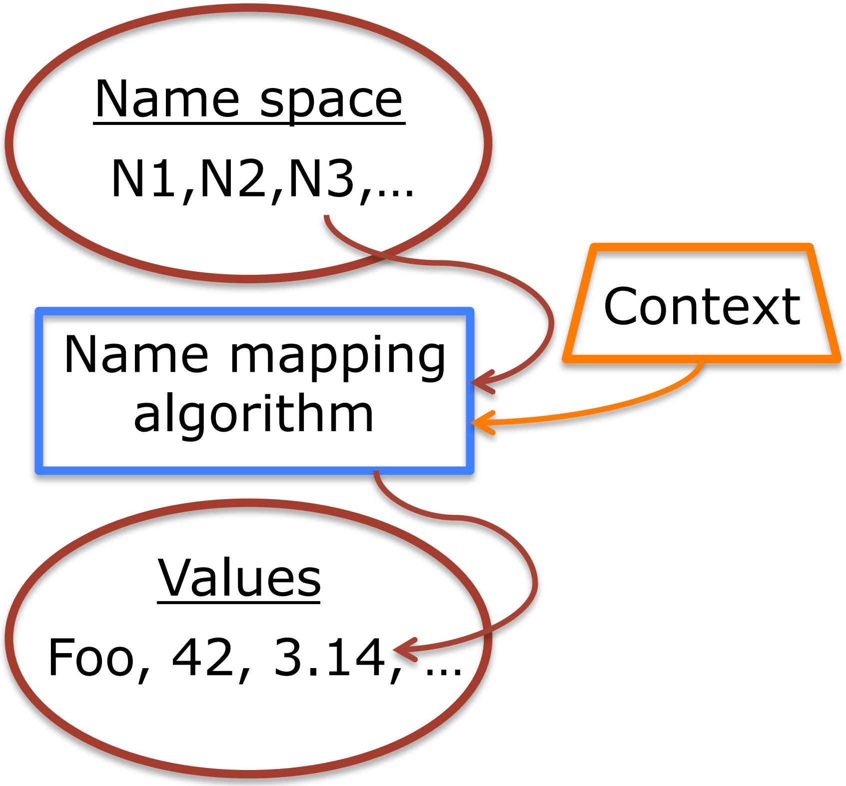
- R1
- 1742
- 18.7.22.69
- web.mit.edu
- http://web.mit.edu/6.033
- 6.033-staff@mit.edu
- amsterdam
- /mit/6.033/www
- foo.c
- .. (as in cd .. or ls ..)
- wc
- (617)253-7149, x37149
- 021-84-2030

address is overloaded name with location info (e.g., LOAD 1742, R1)

Names require a mapping scheme

Source: Saltzer & Kaashoek & Morris (partial)

Name Mapping



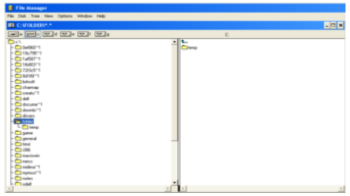
- How can we map names?

- Table lookup
  - Files inside directories
- Recursive lookup
  - Path names in file systems or URLs
- Multiple lookup
  - Java class loading

Fundamental abstractions

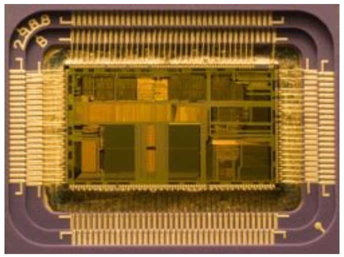
- ***Memory***
  - Read/write





- ***Interpreters***
  - Instruction repertoire
  - Environment
  - Instruction pointer

(loop (print (eval (read))))

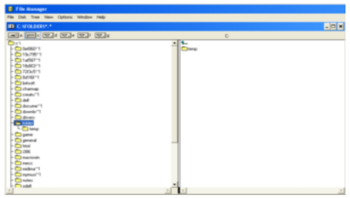


- ***Communication links***
  - Send/receive

Source: Saltzer & Kaashoek & Morris (partial)

Memory

- ***Memory***
  - READ(name) → value
  - WRITE(name, value)



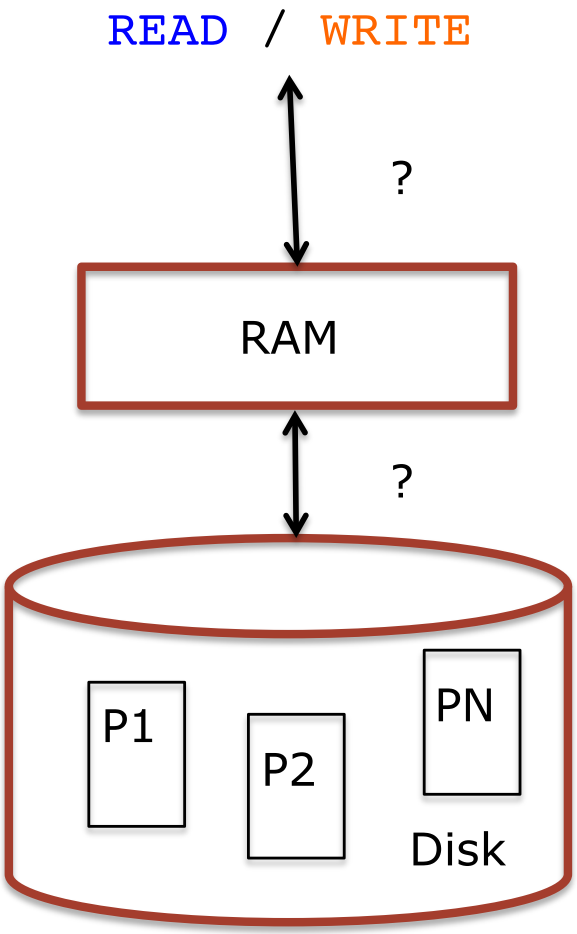
- Examples of Memory
  - Physical memory (RAM)
  - Multi-level memory hierarchy (registers, caches, RAM, flash, disk, tape)
  - Address spaces and virtual memory with paging
  - Transactional memory (hardware and software variants)
  - Database storage engines
  - Key-value stores (e.g., Cassandra, Dynamo)

Source: Saltzer & Kaashoek & Morris (partial)

How would you design a two-level memory abstraction consolidating disk and RAM?

- Characteristics of storage technologies
  - **RAM**: high cost per gigabyte, low latency, volatile
  - **Disk**: low cost per gigabyte, high latency, nonvolatile

- Design top-level abstraction respecting *Memory API*
- Abstraction must:
  - Address as much data as fits in disk
  - Use fixed-size pages for disk transfers
  - Use RAM efficiently to provide for low latency (on average)
  - Neither disk nor memory directly exposed, only READ/WRITE API



Write the pseudocode down!

Address Space Mapping

- Address spaces modular way to multiplex memory
- Naming scheme translating virtual into physical addresses
- Page map
  - Updated by kernel code
  - Lookup implemented in hardware
  - Concerns: Protection (Pr), representation, efficiency

Page Map

Page#	Block#	Pr
42	2	RX
53	45	R
64	97	RW
…	…	…

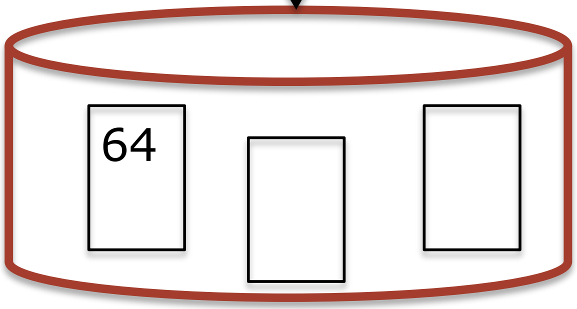
Address Space Mapping: Introducing Disks

- Use disk to store more blocks
- Pages may be either in memory or on disk

- Resident bit (R)
  - Access to non-resident pages results in page faults
- Page Fault
  - An indirection exception for missing pages

Page Map

Page#	Block#	Pr	R	D	P
42	2	RX	1	1	0
53	45	R	1	0	1
64	D-42	RW	0	0	0
...	...	...	...	...	...

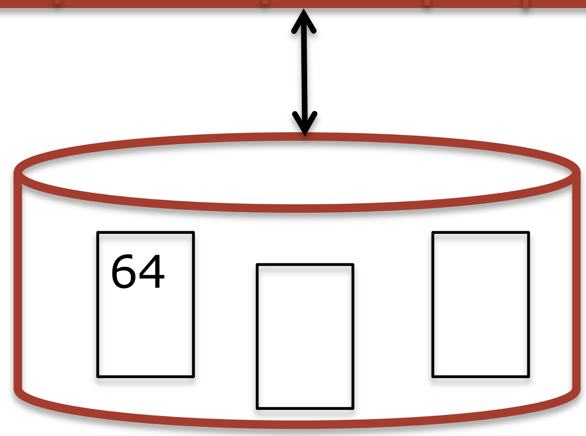


Address Space Mapping: Introducing Disks

- Handling page faults
  - Trap to OS handler
  - Handler loads block from disk and updates mapping
  - If memory full, must choose some *victim* block for replacement
  - Page replacement algorithm, e.g., LRU
- Other metadata
  - *Dirty bit (D)*: Only write page back when it has changed!
  - *Pin bit (P)*: do not remove certain pages (e.g., code of OS handler itself)

Page Map

Page#	Block#	Pr	R	D	P
42	2	RX	1	1	0
53	45	R	1	0	1
64	97	RW	1	0	0
…	…	…	…	…	…

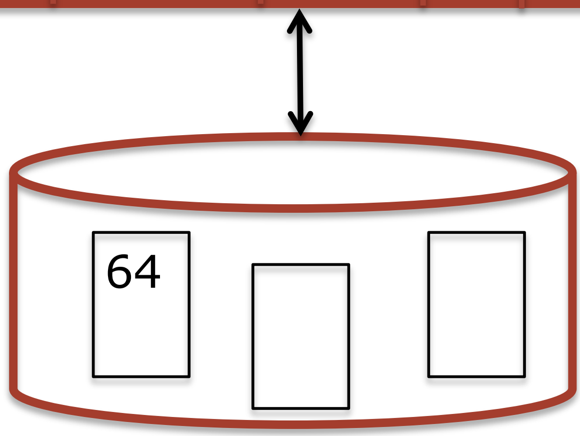


Virtual Memory with Paging: Abstractions, Performance, Fault-Tolerance

- ***Abstraction:*** Do we have any guarantees on two concurrent threads writing to the same memory?
- ***Performance:*** Do we get average latency close to RAM latency?
- ***Fault-Tolerance:*** What happens on failure? Do we have any guarantees about the state that is on disk?

Page Map

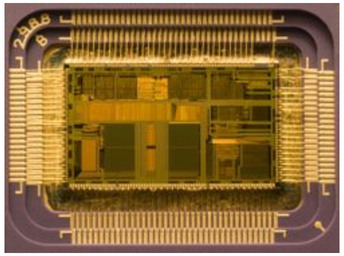
Page#	Block#	Pr	R	D	P
42	2	RX	1	1	0
53	45	R	1	0	1
64	97	RW	1	0	0
…	…	…	…	…	…



Interpreters

- ***Interpreter***
  - Instruction repertoire
  - Environment
  - Instruction pointer

(loop (print (eval (read))))



```
procedure INTERPRET()
        do forever
                instruction ← READ(instruction_pointer)
                perform instruction in environment context
                if interrupt_signal = TRUE then
                instruction_pointer ← entry of INTERRUPT_HANDLER
                environment ← environment of INTERRUPT_HANDLER
```

- Examples of Interpreters
  - Processors (CPU)
  - Programming language interpreters
  - Frameworks, e.g., MapReduce
  - Your own (layered) programs! (RPCs)

Source: Saltzer & Kaashoek & Morris (partial)

Communication links

- ***Communication links***
  - `SEND(linkName, outgoingMessageBuffer)`
  - `RECEIVE(linkName, incomingMessageBuffer)`

- Examples of Communication Links
  - Ethernet interface
  - IP datagram service
  - TCP sockets
  - Message-Oriented Middleware (MOM)
  - Streams
  - Multicast (e.g., CATOCS: Causal and Totally-Ordered Communication System)

Source: Saltzer & Kaashoek & Morris (partial)

Memory, Interpreters, Communication Links: Is that all there is?

- Other abstractions also useful!

- ***Synchronization***
  - Locks
  - Condition variables & monitors

  (see, e.g., Chubby lock service from Google)

- ***Data processing***
  - Data transformations
  - Operators

  (see, e.g., data-parallel implementations)

What should we learn today?

- Identify the fundamental abstractions in computer systems and their APIs, including memory, interpreters, communication links
- Explain how names are used in the fundamental abstractions
- Design a top-level abstraction, respecting its correspondent API, based on lower-level abstractions
- Discuss performance and fault-tolerance aspects of such a design