Principles of Computer System Design (PCSD)

Marcos Vaz Salles

Assistant Professor, DIKU

Matchmaking event: Data and Biotech

What:

- Meet the following companies: Novozymes, Novo Nordisk, Lundbeck, Biomediq, Chr. Hansen, CVIVA and Statens Serum Institut.
- Each company will present problems/challenges that they want YOU to solve in a Master Thesis.
- After the presentations you will have the opportunity to meet up with 2-3 companies for a one-on-one discussion about future collaboration.

When:

November 22nd, 1pm-6pm – **THIS FRIDAY**
BUT REGISTER BY TOMORROW

Where:

HCØ, Universitetsparken 5

Sign up and read more: http://www.science.ku.dk/matchmaking/

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

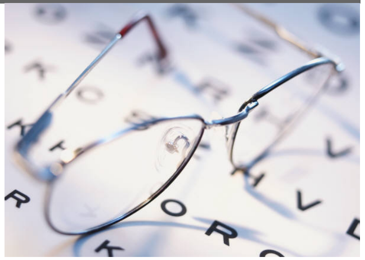
  How can I understand the guarantees and reliability of scalable services offered to me on the cloud?

- The Cloudera/Greenplum/ Teradata question

  How do I build systems to process TBs to PBs of data?

Intro Video by Michael Brodie

Computer Science 2.0
presented at VLDB 2007, Vienna, Austria



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.

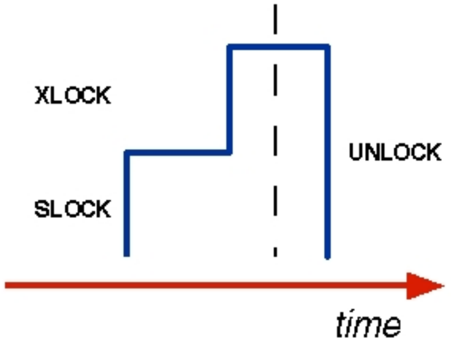
PCSD: What will we study?

- 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

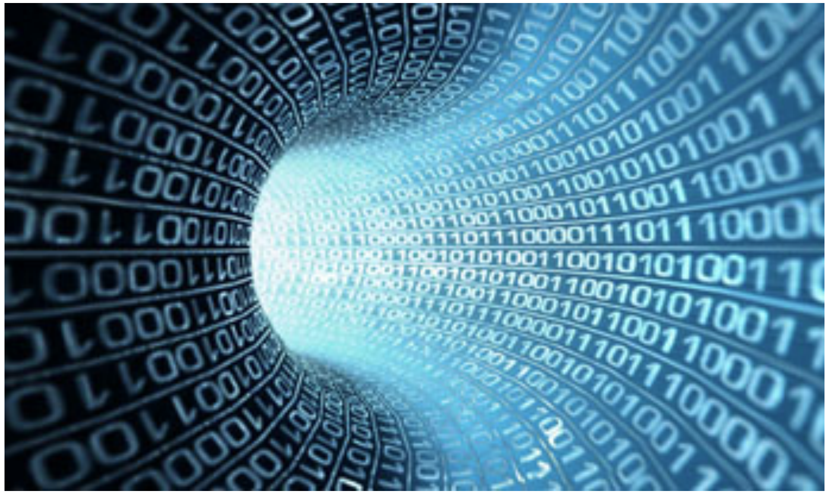
PCSD: What will we study?

- 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., duplicate elimination & grouping, set operations, joins)
  - Parallelism



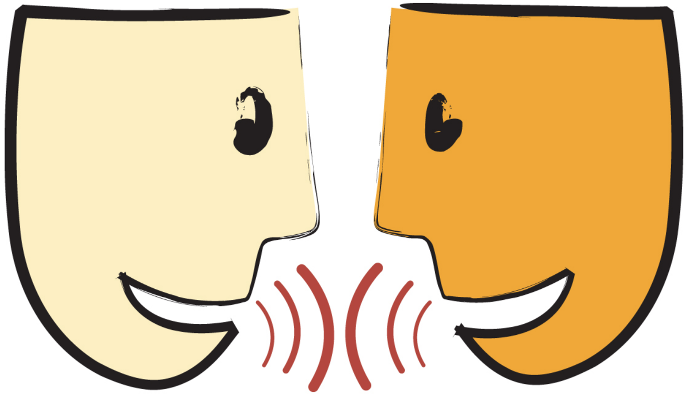
Property: Scalability with Data Size

PCSD: What will we study?

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

PCSD: What will we study?

- Guest Lectures
  - **Date: December 17**
  - **Service-Oriented Architecture (SOA) in the Real World**, by Morten Steffensen of **Netcompany**
  - **Recovery in Practice,** by Paz Padilla Thygesen of **IBM**



References & Course Materials

- Course webpage
  - Kurser: http://kurser.ku.dk/course/ndaa09004u/2013-2014

- 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.

- 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
  - Jyrki Katajainen
    - jyrki@diku.dk
  - Office hours:
    - By email appointment

- TAs
  - Vivek Shah
  - Ashwini Satish Joshi
  - Jacob Jepsen
  - 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

Learning is the main goal!

First Steps

- **Java Warm-up Exercise**
  - Available on Absalon
  - If you passed Advanced Java, you do not need it ☺
  - If you did not take Advanced Java, the warm-up assignment will tell you the level of Java you **need** for this course

- **First TA Session**
  - This Thursday, **November 21**
  - **Brief review** of Java Warm-up Exercise
  - 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

Assignments

- **4 + 1 take-home assignments**
  - **Groups: 2 people** strongly recommended
  - Each assignment worth **10 points**
  - Minimum of **30 points** to qualify for exam
- **First 4 assignments**
  - Build **specific** skills and concepts on **weekly** basis
  - Include both theory exercises and programming
  - **Due dates: December 1, December 8,
    December 15, December 22**
- **Final 5th assignment**
  - **Integrates** multiple skills and concepts into a single assignment
  - **Exam-style**: Based on last year's exam
  - Includes both theory exercises and programming
  - **Due date: January 8**

Exam

- **Exam format**
  - 5-day take-home assignment with external grading on 7-point scale, between **January 15** and **January 21**
  - Must be solved individually, no groups allowed
  - Includes both theory exercises and 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), Nesime Tatbul (ETH Zurich), James Kurose & Keith Ross (U Mass Amherst & NYU, networking textbook), Jens Dittrich (Saarland University)

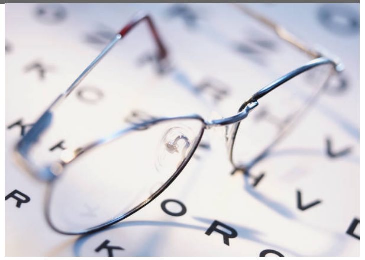
PCSD: Evaluation and Evolution

- PCSD is evolving based on your feedback
  - Positive sentiment last year after changes to syllabus
  - Feedback from last year
    - Syllabus is great!
    - Assignments were too steep in workload / learning curve

- We are listening
  - Course syllabus kept this year
  - Assignments broken down into 4 + 1 model
  - Skills building course: Advanced Java

- Helping out
  - Come and talk to me, give us your feedback!
  - Make sure to fill out our evaluation questionnaires – WE READ THEM

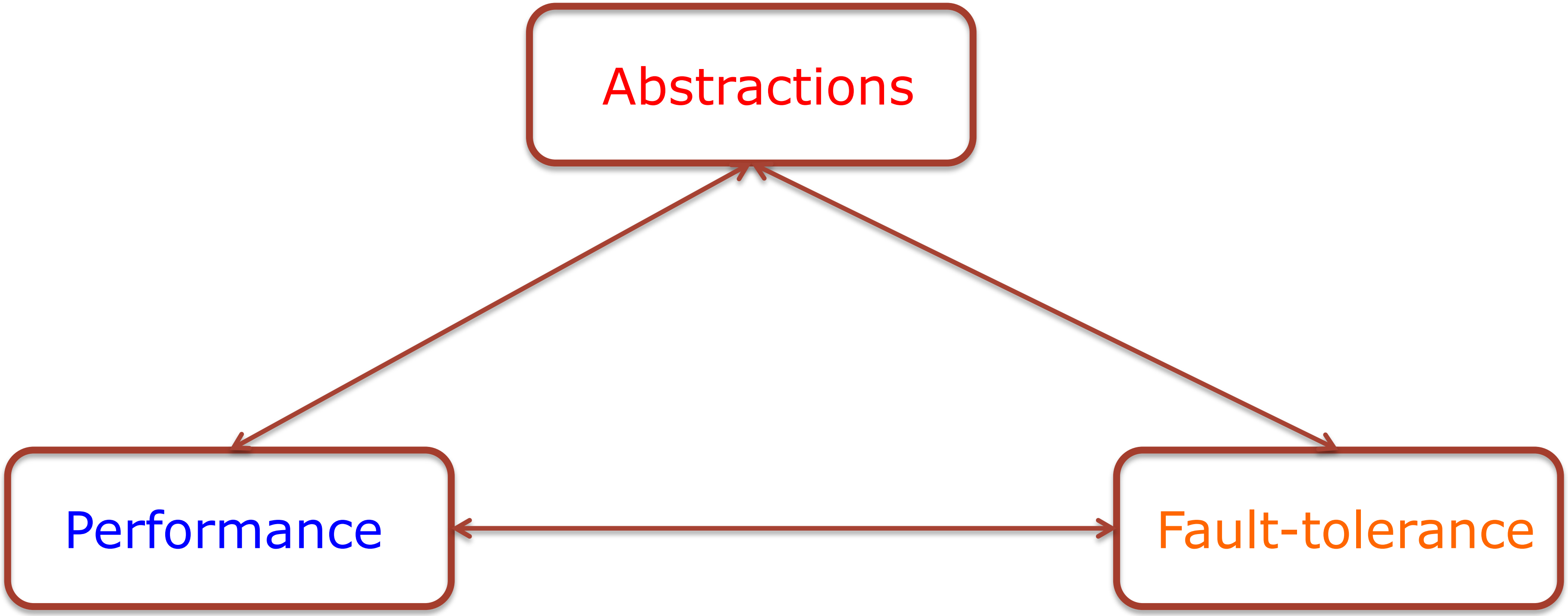
Questions so far?



What should we learn today?

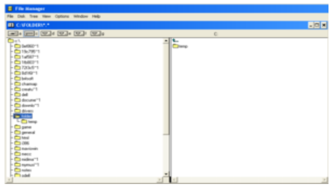
- Identify the fundamental abstractions in computer systems and their APIs, including memory, interpreters, communication links
- Explain how names are used in the fundamental abstractions
- Design a top-level abstraction, respecting its correspondent API, based on lower-level abstractions
- Discuss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

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



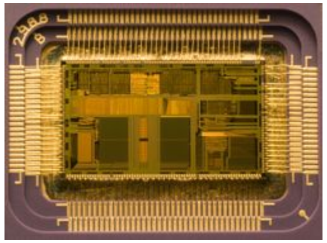
Fundamental abstractions

- ***Memory***
  - Read/write
- ***Interpreters***
  - Instruction repertoire
  - Environment
  - Instruction pointer
- ***Communication links***
  - Send/receive



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

Names make connections



Source: Saltzer & Kaashoek & Morris (partial)

Examples of Names

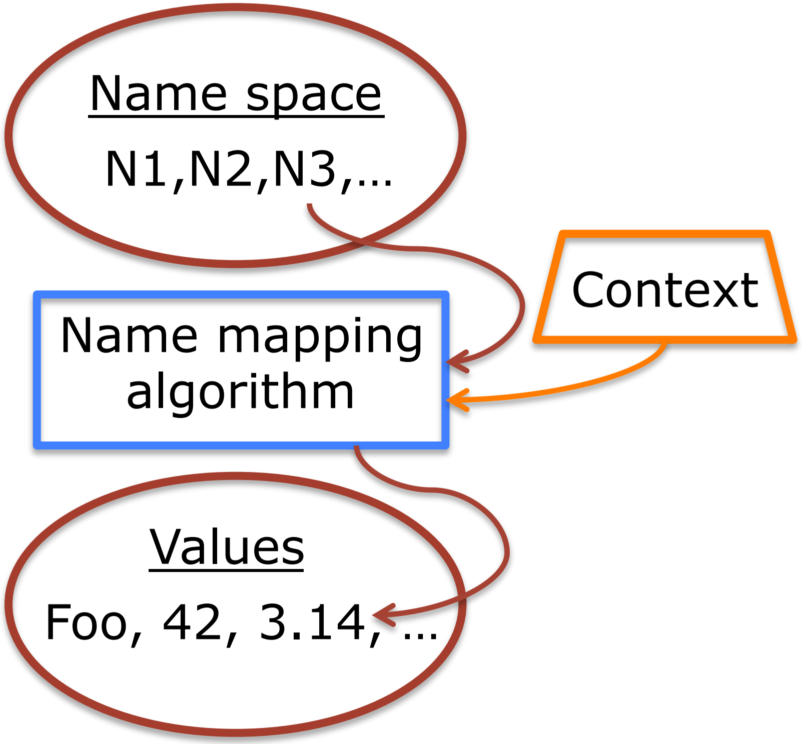
- R1
- 1742
- 18.7.22.69
- web.mit.edu
- http://web.mit.edu/6.033
- 6.033-staff@mit.edu
- amsterdam
- /mit/6.033/www
- foo.c
- .. (as in cd .. or ls ..)
- wc
- (617)253-7149, x37149
- 021-84-2030

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

Names require a mapping scheme

Source: Saltzer & Kaashoek & Morris (partial)

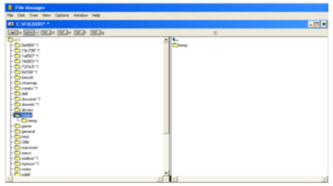
Name Mapping



- How can we map names?
- Table lookup
  - Files inside directories
- Recursive lookup
  - Path names in file systems or URLs
- Multiple lookup
  - Java class loading

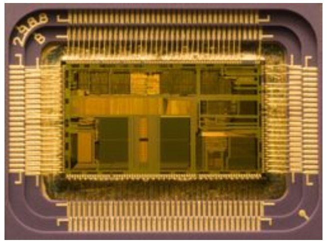
Fundamental abstractions

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



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

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

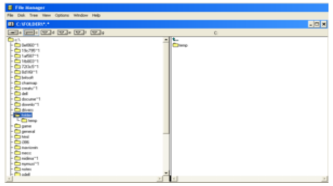


- ***Communication links***
  - Send/receive

Source: Saltzer & Kaashoek & Morris (partial)

Memory

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



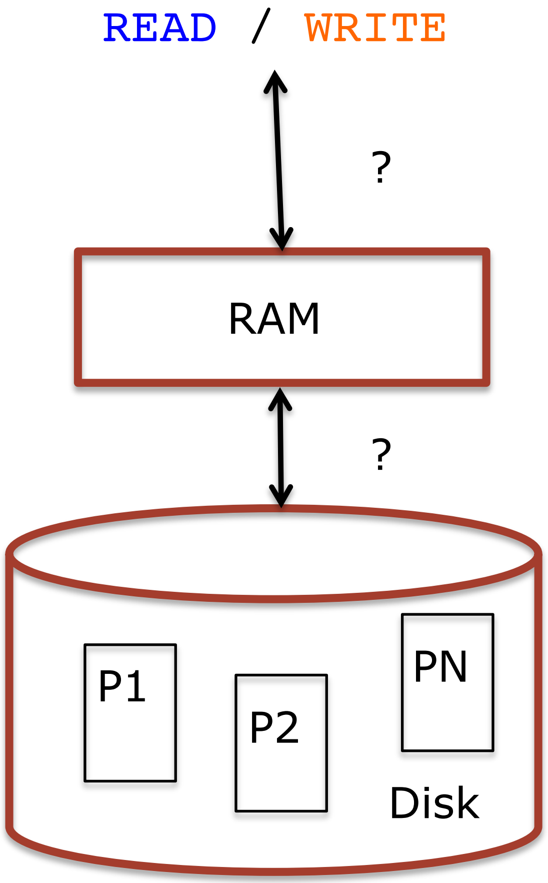
- Examples of Memory
  - Physical memory (RAM)
  - Multi-level memory hierarchy (registers, caches, RAM, flash, disk, tape)
  - Address spaces and virtual memory with paging
  - Transactional memory (hardware and software variants)
  - Database storage engines
  - Key-value stores (e.g., Cassandra, Dynamo)

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

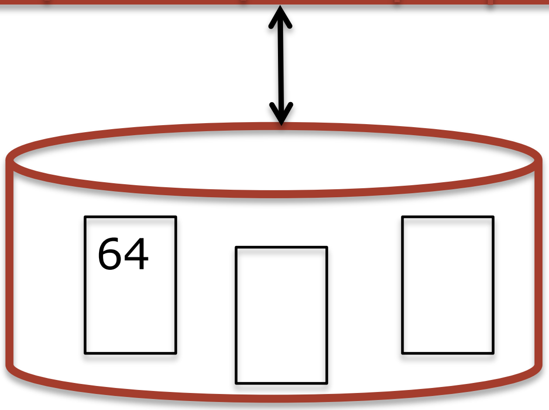
Page#	Block#	Pr
42	2	RX
53	45	R
64	97	RW
…	…	…

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
…	…	…	…	…	…

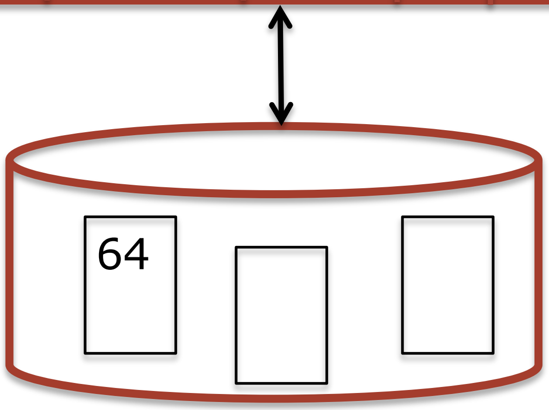


Address Space Mapping: Introducing Disks

- 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
…	…	…	…	…	…

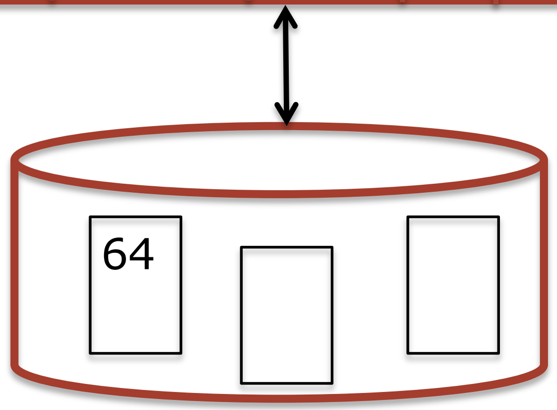


Virtual Memory with Paging: Abstractions, Performance, Fault-Tolerance

- ***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

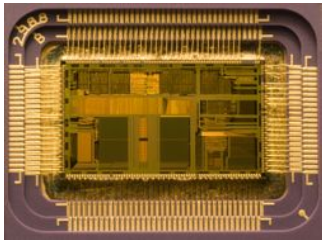
Page#	Block#	Pr	R	D	P
42	2	RX	1	1	0
53	45	R	1	0	1
64	97	RW	1	0	0
…	…	…	…	…	…



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?

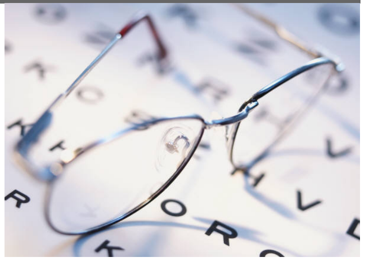
- Other abstractions also useful!

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

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

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

  (see, e.g., parallel implementations of SQL)



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