18-600 Foundations of Computer Systems

Lecture 20: "Parallel Systems & Programming"

John Paul Shen

November 8, 2017

➢ Recommended Reference:
- "Parallel Computer Organization and Design," by Michel Dubois, Murali Annavaram, Per Stenstrom, Chapters 5 and 7, 2012.

Electrical & Computer ENGINEERING

Carnegie Mellon University

18-600 Foundations of Computer Systems

Lecture 20: "Parallel Systems & Programming"

A. Parallel Architectures
   1. Multicore Processors (MCP)
   2. Multi-Computer Clusters (MCC)

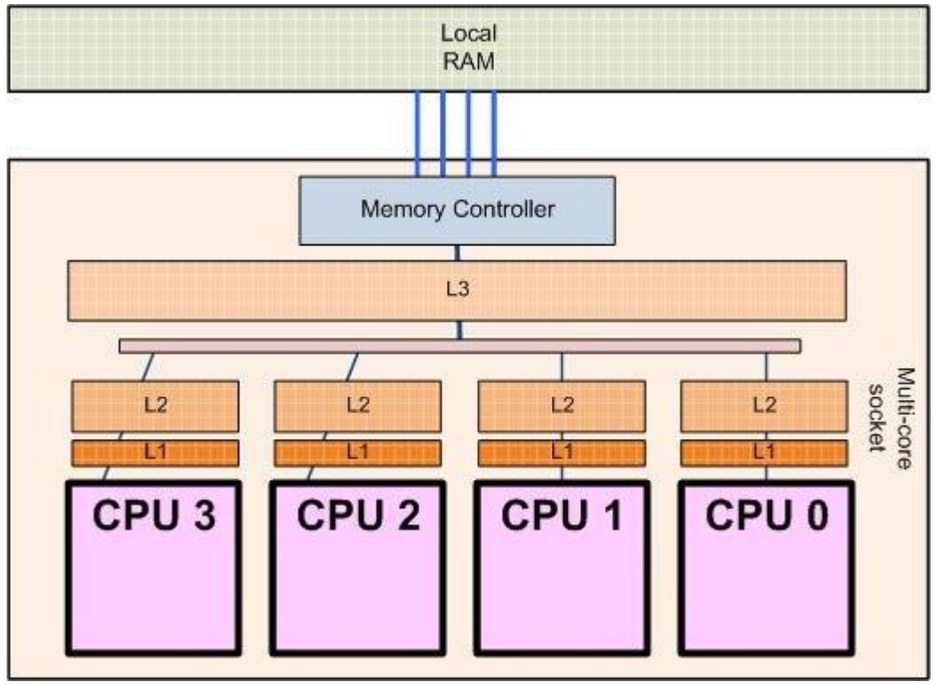
B. Parallel Programming
   1. Finding Parallelism
   2. Programming Models
   3. Shared Memory Model
   4. Message Passing Model

Electrical & Computer ENGINEERING

A.1. Multicore Processors (MCP)

MULTIPROCESSING

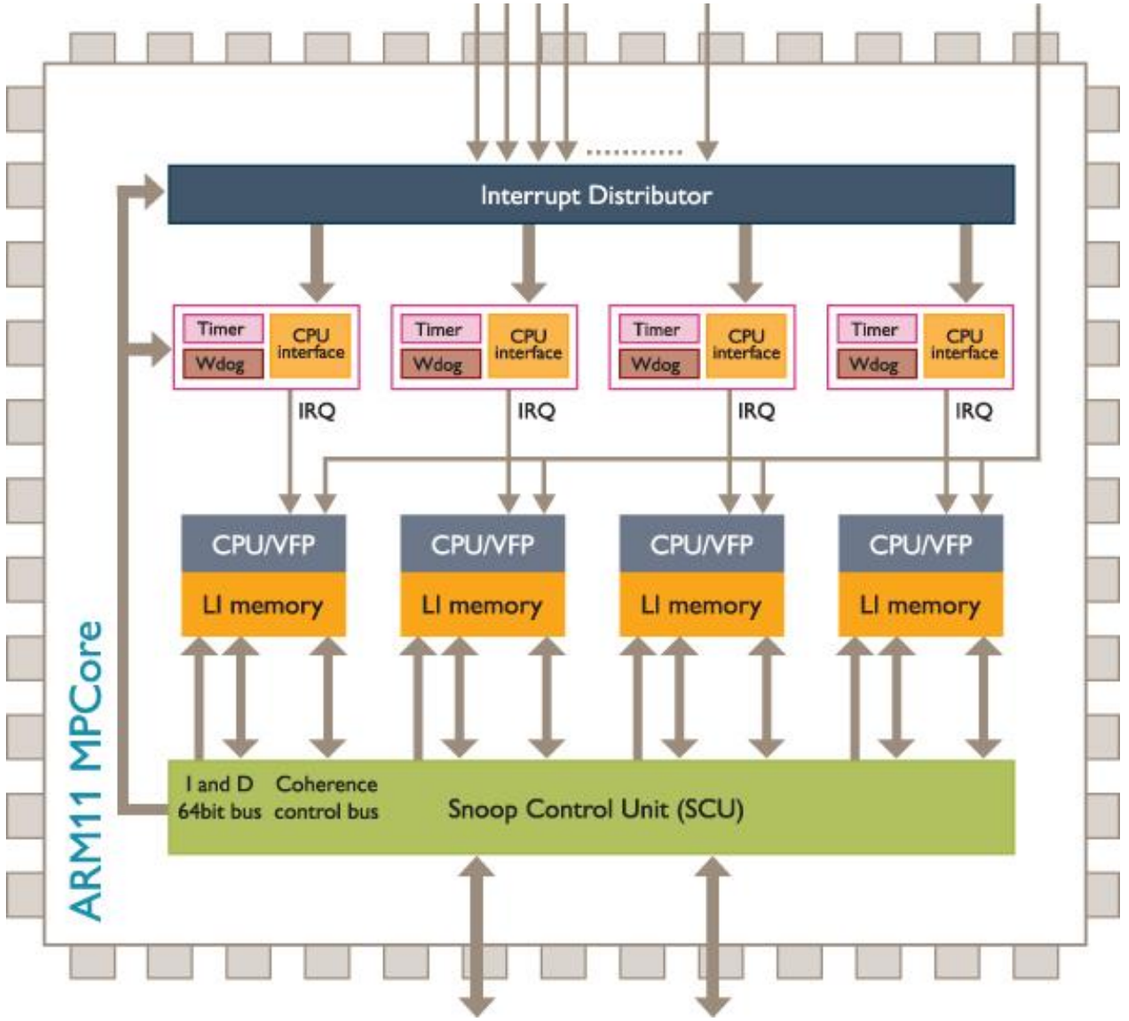
Shared Memory Multicore Processors (MCP) or Chip Multiprocessors (CMP)



Electrical & Computer ENGINEERING

The Case for Multicore Processors

- Stalled Scaling of Single-Core Performance
- Expected continuation of Moore's Law
- Throughput Performance for Server Workloads



Multicore Processor Design Questions

- Type of cores
  - Big or small cores, e.g. few OOO cores Vs many simple cores
  - Same or different cores, e.g. homogeneous or heterogeneous
- Memory hierarchy
  - Which caching levels are shared and which are private
  - How to effectively share a cache
- On-chip interconnect
  - Bus vs. ring vs. scalable interconnect (e.g., mesh)
  - Flat vs. hierarchical organizations
- HW assists for parallel programming
  - HW support for fine-grain scheduling, transactions, etc.

On-Chip Bus/Crossbar Interconnects

- Used widely (Power4/5/6/7 Piranha, Niagara, etc.)
  - Assumed not scalable
  - Is this really true, given on-chip characteristics?
  - May scale "far enough": watch out for arguments at the limit
  - e.g. swizzle-switch makes x-bar scalable enough [UMich]
- Simple, straightforward, nice ordering properties
  - Wiring can be a nightmare (for crossbar)
  - Bus bandwidth is weak (even multiple busses)
  - Compare DEC Piranha 8-lane bus (32GB/s) to Power4 crossbar (100+GB/s)
  - Workload demands: commercial vs. scientific

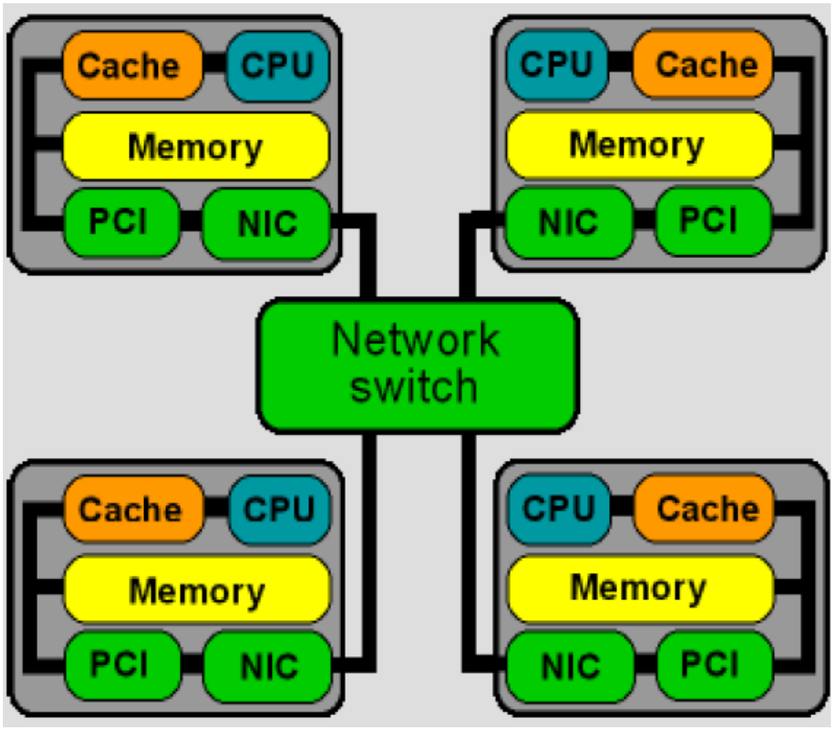
A.2. Multi-Computer Clusters (MCC)

MULTIPROCESSING

Shared Memory Multicore Processors (MCP) or Chip Multiprocessors (CMP)

CLUSTER COMPUTING

Shared File System and LAN Connected Multi-Computer Clusters (MCC)



Electrical & Computer ENGINEERING

[J.E. Smith, 2007, U. Wisc. ECE/CS 757]

What is a Multi-Computer Cluster?

- Cluster = a set of computers connected with a network
  - *Cluster can be viewed by the user as a single system (or a "multi-computer" system)*
  - Typically commodity computers (PCs) connected by commodity LAN (Ethernet)
  - Each computer (cluster node) runs its own OS and has its own address space
  - Supports execution of parallel programs via message passing with a master node
  - Typically supported by a shared file system and some "clustering middleware"
- Advantages
  - Easy to build: early systems were customer built using commodity PCs and networks
  - Relatively inexpensive: can get significant throughput performance at very low cost
  - Wide range of systems: can vary from small personal clusters to supercomputers
  - *Leverage concurrent advancements in personal computers and local area networks*

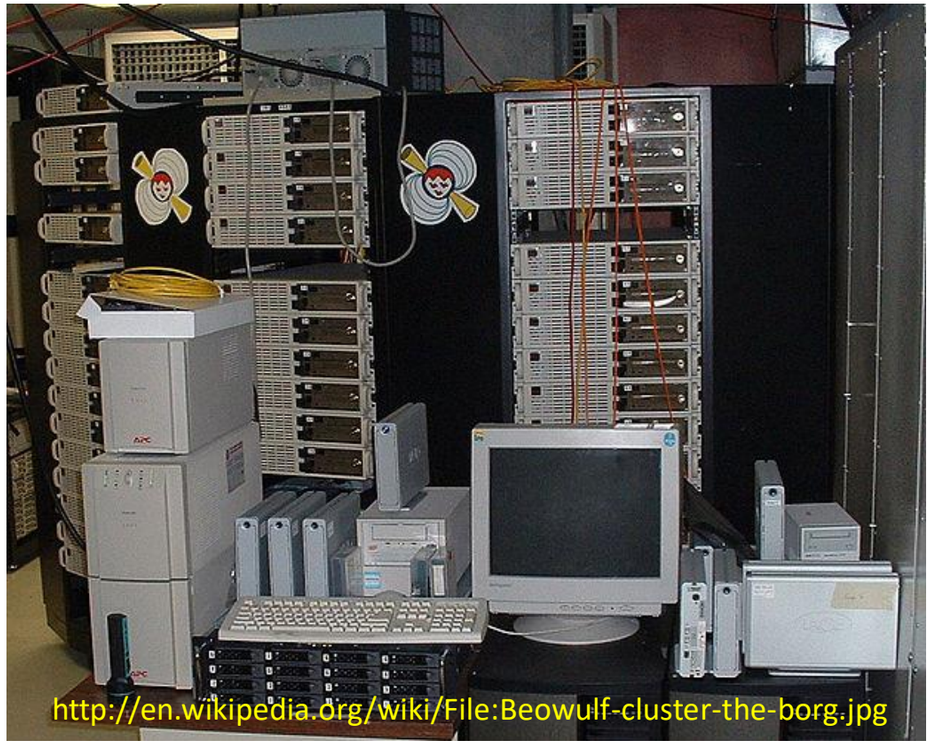
The Beowulf Cluster

[Thomas Sterling & Donald Becker, NASA, 1994]

"Beowulf is a multi-computer which can be used for parallel computations. It is a system with one server node, and one or more client nodes connected via Ethernet. Beowulf also uses commodity software like FreeBSD, Linux, PVM (Parallel Virtual Machine) and MPI (Message Passing Interface). Server node controls the cluster and serves files to the client nodes."

"If you have two networked computers which share at least the `/home` *file system via NFS, and trust each other to execute remote shell (*rsh*), then it could be argued that you have a simple, two node Beowulf machine."*

http://en.wikipedia.org/wiki/File:Beowulf.jpg



http://en.wikipedia.org/wiki/File:Beowulf-cluster-the-borg.jpg

Beowulf Cluster Attributes

- Multi-Computer Architecture
  - Cluster Nodes: commodity PC's (each with its OS instance, memory address space, disk}
  - Cluster Interconnect: commodity LAN's, Ethernet, switches
  - Operating System: Unix, BSD, Linux (same OS running on each node)
  - Cluster Storage: local storage in each node, centralized shared storage
- Parallel Programming Model
  - "Clustering middleware:" a SW layer atop the nodes to orchestrate the nodes for users
  - Application programs do not see the cluster nodes, only interface with master node
  - PVM: (Oak Ridge NL) library installed on every node, runtime environment for message passing, task and resource management.
  - MPI: (ARPA, NSF) use TCP/IP and socket connection, now widely available, library to achieve high performance at low cost.

Applications

- Commercial
  - Large shared databases
  - Largely independent threads
  - “Architecture” often means software architecture
  - May use higher performance storage area network (SAN)
- Scientific
  - Inexpensive high performance computing
  - Based on message passing
  - Also PGAS (partitioned global address space); software shared memory
  - May use higher performance node-to-node network
  - Where HPC clusters end and MPPs begin isn’t always clear

Software Considerations

- Throughput Parallelism
  - As in many commercial servers
  - Distributed OS message passing
  - VAXcluster early example
- True Parallelism
  - As in many scientific/engineering applications
  - Use programming model and user-level API
- Programming Models
  - Message-Passing
    - Commonly used
  - Shared memory
    - Virtual Shared Memory, Software Distributed Shared Memory
    - PGAS – software abstraction, runtime invokes remote DMA
- Of course, a real system can do both throughput and true parallelism

Computer Cluster – Summary

- Reasons for clusters
  - Performance – horizontal scaling
  - Cost: commodity h/w, commodity s/w
  - Redundancy/fault tolerance
- Hardware Challenges
  - Cost, commodity components
  - Power, cooling, energy proportionality
  - Reliability, usability, maintainability
- Software Challenges
  - Programming model
  - Applications with suitable characteristics
  - Load balancing

Google Cluster

[Luiz André Barroso , Jeffrey Dean , Urs Hölzle, 2003]

Web Search For a Planet: The Google Cluster Architecture

Luiz Andre Barroso, Jeffrey Dean, Urs Holzle
Google

Google Cluster Architecture

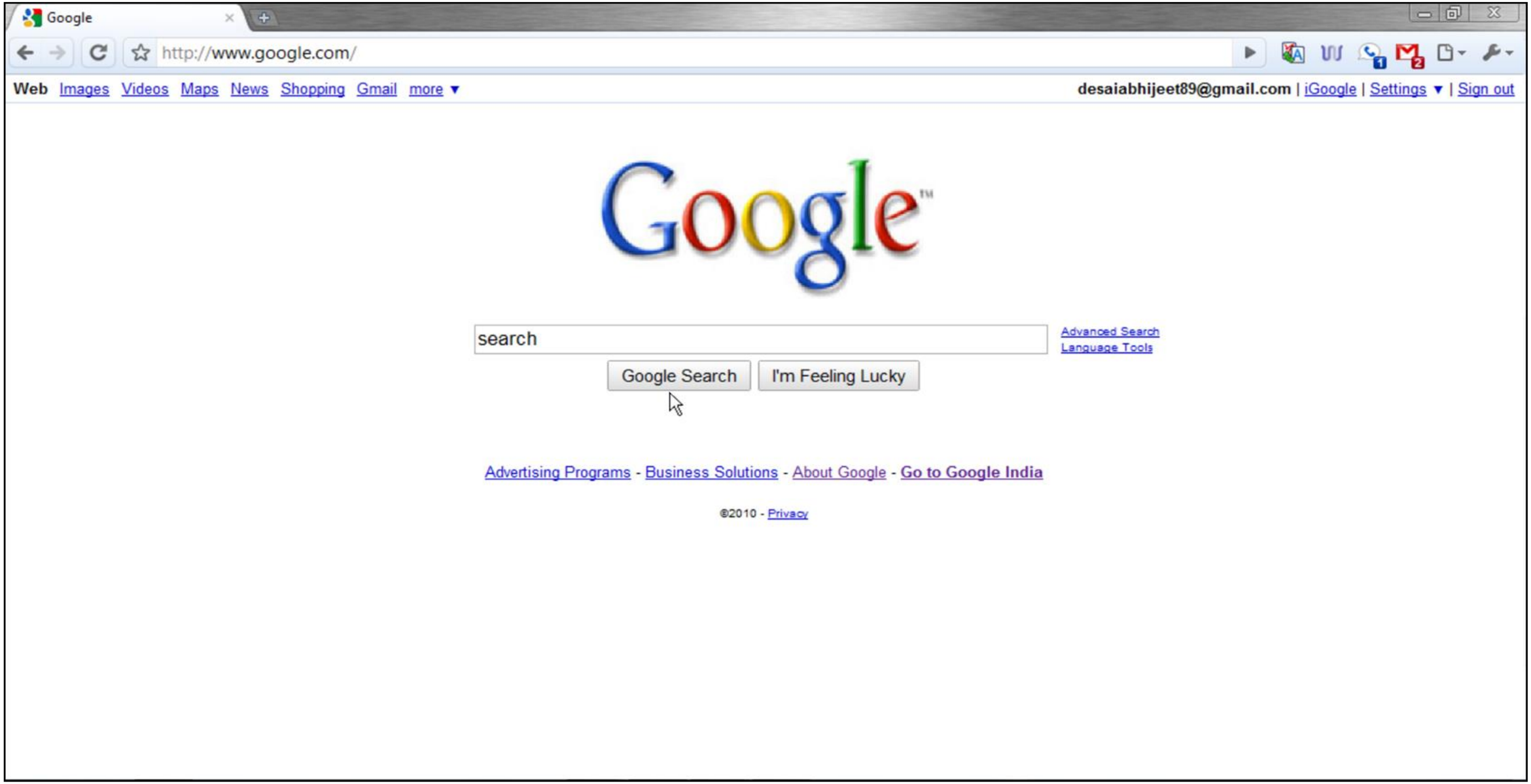
Web Search for a Planet and much more….

Abhijeet Desai
desaiabhijeet89@gmail.com

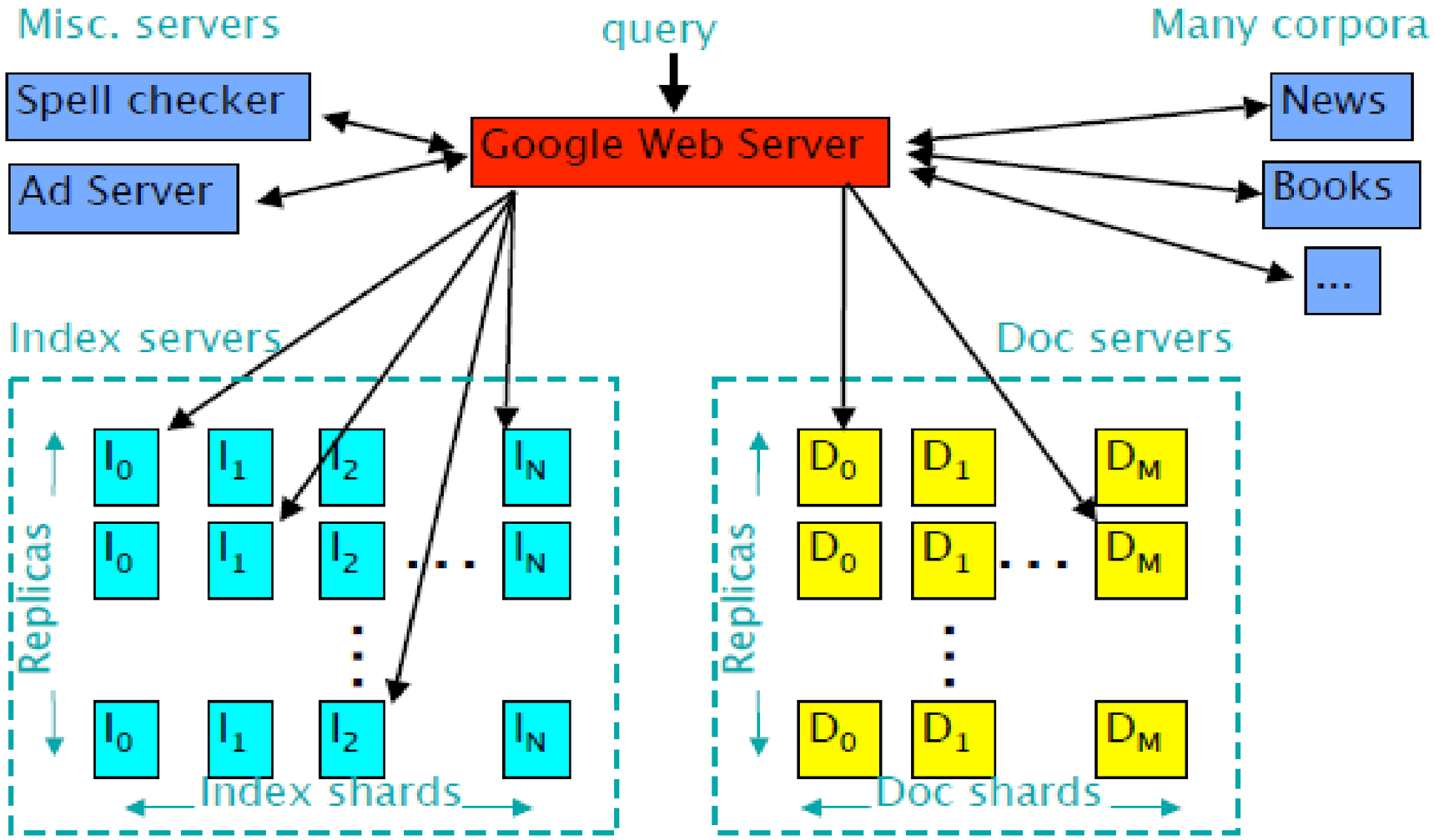
Electrical & Computer ENGINEERING

Design Principles of Google Clusters

- Software level reliability
  - No fault-tolerant hardware features; e.g.
    - redundant power supplies
    - A redundant array of inexpensive disks (RAID)
- Use replication
  - for better request throughput and availability
- Price/performance beats peak performance
  - CPUs giving the best performance per unit price
  - Not the CPUs with best absolute performance
- Using commodity PCs
  - reduces the cost of computation


Google
http://www.google.com/
Web Images Videos Maps News Shopping Gmail more ▼
desaiabhijeet89@gmail.com | iGoogle | Settings ▼ | Sign out
Google
search
Advanced Search
Language Tools
Google Search
I'm Feeling Lucky
Advertising Programs - Business Solutions - About Google - Go to Google India
©2010 - Privacy

Google Query Serving Infrastructure



Elapsed time: 0.25s, machines involved: 1000s+

Application Summary: Serving a Google Query

- Geographically distributed clusters
  - Each with many thousands of machines
- First perform Domain Name System (DNS) lookup
  - Maps request to a nearby cluster
- Send HTTP request to selected cluster
  - Request serviced locally w/in that cluster
- Clusters consist of Google Web Servers (GWSes)
  - Hardware-based load balancer distributes load among GWSes

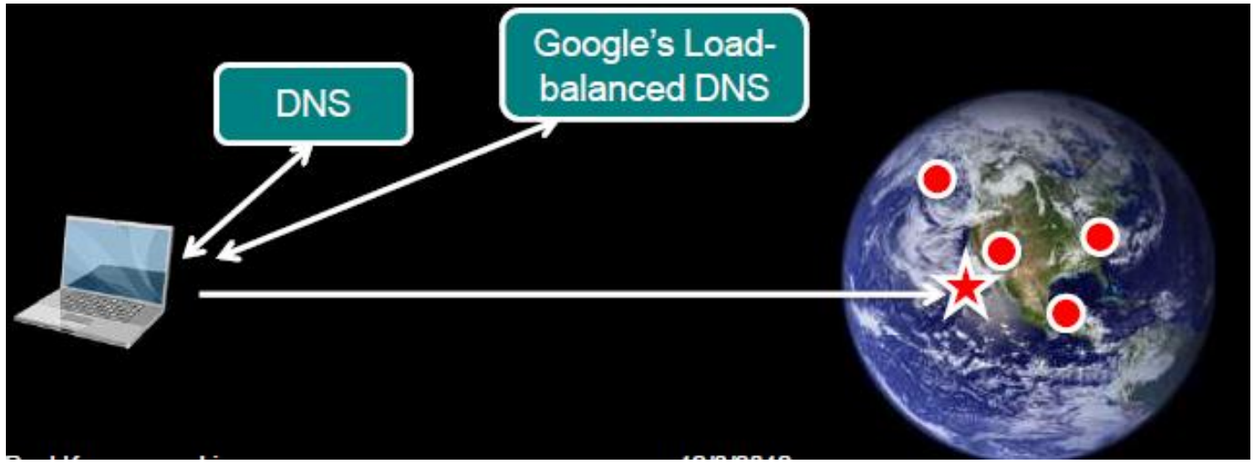
Query Execution

- Phase 1:
  - Index servers consult inverted index that maps query word to list of documents
  - Intersect hit lists and compute relevance index, score for each doc (*secret sauce*)
  - Results in ordered list of document ids (*docids*)
- Both documents and inverted index consume terabytes of data
- Index partitioned into "shards", shards are replicated
  - Index search becomes highly parallel; multiple servers per shard load balanced requests
  - Replicated shards add to parallelism and fault tolerance
- Phase 2:
  - Start w/ docids and determine title, resource locator, document summary
- Done by document servers
  - Partition documents into shards
  - Replicate on multiple servers
  - Route requests through load balancers

Step 1 – DNS

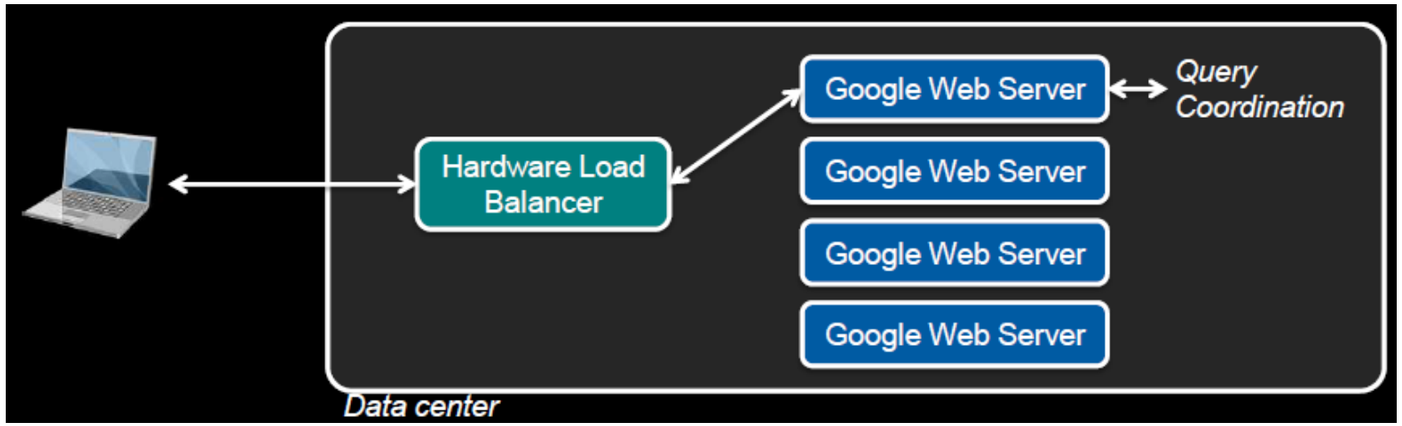
[Paul Krzyzanowski, 2012, Rutgers]

- User's browser must map google.com to an IP address
- "google.com" comprises
  - Multiple clusters distributed worldwide
  - Each cluster contains thousands of machines
- DNS-based load balancing
  - Select cluster by taking user's geographic proximity into account
  - Load balance across clusters
  - [similar to Akamai's approach]



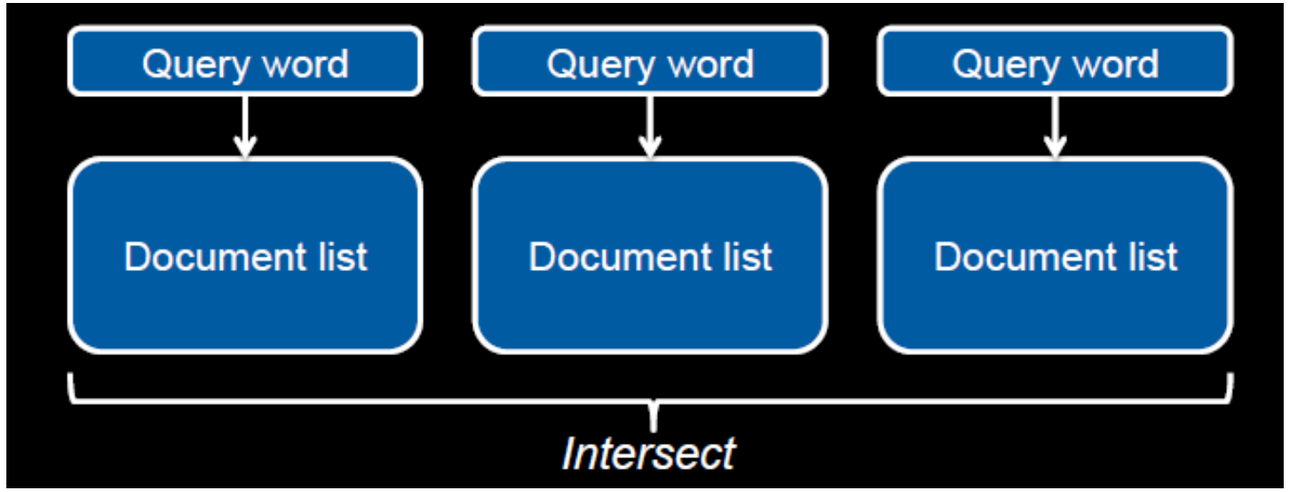
Step 2 – Send HTTP Request

- IP address corresponds to a load balancer within a cluster
- Load balancer
  - Monitors the set of Google Web Servers (GWS)
  - Performs local load balancing of requests among available servers
- GWS machine receives the query
  - Coordinates the execution of the query
  - Formats results into an HTML response to the user



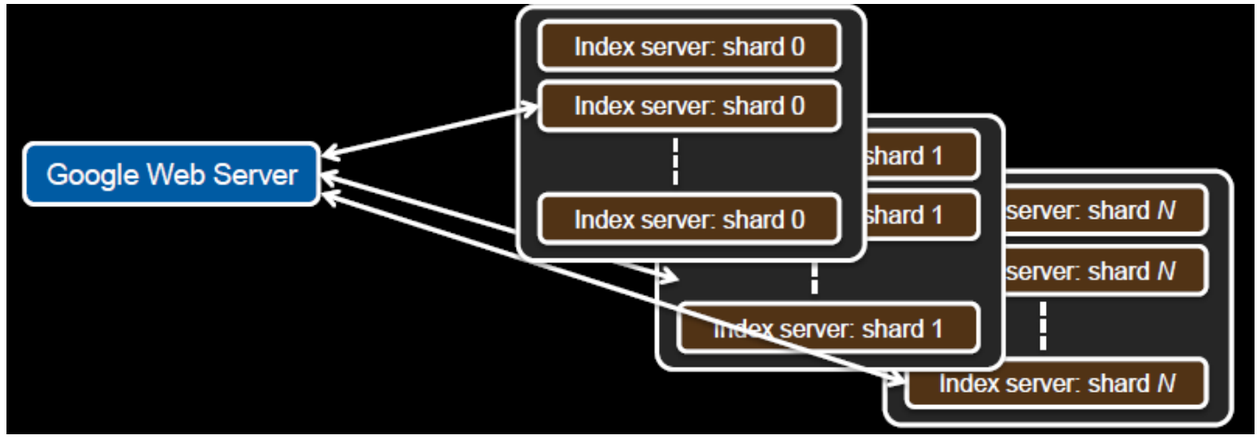
Step 3 – Find Document via Inverted Index

- Index Servers
  - Map each query word → {list of documents} (hit list)
    - Inverted index generated from web crawlers → MapReduce
  - Intersect the hit lists of each per-word query
    - Compute relevance score for each document
    - Determine set of documents
    - Sort by relevance score



Parallelizing the Inverted Index

- Inverted index is 10s of terabytes
- Search is parallelized
  - Index is divided into *index shards*
    - Each index shard is built from a randomly chosen subset of documents
    - Pool of machines serves requests for each shard
    - Pools are load balanced
  - Query goes to one machine per pool responsible for a shard
- Final result is ordered list of document identifiers (*docids*)

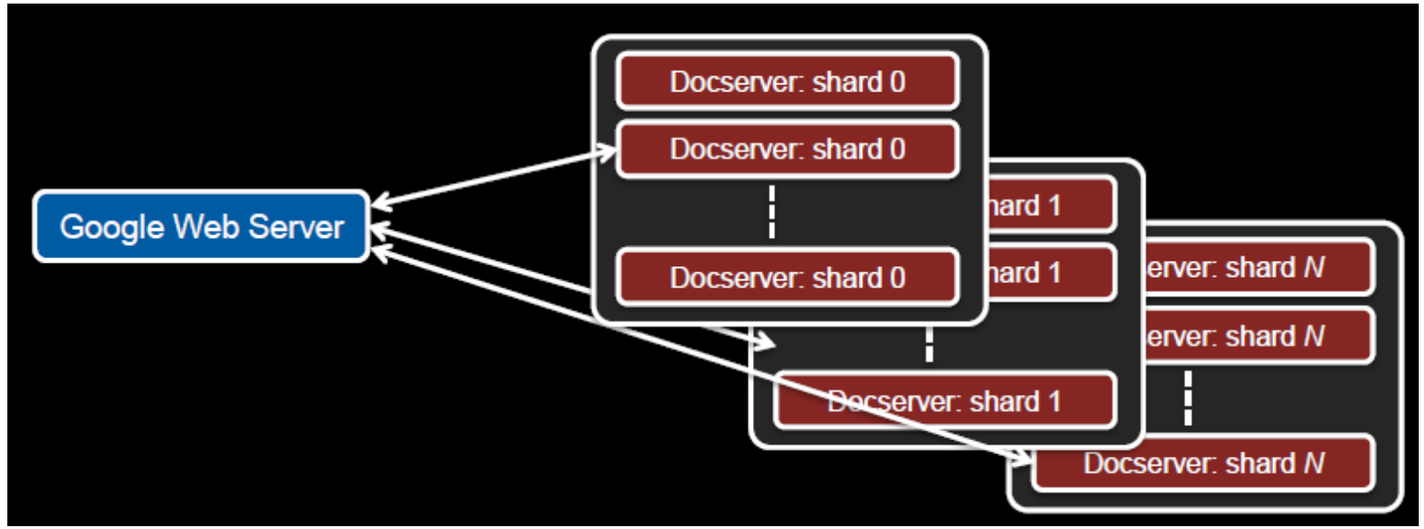


Step 4 – Get Title and URL for Each docid

- For each docid, the GWS looks up
  - Page title
  - URL
  - Relevant text: document summary specific to the query
- Handled by document servers (docservers)

Parallelizing Document Lookup

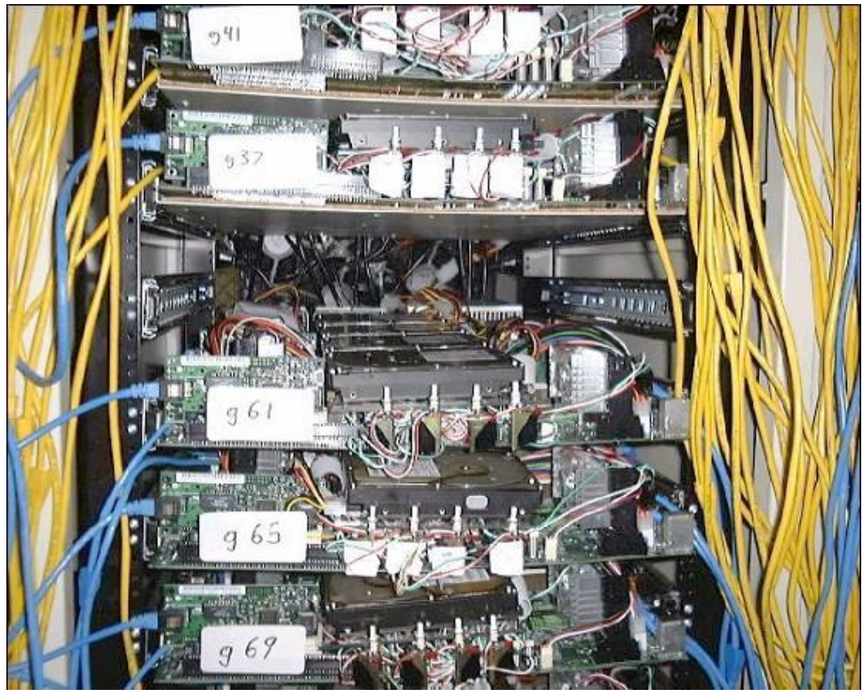
- Like index lookup, document lookup is partitioned & parallelized
- Documents distributed into smaller shards
  - Each shard = subset of documents
- Pool of load-balanced servers responsible for processing each shard
- Together, document servers access a cached copy of the entire web!



Google Clusters Through the Years

"Google" Circa 1997 (google.stanford.edu)

Google (circa 1999)



Google Clusters Through the Years

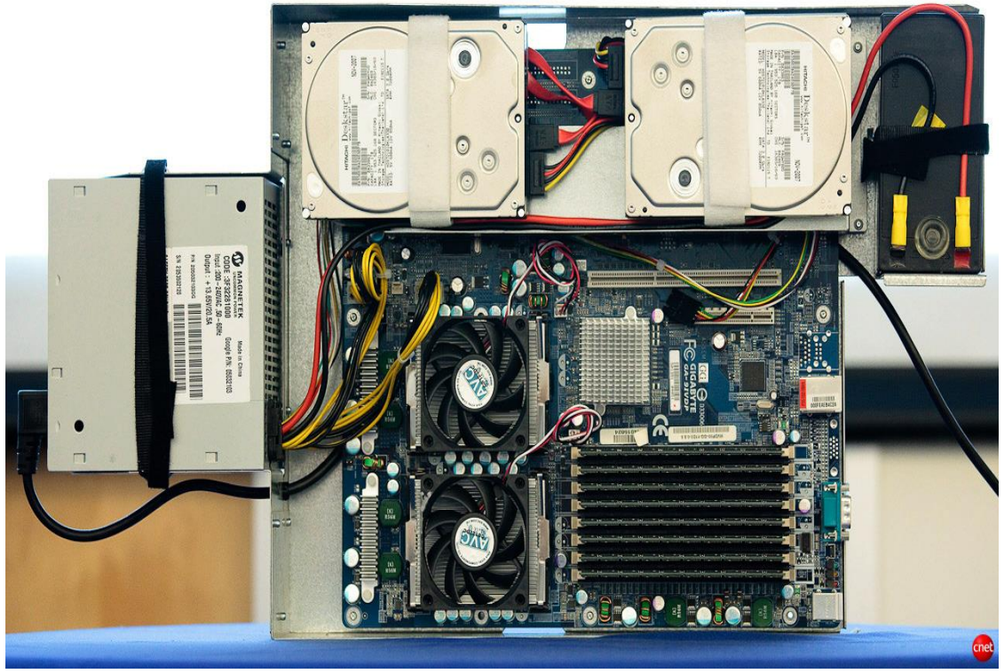
Google Data Center (Circa 2000)

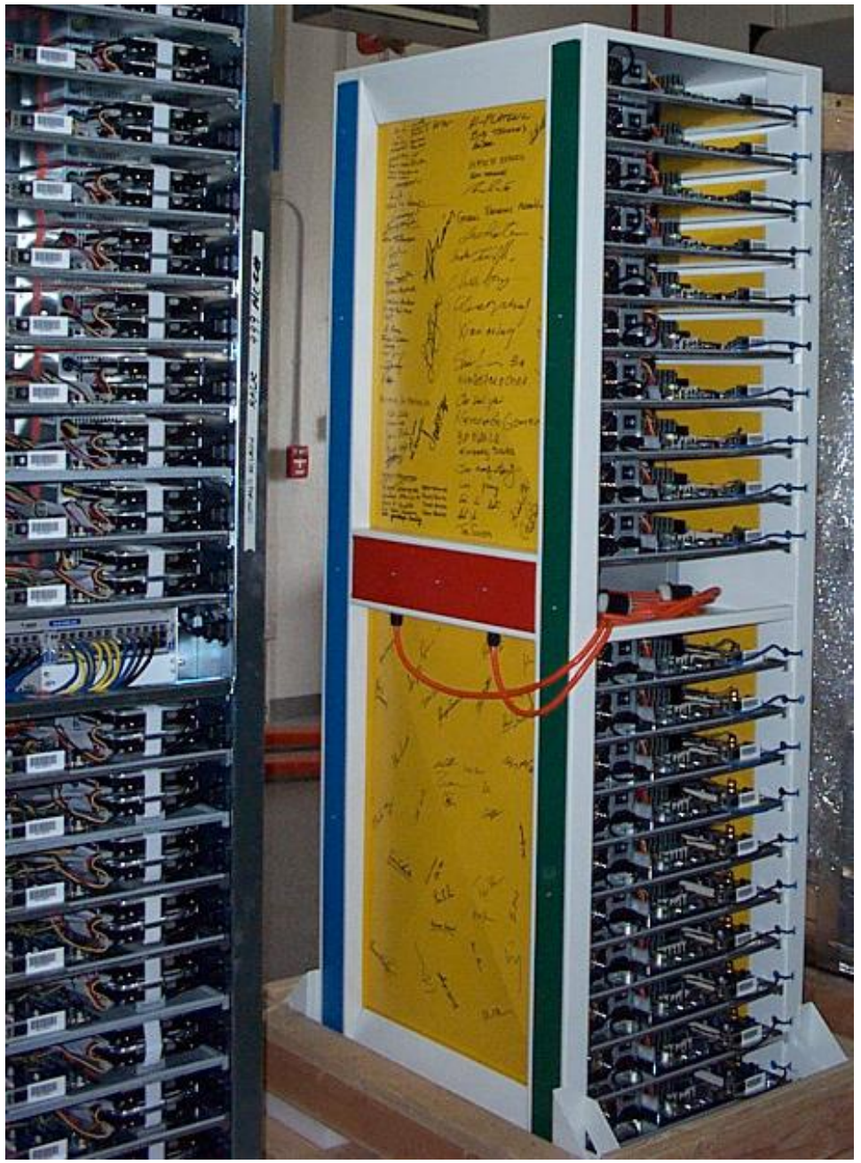
Google (new data center 2001)

3 days later

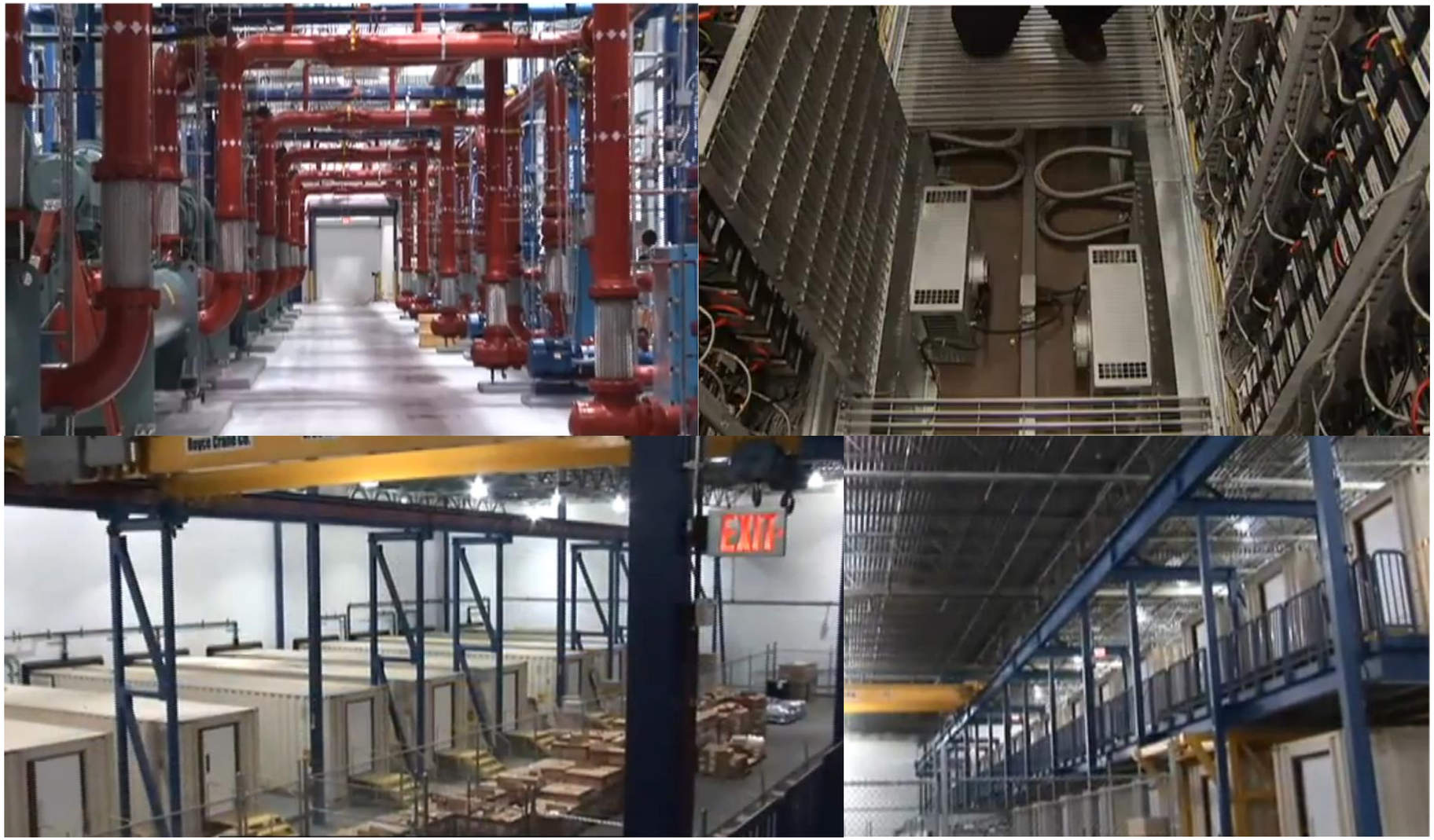
Recent Design

- In-house rack design
- PC-class motherboards
- Low-end storage and networking hardware
- Linux
- + in-house software





Container Datacenter



Container Datacenter

The Google "Cloud"

[NYT: June 8, 2006]

Google Cluster – Conclusions

- For a large scale web service system like Google
  - Design the algorithm which can be easily parallelized
  - Design the architecture using replication to achieve distributed computing/storage and fault tolerance
  - Be aware of the power problem which significantly restricts the use of parallelism

- Several key pieces of infrastructure for search systems:
  - GFS
  - MapReduce
  - BigTable

Google Cluster – References

1. Luiz André Barroso , Jeffrey Dean , Urs Hölzle, Web Search for a Planet: The Google Cluster Architecture, IEEE Micro, v.23 n.2, p.22-28, March 2003 [doi>10.1109/MM.2003.1196112]
2. S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," Proc. Seventh World Wide Web Conf. (WWW7), International World Wide Web Conference Committee (IW3C2), 1998, pp. 107-117.
3. "TPC Benchmark C Full Disclosure Report for IBM eserver xSeries 440 using Microsoft SQL Server 2000 Enterprise Edition and Microsoft Windows .NET Datacenter Server 2003, TPC-C Version 5.0," http://www.tpc.org/results/FDR/TPCC/ibm.x4408way.c5.fdr.02110801.pdf.
4. D. Marr et al., "Hyper-Threading Technology Architecture and Microarchitecture: A Hypertext History," Intel Technology J., vol. 6, issue 1, Feb. 2002.
5. L. Hammond, B. Nayfeh, and K. Olukotun, "A Single-Chip Multiprocessor," Computer, vol. 30, no. 9, Sept. 1997, pp. 79-85.
6. L.A. Barroso et al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," Proc. 27th ACM Int'l Symp. Computer Architecture, ACM Press, 2000, pp. 282-293.
7. L.A. Barroso, K. Gharachorloo, and E. Bugnion, "Memory System Characterization of Commercial Workloads," Proc. 25th ACM Int'l Symp. Computer Architecture, ACM Press, 1998, pp. 3-14.

Change to Caffeine

[Paul Krzyzanowski, 2012, Rutgers]

- In 2010, Google remodeled its search infrastructure
- Old system
  - Based on MapReduce (on GFS) to generate index files
  - Batch process: next phase of MapReduce cannot start until first is complete
    - Web crawling → MapReduce → propagation
  - Initially, Google updated its index every 4 months. Around 2000, it reindexed and propagated changes every month
    - Process took about 10 days
    - Users hitting different servers might get different results
- New system, named *Caffeine*
  - Fully incremental system: Based on BigTable running on GFS2
  - Support indexing many more documents: ~100 petabytes
  - High degree of interactivity: web crawlers can update tables dynamically
  - Analyze web continuously in small chunks
    - Identify pages that are likely to change frequently
  - BTW, MapReduce is not dead. Caffeine uses it in some places, as do lots of other services.

GFS to GFS2

[Paul Krzyzanowski, 2012, Rutgers]

- GFS was designed with MapReduce in mind
  - But found lots of other applications
  - Designed for batch-oriented operations
- Problems
  - Single *master node* in charge of chunkservers
  - All info (metadata) about files is stored in the master's memory – limits total number of files
  - Problems when storage grew to tens of petabytes (10^{12} bytes)
  - Automatic failover added (but still takes 10 seconds)
  - Designed for high throughput but delivers high latency: master can become a bottleneck
  - Delays due to recovering from a failed replica chunkserver delay the client
- GFS2
  - Distributed masters
  - Support smaller files: chunks go from 64 MB to 1 MB
  - Designed specifically for BigTable (does not make GFS obsolete)

More References

[Paul Krzyzanowski, 2012, Rutgers]

- *Web Search for a Planet: The Google Cluster Architecture*
  Luiz André Barroso, Jeffrey Dean, Urs Hölzle Google, Inc.
  research.google.com/archive/googlecluster.html

- *Our new search index: Caffeine*
  - The Official Google Blog
  - http://googleblog.blogspot.com/2010/06/our-new-search-index-caffeine.html
- *GFS: Evolution on Fast-forward*
  - Marshall Kirk McKusick, Sean Qunlan
  - Association for Computing Machinery, August 2009
  - http://queue.acm.org/detail.cfm?id=1594206
- *Google search index splits with MapReduce*
  - Cade Metz
  - The Register, September 2010
  - http://Sept/2009/08/12/google_file_system_part_deux/
- *Google File System II: Dawn of the Multiplying Master Nodes*
  - Cade Metz
  - The Register, August 2009
  - http://www.theregister.co.uk/2010/09/09/google_caffeine_explained/
- *Exclusive: How Google's Algorithm Rules the Web*
  - Steven Levy
  - Wired Magazine, March 2010
  - http://www.wired.com/magazine/2010/02/ff_google_algorithm/all/1

B. Parallel Programming

- Why is Parallel Programming so hard?
  - Conscious mind is inherently sequential
  - (sub-conscious mind is extremely parallel)
- Identifying parallelism in the problem
- Expressing parallelism to the hardware
- Effectively utilizing parallel hardware
  - Balancing work
  - Coordinating work
- Debugging parallel algorithms

B.1. Finding Parallelism

1. Functional parallelism
   - Car: {engine, brakes, entertain, nav, …}
   - Game: {physics, logic, UI, render, …}
   - Signal processing: {transform, filter, scaling, …}
2. Automatic extraction
   - Decompose serial programs
3. Data parallelism
   - Vector, matrix, DB table, pixels, …
4. Request parallelism
   - Web, shared database, telephony, …

1. Functional Parallelism

1. Functional parallelism
   - Car: {engine, brakes, entertain, nav, …}
   - Game: {physics, logic, UI, render, …}
   - Signal processing: {transform, filter, scaling, …}
- Relatively easy to identify and utilize
- Provides small-scale parallelism
   - 3x-10x
- Balancing stages/functions is difficult

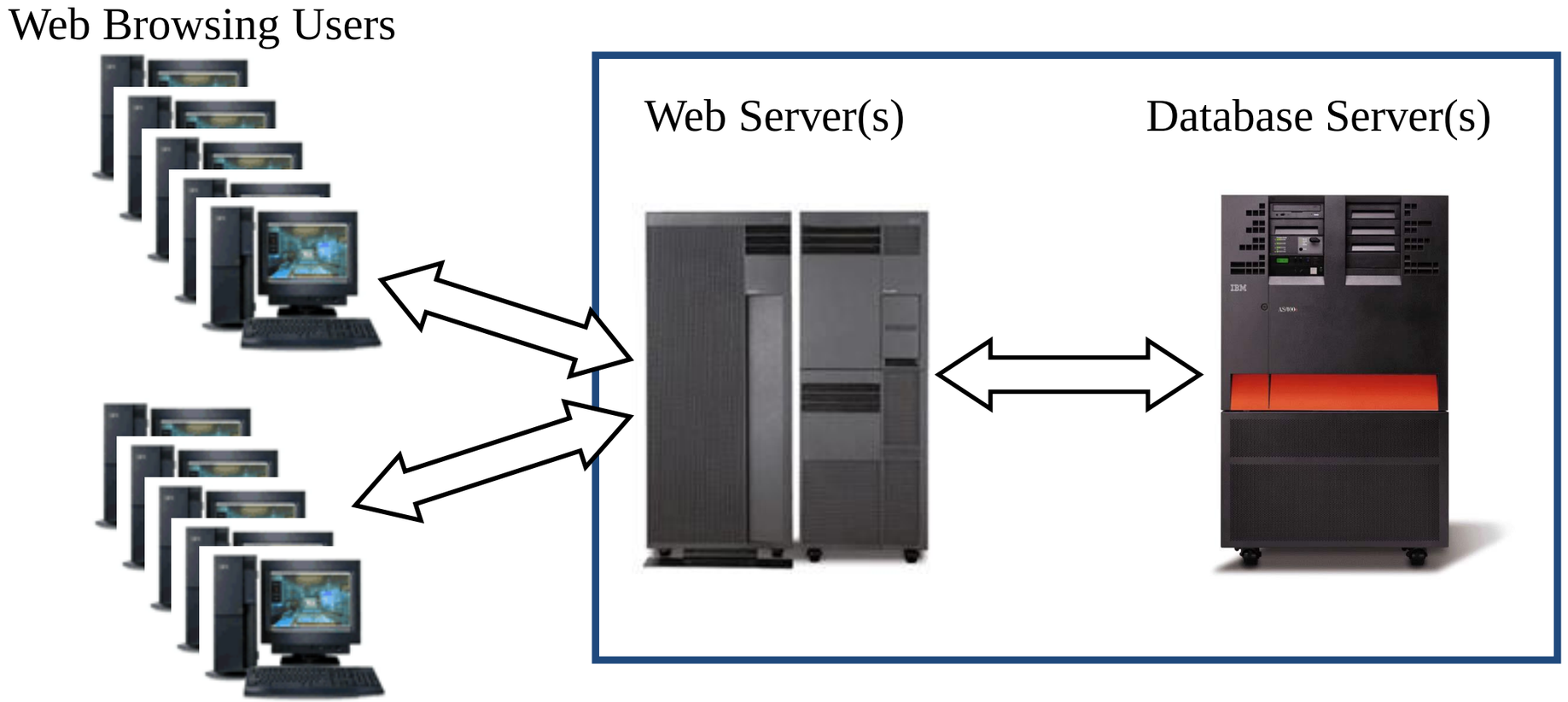
2. Automatic Extraction

2. Automatic extraction
   - Decompose serial programs
- Works well for certain application types
   - Regular control flow and memory accesses
- Difficult to guarantee correctness in all cases
   - Ambiguous memory dependences
   - Requires speculation, support for recovery
- Degree of parallelism
   - Large (1000x) for *easy* cases
   - Small (3x-10x) for *difficult* cases

3. Data Parallelism

3. Data parallelism
   - Vector, matrix, DB table, pixels, …
- Large data => significant parallelism
- Many ways to express parallelism
  - Vector/SIMD
  - Threads, processes, shared memory
  - Message-passing
- Challenges:
  - Balancing & coordinating work
  - Communication vs. computation at scale

4. Request Parallelism



- Multiple users => significant parallelism
- Challenges
  - Synchronization, communication, balancing work

Balancing Work

- Amdahl's parallel phase f: all processors busy
- If not perfectly balanced
  - (1-f) term grows (f not fully parallel)
  - Performance scaling suffers
- Manageable for data & request parallel apps
- Very difficult problem for other two:
  - Functional parallelism
  - Automatically extracted

Coordinating Work

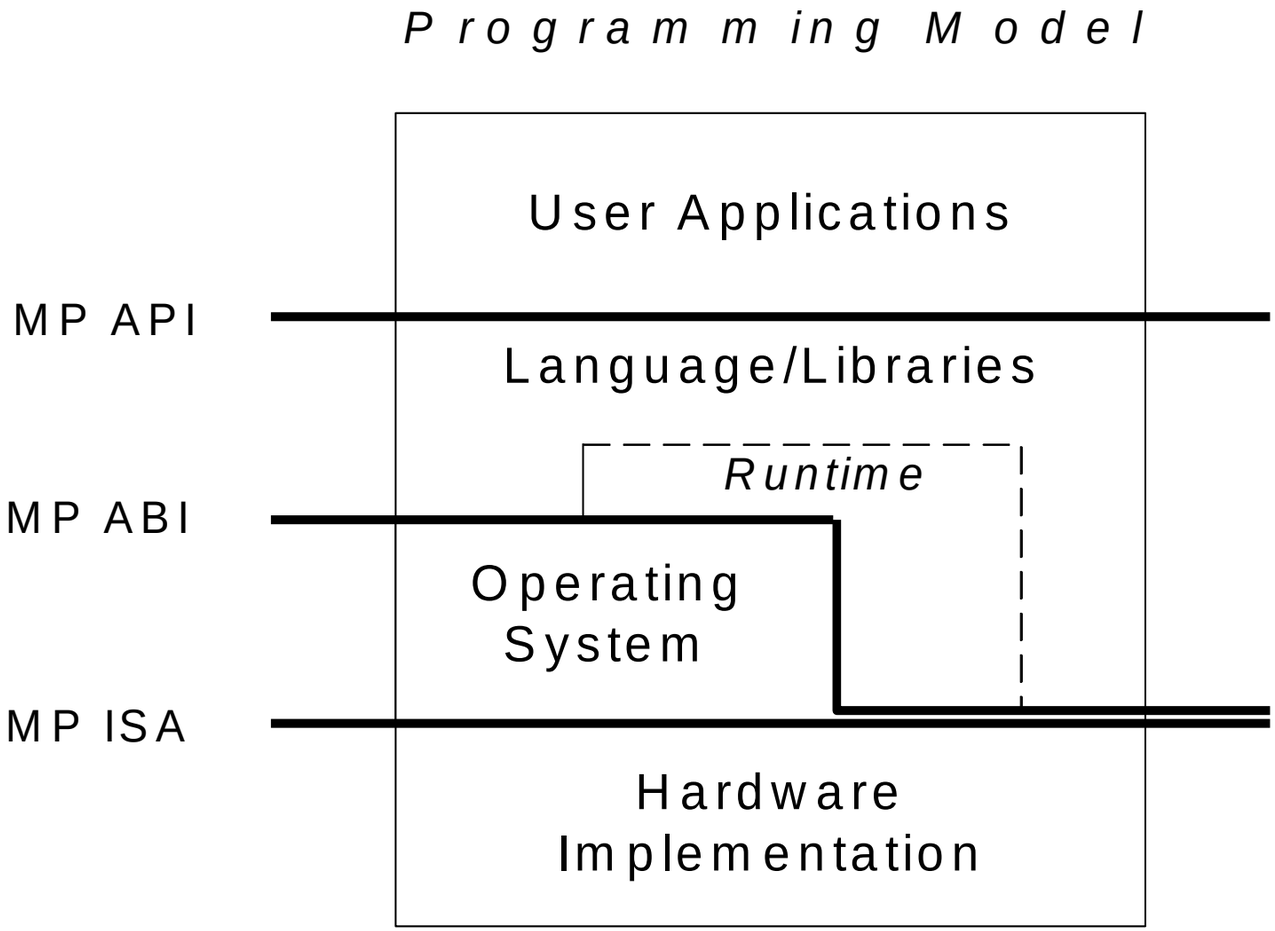
- Synchronization & Communication
  - Some data somewhere is shared
  - Coordinate/order updates and reads
  - Otherwise → chaos
- Traditionally: locks and mutual exclusion
  - Hard to get right, even harder to tune for performance
- Research into practice: Transactional Memory
  - Programmer: Declare potential conflict
  - Hardware and/or software: speculate & check
  - Commit or roll back and retry

Expressing Parallelism

- SIMD – Cray-1 case study (later)
  - MMX, SSE/SSE2/SSE3/SSE4, AVX at small scale
- SPMD – GPGPU model (later)
  - All processors execute same program on disjoint data
  - Loose synchronization vs. rigid lockstep of SIMD
- MIMD – most general (this lecture)
  - Each processor executes its own program
- Expressed through standard interfaces
  - API, ABI, ISA

MP ("Multiprocessing") Interfaces

- *Levels of abstraction* enable complex system designs (such as MP computers)
- Fairly natural extensions of uniprocessor model
  - Historical evolution



B.2. Programming Models

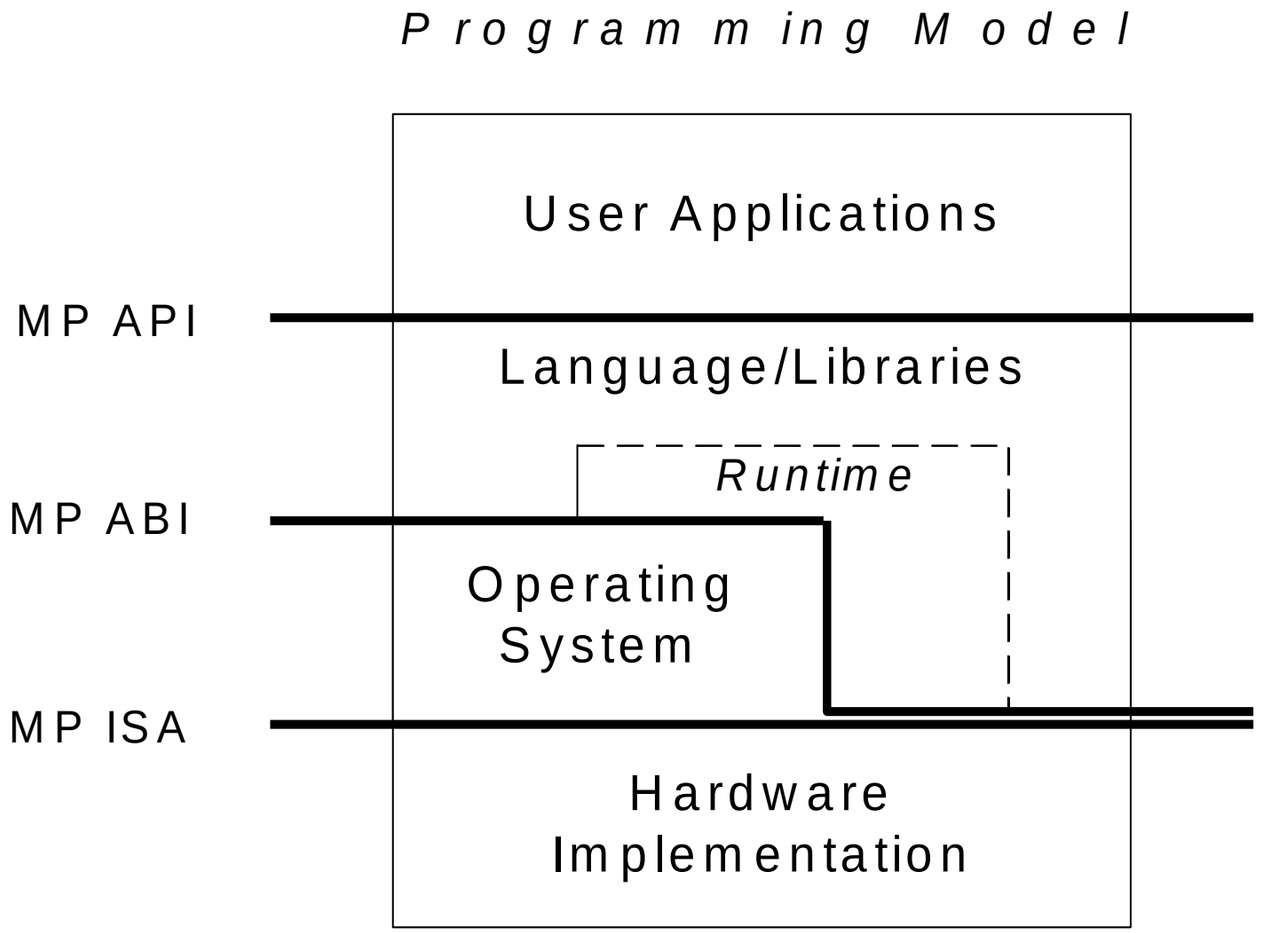
- High level paradigms for expressing an algorithm
  - Examples:
    - Functional
    - Sequential, procedural
    - Shared memory
    - Message Passing
- Embodied in high level languages that support concurrent execution
  - Incorporated into HLL constructs
  - Incorporated as libraries added to existing sequential language
- Top level features:
  - For conventional models – **shared memory**, **message passing**
  - Multiple threads are conceptually visible to programmer
  - Communication/synchronization are visible to programmer

Application Programming Interface (API)

- Interface where HLL programmer works
- High level language plus libraries
  - Individual libraries are sometimes referred to as an “API”
- User level runtime software is often part of API implementation
  - Executes procedures
  - Manages user-level state
- Examples:
  - C and pthreads
  - FORTRAN and MPI

Application Binary Interface (ABI)

- Program in API is compiled to ABI
- Consists of:
  - OS call interface
  - User level instructions (part of ISA)



Instruction Set Architecture (ISA)

- Interface between hardware and software
  - What the hardware implements
- Architected state
  - Registers
  - Memory architecture
- All instructions
  - May include parallel (SIMD) operations
  - Both non-privileged and privileged
- Exceptions (traps, interrupts)

Major Abstractions

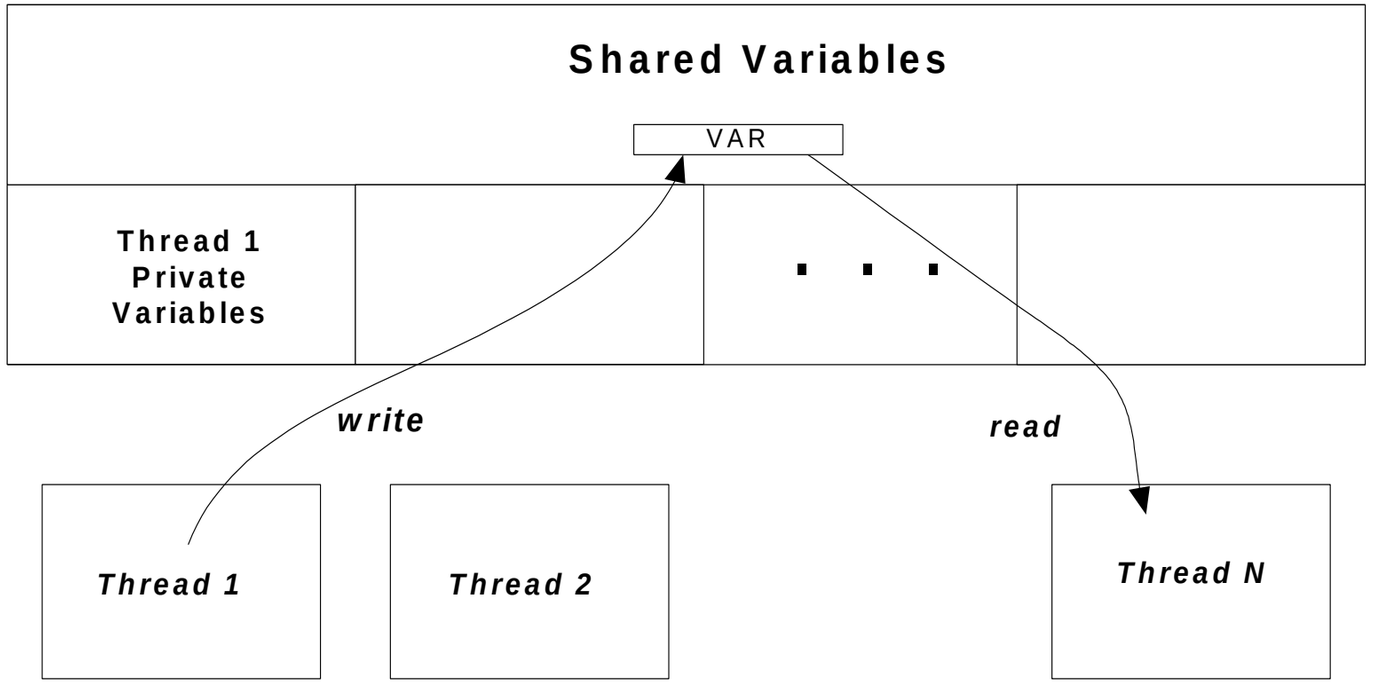
- For both Shared Memory & Message Passing (programming models)
- Processes and Threads (parallelism expressed)
  - ***Process:*** A shared address space and one or more threads of control
  - ***Thread:*** A program sequencer and private address space
  - ***Task***: Less formal term – part of an overall job
  - Created, terminated, scheduled, etc.
- Communication
  - Passing of data
- Synchronization
  - Communicating control information
  - To ensure reliable, deterministic communication

B.3. Shared Memory Model

- Shared Memory Model
  - API-level Processes, Threads
  - API-level Communication
  - API-level Synchronization
- Shared Memory Implementation
  - Implementing Processes, Threads at ABI/ISA levels
  - Implementing Communication at ABI/ISA levels
  - Implementing Synchronization at ABI/ISA levels

Shared Memory Model

- Flat shared memory or object heap
  - Synchronization via memory variables enables reliable sharing
- Single process
- Multiple threads per process
  - Private memory per thread
- Typically built on shared memory hardware system



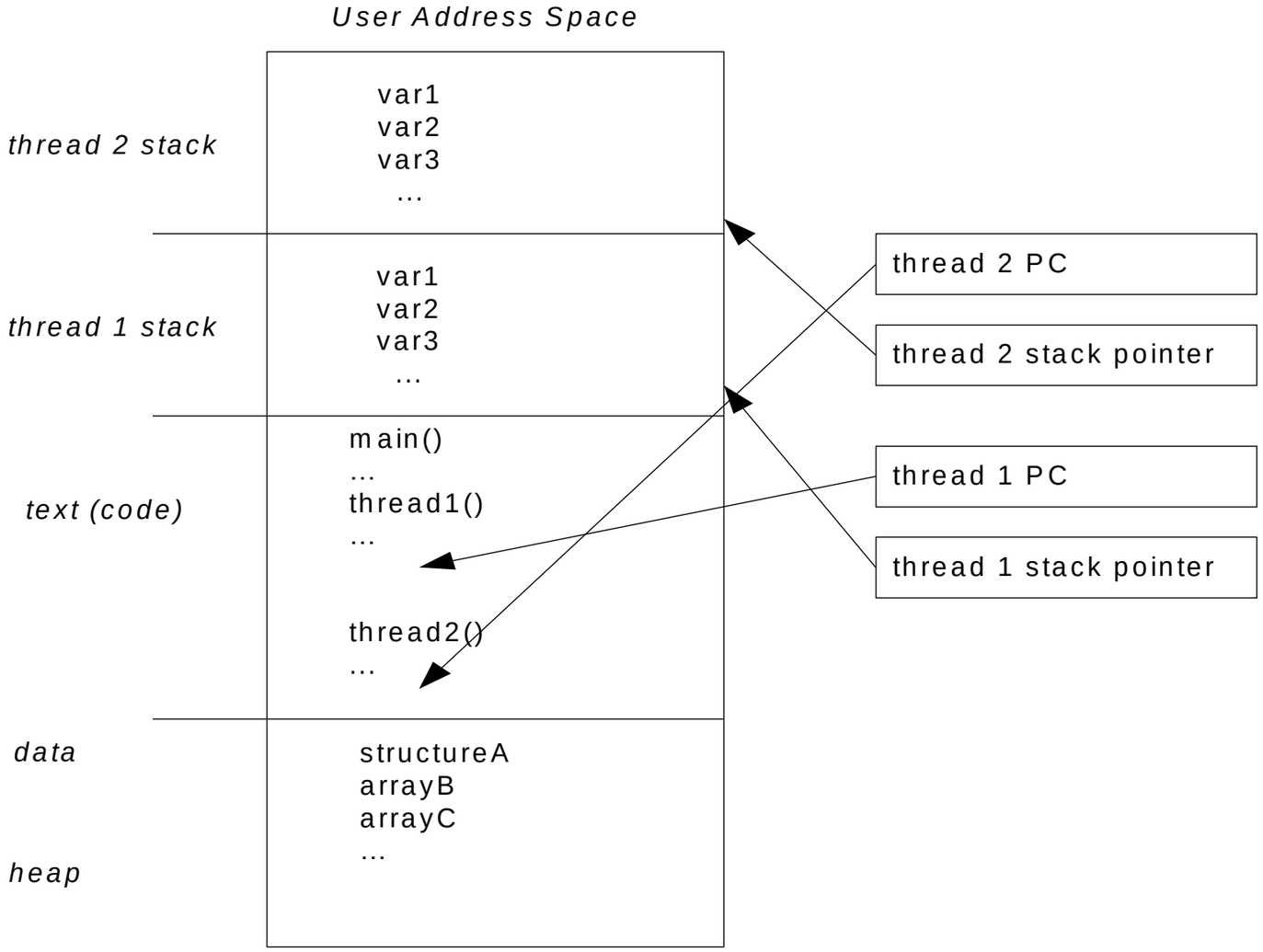
Threads and Processes

- Creation
  - generic -- Fork
    - (Unix forks a process, not a thread)
  - pthread_create(….*thread_function….)
    - creates new thread in current address space
- Termination
  - pthread_exit
    - or terminates when thread_function terminates
  - pthread_kill
    - one thread can kill another

Example

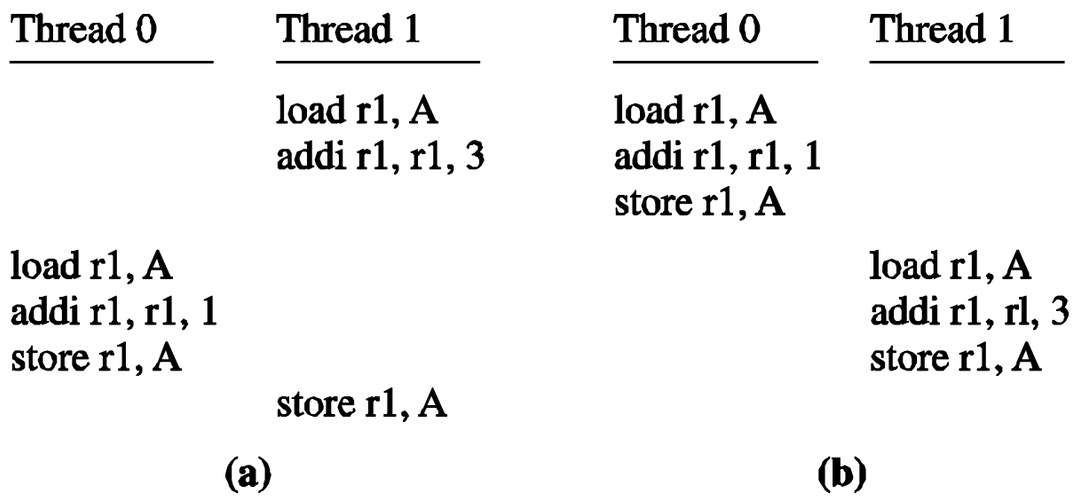
- Unix process with two threads

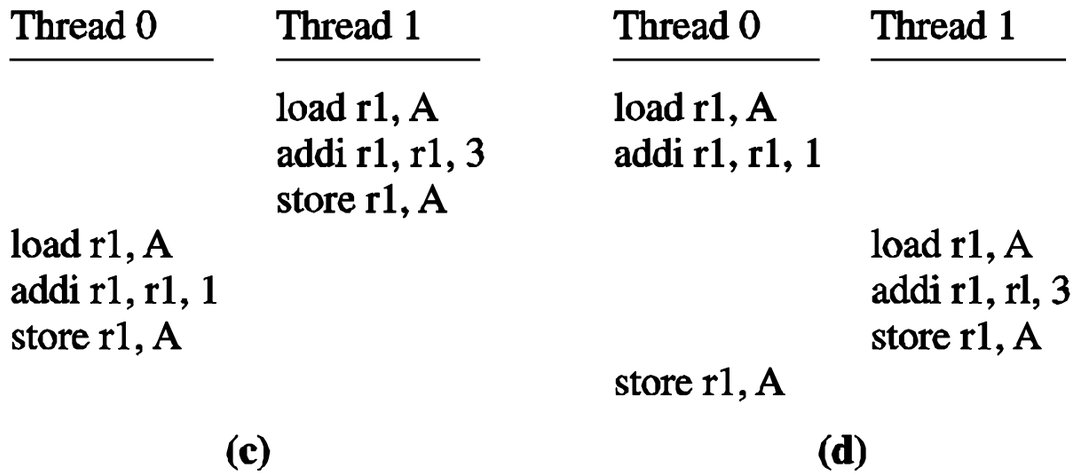
  (PC and stack pointer actually part of ABI/ISA implementation)



Shared Memory Communication

- Reads and writes to shared variables via normal language (assignment) statements (e.g. assembly load/store)





Shared Memory Synchronization

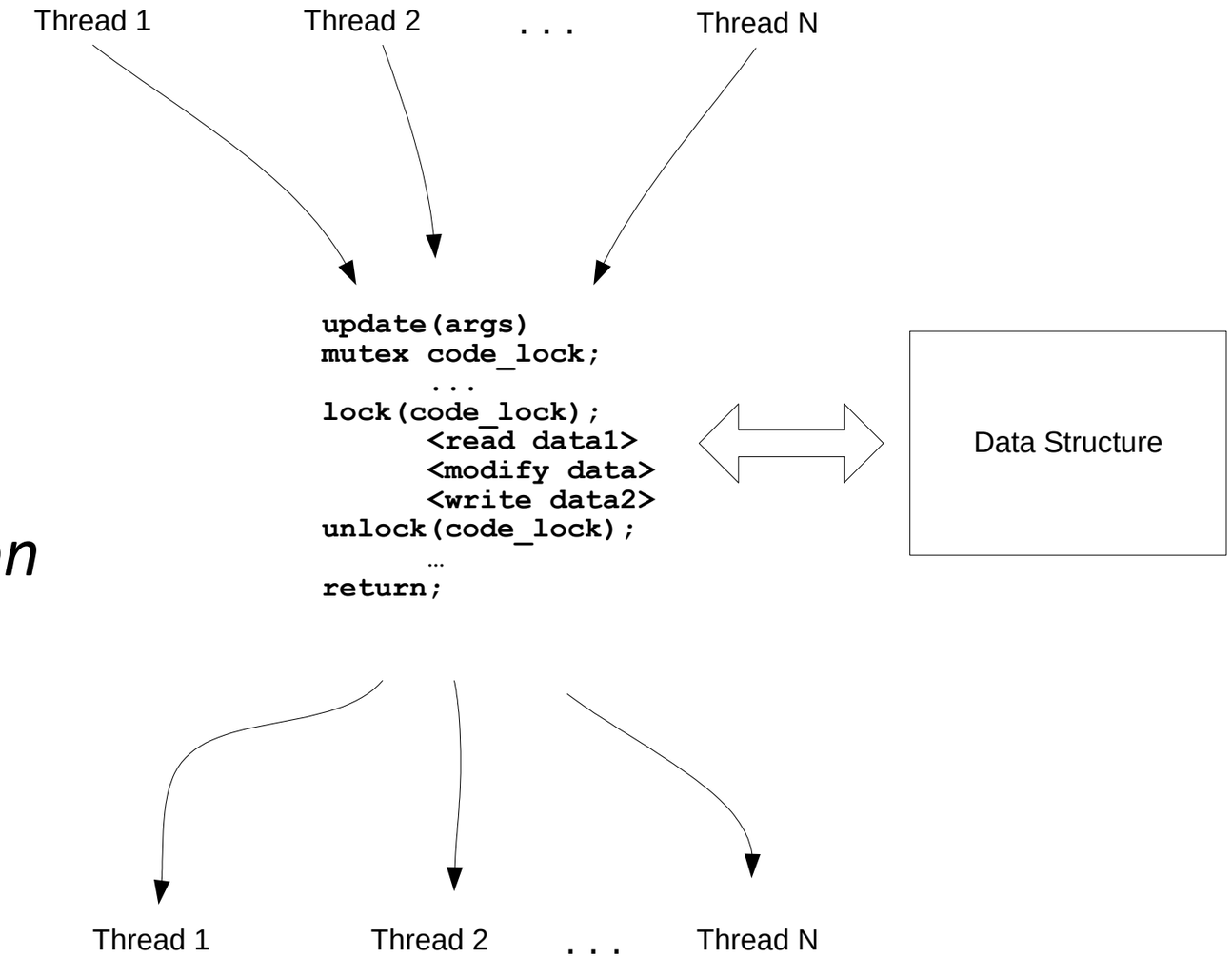
- What really gives shared memory programming its structure
- Usually explicit in shared memory model
  - Through language constructs or API
- Three major classes of synchronization
  - Mutual exclusion (mutex)
  - Point-to-point synchronization
  - Rendezvous
- Employed by *application design patterns*
  - *A general description or template for the solution to a commonly recurring software design problem.*

Mutual Exclusion (mutex)

- Assures that only one thread at a time can access a code or data region
- Usually done via *locks*
  - One thread acquires the lock
  - All other threads excluded until lock is released
- Examples
  - pthread_mutex_lock
  - pthread_mutex_unlock
- Two main application programming patterns
  - Code locking
  - Data locking

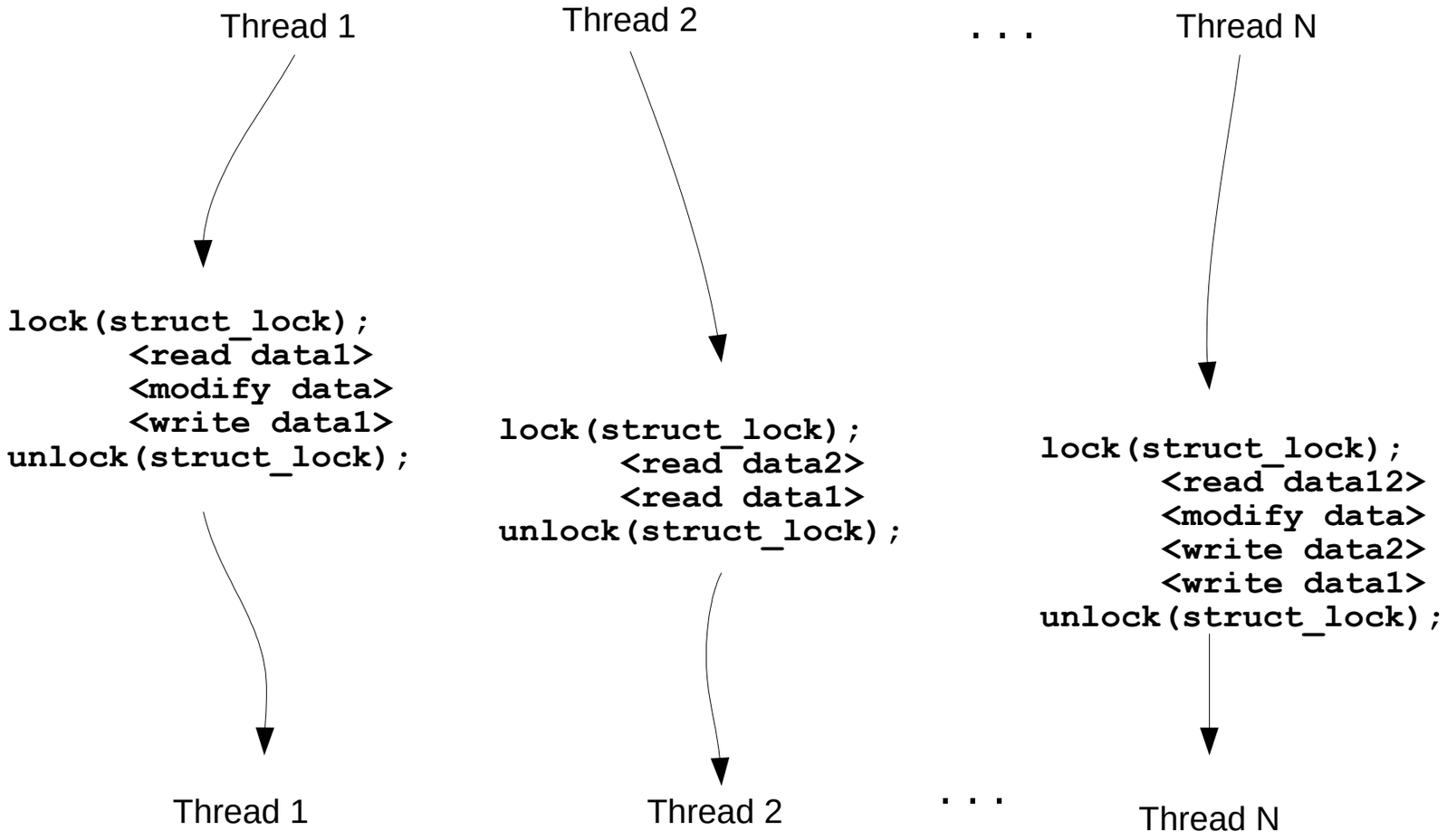
Code Locking

- Protect shared data by locking the code that accesses it
- Also called a *monitor* pattern
- Example of a *critical section*



Data Locking

- Protect shared data by locking data structure



Data Locking

- Preferred when data structures are read/written in combinations
- Example:

```
<thread 0>                <thread 1>                <thread 2>
Lock(mutex_struct1)       Lock(mutex_struct1)       Lock(mutex_struct2)
Lock(mutex_struct2)       Lock(mutex_struct3)       Lock(mutex_struct3)
   <access struct1>          <access struct1>          <access struct2>
   <access struct2>          <access struct3>          <access struct3>
Unlock(mutex_data1)       Unlock(mutex_data1)       Unlock(mutex_data2)
Unlock(mutex_data2)       Unlock(mutex_data3)       Unlock(mutex_data3)
```

Deadlock

- Data locking is prone to deadlock
  - If locks are acquired in an unsafe order
- Example:

<thread 0>
```
Lock(mutex_data1)
Lock(mutex_data2)
   <access data1>
   <access data2>
Unlock(mutex_data1)
Unlock(mutex_data2)
```

<thread 1>
```
Lock(mutex_data2)
Lock(mutex_data1)
   <access data1>
   <access data2
Unlock(mutex_data1)
Unlock (mutex_data2)
```

- Complexity
  - Disciplined locking order must be maintained, else deadlock
  - Also, composability problems
    - Locking structures in a nest of called procedures

Efficiency

- Lock Contention
  - Causes threads to wait
- Function of lock *granularity*
  - Size of data structure or code that is being locked
- Extreme Case:
  - "One big lock" model for multithreaded OSes
  - Easy to implement, but very inefficient
- Finer granularity
  + Less contention
  - More locks, more locking code
  - Perhaps more deadlock opportunities
- Coarser granularity
  - Opposite +/- of above

Point-to-Point Synchronization

- One thread signals another that a condition holds
  - Can be done via API routines
  - Can be done via normal load/stores
- Examples
  - pthread_cond_signal
  - pthread_cond_wait
    - suspends thread if condition not true
- Application program pattern
  - Producer/Consumer

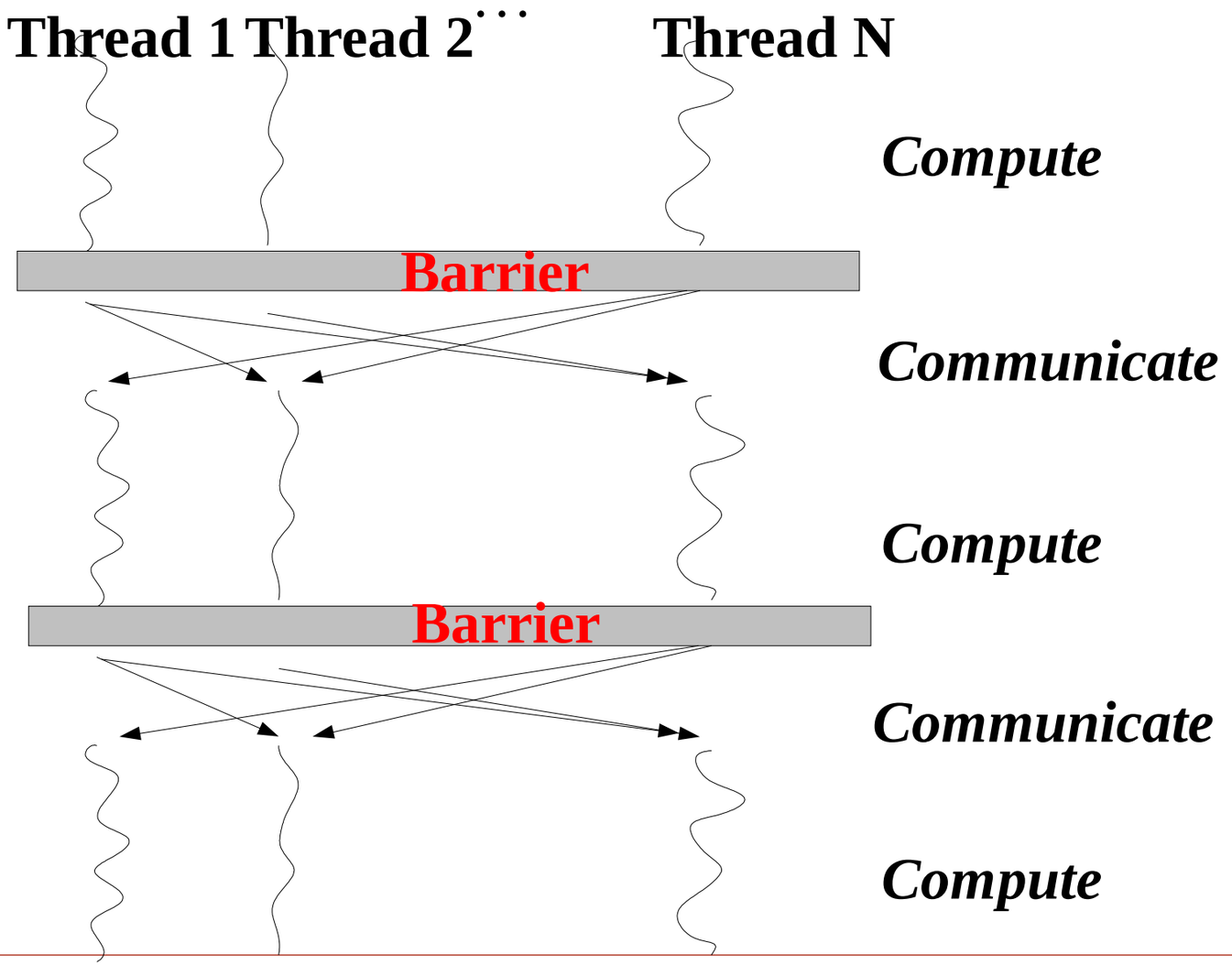
```
<Producer>
while (full == 1){}; wait
buffer = value;
full = 1;
```

```
<Consumer>
while (full == 0){}; wait
b = buffer;
full = 0;
```

Rendezvous

- Two or more cooperating threads must reach a program point before proceeding
- Examples
  - Wait for another thread at a join point before proceeding
    - example: pthread_join
  - Barrier synchronization
    - many (or all) threads wait at a given point
- Application program pattern
  - Bulk synchronous programming pattern

Bulk Synchronous Program Pattern

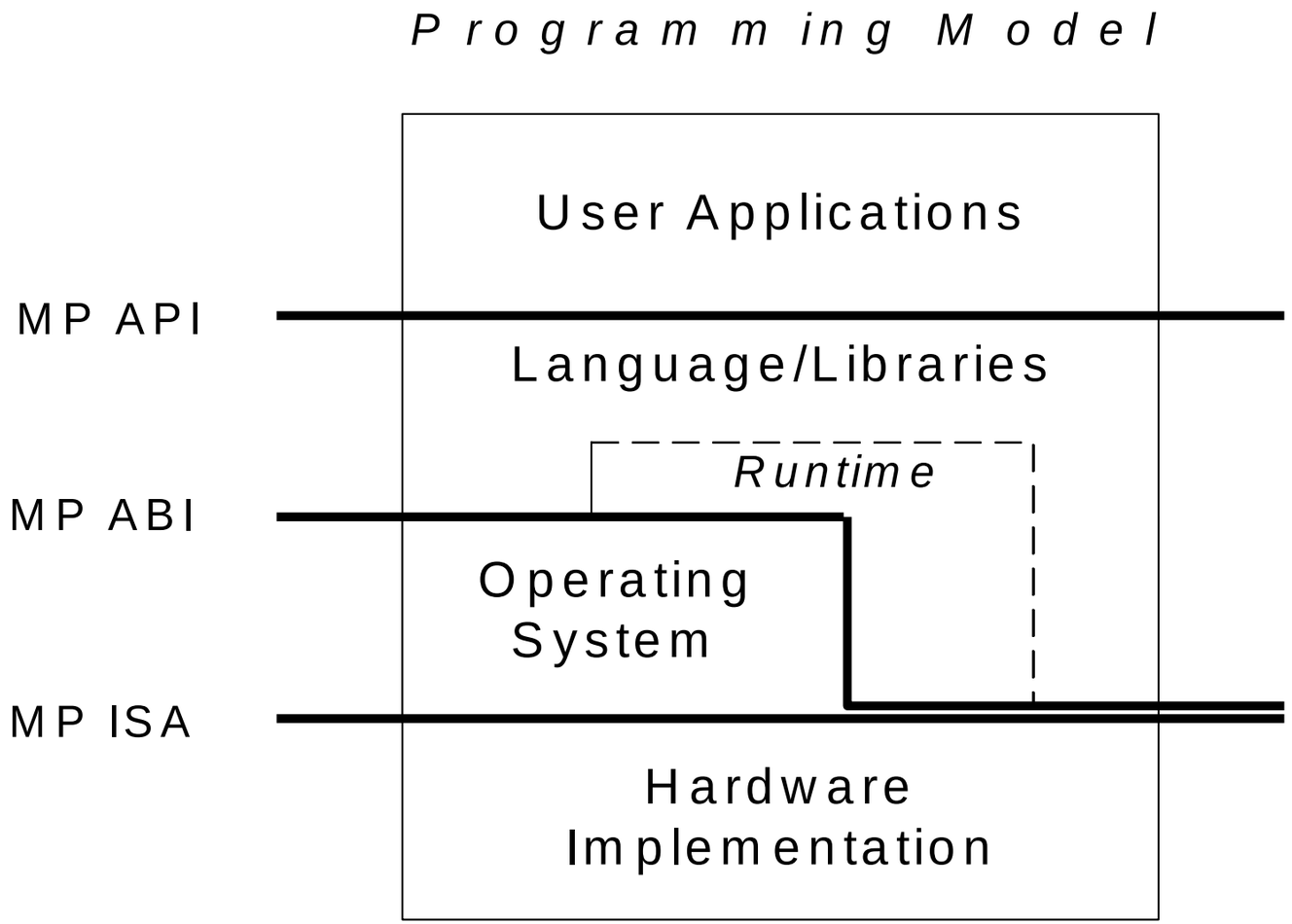


Summary: Synchronization and Patterns

- Mutex (mutual exclusion)
  - Code locking (monitors)
  - Data locking
- Point to point
  - Producer/consumer
- Rendezvous
  - Bulk synchronous

API Implementation

- Implemented at ABI and ISA level
  - OS calls
  - Runtime software
  - Special instructions



Processes and Threads

- Three models
  - OS processes
  - OS threads
  - User threads

OS Processes

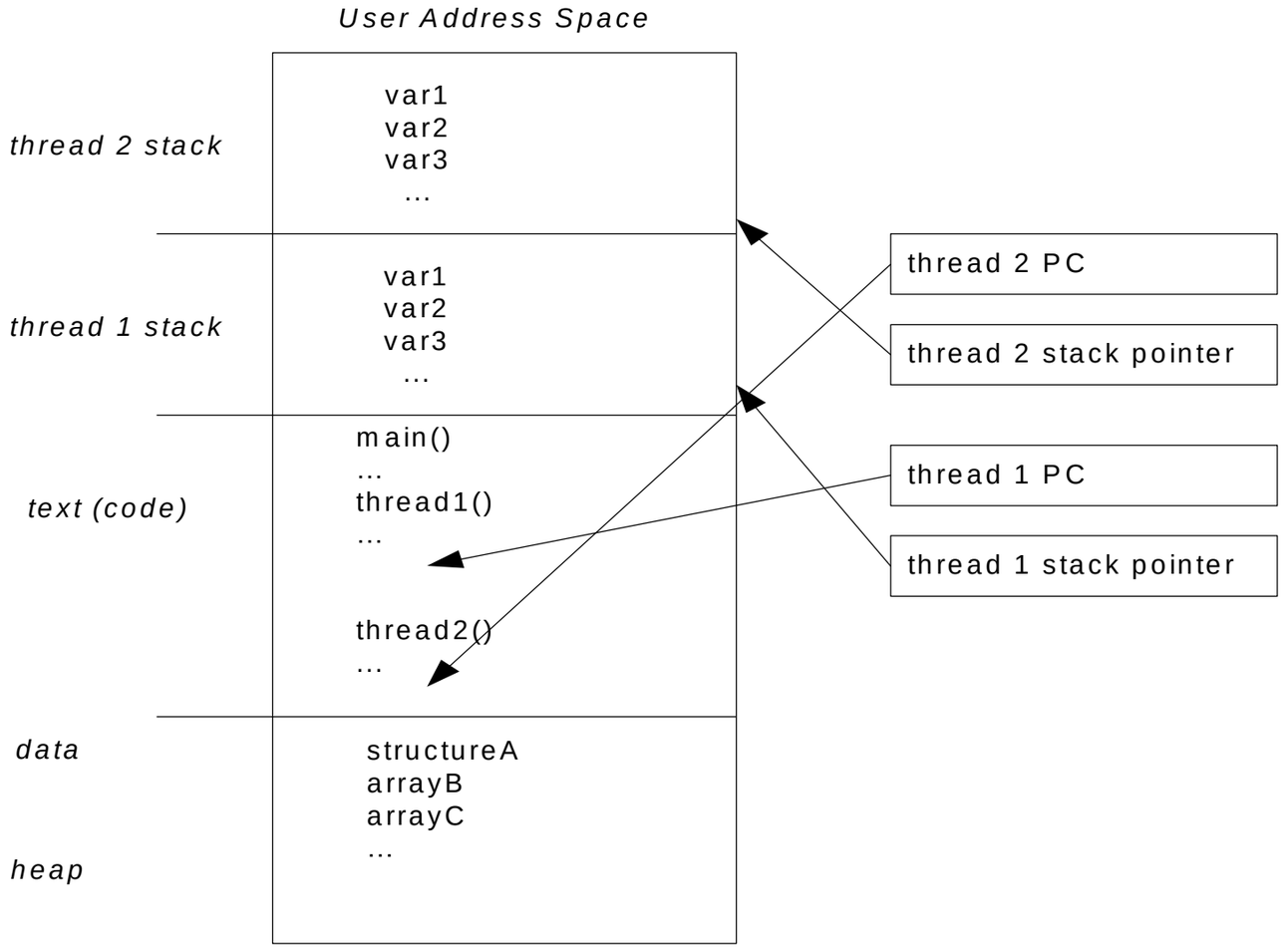
- Thread == Process
- Use OS fork to create processes
- Use OS calls to set up shared address space
- OS manages processes (and threads) via run queue
- Heavyweight thread switches
  - OS call followed by:
  - Switch address mappings
  - Switch process-related tables
  - Full register switch
- Advantage
  - Threads have protected private memory

OS (Kernel) Threads

- API pthread_create() maps to Linux clone()
  - Allows multiple threads sharing memory address space
- OS manages threads via run queue
- Lighter weight thread switch
  - Still requires OS call
  - OS switches architected register state and stack pointer

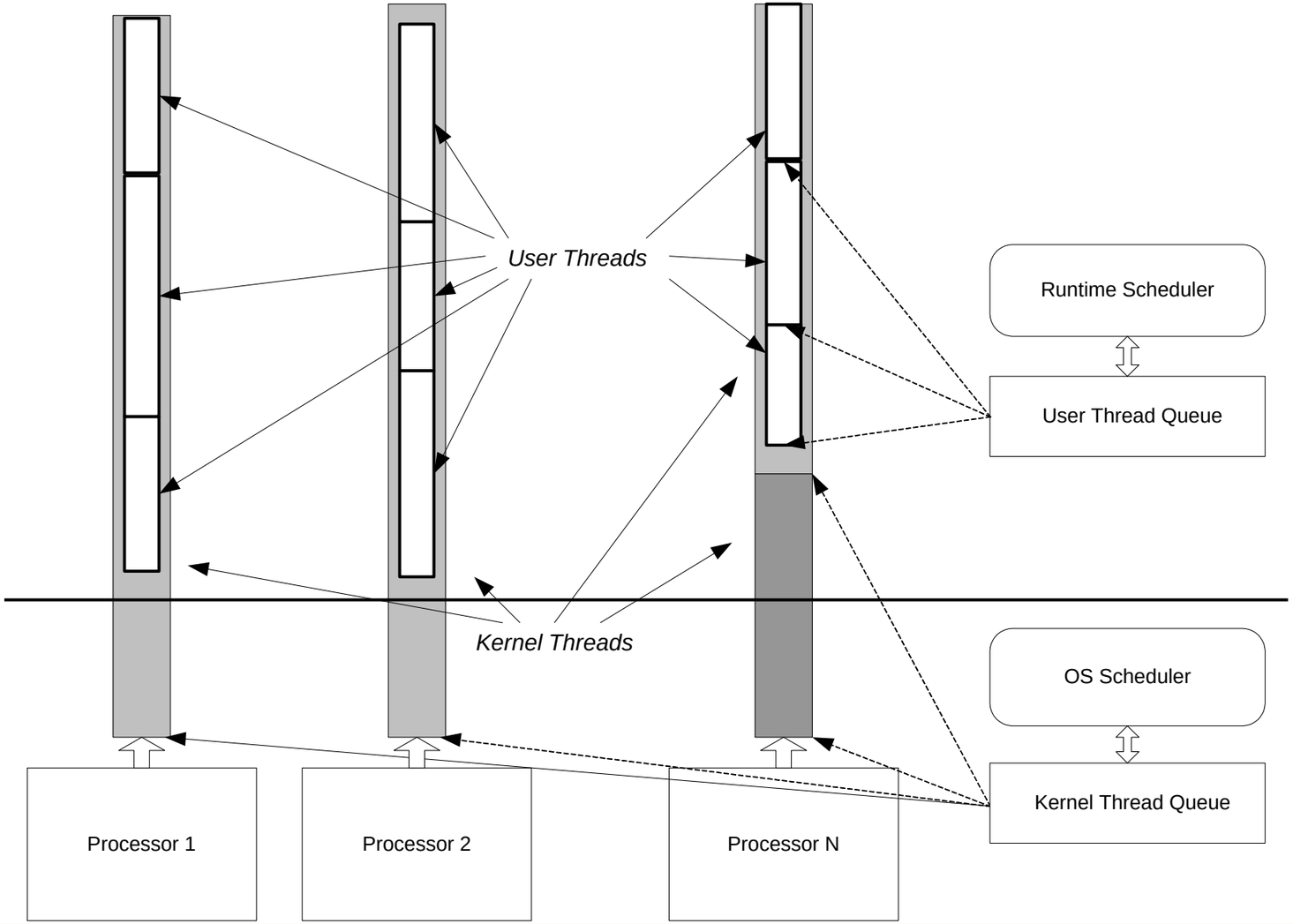
User Threads

- If memory mapping doesn't change, why involve OS at all?
- Runtime creates threads simply by allocating stack space
- Runtime switches threads via user level instructions
  - thread switch via jumps



Implementing User Threads

- Multiple kernel threads needed to get control of multiple hardware processors
- Create kernel threads (OS schedules)
- Create user threads that runtime schedules onto kernel threads



Communication

- *Easy*

  Just map high level access to variables to ISA level loads and stores
- *Except for*

  Ordering of memory accesses -- later

Synchronization

- Implement locks and rendezvous (barriers)
- Use loads and stores to implement lock:

```
<thread 0>
        .
        .
LAB1:   Load R1, Lock
        Branch LAB1 if R1==1
        Enter R1, 1
        Store Lock, R1
        .
        <critical section>
        .
        Enter R1, 0
        Store  Lock, R1
```

```
<thread 1>
        .
        .
LAB2:   Load R1, Lock
        Branch LAB2 if R1==1
        Enter R1,1
        Store Lock, R1
        .
        <critical section>
        .
        Enter R1, 0
        Store Lock, R1
```

Lock Implementation

- *Does not work*
- Violates mutual exclusion if both threads attempt to lock at the same time
  - In practice, may work *most* of the time…
  - Leading to an unexplainable system hang every few days

<thread 0>

```
        .
        .
LAB1:   Load R1, Lock
        Branch LAB1 if R1==1
        Enter R1, 1
        Store Lock, R1
```

<thread 1>

```
        .
        .
LAB2:   Load R1, Lock
        Branch LAB2 if R1==1
        Enter R1,1
        Store Lock, R1
```

Lock Implementation

- Reliable locking can be done with *atomic* read-modify-write instruction
- Example: test&set
  - read lock and write a one
  - some ISAs also set CCs (test)

<thread 1>

```
        .
LAB1:   Test&Set R1, Lock
        Branch LAB1 if R1==1
        .
        <critical section>
        .
        Reset Lock
```

<thread 2>

```
        .
LAB2:   Test&Set R1, Lock
        Branch LAB2 if R1==1
        .
        <critical section>
        .
        Reset Lock
```

Atomic Read-Modify-Write

- Many such instructions have been used in ISAs

```
Test&Set(reg,lock)
reg ←mem(lock);
mem(lock) ← 1;
```

```
Fetch&Add(reg,value,sum)
reg ← mem(sum);
mem(sum)←mem(sum)+value;
```

```
Swap(reg,opnd)
temp←mem(opnd);
mem(opnd)← reg;
reg ← temp
```

- More-or-less equivalent
  - One can be used to implement the others
  - Implement Fetch&Add with Test&Set:

```
try:  Test&Set(lock);
      if lock == 1 go to try;
      reg ←mem(sum);
      mem(sum) ← reg+value;
      reset (lock);
```

Lock Efficiency

- Spin Locks
  - tight loop until lock is acquired

```
LAB1:  Test&Set R1, Lock
       Branch LAB1 if R1==1
```

- Inefficiencies:
  - Memory/Interconnect resources, spinning on read/writes
  - With a cache-based systems,
    writes $\Rightarrow$ lots of coherence traffic
  - Processor resource
    - not executing useful instructions

Efficient Lock Implementations

- Test&Test&Set
  - spin on check for unlock only, then try to lock
  - with cache systems, all reads can be local
    - no bus or external memory resources used

```
test_it: load        reg, mem(lock)
         branch      test_it if reg==1
lock_it: test&set    reg, mem(lock)
         branch      test_it if reg==1
```

- Test&Set with Backoff
  - Insert delay between test&set operations (not too long)
  - Each failed attempt ⇒ longer delay
    (Like ethernet collision avoidance)

Efficient Lock Implementations

- Solutions just given save memory/interconnect resource
  - Still waste processor resource
- Use runtime to suspend waiting process
  - Detect lock
  - Place on wait queue
  - Schedule another thread from run queue
  - When lock is released move from wait queue to run queue

Point-to-Point Synchronization

- *Can* use normal variables as flags

```
while (full ==1){}  ;spin          while (full == 0){} ;spin
a = value;                         b = value;
full = 1;                          full = 0;
```

- Assumes sequential consistency (later)
  - Using normal variables may cause problems with relaxed consistency models
- May be better to use special opcodes for flag set/clear

Barrier Synchronization

- Uses a lock, a counter, and a flag
  - lock for updating counter
  - flag indicates all threads have incremented counter

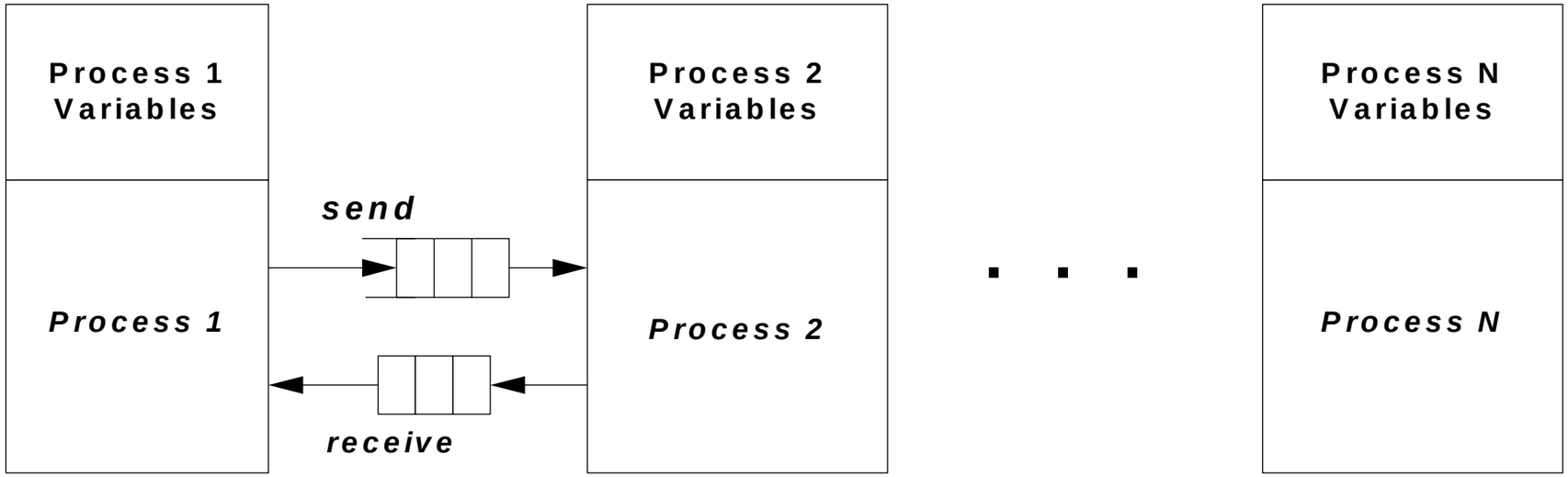
```
Barrier (bar_name, n) {
    Lock (bar_name.lock);
    if (bar_name.counter = 0)  bar_name.flag = 0;
    mycount = bar_name.counter++;
    Unlock (bar_name.lock);
    if (mycount == n) {
          bar_name.counter = 0;
          bar_name.flag = 1;
    }
    else  while(bar_name.flag = 0) {}; /* busy wait */
}
```

B.4. Message Passing Model

- Message Passing Model
  - API-level Processes, Threads
  - API-level Communication
  - API-level Synchronization
- Message Passing Implementation
  - Implementing Processes, Threads at ABI/ISA levels
  - Implementing Communication at ABI/ISA levels
  - Implementing Synchronization at ABI/ISA levels

Message Passing

- Multiple processes (or threads)
- Logical data partitioning
  - No shared variables
- Message Passing
  - Threads of control communicate by sending and receiving messages
  - May be implicit in language constructs
  - More commonly explicit via API



MPI – Message Passing Interface API

- A widely used standard
  - For a variety of distributed memory systems
    - SMP Clusters, workstation clusters, MPPs, heterogeneous systems
- Also works on Shared Memory MPs
  - Easy to emulate distributed memory on shared memory HW
- Can be used with a number of high level languages

Processes and Threads

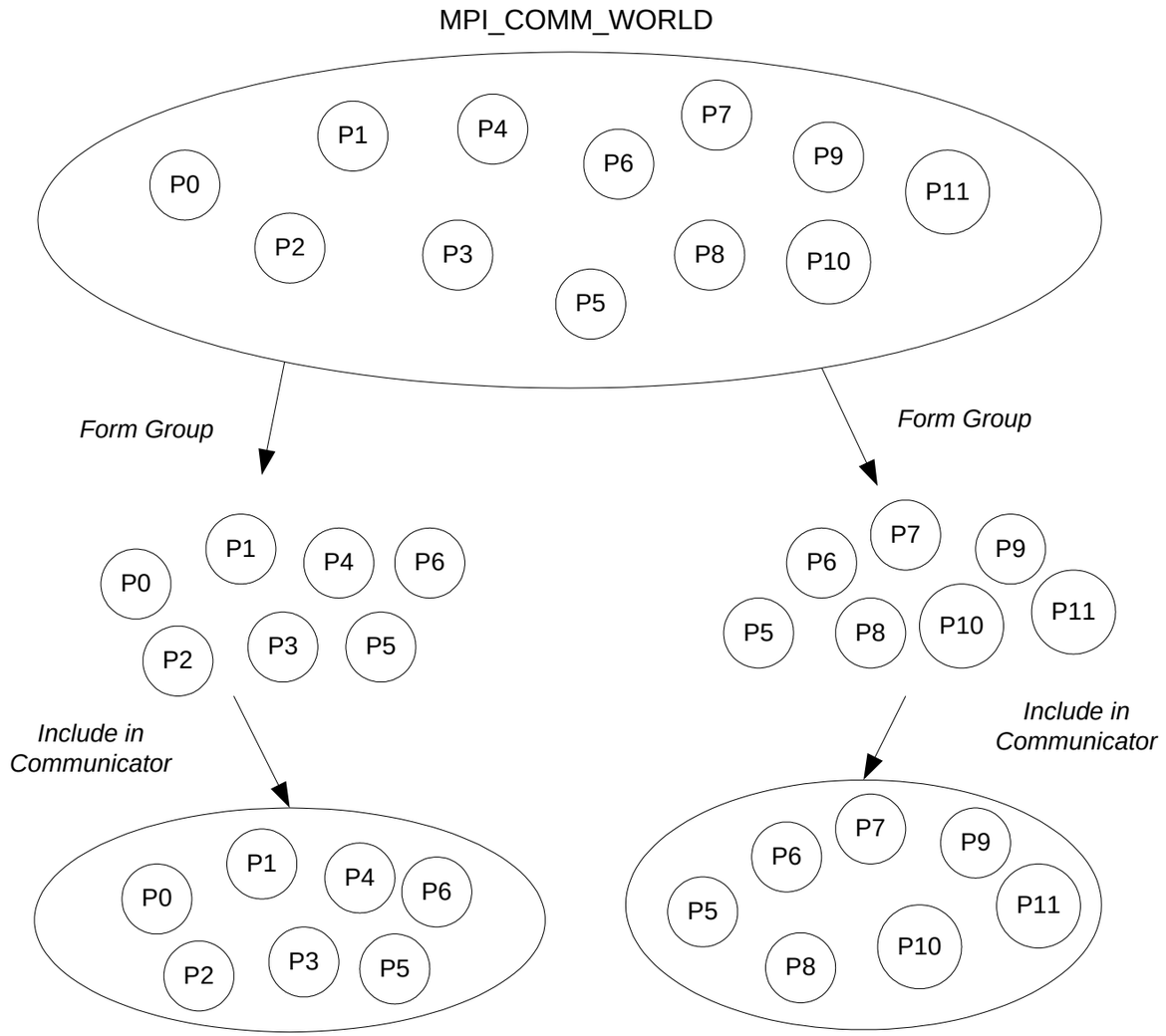
- Lots of flexibility (advantage of message passing)
  - 1) Multiple threads sharing an address space
  - 2) Multiple processes sharing an address space
  - 3) Multiple processes with different address spaces and different OSes
- 1) and 2) are easily implemented on shared memory hardware (with single OS)
  - Process and thread creation/management similar to shared memory
- 3) probably more common in practice
  - Process creation often external to execution environment; e.g. shell script
  - Hard for user process on one system to create process on another OS

Process Management

- Processes are given identifiers (PIDs)
  - "rank" in MPI
- Process can acquire own PID
- Operations can be conditional on PID
- Message can be sent/received via PIDs

Process Management

- Organize into groups
  - For collective management and communication



Communication and Synchronization

- Combined in the message passing paradigm
  - Synchronization of messages part of communication semantics
- Point-to-point communication
  - From one process to another
- Collective communication
  - Involves groups of processes
  - e.g., broadcast

Point to Point Communication

- Use sends/receives primitives
- Send(RecProc, SendBuf,…)
  - RecProc is destination (wildcards may be used)
  - SendBuf names buffer holding message to be sent
- Receive(SendProc, RecBuf,…)
  - SendProc names sending process (wildcards may be used)
  - RecBuf names buffer where message should be placed

MPI Examples

- MPI_Send(buffer,count,type,dest,tag,comm)
  - buffer – address of data to be sent
  - count – number of data items
  - type – type of data items
  - dest – rank of the receiving process
  - tag – arbitrary programmer-defined identifier
    - tag of send and receive must match
  - comm – communicator number
- MPI_Recv(buffer,count,type,source,tag,comm,status)
  - buffer – address of data to be sent
  - count – number of data items
  - type – type of data items
  - source – rank of the sending process; may be a wildcard
  - tag – arbitrary programmer-defined identifier; may be a wildcard
    - tag of send and receive must match
  - comm – communicator number
  - status – indicates source, tag, and number of bytes transferred

Message Synchronization

- After a send or receive is executed…
  - *Has message actually been sent? or received?*
- Asynchronous versus Synchronous
  - Higher level concept
- Blocking versus non-Blocking
  - Lower level – depends on buffer implementation
    - *but is reflected up into the API*

Synchronous vs. Asynchronous

- Synchronous Send
  - Stall until message has actually been received
  - Implies a message acknowledgement from receiver to sender
- Synchronous Receive
  - Stall until message has actually been received
- Asynchronous Send and Receive
  - Sender and receiver can proceed regardless
  - Returns *request handle* that can be tested for message receipt
  - Request handle can be tested to see if message has been sent/received

Blocking vs. Non-Blocking

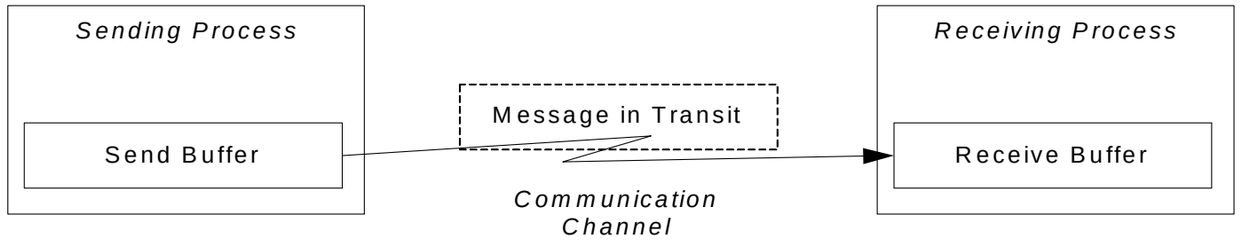
- *Blocking send* blocks if send buffer is not available for new message
- *Blocking receive* blocks if no message in its receive buffer
- Non-blocking versions don't block…
- Operation depends on buffering in implementation

Blocking vs. Non-Blocking

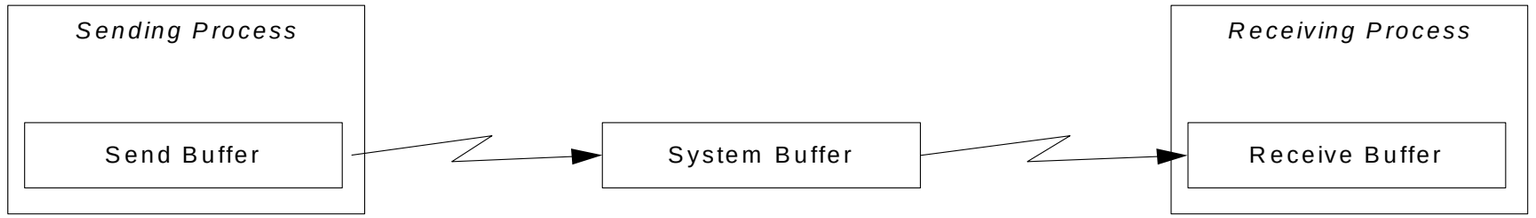
- Buffer implementations

a) Message goes directly from sender to receiver reduces copying time

b) Message is buffered by system in between may free up send buffer sooner (less blocking)



(a)



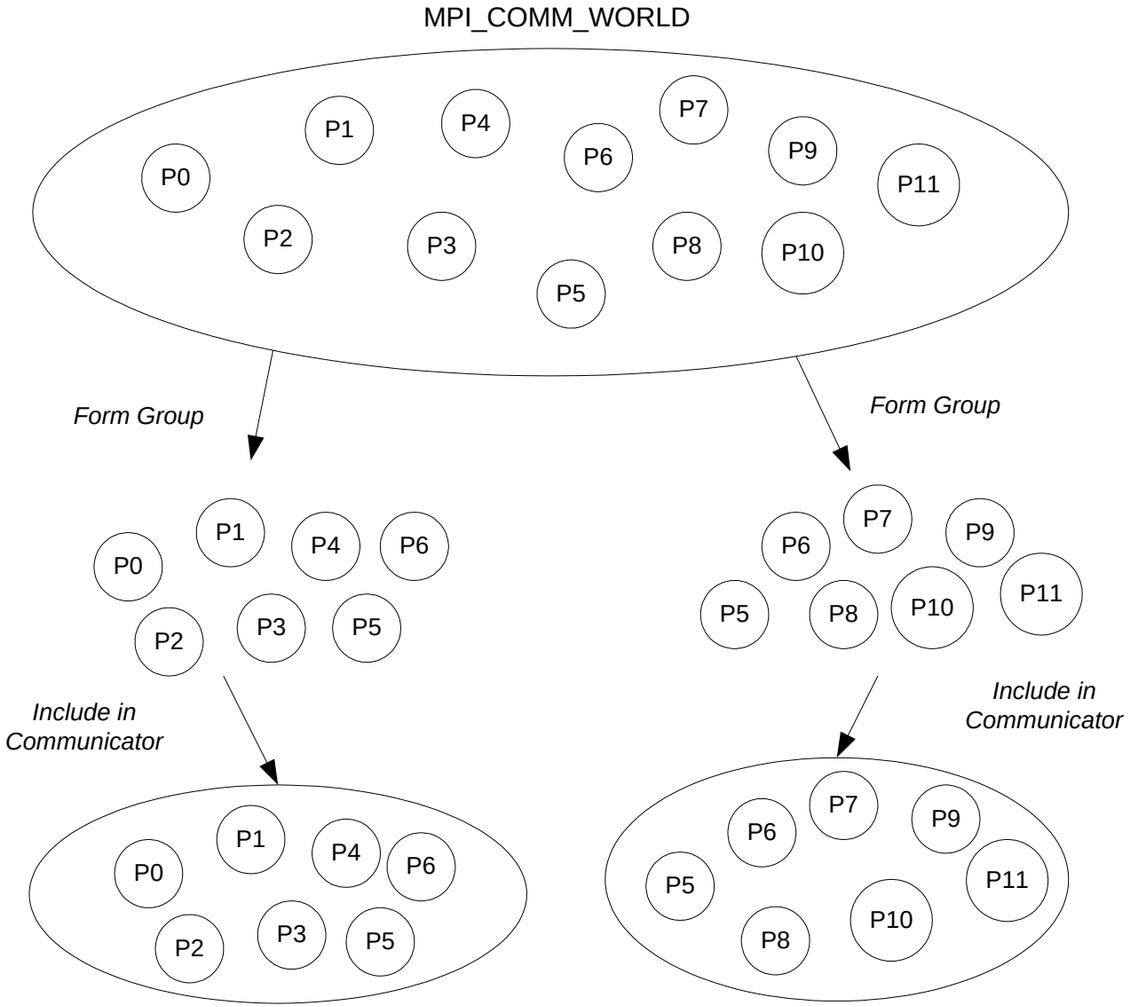
(b)

Collective Communications

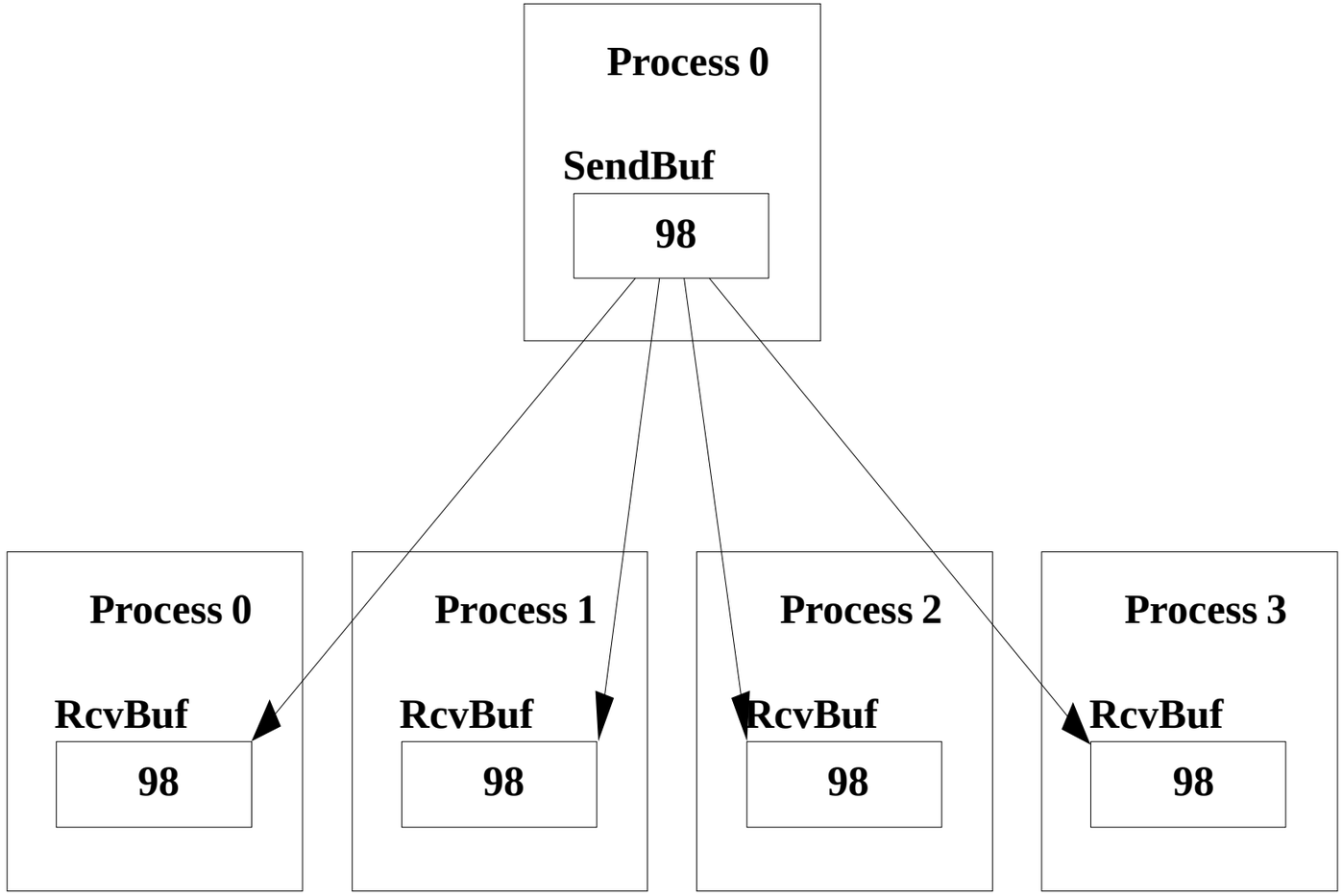
- Involve all processes within a communicator
- Blocking
- MPI_Barrier (comm)
  - Barrier synchronization
- MPI_Bcast (*buffer,count,datatype,root,comm)
  - Broadcasts from process of rank “root” to all other processes
- MPI_Scatter (*sendbuf,sendcnt,sendtype,*recvbuf,
  ..... recvcnt,recvtype,root,comm)
  - Sends different messages to each process in a group
- MPI_Gather (*sendbuf,sendcnt,sendtype,*recvbuf,
  ..... recvcount,recvtype,root,comm)
  - Gathers different messages from each process in a group
- Also reductions

Communicators and Groups

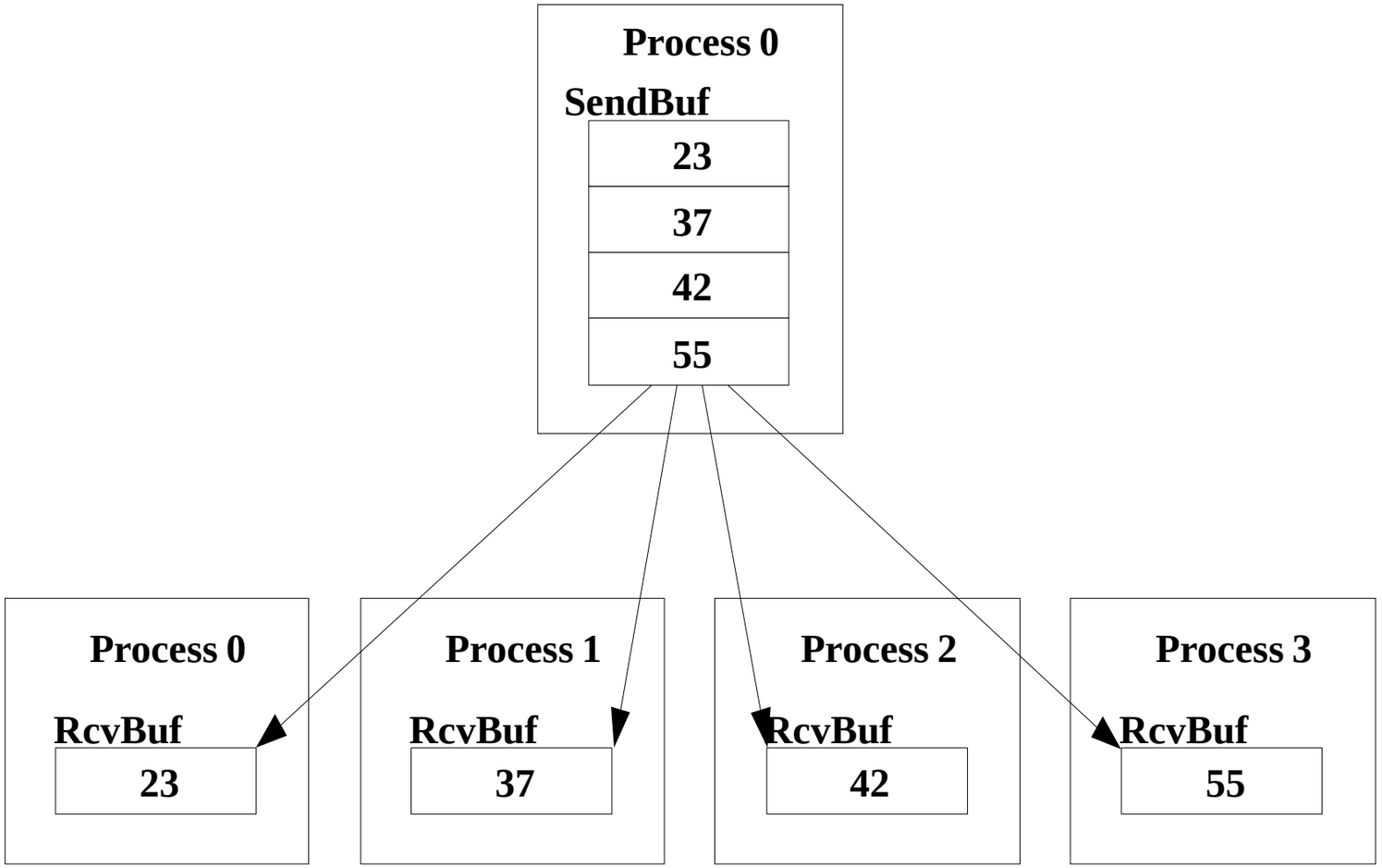
- Define collections of processes that may communicate
  - Often specified in message argument
  - MPI_COMM_WORLD – predefined communicator that contains all processes



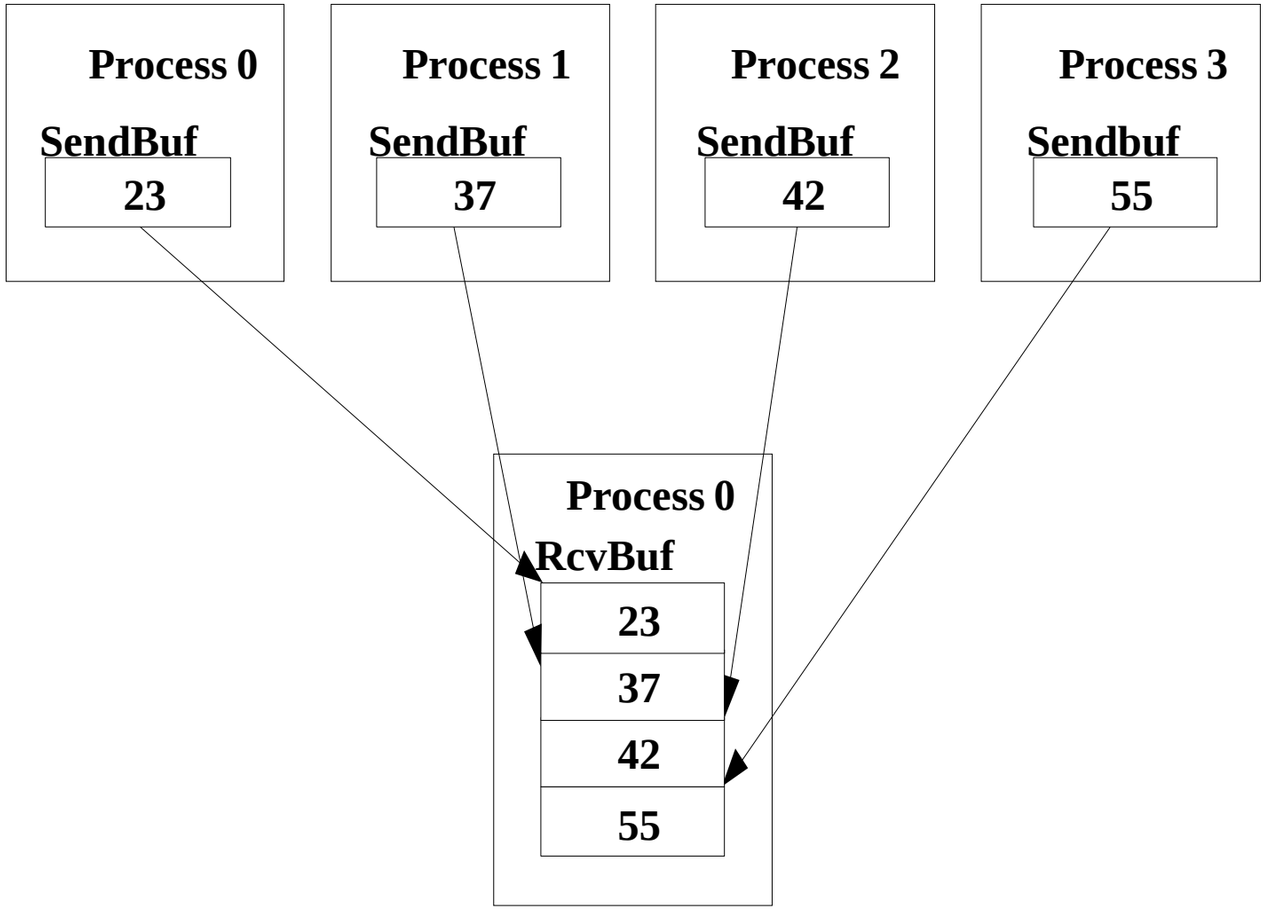
Broadcast Example



Scatter Example



Gather Example



Message Passing Implementation

- At the ABI and ISA level
  - No special support (beyond that needed for shared memory)
  - Most of the implementation is in the runtime
    - user-level libraries
  - Makes message passing relatively portable
- Three implementation models (given earlier)
  1) Multiple threads sharing an address space
  2) Multiple processes sharing an address space
  3) Multiple processes with non-shared address space (and different Oses)

Multiple Threads Sharing Address Space

- Runtime manages buffering and tracks communication
  - Communication via normal loads and stores using shared memory
- Example: Send/Receive
  - Send calls runtime, runtime posts availability of message in runtime-managed table
  - Receive calls runtime, runtime checks table, finds message
  - Runtime copies data from send buffer to store buffer via load/stores
- Fast/Efficient Implementation
  - May even be advantageous over shared memory paradigm
    - considering portability, software engineering aspects
  - Can use runtime thread scheduling
  - Problem with protecting private memories and runtime data area

Multiple Processes Sharing Address Space

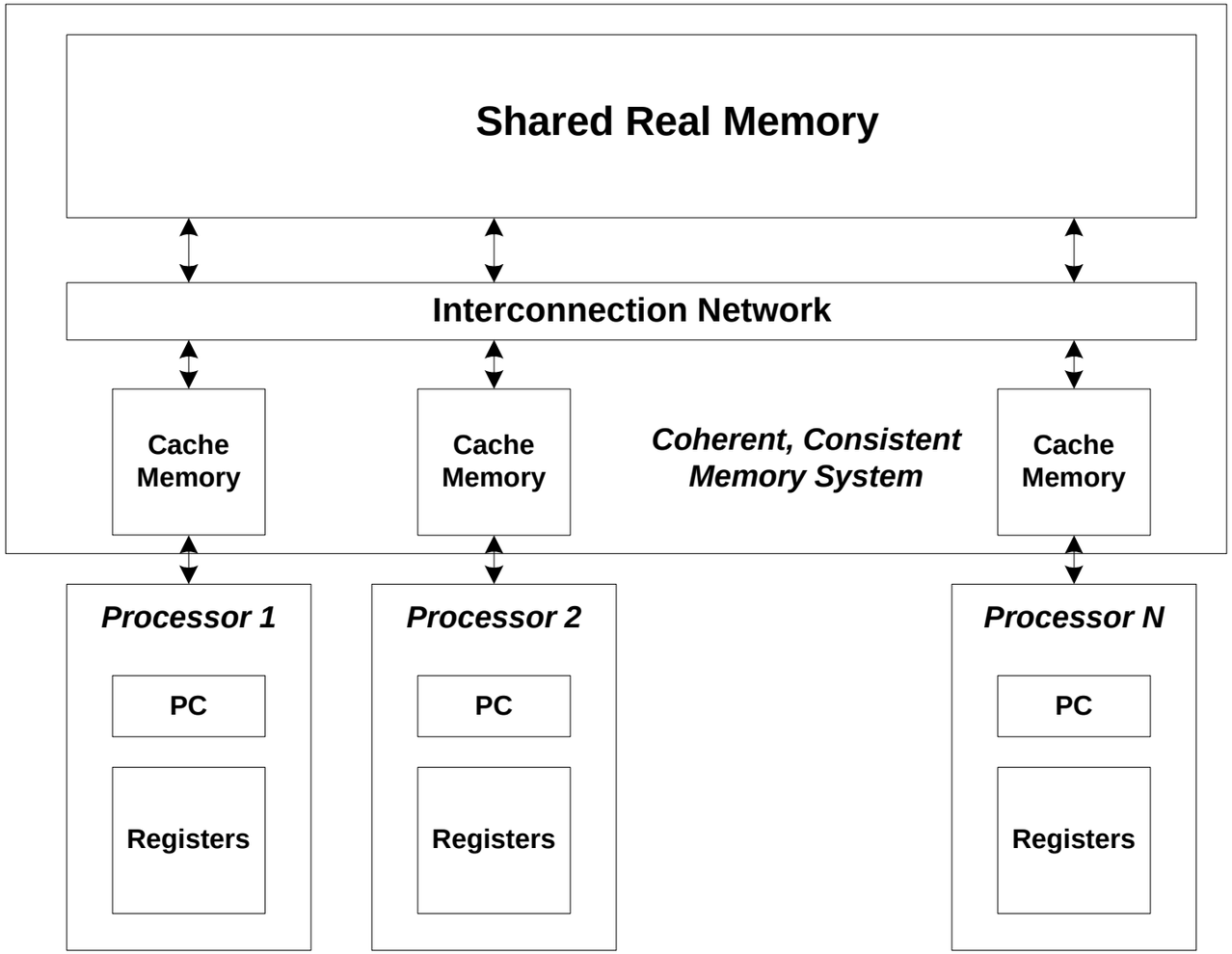
- Similar to multiple threads sharing address space
- Would rely on kernel scheduling
- May offer more memory protection
  - With intermediate runtime buffering
  - User processes can not access others' private memory

Multiple Processes w/ Non-Shared Address Space

- Most common implementation
- Communicate via networking hardware
- Send/receive to runtime
  - Runtime converts to OS (network) calls
- Relatively high overhead
  - Most HPC systems use special low-latency, high-bandwidth networks
  - Buffering in receiver's runtime space may save some overhead for receive (doesn't require OS call)

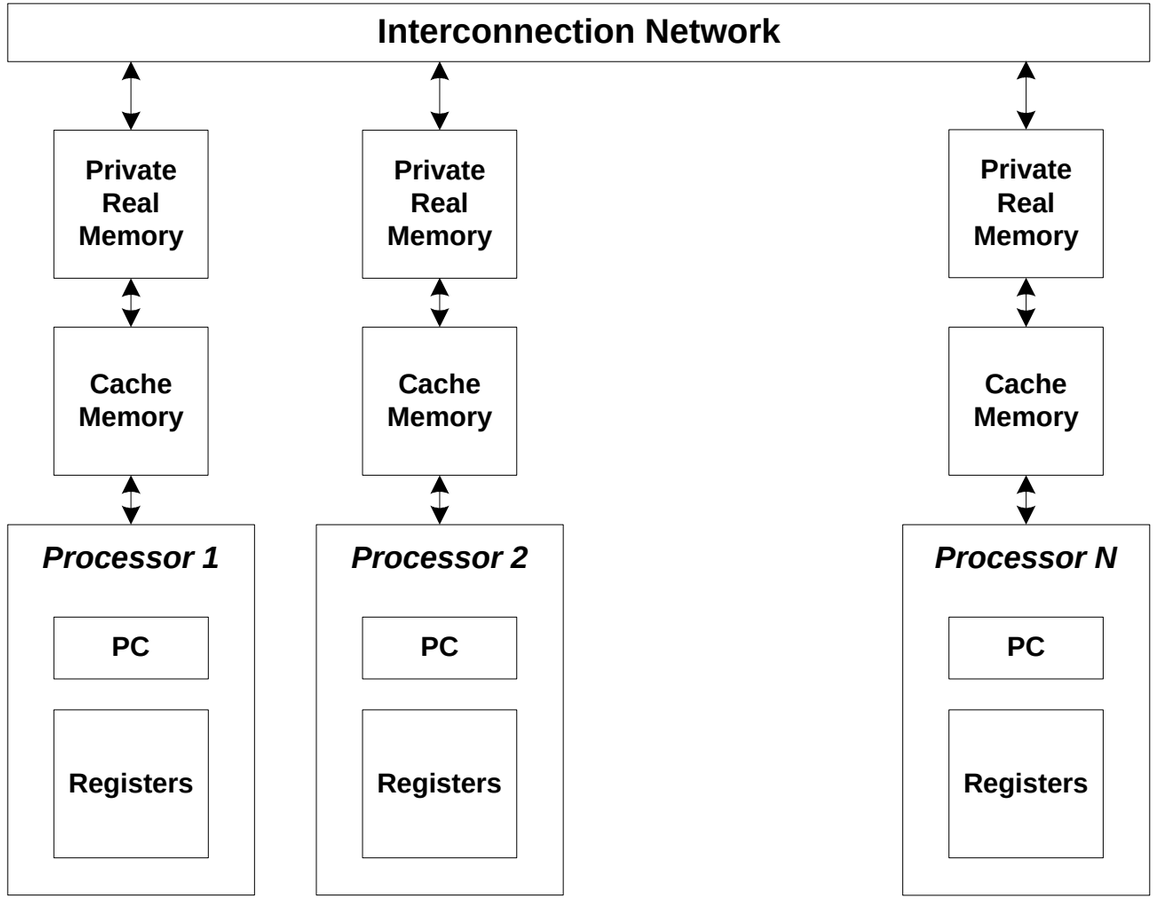
At the ISA Level: Shared Memory

- Multiple processors
- Architected shared virtual memory
- Architected Synchronization instructions
- Architected Cache Coherence
- Architected Memory Consistency



At the ISA Level: Message Passing

- Multiple processors
- Shared or non-shared real memory (multi-computers)
- Limited ISA support (if any)
  - An advantage of distributed memory systems --Just connect a bunch of small computers
  - Some implementations may use shared memory managed by runtime



18-600 Foundations of Computer Systems

Lecture 21: "Network Programming – Part 1"

John P. Shen & Gregory Kesden

November 13, 2017

Next Time ...

- Required Reading Assignment:
  - **Chapter 11 of CS:APP (3rd edition) by Randy Bryant & Dave O'Hallaron.**

Electrical & Computer ENGINEERING

Carnegie Mellon University