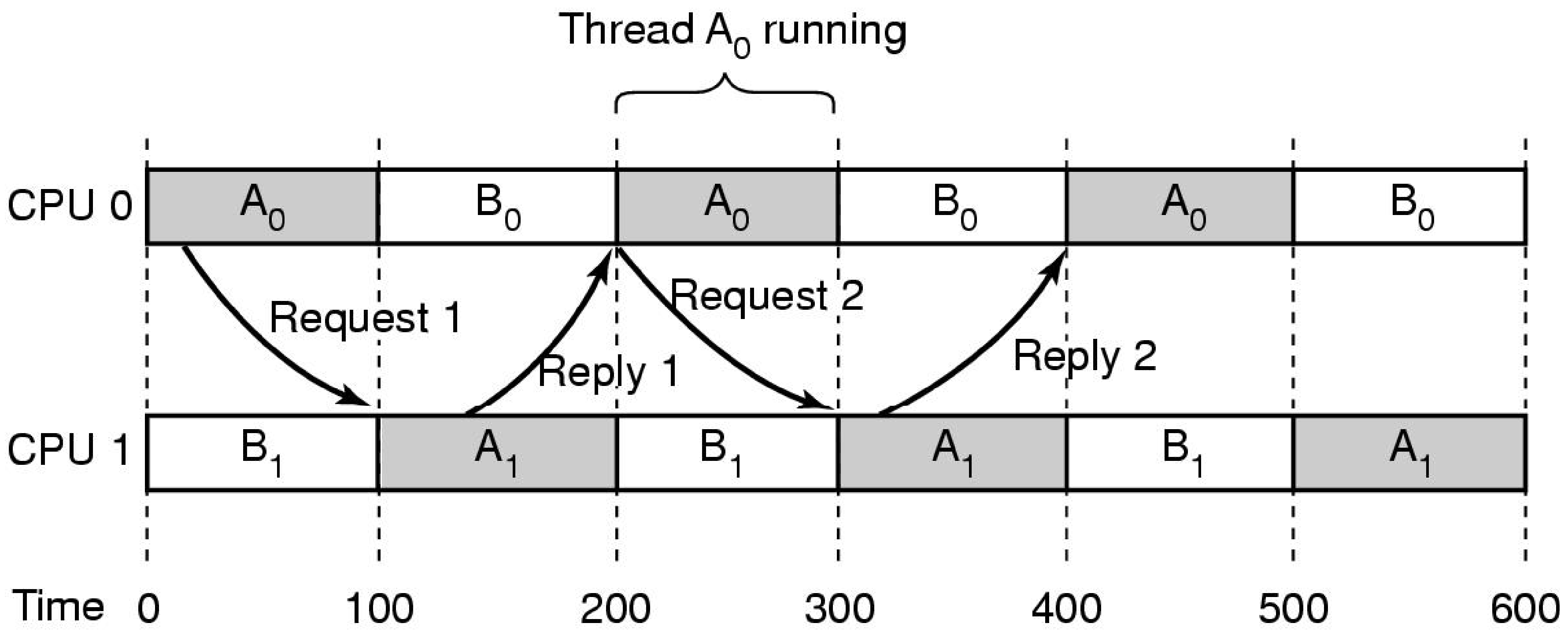
Multiprocessor/Multicore Systems

Scheduling, Synchronization, cont

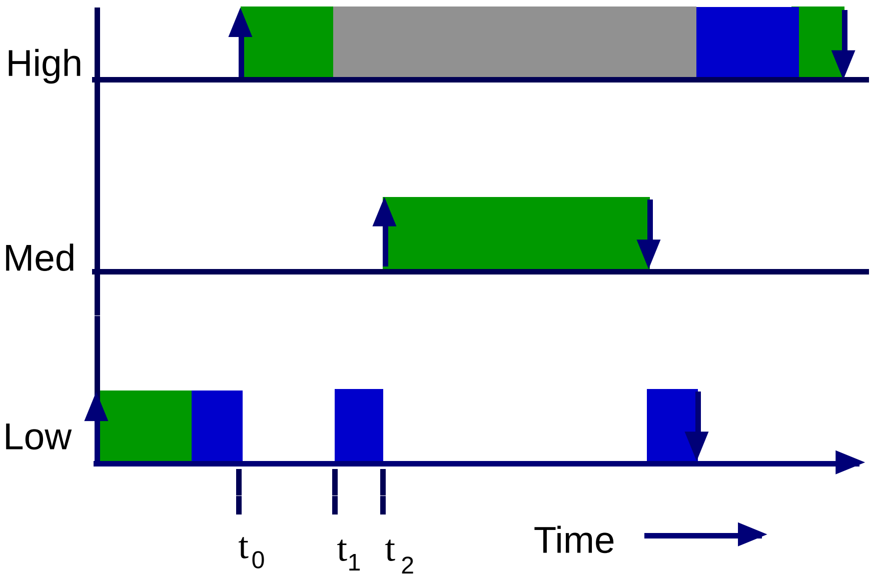
Recall:
Multiprocessor Scheduling: a problem



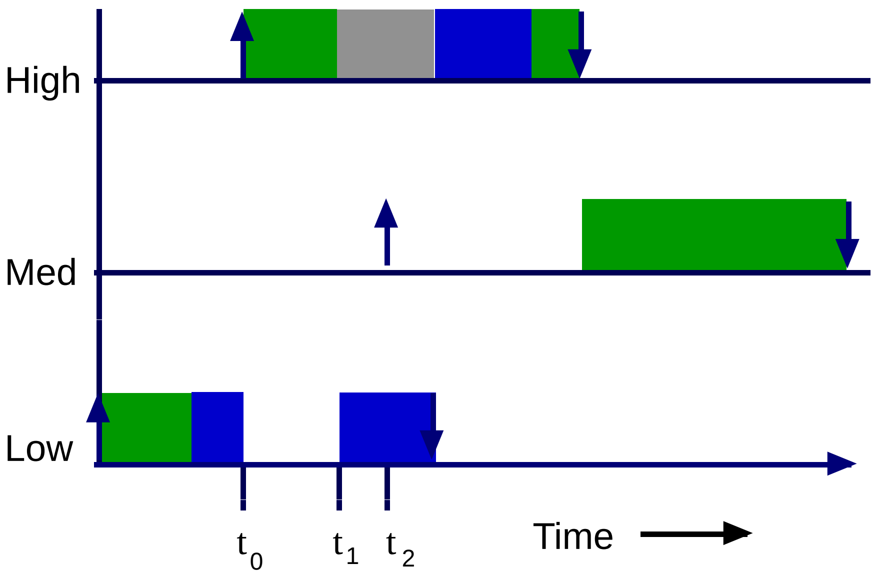
- Problem with communication between two threads
  - both belong to process *A*
  - both running out of phase
- Scheduling and synchronization inter-related in multiprocessors

The *Priority Inversion* Problem

Uncontrolled use of locks in RT systems can result in unbounded blocking due to *priority inversions*.



Possible solution: Limit priority Inversions by *modifying task priorities*.



🟦 lock

▨ Priority Inversion

🟩 Computation not involving shared object accesses

Scheduling and Synchronization

Priorities + locks may result in:

priority inversion: To cope/avoid this:

- use **priority inheritance**
- **Avoid locks in synchronization** (wait-free, lock-free, optimistic synchronization)

convoy effect: processes need a resource for short time, the process holding it may block them for long time (hence, poor utilization)

- **Avoiding locks** is good here, too

Readers-Writers and non-blocking synchronization

(some slides are adapted from J. Anderson's slides on same topic)

The Mutual Exclusion Problem

Locking Synchronization

```
while true do
    Noncritical Section;
    Entry Section;
        Critical Section;
    Exit Section
od
```

- N processes, each with this structure:
- Basic Requirements:
  - Exclusion: Invariant(# in CS $\leq$ 1).
  - Starvation-freedom: (process *i* in Entry) leads-to (process *i* in CS).
- Can implement by "busy waiting" (spin locks) or using kernel calls.

Synchronization without locks

- The problem:
  - Implement a shared object *without mutual exclusion.*
    - **Shared Object:** A data structure (*e.g.*, queue) shared by concurrent processes.
  - **Why?**
    - To avoid performance problems that result when a lock-holding task is delayed.
    - To enable more interleaving (enhancing parallelism)
    - To avoid priority inversions

~~Locking~~

Synchronization without locks

- Two variants:
  - **Lock-free:**
    - system-wide progress is guaranteed.
    - Usually implemented using "retry loops."
  - **Wait-free:**
    - Individual progress is guaranteed.
    - More involved algorithmic methods

Readers/Writers Problem

[Courtois, et al. 1971.]

- Similar to mutual exclusion, but several readers can execute "critical section" at the same time.
- If a writer is in its critical section, then no other process can be in its critical section.
- + no starvation, fairness

Solution 1

Readers have “priority”…

```
w, mutex: boolean semaphore
Initially 1

Writer::
P(w);
  CS;
V(w)
```

```
Reader::
P(mutex);
  rc := rc + 1;
  if rc = 1 then P(w) fi;
V(mutex);
CS;
P(mutex);
  rc := rc – 1;
  if rc = 0 then V(w) fi;
V(mutex)
```

“First” reader executes P(w). “Last” one executes V(w).

Concurrent Reading and Writing [Lamport '77]

- Previous solutions to the readers/writers problem use some form of mutual exclusion.
- Lamport considers solutions in which readers and writers access a shared object *concurrently*.
- Motivation:
  - Don't want writers to wait for readers.
  - Readers/writers solution may be needed to implement mutual exclusion (circularity problem).

Interesting Factoids

- This is the first ever **lock-free algorithm: guarantees consistency without locks**
- An algorithm very similar to this has been implemented within an embedded controller in Mercedes automobiles

The Problem

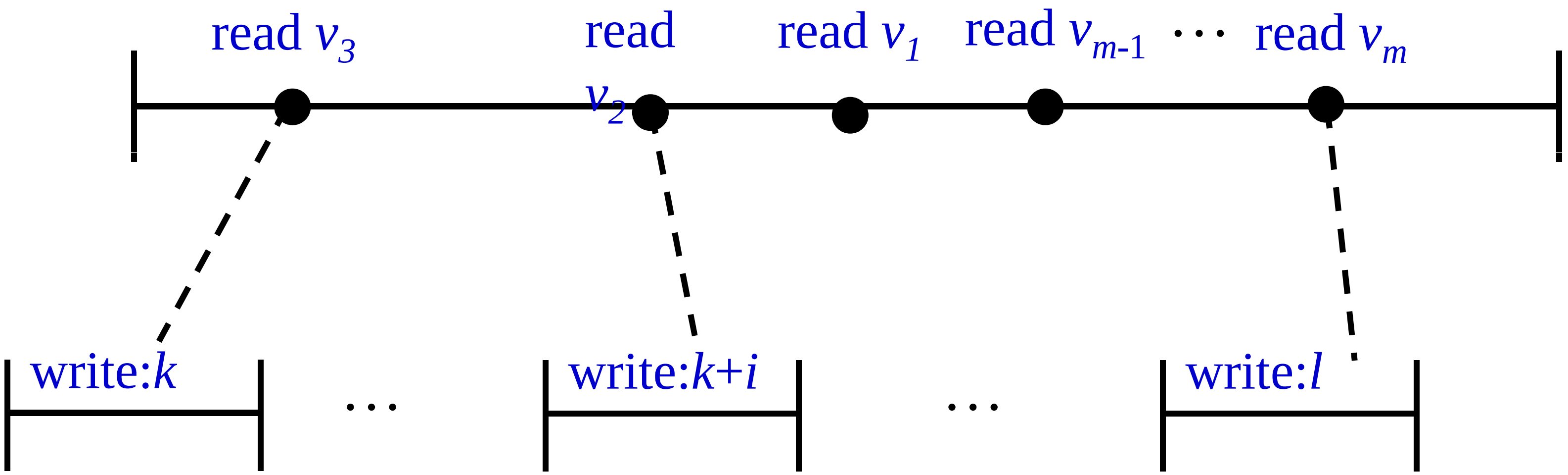
- Let *v* be a data item, consisting of one or more sub-items.
  - For example,
    - *v* = 256 consists of three digits, "2", "5", and "6".
    - String "I love spring" consists of 3 words (or 13 characters)
    - A book consists of several chapters
    - ….
- **Underlying model:** subitems can be read and written atomically.
- **Objective:** Simulate atomic reads and writes of the data item *v*.

Preliminaries

- **Definition:** $v^{[i]}$, where $i \geq 0$, denotes the i^{th} value written to v. ($v^{[0]}$ is v's initial value.)
- **Note:** No concurrent writing of v.
- Partitioning of v: $v_1 \cdots v_m$.
  - **To start, focus on v being a number**
  - v_i may consist of multiple digits.
- **To read v:** Read each v_i (in some order).
- **To write v:** Write each v_i (in some order).

More Preliminaries

read r:



We say: r reads $v^{[k,l]}$.

Value is consistent if $k = l$.

Main Theorem

Assume that $i \leq j$ implies that $v^{[i]} \leq v^{[j]}$, where $v = d_1 \ldots d_m$.

(a) If v is always written from right to left, then a read from left to right obtains a value $v^{[k,l]} \leq v^{[l]}$.

(b) If v is always written from left to right, then a read from right to left obtains a value $v^{[k,l]} \geq v^{[k]}$.

discuss why

Readers/Writers Solution

Writer::

```
$\overrightarrow{V1}$ :> V1;
write D;
$\overleftarrow{V2}$ := V1
```

Reader::

```
repeat temp := $\overrightarrow{V2}$
    read D
until $\overleftarrow{V1}$ = temp
```

:> means assign larger value.

$\overrightarrow{V1}$ means “left to right”.

$\overleftarrow{V2}$ means “right to left”.

Useful Synchronization Primitives

Usually *Necessary* in Nonblocking Algorithms

```
CAS(var, old, new)
  ⟨ if var ≠ old then return false fi;
    var := new;
    return true ⟩
```

CAS2
extends
this

```
LL(var)
  ⟨ establish “link” to var;
    return var ⟩

SC(var, val)
⟨ if “link” to var still exists then
      break all current links of all processes;
      var := val;
      return true
   else
      return false
   fi ⟩
```

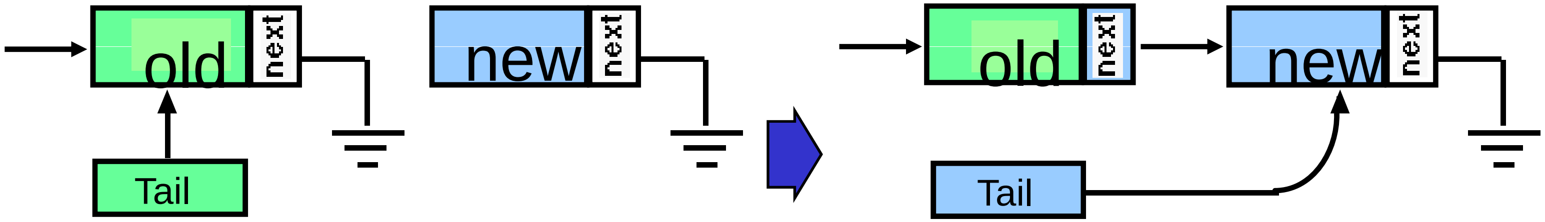
Another Lock-free Example

Shared Queue

```
type  Qtype = record v: valtype; next:  pointer to Qtype end
shared var  Tail:  pointer to Qtype;
local var  old, new: pointer to Qtype

procedure Enqueue (input: valtype)
    new := (input,  NIL);
    repeat  old := Tail
    until  CAS2(Tail, old->next, old, NIL, new, new)
```

retry loop

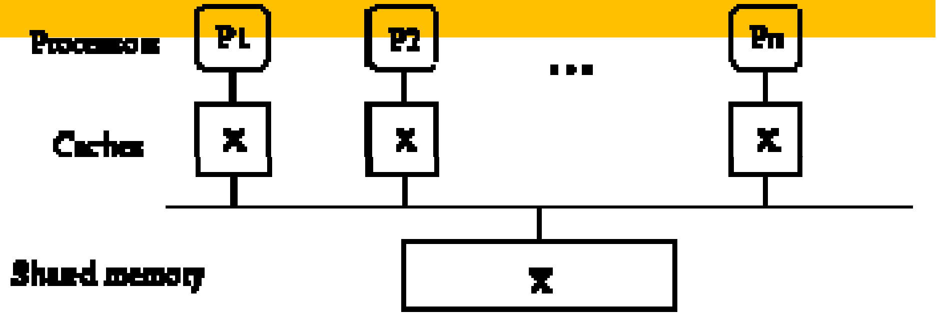


Cache-coherence

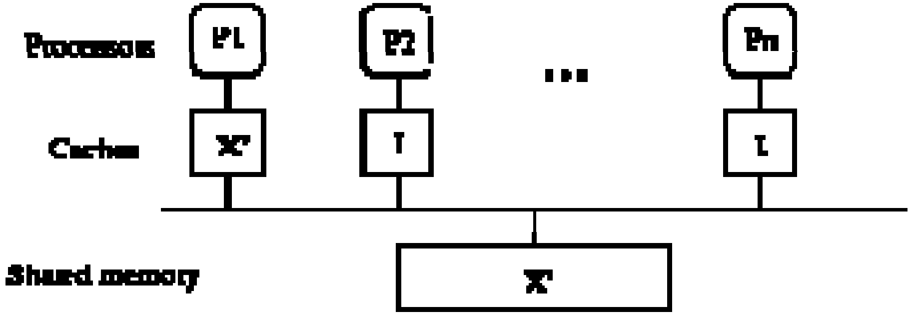
cache coherency protocols are based on a set of (cache block) *states* and *state transitions* : 2 main types of protocols

- write-update
- write-invalidate

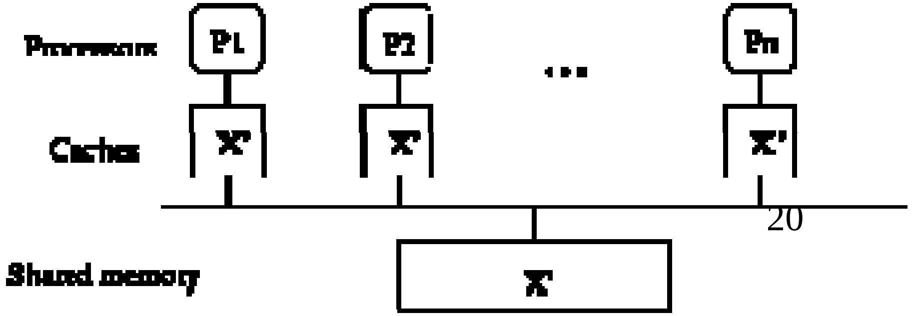
- Reminds readers/writers?



Write-invalidate protocol



Write-update protocol



Multiprocessor architectures, memory consistency

- Memory access protocols and cache coherence protocols define memory consistency models
- Examples:
  - Sequential consistency: e.g. SGI Origin (more and more seldom found now...)
  - Weak consistency: sequential consistency for special synchronization variables and actions before/after access to such variables. No ordering of other actions. e.g. SPARC architectures
- Memory consistency also relevant at compiler-level
  - i.e. The latter may reorder for optimization purposes

Distributed OS issues:
IPC: Client/Server, RPC mechanisms
Clusters, load balncing, Middleware

Multicomputers

- Definition:
  Tightly-coupled CPUs that do not share memory

- Also known as
  - cluster computers
  - clusters of workstations (COWs)

  - illusion is one machine
  - Alternative to symmetric multiprocessing (SMP)

Clusters

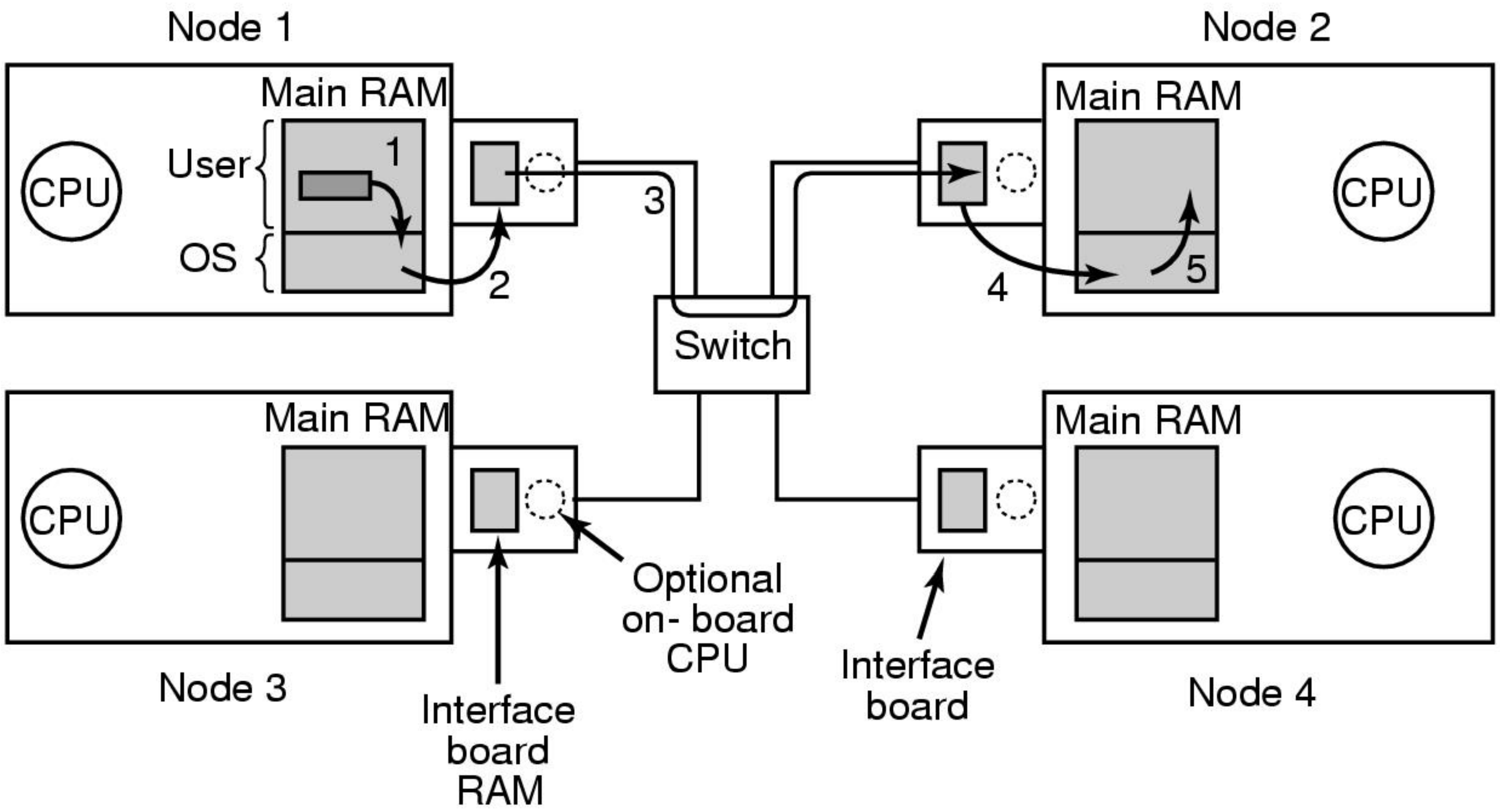
Benefits of Clusters

- Scalability
  - Can have dozens of machines each of which is a multiprocessor
  - Add new systems in small increments
- Availability
  - Failure of one node does not mean loss of service (well, not necessarily at least… why?)
- Superior price/performance
  - Cluster can offer equal or greater computing power than a single large machine at a much lower cost

BUT:

- think about communication!!!
- The above picture is changing with multicore systems

Multicomputer Hardware example



Network interface boards in a multicomputer

Clusters: Operating System Design Issues

Failure management

- offers a high probability that all resources will be in service
- Fault-tolerant cluster ensures that all resources are always available (replication needed)

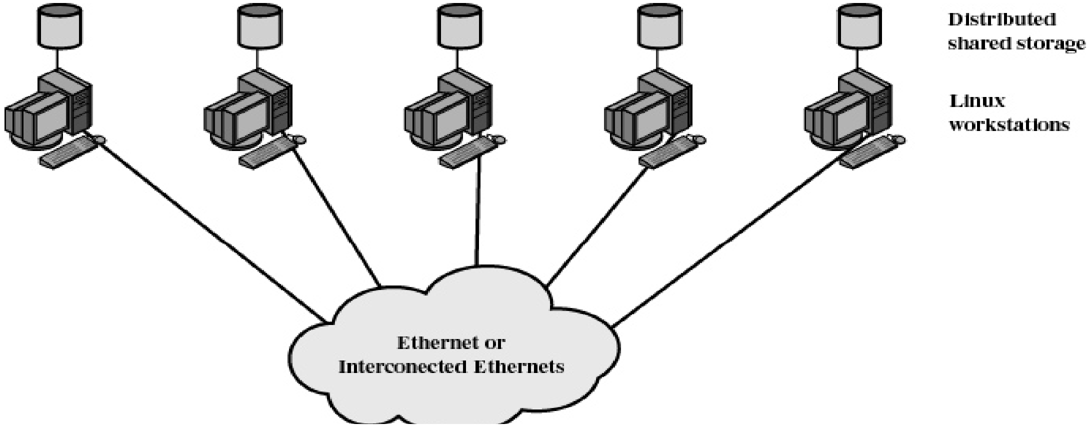
Load balancing

- When new computer added to the cluster, automatically include this computer in scheduling applications

Parallelism

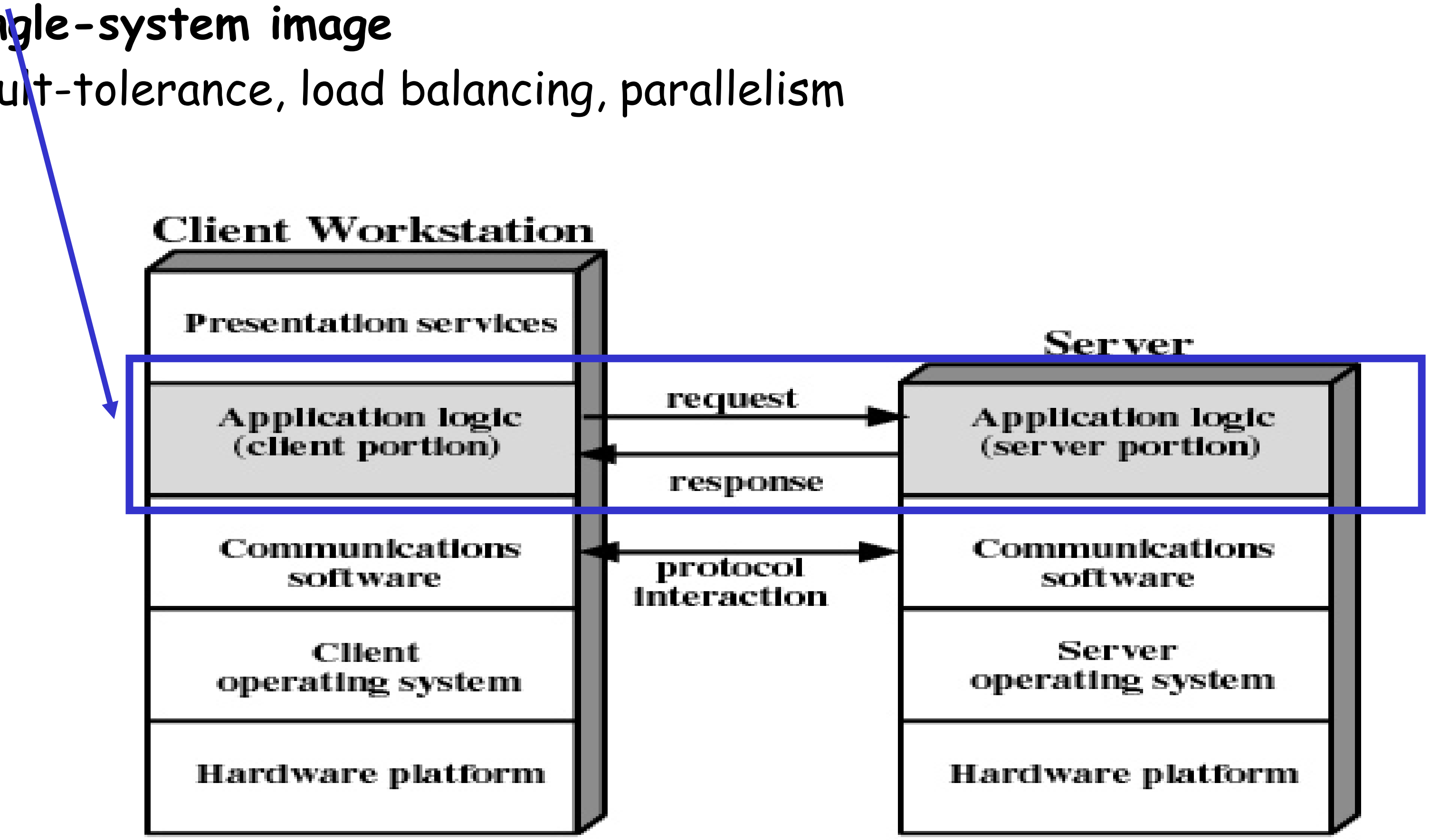
- parallelizing compiler or application

e.g. beowulf, linux clusters



Cluster Computer Architecture

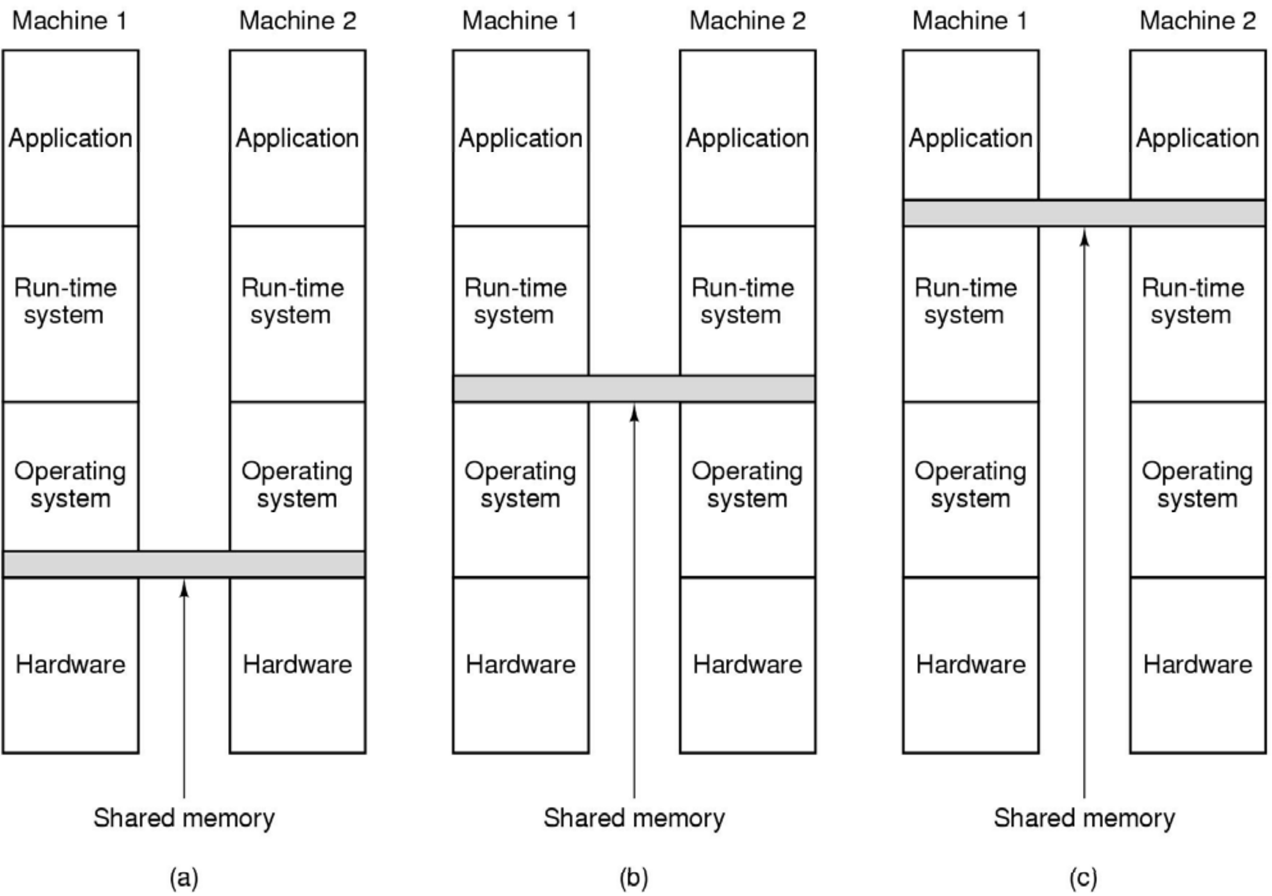
- Network
- Middleware layer to provide
  - **single-system image**
  - fault-tolerance, load balancing, parallelism



IPC

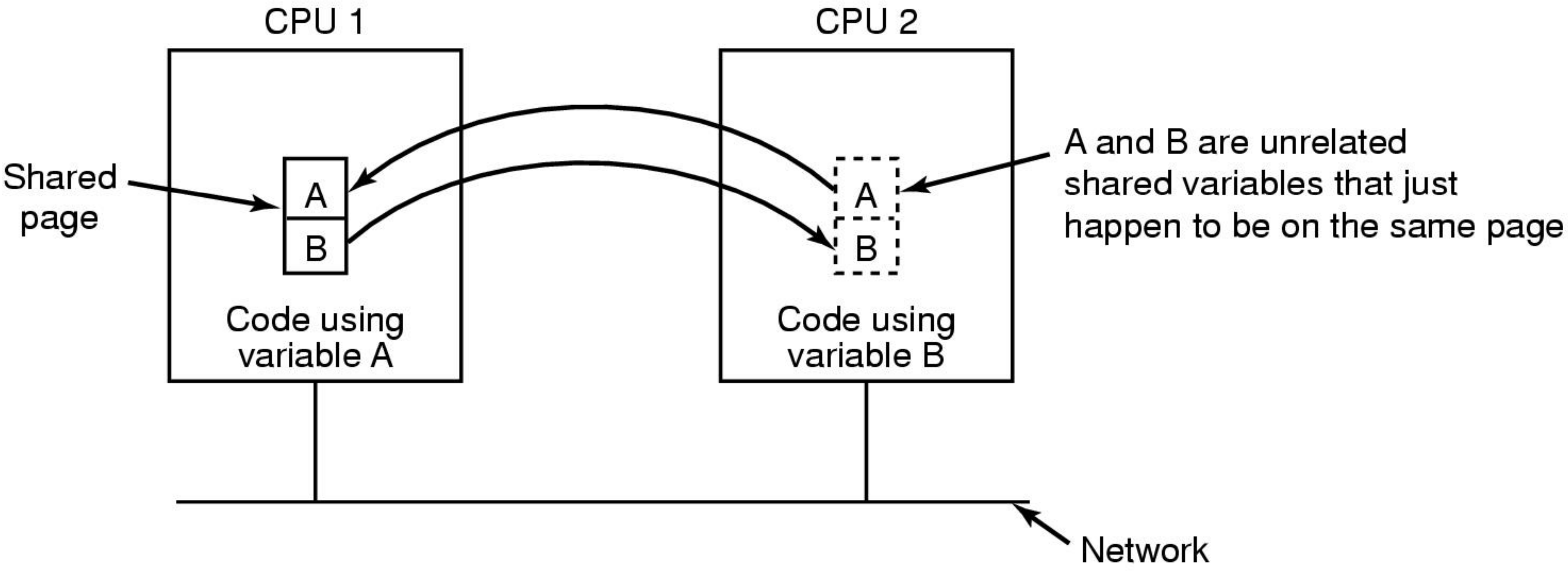
- Client-Server Computing
- Remote Procedure Calls
- P2P collaboration (related to overlays, cf. advanced networks and distr. Sys course)
- Distributed shared memory (cf. advanced distr. Sys course)

Distributed Shared Memory (1)



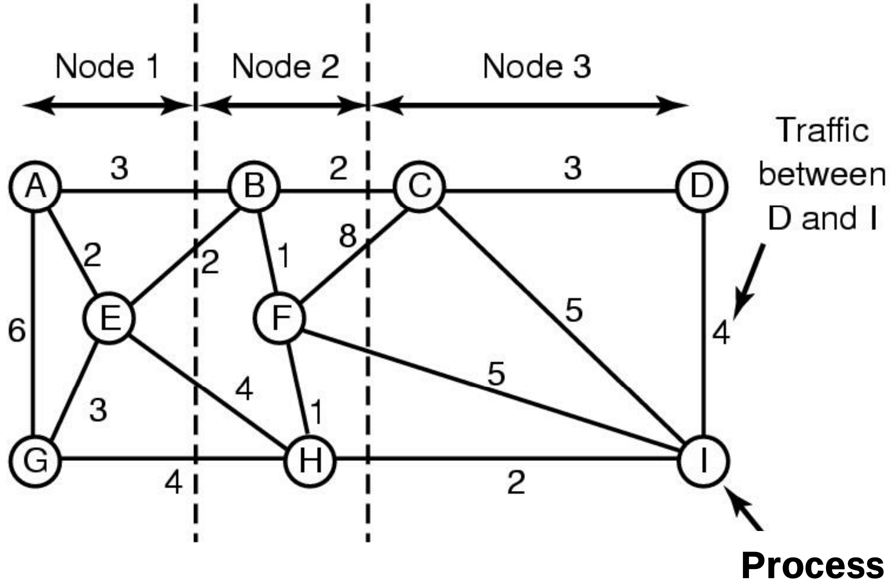
- Note layers where it can be implemented
  - hardware
  - operating system
  - user-level software

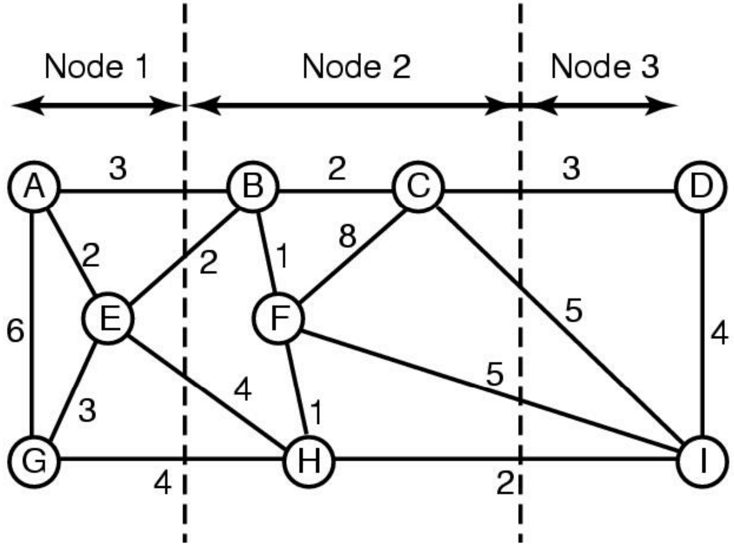
Distributed Shared Memory (2)



- False Sharing
- Must also achieve consistency
- Both issues also in cache protocols

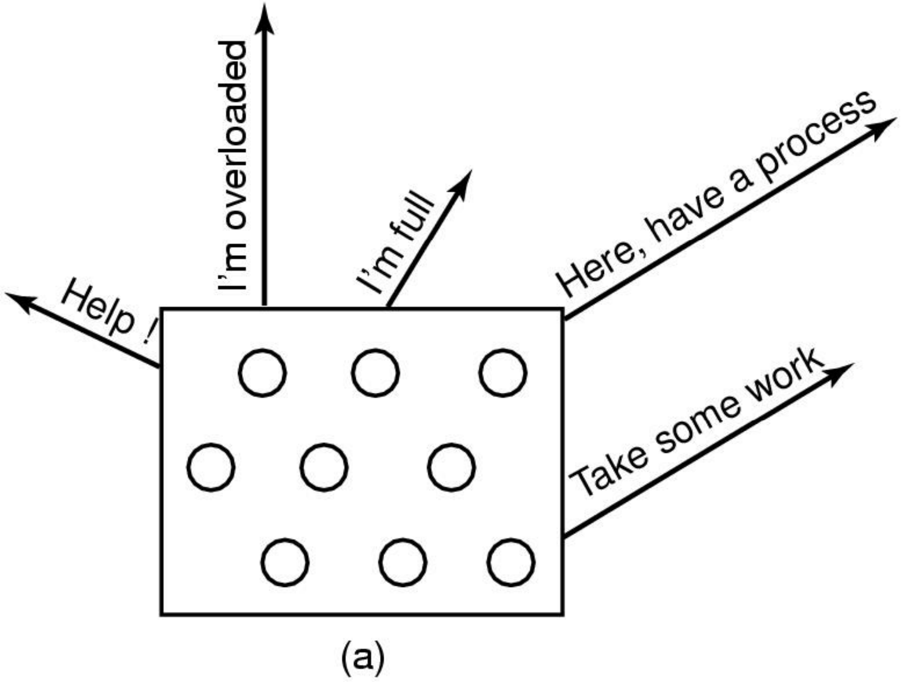
Multicomputer Scheduling
Load Balancing (1)





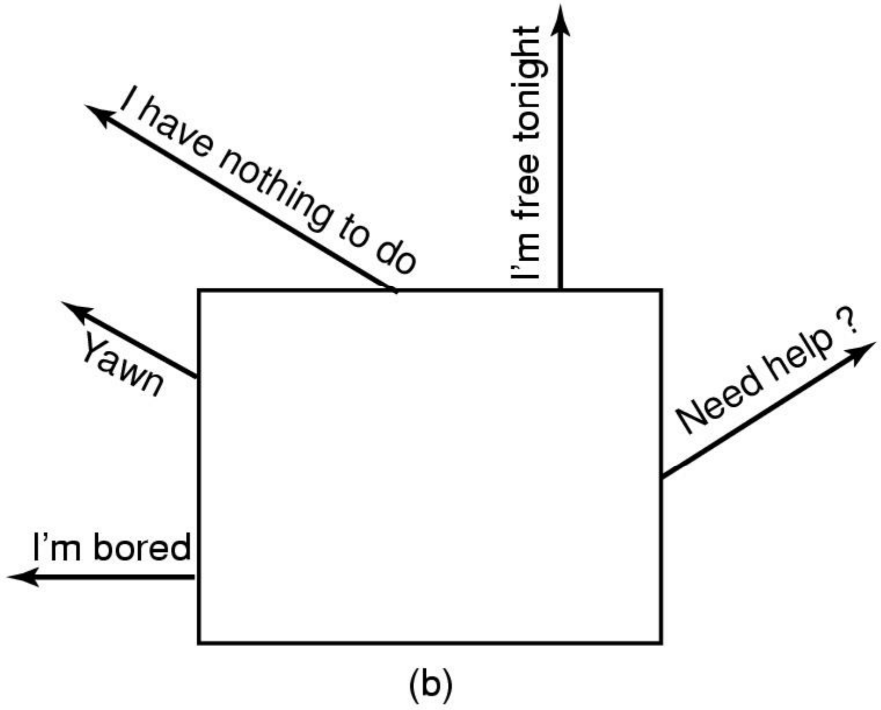
- Graph-theoretic deterministic algorithm

Load Balancing (2)



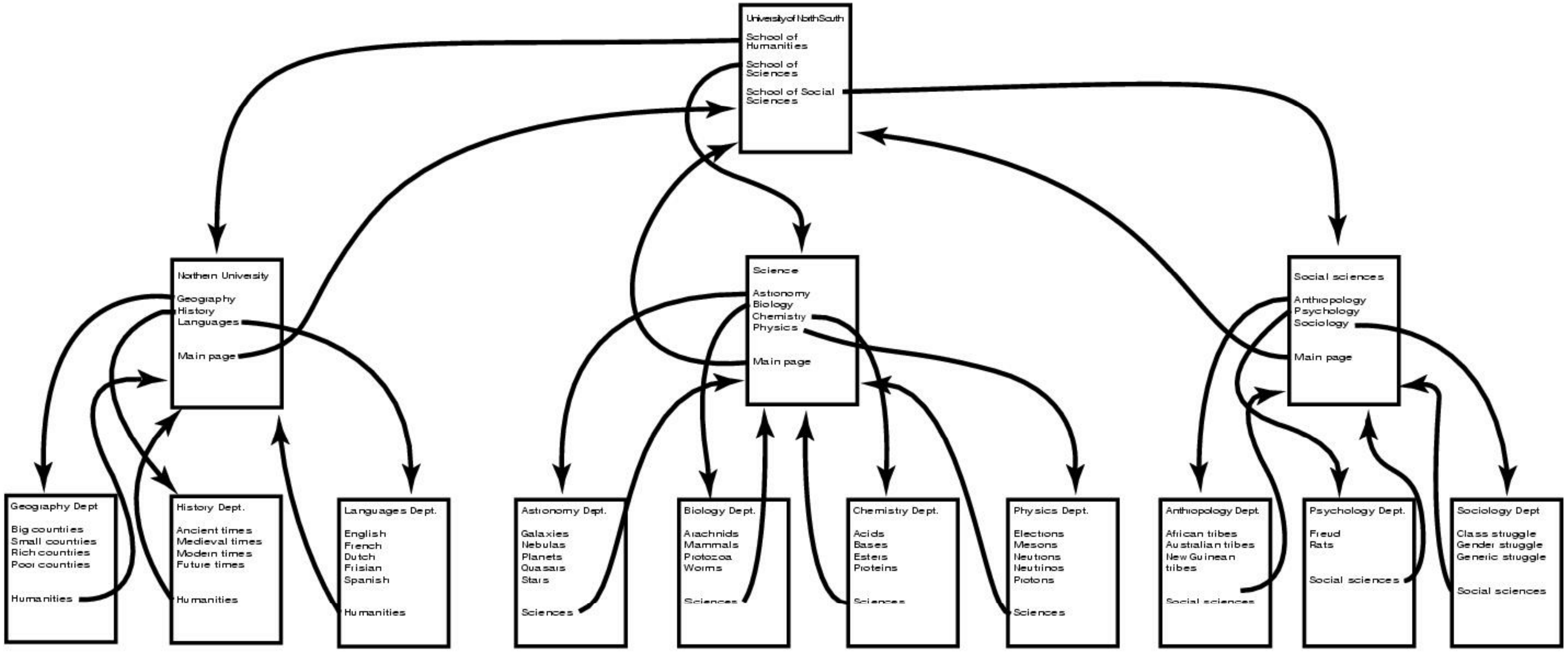
- Sender-initiated distributed heuristic algorithm
  - overloaded sender

Load Balancing (3)



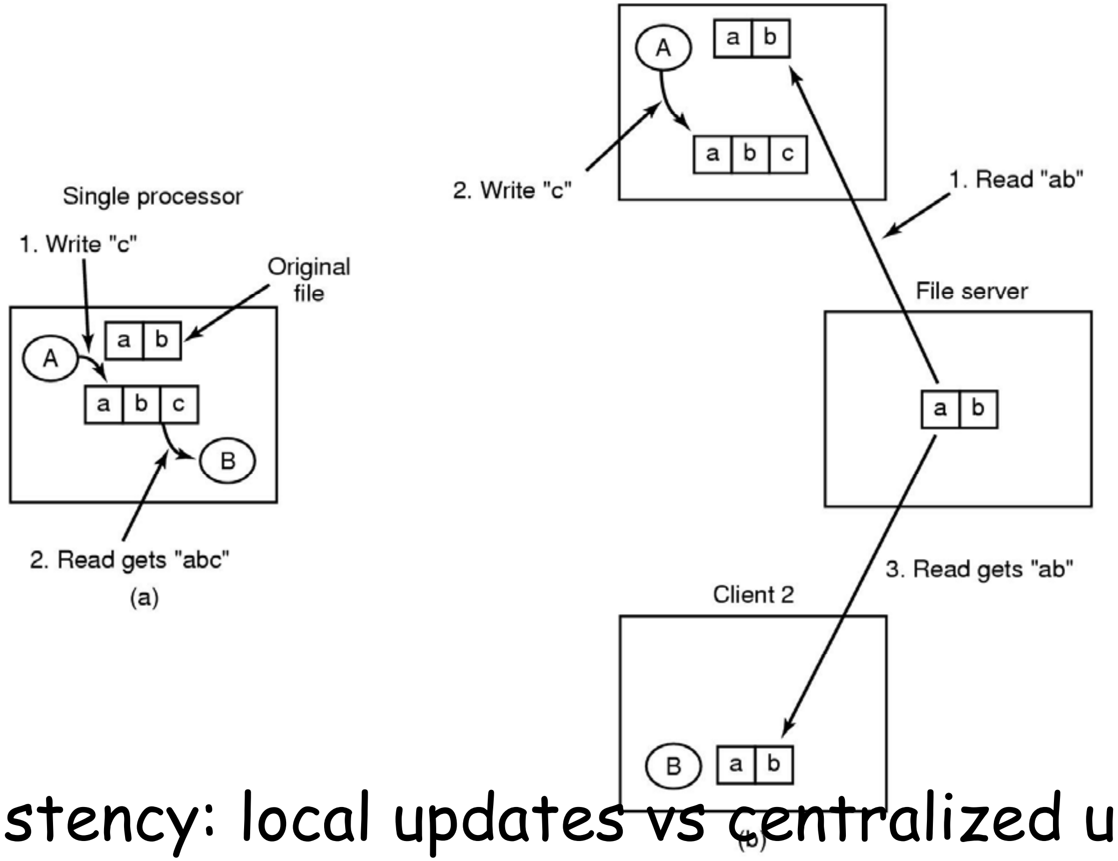
- Receiver-initiated distributed heuristic algorithm
  - under loaded receiver

Document-Based Middleware



- E.g. The Web
  - a big directed graph of documents

File System-Based Middleware

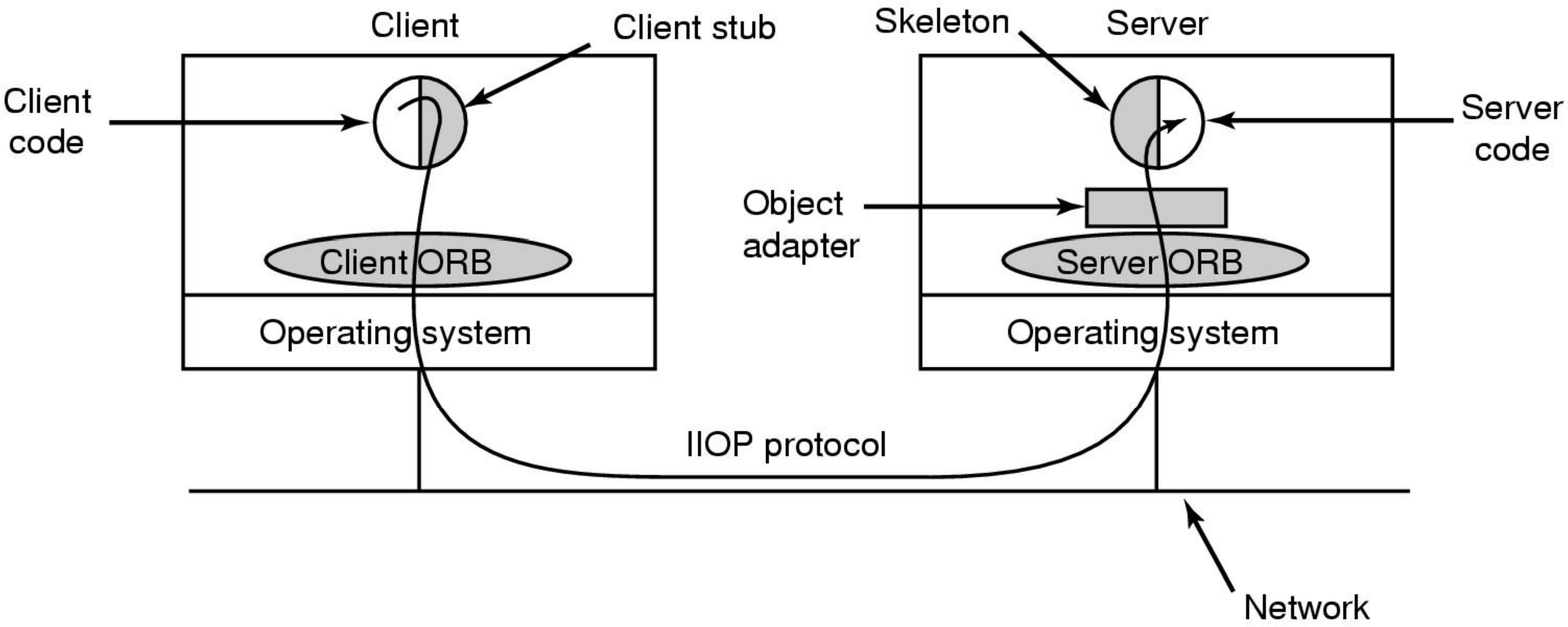


(a)

(b)

- Needs consistency: local updates vs centralized updates
- Some issues similar to cache coherence
- Semantics of File sharing and trade-offs
  - (a) single processor gives sequential consistency
  - (b) distributed system may return obsolete value

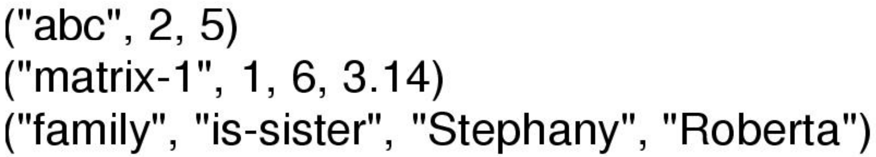
Shared Object-Based Middleware



- E.g. CORBA based system
  - Common Object Request Broker Architecture; IIOP: Internet InterORB protocol

Coordination-Based Middleware

- E.g. via Linda system for communication & synch
  - independent processes
  - communicate via abstract tuple space
  - Tuple
    - like a structure in C, record in Pascal



```
("abc", 2, 5)
("matrix-1", 1, 6, 3.14)
("family", "is-sister", "Stephany", "Roberta")
```

  - Operations: out (insert), in (remove), read (without removing) , eval (evaluate parameters)
- E.g. Jini - based on Linda model
  - devices plugged into a network
  - offer, use services

That's all folks! ☺ (for now)

- Summary: OS takes cares of processes needs
  - memory, CPU, data, files, IO, synchronization, resources,
- We have seen methods and instantaitions in maistream OS
- Recall ...

Recall …

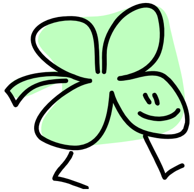
- After successful completion of the course students will be able to demonstrate knowledge and understanding of:
  - The core functionality of modern operating systems.
  - Key concepts and algorithms in operating system implementations.
  - Implementation of simple OS components.

The students will also be able to:

- Write programs that interface to the operating system at the system-call level.
- Implement a piece of system-level code.

Exam

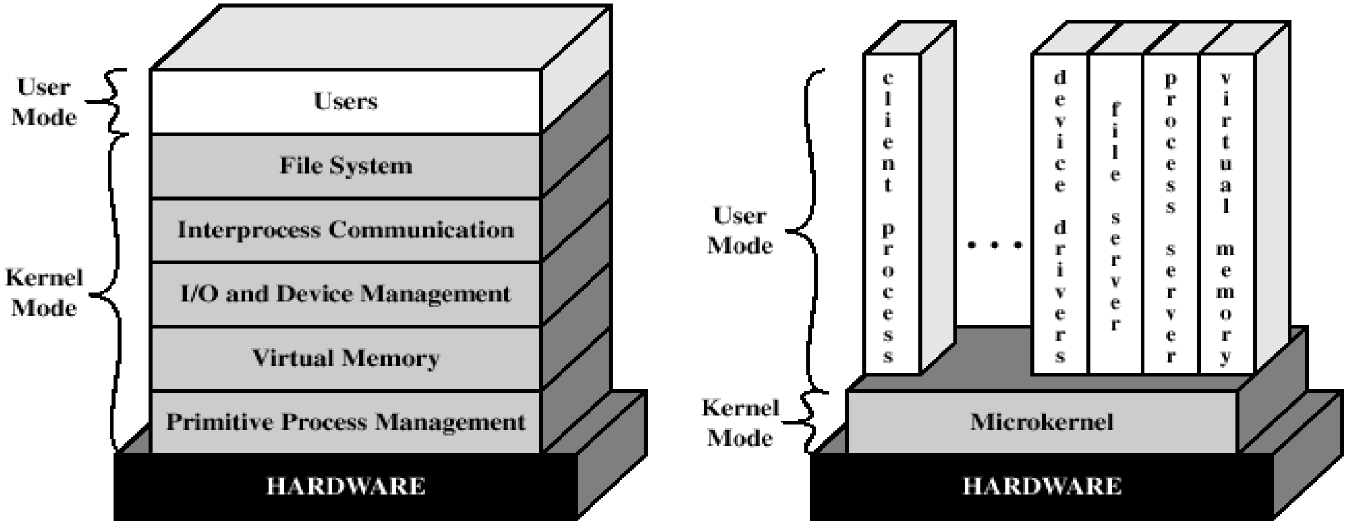
- 15 march, 8.30-12.30 M building
- Welcome and best wishes from the course support team!
- Thank you!



Extra notes on distr/multiproc OS

Also of relevance to Distributed Systems (and more): Microkernel OS organization

- Small OS core; contains only essential OS functions:
  - Low-level memory management (address space mapping)
  - Process scheduling
  - I/O and interrupt management
- Many services traditionally included in the OS kernel are now external subsystems
  - device drivers, file systems, virtual memory manager, windowing system, security services

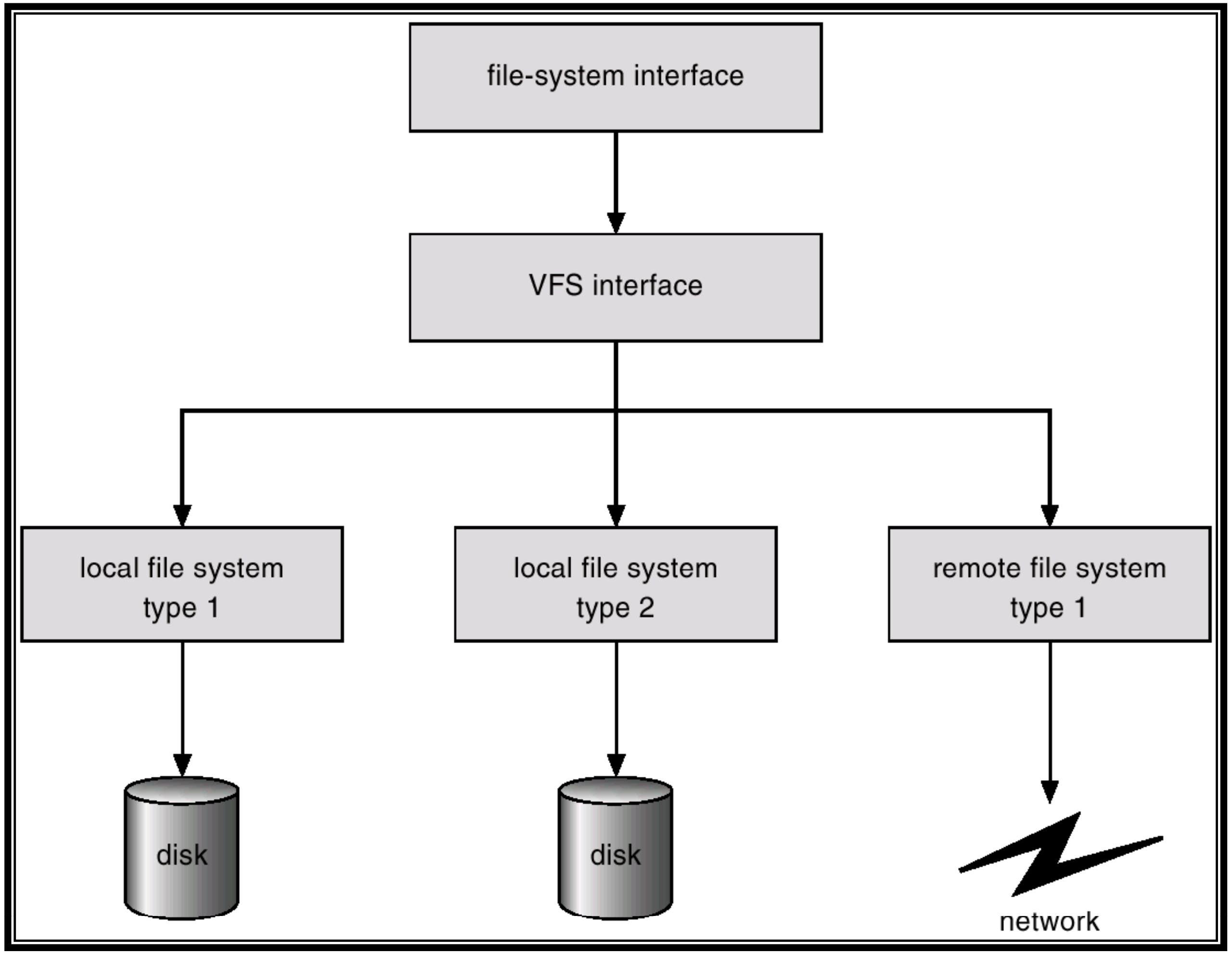


(a) Layered kernel (b) Microkernel

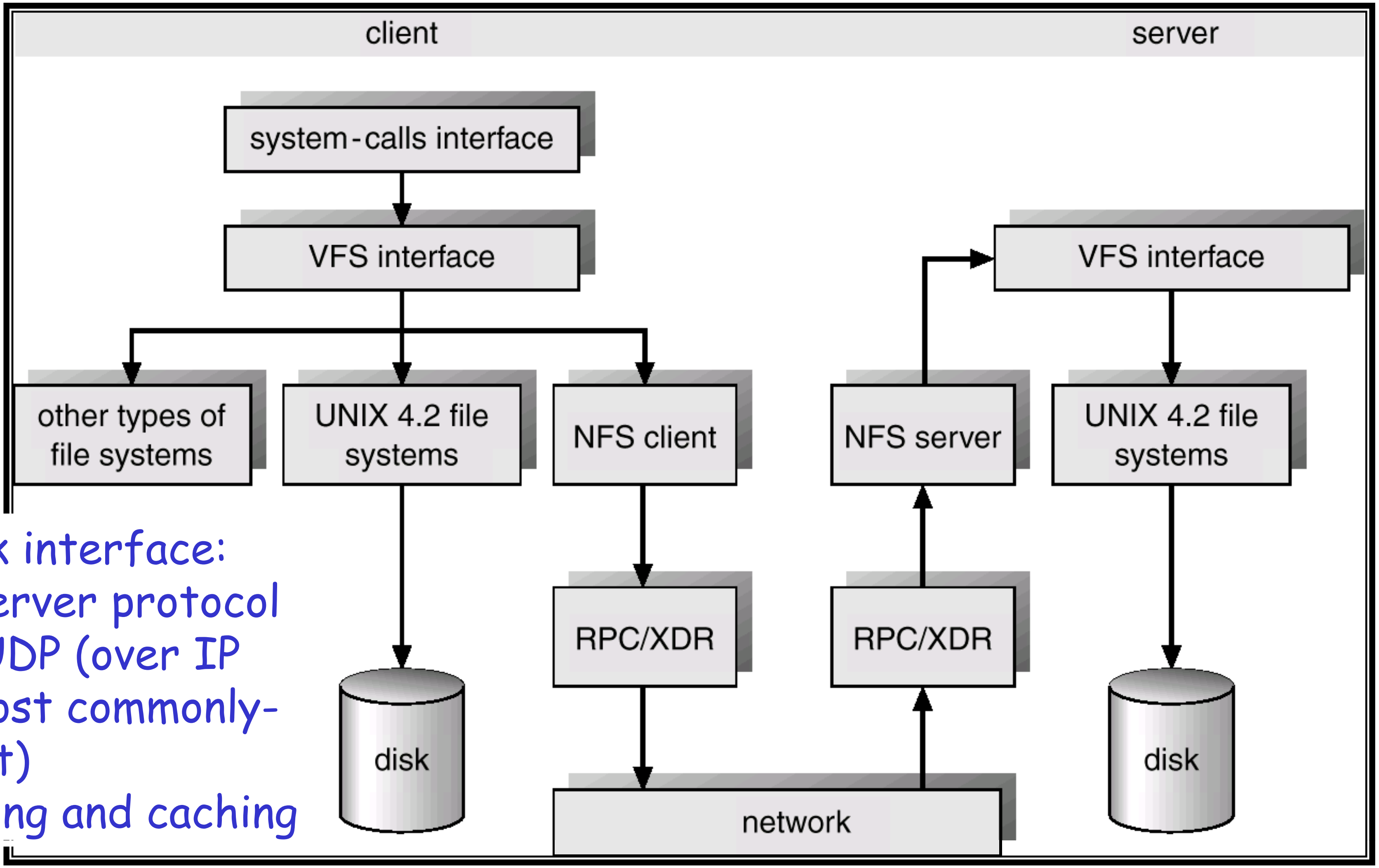
Benefits of a Microkernel Organization

- Uniform interface on request made by a process
  - All services are provided by means of message passing
- Distributed system support
  - Messages are sent without knowing what the target machine is
- Extensibility
  - Allows the addition/removal of services and features
- Portability
  - Changes needed to port the system to a new processor is changed in the microkernel - not in the other services
- Object-oriented operating system
  - Components are objects with clearly defined interfaces that can be interconnected
- Reliability
  - Modular design;
  - Small microkernel can be rigorously tested

Schematic View of Virtual File System



Schematic View of NFS Architecture



Network interface: client-server protocol

- Uses UDP (over IP over –most commonly- ethernet)
- Mounting and caching

Solution 2 readers writers

Writers have “priority” …

readers should not build long queue on r, so that writers can overtake => mutex3

```
Reader::
P(mutex3);
    P(r);
          P(mutex1);
          rc := rc + 1;
          if rc = 1 then P(w) fi;
          V(mutex1);
    V(r);
V(mutex3);
CS;
P(mutex1);
  rc := rc – 1;
  if rc = 0 then V(w) fi;
V(mutex1)
```

```
Writer::
P(mutex2);
  wc := wc + 1;
  if wc = 1 then P(r) fi;
V(mutex2);
P(w);
  CS;
V(w);
P(mutex2);
  wc := wc – 1;
  if wc = 0 then V(r) fi;
V(mutex2)
```

Properties

- If several writers try to enter their critical sections, one will execute P(*r*), blocking readers.
- Works assuming V(*r*) has the effect of picking a process waiting to execute P(*r*) to proceed.
- Due to *mutex3*, if a reader executes V(*r*) and a writer is at P(*r*), then the writer is picked to proceed.

On Lamport's R/W

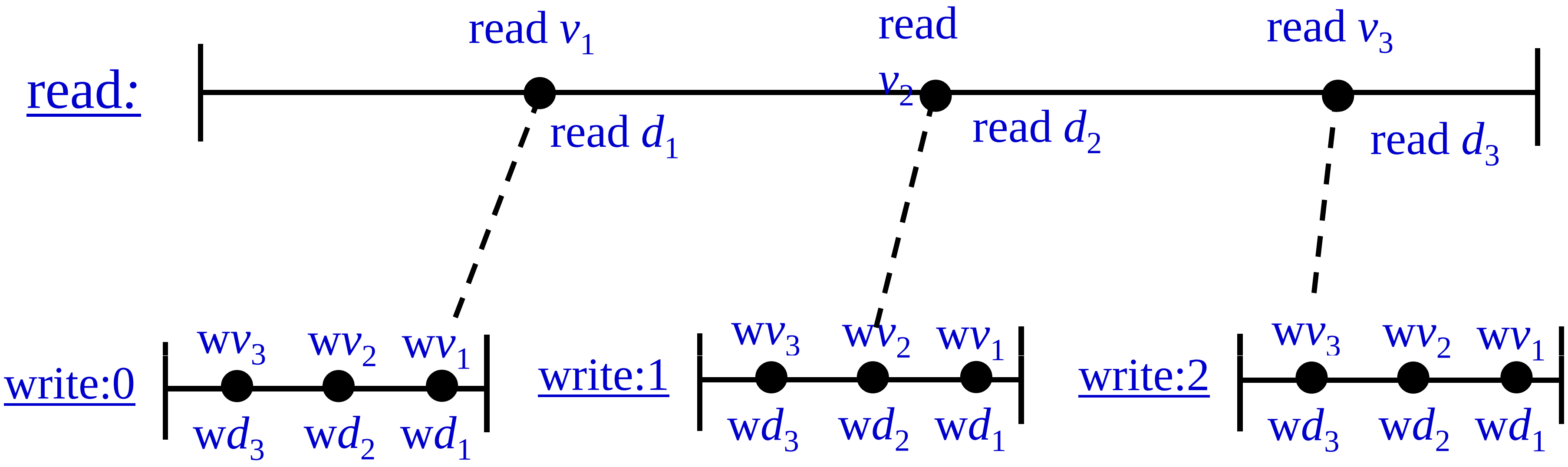
Theorem 1

If v is always written from right to left, then a read from left to right obtains a value

$$v_1^{[k1,l1]}\, v_2^{[k2,l2]} \ldots v_m^{[km,lm]}$$

where $k_1 \le l_1 \le k_2 \le l_2 \le \ldots \le k_m \le l_m$.

Example: $v = v_1 v_2 v_3 = d_1 d_2 d_3$



read: read v_1 read d_1 read v_2 read d_2 read v_3 read d_3

write:0 wv_3 wd_3 wv_2 wd_2 wv_1 wd_1

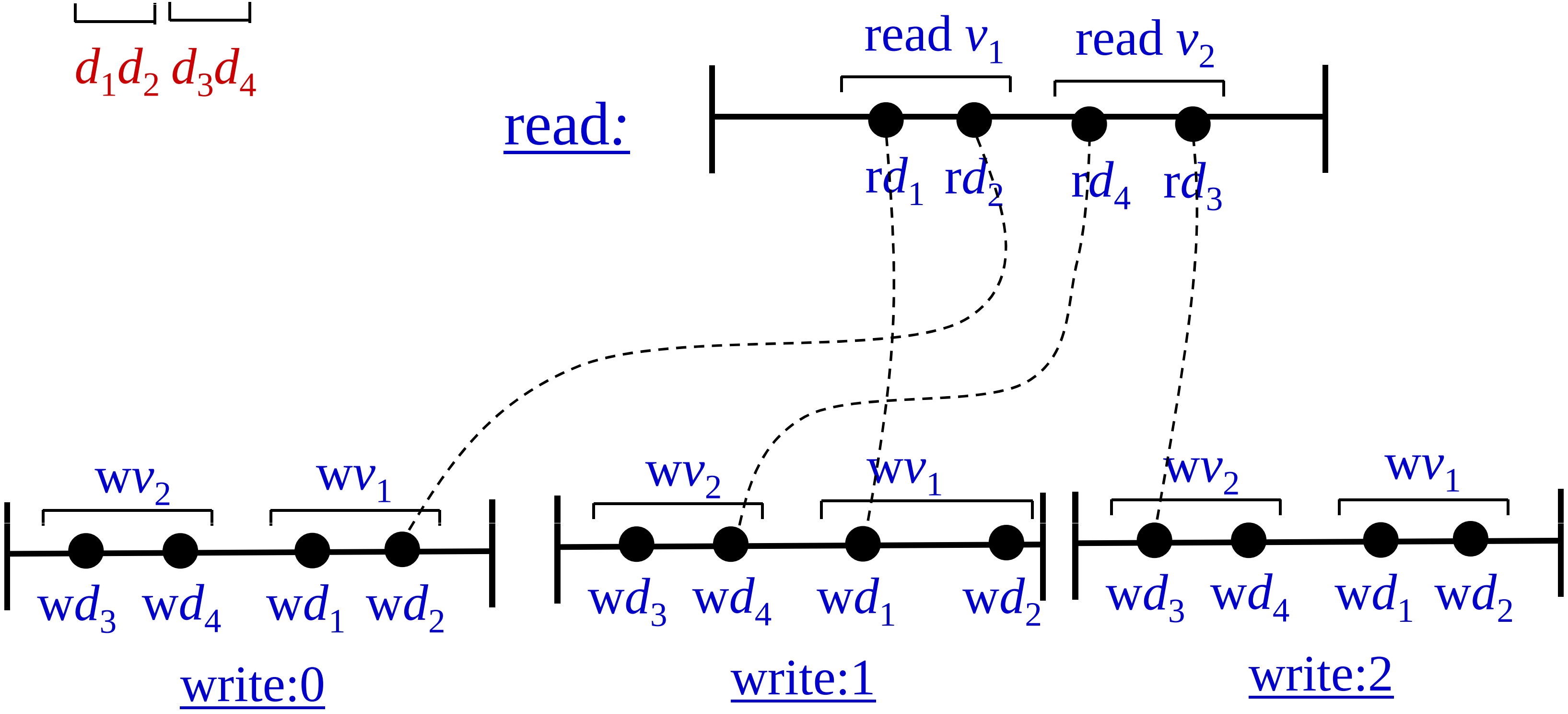
write:1 wv_3 wd_3 wv_2 wd_2 wv_1 wd_1

write:2 wv_3 wd_3 wv_2 wd_2 wv_1 wd_1

Read reads $v_1^{[0,0]}\, v_2^{[1,1]}\, v_3^{[2,2]}$.

Another Example

$$v = \underbrace{v_1}_{d_1 d_2} \; \underbrace{v_2}_{d_3 d_4}$$



Read reads $v_1^{[0,1]} v_2^{[1,2]}$.

Proof Obligation

- Assume reader reads $V2^{[k1,\ l1]}\ D^{[k2,\ l2]}\ V1^{[k3,\ l3]}$.
- **Proof Obligation:** $V2^{[k1,\ l1]} = V1^{[k3,\ l3]} \Rightarrow k_2 = l_2$.

Proof

By Theorem 2,

$$V2^{[k1,l1]} \leq V2^{[l1]} \text{ and } V1^{[k3]} \leq V1^{[k3,l3]}. \qquad (1)$$

Applying Theorem 1 to V2 D V1,

$$k_1 \leq l_1 \leq k_2 \leq l_2 \leq k_3 \leq l_3\ . \qquad (2)$$

By the writer program,

$$l_1 \leq k_3 \Rightarrow V2^{[l1]} \leq V1^{[k3]}. \qquad (3)$$

(1), (2), and (3) imply

$$V2^{[k1,l1]} \leq V2^{[l1]} \leq V1^{[k3]} \leq V1^{[k3,l3]}.$$

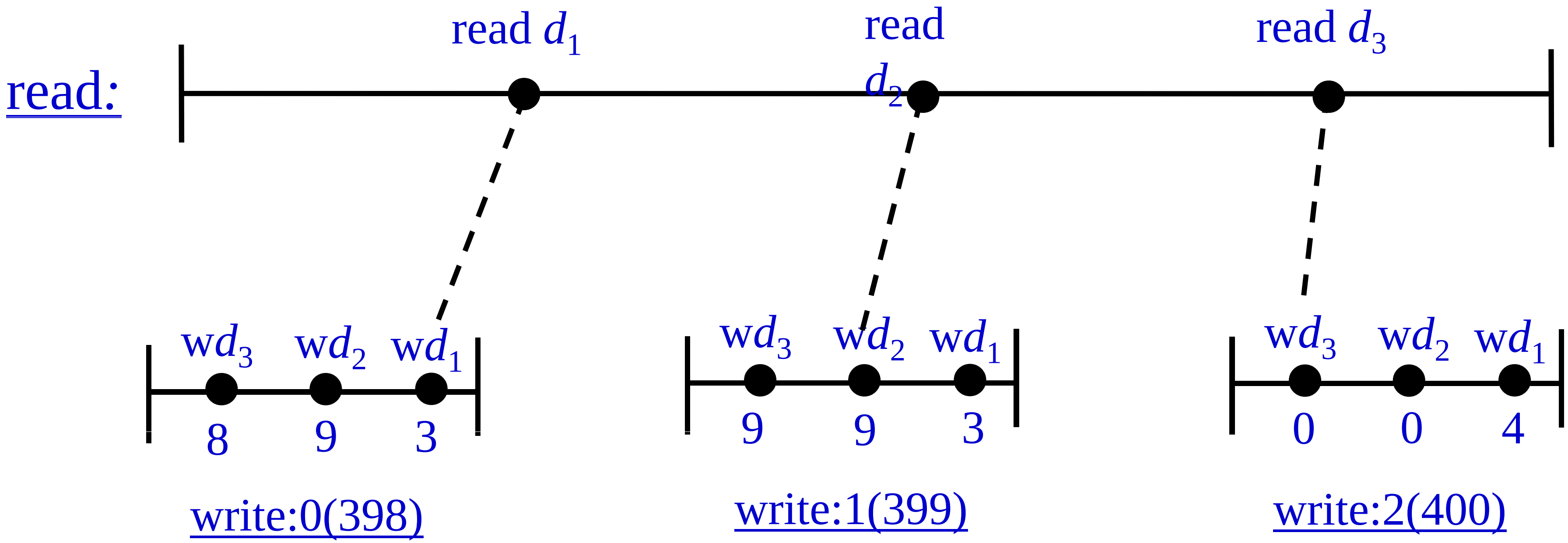
Hence, $V2^{[k1,l1]} = V1^{[k3,l3]} \Rightarrow V2^{[l1]} = V1^{[k3]}$

$\Rightarrow l_1 = k_3$, by the writer's program.

$\Rightarrow k_2 = l_2$ by (2).

Example of (a) in main theorem

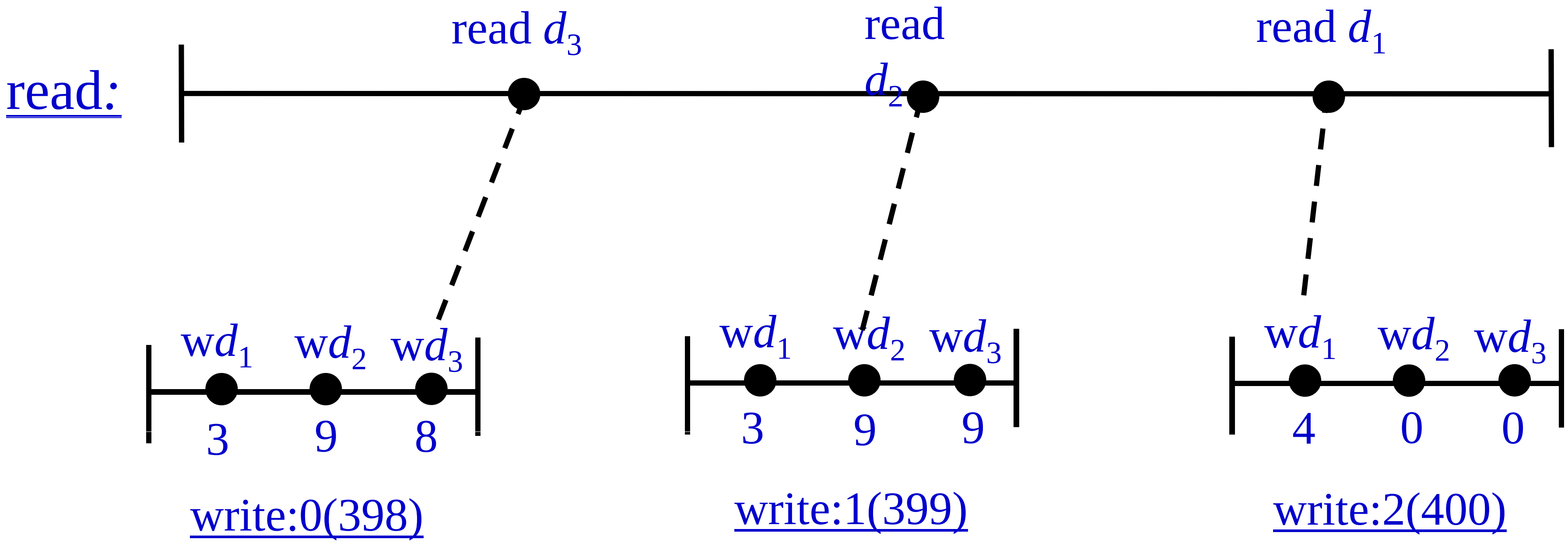
$v = d_1 d_2 d_3$



Read obtains $v^{[0,2]} = 390 < 400 = v^{[2]}$.

Example of (b) in main theorem

$v = d_1 d_2 d_3$



Read obtains $v^{[0,2]} = 498 > 398 = v^{[0]}$.

Supplemental Reading lock-free synch

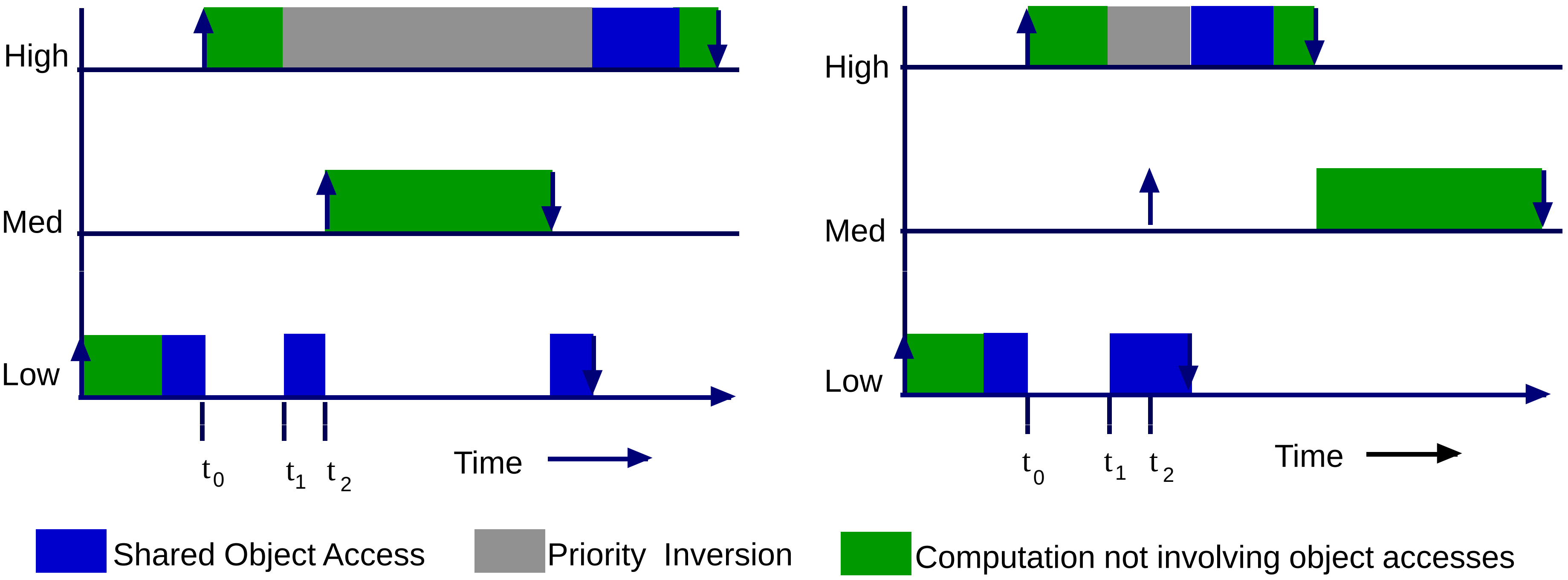
- check:
  - G.L. Peterson, "Concurrent Reading While Writing", ACM TOPLAS, Vol. 5, No. 1, 1983, pp. 46-55.
  - Solves the same problem in a wait-free manner:
    - guarantees consistency without locks and
    - the unbounded reader loop is eliminated.
  - First paper on wait-free synchronization.
- Now, very rich literature on the topic. Check also:
  - PhD thesis A. Gidenstam, 2006, CTH
  - PhD Thesis H. Sundell, 2005, CTH

Using Locks in Real-time Systems

The *Priority Inversion* Problem

Uncontrolled use of locks in RT systems can result in unbounded blocking due to *priority inversions*.

Solution: Limit priority inversions by *modifying task priorities*.



Dealing with Priority Inversions

- **Common Approach:** Use lock-based schemes that bound their duration (as shown).
  - **Examples:** Priority-inheritance protocols.
  - **Disadvantages:** Kernel support, very inefficient on multiprocessors.
- **Alternative:** Use non-blocking objects.
  - No priority inversions or kernel support.
  - Wait-free algorithms are clearly applicable here.
  - What about lock-free algorithms?
    - **Advantage:** Usually simpler than wait-free algorithms.
    - **Disadvantage:** Access times are *potentially unbounded*.
    - But for periodic task sets access times are also predictable!! (check further-reading-pointers)

Key issue in load balancing: Process Migration

- Transfer of sufficient amount of the state of a process from one machine to another; process continues execution on the target machine (processor)

Why to migrate?

- Load sharing/balancing
- Communications performance
  - Processes that interact intensively can be moved to the same node to reduce communications cost
  - move process to where the data reside when the data is large
- Availability
  - Long-running process may need to move if the machine it is running on will be down
- Utilizing special capabilities
  - Process can take advantage of unique hardware or software capabilities

Initiation of Migration

- Operating system: When goal is load balancing, performance optimization,
- Process: When goal is to reach a particular resource

What is Migrated?

- Must destroy the process on source system and create it on target system; PCB info and address space are needed
  - **Transfer-all:** Transfer entire address space
    - expensive if address space is large and if the process does not need most of it
    - Modification: **Precopy:** Process continues to execute on source node while address space is copied
      - Pages modified on source during pre-copy have to be copied again
      - Reduces the time a process cannot execute during migration
  - **Transfer-dirty:** Transfer only the portion of the address space that is in main memory and has been modified
    - additional blocks of the virtual address space are transferred on demand
    - source machine is involved throughout the life of the process
    - Variation: **Copy-on-reference**: Pages are brought on demand
      - Has lowest initial cost of process migration