

Operating System Support

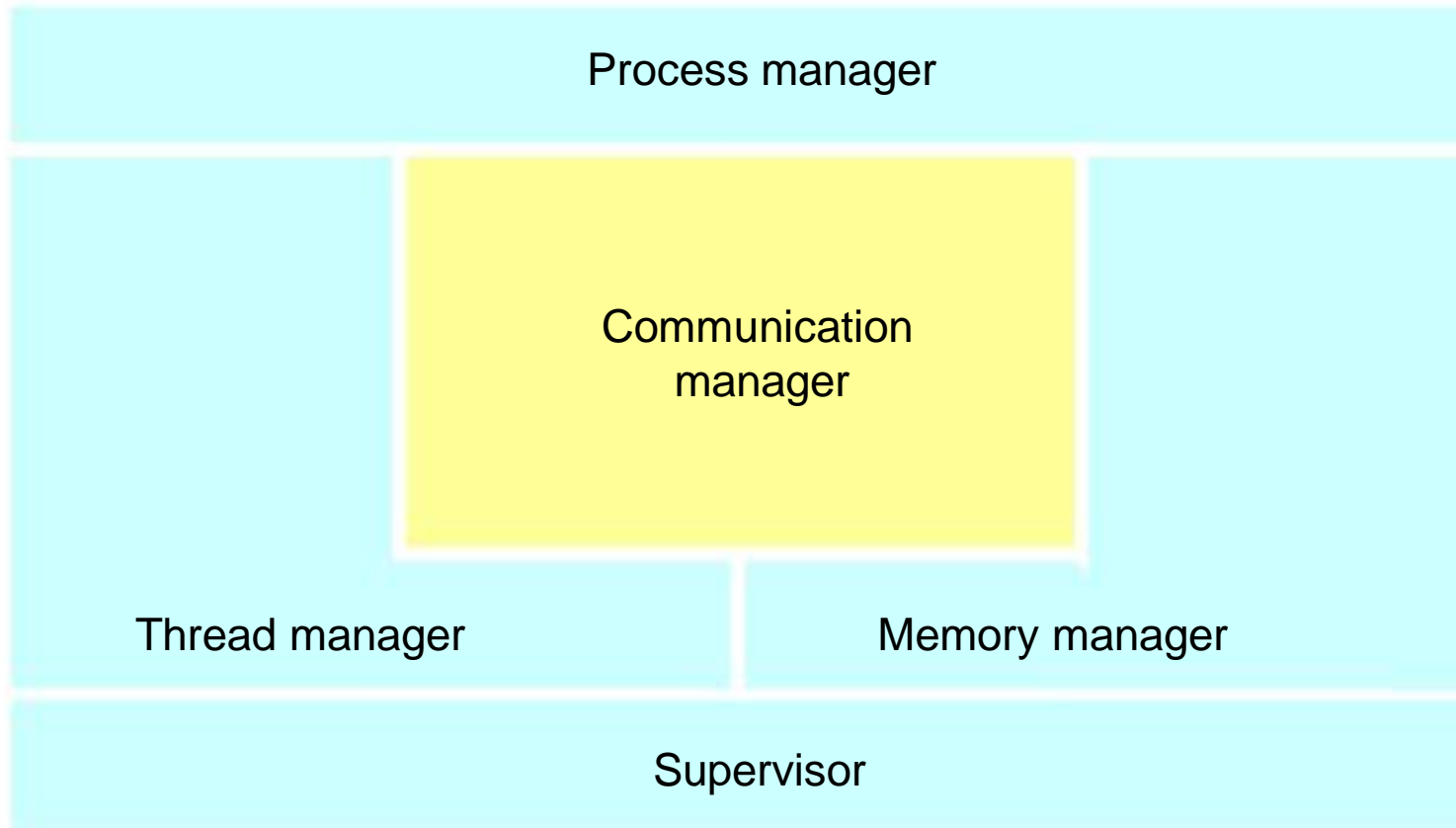
Overview

- Functionality of the Operating System (OS)
 - resource management (CPU, memory, ...)
- Processes and Threads
 - similarities, differences
 - multi-threaded servers and clients
- Implementation of...
 - communication primitives
 - invocation

Functionality of OS

- **Resource sharing**
 - **CPU** (single/multiprocessor machines)
 - concurrent processes/threads
 - communication/synchronisation primitives
 - process scheduling
 - **memory** (static/dynamic allocation to programs)
 - memory manager
 - **file storage** and **devices**
 - file manager, printer driver, etc
- **OS kernel**
 - implements CPU and memory sharing
 - abstracts hardware

Core OS functionality



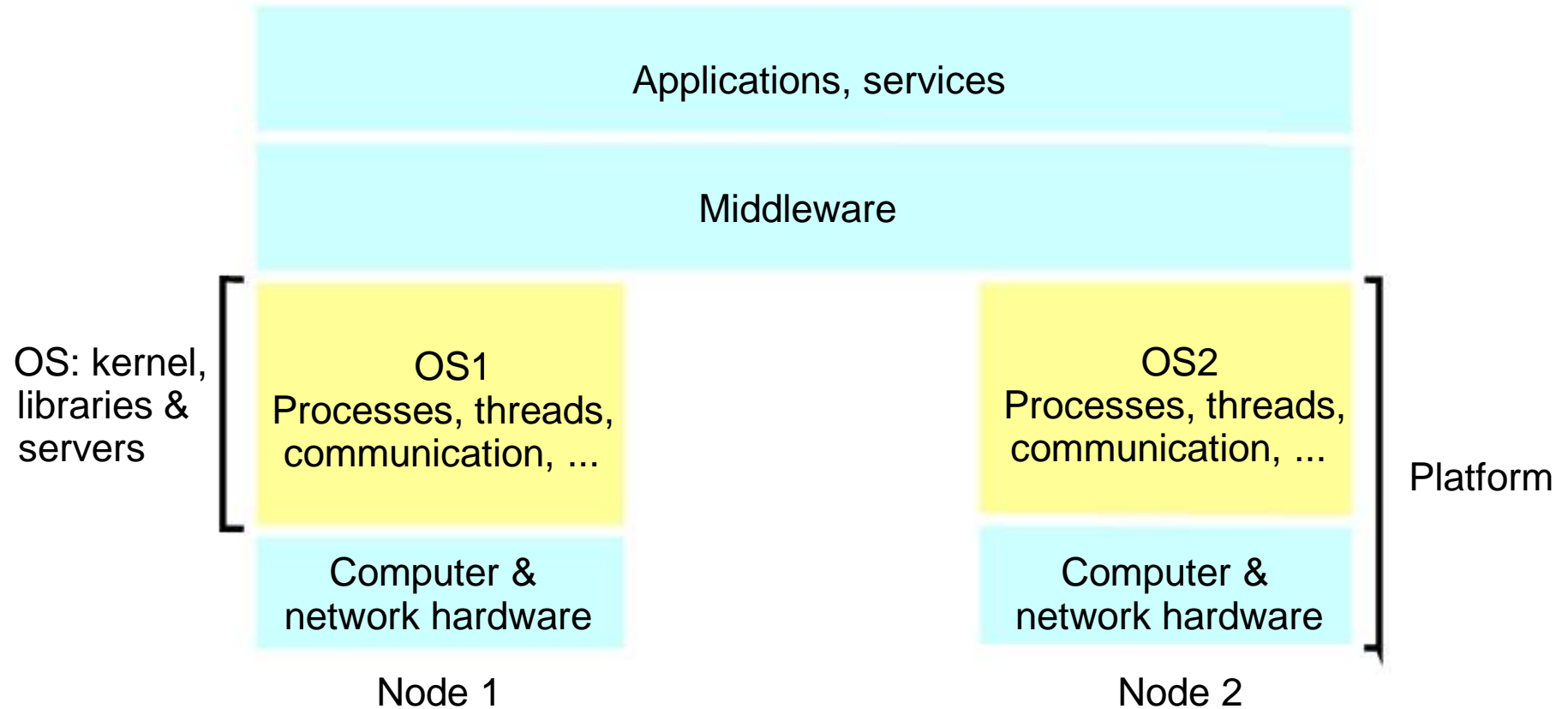
Core OS components

- Process manager
 - creation and operations on **processes** (= space+**threads**)
- Threads manager
 - threads creation, synchronisation, scheduling
- Communication manager
 - communication between threads (sockets, semaphores)
- Memory manager
 - physical (RAM) and virtual (disk) memory
- Supervisor
 - hardware abstraction (interrupts, exceptions, caches)

Why middleware again...

- Network OS (UNIX, Windows NT)
 - network transparent access for remote files (NFS)
 - no task/process scheduling across different nodes
- Distributed OS
 - transparent process scheduling across nodes
 - load balancing
 - none in use... cost of switching OS too high, load balancing not always easy to achieve
- Middleware
 - built on top of different network OSs
 - offers distributed resource sharing

System layers



In this lecture...

which **OS mechanisms** are needed for middleware

- **Concurrent processing** of client/server processes
 - creation, execution, etc
 - data encapsulation
 - protection against illegal access
- **Implementation** of invocation
 - communication (parameter passing, local or remote)
 - scheduling of invoked operations

Protection

- **Kernel**
 - complete access privileges to all physical resources
 - executes in **supervisor** mode
- **Application** programs
 - have own **address space**, separate from kernel and others
 - execute in **user** mode
- **Access** to resources
 - **calls** to kernel (system call trap), interrupts
 - **switch** to kernel address space
 - can be expensive in terms of time

Processes and threads

- Processes

- historically first abstraction of single **thread of activity**
- can run **concurrently**, CPU sharing if single CPU
- need own **execution environment**

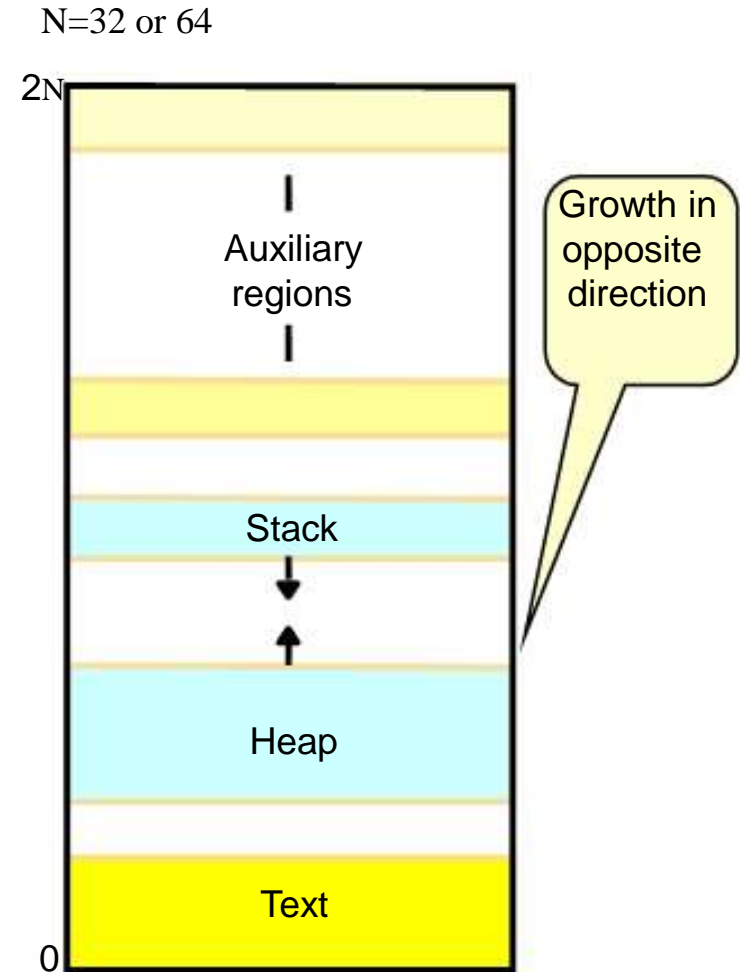
- address space, registers, synchronisation resources (semaphores)
- scheduling requires **switching** of environment

- **Threads** (=lightweight processes)

- can **share** execution environment
 - **no** need for expensive switching
- can be created/destroyed dynamically
 - **multi-threaded** processes
 - increased **parallelism** of operations (=speed up)

Process/thread address space

- Unit of virtual memory
- One or more **regions**
 - contiguous
 - non-overlapping
 - gaps for growth
- Allocation
 - **new** region for each thread
 - **sharing** of some regions
 - shared libraries, data,...



Process/thread creation

- OS kernel operation (UNIX fork, exec)
- Varying policies for
 - choice of host
 - clusters, single- or multi-processors
 - load balancing
 - creation of execution environment
 - allocate address space
 - initialise or copy from parent?

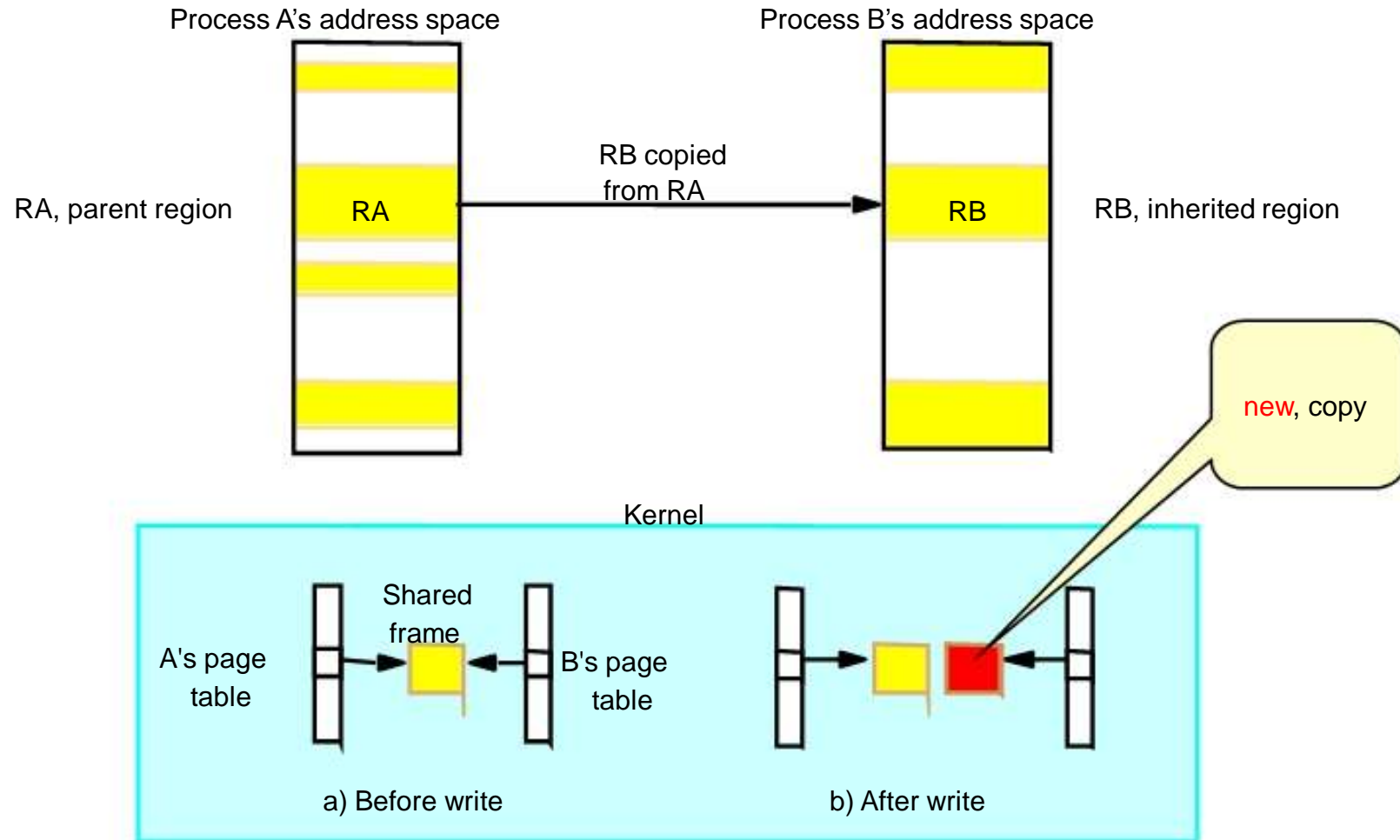
Choosing a host...

- Local or remote?
 - **migrate** process if load on local host high
- Load sharing to optimise throughput?
 - **static**: choose host **at random/deterministically**
 - **adaptive**: **observe state** of the system, measure load & use heuristics
- Many approaches
 - simplicity preferred
 - load measuring expensive.

Creating execution environment

- Allocate address space
- Initialise contents
 - fill with values from file or zeroes
 - for static address space but time consuming
 - copy-on-write
 - allow sharing of regions between parent & child
 - physical copying **only** when either attempts to modify (hardware page fault)

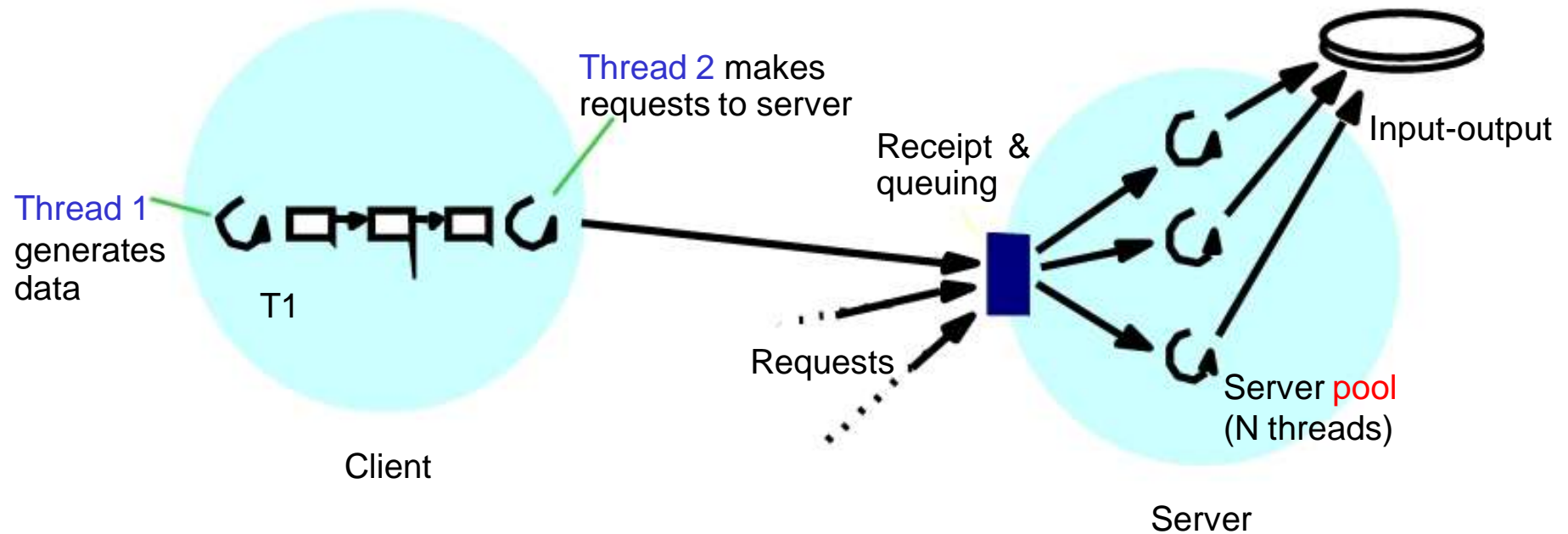
Copy-on-write



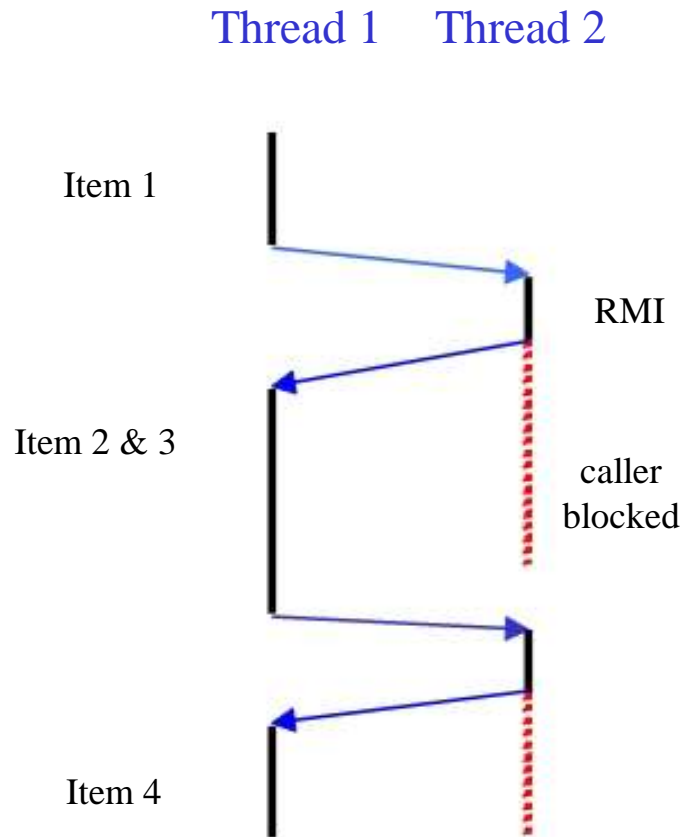
Role of threads in clients/servers

- On a **single CPU** system
 - threads help to logically decompose problem -
 - not much speed-up from CPU-sharing
- In a **distributed system**, more **waiting**
 - for remote invocations (blocking of invoker)
 - for disk access (unless caching)
 - obtain better **speed up** with threads

Multi-threaded client/server



Threads within clients



- Separate
 - data **production**
 - **RMI calls** to server
- Pass data via buffer
- Run **concurrently**
- Improved speed, throughput

Server threads and throughput

Assume **stream of client requests**, each 2ms processing + 8ms I/O.

- **Single** thread
 - max 100 client requests per second $= 1000 / (2 + 8)$
- **Two** threads, **no** disk caching
 - max 125 client requests per second $= 1000 / 8$
- **Two** threads, with **disk caching** (75% hit rate)
 - max 400 client requests per second $= 1000 / (0.75 * 0 + 0.25 * 8)$

Multi-threaded server architectures

- Worker pool

-fixed pool of worker threads, size does not change - can accommodate priorities but inflexible

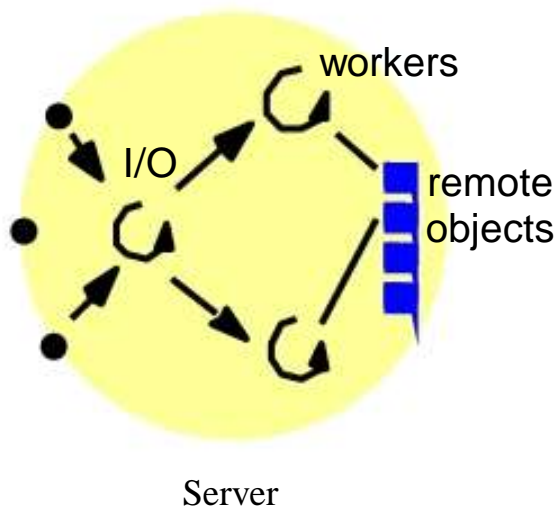
- Other architectures

- thread-per-request -
thread-per-connection -
thread-per-object

- Physical parallelism

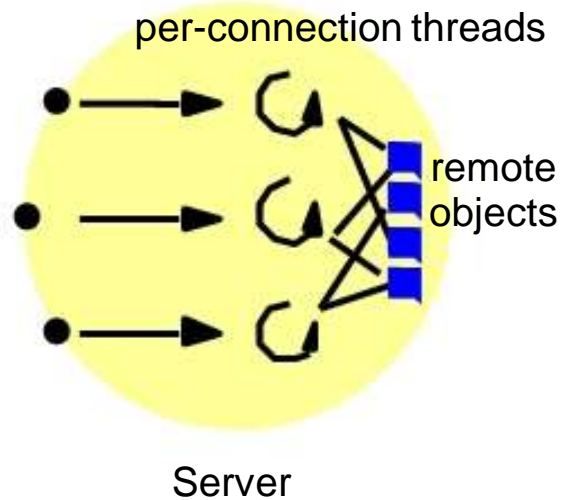
- multi-processor machines (casper, SoCS file server; noo-noo)

Thread-per-request



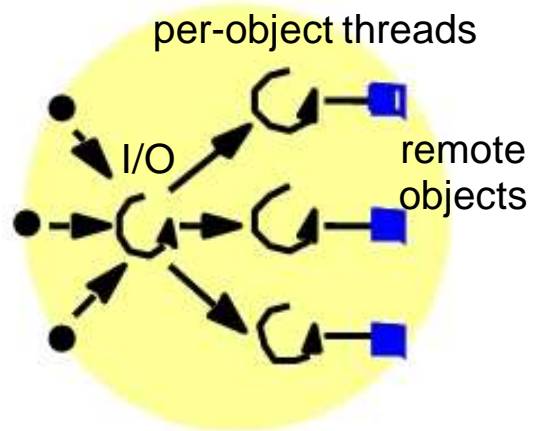
- Spawns
 - **new** worker for each request
 - worker destroys itself when finished
- Allows **max throughput**
 - no queuing
 - no I/O delays
- **but** overhead of creation & destruction high

Thread-per-connection



- Create **new** thread for **each connection**
- **Multiple** requests
- Destroy thread on close
- Lower o/heads
- **but** unbalanced load

Thread-per-object



As per-connection, but
new thread created **for
each object**.

Why threads not processes?

- Process context switching
 - requires save/restore of execution environment
 - registers, program counters, etc
- Threads within a process
 - cheaper to create/manage
 - no need to save execution environments (shared between threads)
 - resource sharing more efficient and convenient
 - but less protection from interference by other threads

Storing execution environment

Execution environment	Thread
Address space tables	Saved processor registers
Communication interfaces, open files	Priority and execution state (such as BLOCKED)
Semaphores, other synchronisation objects	Software interrupt handling information
List of thread identifiers	Execution environment identifier
Pages of address space resident in memory; hardware cache entries	

An aside: Java threads

- Class Thread

- constructor/destructor, SUSPENDED/RUNNABLE - priorities (useful for servlets, dynamic web pages)

- Synchronisation

- monitors (keyword synchronised) - at most one thread within monitor

- Scheduling

- Preemptive (suspended at any time), non-preemptive
 - no real-time thread scheduling

- More info www.cdk3.net

Java threads: management

`Thread(ThreadGroup group, Runnable target, String name)`

Creates a new thread in the **SUSPENDED** state, which will belong to `group` and be identified as `name`; the thread will execute the `run()` method of `target`. `setPriority(int newPriority)`, `getPriority()`

Set and return the thread's priority.

`run()`

A thread executes the `run()` method of its target object, if it has one, and otherwise its own `run()` method (`Thread` implements `Runnable`).

`start()`

Change the state of the thread from **SUSPENDED** to **RUNNABLE**. `sleep(int millisecs)`

Cause the thread to enter the **SUSPENDED** state for the specified time. `yield()`

Enter the **READY** state and invoke the scheduler. `destroy()`

Destroy the thread.

Java threads: synchronisation

`thread.join(int millisecs)`

Blocks the calling thread for up to the specified time until thread has terminated. `thread.interrupt()`

Interrupts thread: causes it to return from a blocking method call such as `sleep()`. `object.wait(long millisecs, int nanosecs)`

Blocks the calling thread until a call made to `notify()` or `notifyAll()` on object wakes the thread, or the thread is interrupted, or the specified time has elapsed.

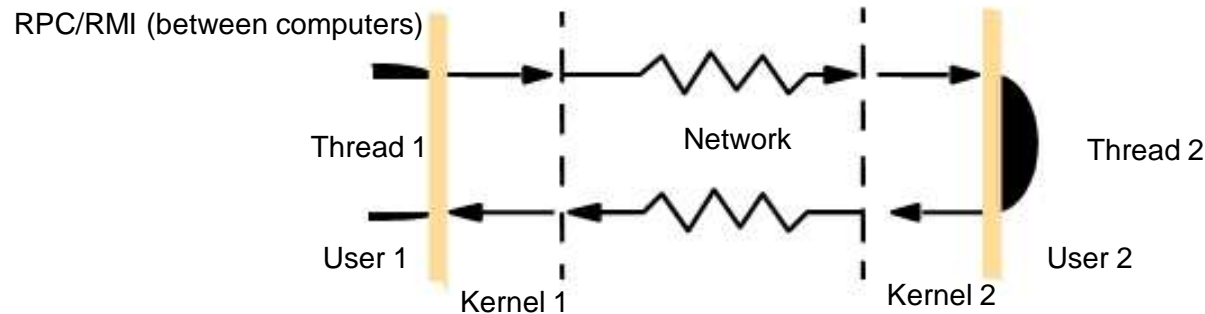
`object.notify()`, `object.notifyAll()`

Wakes, respectively, one or all of any threads that have called `wait()` on object.

Implementation of invocation

- Types of invocation
 - system call, RMI/RPC call, sending a message...
- Performance critical!
 - very high number of invocation per system lifetime
 - high latencies over WANs, Internet
- Counting cost of invocation
 - does it cross address space or not
 - synchronous or asynchronous?
 - over the network or within computer?

Costing invocations over network



- **Latency** (=time of **null** invocation)
 - 0.1millisecond for RPC vs fraction of microsecond for local call
- **Delay** (=total RPC/RMI time experience by user)
 - marshalling, thread switching, which protocol, etc
- Need to design OS carefully!

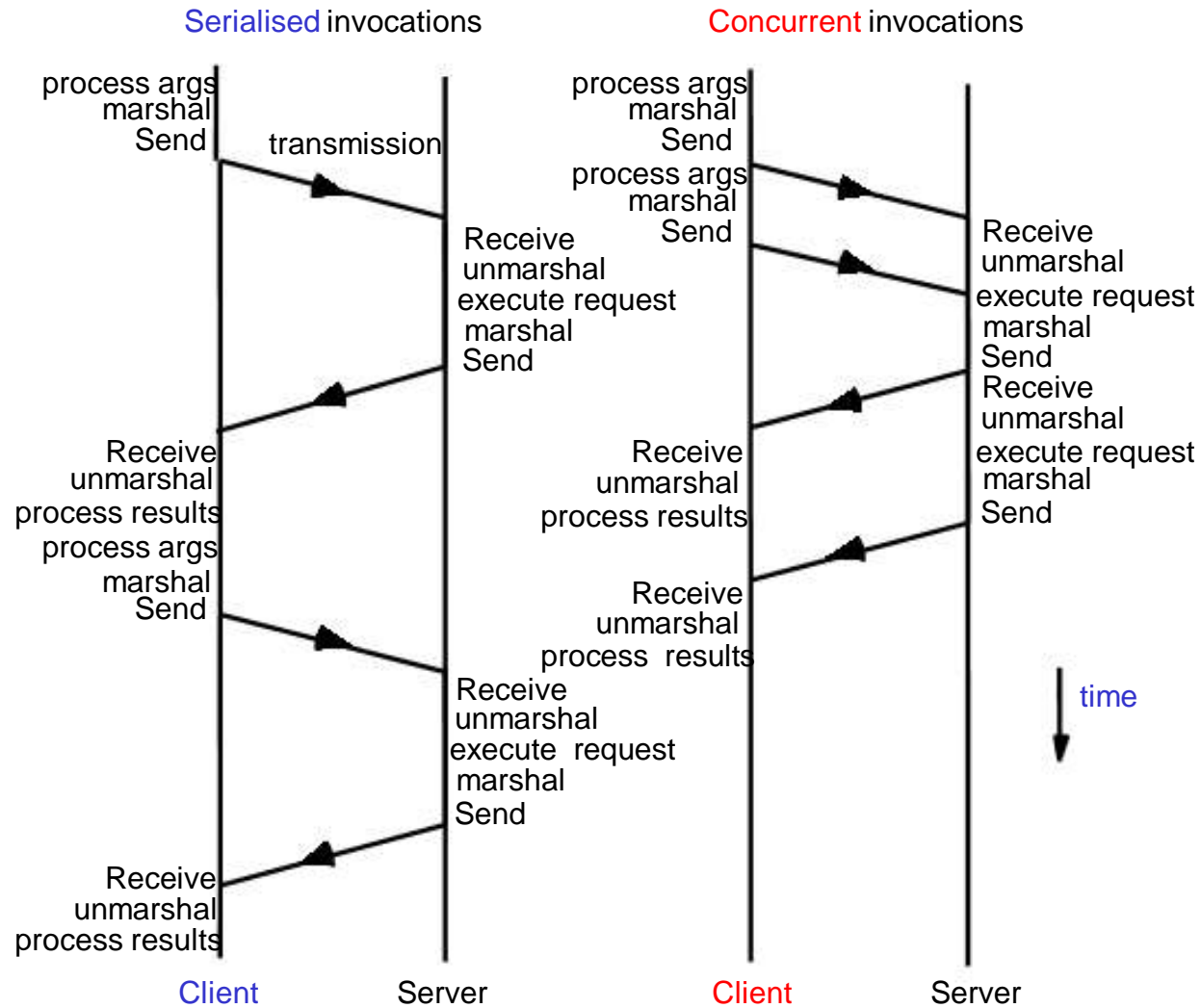
Factors affecting RPC/RMI delays

- Marshalling
- Data copying
 - user to kernel, across network, etc
- Packet initialisation
 - protocol headers, checksums
- Thread scheduling, context switching
- Waiting for acknowledgement
 - TCP or UDP?

Concurrent invocations

- Idea (similar to client threads earlier)
 - **blocking** invocations
 - perform them **concurrently**
- Example: web browser
 - issues **separate** HTTP GET requests for images within webpage
 - performed **concurrently**
- Gains
 - improved total delay and throughput
 - communication overlaps with rendering

Serialised and concurrent invocations



Asynchronous invocation

- **Non-blocking** invocation
 - client makes call (cf Mercury obtains promise)
 - continues processing
- Response
 - sometimes not needed
 - otherwise, separate **call to collect** results, - then claim on **promise** (test if results ready, block until results ready)
- Improved delay and throughput

Summary

- OS support **crucial to performance** of distributed systems

-threads/process management - communication

(sockets), protocols - support for

asynchronous/concurrent invocation

- Design issues

- structure and relationship of **kernel** & **middleware**

- selection of **multi-threaded** or **multi-processor** architecture

- understanding system requirements

- max number of **requests**, min acceptable **delay**, **throughput**
- network **latency**, **bandwidth**, etc