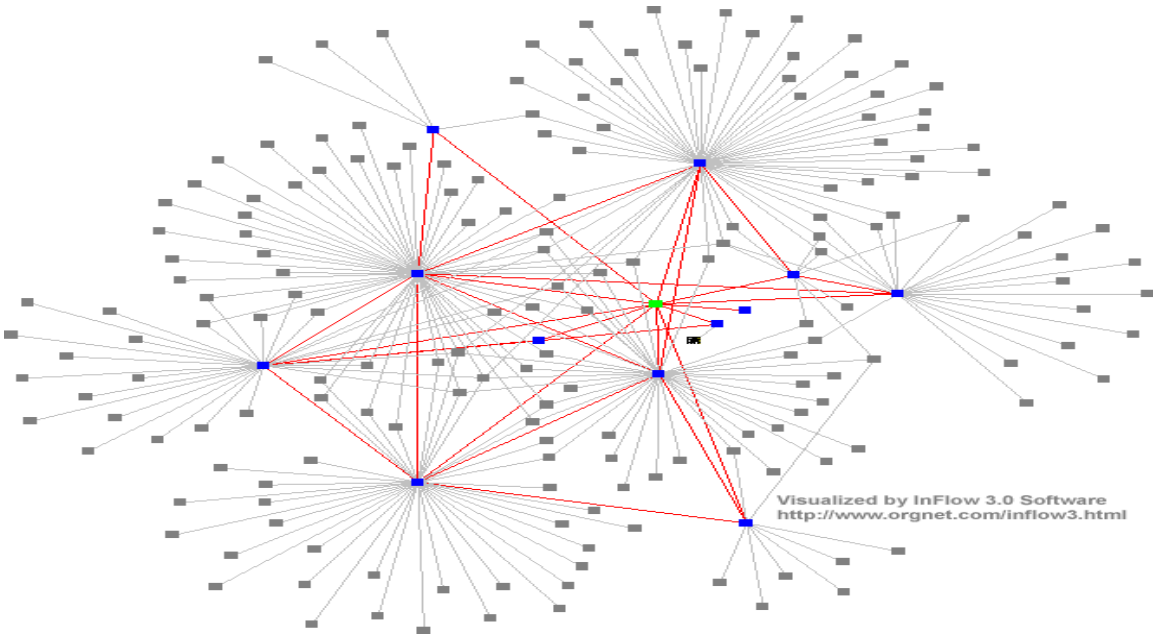
DCS Processes/Threads

OS support for DCS



Dr. Sunny Jeong. spjeong@uic.edu.cn

With Thanks to Prof. G. Coulouris, Prof. A.S. Tanenbaum and Prof. S.C Joo

Overview

- Functionality of the Operating System (OS)
  - resource management (CPU, memory, …)
- Processes and Threads
  - Similarities V.S. differences
  - multi-threaded servers and clients
- Implementation of...
  - communication primitives
  - Invocations

Functionality of OS

- Resource sharing
  - CPU (single/multiprocessor machines)
    - concurrent processes/threads
    - communication/synchronization 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

. 操作系统的功能（Functionality of OS）
操作系统的主要功能是资源管理，**包括：**

（1）CPU**管理**
处理多个**进程（Process）和线程（Thread）**的并发执行。

采用调度算法（Process Scheduling）**管理**CPU时间，使系统高效运行。

提供进程间通信（IPC, Inter-Process Communication）**和同步机制**（Synchronization Primitives），**例如信号量**（Semaphore）**和共享内存**（Shared Memory）。

（2）**内存管理**
静态/动态内存分配：负责程序运行时的内存分配和释放。

虚拟内存管理（Virtual Memory）：**使用**页表（Page Table）将物理内存与进程的虚拟地址空间映射，允许程序运行超出物理内存限制。

内存回收机制：包括垃圾回收（Garbage Collection）**和内存碎片整理**（Compaction）。

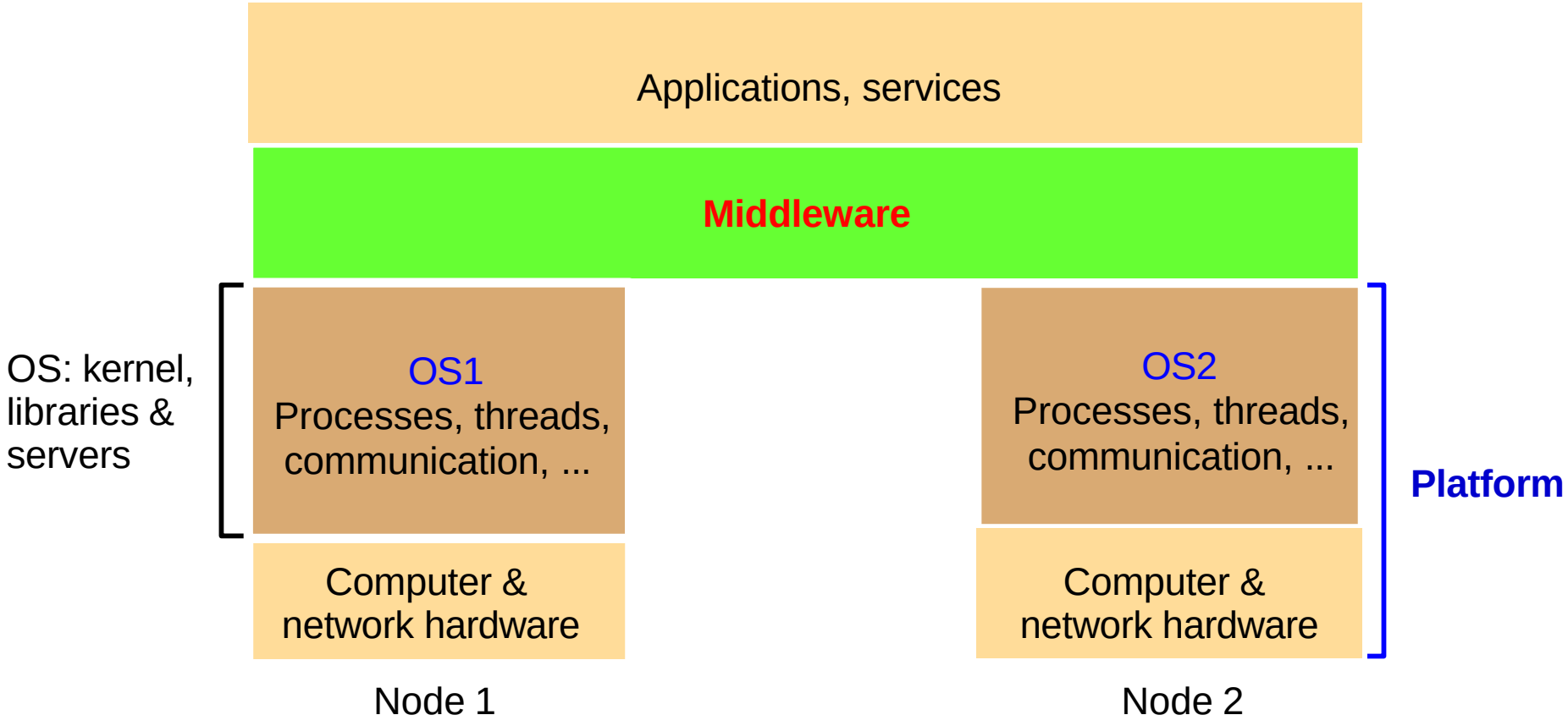
（3）文件存储与设备管理
文件系统（File System）：**如** NTFS、EXT4 负责文件存储和访问。

设备驱动程序（Device Drivers）：控制输入输出设备（例如打印机、磁盘、显示器）。

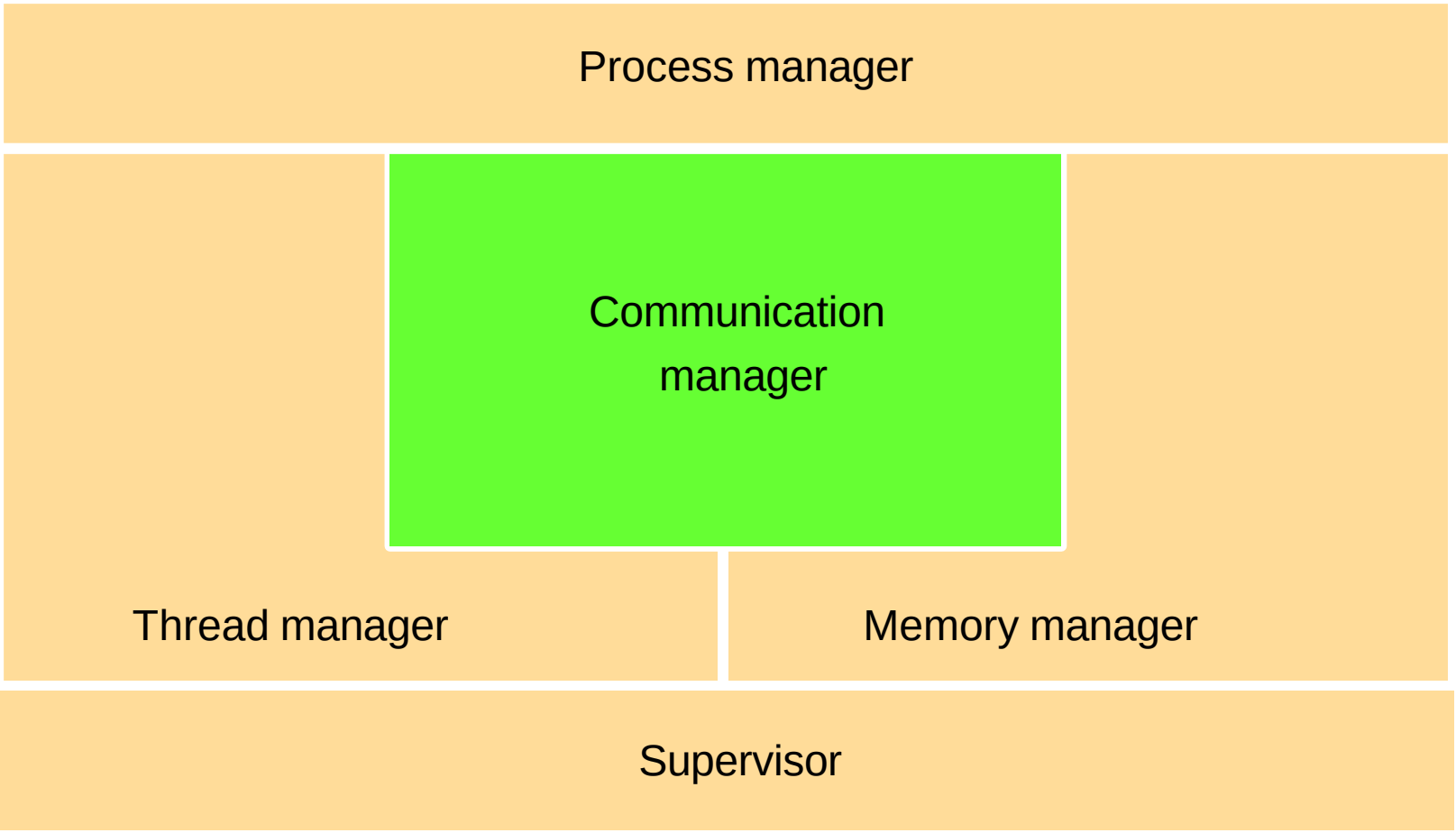
（4）操作系统内核（Kernel）
负责CPU、内存和设备的共享管理。

负责抽象底层硬件，提供统一接口给应用程序

Distributed System layers



Core OS functionality



Core OS components

- Process manager
  - creation and operations on processes (= address space + threads)
- Threads manager
  - threads creation, synchronization, scheduling
- Communication manager
  - communication between threads (sockets, semaphores)
    - ✔ in different processes(concurrency)
    - ✔ on different computers(parallel)
- Memory manager
  - physical (RAM) and virtual (disk) memory
- Supervisor
  - hardware abstraction (dispatching of interrupts, exceptions, system call traps)
  - control of memory managements and hardware cache

什么是进程（Process）？
进程是程序的执行实例，它拥有独立的内存空间和系统资源。

进程是操作系统资源分配的基本单位，**比如**CPU时间、内存、文件等。

每个进程都至少有一个线程（通常是主线程）。

进程的特点
独立性：每个进程有自己的地址空间，进程之间的数据不能直接共享。

资源分配：每个进程都有自己的内存、文件句柄、系统资源等。

切换成本高：进程切换时，操作系统需要切换地址空间，**更新**CPU缓存等，开销较大。

进程的例子
你打开两个 Word **文档，每个** Word 文档是一个独立的进程，它们互不影响。

你同时运行微信、浏览器、VS Code，它们都是不同的进程。

2. 什么是线程（Thread）？
线程是进程中的一个执行单元，线程是更小的调度单位。

线程共享进程的资源（**如内存、文件句柄**），但拥有自己的栈和寄存器。

线程的特点
共享资源：同一进程中的线程共享内存，可以直接访问同一块数据。

轻量级：线程切换的开销比进程切换小，多个线程之间切换速度快。

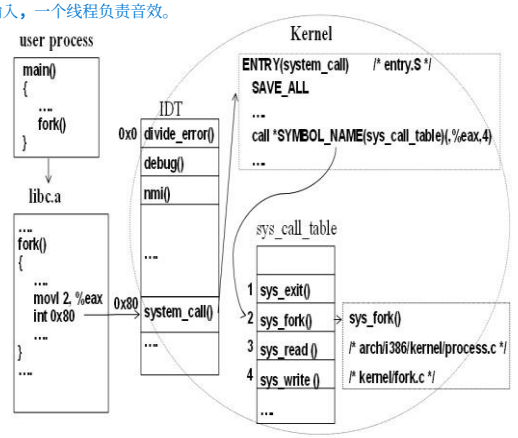
数据同步问题：由于多个线程共享内存，**需要使用****锁（Lock）**或信号量**（Semaphore）****来避免数据竞争。

线程的例子
浏览器：打开多个标签页，每个标签页是一个线程，共享同一个浏览器进程的资源。

Word 自动保存：一个线程负责用户输入，另一个线程负责自动保存文档。

游戏引擎：一个线程渲染画面，一个线程处理输入，一个线程负责音效。

그림 B-2. fork()가 실행될 때의 흐름

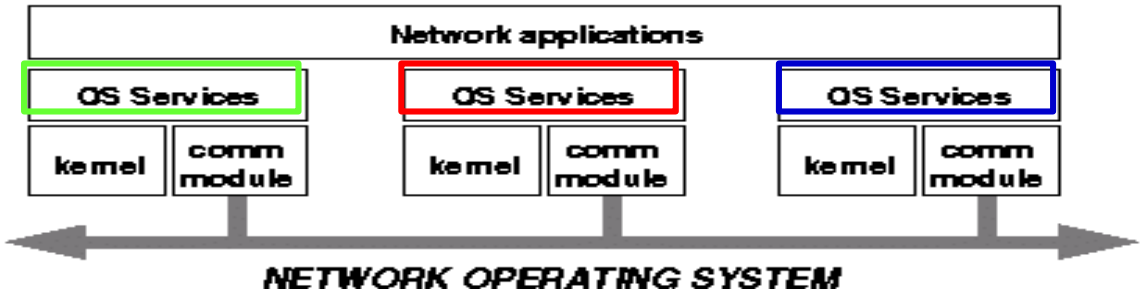


OS and middleware in DCS

Network Operating System

Some characteristics:

- Each computer has its own operating system with networking facilities
- Computers work independently (i.e., they may even have different operating systems)
- Services are tied to individual nodes (ftp, telnet, WWW)
- Highly file oriented (basically, processors share *only* files)



OS and middleware in DCS

1. 网络操作系统（Network OS, NOS）
什么是网络操作系统？
网络操作系统是用于多个计算机通过网络协同工作的操作系统，主要用于文件共享、远程访问和基本的网络服务。每台计算机独立运行自己的 OS，它们通过网络连接并进行数据交换。

NOS 的关键特点
远程文件访问
NOS 允许多个计算机访问共享文件系统，例如：

NFS（Network File System）：使得不同计算机能访问同一个远程文件系统，就像访问本地文件一样。

Samba（Windows-Linux 互通的文件共享协议）。

AFS（Andrew File System），用于大规模分布式文件系统。

独立的任务调度
每个计算机上的 OS 独立管理进程，不会在多个计算机之间调度任务。

进程不能跨计算机迁移，所有计算任务都在本地执行。

基本的网络服务
NS 提供的核心服务包括：

远程登录（rlogin, telnet, SSH）：远程访问其他计算机的终端。

文件传输（ftp, NFS, Samba）：允许多个用户共享文件。

Web 服务（WWW）：提供网站访问能力。

VPN（虚拟专用网络）：提供安全的远程访问。

❑Network OS

- ex) UNIX, Windows NT->Windows Server
- network transparent access for remote files (NFS)
- no task/process scheduling across different nodes
- services
  - ✔ rlogin, telnet, ftp, WWW, SSH, VPN…

OS and middleware in DCS

2. 分布式操作系统（Distributed OS, DOS）
什么是分布式操作系统？
分布式操作系统（DOS）让多个计算机协同工作，提供统一的资源管理和调度，让用户感觉像是在使用单个计算机，而不是多个独立的系统。

DOS 的关键特点
资源共享
计算任务可以分配到不同的计算机执行，多个节点共享 CPU、内存、存储等资源。

例如，在一个分布式数据库系统中，不同节点可以存储和管理不同部分的数据。

并行处理
DOS 允许多个计算机并行运行进程，提高计算速度。

例如，Hadoop 将大数据任务分解，并行执行多个子任务。

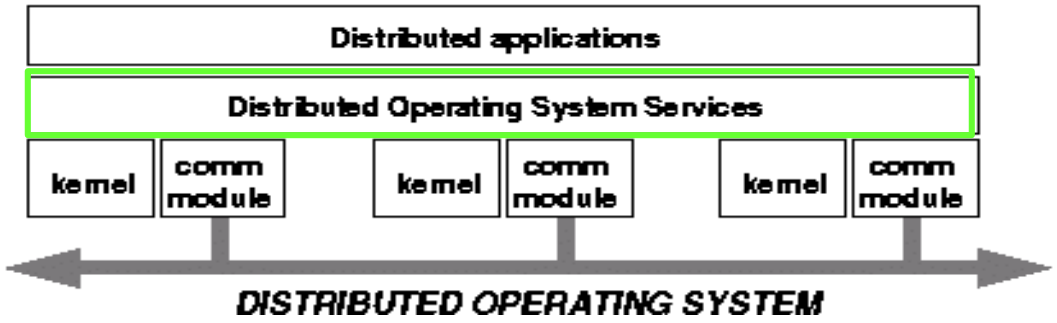
容错性
即使一个节点发生故障，任务可以在其他节点上重新分配，提高系统的可靠性。

例如，Google 的 Borg 系统使用任务重启机制，防止计算任务因硬件故障中断。

Distributed Operating System

Some characteristics:

- OS on each computer knows about the other computers
- OS on different computers generally the same
- Services are generally (transparently) distributed across computers



特点 网络操作系统（NOS） 分布式操作系统（DOS）
进程管理 每台计算机独立管理进程 进程可以跨多个节点迁移和调度
任务调度 仅在本地执行 可以跨计算机调度任务
文件系统 远程访问（如 NFS） 透明访问（如分布式文件系统）
用户体验 用户感知多个计算机 用户感知为单一计算机
应用案例 企业服务器、数据中心 云计算、大数据、分布式 AI

OS and middleware in DCS

❑ Distributed OS

- transparent process scheduling across nodes
- load balancing
- limited in use:

  cost of switching OS too high, load balancing not always easy to achieve

NOS versus DOS

Item	NOS	MCOS	MPOS
Transparency	No	Yes	Yes
Same OS/node?	No	Yes	Yes
Communication	Shared files	Messages	Shared memory
Resource management	Per node only	Global, distributed	Global, central

MCOS = Multicomputer operating system
MPOS = Multiprocessor operating system

What is a Distributed Operating System (DOS)?

A Distributed Operating System (DOS) is a system where multiple independent computers work together as a single system. It allows multiple machines to share resources and execute processes in a distributed manner.

Key Characteristics of DOS

Resource Sharing: Uses multiple machines to distribute workload.

Parallel Processing: Distributes tasks among different nodes.

High Availability & Fault Tolerance: System continues operation even if some nodes fail.

Transparency: Users experience as a single unified system.

Examples of Distributed Operating Systems

- Amoeba: A research-based distributed OS (Prof. A Tanenbaum.)
- Plan 9 - Bell Labs: Uses a distributed file system approach.
- Inferno - Bell Labs: On Virtual Machine, A light version DOS.
- Sprite - UC Berkeley: focusing on process migration.
- Barrelfish: A Microsoft and ETH Zürich research project.

DS services Inspired by DOS

- Google Borg & Kubernetes: Cluster management and orchestration.
- Apache Mesos: Distributed resource management.
- Hadoop YARN: Distributed computing for big data.
- Microsoft Azure Service Fabric: Cloud-based microservice orchestration.

Reality of DOS

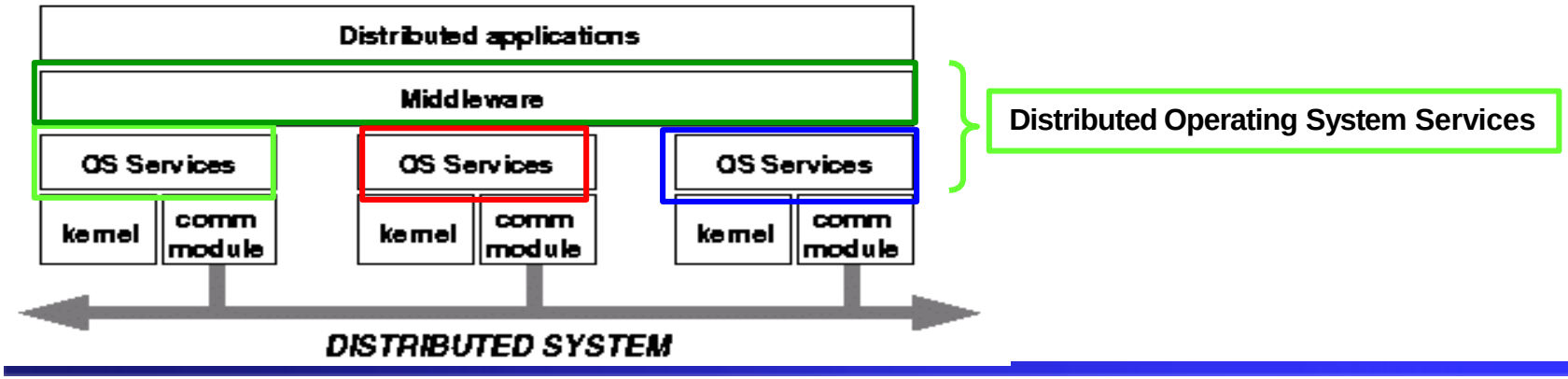
- DOS as a fully commercial OS is not widely used.
- However, many cloud computing and distributed computing frameworks use DOS principles.
- Technologies like Kubernetes, Hadoop, and Azure Service Fabric apply DOS concepts.
- The future may see more advanced distributed OS implementations.

OS and middleware in DCS

Distributed System (Middleware)

Some characteristics:

- OS on each computer need not know about the other computers : **NOS**
- OS on different computers need not generally be the same : **NOS**
- Services are generally (transparently) distributed across computers : **DOS**



OS and Middleware

- Middleware
  - built on top of different NOSs
  - offers distributed resource sharing
    - via remote invocations
  - Similar to functionalities of DOS possible

OS and Middleware

Need for Middleware

Motivation: Too many networked applications were hard or difficult to integrate:

- Departments are running different NOSs
- Integration and interoperability only at level of primitive NOS services
- Need for federated information systems:
  - Combining different databases, but providing a single view to applications
  - Setting up enterprise-wide Internet services, making use of existing information systems
  - Allow transactions across different databases
  - Allow extensibility for future services (e.g., mobility, teleworking, collaborative applications)
- Constraint: use the existing operating systems, and treat them as the underlying environment (they provided the basic functionality anyway)

OS and Middleware

Middleware Services

Communication services: Abandon primitive socket-based message passing in favor of:

- Procedure calls across networks
- Remote-object method invocation
- Message-queuing systems
- Advanced communication streams
- Event notification service

Information system services: Services that help manage data in a distributed system:

- Large-scale, systemwide naming services
- Advanced directory services (search engines)
- Location services for tracking mobile objects
- Persistent storage facilities
- Data caching and replication

Control services: Services giving applications control over when, where, and how they access data:

- Transaction processing
- Process migration
- Task scheduling

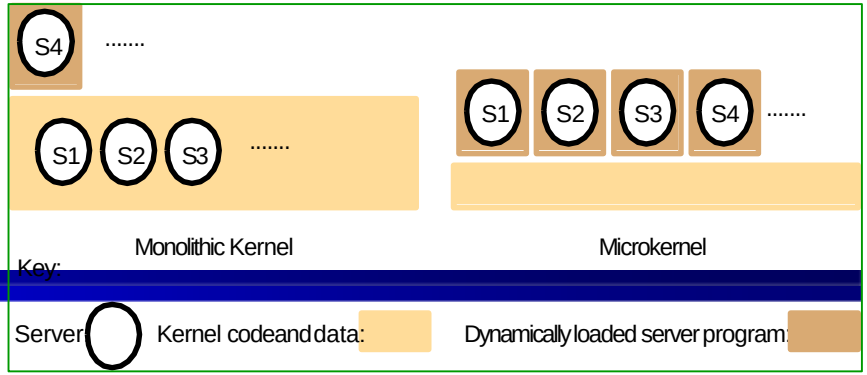
Security services: Services for secure processing and communication:

- Authentication and authorization services
- Simple encryption services
- Auditing service

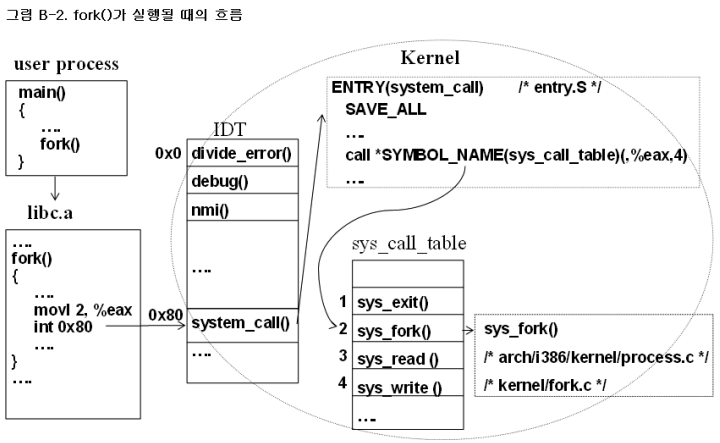
OS and Middleware

- OS mechanisms are needed for middleware
  - Encapsulation
  - Protection illegitimate
  - Concurrent control
- 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
  - sets up address spaces to protect processes, and provides virtual memory
  - Another process executes in user mode
- Application programs
  - have own address space, separate from kernel and others(=user mode)
  - execute in user mode
- Access to resources
  - calls to kernel (system call trap), interrupts(exception)
  - switch to kernel address space
  - can be expensive in terms of time



Threaded Applications

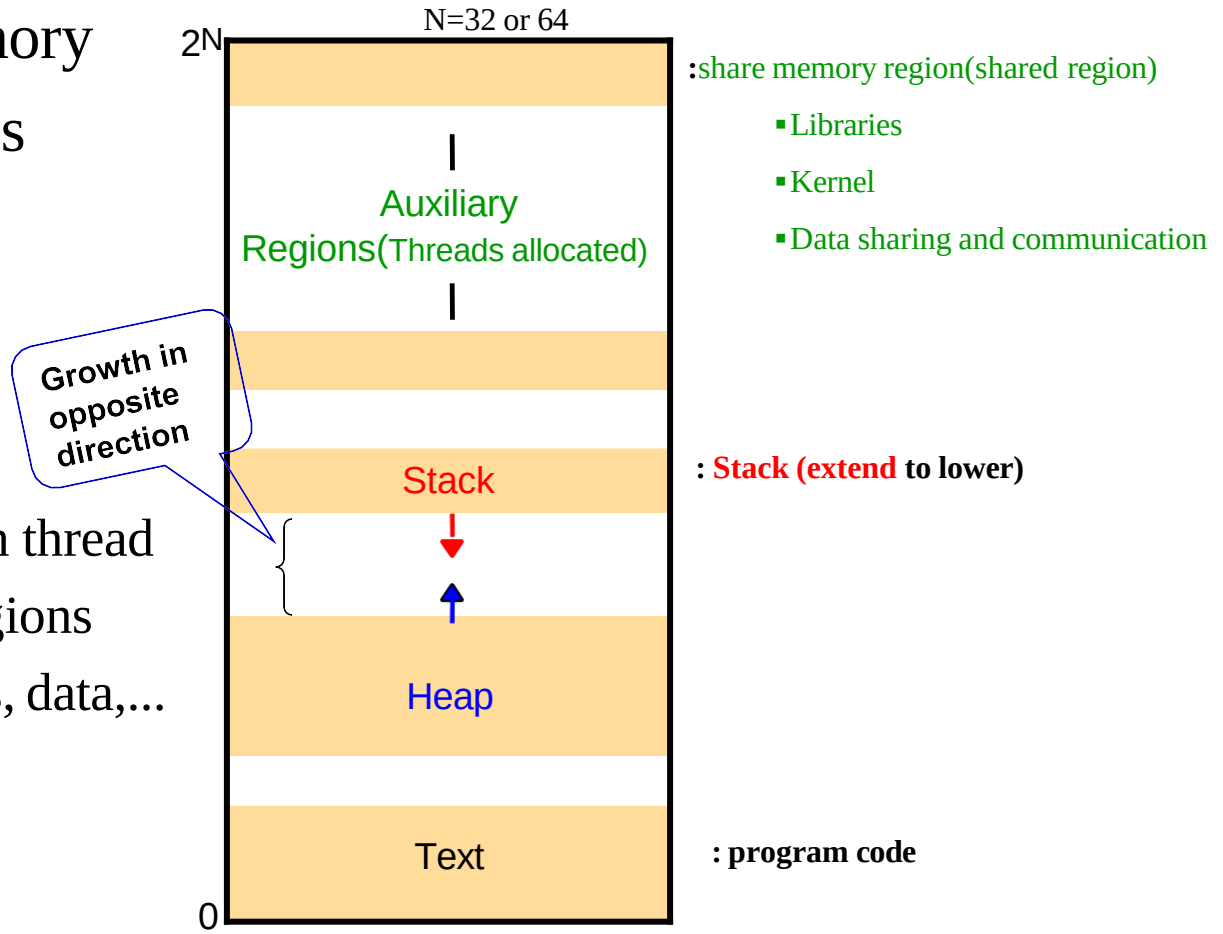
- Modern Systems
  - Multiple applications run concurrently!
  - This means that… there are multiple processes on your computer

Processes and threads

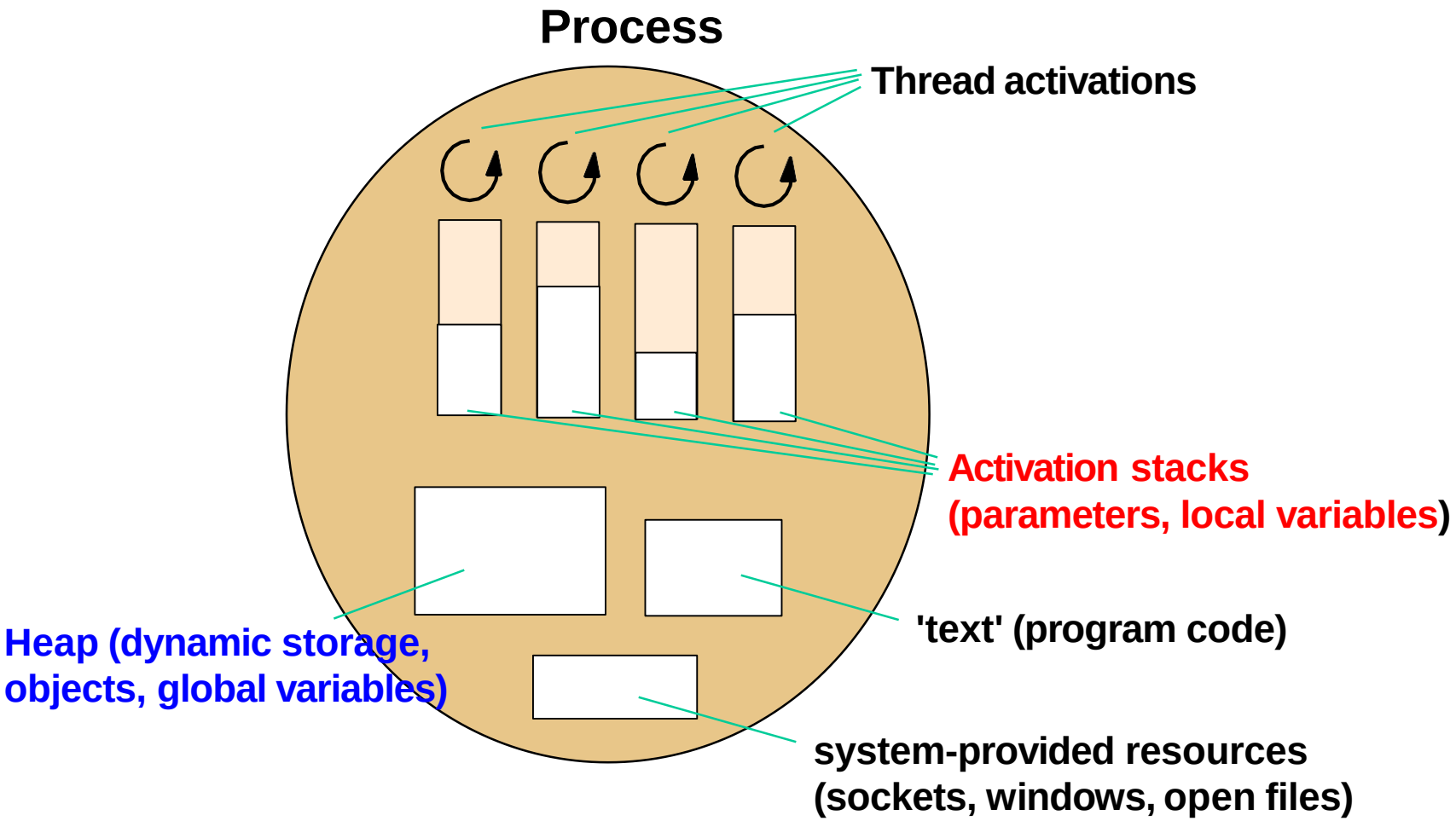
- Processes
  - historically first abstraction of single thread of activity
  - can run concurrently, CPU sharing if single CPU
  - need own execution environment
    - address space, registers, synchronization resources (semaphores)
  - scheduling requires switching of environment
- Threads (=lightweight processes)
  - can share an 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 concepts



Process/thread creation

- OS kernel operation (cf UNIX *fork*, *exec*)

- Varying policies for
  - choice of host
    - clusters, single- or multi-processors
    - load balancing
  - creation of execution environment
    - allocate address space
    - initialize or copy from parent?

Choosing a host...

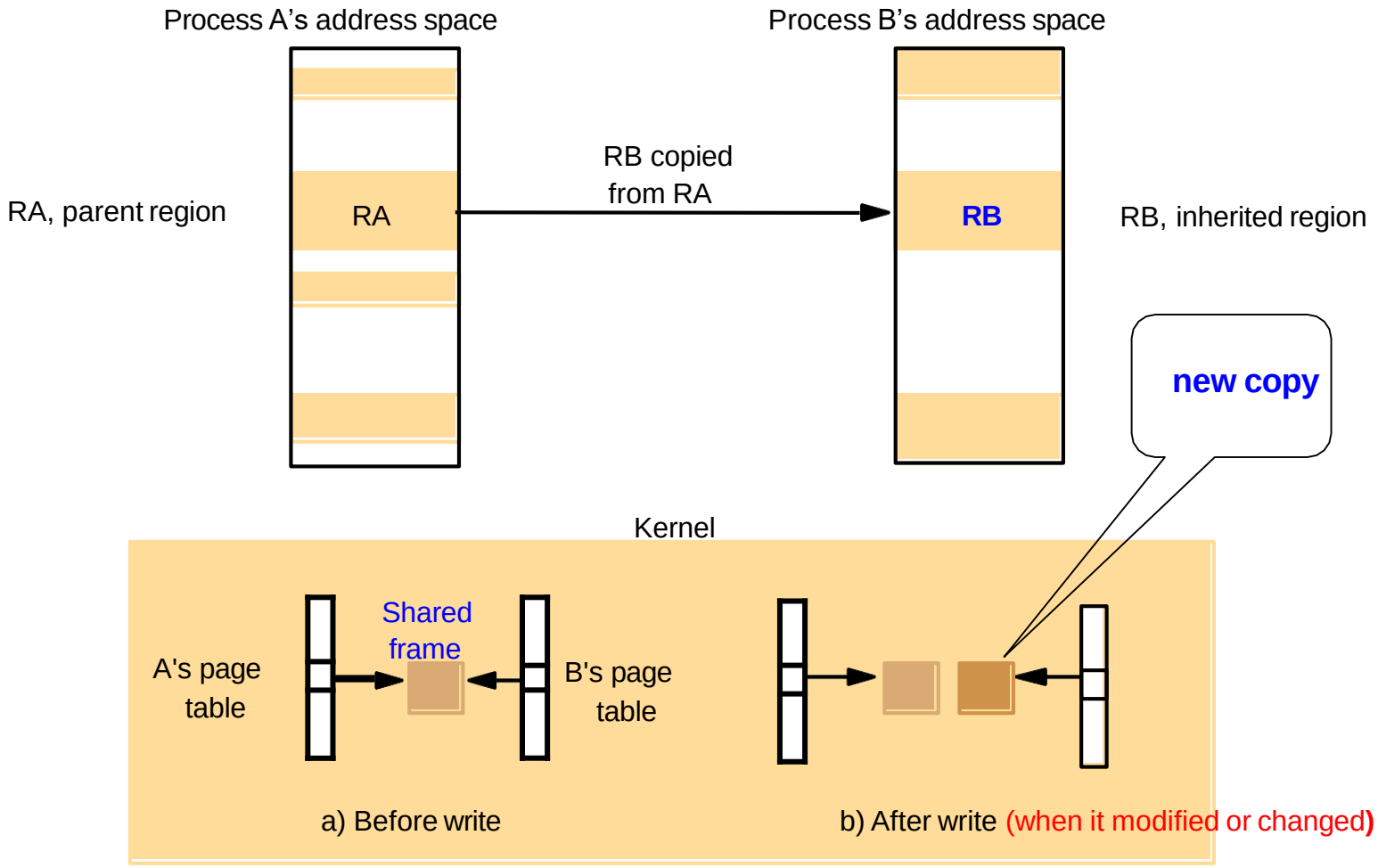
- Local or remote?
  - migrate process if load on local host is high
- Load sharing to optimize throughput?
  - static: choose host at random/deterministically
  - adaptive: observe state of the system, measure load & use heuristics
- Many approaches
  - simplicity preferred
  - load measuring expensive.

Creating execution environment

- Allocate address space

- Initialize 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

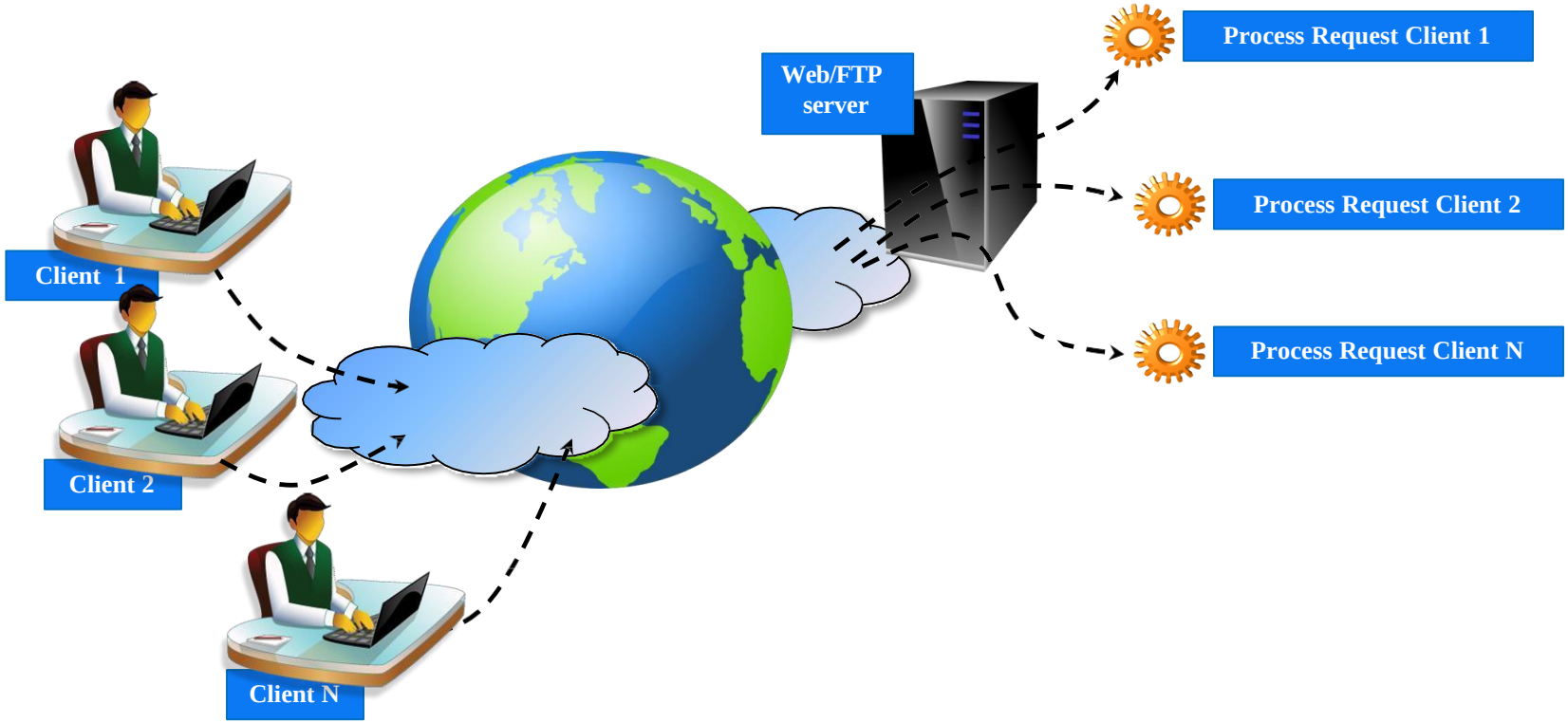


Threaded Applications

- Modern Applications
  - Example: Internet Browser + Youtube

Multithreaded Server: For Serving Multiple Clients Concurrently

- Modern Applications
  - Example: Multithreaded Web Server

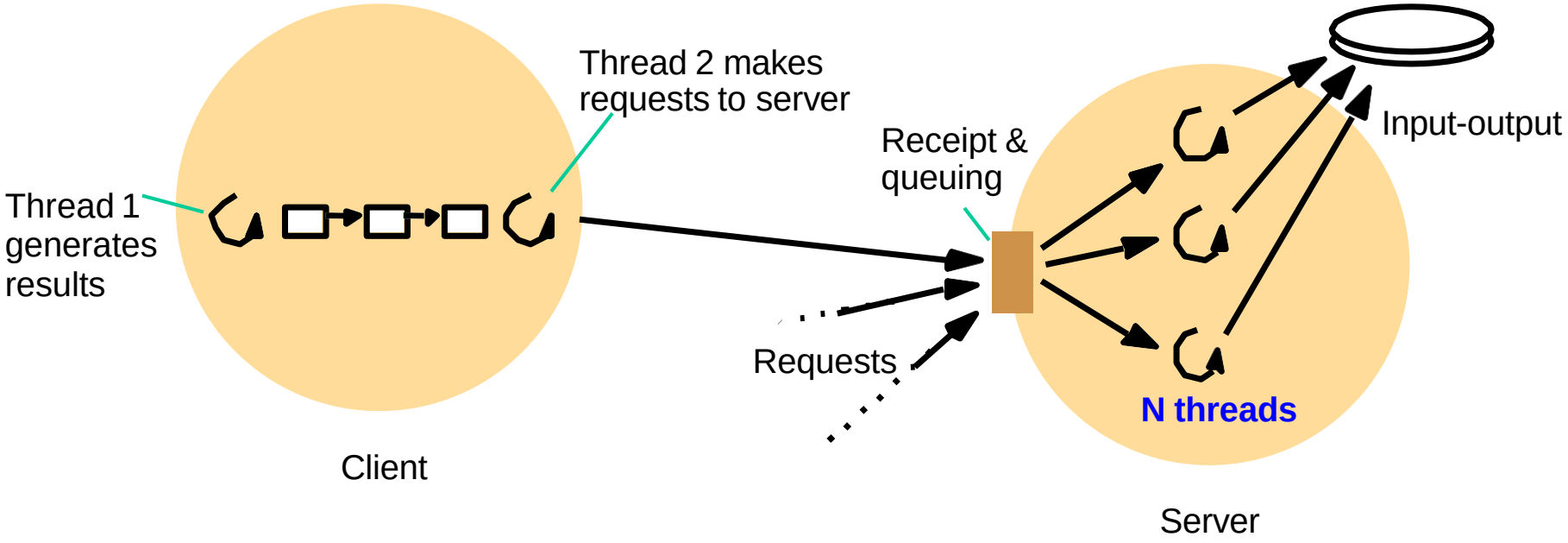


Role of threads in clients/servers

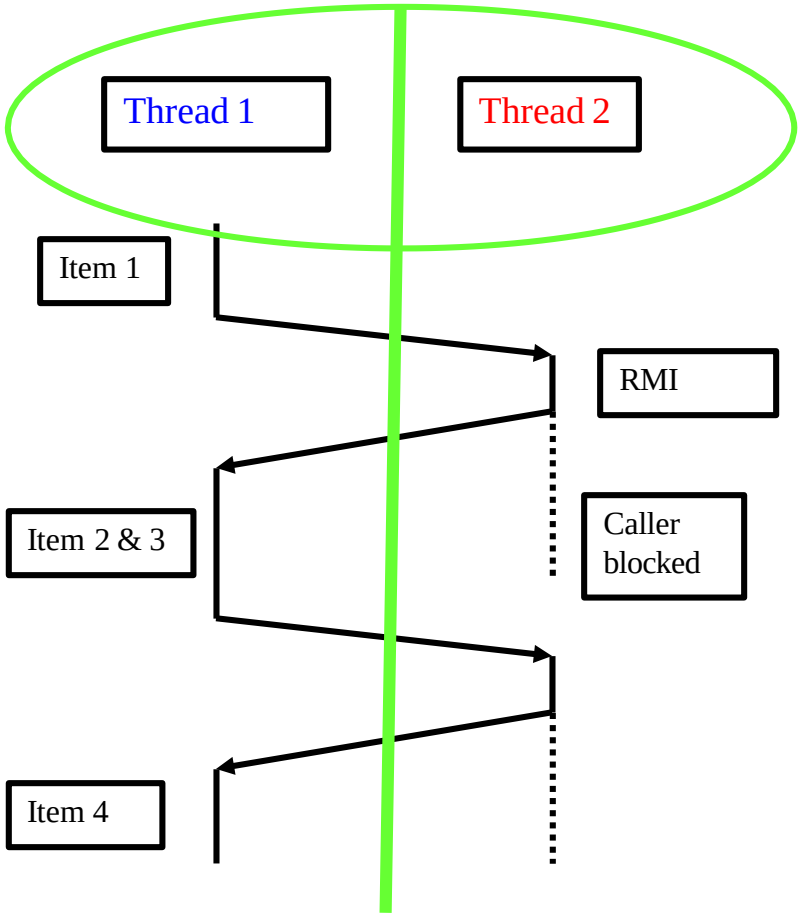
- On a single CPU system
  - threads help to logically decompose a given problem(program)
  - not much speed-up from CPU-sharing

- In a distributed system, more waiting
  - for remote invocations (blocking of invoker)
  - for disk access (unless caching)
  - But, obtain better speed up with threads

Multi-threaded client/server



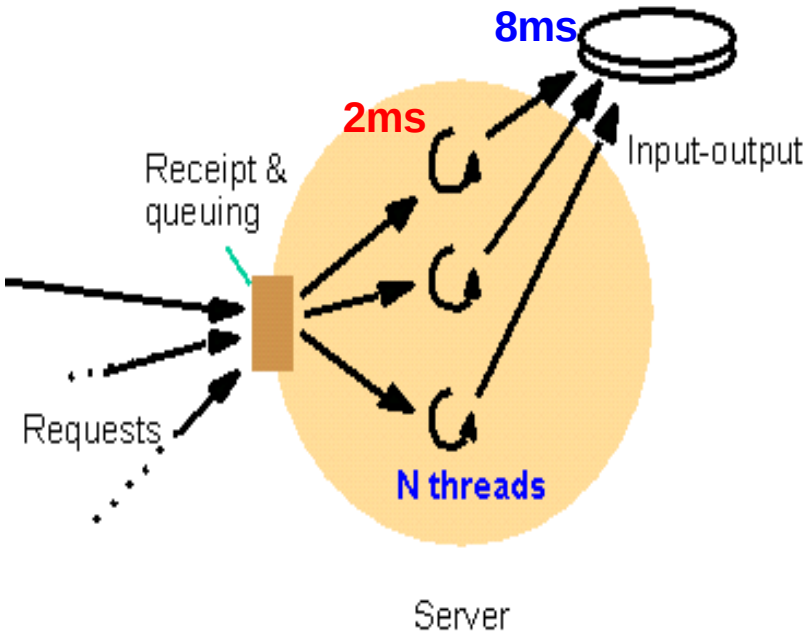
Threads within clients



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

Server threads and throughput

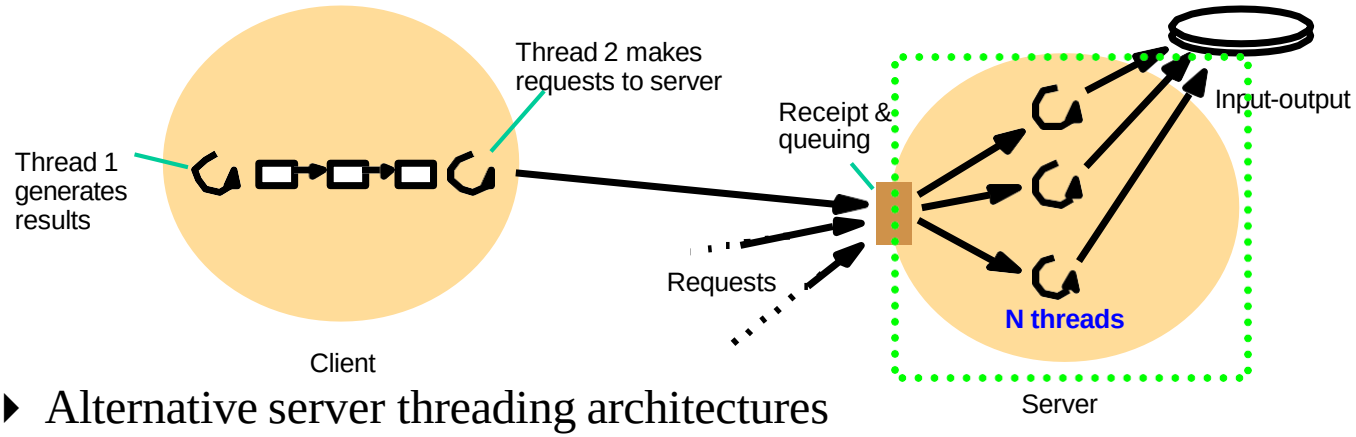
Assume stream of client requests,
(each client request time :
=*2ms for processing + 8ms for I/O*)
* 1 *sec* = 1000*ms*



- Single thread
  - max client requests per second ?
    =1000ms/(2+8)ms = 100 requests/sec
- Two threads, no disk caching
  - max client requests per second ?
    =1000ms/min(8, 8+2)ms = 125 requests/sec
- Two threads, with disk caching (75% hit rate)
  - max client requests per second ?
    *=1000ms/(0.75*0+0.25*8)ms=500 requests/sec*

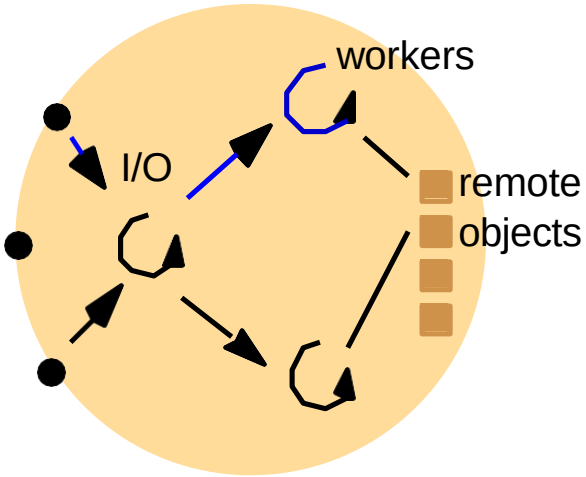
Multi-threaded server architectures

- Worker pool Architecture
  - fixed pool of worker threads, size does not change
  - can accommodate priorities but inflexible, I/O switching



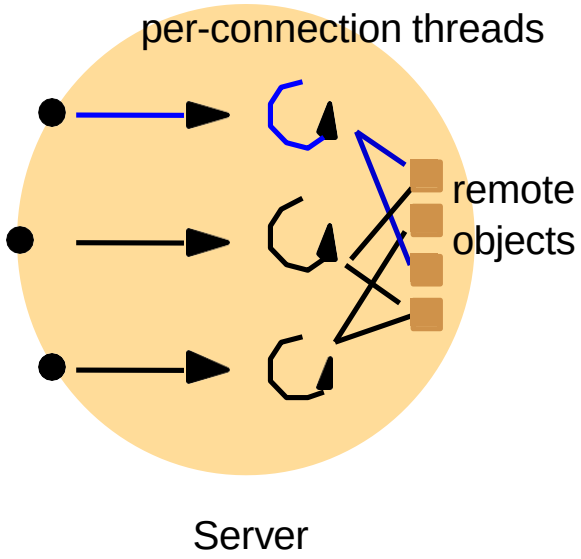
  - Alternative server threading architectures
    - ✔ thread-per-request architecture
    - ✔ thread-per-connection architecture
    - ✔ thread-per-object architecture
- Physical parallelism
  - multi-processor machines (cf. Casper, SoCS file server; noo-noo)

Thread-per-request



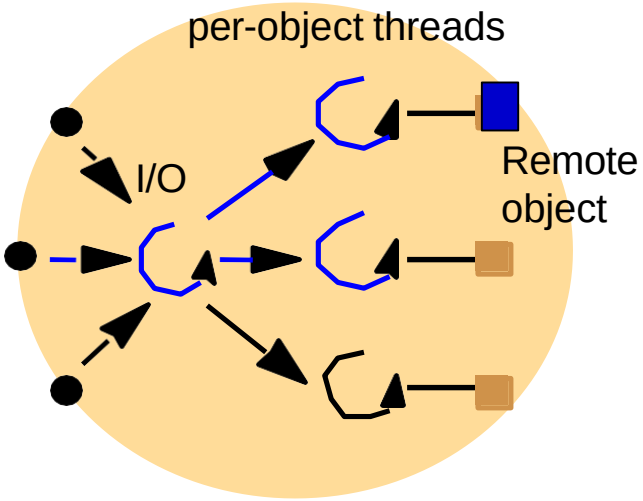
- Spawns
  - A new worker(thread) creates for each request
  - worker destroys itself when finished
- Allows max throughput
  - no queuing
  - no I/O delays(caching)
- But, overhead of creation & destruction of threads is high

Thread-per-connection



- Create a new thread for each connection
- Multiple requests
- Destroy thread on close
- Lower overheads but, unbalanced load

Thread-per-object



- As per-connection, but, a new thread created for each object.
- As thread-per-connection, lower thread management
- Per-object queue

- At thread-per-connection and thread-per-object, each server has lower thread management overhead compared with thread-per-request , but client may be delayed due to higher priority requests

Why threads, not multi-processes?

- Process context switching
  - requires save/restore of execution environment

- Threads within a process V.S. multi-processes

  (why Multi-threads?)
  - Creating a thread is (much) cheaper than a process (~10-20 times).
  - Switching to a different thread in same process is (much) cheaper (5-50 times).
  - Threads within same process can share data and other resources more conveniently and efficiently (without copying or messages).
  - Threads within a process are not protected from each other.

Storing execution environment

Execution environment(=process)	***Thread***
Address space tables Communication interfaces, open files	Saved processor registers Priority and execution state (such as *BLOCKED*)
Semaphores, other synchronization objects List of thread identifiers	Software interrupt handling information Execution environment identifier

Thread scheduling

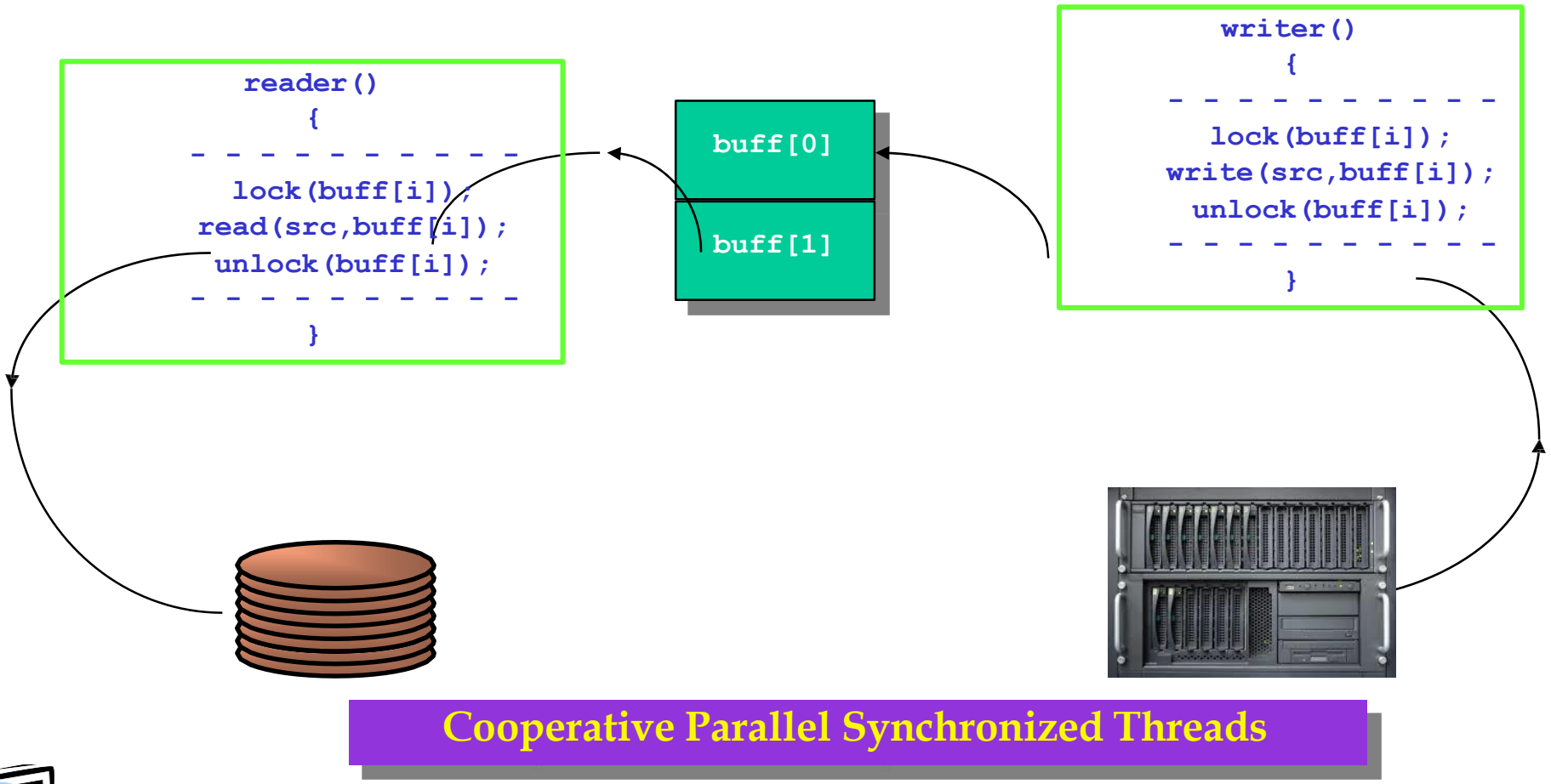
Non-preemptive scheduling

- A thread runs until it makes a call to the threading system.
- Easy to synchronize.
- Be careful to write long-running sections of code that do not contain calls to the threading system.
- Unsuited to real-time applications.

Preemptive scheduling

- A thread may be suspended at any point to make way for another thread,

Multithreaded/Parallel File Copy



Cooperative Parallel Synchronized Threads

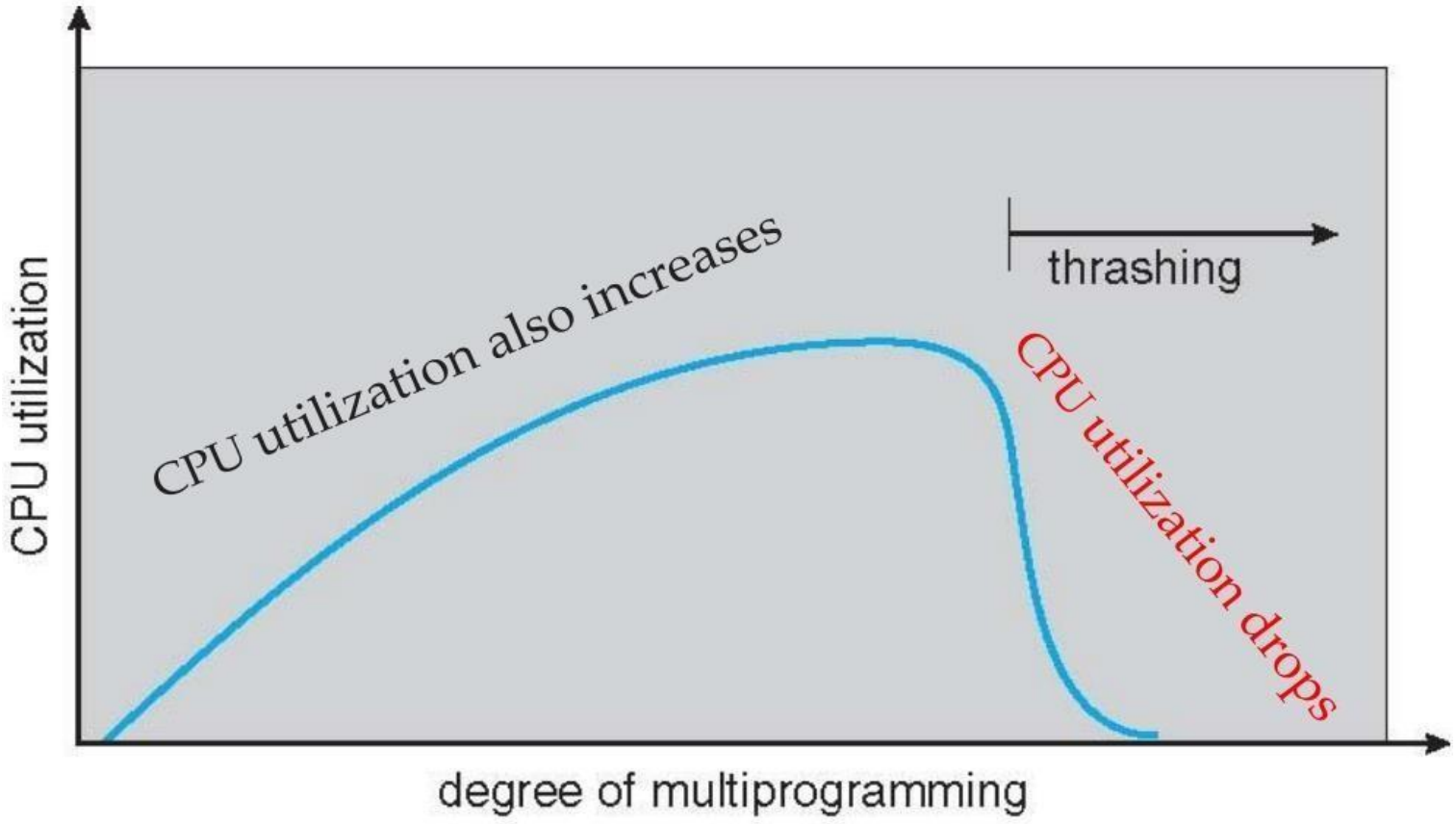
Limit the number of threads in a server process

- Threads require memory for setting up their own private stack. Consequently, having many threads may consume too much memory for the server to work properly.

- In a virtual memory system it may be difficult to build a relatively stable working set, resulting in many page faults and thus I/O. Having many threads may lead to a performance degradation resulting from page thrashing

- Even in those cases where everything fits into memory, we may easily see that memory is accessed following a chaotic pattern rendering caches useless.
  Again, performance may degrade in comparison to the single-threaded case.

Thread and Thrashing



An aside: Java threads

- Class *Thread* (http://java.sun.com/j2se/1.5/docs/api/index.html)
  - constructor/destructor, SUSPENDED/RUNNABLE
  - priorities (useful for *servlets*, dynamic web pages)
- Synchronization
  - monitors (keyword *synchronized*)
  - at most one thread within monitor
- Scheduling
  - Preemptive (suspended at any time), non-preemptive
  - no real-time thread scheduling
- Threads implementation
  - Multi-threaded process

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: synchronization

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()

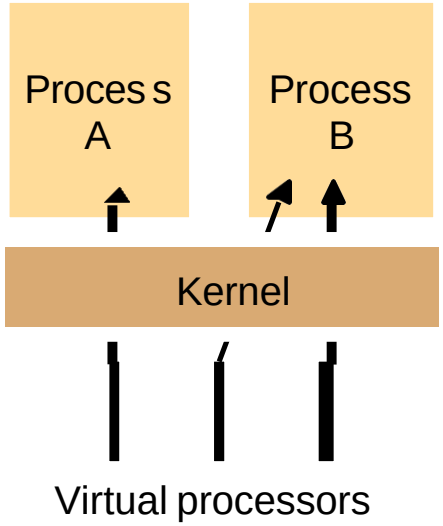
Wakes, respectively, one or all of any threads that have called *wait()* on *object*.

Threads Implementation

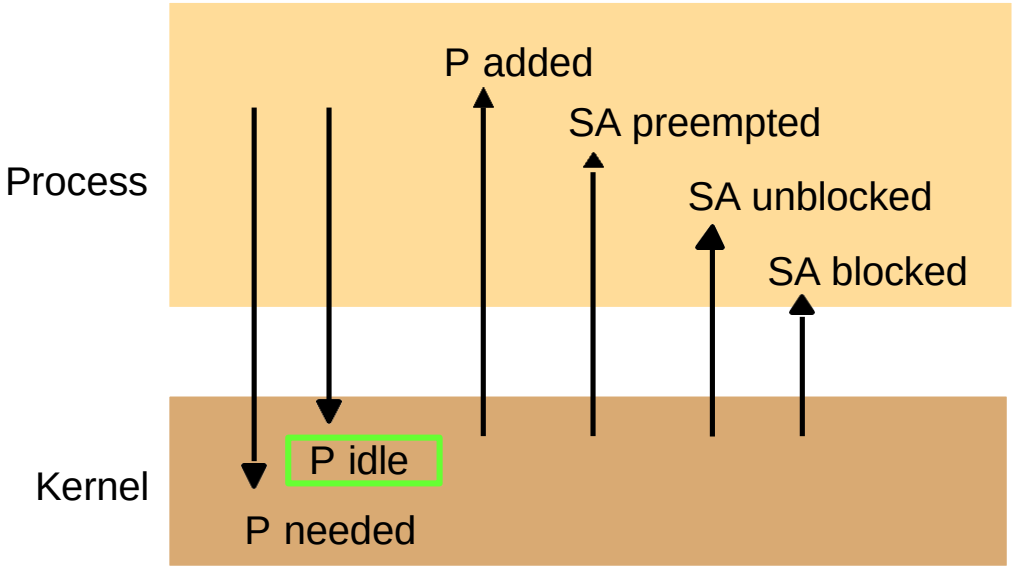
- Kernels
  - supporting multi-threads
  - providing thread creation and management system calls
  - ex) Window NT, Mach, Chorus
- when no kernel support for multi-threading is provided, user-level threads implementation suffers as following;
  - threads within a process can not take advantage of multiprocessor
  - a thread that takes a page fault blocks the entire process and all threads within it
  - threads within different processes can not be scheduled as a single scheme
- User-level threads implementation's advantages
  - thread operations are less costly
  - thread scheduling module is implemented outside the kernel
  - many more use-level threads can be supported than kernel level

Threads Implementation - ctd

- Schedule activation(SA)



A. Assignment of virtual processors to processes

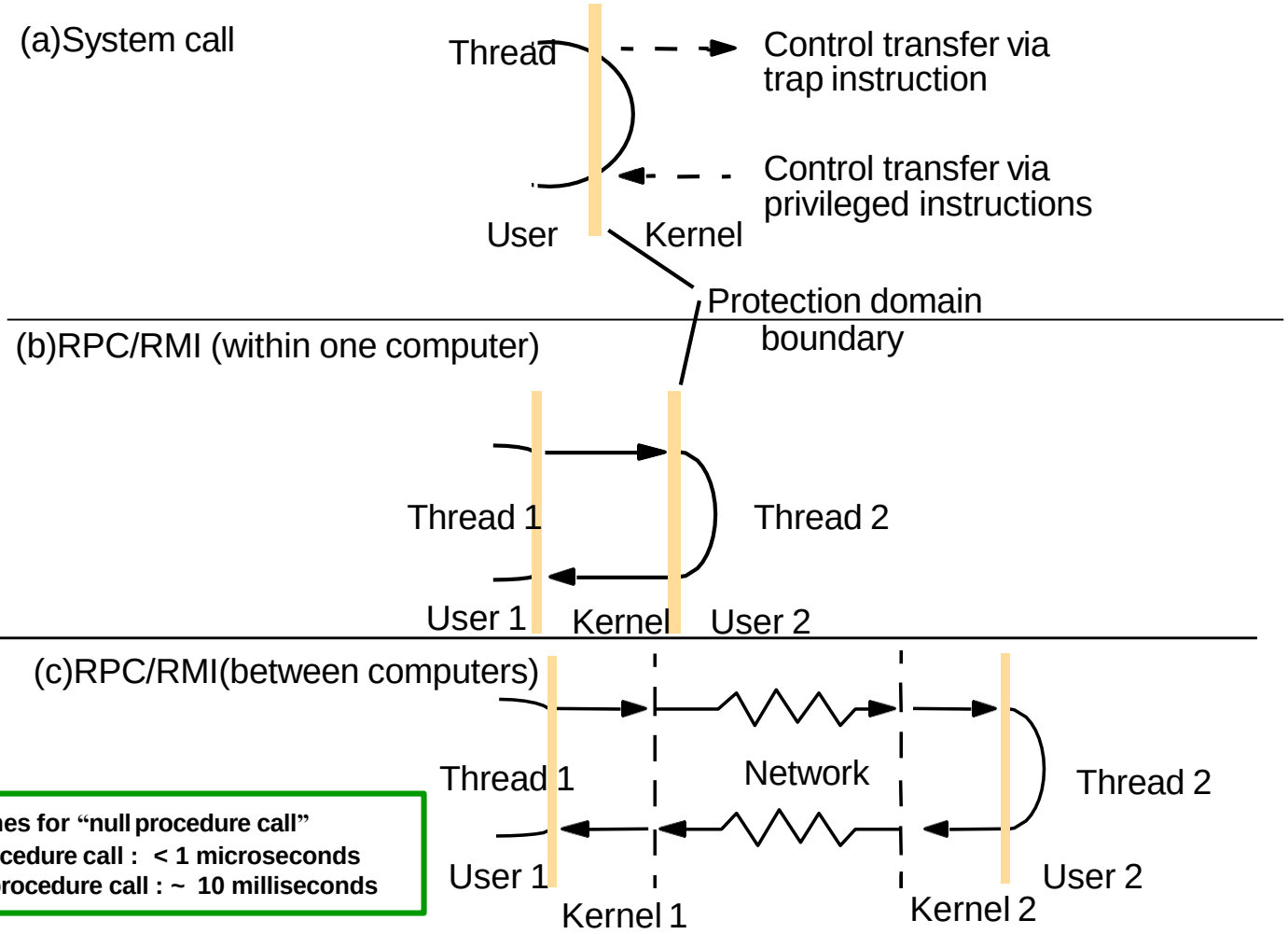


B. Events between user-level scheduler & kernel
Key: P = processor; SA = scheduler activation

Implementation of Thread 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 of a thread
  - does it cross address space or not?
  - synchronous or asynchronous?
  - over the network or within a computer?

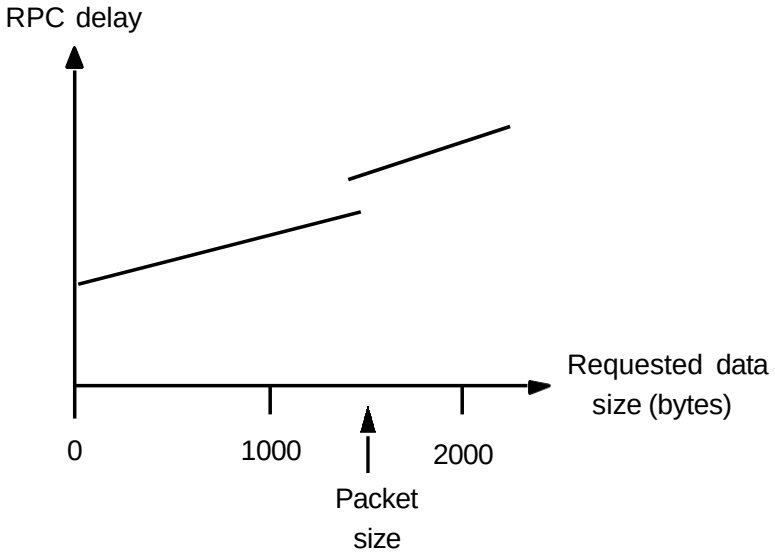
Implementation of Thread invocation



- **Typical times for "null procedure call"**
  - **Local procedure call : < 1 microseconds**
  - **Remote procedure call : ~ 10 milliseconds**

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!



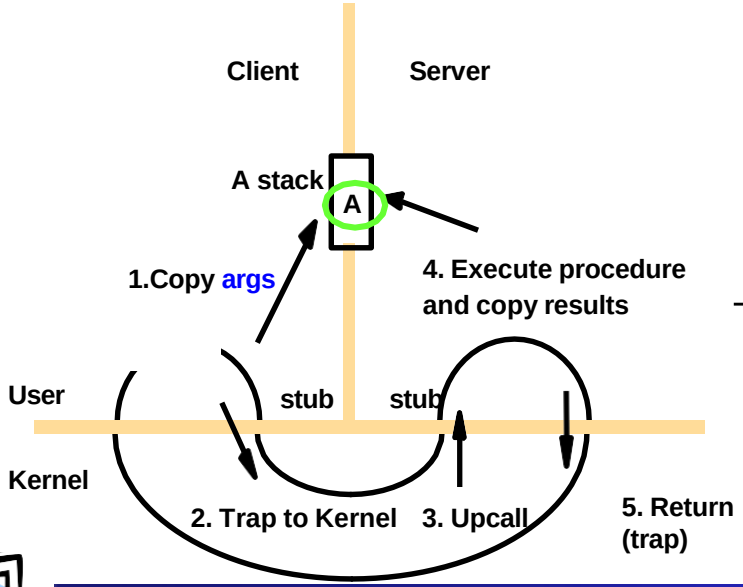
RPC delay against parameter size(ex) 1500bytes)

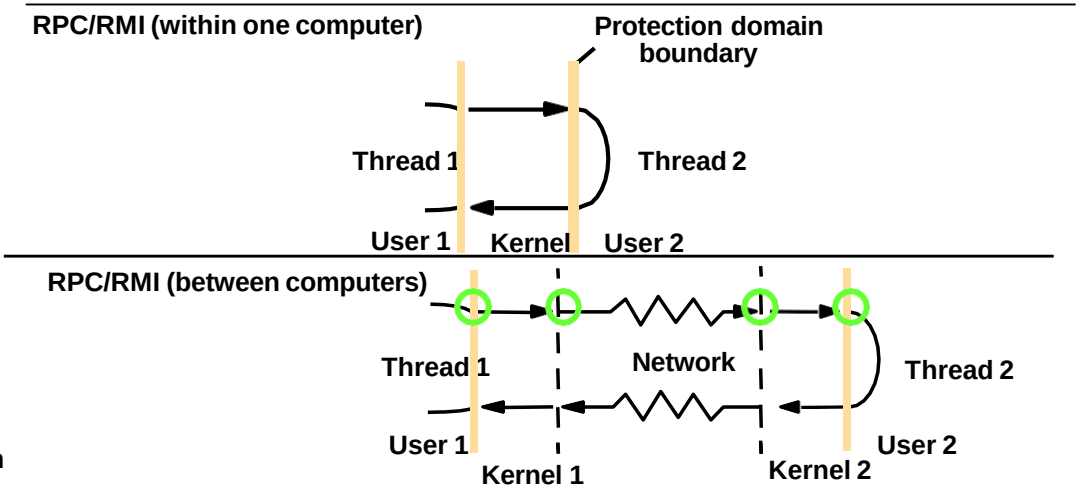
Factors affecting RPC/RMI delays

- Marshalling
- Data copying **(RPC vs. Lightweight RPC)**
  - user to kernel, across network, etc
- Packet initialization
  - protocol headers, checksums
- Thread scheduling & Context switching
- Waiting for acknowledgement
  - TCP or UDP?

Factors affecting RPC/RMI delays

- **RPC vs. Lightweight RPC within a computer**
  - ✓ **Data Copying, Thread Scheduling=> optimization**
  - ✓ **Data Copying**
    - **RPC : client's stub stack -> kernel buffer -> server's message -> server's stub stack(data copying 4 times)**
    - **Lightweight RPC :use Shared region (stack A) (data copying 1 time)**

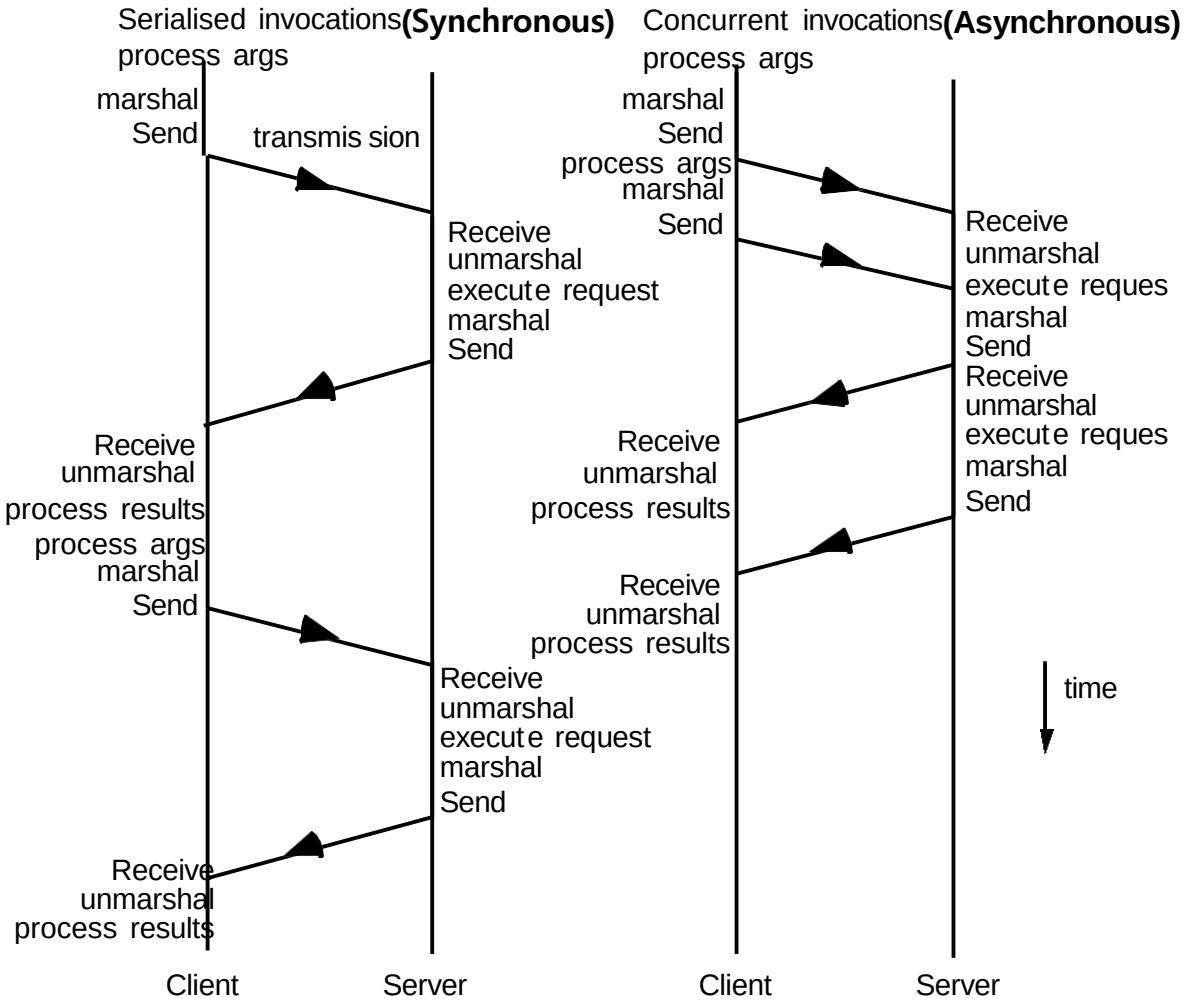




Concurrent invocations

- 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

Serialized 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

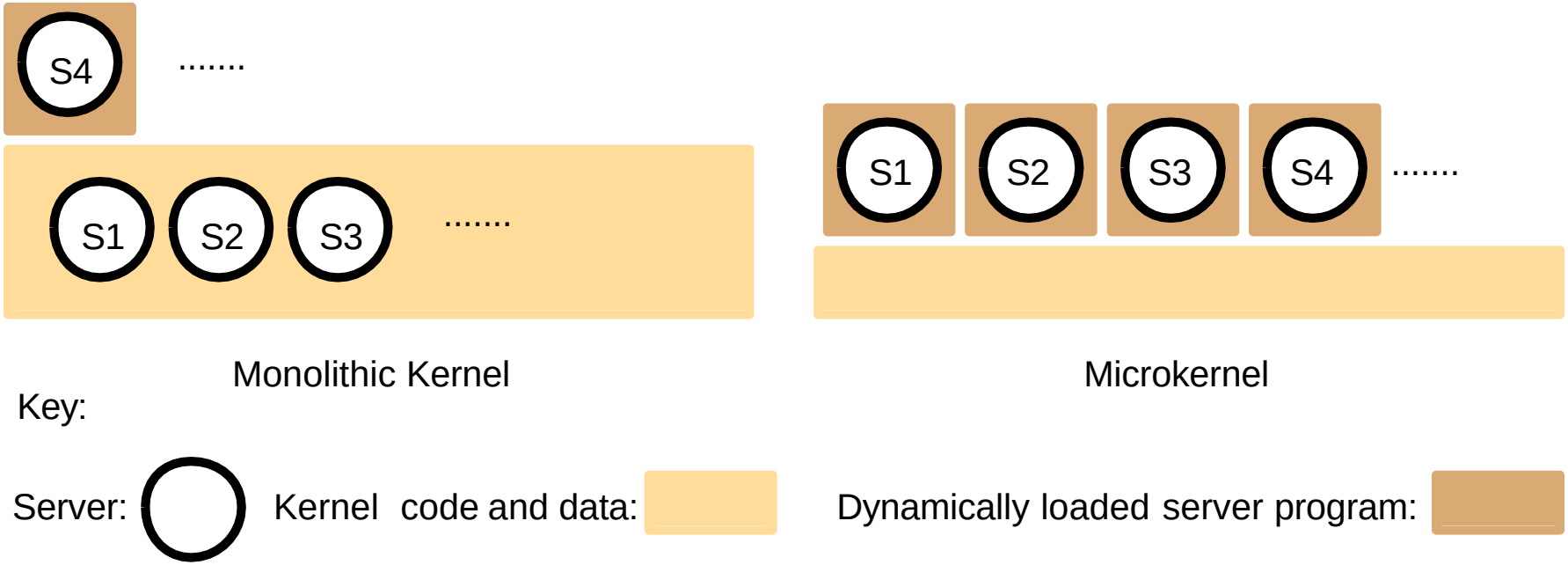
Operating system architecture

- An architecture of kernel suitable for a distributed systems

- Open distributed system
  - Running only that system software carrying out **particular roles at each computer** in the system architecture
  - allows **the software implementing any particular system service** to be changed independently of other facilities
  - introduces **new services without prejudice** to existing ones

Operating system architecture

- Monolithic kernel and Microkernel



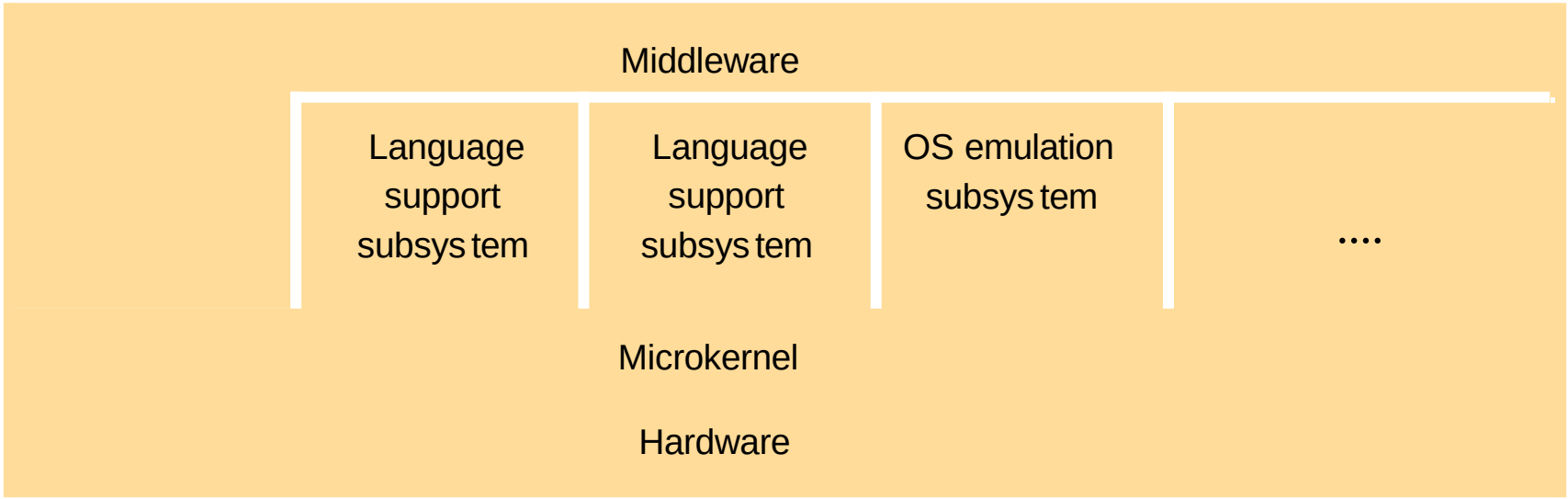
Operating system architecture-ctd

- Monolithic kernel
  - Massive
  - Undifferentiated(
  - Intractable

- Considerations of Microkernel design
  - Kernel ; abstractions(processor, memory, IPC)
  - Server use system call (to use HW)

Operating system architecture-ctd

- The role of Microkernel



The microkernel supports middleware via subsystems

Operating system architecture-ctd

- Microkernel's Pros and Cons
  - flexibility and extensibility
    - services can be added, modified and debugged
    - small kernel -> fewer bugs
    - protection of services and resources is still maintained
  - service invocation expensive
    - unless LRPC is used
    - extra system calls by services for access to protected resources

- Monolithic kernel's advantages
    - efficient without switching between address spaces, when operation calls
    - solving defects of monolithic kernel
      - Needs Layering and object-oriented design

Summary

- OS support is 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
- Open distributed system(Operating system architecture)
  - Monolithic kernel→Microkernel