OS Support for Building Distributed Applications: Multithreaded Programming using Java Threads

Ali Fanian
Isfahan University of Technology

http://www.fanian.iut.ac.ir

Outline

- Introduction
- Thread Applications
- Defining Threads
- Java Threads and States
- Architecture of Multithreaded servers
- Threads Synchronization
- Thread Concurrency Models
- Summary

Learning objectives

- Know what a modern operating system does to support distributed applications and middleware
  - Definition of network OS
  - Definition of distributed OS
- Understand the relevant abstractions and techniques, focussing on:
  - processes, threads, ports and support for invocation mechanisms.

Networked OS to Distributed OS

- Distributed OS
  - Presents users (and applications) with an integrated computing platform that hides the individual computers.
  - Has control over all of the nodes (computers) in the network and allocates their resources to tasks without user involvement.
    - *In a distributed OS, the user doesn't know (or care) where his programs are running.*
  - One OS managing resources on multiple machines
  - Examples:
    - *Cluster computer systems*
    - *Amoeba, V system, Sprite, Globe OS*

Networked OS to Distributed OS

- Network Operation system
  - system retain autonomy in managing their own processing resources
  - there are multiple system images, one per node
  - a user can remotely log into another computer *and run processes there*

Middleware and the Operating System

- In fact, there are no distributed operating systems in general use, only network operating systems such as UNIX, Mac OS and Windows

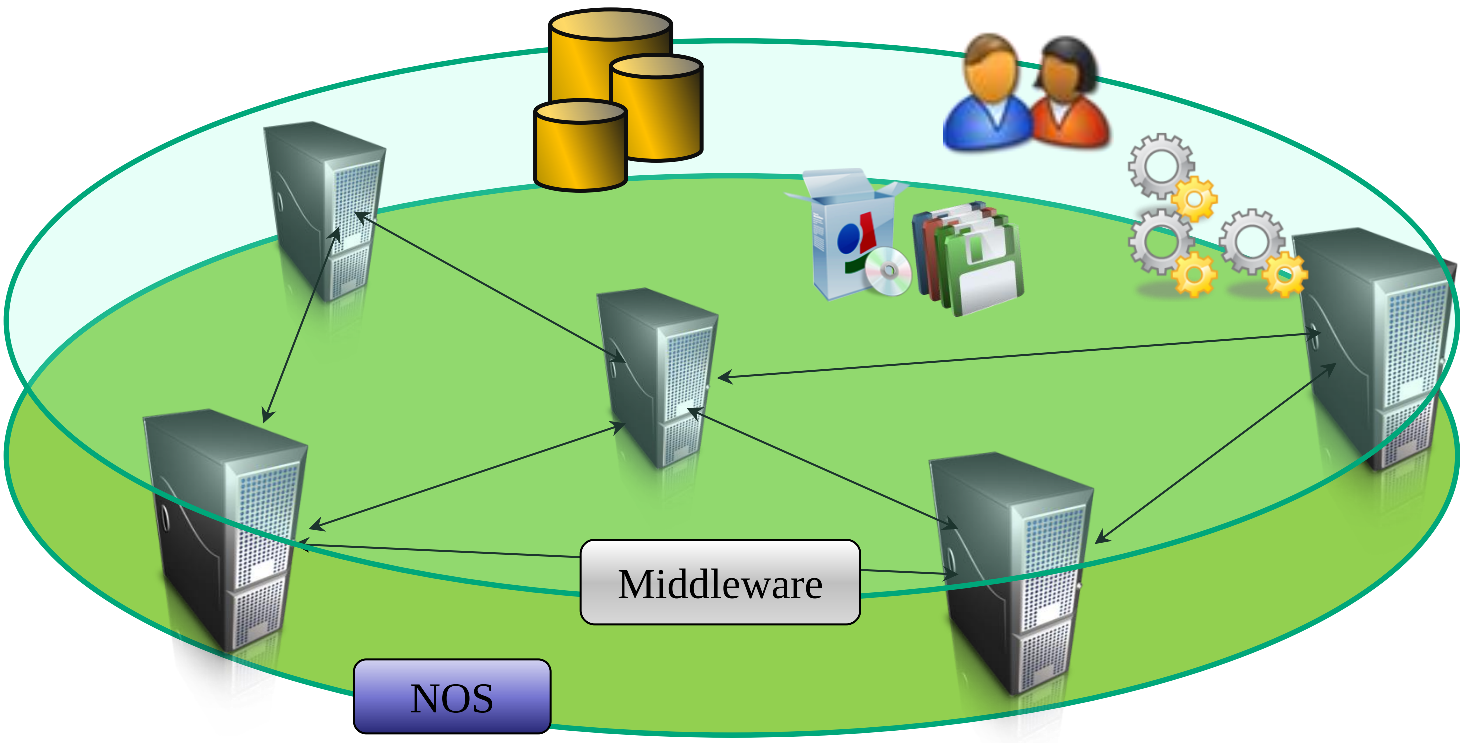
- The combination of middleware and network operating systems provides an acceptable balance between the requirement for autonomy on the one hand and network transparent resource access on the other.

The support required by middleware and distributed applications

- OS manages the basic resources of computer systems
- Tasks:
  - programming interface for these resources:
    - abstractions such as: processes, virtual memory, files, communication channels
    - Protection of the resources used by applications
    - Concurrent processing
  - provide the resources needed for (distributed) services and applications:
    - Communication - network access
    - Processing - processors scheduled at the relevant computers

Middleware

- Building Distributed Systems
  - DOS or NOS are not enough to build a DS!
  - NOS are a good starting point but ….
  - … we need an additional layer "gluing" all together



Building Distributed Systems

- Middleware
  - High-level features for DS
    - Communication
    - Management
    - Application specific
  - Uniform layer where to build DS services
  - Runtime environment of applications
- Operating System
  - Low / medium level (core) features
    - Process / threads management
    - Local hardware (CPU, disk, memory)
    - Security (users, groups, domain, ACLs)
    - Basic networking

System layers

Figure 6.1

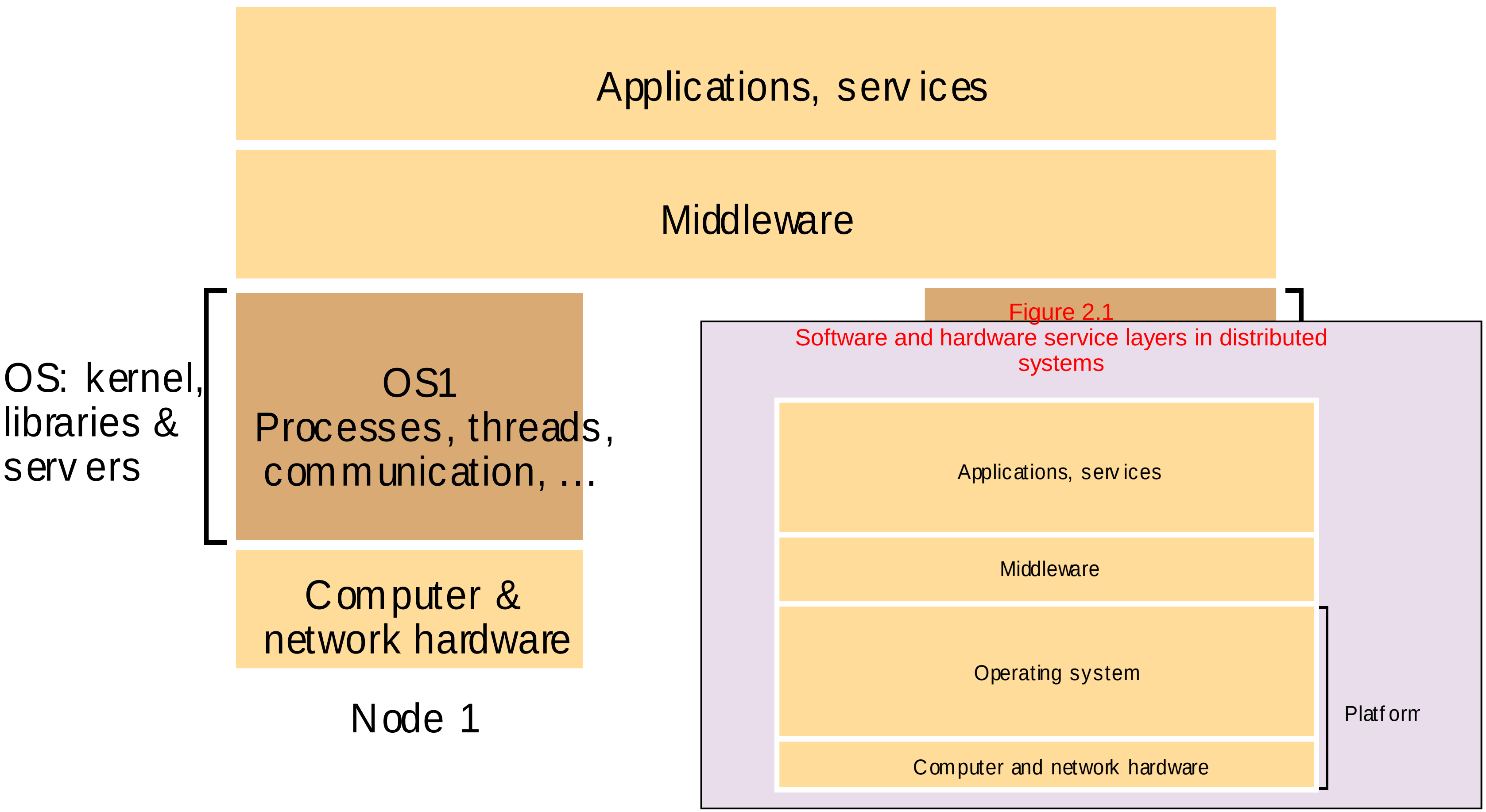


Figure 2.1
Software and hardware service layers in distributed systems

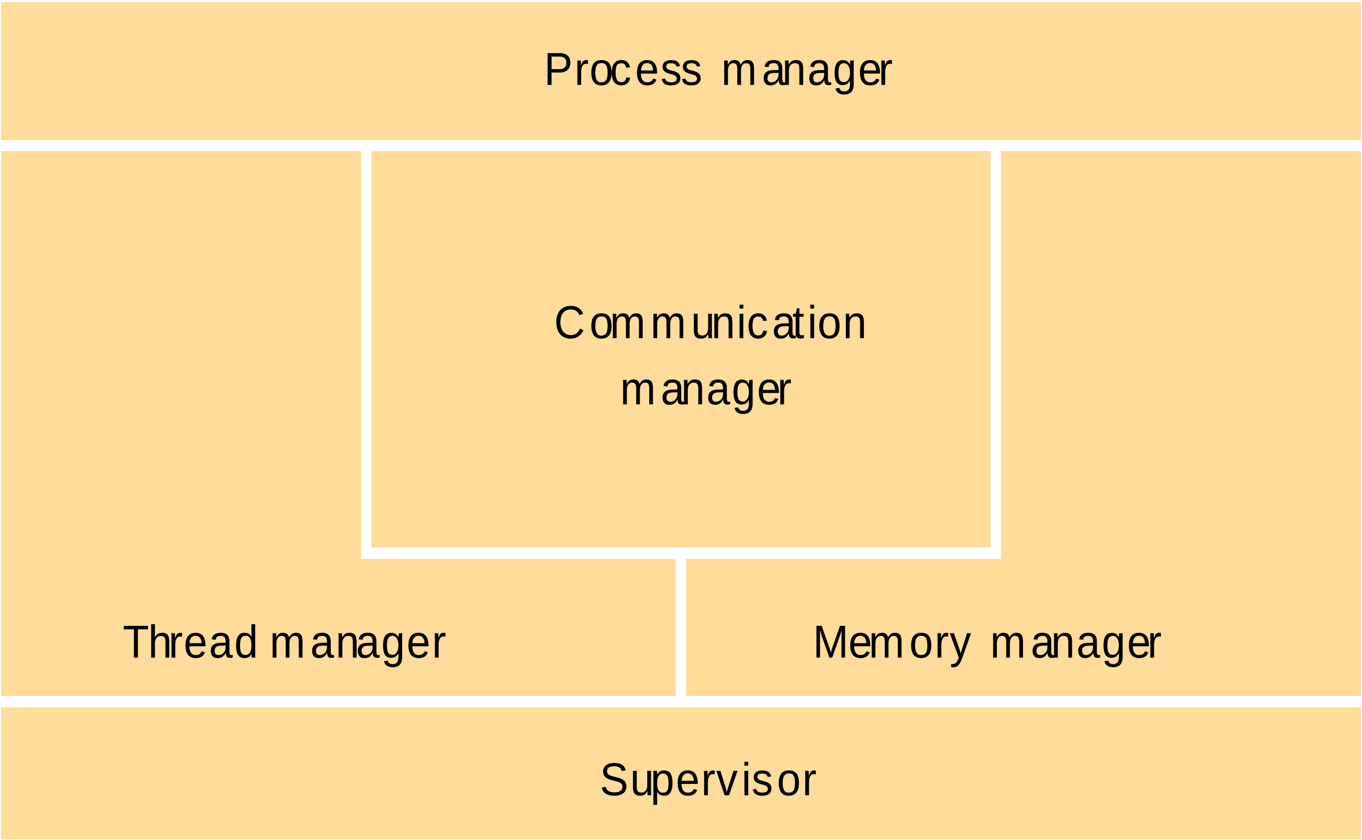
*

The Operating System Layer

- The OS facilitates:
  - Encapsulation : Details such as management of memory and devices used to implement resources should be hidden from clients
  - Protection : Resources require protection from illegitimate accesses
  - Concurrent processing : Clients may share resources and access them concurrently
- Invocation mechanism is a means of accessing an encapsulated resource.

Core OS functionality

Figure 6.2



*

The Operating System Layer

- Figure 2 shows the core OS components:
  - ➢ Process manager
    - Creation of and operations upon processes. A process is a unit of resource management, including an address space and one or more threads
  - ➢ Thread manager
    - Thread creation, synchronization and scheduling
  - ➢ Memory manager
    - Management of physical and virtual memory
  - ➢ Communication manager
    - Communication between threads
  - ➢ Supervisor
    - Dispatching of interrupts, system call traps and other exceptions
    - Control of memory management unit and hardware caches
    - processor and floating-point unit register manipulations.

Protection:

- Resources require protection from illegitimate accesses
  - threats to a system's integrity
    - maliciously contrived code
    - Benign code that contains a bug
- Why does the kernel need to be protected?
- Kernels and protection
  - kernel has all privileges for the physical resources, processor, memory..

Protection:

- Most processors have a hardware mode register whose setting determines whether privileged instructions can be executed
  - Supervisor Mode
  - User Mode
- address space
  - Kernel protect itself and other processes from the accesses of an aberrant process
- user transferred to kernel

  \- system call trap
    - try to invoke kernel resources
    - switch to kernel mode
- cost
  - switching overhead to provide protection

Processes and Threads

- Process
  - A process consists of an execution environment together with one or more threads.
  - An execution environment consists of :
    - An address space
    - Thread synchronization and communication resources (e.g., semaphores, sockets)
    - Higher-level resources (e.g., file systems, windows)

Processes and Threads

- Threads
  - Threads are schedulable activities attached to processes.
  - The aim of having multiple threads of execution is :
    - To maximize degree of concurrent execution between operations
    - To enable the overlap of computation with input and output
    - To enable concurrent processing on multiprocessors.

Processes and Threads

- Threads can be helpful within servers:
  - Concurrent processing of client's requests can reduce the tendency for servers to become bottleneck.
    - E.g. one thread can process a client's request while a second thread serving another request waits for a disk access to complete.

Processes and Threads

- Processes vs. Threads
  - Threads are "lightweight" processes,
  - processes are expensive to create but threads are easier to create and destroy.

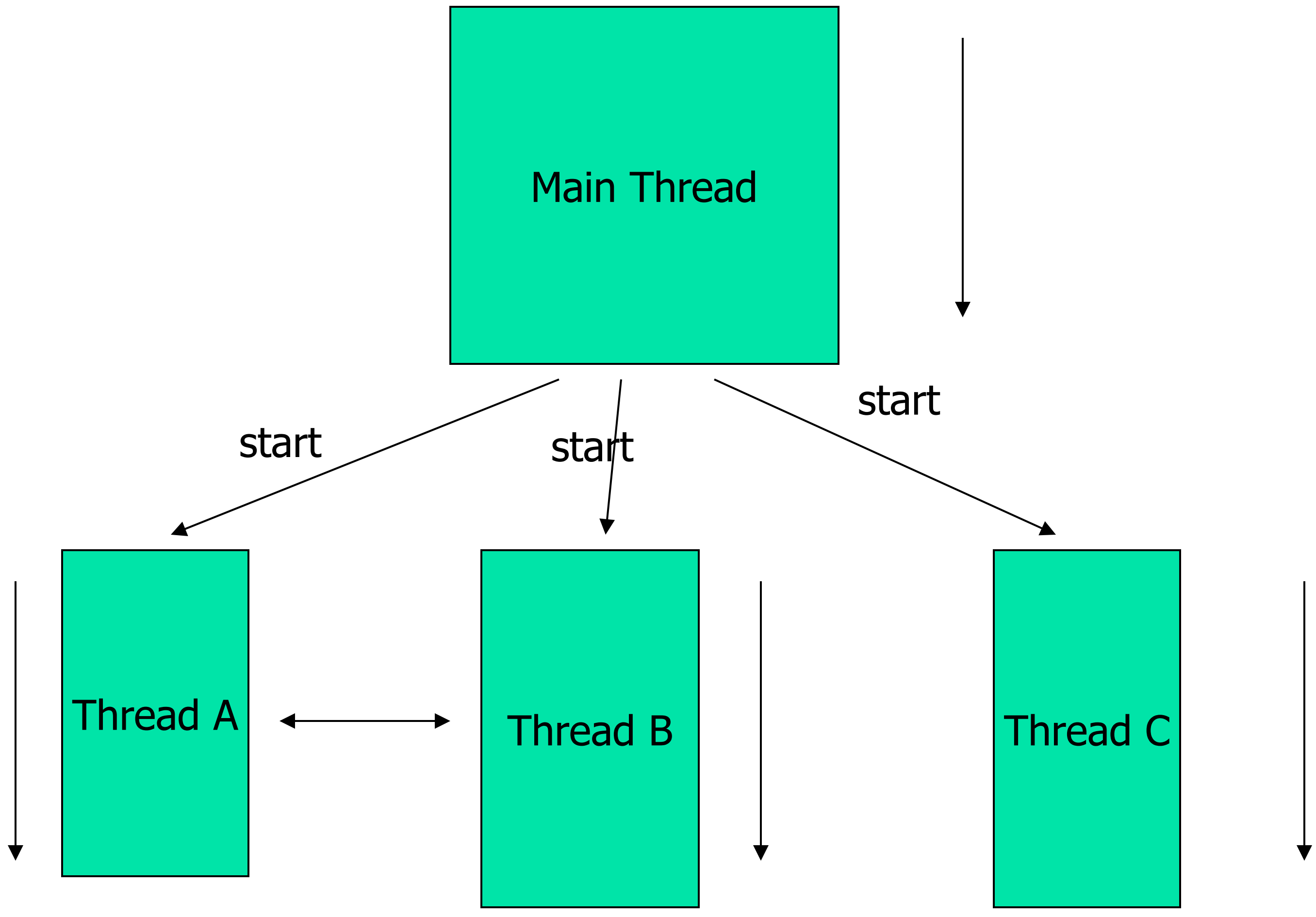
Processes and Threads

- Thread synchronization
  - The main difficult issues in multi-threaded programming are the sharing of objects and the techniques used for thread coordination and cooperation.
  - Each thread's local variables in methods are private to it.
    - Threads have private stack.

Processes and Threads

- Threads can be blocked and woken up
  - The thread awaiting a certain condition calls an object's wait() method.
  - The other thread calls notify() or notifyAll() to awake one or all blocked threads.
- example
  - When a worker thread discovers that there are no requests to process, it calls wait() on the instance of Queue.
  - When the I/O thread adds a request to the queue, it calls the queue's notify() method to wake up the worker.

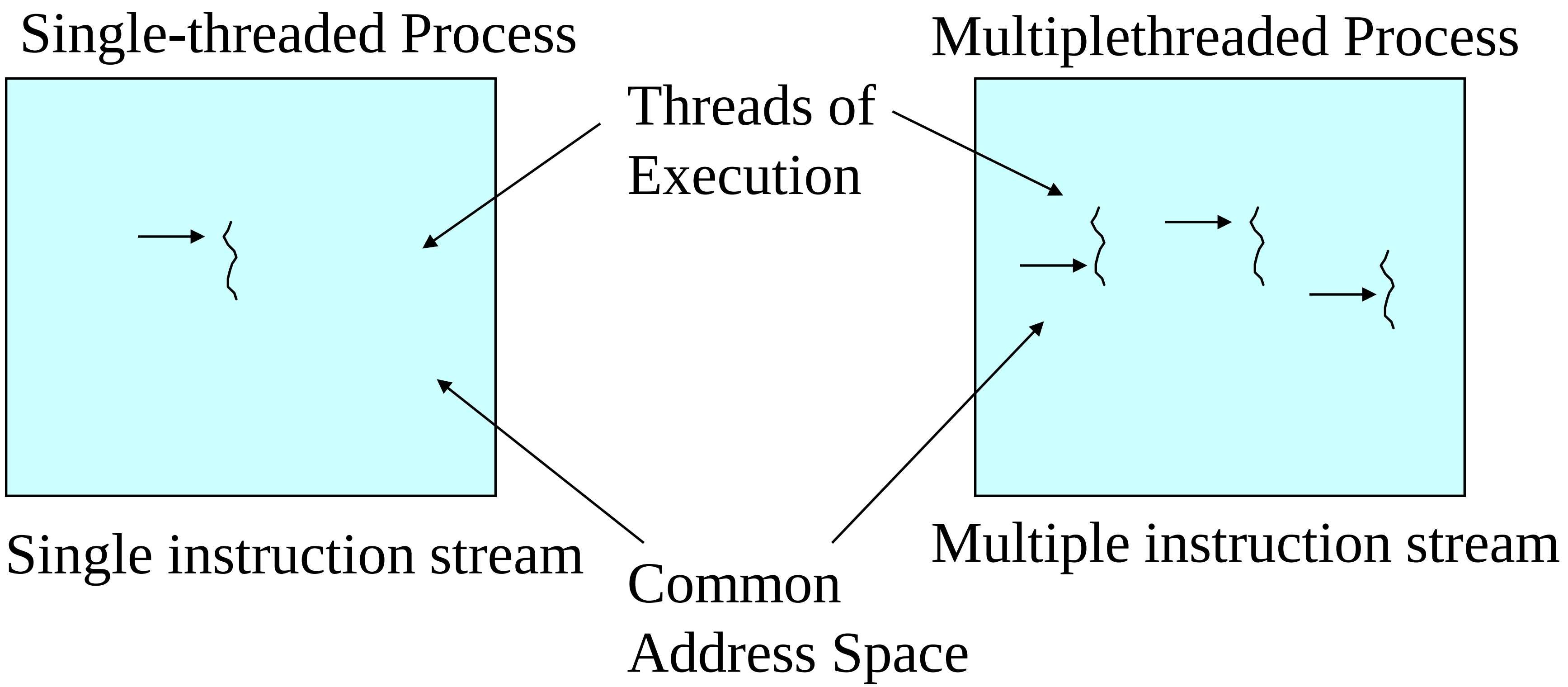
A Multithreaded Program



Threads may switch or exchange data/results

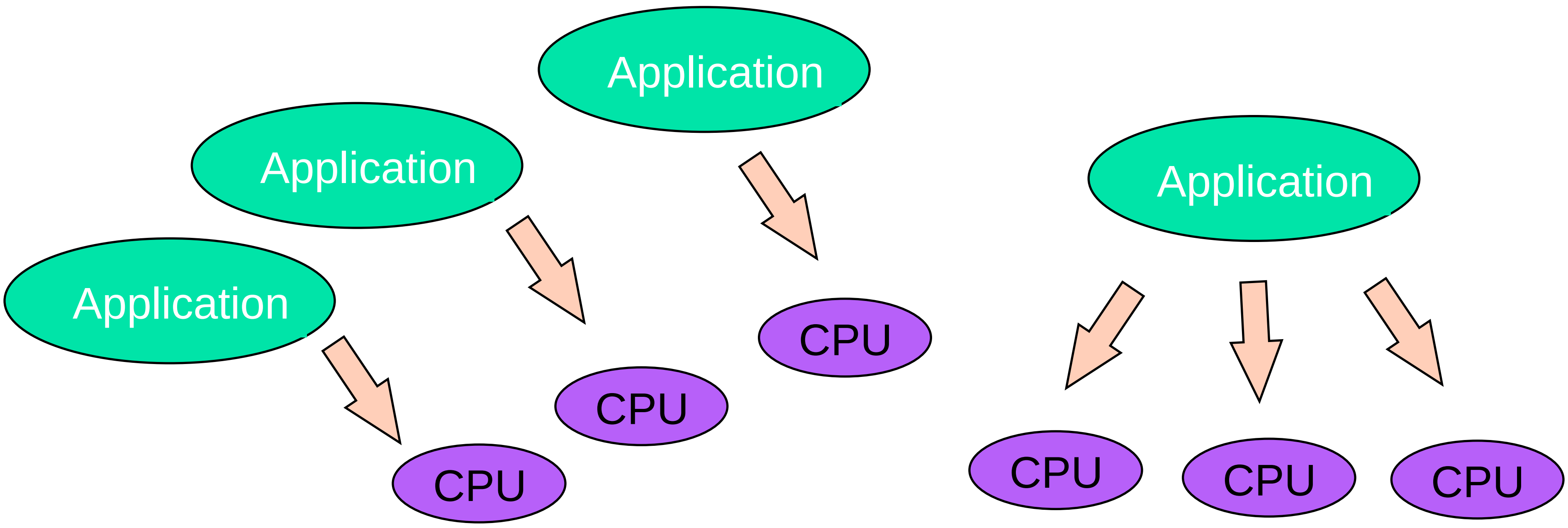
Single and Multithreaded Processes

threads are light-weight processes within a process



Multi-Processing (clusters & grids) and Multi-Threaded Computing

Threaded Libraries, Multi-threaded I/O



Better Response Times in Multiple Application Environments

Higher Throughput for Parallelizeable Applications

Processes and Threads (1)

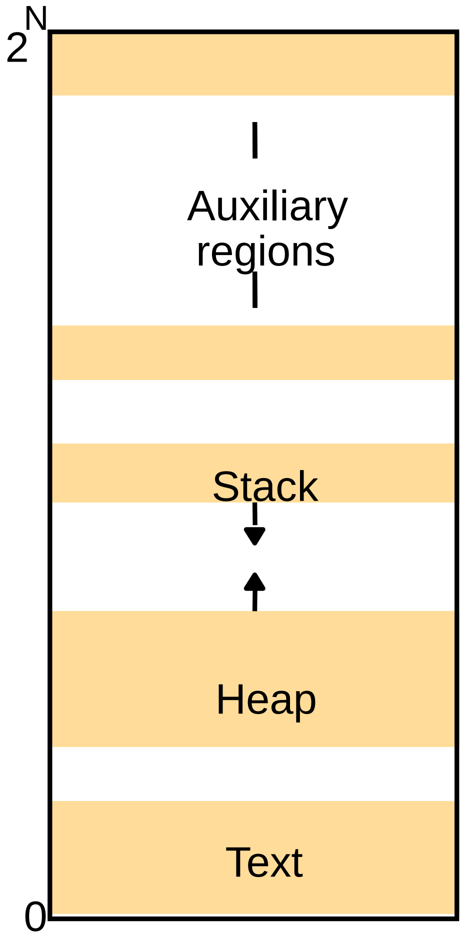
- process has one environment
- thread: activity, "thread" of execution in one environment
- execution environment:
  - an address space
  - synchronization and communication resources
  - i/o resources

Processes and Threads (2)

- Address space
  - unit of management of a process' virtual memory
- Regions
  - Code, heap, stack
- Each region
  - beginning virtual address and size
  - read/write/exe permissions for the process' threads
  - growth direction
- Why regions:
  - different functionalities, for example:
    - different stack regions for threads
    - memory-mapped file
- Shared memory regions among processes?
  - libraries
  - kernel
  - data sharing and communication

Processes and Threads (3): Process address space

Figure 6.3



Creation of a new process in Distributed Systems

- The creation of a new process can be separated into two independent aspects
  - the choice of a target host
    - The choice of the node at which the new process will reside is a matter of policy
  - the creation of an execution environment

The choice of a target

- Eager *et al.* distinguish two policy categories for load sharing
  - *transfer policy*
    - determines whether to situate a new process locally or remotely. This may depend whether the local node is lightly or heavily loaded.
  - *location policy*
    - determines which node should host a new process selected for transfer
    - This decision may depend on the loads of nodes and specialized resources they may possess
    - V system and Sprite both provide commands for users to execute a program at a currently idle workstation
    - In the Amoeba system the *run server chooses a host for each process from a shared pool of processors*

- Process location policies may be static or adaptive
  - Static scheme
    - operate without regard to the current state of the system
    - they are designed according to the system’s expected long-term characteristics
    - They may implement deterministic or probabilistic

- Adaptive scheme
  - apply heuristics to make their allocation decisions, based on unpredictable runtime factors such as a measure of the load on each node
  - Load-sharing systems may be centralized, hierarchical or decentralized

- Load-sharing systems may be centralized, hierarchical or decentralized
  - In the centralized and hierarchical scheme
    - there is one load manager component for centralized and in the second there are several
    - Load managers collect information about the nodes and use it to allocate new processes to nodes
  - In Decentralized manager
    - nodes exchange information with one another directly to make allocation decisions

load-sharing algorithms

- sender-initiated
  - One node that requires a new process to be created is responsible for initiating the transfer decision
  - It typically initiates a transfer when its own load crosses a threshold
- receiver-initiated
  - a node whose load is below a given threshold advertises its existence to other nodes so that relatively loaded nodes can transfer work to it

load-sharing algorithms

- Migratory
  - systems can shift load at any time, not just when a new process is created
- Eager et al. studied three approaches to load sharing
  - He concluded that simplicity is an important property of any load-sharing scheme
  - This is because relatively high overheads

Creation of a new execution environment

- Once the host computer has been selected, a new process requires
  - an execution environment consisting of
    - an address space with initialized contents
    - and perhaps other resources, such as default open files

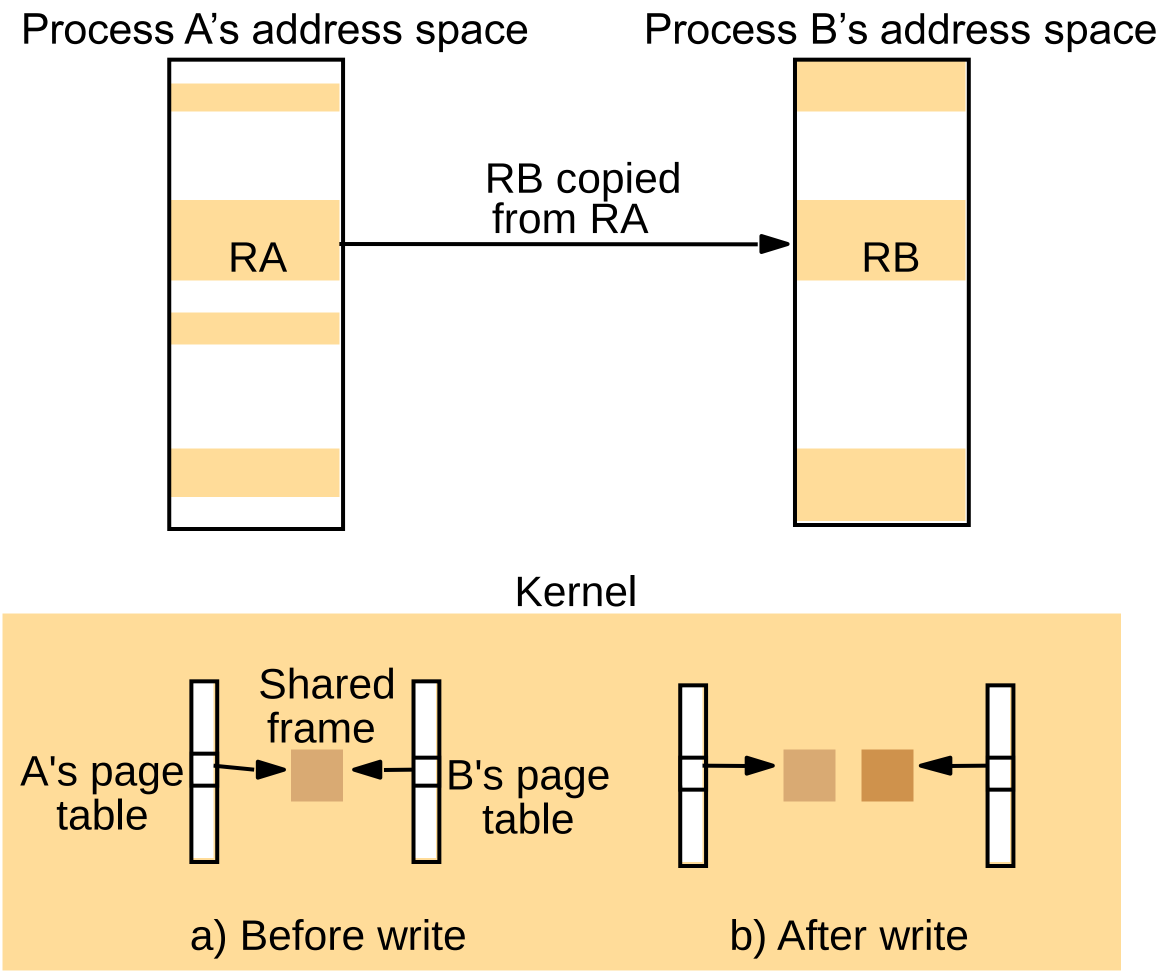
Process Creation

- There are two approaches
  - The first approach is used where the address space is of a statically defined format
    - the address space regions are created from a list specifying their extent
  - Alternatively, the address space can be defined with respect to an existing execution environment (for ex. Fork).
    - the newly created child process physically shares the parent's text region and has heap and stack regions that are copies of the parent's in extent

Some improvement scheme

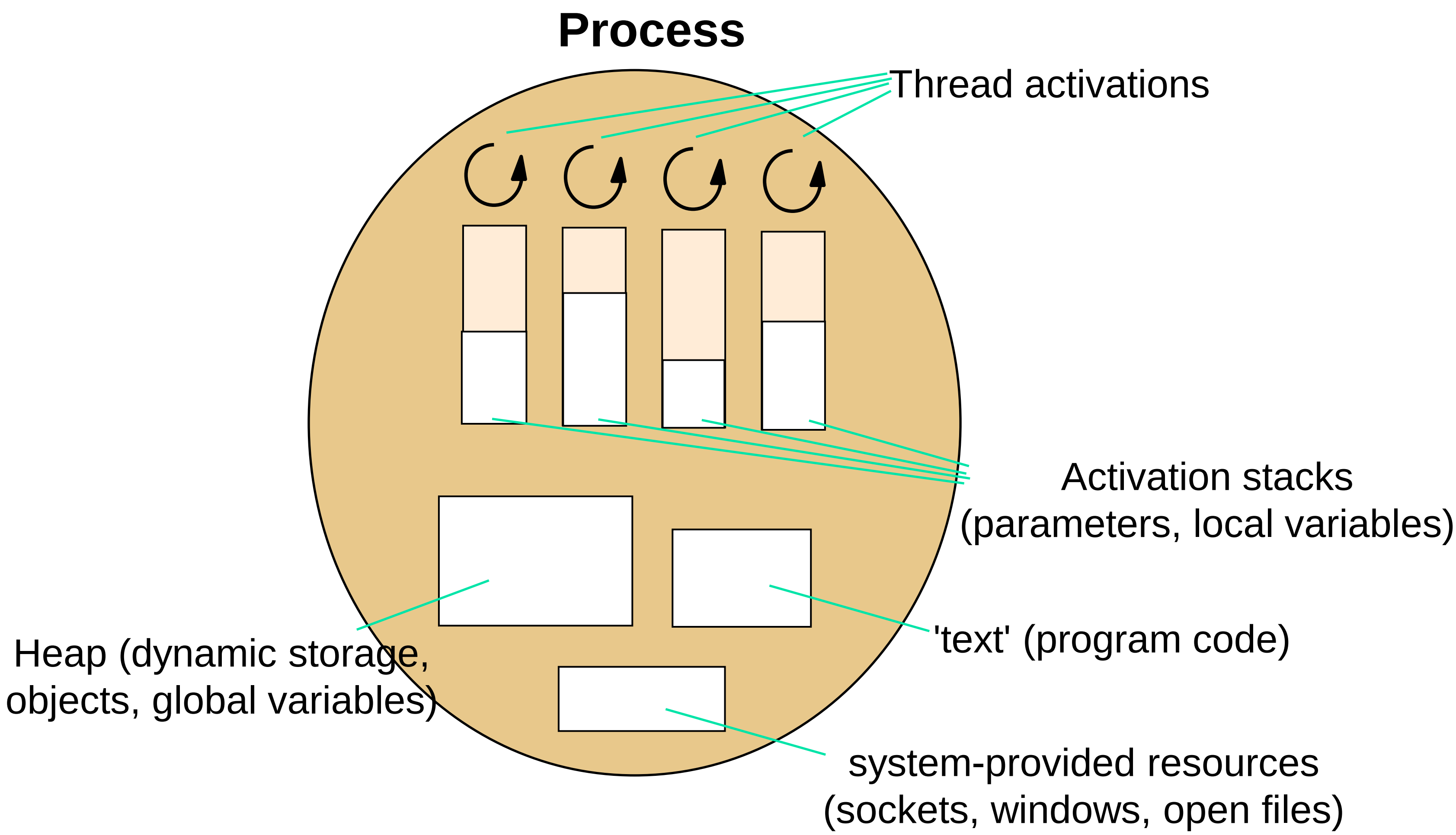
- copy-on-write
  - The region is copied, but no physical copying takes place by default
  - A page in the region is only physically copied when one or another process attempts to modify it

Processes and Threads (6): Copy-on-write



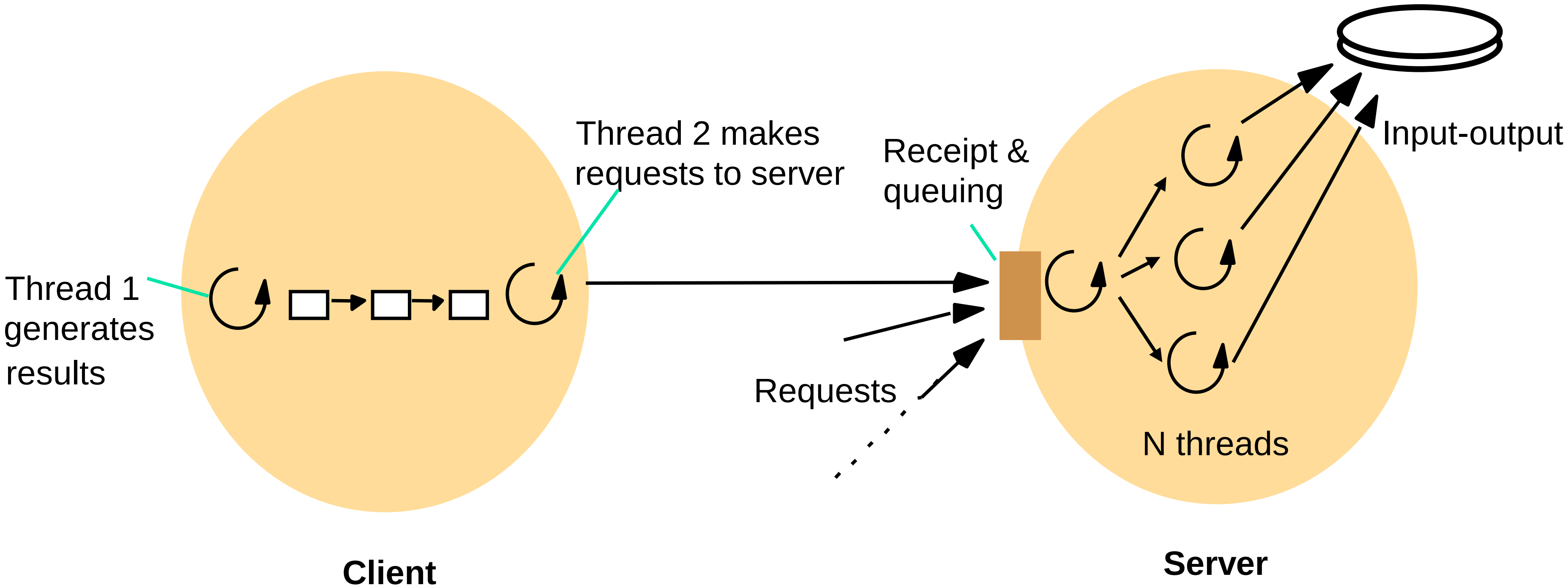
*

Processes and Threads (7): Thread memory regions



*

Processes and Threads (8): Client and server



The 'worker pool' architecture

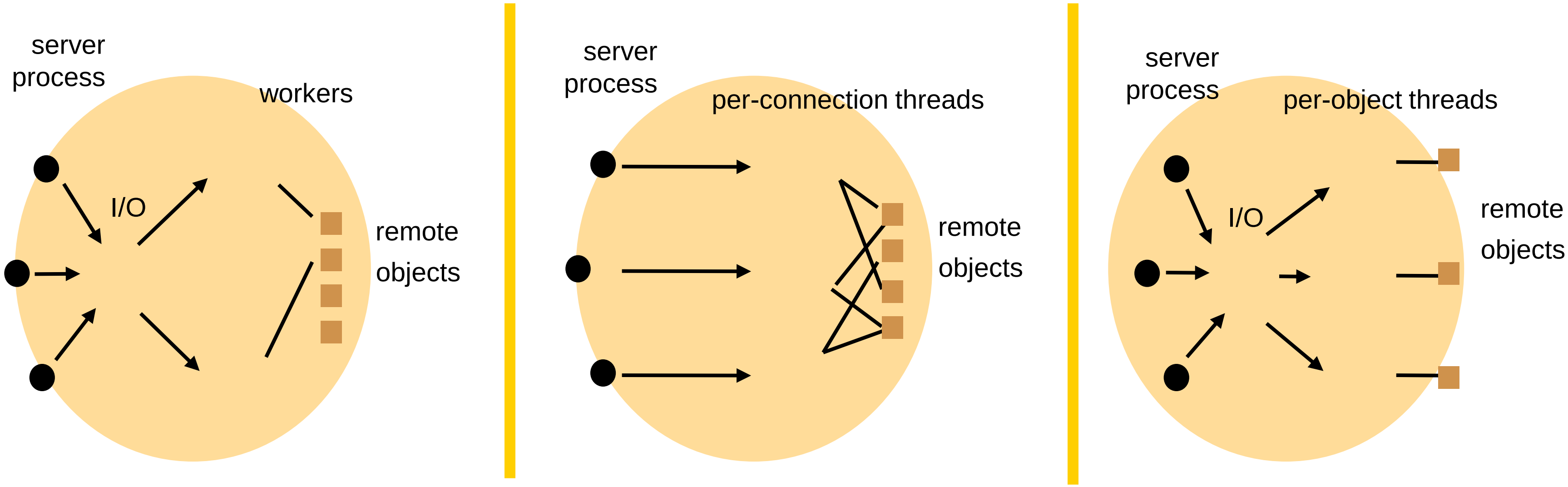
*

Processes and Threads (9)

- average interval of successive job completions
  - one request: 2 milliseconds of processing and 8 for i/o delay
  - one thread: 2+8 = 10 milliseconds, 100 requests/second

  - two threads: 125 requests/second, serial i/o, why?
  - two threads: 200 requests/second, concurrent i/o, why?
  - two threads with cache (75% hit):
    - *2 milliseconds (.75*0 + .25*8), 500 requests/sec*
  - cpu overhead of caching: 2.5 milliseconds, 400 requests/sec

Processes and Threads (10): server threading architectures



a. Thread-per-request b. Thread-per-connection c. Thread-per-object

*

Processes and Threads (11): Threads vs processes

- 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

State associated with execution environments and threads

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

*

Processes and Threads (12): Concurrency

- Issues in concurrency:
  - Race condition
  - Deadlock
- Programming support
  - library (POSIX pthreads)
  - language support (Ada95, Modula-3, Java)

Processes and Threads (13)

- thread (process) execution
  - create/fork
  - exit
  - join/wait
  - yield

Processes and Threads (14)

- Synchronization
  - coordinate current tasks and prevent race conditions on shared objects
  - Critical region: only one thread/process at a time is allowed
  - Why critical regions should be as small as possible?
- Programming support
  - Mutual exclusion
  - Condition Variables
  - Semaphores

Processes and Threads (15): Mutual Exclusion

- Mutual exclusion (mutex)
  - critical region/section
  - before entering critical region, try to lock
  - mutex_lock(l):
    - *if try to lock is successful*
      - lock and continue
    - *else*
      - blocked
  - mutex_unlock(l): release the lock

Processes and Threads (17): Condition Variables

- Condition variable
  - wait for an event (condition) before proceeding
  - Assoicated mutex with the condition
- Waiting for an event
  1. lock associated mutex m
  2. while (predicate is not true)   // "if" could work, but less safe
  3.   cv_wait(c, m)
  4. do work
  5. unlock associated mutex m
- Signaling an event
  1. lock associated mutex m
  2. set predicate to true
  3. cv_signal(c)   // signal condition variable (wake-up one or all)
  4. unlock associated mutex m

Thread scheduling

- Preemptive
  - a thread can be suspended at any point for another thread to run
- Non-preemptive
  - a thread can only be suspended when it de-schedules itself (e.g. blocked by I/O, sync...) [critical region between calls that de-schedule]
  - any section of code that does not contain a call to the threading system is automatically a critical section
  - Race conditions are thus conveniently avoided
  - The programmer may need to insert special *yield() calls*

Threads implementation

- Some kernels provide
  - Thread creation, management and scheduling
- Some other kernels have only a single-threaded process abstraction
  - Multithreaded processes must then be implemented in a library of procedures linked to application programs
  - the kernel has no knowledge of these user-level threads and therefore cannot schedule them independently
  - A threads runtime library organizes the scheduling of threads
  - A thread would block the process, and therefore all threads within it, if it made a blocking system call

Threads implementation

- user-level threads implementation suffers from the following problems
  - The threads within a process cannot take advantage of a multiprocessor.
  - A thread that takes a page fault blocks the entire process and all threads within it.

Threads implementation

- Advantages user-level threads implementation
  - Certain thread operations are significantly less costly
    - Switching between threads belonging to the same process does not necessarily involve a system call
  - scheduling can be customized
  - more user-level threads can be supported

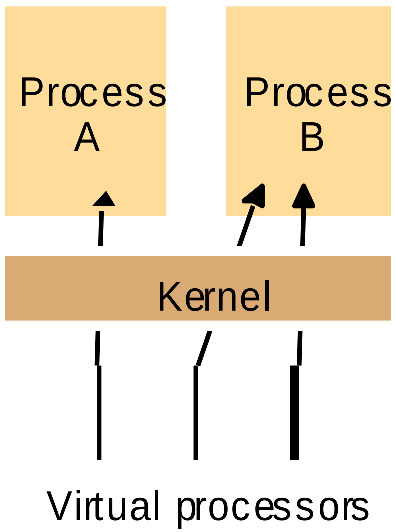
Processes and Threads

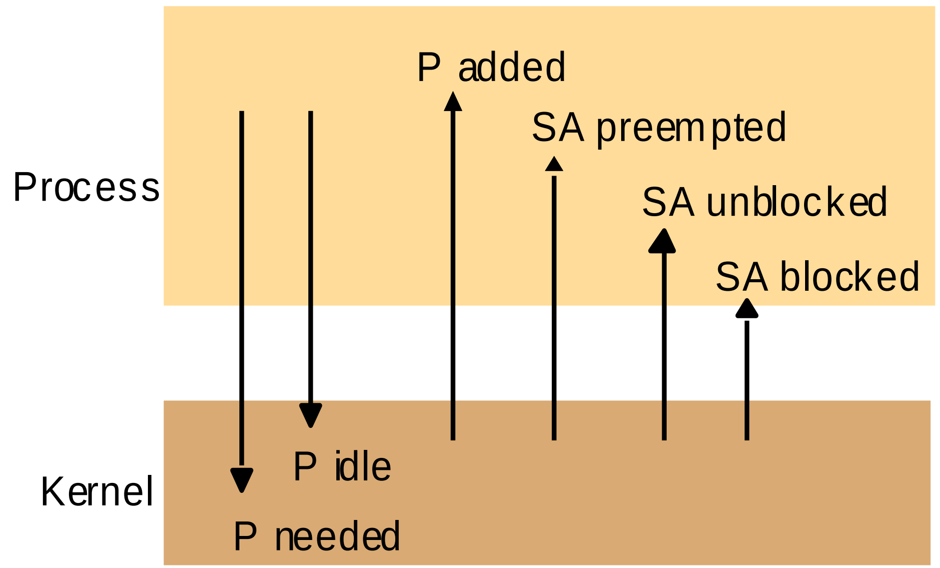
- Mixed
  - Mach:
    - user-level code to provide scheduling hints to the kernel
  - Solaris:
    - assign each user-level thread to a kernel-level thread (multiple user threads can be in one kernel thread)
    - creation/switching at the user level
    - scheduling at the kernel level

Processes and Threads

- FastThread package
  - hierarchical, event-based scheduling
  - each process has a user-level thread scheduler
  - virtual processors are allocated to processes
    - the # of virtual processors depends on a process's needs
    - physical processors are assigned to virtual processors
    - virtual processors can be dynamically allocated and deallocated to a process according to its needs.
  - Scheduler Activation (SA)
    - event/call from kernel to user-level scheduler
    - user-level scheduler can assign threads to SA's

Processes and Threads : Scheduler activations





A. Assignment of virtual processors to processes

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

Skip Sections 6.5 and 6.6

Operating System Architecture

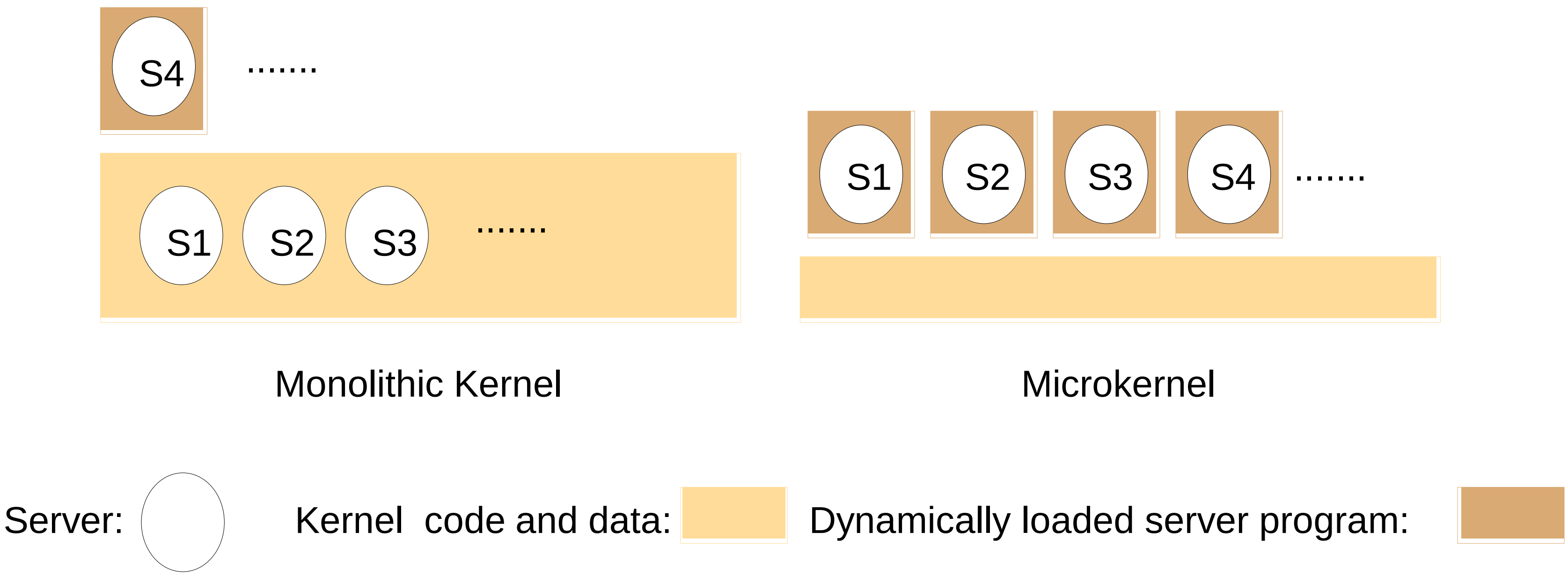
- The kernel would provide only the most basic mechanisms upon which the general resource management tasks at a node are carried out.
- Server modules would be dynamically loaded as required, to implement the required resourced management policies for the currently running applications.

- The major kernel architectures:
  - Monolithic kernels
  - Micro-kernels

Operating System Architecture

- Monolithic Kernels
    - A monolithic kernel can contain some server processes that execute within its address space, including file servers and some networking.
    - The code that these processes execute is part or the standard kernel configuration.

Operating System Architecture

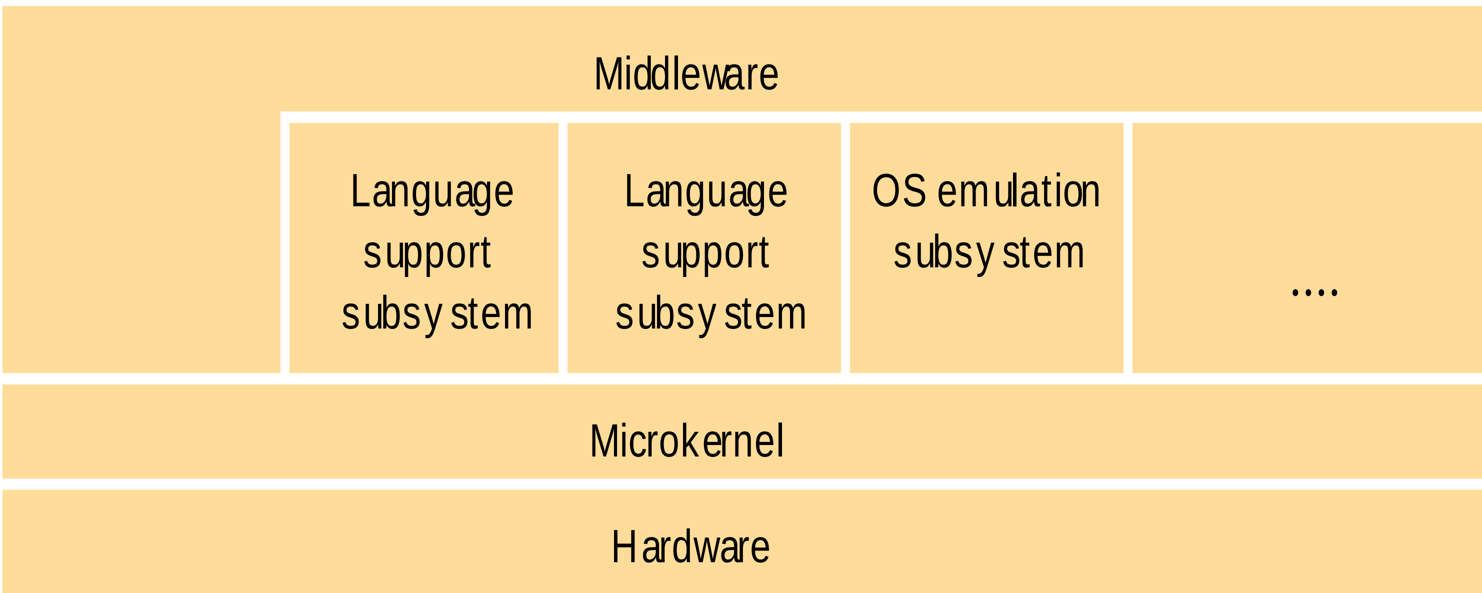


Monolithic kernel and microkernel

Operating System Architecture

- Microkernel
  - The microkernel appears as a layer between hardware layer and a layer consisting of major systems.
  - If performance is the goal, rather than portability, then middleware may use the facilities of the microkernel directly.

Operating System Architecture



The microkernel supports middleware via subsystems

Figure 6. The role of the microkernel

Operating System Architecture

- Monolithic and Microkernel comparison
  - The advantages of a microkernel
    - Its extensibility
    - Its ability to enforce modularity behind memory protection boundaries.
    - Its small kernel has less complexity.
  - The advantages of a monolithic
    - The relative efficiency with which operations can be invoked because even invocation to a separate user-level address space on the same node is more costly.

Operating System Architecture

- Hybrid Approaches
  - Pure microkernel operating system such as Chorus & Mach have changed over a time to allow servers to be loaded dynamically into the kernel address space or into a user-level address space.
  - In some operating system such as SPIN, the kernel and all dynamically loaded modules grafted onto the kernel execute within a single address space.