Chapter 7: Operating System Support

From **Coulouris, Dollimore, Kindberg and Blair**
Distributed Systems: Concepts and Design

Edition 5, © Addison-Wesley 2012

Outlines

- Introduction
- The operating system layer
- Protection
- Processes and threads
- Communication and invocation

Introduction

- An important aspect of **distributed systems** is **resource sharing**.
- **Client applications** invoke operations on resources that are often on another node or at least in another process.
- **Applications** (in the form of clients) and **services** (in the form of resource managers) use the middleware layer for their interactions.
- **Middleware** enables remote communication between objects or processes at the nodes of a distributed system.
- Below the middleware layer is the **operating system (OS) layer.**

When we say resources, this means computer resources such as hard disk, memory, operating system, etc.

Introduction

- The task of any **operating system** is to provide problem-oriented abstractions of the underlying physical resources – the processors, memory, networks, and storage media.
- An **operating system** such as UNIX (and its variants, such as Linux and Mac OS X) or Windows (and its variants, such as XP, Vista and Windows 7) provides the programmer with, for example, files rather than disk blocks, and with sockets rather than raw network access.
- The **OS** takes over the physical resources on a single node and manages them to present these resource abstractions through the system-call interface.
- The **OS** facilitates the encapsulation and protection of resources inside servers and supports the mechanisms required for accessing these resources.

Introduction

Two operating system concepts of distributed systems:

Network Operating System (NOS) have networking ability to access remote interfaces. Multiple system images, one on each node, making autonomy (self independent) in managing their own resources.

- Both **UNIX** and **Windows** are examples of network operating systems. They have a networking capability built into them and so can be used to access remote resources.
- Access is **network-transparent** for some – not all – types of resource. For example, through a **distributed file system** such as **Network File System (NFS),** users have network-transparent access to files. That is, many of the files that user's access are stored remotely, on a server, and this is largely transparent to their applications.

Distributed Operating System (DOS) has a single system image that control over all nodes in DS.

Distributed Operating System

- When we say, "**Distributed Operating System**", it does not mean the operating system itself. However, it means it is partitioned to several parts and these parts are distributed on several machines. That is, it means two things:

1) In case of homogenous machines: The image of the operating system exist in several homogenous machines which facilitates distributed processing among them.
2) In case of one computer, the operating system facilitates distributed processing among processes or processors on the machine.

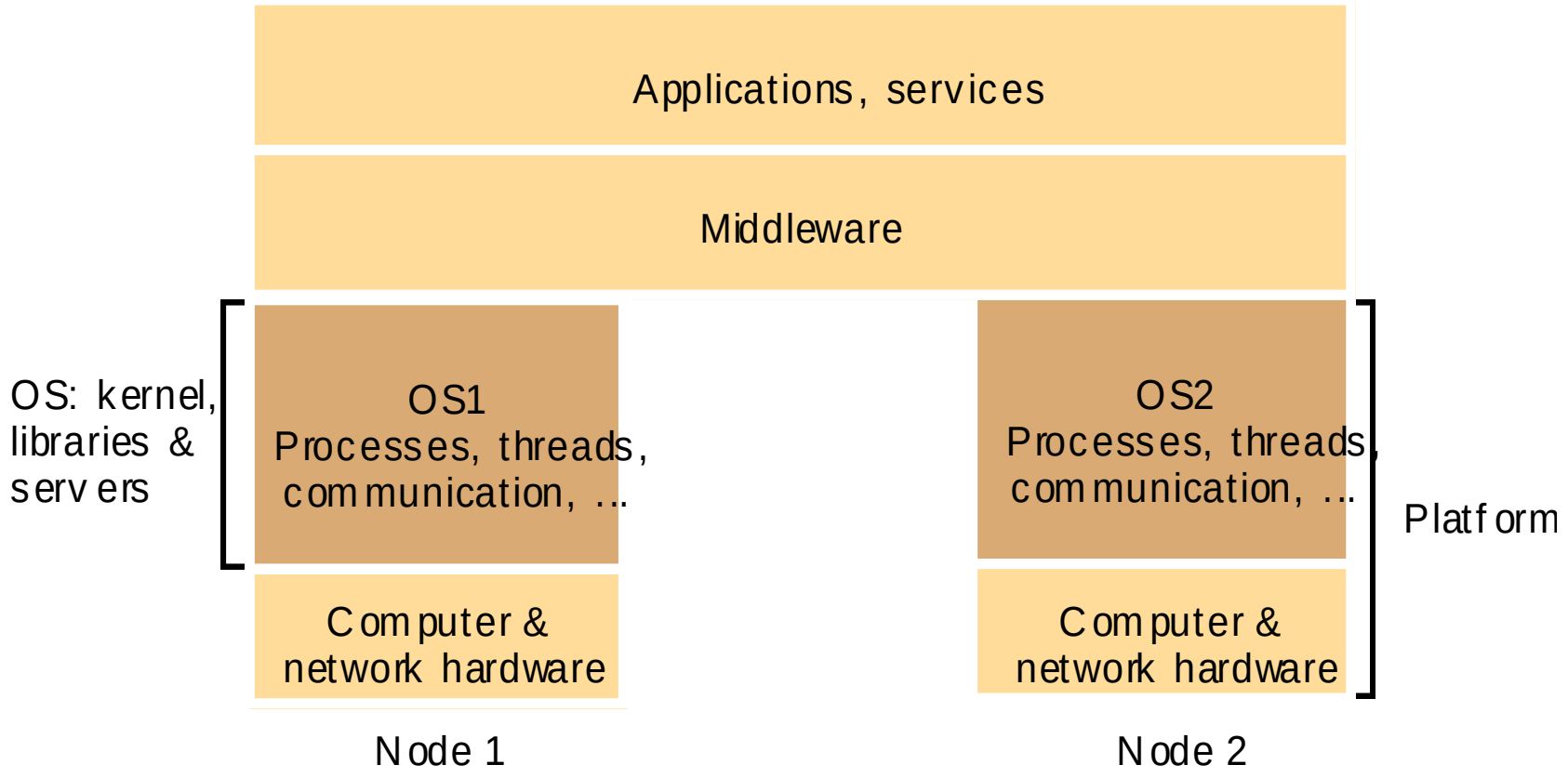
DOS vs. NOS

System	Description	Main Goal
DOS	Tightly-coupled operating system for multi-processors and homogeneous multicomputers.	Hide and manage hardware resources.
NOS	Loosely-coupled operating system for heterogeneous multicomputers (LAN and WAN).	Offer local services to remote clients.
NOS + Middleware	Loosely-coupled operating system for heterogeneous multicomputers (LAN and WAN) + Additional layer at top of NOS implementing general-purpose services.	Offer local services to remote clients + Provide distribution transparency.

Operating System Layer

- **Middleware** runs on a variety of OS – hardware combinations (platforms) at the nodes of a distributed system.
- The **OS** running at a node – a kernel and associated user-level services such as communication libraries – provides its own flavor of abstractions of local hardware resources for processing, storage and communication.
- **Middleware** utilizes a combination of these local resources to implement its mechanisms for remote invocations between objects or processes at the nodes.
- **Figure 7.1** shows how the **operating system layer** at each of two nodes supports a common middleware layer in providing a distributed infrastructure for applications and services.

Figure 7.1 System Layers



Operating System Layer

Kernels and **server processes** are the components that manage resources and present clients with an interface to the resources. As such, the following are required:

- ***Encapsulation:*** They should provide a useful service interface to their resources – that is, a set of operations that meet their clients' needs. Details such as management of memory and devices used to implement resources should be hidden from clients.
- ***Protection:*** Resources require protection from accesses – for example, files are protected from being read by users without read permissions.
- ***Concurrent processing:*** Clients may share resources and access them concurrently. Resource managers are responsible for achieving concurrency transparency.

Operating System Layer

Clients access resources by making remote method invocations to a server object, or system calls to a kernel.

A combination of **libraries**, **kernels** and **servers** may be called upon to perform the following invocation-related tasks:

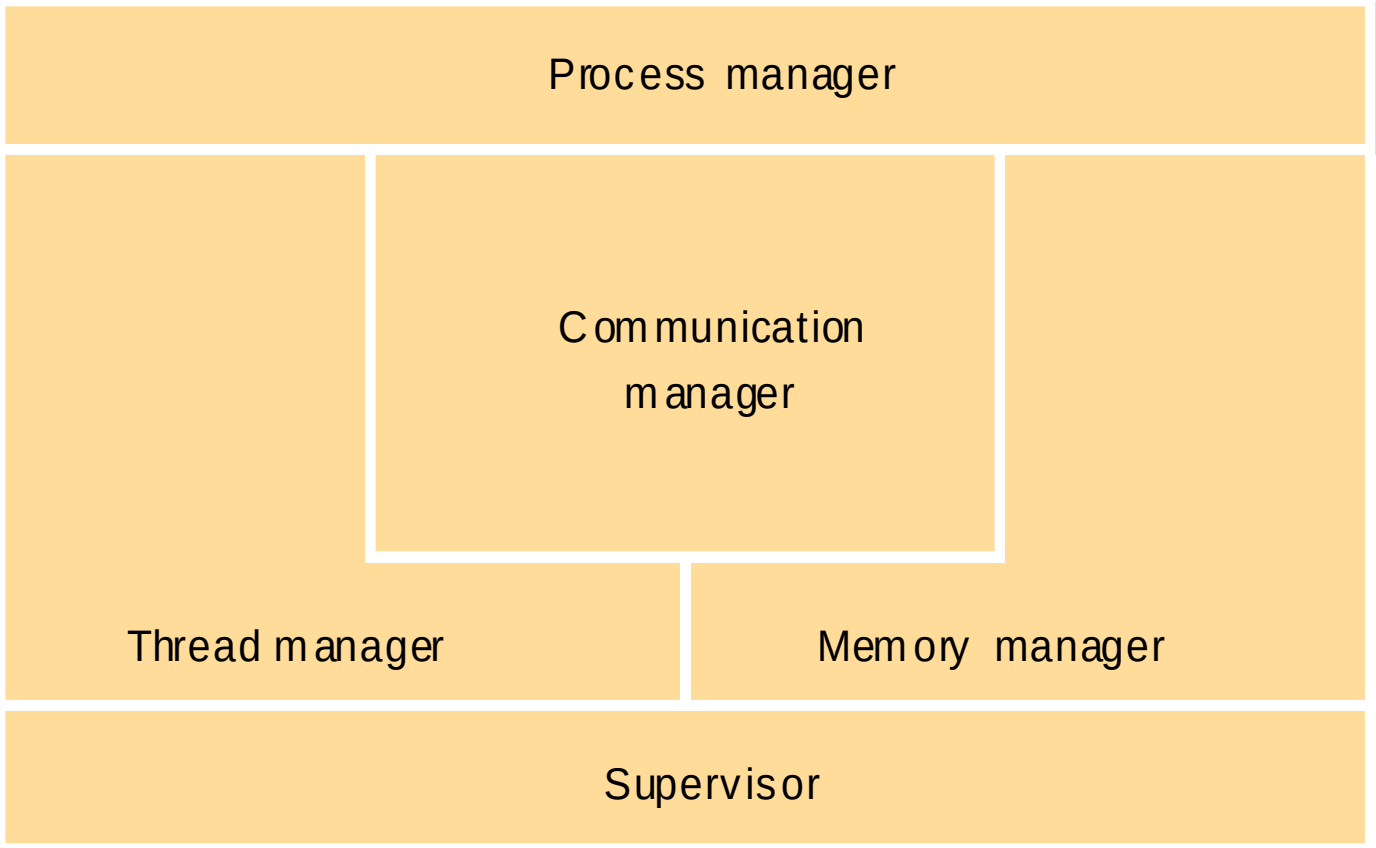
- ***Communication:*** Operation parameters and results must be passed to and from resource managers, over a network or within a computer.
- ***Scheduling:*** When an operation is invoked, its processing must be scheduled within the kernel or server.

Operating System Layer

Figure 7.2 shows the **core OS functionality** that are concerned:

- Process and thread management
- Memory management
- Communication between processes on the same computer (horizontal divisions in the figure denote dependencies).

Figure 7.2 Core OS Functionality



Operating System Layer

The **core OS components** and their responsibilities are:

- ***Process manager:*** creates ,executes and terminates processes. Thus, a **process** is a unit of resource management, including an address space and one or more threads.
- ***Thread manager:*** creates and executes threads within a process. Also, it provides thread synchronization and scheduling. Threads are schedulable activities attached to processes.
- ***Communication manager:*** communicates between threads attached to different processes on the same node (computer).
- ***Memory manager:*** Management of physical and virtual memory.
- ***Supervisor:*** Dispatching of interrupts, system call traps, and exceptions. Control of memory management unit and hardware caches.

- **Physical Memory: RAM (represented as frames).**
- **Virtual Memory (represented as pages): If memory exceeds RAM capacity as it is needed, then, some parts in RAM are returned back to hard disk and other parts brought from hard disk to RAM. It is used when a program wants to run, and it is larger than available RAM size.**

Protection

What is Kernel?

- Kernel is a program that is distinguished by the facts that it remains loaded from system initialization.
- Its code is executed with complete access privileges for the physical resources on its host computer.
- It can control the memory management unit and set the processor registers so that no other code may access the machine’s physical resources except in acceptable ways.

Protection

Types of processes:

1) **User Process**
   - A process related to user application.
   - It works in its own address space.
   - It cannot directly access other system resource or address space outside its own.
   - When the processor is executing this process, the processor goes into user (unprivileged) mode.
2) **Kernel Process**
   - A process related to accessing system recourses.
   - It has the right to access wide range of address space.
   - When the processor is executing this process, the processor goes into supervisor (privileged) mode.
   - The kernel makes sure processor stays in user mode when executing user process.

Protection

- In DS, resources (memory, Hard Disk, CPU, etc.) require protection from **illegitimate accesses**.
- Preventing illegitimate accesses to resources is done via Kernel.

Protection

In this case the process is "user process" and it executes while processor is in user mode.

When the processor encounters a kernel statement. This means it needs special privileges, then the process makes a **system call trap** via the **kernel** to allow it to become a "kernel process" then processor works under supervisor mode and code is executed. After kernel code is executed, the process becomes user process and processor goes back to user mode.

A Program Run by a Process

Suppose these statement are **user code**.

Suppose these statement are **kernel code**.

Statement 1
Statement 2
Statement 3
..
..
..
..
Statement *n*

Note: Switching between user mode and supervised mode is expensive because it requires switching from user address space to kernel address space and vice versa.

Processes and Threads

- A **process** is a program under execution.
- A **process** consists of an execution environment together with one or more threads.
- A **thread** is an OS abstraction of an activity within a process.

Execution environment (for each process) consists of:

1. **Address space** resource (e.g., windows).
2. **Threads synchronization** and **communication** resources such as semaphores and communication interfaces (for example, sockets).
3. **Higher-level** resources such as open files.

Semaphores

- Processes might share variables.
- In this case, some parts can only be accessed one process at a time. We call it "**Critical Section**".
- **Semaphores** can guarantee that only one process enters the critical section at one time.
- **Mutex** Semaphores use two operations:

  1) **Wait**: Let a process wait in a queue.

  2) **Signal**: Allow some or all of processes in the waiting queue to proceed their work.

Example of using Semaphores

```
Function withdraw_account(account, amount){
    wait (account);
    x = getBalance(account);
    x = x – amount;
    putBalance(account, x);
    signal(account);
    printBalance(x);
}
```

```
Class Account {
    double balance;
    Queue waitingQ;
}
```

Suppose 3 processes wants to withdraw from account.

P1: wait (account);
P1: x = getBalance(account);
P2: wait (account);
P3: wait (account);
P1: x = x – amount;
P1: putBalance(account, x);
P1: signal(account);
P1: printBalance(x);

Time

P1 granted access because its not being used by any other process.

P2 and P3 have to wait in the queue of processed waiting for account.

This lets P2 and P3 out of the waiting queue.

Processes and Threads

- **Execution environments** are normally expensive to create and manage, but several threads can share them – that is, they can share all resources accessible within them.
- An **execution environment** represents the protection domain in which its threads execute.
- **Threads** can be created and destroyed dynamically, as needed.
- The aim of having **multiple threads** of execution is to maximize the degree of concurrent execution between operations, thus enabling the overlap of computation with input and output and enabling concurrent processing on multiprocessors.

Processes and Threads: **Address Spaces**

- An **address space** is a unit of management of a process's virtual memory.
- It is **large** (typically up to 2^{32} bytes, and sometimes up to 2^{64} bytes) and consists of one or more regions, separated by inaccessible areas of virtual memory (i.e., Inaccessible because they are reserved for implementing the virtual memory concept that we explained before).
- A **region** (**Fig. 7.3**) is an area of contiguous virtual memory that is accessible by the threads of the owning process.
- **Regions** do not overlap.

Processes and Threads: Address Spaces

Each **region** is specified by the following **properties**:

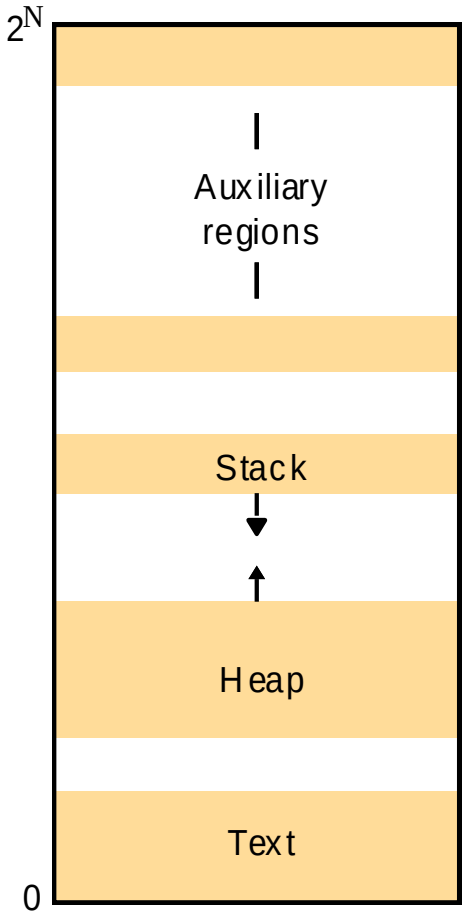
1. Its extent (lowest virtual address and size).
2. read/write/execute permissions for the process's threads.
3. Whether it can be grown upwards or downwards.

Fig. 7.3 Shows page-oriented model, and it has **three regions**:

1. A fixed, unmodifiable **text region** containing program code.
2. A **heap**, part of which is initialized by values stored in the program's binary file, and which is extensible towards higher virtual addresses.
3. A **stack**, which is extensible towards lower virtual addresses.

- **Heap:** A memory that can be used by the process for example to store some variables.
- **Stack:** A memory used mainly for function calls. It helps specially in nested function calls.

Figure 7.3 Address Space

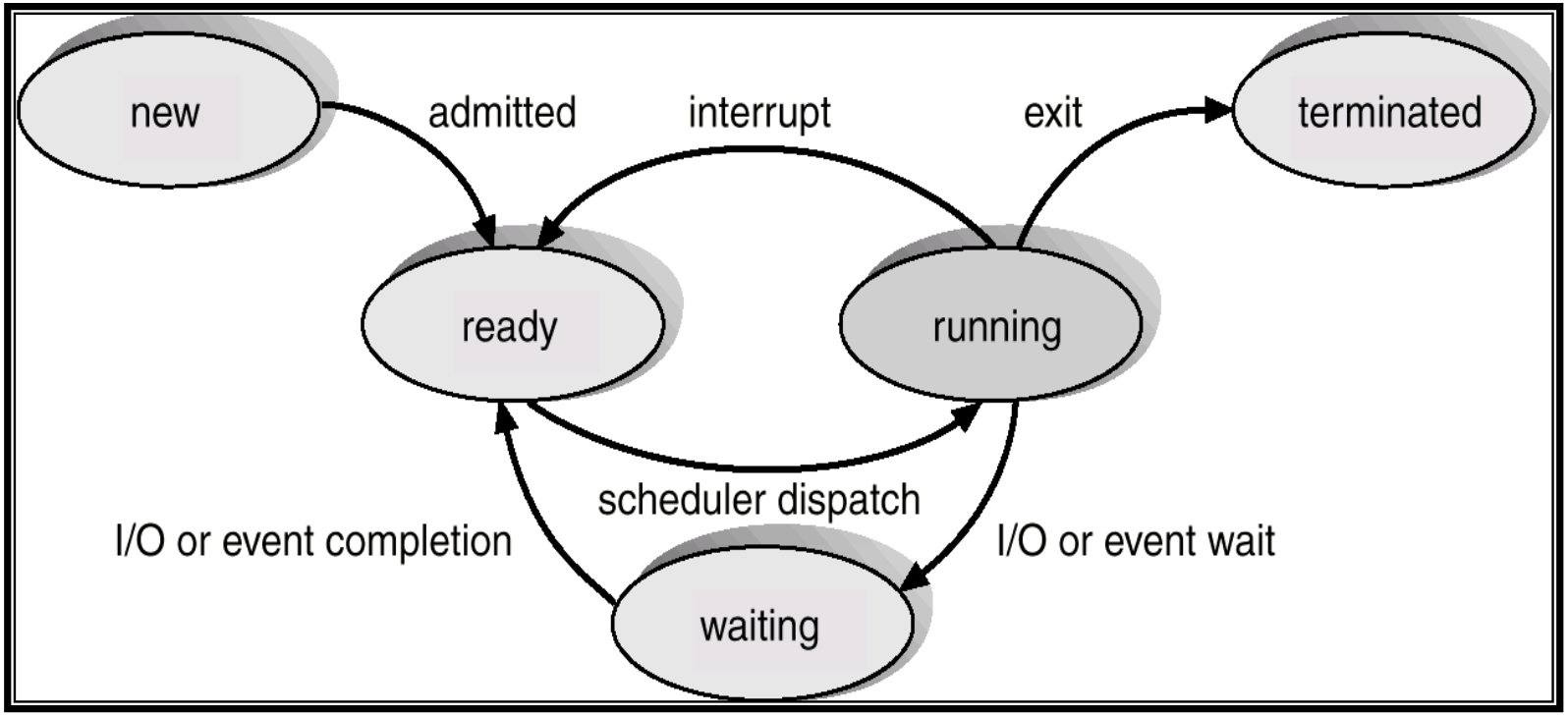


Process States

As a process executes, it changes ***state*:**

- **New**: The process is being created.
- **Running**: Instructions are being executed.
- **Waiting**: The process is waiting for some event to occur.
- **Ready**: The process is waiting to be assigned to a processor.
- **Terminated**: The process has finished execution.

Process States



Scheduler Dispatch: The scheduler assigns a process to a processor (CPU).

Processes and Threads: **Creation of Process**

- The **creation of a new process** is provided by the operating system. For example, the UNIX ***fork*** system call creates a process.
- For a **distributed system**, the design of the process-creation mechanism must consider the utilization of multiple computers.

The **creation of a new process** can be separated into two independent aspects:

1. The choice of a target host, for example, the host may be chosen from among the nodes in a cluster of computers acting as a compute server.
2. The creation of an execution environment inside the target host (and an initial thread within it).

Processes and Threads: Creation of Process

The choice of **hosting process** based on:

1. **Transfer policy** determines whether to locate a new process locally or remotely. This may depend on whether the local node is lightly or heavily loaded.

2. **Location policy** depends on the relative loads of nodes on their machine architectures.

Processes and Threads: **Creation of 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 other resources, such as default open files).

There are **two approaches** to defining and initializing the address space of a newly created process:

1. The **first approach** is used where the address space is of a statically defined format. For example, it could contain just a program text region, heap region and stack region.
2. The **second approach** is when the address space can be defined with respect to an existing execution environment (example, parent-child processes share a region).

Processes and Threads: Creation of Execution Environment

- When **parent and child share a region**, the page frames (units of physical memory corresponding to virtual memory pages) belonging to the parent's region are mapped simultaneously into the corresponding child region.

Processes and Threads: **Threads**

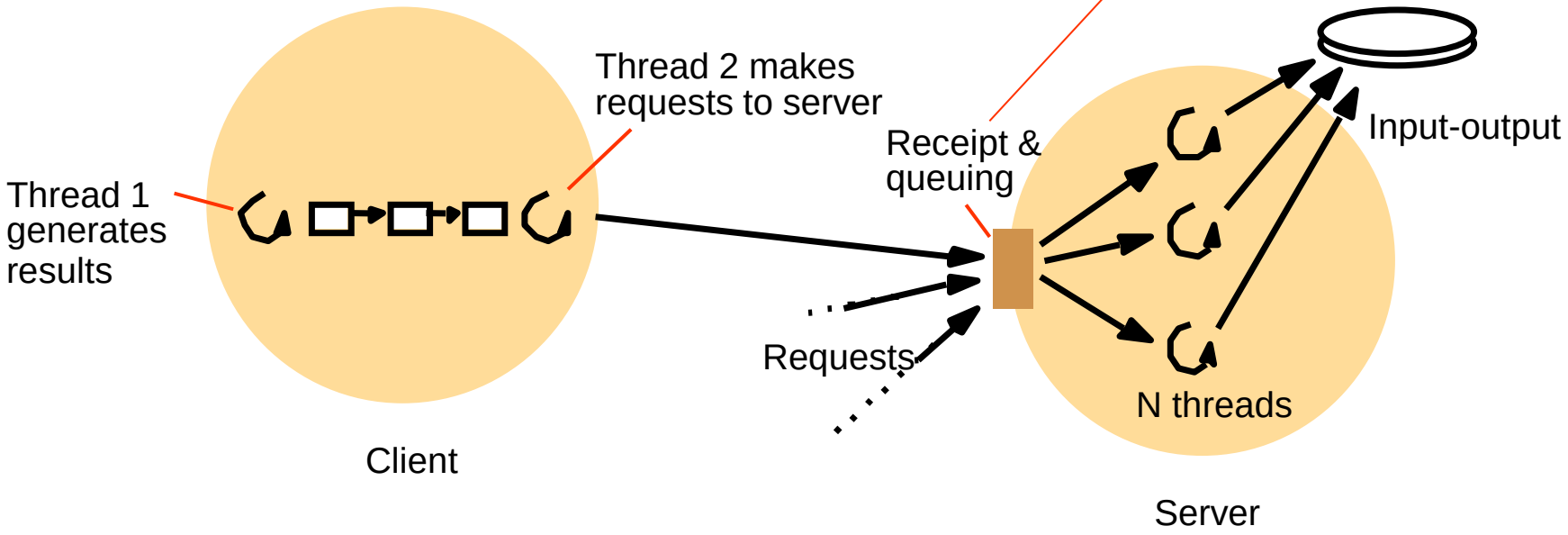
Threads increase servers' throughput (**Figs: 7.5**).

- Consider the server shown in **Fig. 7.5**, the server has a pool of one or more threads, each of which repeatedly removes a request from a queue of received requests and processes it.
- Assume that each request takes, on average, 2 milliseconds of processing plus 8 milliseconds of I/O (input/output) delay when the server reads from a disk.
- If a single thread has to perform all processing, then the turnaround time for handling any request is on average 2 + 8 = 10 milliseconds.

- Assuming one CPU.
- Remember that processing requires utilizing the CPU.
- Remember that I/O operation requires using I/O devices, not the CPU.
- So, several threads cannot use the CPU at the same time.
- However, one thread can be working on CPU and another thread can be working on some I/O device simultaneously.

Figure 7.5 Client and server with threads

This is I/O operation which is also done with a separate thread.



Two threads at client:

- **Thread 2** contacts the server and receives results (processing).
- **Thread 1** displays results (I/O operation) coming from requests.

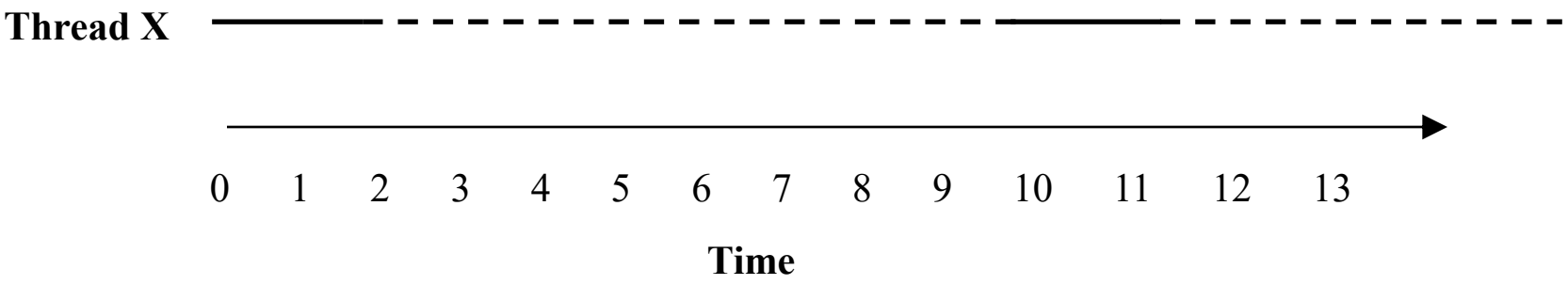
With several threads on the server:

- Assume one CPU at the server.
- The server assign a specific thread for each request in the queue.
- For example, **thread X** handles one request and **thread Y** handles another request.
- When **thread X** needs to use I/O device at server, no need for it to keep holding the CPU. So, CPU is given to **thread Y** while **thread X** is using its I/O operation.

Processes and Threads: Threads

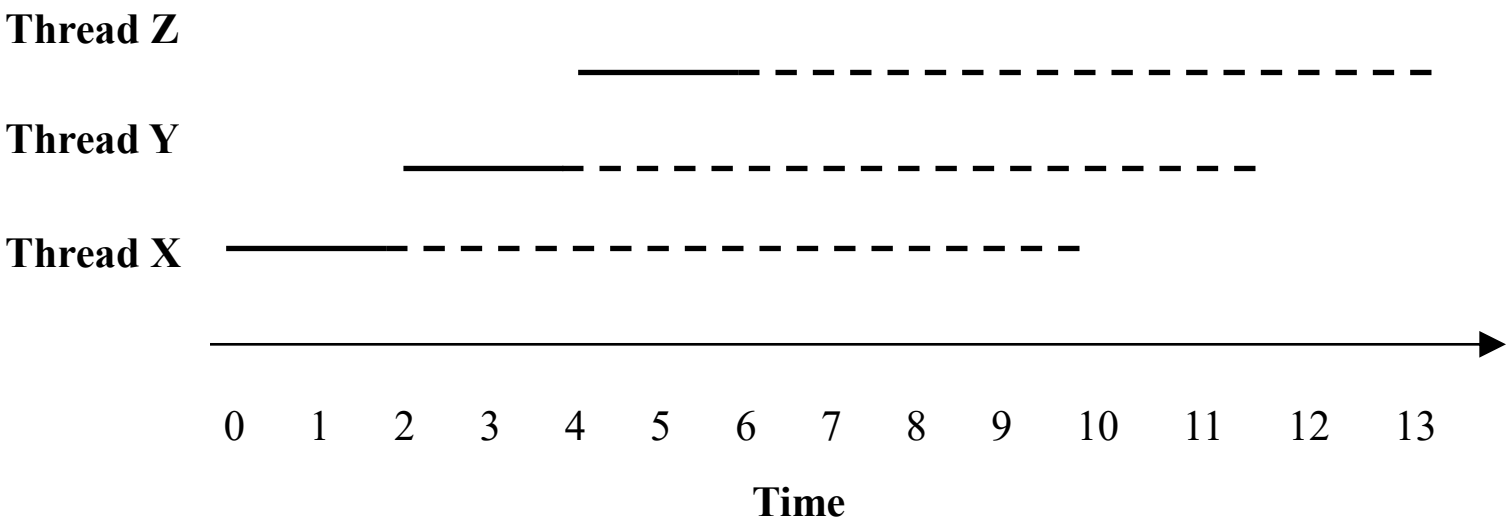
- **Multithreading** enables servers to maximize their throughput.
- **Figure 7.5 shows** one of the possible threading architectures, the **worker pool architecture**.
- The server creates a fixed pool of '**worker**' threads to process the requests when it starts up.
- The module marked '**receipt and queuing**' in **Fig. 7.5** is implemented by an '**I/O**' thread, which receives requests from a collection of sockets or ports and places them on a shared request queue for retrieval by the workers.

Without Multithreading



- Solid line is processing (CPU Time).
- Dashed line is performing I/O operation (I/O Time).
- In this case the results of each thread appears after 10 milliseconds.

With Multithreading



- Solid line is processing (CPU Time).
- Dashed line is performing I/O operation (I/O Time).
- The results of **thread X** appears after 10 milliseconds.
- The results of all subsequent threads starts appearing in 2 seconds after the previous thread.

Processes and Threads: Threads

Threads lifetime states: run, suspended, ready.

Comparison between **processes and threads**:

1. Creating a new thread in an existing process is cheaper than creating a new process.
2. Switching among threads is cheaper.
3. Threads can share data and resources.
4. Threads are not protected from one another.

- All the previous points stem from the fact that creating a new process requires creating a new memory space for it which is a time-consuming process.
- However, threads don't require the same thing because they share the same address space of the parent.

Communication and Invocation

Here we concentrate on **communication** as a part of the implementation of what we have called an invocation; such as, Remote Procedure Call (**RPC**) or Remote Method Invocation (**RMI**).

OS system design issues and concepts involves the following:

- **Communication primitives** provided by OS: Some kernels designed for DS have provided communication primitives.
- **Protocol and openness:** One of the main requirements of the OS is to provide standard protocols to enable interworking between middleware implementation of different platforms.

Communication and Invocation

Figure 7.11 shows the particular cases of a system call, a remote invocation between processes hosted at the same computer, and a remote invocation between processes at different nodes in the distributed system.

Figure 7.11 Invocations between address spaces