Distributed Operating Systems
Processes

Ewa Niewiadomska-Szynkiewicz and Adam Kozakiewicz

ens@ia.pw.edu.pl, akozakie@ia.pw.edu.pl

Institute of Control and Computation Engineering
Warsaw University of Technology

Lecture (2)

Summary:

- √ Introduction
- √ Processes and Threads
- √ Client-server Model
- √ Software Agents
- √ Processes Migration
- √ Local and Global Scheduling; Load Sharing and Load Balancing Algorithms

Introduction

- √ Different types of processes play a crucial role in distributed systems.
- √ The management and scheduling of processes are the most important issues to deal with in DS.
- √ Communication takes place between processes.
- √ Moving processes between different machines can help in achieving scalability and dynamic configure client and servers.

Processes

Process

A process is an instance of a computer program that is being **sequentially executed**. While a program itself is just a passive collection of instructions, a process is the actual execution of those instructions.

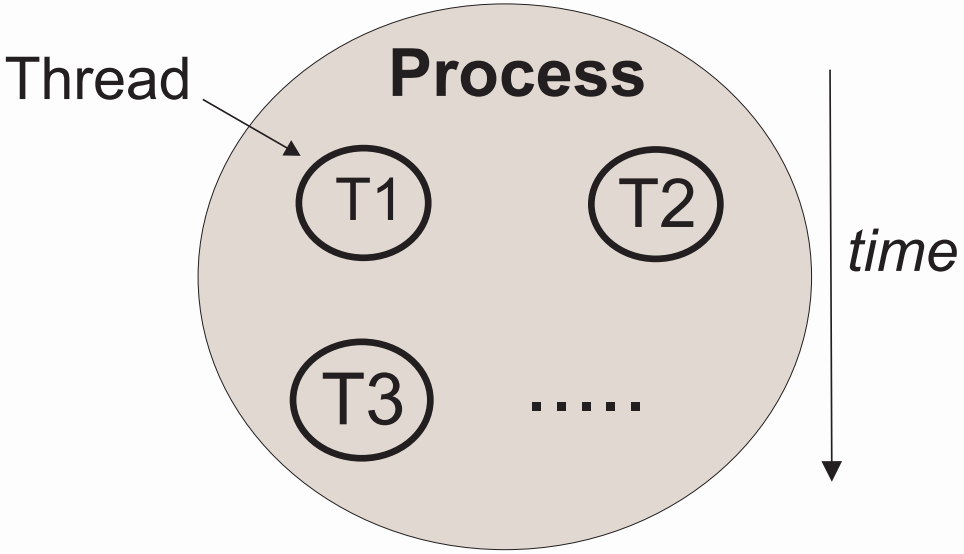
A computer system process consists of the following resources (in general):

- √ an image of the executable machine code associated with a program,
- √ memory which includes the executable code, process-specific data (input and output), a call stack (to keep track of active subroutines and/or other events), and a heap to hold intermediate computation data generated during run time,
- √ operating system descriptors of resources allocated to the process (descriptors in Unix or handles in Windows), data sources and sinks,
- √ processor state (content of registers, physical memory addressing, etc.),
- √ security attributes, such as the process owner and the process' set of permissions (allowable operations).

Threads

Thread

Thread is a part of a program that can execute independently of other parts.



A single process may contain several executable programs (threads) that work together as a coherent whole.

Operating systems that support multithreading enable programmers to design programs whose threaded parts can execute concurrently.

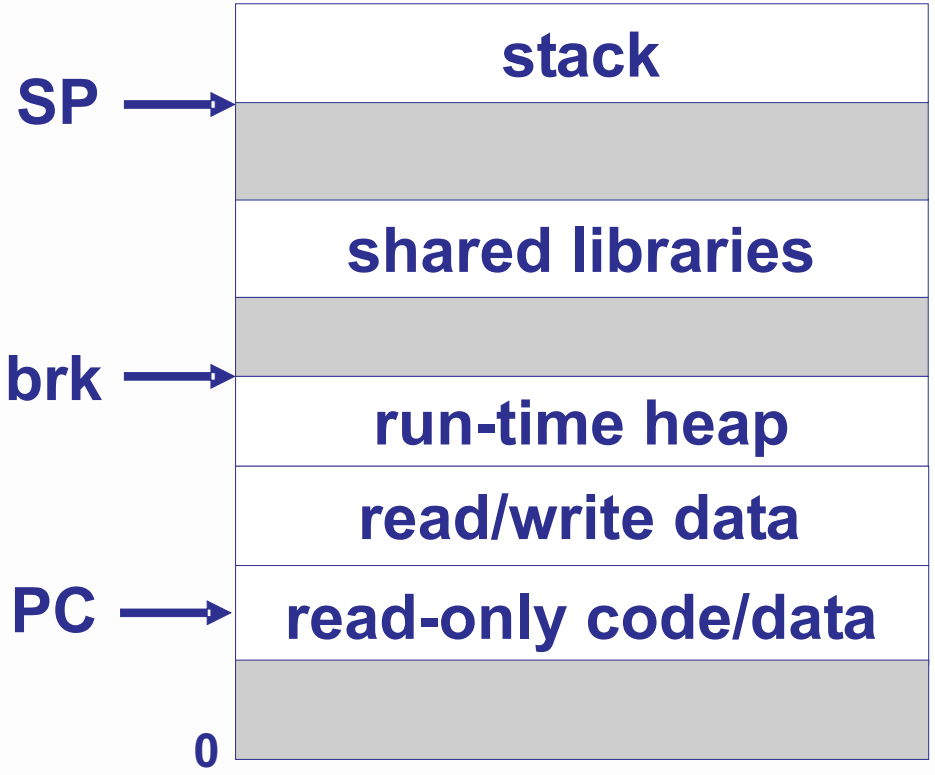
Traditional View of a Process

Process context

Program context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

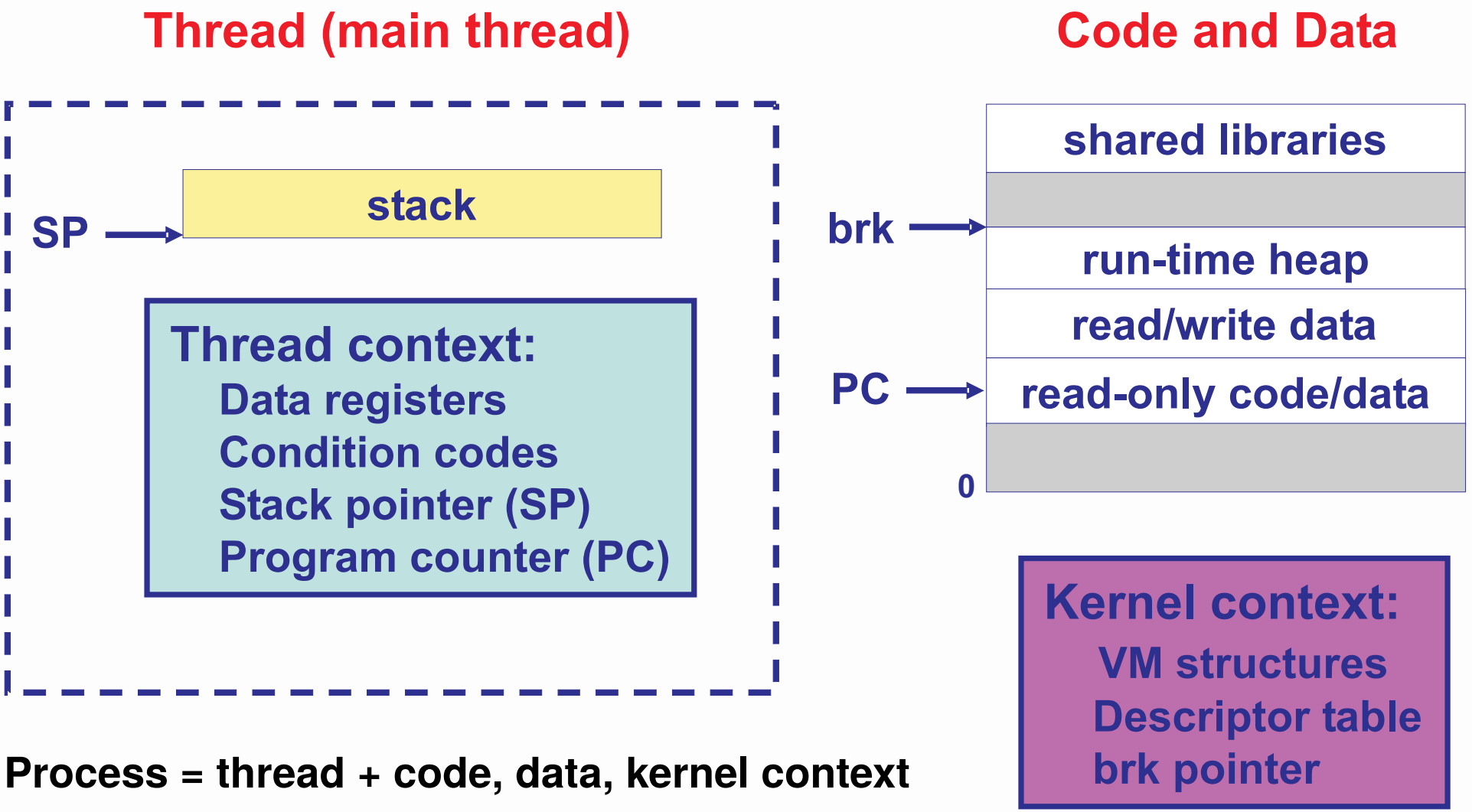
Kernel context:
- VM structures
- Descriptor table
- brk pointer

Code, data, and stack



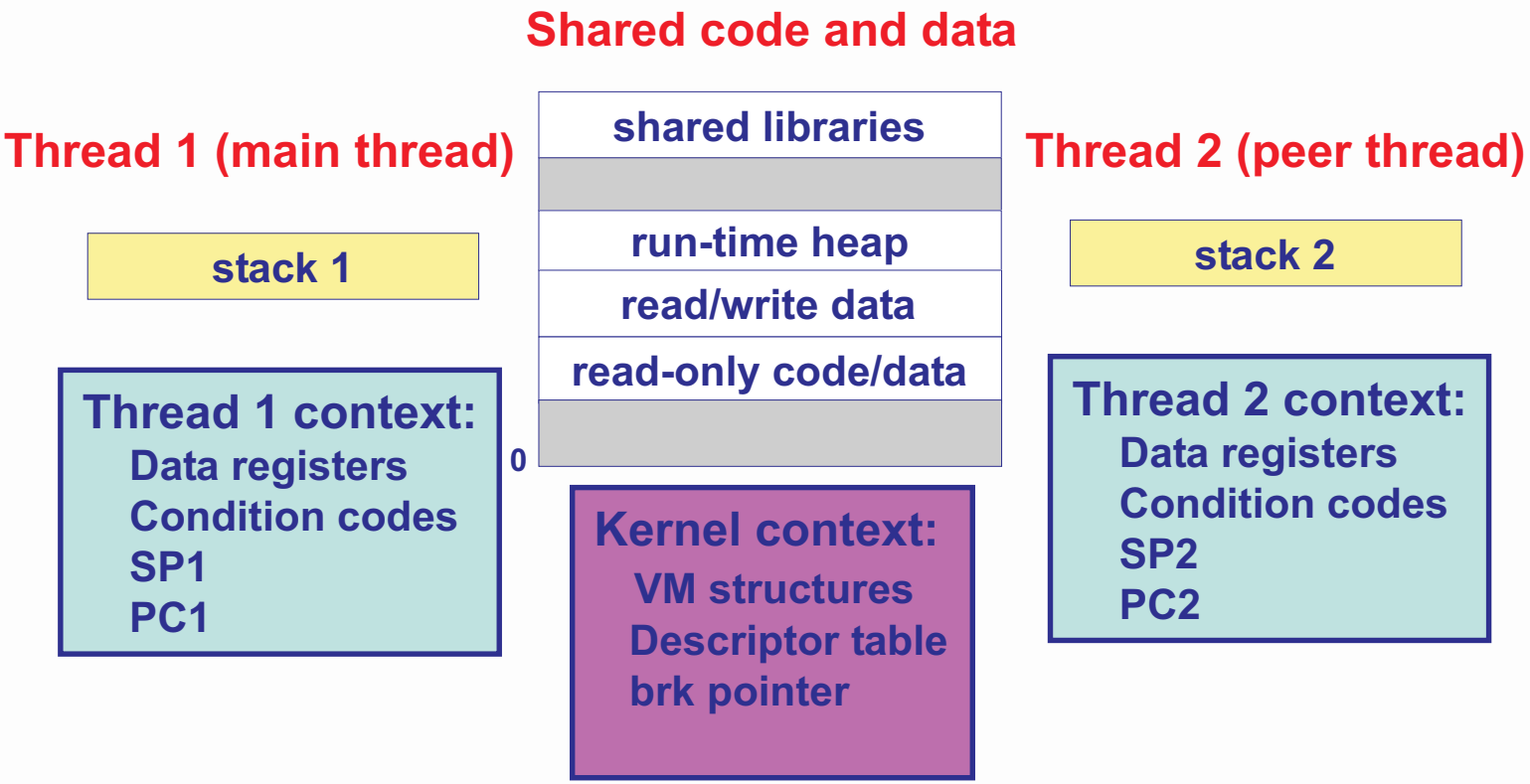
Process = process context + code, data, stack

Alternate View of a Process



Process = thread + code, data, kernel context

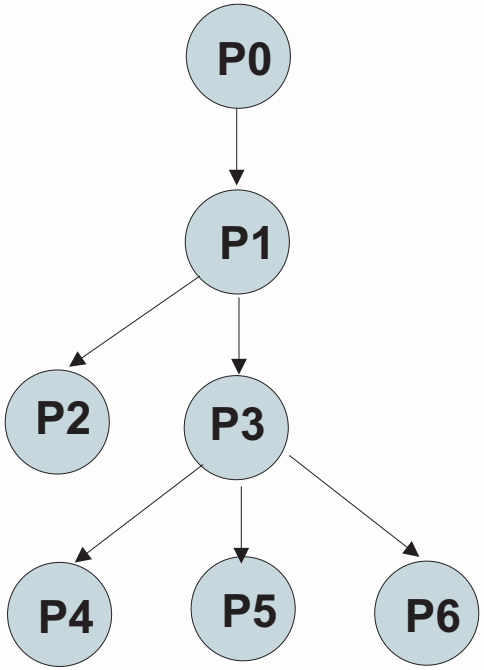
Process with Multiple Threads



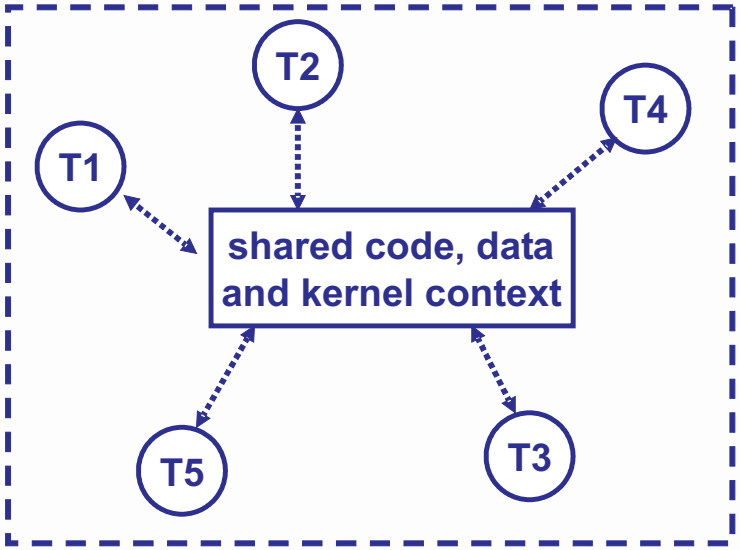
Multiple threads can be associated with a process.

- √ each thread has its own logical control flow,
- √ each thread shares the some code, data and kernel context, each thread has its own thread id (TID).

Logical View of Processes and Threads



Process hierarchy

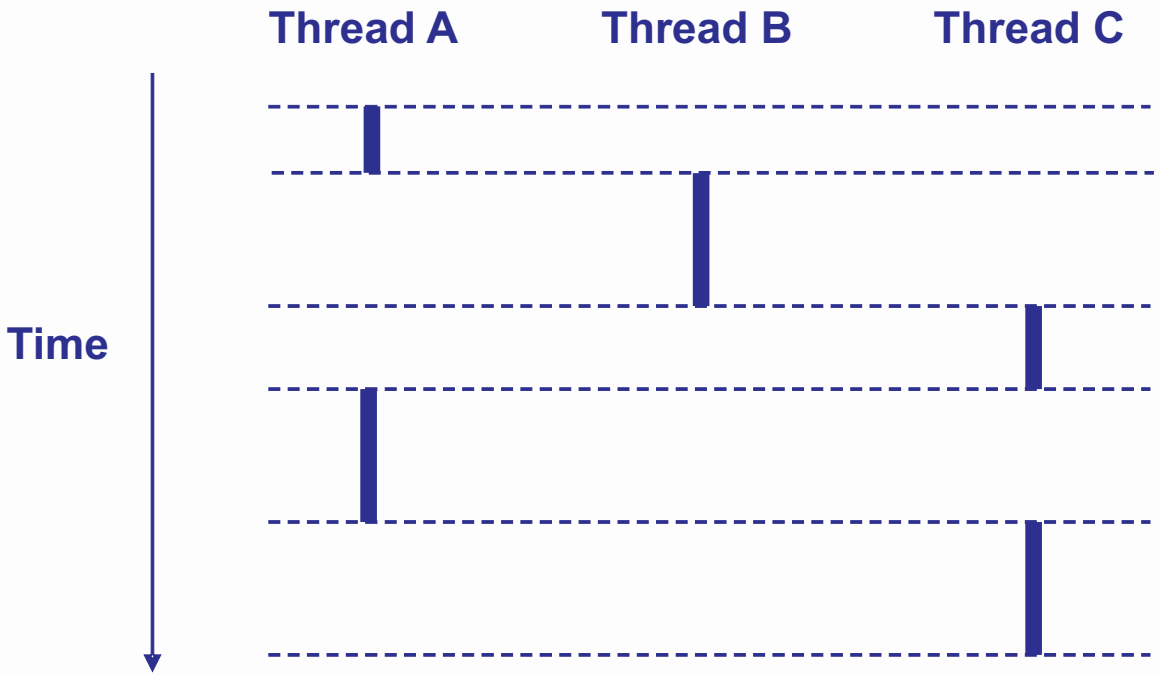


Threads associated with process P5

- √ Processes form a tree hierarchy
- √ Threads associated with a process form a pool of peers

Concurrent Thread Execution

Two threads run concurrently if their logical flows overlap in time.



- √ concurrent: A & B, A & C
- √ sequential: B & C

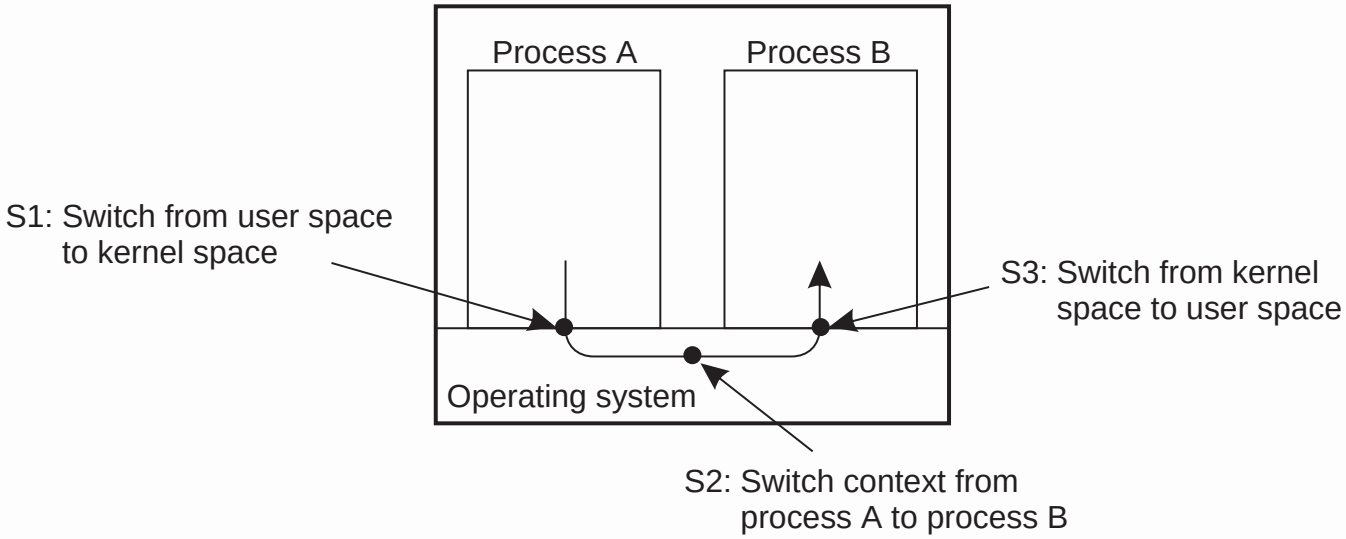
Process and Thread States

Process/thread states determine how the process/thread is handled by the operating system kernel (the specific implementations of states vary in different operating systems).

Processes/threads go through the following states:

- √ *Created* (or *New*) - when a process/thread is first created. It awaits admission to the *Ready* state. This admission will be approved or delayed;
- √ *Ready* (or *Waiting*) - process/thread has been loaded into main memory and is awaiting execution on a CPU;
- √ *Running* (or *Active*) - process/thread which is currently executing on CPU;
- √ *Blocked* (or *Sleeping*) - process/thread is ”block” on a resource (file, a semaphore or a device), it will be removed from the CPU. It will remain *Blocked* until its resource becomes available;
- √ *Terminated* - a process/thread is terminated.

Context Switching between Processes



Context switching between processes as the result of interprocess communication (IPC)

The applications can be developed as a collecting cooperating processes (approach typical in UNIX).
Processes cooperation implemented by means of interprocess communication mechanism (IPC).
IPC drawback – communication requires extensive context switching (3 points in the figure).

Threads and Processes - Summary

A **thread** is the ”lightest” unit of kernel scheduling.

- √ At least one thread exists within each process.
- √ Multiple threads can exist within a process (they share memory and other resources).
- √ Threads do not own resources except for a stack and a copy of the registers including the program counter.

A **process** is the ”heaviest” unit of kernel scheduling.

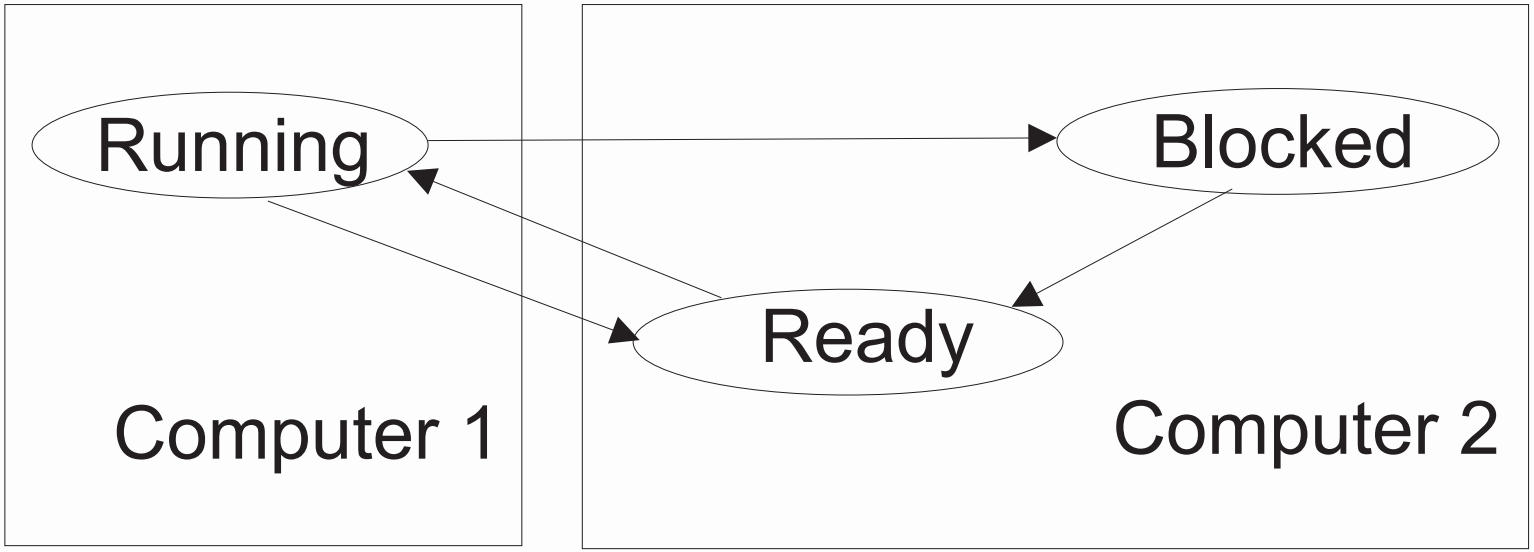
- √ Typically independent. Process owns resources allocated by the operating system (memory, file handles, sockets, device handles, and windows).
- √ Process does not share address spaces or file resources (except through explicit methods).
- √ Interact through system-provided interprocess communication mechanisms.

Context switching between threads in the same process is typically faster than context switching between processes.

Distributed Processes

Distributed process

A set of cooperating sequential processes executed at the same time (concurrently) in distributed computer architecture. They carry out the common goal. They are distributed in space and state.



Models of Processes

Static process model - processes are created after start the program execution.

Dynamic process model - processes can be created and destroy at any time during the program execution.

Client-server Communication Model

Client-server (Master-slave)

Client-server is a computing architecture which separates a **client** (initiates services) from a **server** (performs services), and is implemented over a computer network.

Characteristics of a client:

- √ initiates requests (service),
- √ waits for and receives replies,
- √ usually connects to a small number of servers at one time,
- √ typically interacts directly with end-users.

Characteristics of a server:

- √ waits for requests from clients (passive - slave),
- √ processes requests and then replies to the client,
- √ usually accepts connections from a large number of clients,
- √ typically does not interact directly with end-users

Examples: HTTP, FTP, DNS, finger, gopher, etc.

Peer-to-Peer Communication Model

Peer-to-peer (P2P)

A peer-to-peer network does not have the notion of clients or servers, but only equal peer nodes that simultaneously function as both **client** and **server** to the other nodes on the network.

Client-server Model – Advantages

- √ **Greater ease of maintenance** (independence from change) – A client-server architecture enables the roles and responsibilities of a computing system to be distributed among several independent computers.
- √ **Security** – The data are stored on the servers, which generally have far greater security controls than most clients. Servers can better control access and resources, to guarantee that only those clients with the appropriate permissions may access and change data.
- √ **Data administration** – Since data storage is centralized, updates to those data are far easier to administer.
- √ Many client-server technologies are already available (ensure: security, 'friendliness' of the user interface, and ease of use).
- √ It functions with multiple different clients of different capabilities.

Client-server Model - Disadvantages

- √ **Traffic congestion** – As the number of simultaneous client requests to a given server increases, the server can become severely overloaded.
- √ **Robustness** – The client-server paradigm lacks the robustness of a good P2P network. In the case of a critical server fail, clients requests cannot be fulfilled.

Multithread Clients

Benefits to using multithread clients:

- √ **Transparent architecture** – the application consists of several modules.
- √ **Many tasks can be performed simultaneously** – developing an application as a multithread client simplifies matters considerably.
- √ **Many connections to the servers can be opened simultaneously** – e.g. replicated servers (located at the same site and known under the same name).
- √ **Friendliness** – the part of client's application can be executed but the user that can control it.

Multithread Servers

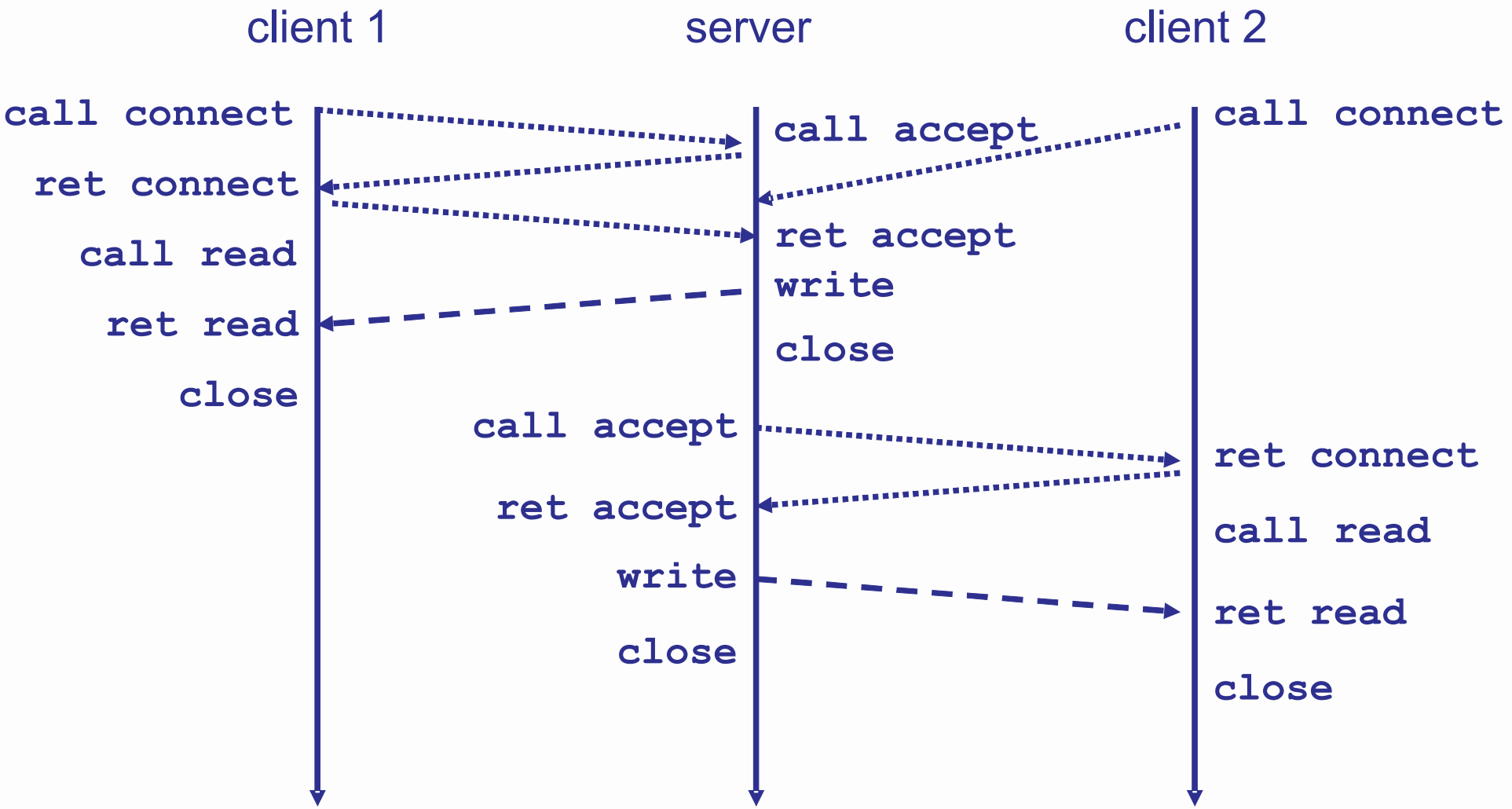
We can distinguish different types w.r.t.:

- √ The manner to serve the client's request:
  1. **iterative server** – services one client at a time
  2. **concurrent server** – can service multiple clients concurrently
- √ Maintaining of the state information:
  1. **Stateless server**
  2. **Stateful server**

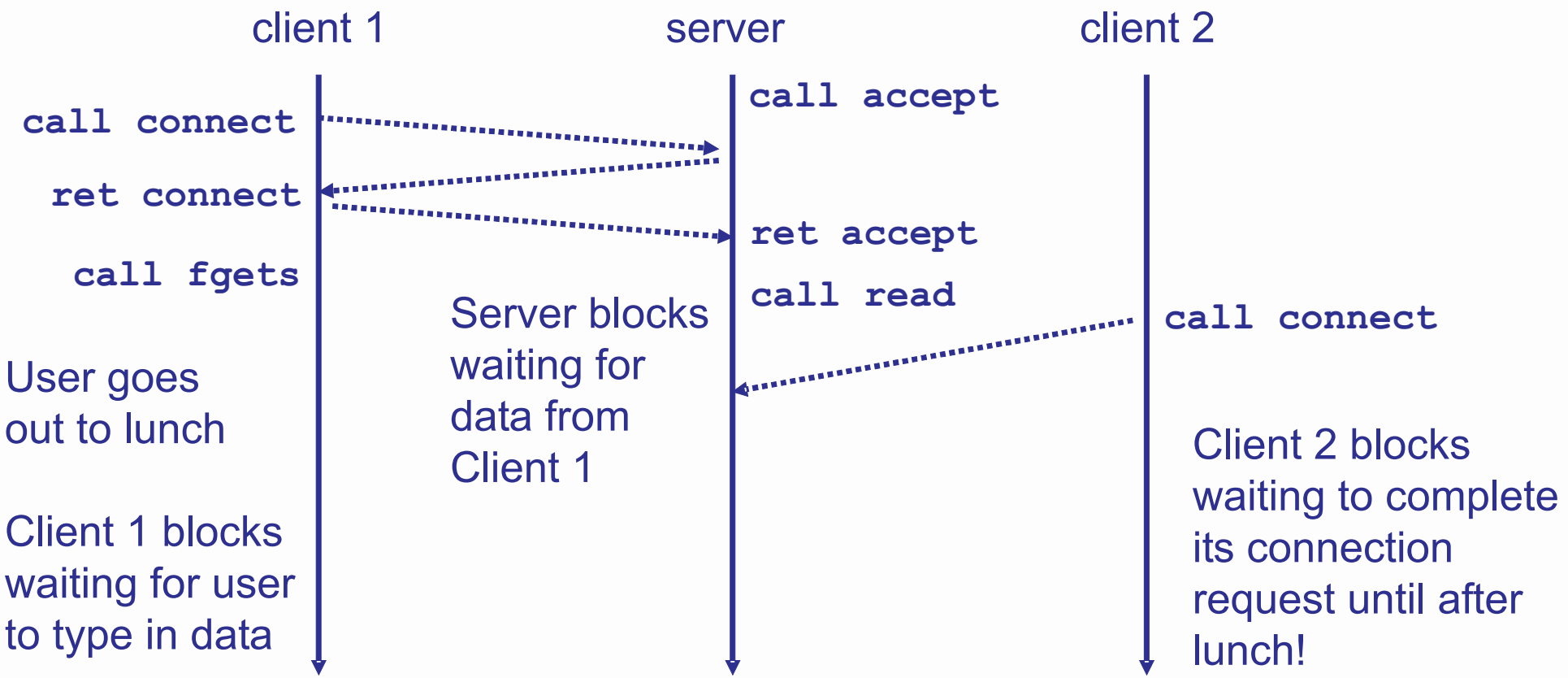
Which type of server is chosen is a design issue.

Iterative Server

Processes one request at a time



Fundamental Flaw of Iterative Server

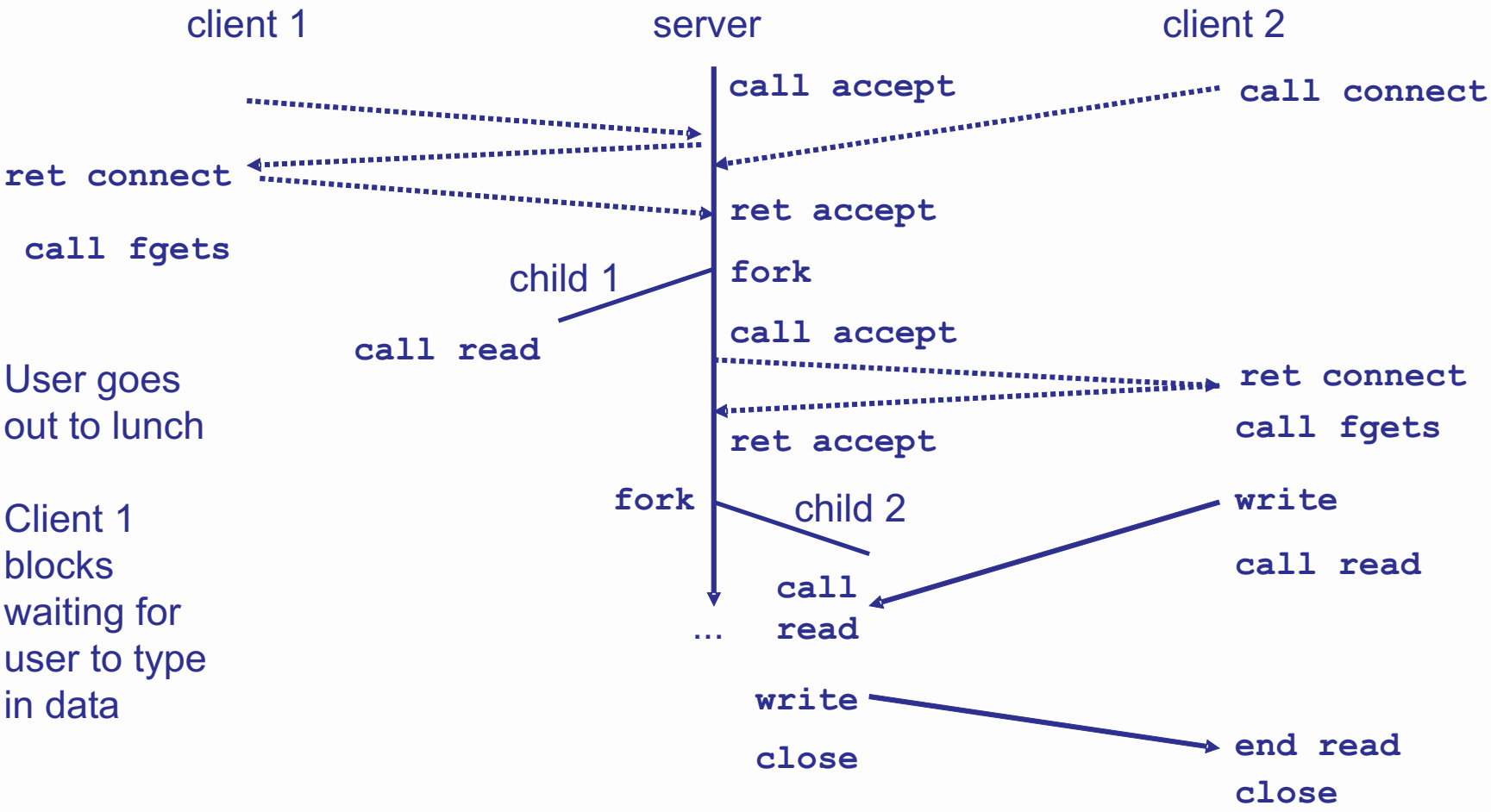


Solution – concurrent server

Concurrent Server

Use multiple concurrent flows to serve multiple clients at the same time

Concurrent servers handle multiple requests concurrently.

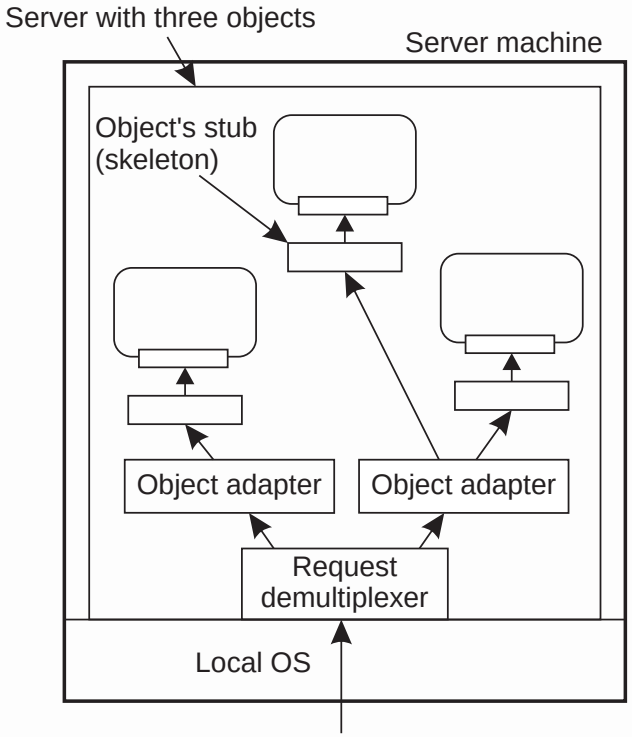


Stateful vs. Stateless Server

- √ **Stateless server** – does not keep information on the state of its clients, and can change its own state without having to inform any client (example: Web server).
- √ **Stateful server** – maintain state information on its clients (example: file server).
- √ **Stateless server** is straightforward to code.
- √ **Stateful server** is harder to code, but the state information maintained by the server can reduce the data exchanged, and allows enhancements to a basic service.
- √ Maintaining stateful information is difficult in the presence of failures.

Object Server

A server tailored to support distributed objects.
It does not really provide a specific service – specific services are implemented by the objects that reside in the server. The server provides only the means to invoke local objects, based on requests from remote clients.



Organization of an object server supporting different activation policies (object adapters).

Software Agents

Software agent

Autonomous process capable of reacting to, and initiating changes in its environment, possibly in collaboration with users and other agents.

Broad definition – many different types of processes can easily be called an agent.

Types of Agents:

- √ **collaborative agents** - agents seek to achieve common goal through collaboration,
- √ **mobile agents** - have capability to move between different machines,
- √ **information agents** - manage information from many different sources,
- √ **interface agents** - assist an end user in the use of one or more applications.

ACC (*Agent Communication Channel*) - take care of all communication between different agent platforms (in particular is responsible for for reliable and ordered point-to-point communication with other platforms).
ACL (*Agent Communication Language*) - an application-level communication protocol.

Remote Operations

Distributed systems - local and remote operations.

Example of remote actions:

- √ creation
- √ remote start up
- √ migration
- √ suspending
- √ restart
- √ termination, etc.

Code and Process Migration

Code migration

Code is moved between different machines.

Process migration

Entire running process is moved from one machine to another. The goal: load balancing, flexibility, reliability and availability.

To migrate a task, the system

- √ freezes the task,
- √ saves its state,
- √ transfers the saved state to a new processor,
- √ restarts the task.

There is substantial overhead involved in migrating a running task.

Models for Code Migration

Alternatives for code migration:

- √ Type of transferred information:
  1. **weak mobility** - only the code segment can be transferred (along with perhaps some initialization data),
  2. **strong mobility** - the code segment and the execution segment can be transferred.
- √ Migration initiation:
  1. **sender-initiated migration** - migration is initiated at the machine where the code currently resides or is executed.
  2. **receiver-initiated migration** - the migration is initiated by the target machine (simpler to implement).

Migration and Local Resources

The resource segment requires some special attention. Resource segment cannot always be simply transferred along with the other segments without being changed (why migration is so difficult).

Three types of process-to-resource bindings:

1. Binding by **identifier** - a process refers to a resource by its identifier (the strongest binding).
2. Binding by **value** - only the value of a resource is needed (a weaker form of binding).
3. Binding by **type** - a process indicates it needs only a resource of a specific type (the weakest type of binding).

Resource-to-machine Binding

1. **Unattached resources** can be easily moved between different machines (e.g. data files).
2. **Fastened resources** can be moved, but only at relatively high cost (e.g. local databases, Web sites).
3. **Fixed resources** are intimately bound to a specific machine or environment and cannot be moved (e.g. local devices, local communication endpoits).

Migration and Communication

Problems with the further communication.

What should be done with the future messages to the transferred process?

Solution:

- √ **message redirection** (disadvantage - the number of intermediate computers increases when the process is moved to the other machines),
- √ **message loss prevention**,
- √ **message loss recovery** (simple and fast method).

Distributed and Multiprocessor Scheduling

CPU scheduling - part of a broader class of resource allocation problems, very important in processes management.

The scheduling problem for distributed systems:
*H*ow can we execute a set of tasks **T** on a set of processors **P** subject to some set of optimizing criteria **C**?

Scheduling - goals:

- √ minimization of the expected runtime of a task set,
- √ cost minimization,
- √ communication delay minimization, etc.

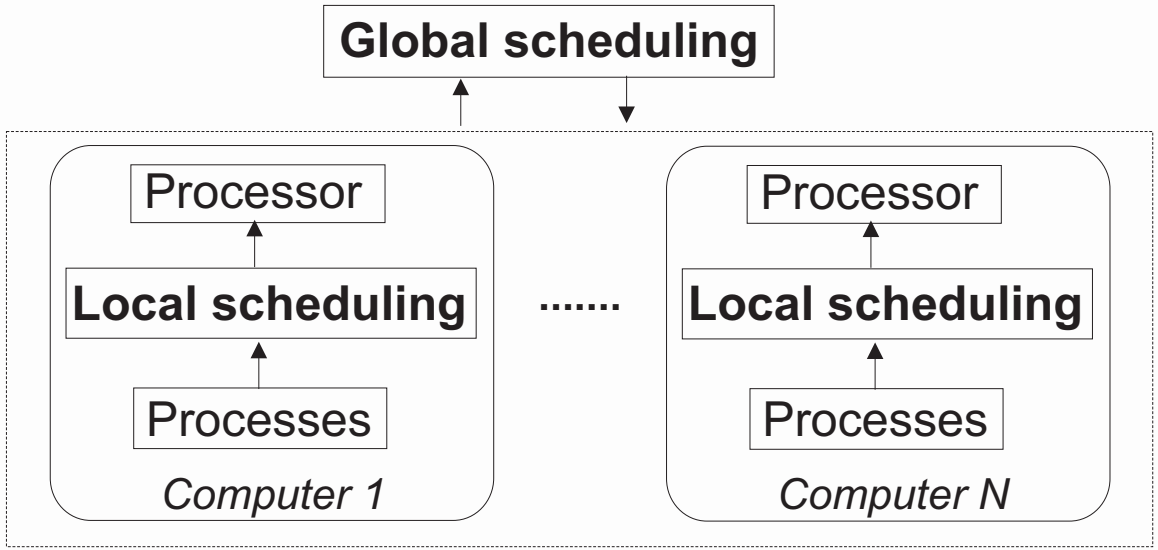
The scheduling policy usually embodies a mixture of several of these criteria.

Local and Global Scheduling

Two levels of scheduling in distributed systems:

- √ **Local scheduling**: determines which of the set of available tasks at a processor runs next on this processor.
- √ **Global scheduling** - involves assigning a task to a particular processor within the system (mapping, task placement, and matching).

Global scheduling takes places before local scheduling, although task migration, or dynamic reassignment, can change the global mapping by moving a task to a new processor.



Global Scheduling

Load sharing

One of the main uses for global scheduling is to perform load sharing between processors. Load sharing allows busy processors to offload some of their work to less busy, or even idle, processors.

Load balancing

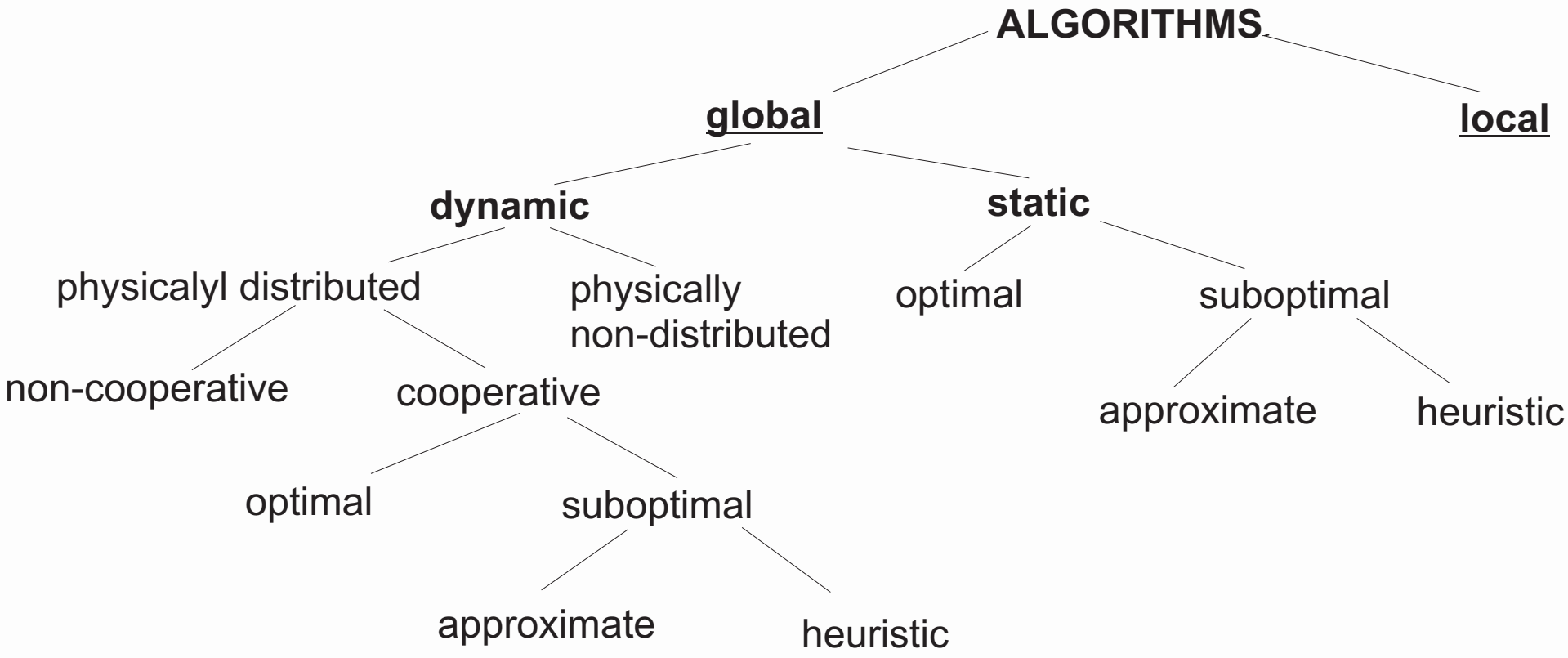
Evenly distributed load on all machines in the system - special case of load sharing.
A technique to spread work between many computers to keep the load even (balanced) across all processors in order to optimal resource utilization and decrease computing time.

Global Scheduling – Classification (1)

The two major categories of global algorithms are **static** and **dynamic**.

- √ **static** – make scheduling decisions based purely on information available at compilation time (example – the typical input to static algorithms: the machine configuration, the number of tasks and estimates of their running time).
- √ **dynamic** – take into account the current load on each processor,
- √ **adaptive** (special class of dynamic) – may change the policy based on dynamic information.

Global Scheduling – Classification (2)



A taxonomy of distributed scheduling algorithms

Global Scheduling – Classification (3)

- √ **Physically Non-distributed** – a single processor makes all decisions regarding task placement.
- √ **Non-cooperative** – individual processors make scheduling choices independent of the choices made by other processors.
- √ **Cooperative** – processors subordinate local autonomy to the achievement of a common goal.
- √ **Optimal** – if complete information describing the system and the task force is available.

Load Sharing (1)

Load sharing algorithms are based on master-slave concept.
The master:

- √ creates the workers (on idle machines)
- √ decides how much of the work load is to be distributed among the workers (the work load distribution is decided on the basis of a certain criteria).

Two types:

- √ **Sender-initiated load** – sharing occurs when busy processors try to find idle processors to offload some work,
- √ **Receiver-initiated load** – occurs when idle processors seek busy processors.

Load Sharing (2)

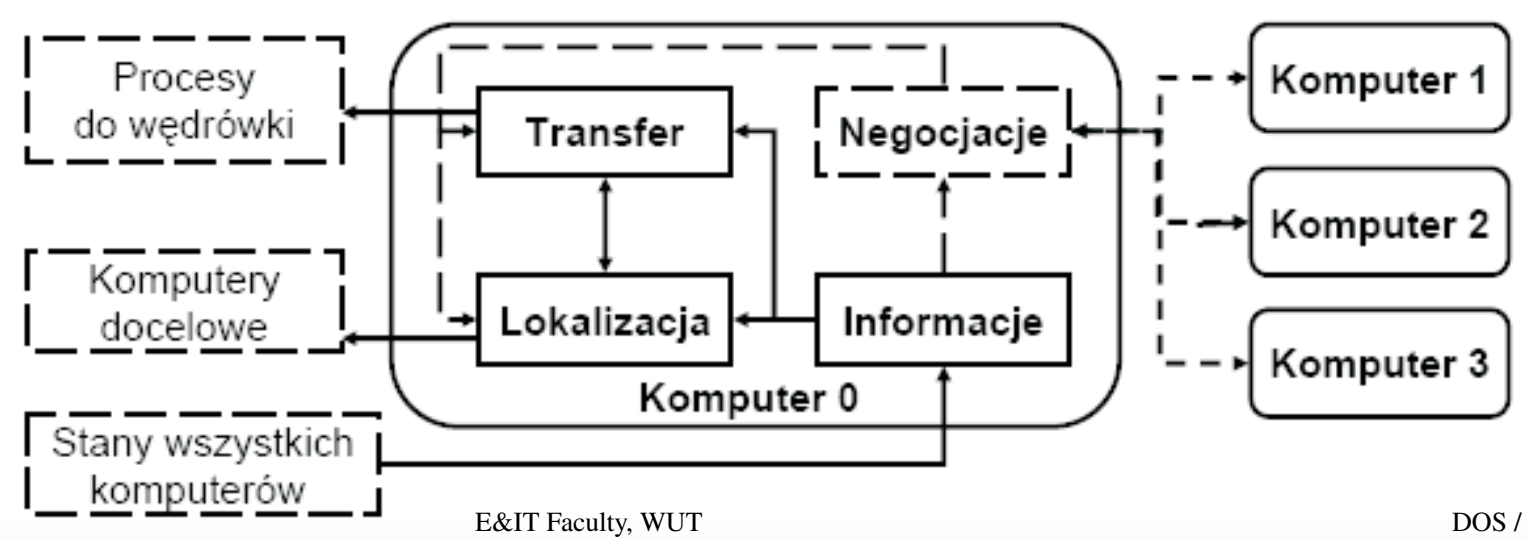
Load sharing algorithms - variants:

- √ **fixed granularity** – the task size is fixed before execution and the same task size is assigned to each worker,
- √ **variable granularity** – the task size is decreased continuously from beginning to the end of execution,
- √ **adaptive granularity** – takes current system load into consideration for work load distribution.

Load Balancing

Goal: moving processes in the distributed system in order to get optimal resource utilization and decrease computing time (usually performed by load balancers). It can also allow the service to continue even in the face of computer down time due to node failure or node maintenance.

Components: transfer component (selection of a process to be moved), location component (selection of an appropriate machine), information component (collection of the data about the state of the system), negotiations component (submitting and acceptance of offers).



Static Load Balancing

Static load balancing

– the basic method of load balancing.

The performance of the workers is determined before the execution. Then depending upon their performance the work load is distributed in the start.

The advantage: less communication between the master and the workers which improve the performance.
The drawback: it cannot adjust the runtime performance of the workers and non homogeneous nature of the application.

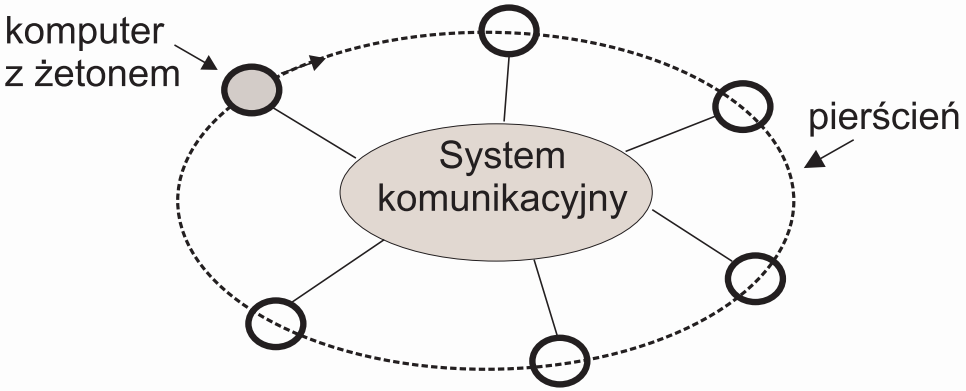
Examples: round robin, random, weight-based, etc.

Random Allocation

The requests are assigned to any workstation picked randomly among the group of workstation.
In such a case, one of the workstations may be assigned many more requests to process, while the other workstations are sitting idle.
However, on average, each workstation gets its share of the load due to the random selection.

The advantage: simple to implement.
The drawback: can lead to overloading of one server while under-utilization of others.

Round-Robin Allocation



The requests are assigned to a list of the workstations on a rotating basis. The first request is allocated to a workstation picked randomly from the group. For the subsequent requests, the *t*oken follows the circular order to redirect the request. Once a workstation is assigned a request, the workstation is moved to the end of the list. This keeps the workstations equally assigned.

The advantage: Better than random allocation because the requests are equally divided among the available servers in an orderly fashion.
The drawback: Round robin algorithm is not enough for load balancing based on processing overhead required and if the server specifications are not identical to each other in the server group.

Weighted Round-Robin Allocation

An advanced version of the round-robin that eliminates the deficiencies of the plain round robin algorithm.
One can assign a weight to each workstation in the group so that if one workstation is capable of handling twice as much load as the other, the powerful workstation gets a weight of 2. In such cases, two requests are assigned to the powerful workstation for each request assigned to the weaker one.

The advantage: takes care of the capacity of the servers in the group.
The drawback: does not consider the advanced load balancing requirements such as processing times for each individual request.

Dynamic Load Balancing

Dynamic load balancing

– adapt their load distribution to system load at runtime.

The distribution of workload is determined at runtime. A new task is assigned to the worker depending on the recent information collected (react to the current state when making transfer decisions). The algorithm may migrate running processes from one host to another if deemed beneficial.

The advantage: better performance, more flexible.
The drawback: overhead associated with communication, more computationally expensive.

Examples - various criteria: network response time, server load-based.

Dynamic Load Balancing – Steps

There are many dynamic load balancing algorithms.

Four steps that nearly all algorithms have in common:

1. Load monitoring – monitoring workstation performance
2. Synchronization – exchanging this information among workstations
3. Rebalancing criteria – calculating new distributions and making the work movement decision
4. Job migration – actual data (processes) movement

Dynamic Load Balancing – Algorithms

Bryant and Finkel algorithm combines load balancing, dynamic reassignment, and probabilistic scheduling to ensure stability under task migration.
This method uses neighbor-to-neighbor communication and forced acceptance to load balance between **pairs of machines**: overload and idle or less busy.

Barak and Shiloh algorithm
combines load balancing and probabilistic scheduling. Decision about migration is made based on the local load and the load of a machine randomly selected from a set.

Bryant and Finkel Algorithm

1. Computer $k1$ sends the request to the chosen neighbor node $k2$.
2. $k2$ rejects the request ($k1$ sends the request to the other node).
3. $k2$ accepts the request.
   The couple ($k1$, $k2$) is created. The process to move is chosen w.r.t. the criterion - **decrease the response time** q
   $q = RT_1(p)/[RT_2(p) + T_{12}(p)]$
   where: $RT_1(p)$ i $RT_2(p)$ - response time of process p executing on the machines $k1$ and $k2$, $T_{12}(p)$ - transmission time of p from the computer $k1$ to $k2$.
   The response times for each processes are calculated based on the time periods used for decision making.
4. Removing the couple or choosing another process to migration.

The algorithm is repeated until there is no possibility to improve the efficiency of any processes executed on the machine $k2$.

Barak and Shiloh Algorithm

Each process performs the following actions:

1. Updating of the local load vector $L[0, 1, \ldots, m-1]$ ($L[0]$ - local computer load, others - other machines loads).
2. The computer k is randomly chosen.
3. The half of the load vector $L_p[0, \ldots, m/2-1]$ (m - dimension of L), is sent to the k-th machine.
4. The received vector L_p is added to the own one w.r.t. the schema:

$$L[2i] := L[i] \text{ if } 1 \leqslant \text{ i } \leqslant m/2-1$$

$$L[2i+1] := L_p[i] \text{ if } 0 \leqslant \text{ i } \leqslant m/2-1$$

Barak and Shiloh Algorithm – Example