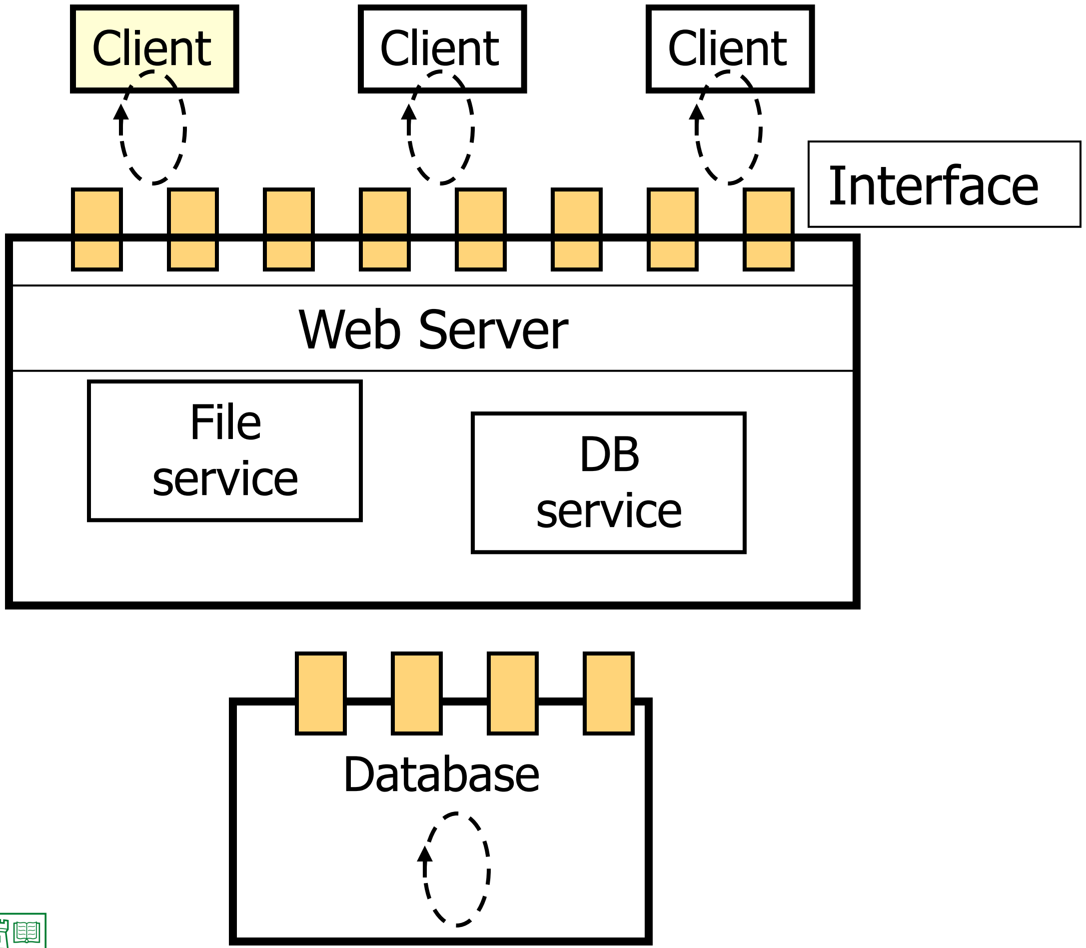
Concurrent and Distributed Systems

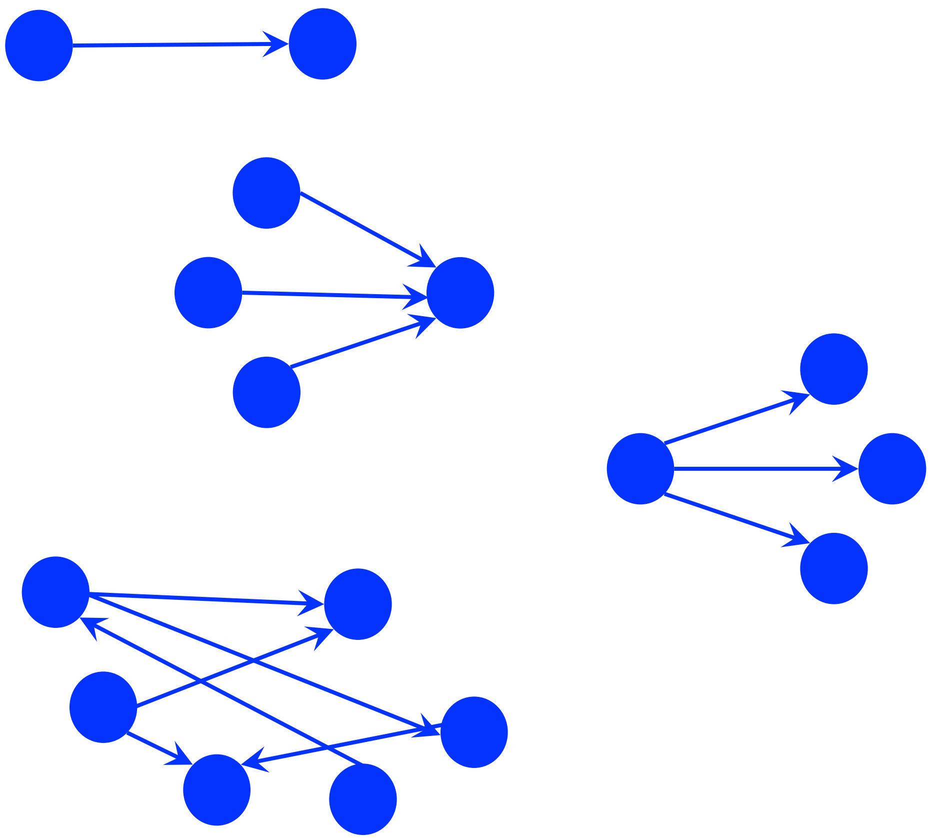
Running in Parallel - Concurrency

An Example


Client
Client
Client
Interface
Web Server
File service
DB service
Database

Process Interactions

- A general classification
  - One to one
  - Any to one
  - One to (one of) many
  - Many to Many



A Classification

- One to one
  - Appropriate in systems with static configurations of interactions between individual processes
  - Example: Pipeline in Unix commands

- Any to one
  - Multiple clients interact with a single server
  - Clients invoke a well known server
  - Server accepts requests from any client
  - Server does not know which client will interact next, waits for the next client
  - Mail server + client, Web server + client

A Classification

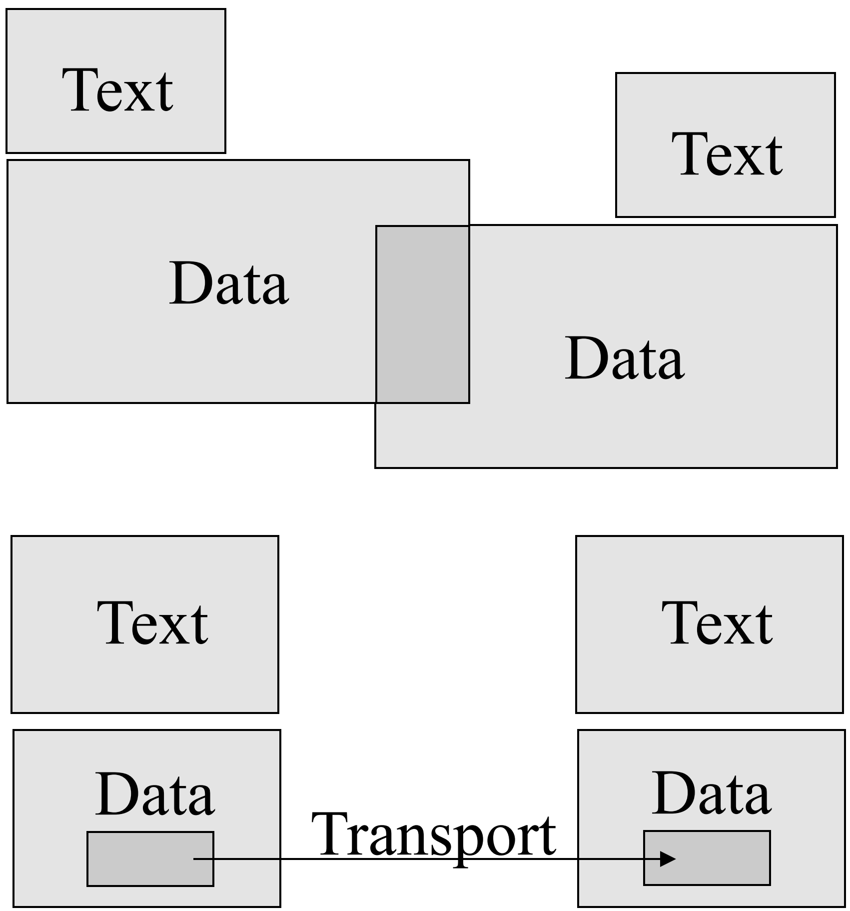
- One to many
  - Used to notify a set of interested clients
  - Broadcast (sent out to everyone)
    - Usually no record of reception of communication
    - Clients ‘listen’ for information
  - Multicast (sent out to a specific set of recipients)
    - How to identify the recipients (clients join a list – mailing list)
    - Reliable, Unreliable (like broadcast)
    - Used in fault tolerant systems

A Classification

- Any to (one of) many
  - Service offered by anonymous server processes
  - Clients requests service from any available server
  - This style usually reduces to one of the other styles
- Many to many
  - Usually implemented by shared data
  - Any number of processes can interact
  - Requires synchronisation to prevent chaos

Forms of Process Interactions

- Co-operation (shared memory)
- Communication (message passing)



Implementing IPC

- Shared Memory
  - Processes/threads involved share a common buffer pool
  - Buffer can be explicitly implemented by programmer

- Inter-Process Communication without shared memory
  - IPC has at least two operations
    - Send (message)
    - Receive(message)
  - Messages can be either fixed or variable size
  - A link between the involved processes must exist

Implementation of IPC

- Physical implementation
  - Shared memory
  - Hardware bus
  - Network

- Logical implementation of link and send() & receive():
  - Direct or indirect communication
    (naming; processes need to have a way to identify each other)
  - Synchronisation: blocking or non-blocking send/receive
  - Automatic or explicit buffering
  - Send by copy or send by reference
  - Fixed-sized or variable-sized messages
  - continued … !

Direct communication

- Processes need to explicitly name the receptionist / sender (synchronous addressing)
  - Send(P, message)
  - Receive(Q, message)
- Link is established automatically between the two parties
- Processes only need to know each other
- A link is established between exactly two processes
- Between each pair of processes there exists exactly one link
- Disadvantage: limited modularity (changing code)
- Also asynchronous addressing possible
  - Send(P, message) – send a message to process P
  - Receive(id, message) – receive a message from any process; id holds the name of the processes with which communication took place

Indirect Communication

- Messages are send to mailboxes or ports
- Mailbox is an abstract concept
  - Object into which messages can be included and removed
  - Each mailbox has its unique identification
- Processes can communicate with other processes via different mailboxes
- Communicating processes need to have shared mailboxes
  - Send(A, message) – send a message to mailbox A
  - Receive(A, message) – receive a message from mailbox A
- A link is only established if the processes share a mailbox

Indirect Communication

- A link may be established between more than two processes
- Between a pair of processes there may be any number of links represented by different mailboxes
- How are messages linked to processes?
  - Allow only links between two processes
  - Allow at most one process at a time to execute receive()
  - Allow the system to select which process will receive the message; the system may identify the receiver to the sender

Indirect Communication

- Mailboxes may be owned by
  - a user process
    - Owner process may only receive messages
    - Other processes (users) may only send messages
    - When the owner dies, the mailbox disappears, too.
    - Users need to be notified of the disappearance of a mailbox
  - the Operating System
    - Independent, not associated with any process
    - Operating system offers mechanisms for
      - Creating a new mailbox
      - Send and receive messages
      - Delete a mailbox

Synchronisation

- Message passing may be blocking or non-blocking (synchronous and asynchronous)

- Blocking send
  - Sending process is blocked until the message has been received by the receiving process or mailbox
- Non-blocking send
  - Sending process resumes operation immediately after sending the message

- Blocking receive
  - The receiving process blocks until a message has been received
- Non-blocking receive
  - The receiver retrieves a valid message or a NULL message

Buffering

- Messages exchanged always reside in a temporary queue

- Zero capacity
  - Maximum length 0 → no messages can ‘wait’ in the queue
  - Sender must block until the receiver gets the message
  - Also called a message passing system without buffering
- Bounded capacity
  - Finite length n → the queue can hold at most n messages
  - Queue not full: message is stored in the queue (either a copy or a ref); sender can continue execution without waiting
  - Queue full: sender blocks until space is available
- Unbounded capacity
  - Potentially infinite length
  - Sender never blocks

Example – Message Queue

```
import java.util.*;

public class MessageQueue{

  private Vector queue;

  public MessageQueue() {
    queue = new Vector();
  }

  public void send(Object item){
    queue.addElement(item);
  }

  public Object receive() {
    Object item;
     if (queue.size() == 0)
       return null;
     else {
       item = queue.firstElement();
       queue.removeElementAt(0);
       return item;
     }
  }

  private Vector queue;
}
```

Example – Message Queue

- Message Queue for Producer Consumer Example from lecture 3
- Buffer is unbounded and provided by Vector class
- send() and receive() are non-blocking
- Consumer needs to evaluate the result from receive()!
  - message may be NULL

When Shared-Memory is useful

- In an unprotected system where all processes and OS run in the same address space
  - Mac-OS (until 7.5)
- The language operates a simple OS
  - Ada, ML
- In systems where multithreading is provided above the OS
  - Sun LWT library

When Shared Memory is not useful

- In protected systems where processes run in separate address spaces
  - Protection and addressing are orthogonal
- Between processes on different CPUs or machines
  - However, distributed shared memory
- In systems where high flexibility is required
  - Distribute process on different machines not possible!
- In systems where process migration is desirable
  - Migration and shared memory are incompatible
  - However, distributed shared memory

Example

- Play with the alternative producer consumer!?

Summary

- Process communications (1:1, 1:m, ...)
- Shared memory – direct communication
- Synchronisation (blocking/non-blocking)
- Buffers (0, finite, …)