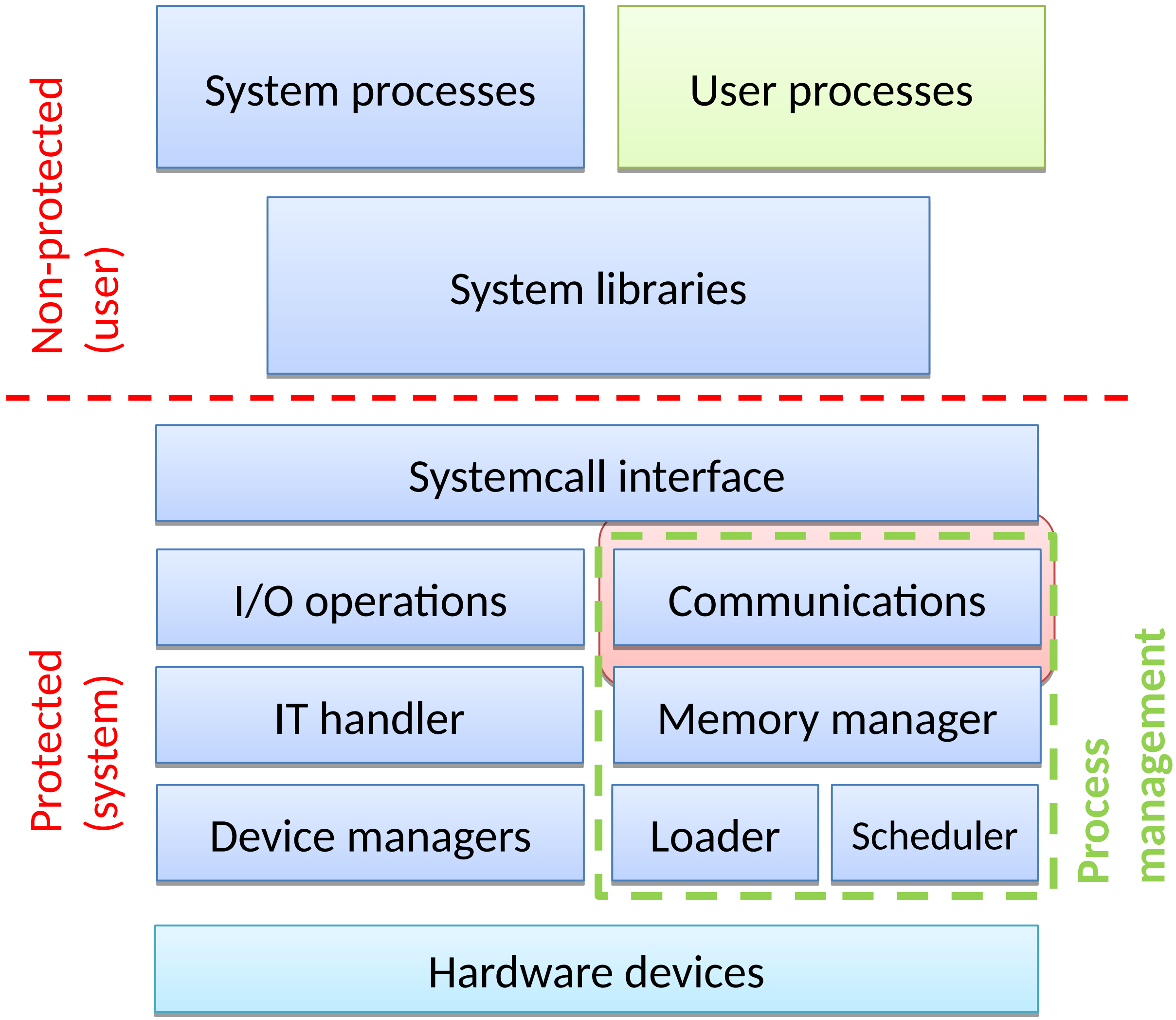
Operating Systems – Interprocess Communication

Hussein Al-Rikabi
rhussein@mit.bme.hu

Budapest University of Technology and Economics (BME)
Department of Measurement and Information Systems (MIT)

The slides of the latest lecture will be on the course page. (https://www.mit.bme.hu/eng/oktatas/targyak/vimiab00)

The main blocks of the OS and the kernel (recap)



The abstract virtual machine concept

- The main goal of the OS to support the execution of user jobs
  - Jobs are implemented by one or multiple tasks
  - Decomposed jobs may require communication between tasks
- Implementing tasks: processes and threads
  - Threads in the same process have a shared memory
    - Simple communication, but conflicts may arise
  - Processes are separated from each other
    - More complex communication
- Memory management separates the processes
  - Every process has its own memory range, which others cannot access
  - For efficient operation there can be overlapped ranges
    - Read-only pages can be used by multiple processes
    - The **Copy-On-Write** method accelerates the process creation

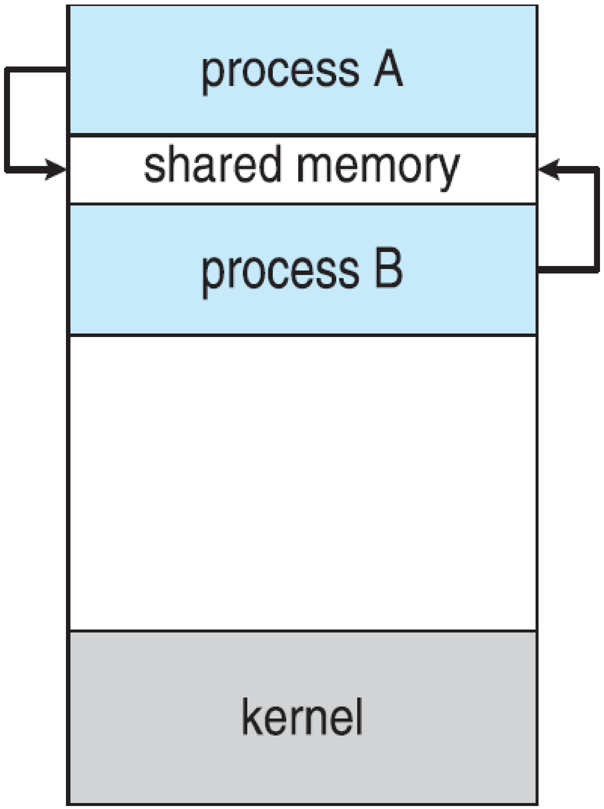
The basic forms of communication

- Through shared memory
  - The tasks memory range has a special range, which is translated to a shared physical memory
  - This shared range can be read and written by multiple tasks simultaneously
  - This method called: **PRAM model** (Pipelined RAM)
  - Competitive situations (e.g. multiple writers at the same time)
    - The execution order of these operations is not defined

- With messaging
  - No shared memory
  - The OS provides the communication mechanisms
    - Many solutions for this problem
  - Operations
    - `Send(recipient, data_pointer[, data_size]);`
    - `Receive(sender, data_pointer[, data_size]);`
  - Examples
    - Network communication, distributed systems, microkernel

The shared memory model (PRAM)

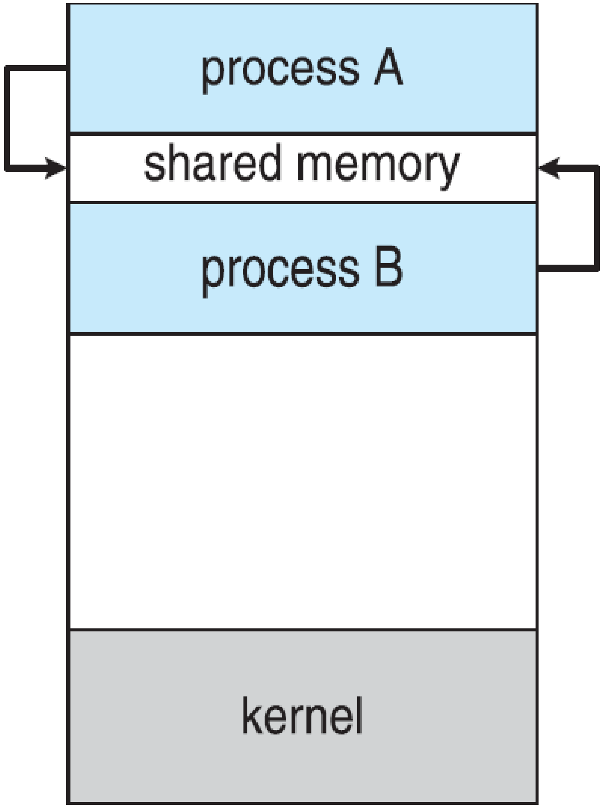
- Parallel tasks can use the shared memory range
  - They can start operations independently from each other
  - The operations may start at the same time
- Rules for the operations arrived at the same time
  - *Read-read*
    - Two simultaneous reads operations will both return the RAM contents
  - *Read-write*
    - The RAM will be written, the read op. will return the new value
  - *Write-write*
    - One of the two values will be written to the RAM



(b)

The shared memory model (PRAM)

- The effect of the rules
  - The operations has no effect to each other (no mix-up), hence the **pipelined** term: the operations are ordered in a pipeline
  - It's undetermined which op. will execute first if they arrive at the same time
  - To avoid undefined operations the tasks have to **synchronize** their operation

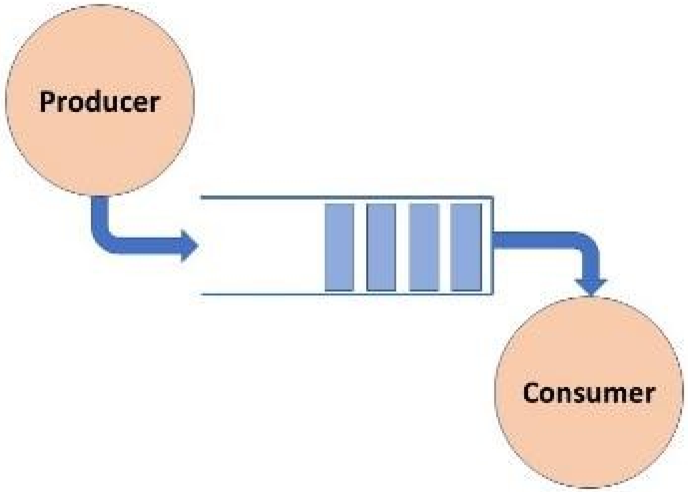


(b)

Communication methods based on the PRAM model

- Data exchange between the **threads** of the same process through global variables
  - Not supervised by the OS, the programmers design the operation
- Shared memory between **processes** (SHM)
  - The virtual memory range of the processes has a special part, which is translated to the same physical address
  - After the assignment, the usual memory operations can be used (no system calls, no additional overhead)
  - Fast method to communicate
  - The capacity of the buffer is limited (kernel defines the size)
  - Standard: POSIX Shared Memory
  - Implementations: System V UNIX, Windows (part of the kernel)
  - Typical applications: database engines

Producer-Consumer Problem

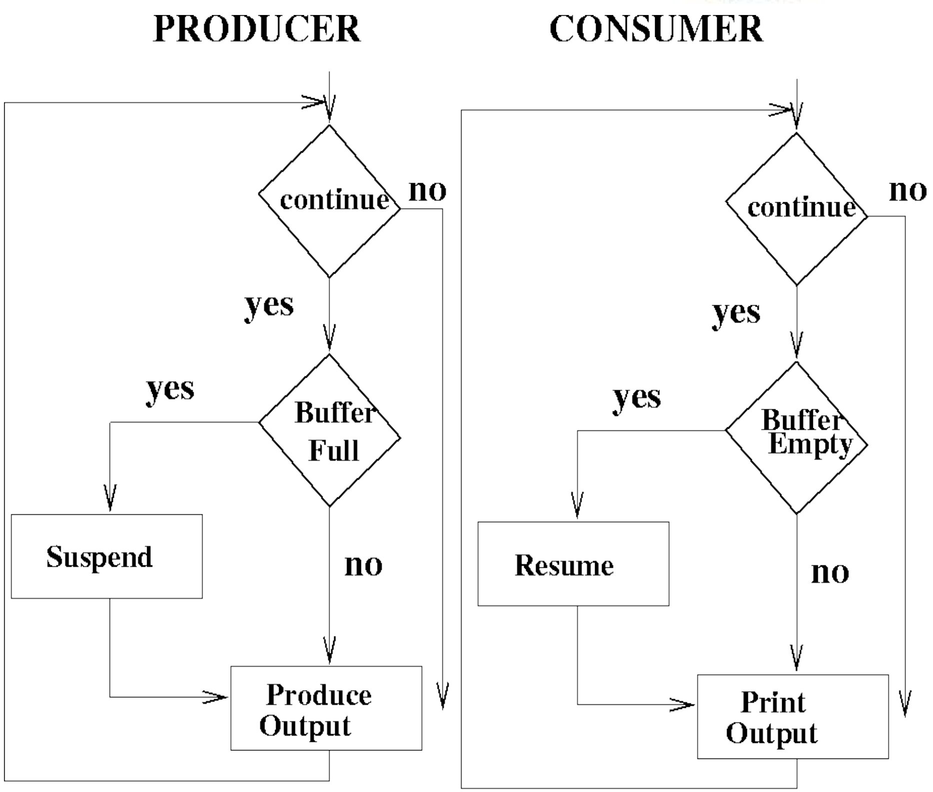


- Producer process produces information that is consumed by a consumer process.

- A **Buffer** is needed which can be:
    - unbounded buffer
    - bounded buffer

- What if Producer and Consumer tries to access the buffer simultaneously?

Synchronization



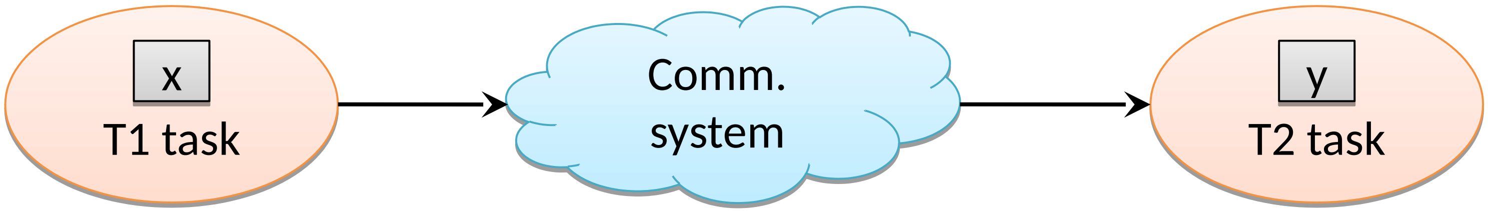
Communication with messaging

Basic questions of messaging (message-passing)

- Addressing: how can the recipient address determined?
  - Direct approach or using a mediator?
  - Single addressing or broadcasting?
  - Simultaneous receiving from multiple senders?
  - If the recipient knows the sender, can it decide whether to receive the message or not?
- Synchronization
  - Is the `send()` function blocking or non-blocking?
    - If not, how can the sender know that the message is received?
  - How the `receive()` function works?
    - What happens if the `receive()` function is called, but the date is not yet arrived?
    - Are acknowledgements mandatory, when the data is received?
- Semantics of data exchange
  - The data stored by the sender, or the receiver, or both?
- Performance, reliability
  - What throughput can be achieved? How much is the delay between send and receive?
  - How many and what size messages can be sent?
  - Who supervise (error handling) the process? What happens if an error occurs?

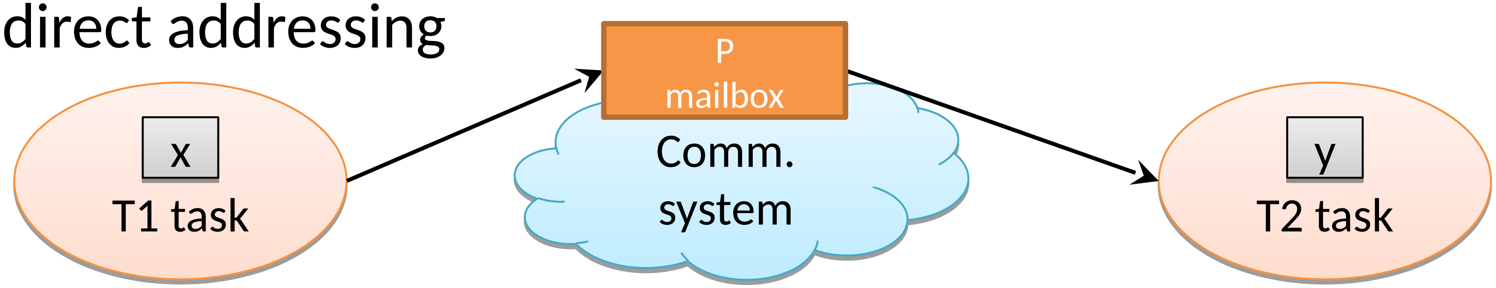
The basic methods of addressing

- Direct addressing



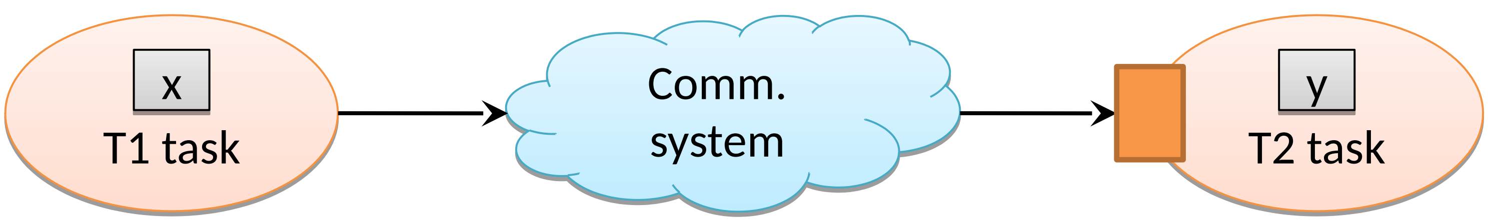

```
send(T2, x);                    receive(T1, y);
```

- Indirect addressing




```
send(P, x);                     receive(P, y);
```

- Asymmetric (direct on the sender's side) addressing




```
send(T2, x);                    receive(y);
```

- Multiple addressing (multicast, broadcast)

Synchrony

- **Synchronous data transfer (blocking)**
  - The operation blocks the task execution (waiting state)
    - Send: it returns only, when the data is received
      - If there are no mediator too long blocking is possible
    - Receive: it returns only, when the data is received
- **Asynchronous data transfer (non-blocking)**
  - The tasks are not blocked during the transfer
  - The results are not available when the function returns
  - The tasks has to be notified by other methods about the completion or the errors
  - The messages which are not received has to be stored
  - operations
    - Send: returns immediately after calling
      - The successful delivery has to be checked later
    - Receive: returns immediately with the data or with „no data received”
      - Busy waiting for a message should be avoided

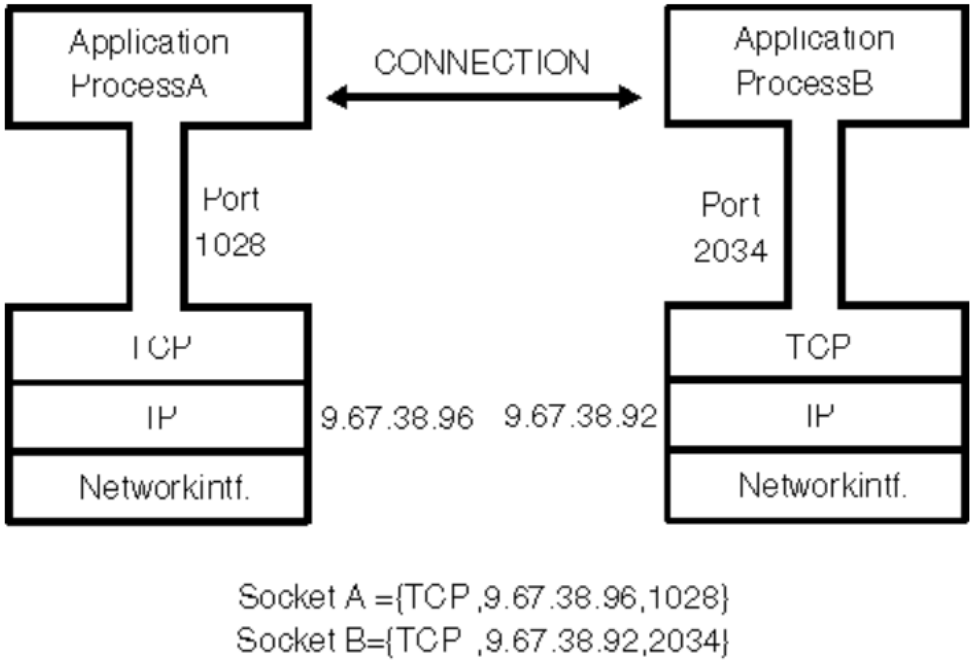
Communication in Client–Server Systems

- Socket communication
  - Asymmetric method based on the client-server model
  - It is widely used on TCP/IP bases
  - The machines identified by IP addresses, the senders and receivers identified by ports
  - The OS provides the network connectors for the communication

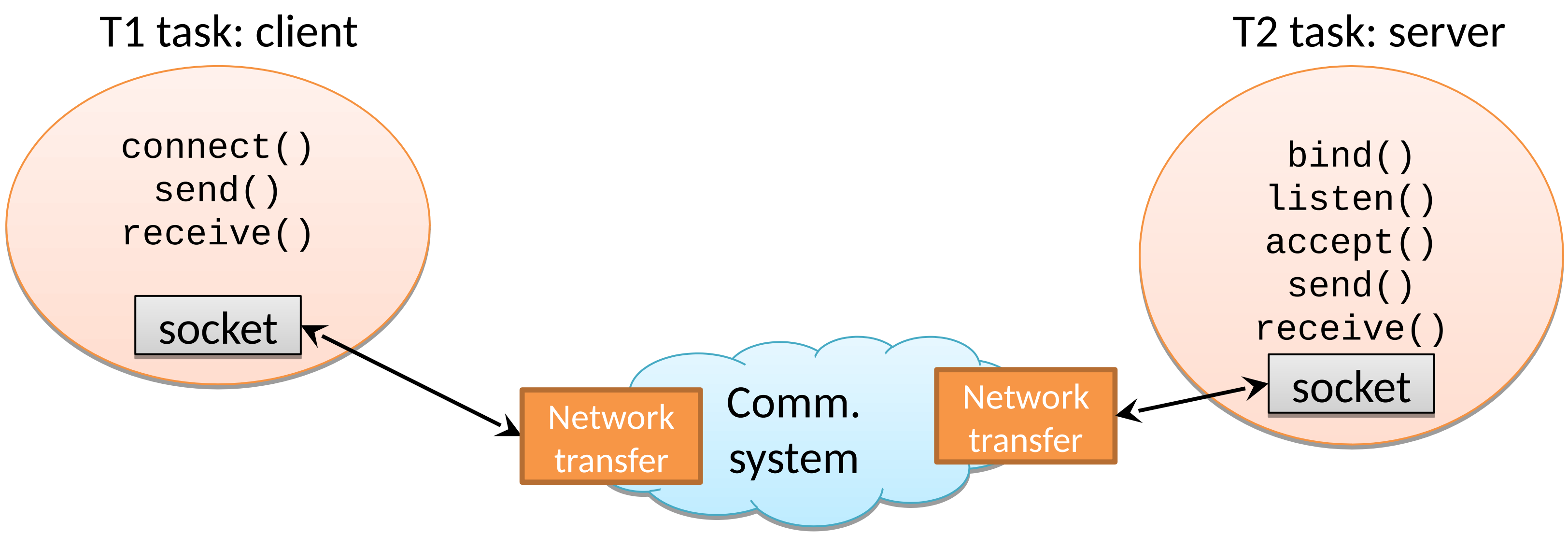
- Remote Procedure Call (RPC)
  - Asymmetric method based on the client-server model
  - Calling a function in another task
  - The sender transfers the function name and arguments and receives the return value
  - The recipient performs the operations with the provided arguments and returns the results
  - It is based on network communications, so it can be used between different machines
  - Besides the communication protocol, the data semantics also determined
    - Exact data types, structures
    - Automatic conversion may be performed between the participants
  - Higher level OS services based on RPC
  - A number of development schemes (design patterns) are based on RPC

Network socket communications

- The communication is based on network protocols and addressing
  - The socket defines the protocol being used: TCP, UDP, or IP.
  - The IP and the port.
  - The identifier of the communication endpoint in tasks



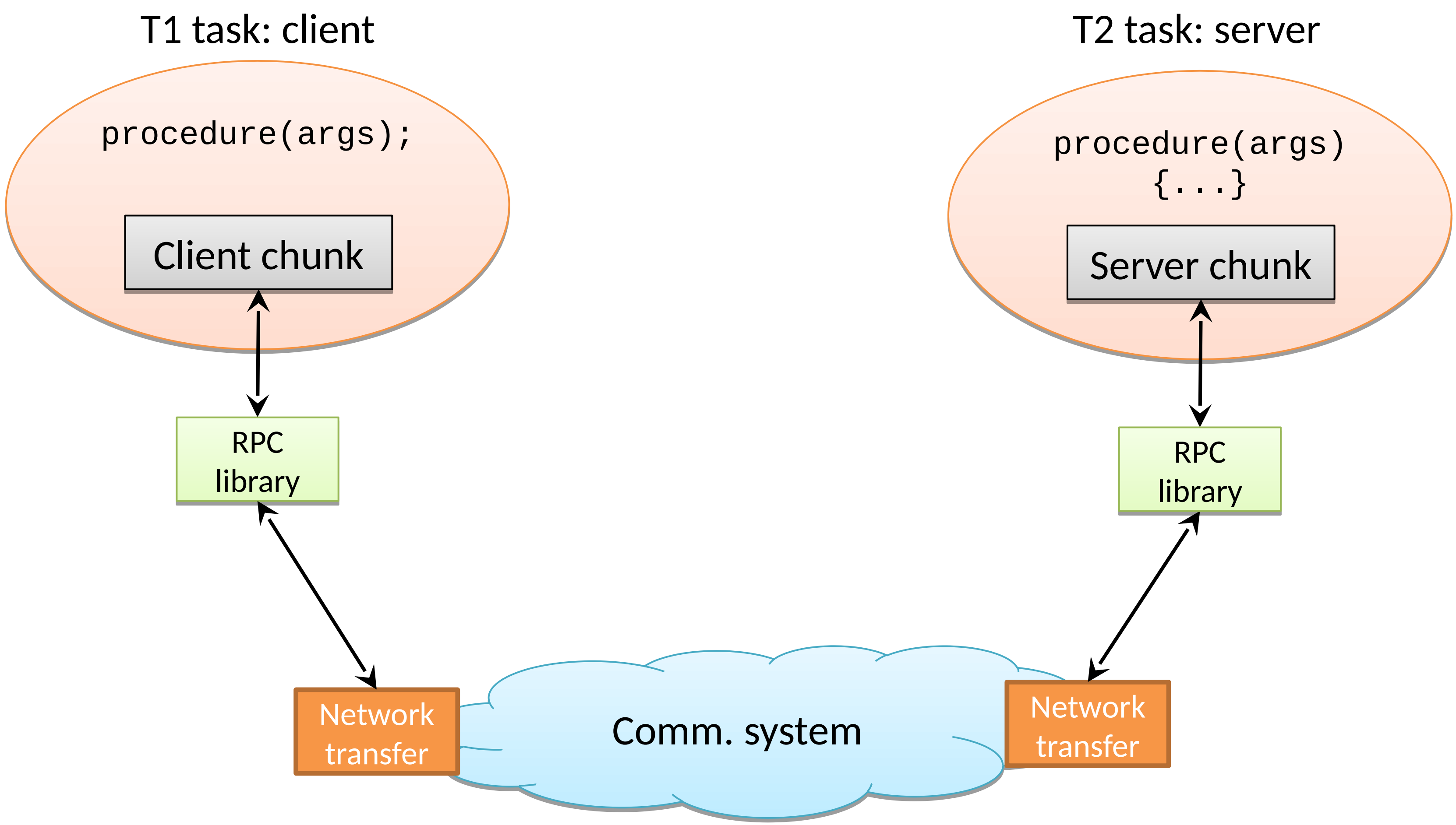
Network socket communications



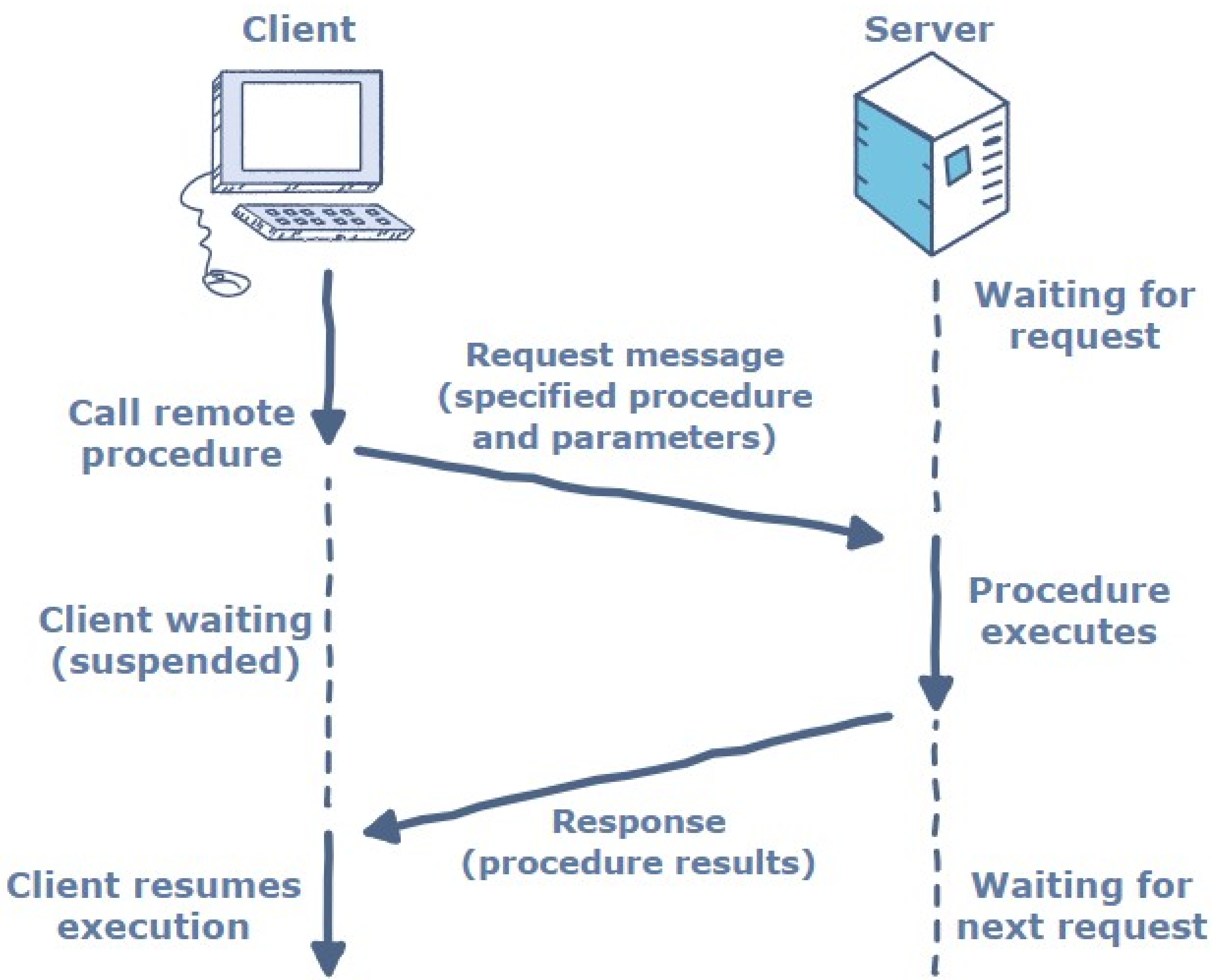
Remote Procedure Call (RPC)

- Distributed system infrastructure based on network communications
  - High level communication between processes
  - Remote function calls in different processes (even on a different machine)
  - Simple implementation due to common API-s
- Structures
  - Communication infrastructures
    - Protocols for transferring functions calls
    - Portmapper: assigning process ID-s and ports

The blocks of the RPC communication



RPC communication



A remote procedure call

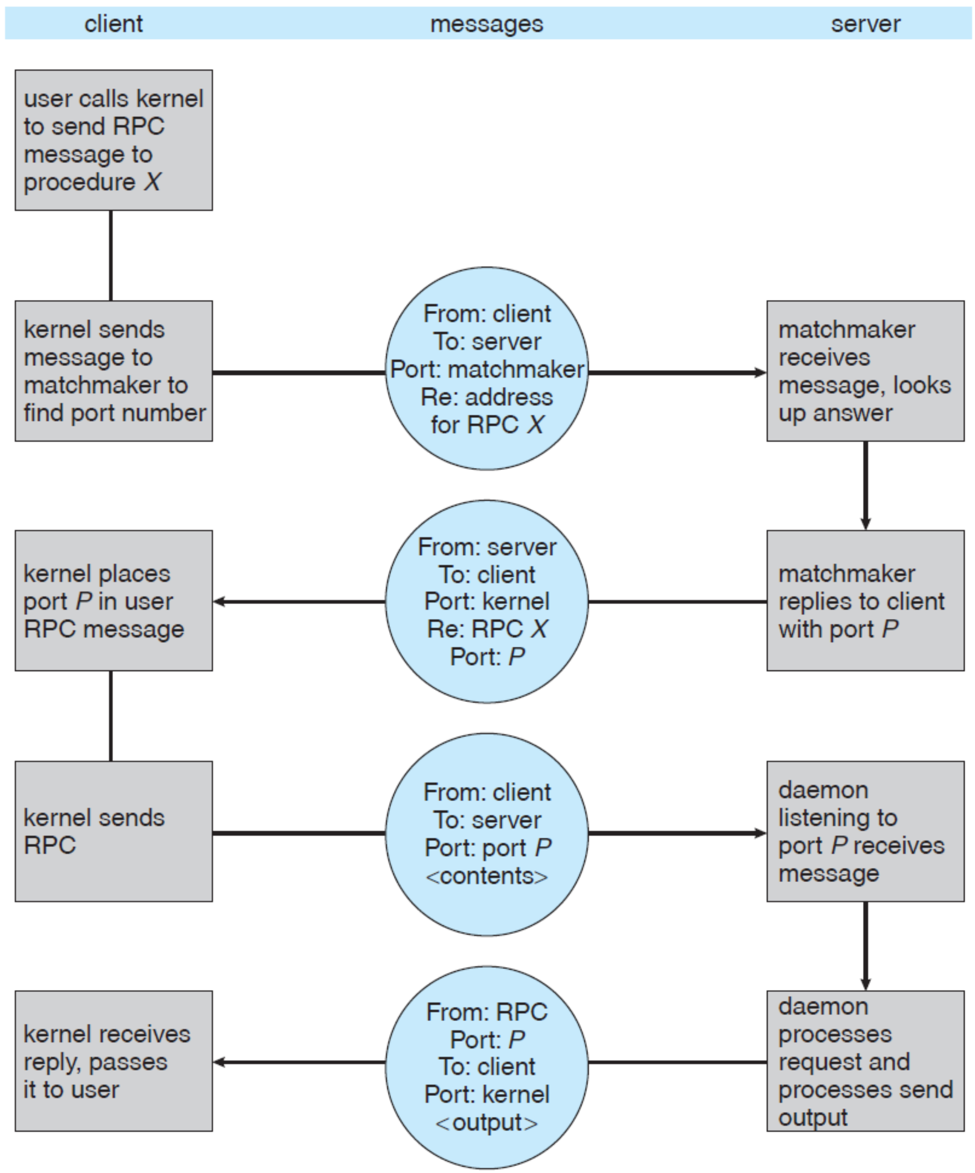


Figure 3.21 Execution of a remote procedure call (RPC).

The basic forms of communication (summary)

- **PRAM model**
  - Shared memory
  - The operations are pipelined
- Synchronous transfer operations
- No addressing
- Applications:
  - Threads between a process
  - **Shared memory between processes**
- Pro-s
  - Fast, simple
  - No overhead after initialization
- Con-s
  - R-W and W-W conflicts
  - Synchronization between participants is necessary
  - Limited capacity

- **Messaging systems**
  - It uses a communication system
  - Send and receive operations
  - Parallel operations are ordered
- Synchronous or asynchronous operation
- Addressing: direct, indirect, multi
- Applications
  - Mailbox
  - Pipeline
  - Message queue
  - **Network socket**
  - **Remote Procedure Call**
- Pro-s
  - Widely available
  - Between different machines
- Con-s
  - Communication errors has to managed
  - Slower

Interprocess communication in practice

POSIX Shared Memory Example

- Several IPC mechanisms are available for POSIX systems, including **shared memory** and message passing.
- A process creates a shared-memory --- shmget()

  - segment id = **shmget**(identifier, size, S_IRUSR | S_IWUSR);

  - identifier(key) - if set to IPC_PRIVATE, new shared-memory segment is created

  - size - the third parameter defines the **Mode** , read or write or both
- A successful call to **shmget**() returns an integer identifier, any task wants to access this memory, must specify this identifier.
- So a process can access it using **shmat(shmid, shmaddr , shmflg)**

  - shmaddr = the second parameter Points to the desired address of the shared memory segment

  - shmflg = Specifies a set of flags that indicate the specific shared memory conditions and options to implement.
- Also the share memory can be detached and removed, and there are many other commands to utilize it. See the code :

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the identifier for the shared memory segment */
int segment_id;
/* a pointer to the shared memory segment */
char *shared_memory;
/* the size (in bytes) of the shared memory segment */
const int size = 4096;

    /* allocate a shared memory segment */
    segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);

    /* attach the shared memory segment */
    shared_memory = (char *) shmat(segment_id, NULL, 0);

    /* write a message to the shared memory segment */
    sprintf(shared_memory, "Hi there!");

    /* now print out the string from shared memory */
    printf("*%s\n", shared_memory);

    /* now detach the shared memory segment */
    shmdt(shared_memory);

    /* now remove the shared memory segment */
    shmctl(segment_id, IPC_RMID, NULL);

    return 0;
}
```

Network Sockets message passing Example:

```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         // now listen for connections
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            // write the Date to the socket
            pout.println(new java.util.Date().toString());

            // close the socket and resume
            // listening for connections
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Importing required libraries

Class : **server** or clint ?

Creating Socket with Port No.

Listening, waiting a clint

Writing data to the socket

Close socket and wait again

Some lines can be added here
to handle errors and exceptions

```
import java.net.*;
import java.io.*;

public class DateClient
{
   public static void main(String[] args) {
      try {
         //make connection to server socket
         Socket sock = new Socket("127.0.0.1",6013);

         InputStream in = sock.getInputStream();
         BufferedReader bin = new
            BufferedReader(new InputStreamReader(in));

         // read the date from the socket
         String line;
         while ( (line = bin.readLine()) != null)
            System.out.println(line);

         // close the socket connection
         sock.close();
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

creates a Socket and requests a connection with the server

client define method of reading from socket, normal stream I/O statements

client can read from the socket

Socket is closed

End of Lecture