Collaboration of Tasks

Tamás Kovácsházy, PhD

13rd Topic

Inter Process Communication with Message Passing

Méréstechnika és
Információs Rendszerek
Tanszék

Looking back, communication solutions

- Using shared memory(RAM or PRAM model):
  - Among threads running in the context of a process (shared memory of the process)
- Messages:
  - No shared memory
    - Among processes running inside an operating system
    - Distributed system (network communication)
  - Microkernel based operating system
- Inter Process Communication, IPC

Messages

- Different from the same word used in computer networks
  - We consider a more generic notion of message
- Message passing
- For example:
  - System call
  - TCP/IP connection (TCP) or message (UDP) for internal (localhost) or external communication (among machines)
- Most cases they are implemented as OS API function/method calls resulting a system call
- The operating system implements them by its services

Some notes

- Semaphore, Critical section object, and Mutex are also implemented by the OS and handled by system calls
  - Threads running in the context of a process communicate using shared memory (fast, low resource utilization)
  - Mutual exclusion and synchronization are solved by messages (using system calls).
  - It has some overhead:
    - Experiments: Lockless programming, transactional memory etc.
    - There is no good solution, but we can pick a better one than the other (Churchill is right).
    - The good solution is application and software architecture dependent

Properties of message passing

- Compared to shared memory:
  - Higher delay
  - Lower bandwidth
  - Unreliable communication channel
    - Shared memory is reliable with the propability of 1
      - RAM, PRAM model
    - It is not true even for system calls in an operating system
      - System overload may happen!
    - Using a computer network is unreliable by definition
      - Random and intentional errors
      - Intentional errors are the worse, because they target the vulnerabilities of the system directly

Addressing messages

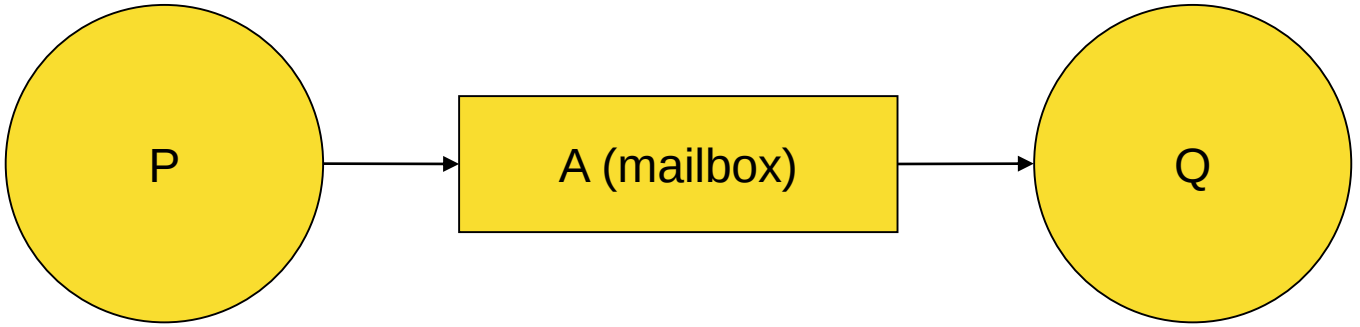
- Computer networks...
- A given process (unicast address).
- All processes (broadcast address).
  - E.g. power management messages
    - Standby, Hibernate, PowerOff, etc.
- A group of processes (multicast address).
- One process from a group of processes (anycast address).
  - E.g. a process that is going to serve the request from the processes that can serve the request because they run a specific service

Direct communication

- Symmetric message based communication
  - send(P, message)
  - receive(Q, message)
  - P, Q are process identifiers
  - Q, the sender, is specified when receive() is called!
  - Message is a data structure containing the information to be sent
- Asymmetric message based communication
  - send(P, message)
  - receive(id, message)
  - P is the process identifier of the recipient
  - The id identifies the sender. The receiver receives from anybody!
    - In other words, id is a return value…
  - Message is a data structure containing the information to be sent
- There is a direct reference in the code to the receiver or the sender (symmetric)
  - Not a good idea...
  - Makes everything too complex.

Indirect communication

- There is a entity in between the communicating parties
  - Proxy design pattern
- This entity can be: Mailbox, MesssageQueue, Port, etc.
- Interface: constructor and destructor plus
  - send(A, message)
  - receive(A, message)
- „A” is the identifier of the entity
  - In distributed systems it may be in part the identifier of the node (identifier of a computer)



Additional properties

- Minimum one sender
- Minimum one receiver
  - After the message is received by a process it is deleted (single read)
  - The message can be read multiple times (explicit delete is needed to remove it)
    - SystemV Shared Memory in UNIX
- Owner can be:
  - Operating system
    - It exists independently of the processes which use it
  - A process (It is located in the memory area of the process)
    - It exists with that process

Blocking

- Non-blocking call = asynchronous call
  - Results and side effects does not available when the call returns (They may not have happened at all)
  - Only execution of the real functionality is started after returning from the call
  - Handling of return values, results, and side effects needs some other solutions from the caller:
    - E.g. Events, signals, callback fuctions, etc. are used
- Blocking call = synchronous call
  - Results and side effects are available on the return of the call (they happened).
  - Handling of return values is simple...

Blocking on the sender side

- Blocking send():
  - The send() call does not return until the message is received (direct communication) or stored into the communication entity (indirect communication)
  - How we handle errors?
    - The send() call returns with errors
- Non-blocking send():
  - After sending the message locally, it returns (does not wait for delivery or positive acknowledgements)
  - A callback function or signal handling, etc.

Blocking on the receiver side

- Blocking receive():
  - The receive() call does not return until something is received (maybe with a timeout)
  - Classic example: TCP/UDP socket listen().
- Non-blocking receive():
  - The receive() call returns immediately with some data
    - If there is a message receive, it return with that
    - If there is no message it is told (pl. empty message with 0 length, null reference, error code, etc.).
    - If there is no message and non-blocking receive is called in an infinite cycle it results busy waiting (eats the CPU).

Implementations 1.

- Mailbox:
  - Indirect communication
  - A single message is stored or multiple one, but the maximum number of messages is specified
  - The mailbox is handled on the OS level
- MessageQueue:
  - Indirect communication
  - Infinite number of messages can be stored
    - Of course, system resources limit the number
  - Message based middlewares
    - MSMQ, IBM's WebSphere MQ, Oracle Advanced Queuing (AQ), JBoss Messaging, Apache Qpid.
- Embedded operating systems typically support Mailbox/MessageQueue type solutions even to communicate among threads
  - Simple, problem free solution

Implementations 2.

- TCP/IP TCP or UDP port:
  - Direct communication
  - Socket interface
  - Localhost (127.0.0.1/8) can be used inside the machine
  - Low level solution, several middlewares are based on it:
    - Remote Procedure Call, RPC
    - Remote method Invocation:
      - CORBA (Common Request Broker Architecture),
      - JAVA RMI (Remote Method Invocation),
      - DCOM/.NET Remoting,
      - SOAP (Simple Object Access Protocol).
    - Message based middlewares (we have already talked about them)

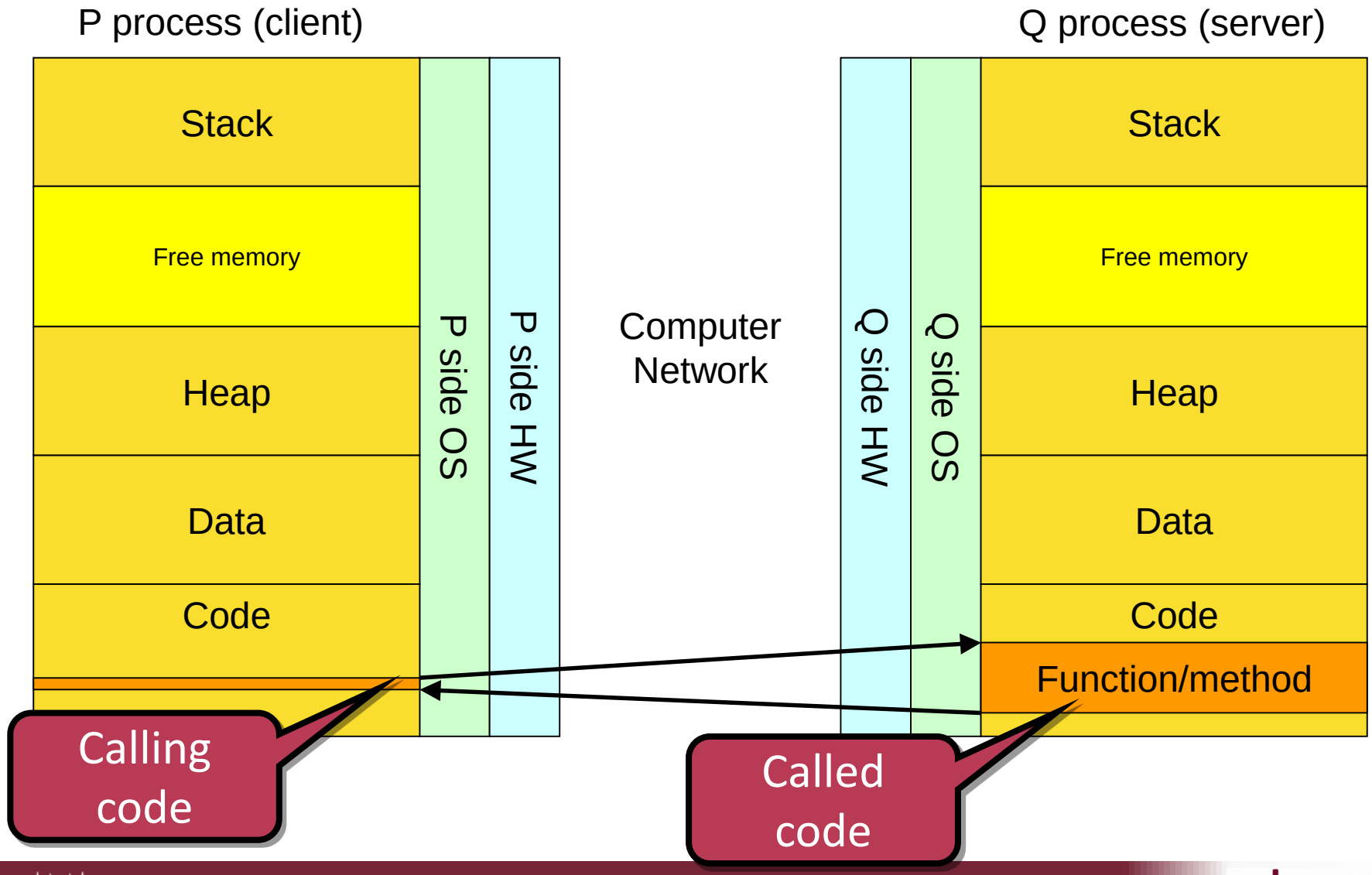
Implementations 3.

- Various pipes and streams:
  - Typically direct but can be indirect (named pipe)
  - E.g. UNIX pipe, Windows Named Pipe, RTLinux FIFO
- System V Shared Memory (UNIX, Linux)
  - Direct
  - Memory based interface using the special features of the MMU
  - In the UNIX lectures is will be introduced

Remote Procedure Call

- It is introduced in detail:
  - It is used even now, primarily for OO, so it is called Remote Method Invocation in this context
  - Very illuminating to see how it works...
- Remote Procedure Call, RPC:
  - Calling a function located in the memory of an other process from the calling process using messages
  - The caller blocks while waiting for the answer
  - The called function runs in a thread of the called process

Architecture of RPC



How the programmer sees RPC

- Practically using a remote procedure is like calling a local one (actually it is calling a local one).
  - The function is available as a stub function in a program library (prepared by the RPC development system)
  - The programmer needs to nothing about where the actual function will be executed (RPC hides the details)
- Implementation of the actual function is similar than writing a local function
  - The programmer gets an interface definition (prepared by the RPC development system) and implements the functionality
  - The programmer needs to nothing about from where the actual function is called (RPC hides the details)

RPC in operation 1.

- The parameters and return value of the call has types
  - Structured message is sent
  - Platform independence is realized by the Operating System and the development system (compiler or interpreter)
    - All sent data is converted to standard formats e.g. Binary Encoding Rules (BER), XML, etc.
- The client program calls a normal local function
  - The local function is an automatically generated stub function handling the RPC
  - The stub hides the details of communication from the programmer using it
  - We do not talk about that how the server is found
    - Let us assume that it is known…

RPC in operation 2.

- The responsibilities of the client side stub are during the call
  - Packing the parameters of the call into a platform independent form and putting it into a message (or messages), and sending it to the server
  - To implement this it uses the services of the operating system and the computer network
- On the server side the RPC service gets the messages containing the parameters of the call (including function name)
  - It converts the parameters to a local form
  - It calls the local function
  - The return values are converted back to standard form
  - The standard form return values are sent back to the client in a message (or messages)

RPC in operation 3.

- The client side stub receives the messages with the return values in standard form
  - It converts the standard form return values to the local form
  - It returns with the return values converted back to local form from the stub into the calling program
- The client thread calling the remote code
  - Waits for an event caused by the incoming message containing the return values
- The server thread waits for incoming calls, and if there is any, it runs (executes the calls)
  - It blocks on listening for incoming messages