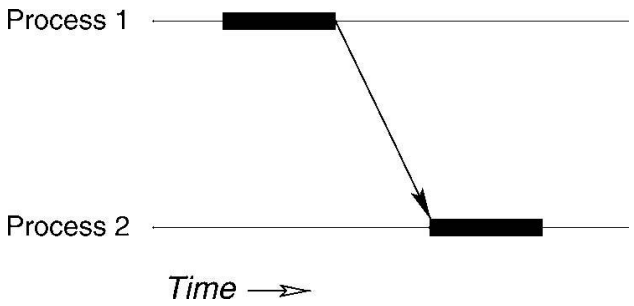
Distributed Systems – Theory
3. Communication in DS

DS vs uniprocessor system | State

- Interprocess communication:
  - Uniprocessor:
    - interprocess communication implicitly assumes the existence of shared memory
    - E.g. synchronization: the semaphare is shared
  - DS: no shared memory, instead messages
- Processes: active components with a state and a behavior
  - State: consists of the data that is managed by the process
    - Active state or passive state
  - Behavior: the implementation of the applications logic
- Message: sequence of bytes that are transported between 2 processes via a communications medium

Communication between 2 processes

- One is the sender, another the receiver
- Space time diagram:
  - Active state:
    - thick black line
    - carry out calculations



- Patterns of message flow:
  - Message-oriented communication - No wait for reply
  - Request oriented communication - With reply from the receiver

Synchronicity of communication mechanism

- describes the time separation between sender and receiver
- *synchronous communication:*
  - the sender is passive during the communications process until the message has arrived at the receiver
- *asynchronous communication:*
  - the sender remains active after a message has been sent
  - more quickly! Better at supporting the parallelism of processes
  - need a DS capable of buffering messages

Classification of communication mechanisms

Communic. pattern	*Level of synchronization*	
	Asynchronous	*Synchronous*
Message-oriented	no-wait-send	rendezvous
Request-oriented	remote service invocation	remote procedure call

Examples:

- RPC: sender sends a request to the receiver and is passive until the receiver delivers the results of the request
- RSI: the sender remains active while the receiver is processing the request
- Datagram services: asynchronous message-oriented communication; the sender transmits a message to the receiver without waiting for a reply or changing to the passive state
- Rendezvous: establish a common (logical) time between sender and receiver

Client-server model

- Advantages:
  - Simplicity – no connection has to be established before
  - One or more clients have access to the service
  - Any kind of communication from previous table
  - Easiness to migrate existing appls based on procedural programming
  - Potential for concurrency
- Disadvantages:
  - restriction to procedural programming paradigms excludes other approaches such as functional or declarative programming
  - transparency can no longer be achieved in the case of radical system failure
  - problems caused by concurrency when processes need to be synchronized

Library procedures for communication services

- *send(dest,&mptr)*
  - sends the message pointed to by *mptr* to a process identified by *dest*
  - causes the caller to be blocked until the message has been sent.
- *receive(addr,&mptr)*.
  - causes the caller to be blocked until a message arrives;
  - when one does, the message is copied to the buffer pointed to by *mptr* and the caller is unblocked.
  - the *addr* parameter specifies the address to which the receiver is listening.
- Many variants of these two procedures and their parameters are possible.

Blocking primitives

- Above message-passing primitives are called blocking primitives
- Sometimes called synchronous primitives
- when a process calls send
  - it specifies a destination and a buffer to send to that destination.
  - while the message is being sent, the sending process is blocked suspended.
  - the instruction following the call to send is not executed until the message has been completely sent.
- a call to receive
  - does not return control until a message has actually been received and put in the message buffer pointed to by the parameter.
  - the process remains suspended in receive until a message arrives, even if it takes hours.
  - In some systems, the receiver can specify from whom it wishes to receive, in which case it remains blocked until a message from that sender arrives.

Non-blocking primitives

- some-times called asynchronous primitives
- If send is nonblocking, it returns control to the caller immediately, before the message is sent.
- The advantage of this scheme is that the sending process can continue computing in parallel with the message transmission, instead of having the CPU go idle.
- The choice between blocking and nonblocking primitives is normally made by the system designers

Problems with non-blocking primitives

- the sender cannot modify the message buffer until the message has been sent.
- the sending process has no idea of when the transmission is done, so it never knows when it is safe to reuse the buffer.
- Solutions:
  - copy the message to an internal buffer &allow the process to continue.
    - Looks like a blocking call: as soon as it gets control back, it is free to reuse the buffer.
    - But the message will not yet have been sent, but the sender is not hindered by this fact.
    - Disadvantage of this method is that every outgoing message has to be copied from user space to kernel space.
  - interrupt the sender when the message has been sent to inform it that the buffer is once again available.
    - user-level interrupts make programming tricky, difficult, and subject to race conditions, which makes them irreproducible.
    - method is highly efficient and allows the most parallelism,
    - disadvantage: programs based on interrupts are difficult to write correctly and nearly impossible to debug when they are wrong.

Views

- OS design view:
  - difference between a synch.& asynch. primitive: the sender can reuse or not the message buffer immediately after getting control back
- Programming language designer:
  - synchr: the sender is blocked until the receiver has accepted the message and the acknowledgement has gotten back to the sender
  - everything else is asynchronous in this view.
  - if the sender gets control back before the message has been copied or sent, the primitive is asynchronous.
  - when the sender is blocked until the receiver has acknowledged the message, we have a synchronous primitive.

Nonblocking receive

- returns control almost immediately.
- how does the caller know when the operation has completed?
  - provide an explicit wait primitive that allows the receiver to block when it wants to
  - or provide a test primitive to allow the receiver to poll the kernel to check on the status
    - Example: conditional-receive, which either gets a message or signals failure, but in any event returns immediately, or within some interval.

Timeouts

- In a system in which send calls block, if there is no reply, the sender will block forever.
- To prevent this situation, in some systems the caller may specify a time interval within which it expects a reply.
- If none arrives in that interval, the send call terminates with an error status.