Message Passing

ARCOS Group

Distributed Systems

Bachelor In Informatics Engineering

Universidad Carlos III de Madrid

Basic concepts

- A mechanism for the communication and synchronization of processes executing on different machines
- “shared nothing” systems
- Defined as:
  - Set of processes having only local memory
    - Message passing different from a “call” since message contains content, not its address
  - Processes exchange messages – they must have a communication link
  - The transfer of data between processes requires cooperation (send has matching receive)
- Primitives:
  - send(dest, message)
  - receive(src, message)

- For connection-oriented communication also need to connect and disconnect

Distributed application paradigms - abstraction levels

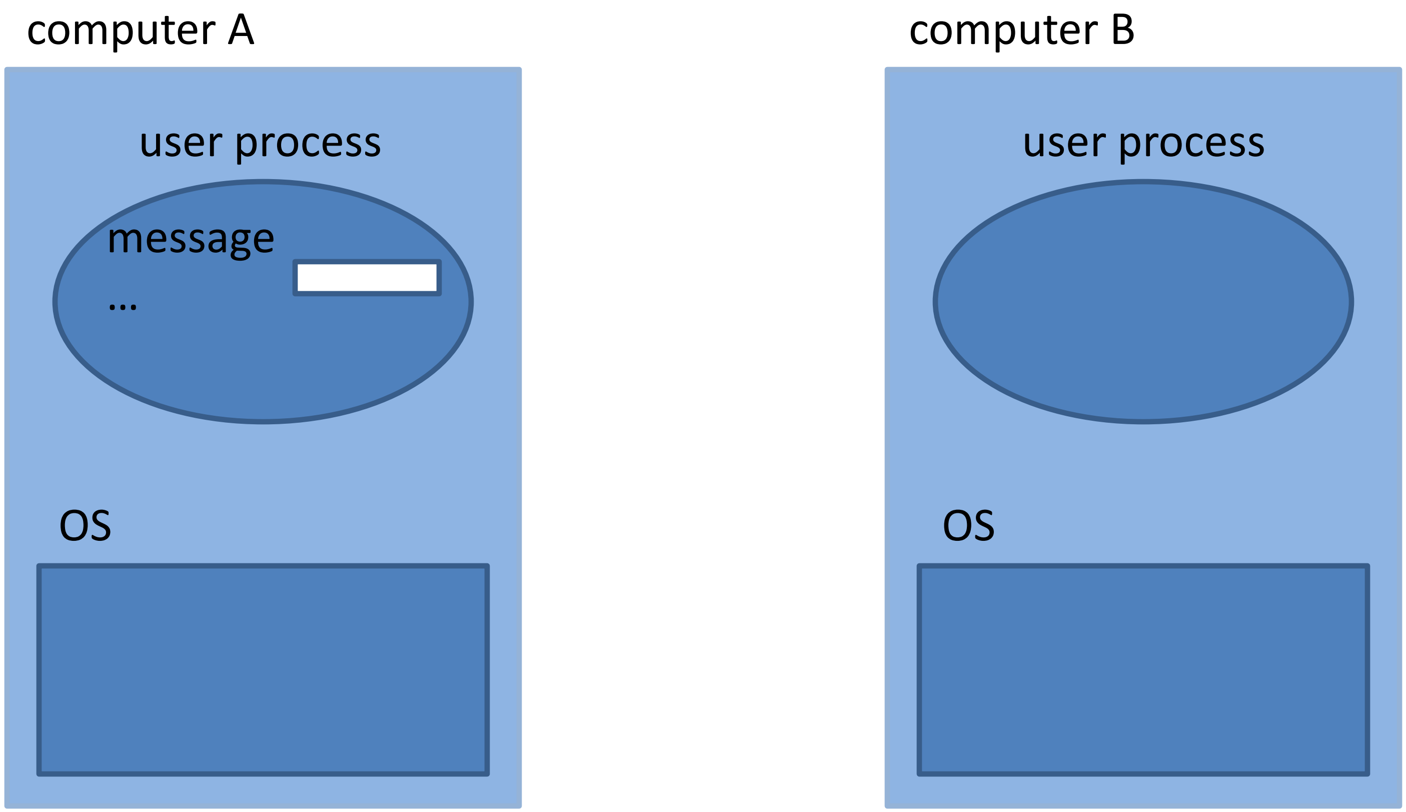
Collaborative apps, object space

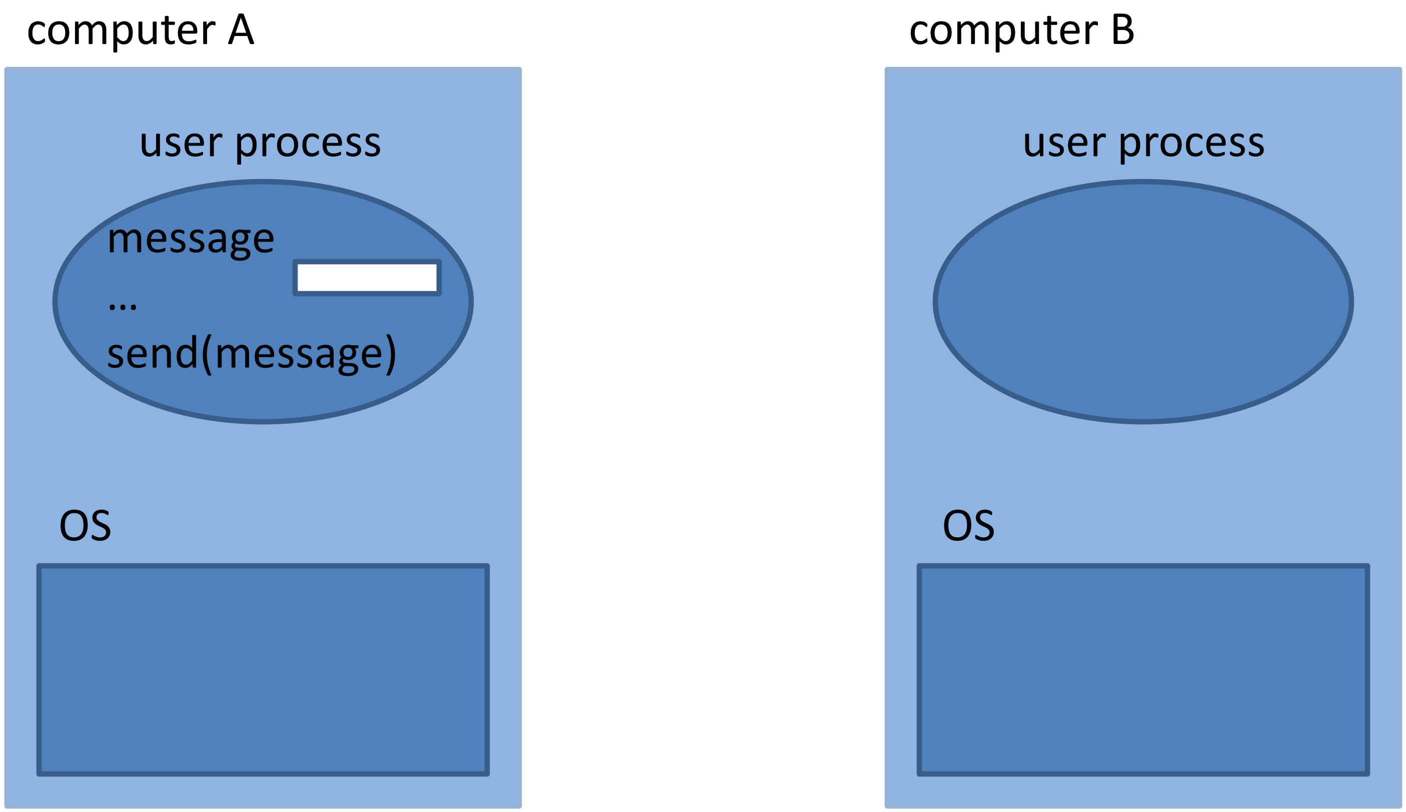
Services, object request broker, mobile agents

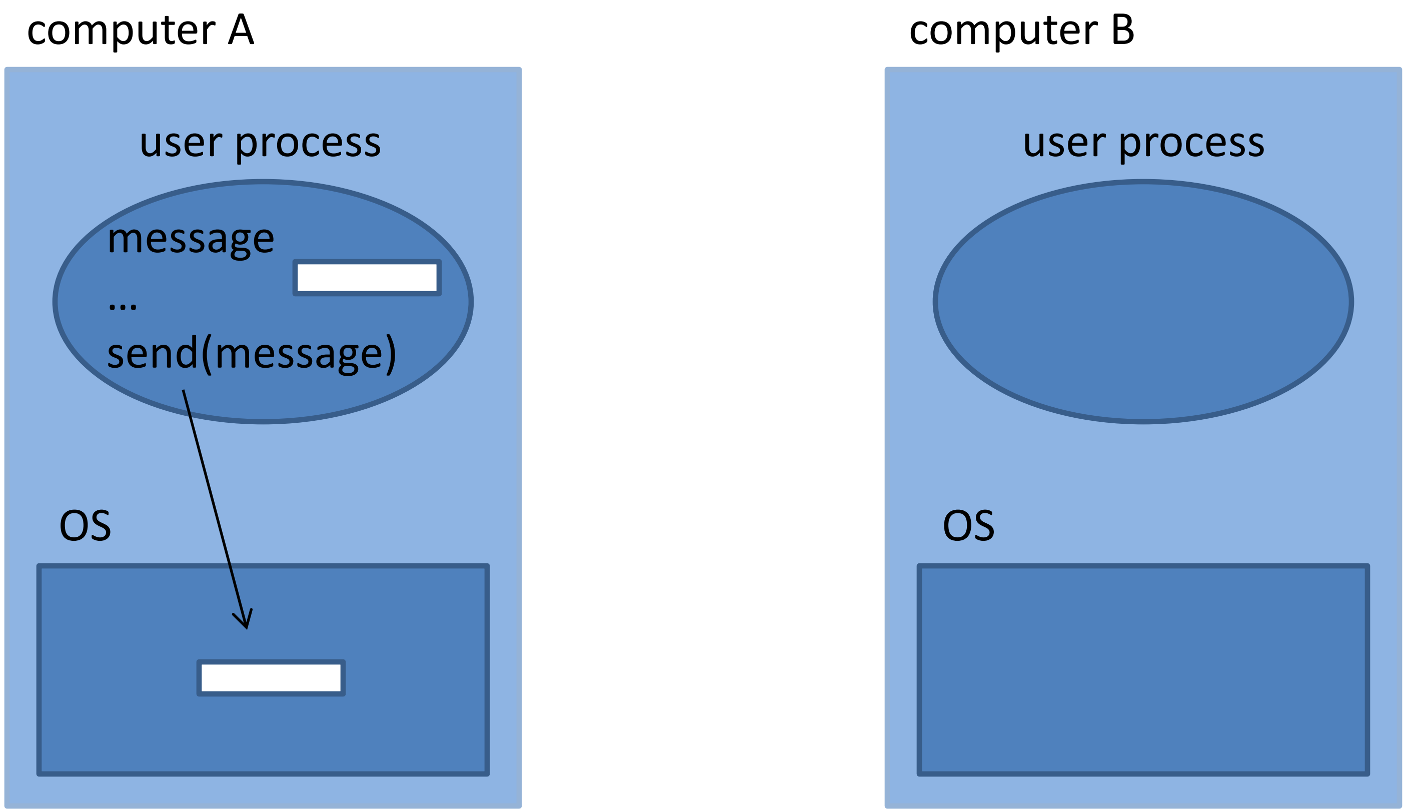
RPC, RMI

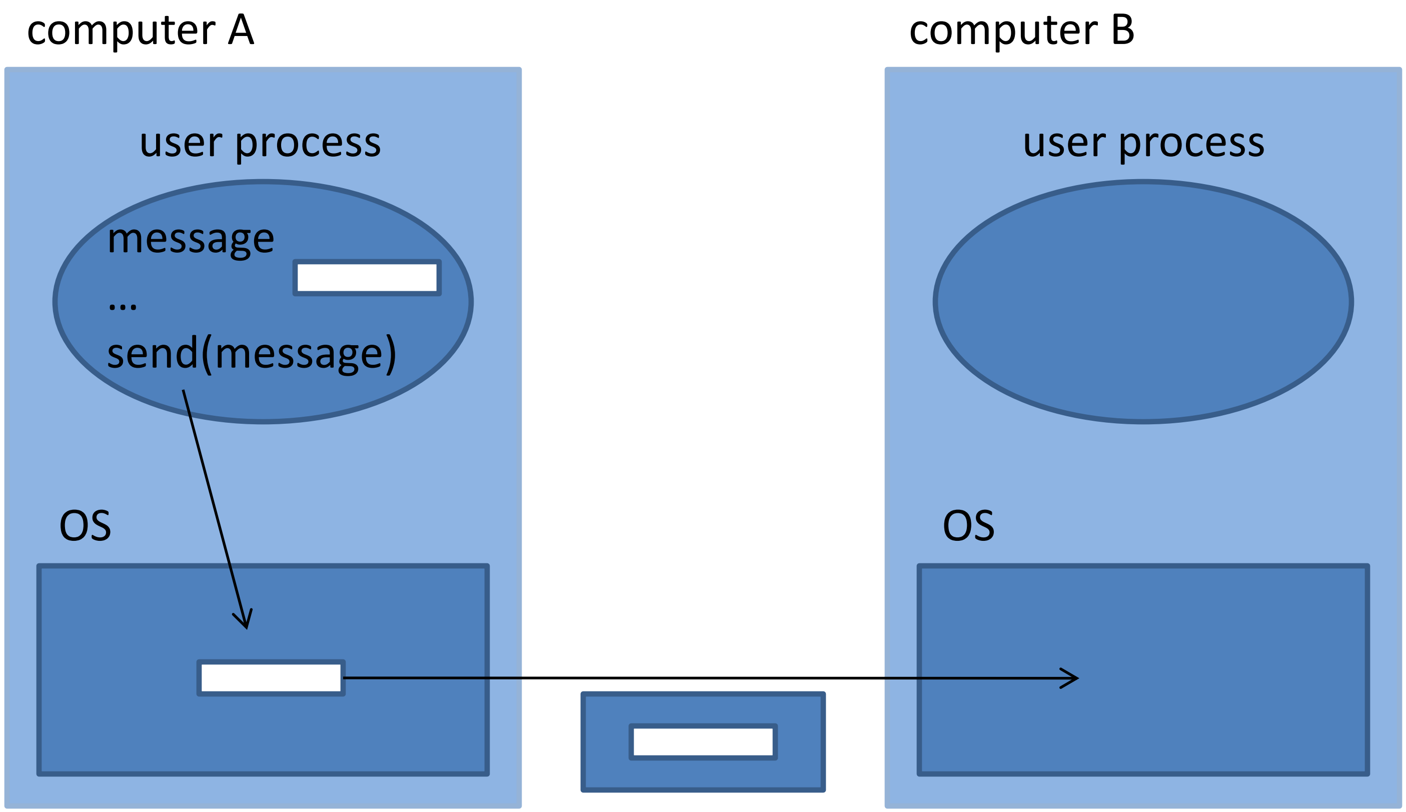
Client-server, P2P

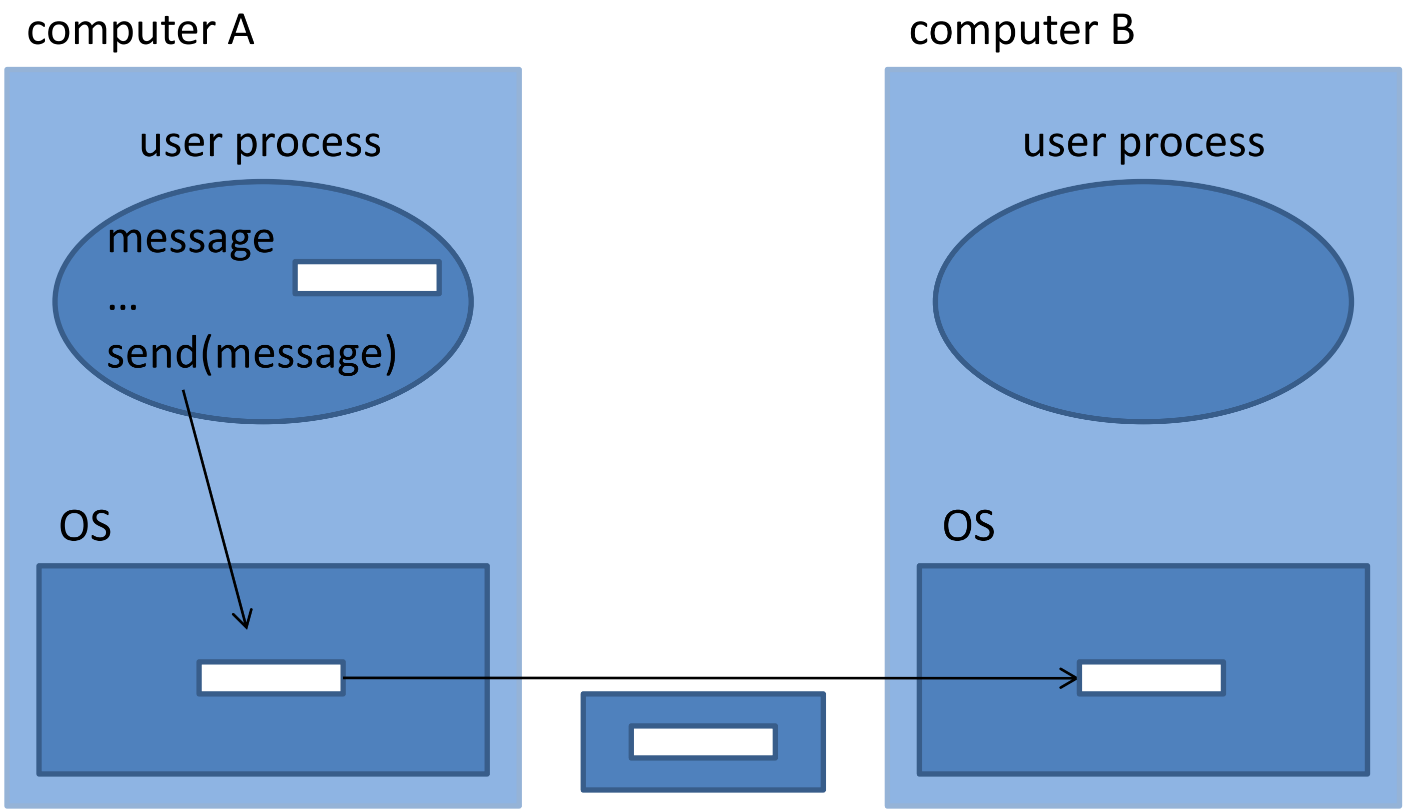
Message passing

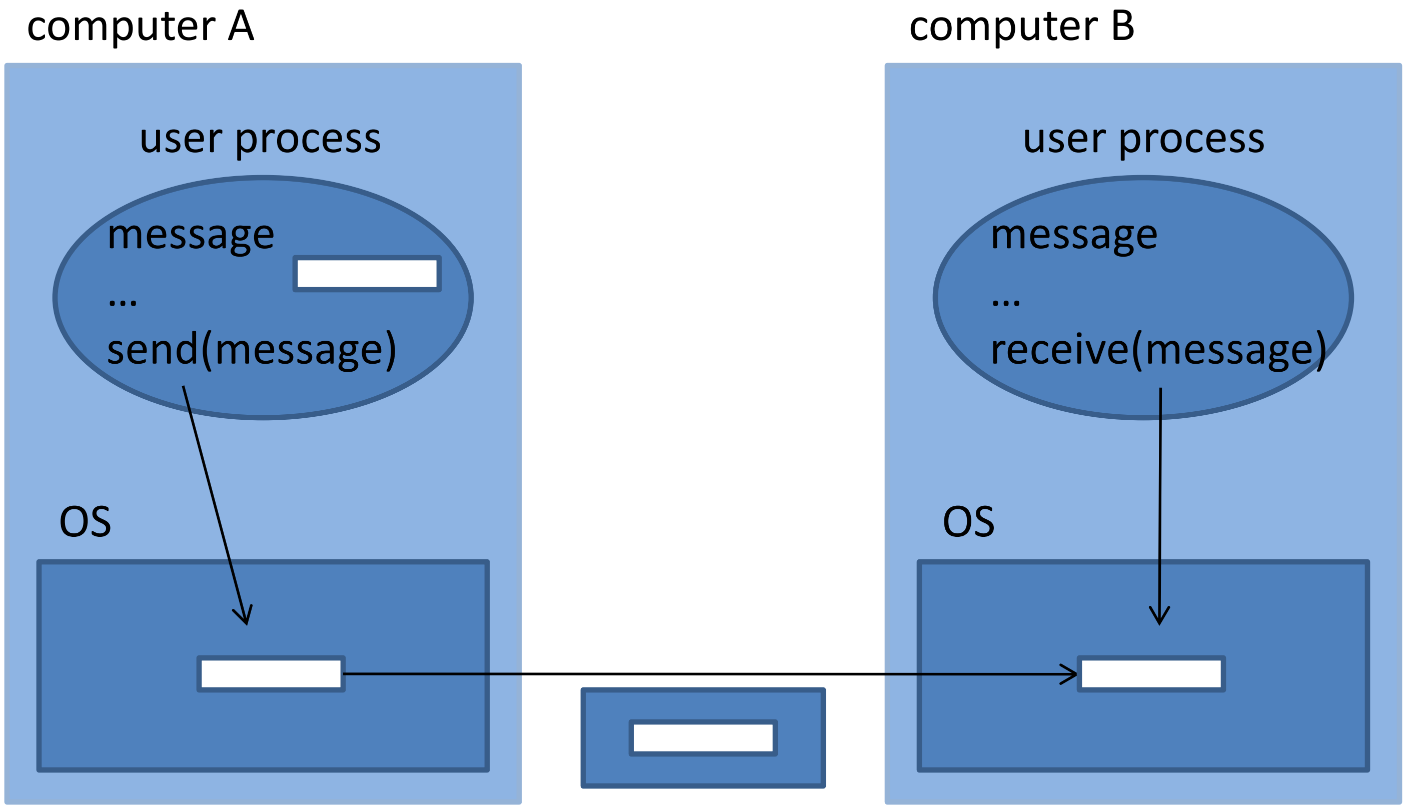

computer A
user process
message
...
OS
computer B
user process
OS

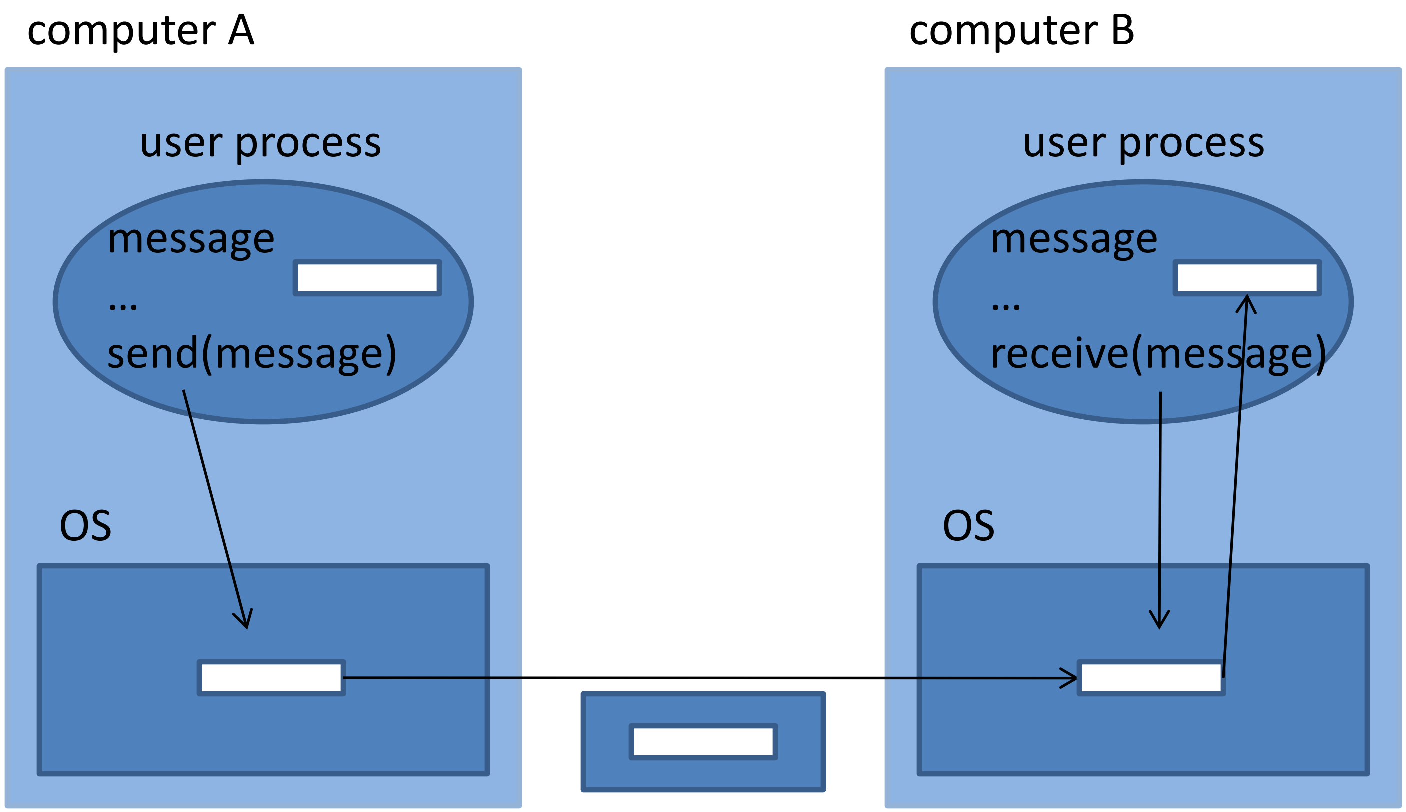

computer A
user process
message
...
send(message)
OS
computer B
user process
OS


computer A
user process
message
...
send(message)
OS
computer B
user process
OS


computer A
computer B
user process
message
...
send(message)
OS
user process
OS


computer A
user process
message
...
send(message)
OS
computer B
user process
OS


computer A
user process
message
...
send(message)
OS
computer B
user process
message
...
receive(message)
OS


computer A
user process
message
...
send(message)
OS
computer B
user process
message
...
receive(message)
OS

Other concurrency models besides message passing (MP)

- Data parallelism – determined by data partitioning
- Shared memory – common memory space
- Remote memory ops – process may access other process' memory w/o its participation
- Threads

Advantages and drawbacks of MP

- Advantages:
  - Hardware match – fits well on parallel supercomputers and clusters
  - Functionality – provides control not enabled by data parallelism and compiler-based models
  - Performance – explicit control of data locality
- Drawback:
  - Responsibility on the programmer: data distribution scheme, communication, synchronization w/o deadlock and race conditions

MP…

- …may be implemented in the programming language, e.g. distributed object systems
- …or in libraries, e.g. MPI

- Generally programming languages define messaging as asynchronous sending (by copy) of data item to communication endpoint

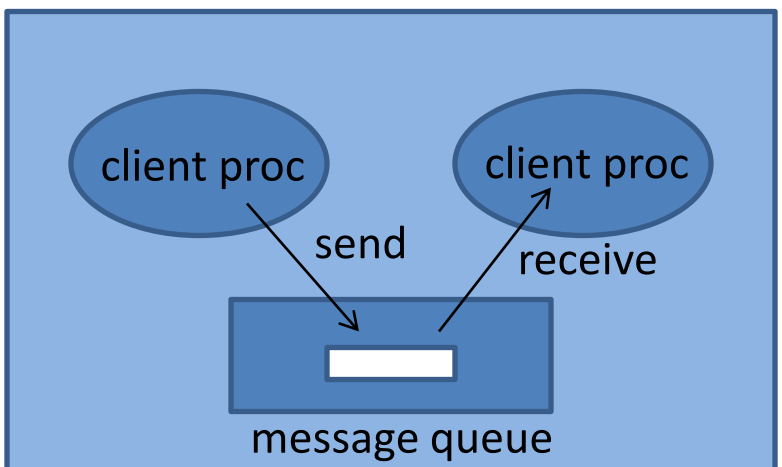
- MP forms the base of prominent theoretical foundations of concurrent computation

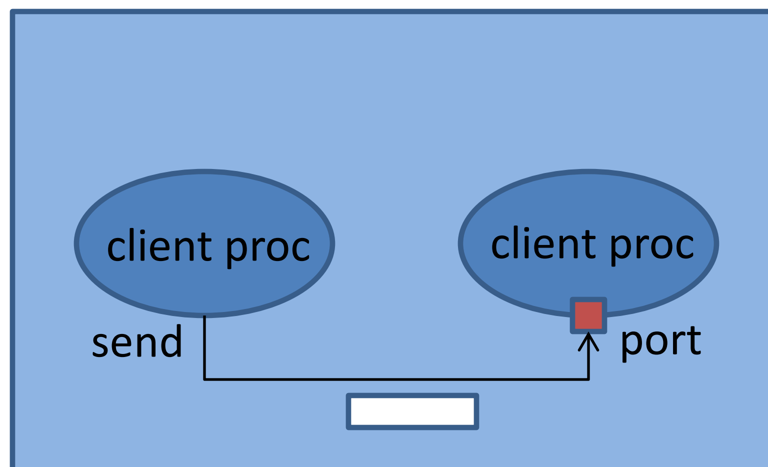
Design issues

- Message size: fixed vs variable
- Data flow: uni- vs bidirectional link
  - Can I both send and receive over a link?
- Naming message destination: direct vs indirect
  - Direct: specify the process to send to / receive from
  - Indirect: message queues, ports
- Communication type: synchronous vs asynchronous, blocking vs nonblocking
- Buffering: temporary copying of the message as part of the transmission protocol
- Reliability: are messages transferred reliably?
- Message order: are messages guaranteed to be delivered in order?

Naming

- Indirect: message sent to intermediary structures rather than to the process directly
  - Ports: associated with a process, e.g. sockets
  - Message queues: may be used by multiple senders / receivers, e.g. POSIX message queues

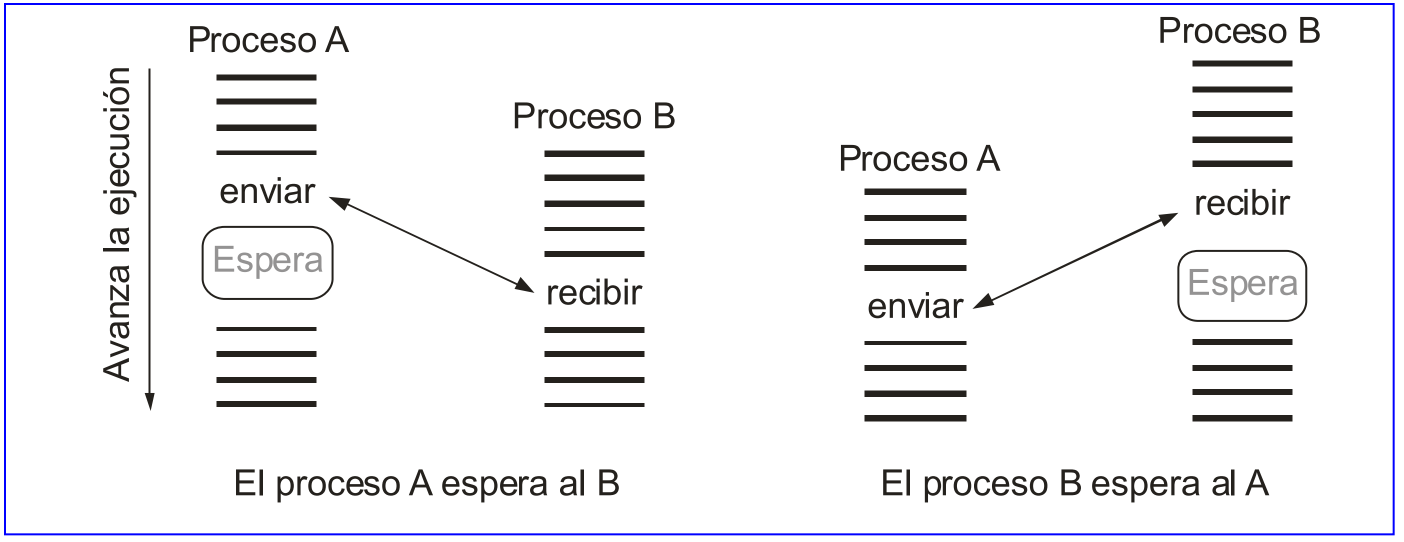




Communication type

- Synchronous: the sender does not return until matching receive in the destination process
  - Advantages: reasoning is simple, buffering not required (message stored on receiving side)
- Asynchronous: sender and receiver place no constraints on each other in terms of completion
  - Advantage: sender and receiver can overlap computation
  - Buffer may become full – must decide whether sender gets blocked (possible deadlock) or message is discarded (communication no longer reliable)
- Synchronous communication may be built on top of asynchronous by forcing the sender to wait for ACK from receiver before continuing

- Synchronous communication



Terminology

- Blocking communication: if the completion of the operation is dependent on certain events
  - For “send”: data successfully sent or safely copied s.t. buffer available for reuse
  - For “receive”: data must be safely stored in the receive buffer
- Nonblocking communication: operation returns w/o waiting for communication events to complete
- Synch/asynch implies blocking/nonblocking BUT
  - Not every blocking op is synchronous!
    - E.g. block on send till receiver machine has received the message, but receiver PROCESS may not have
  - Not every nonblocking is asynch!
    - E.g. synch nonblocking send completes if no receive_complete yet: send_start, send_complete (data copied out of send buffer)

- Solutions to unsafe programs
  - Reorder operations

P0	P1
Send(P1)	Receive(P0)
Receive(P1)	Send(P0)

  - Use operations which supplies receive buffer at the same time as send (if available)

P0	P1
Sendrecv(P1)	Sendrecv(P0)

  - Use nonblocking operations

	P0	P1
return immediately	nbSend(P1)	Rnbeceive(P0)
	nbReceive(P1)	nbSend(P0)
ensure comm. completed	waitall	waitall

Buffering

- Refers to the feature of a communication protocol to temporarily copy the message into a buffer it controls
  - No buffering
  - With buffering: the intermediate structures (e.g. queues) have a given size
    - Queue not full: may send message
    - Otherwise: block or discard messages
    - If buffering is done in OS space it may lead to flooding!

UNIX messages queues

- Used for communication and synchronization
  – Use common queue to send/receive data
  – Blocking / nonblocking semantics
- Indirect naming
- Associated with files: processes can use queues only if they share the same file system
- Variable message size
- Bidirectional data flow
- Asynchronous send, synch/asynch receive
- Messages may have associated priorities

POSIX for managing message queues

- Create a queue with name and attribute

```
mqd_t mq_open(char *name, int flag, mode_t mode, struct mq_attr *attr)
```

`flag:`	
`O_CREAT`	Create queue if it doesn't exist
`O_RDONLY`	Crea e a RO queue
`O_WRONLY`	Create a WO queue
`O_RDWR`	Create a RW queue
`O_NONBLOCK`	nonblocking send and receive
mode:	R, W, Exec rights
attr:	max. nr of messages, message size, etc

- Close a queue

```
int mq_close(mqd_t mqdes)
```

- Delete a queue (after closing)

```
int mq_unlink(char *name)
```

E.g. create/destroy a message queue

```
#include <printf.h>         /* printf */
#include <fcntl.h>          /* flags */
#include <sys/stat.h>       /* modes */
#include <mqueue.h>
#define  NUM_MSGS 50
int main () {
   mqd_t mqd;               /* Queue descriptor */
   struct mq_attr atributes;      /* Atributes */

   atributes.mq_maxmsg = NUM_MSGS;
   atributes.mq_msgsize = sizeof(int);

   if ((mqd=mq_open("/store.txt", O_CREAT|O_WRONLY, 0777, &atributes))==-1){
        printf("Error mq_open\n");
        exit(-1);
   }
   mq_close(mqd);
   mq_unlink("/store.txt");
}
```

POSIX for managing message queues

- Modify queue attributes

```
int mq_setattr(mqd_t mqdes, struct mq_attr *qstat,
               struct mq_attr *oldmqstat)
```

```
      gstat: Contains new attributes
      if oldmqstat != NULL it will store the new attributes
```

- Obtain queue attributes

```
int mq_getattr (mqd_t mqdes, struct mq_attr *qstat)
```

UNIX

- Send message to a queue

```
int mq_send(mqd_t mqdes, char *msg, size_t len,
            int prio)
```

Message msg of size len sent to queue mqdes with priority prio

If queue full send may be blocking or not depending on **O_NONBLOCK** and it returns nr of bytes sent or -1 (error)

- Receive message from a queue

```
int mq_receive(mqd_t mqdes, char *msg,
               size_t len, int *prio)
```

Extract message with highest priority

If queue empty block receive if not **O_NONBLOCK** and return nr of bytes read or -1 (error)

Producer-consumer with MP

```
Producer () {
   for(;;) {
        <Produce data>
         send(Consumer, data);
   } /* end for */
}
```

```
Consumer () {
   for(;;) {
      receive(Producer, data);
      <Consume data>
   } /* end for */
}
```

Producer-consumer with message queues

```
#include <mqueue.h>
#include <stdio.h>
#define MAX_BUFFER                    1024      /* buffer size */
#define PROD_DATA                     100000    /* data produced */

mqd_t store;                          /* queue for inserting produced
                                      data and removing consumed data */
void main(void){
        struct mq_attr attr;
        attr.mq_maxmsg = MAX_BUFFER;
        attr.mq_msgsize = sizeof (int);
        store = mq_open("STORE", O_CREAT|O_WRONLY, 0700, &attr);
        if (store == -1){
                perror ("mq_open");
                exit(-1);
        }
        Producer();
        mq_close(store);
        exit(0);
}
```

```
Producer(void){
        int data;
        int i;

        for(i=0;i<PROD_DATA;i++){
                /* produce data */
                data = i;
                if (mq_send(store, &data, siezof(int), 0)== -1){
                        perror("mq_send");
                        mq_close(store);
                        exit(1);
                }
        } /* end for */
        return;
} /* end producer */
```

```c
#include <mqueue.h>
#include <stdio.h>
#define MAX_BUFFER                    1024     /* buffer size */
#define PROD_DATA                     100000   /* data produced */

mqd_t store;

void main(void){
        struct mq_attr attr;
        store = mq_open("STORE", O_RDONLY);
        if (store == -1){
                perror ("mq_open");
                exit(-1);
        }
        Consumer();
        mq_close(store);

        exit(0);
}
```

```
Consumer(void){
	int data;
	int i;

	for(i=0;i<PROD_DATA;i++){
		/* receive data */
		data = i;
		if (mq_receive(store,
				&data, siezof(int), 0)== -1){
			perror("mq_send");
			mq_close(store);
			exit(1);
		}
		/* Consume data */
		printf("data consumed: %d\n", data);
	} /* end for */
	return;
} /* end consumer */
```

Group communication

- Data movement
  - One-to-many: multicast, broadcast
  - Many-to-one: client-server
  - All-to-all
  - Gather/scatter
- Synchronization
- Collective computation (reductions)
- Useful for:
  - Fault tolerance via replication of services
  - Better performance via data replication
  - Multiple updates
  - Reduce operations for parallel computation
- Collective operations are blocking

Examples of MP-style languages

- Actors
- Amorphous computing
- Antiobjects
- Flow-based programming
- SOAP protocol
- Smalltalk

- Lambda calculus is considered the earliest MP programming language

Actors

- Before actors concurrency defined in terms of threads, locks, and buffers
- Caracteristics:
  - Asynchronous communication
  - Direct naming: message recipients identified by address
    - Recipient address may be included in message
  - No restriction on message arrival – similar to packet switching systems, enables optimizations: buffering packets, send on different paths ,resend, pipeline message processing, etc
  - In most formalisms message delivery is guaranteed
  - Not necessarily buffered

MPL programming: performance guidelines

- Start with tuned serial program
- Control process granularity: increased granularity decreases communication /computation overhead
- Overlap communication and computation: use nonblocking communication to hide comm. costs
- Avoid unnecessary synchronization
- Avoid buffering when possible: extra copies of messages decrease bandwidth
- Keep data local