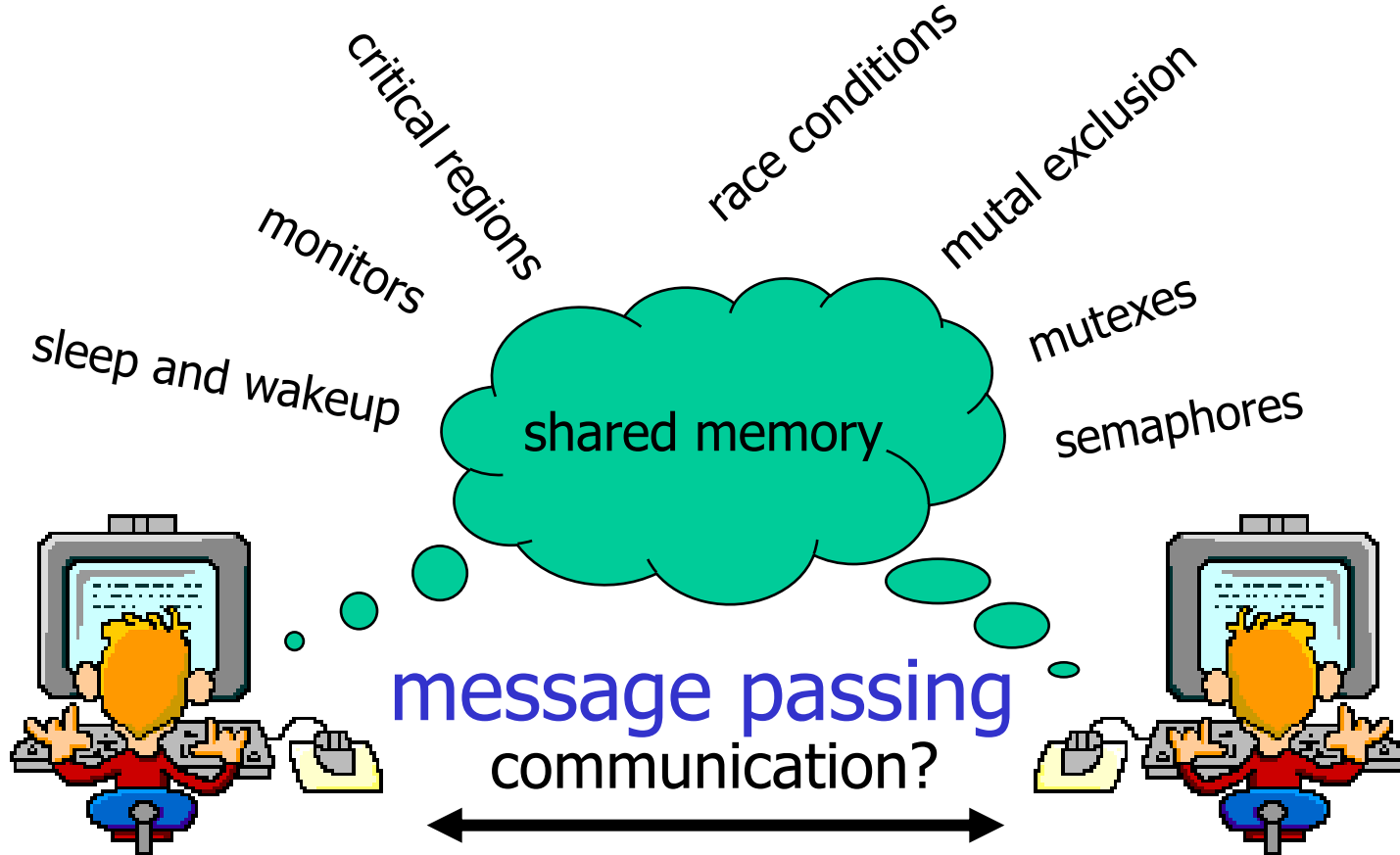


Inter-Process Communication: Message Passing

Tore Larsen

Slides by T. Plagemann, Pål Halvorsen,
Kai Li, and Andrew S. Tanenbaum

Big Picture



Message Passing API

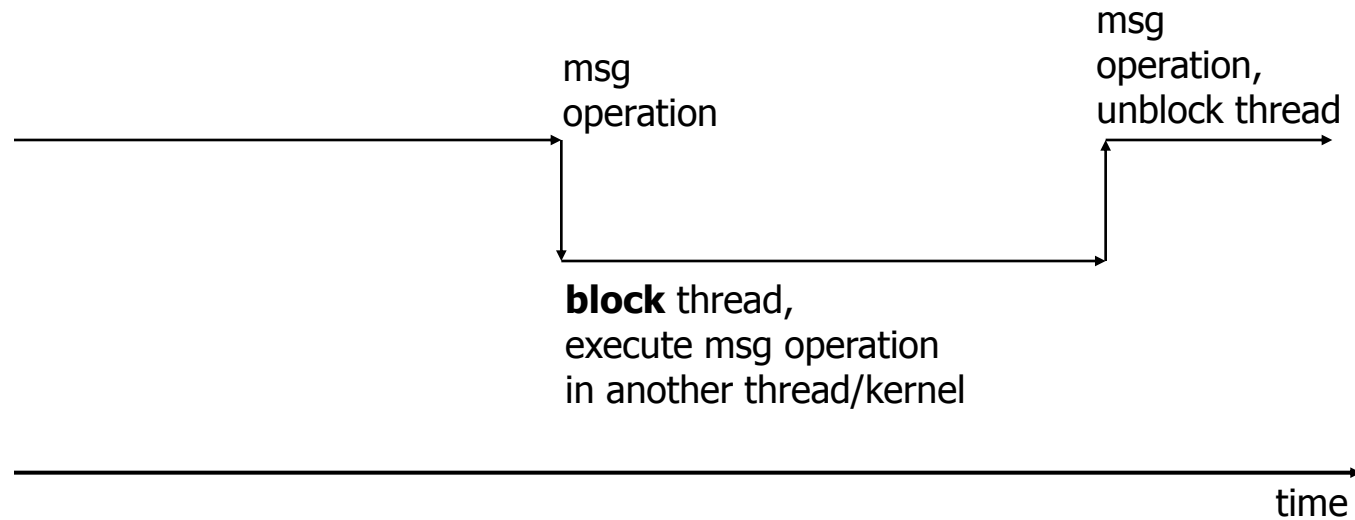
- Generic API
 - `send(dest, &msg)`
 - `recv(src, &msg)`
- What should the “`dest`” and “`src`” be?
 - `pid`
 - `file`: e.g. a (named) pipe
 - `port`: network address, `pid`, etc
 - `no src`: receive any message
 - `src` combines both specific and any
- What should “`msg`” be?
 - Need both buffer and size for a variable sized message

Issues

- Asynchronous vs. synchronous
- Direct vs. indirect
- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between any pair?
- What is the capacity of a link?
- What is the size of a message?
- Is a link unidirectional or bidirectional?

Asynchronous vs. Synchronous

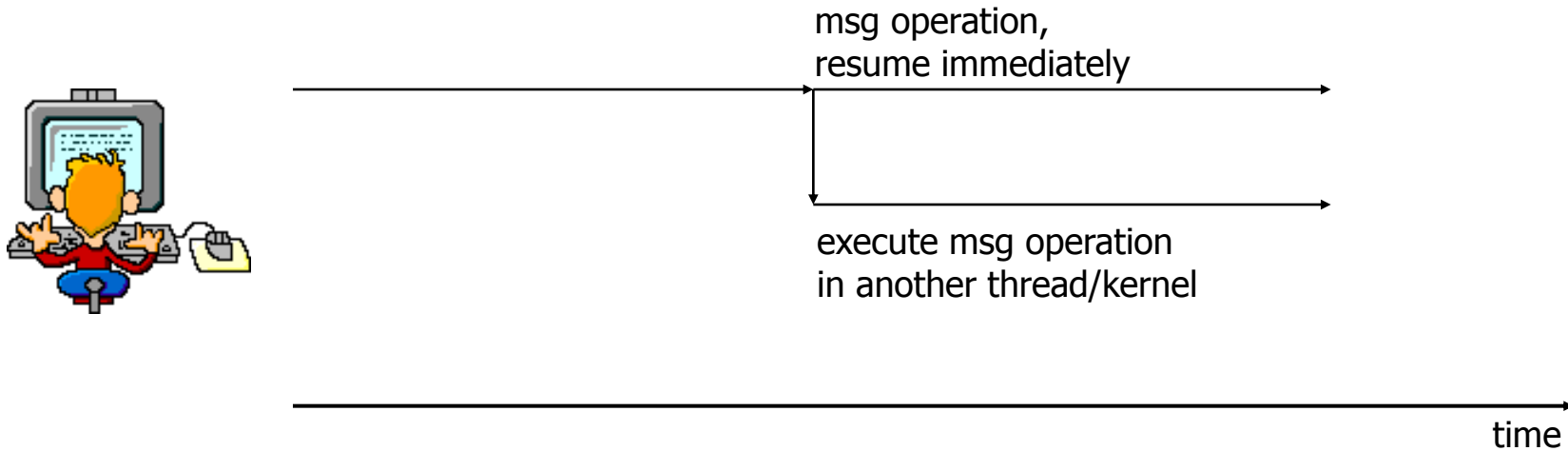
- Synchronous (blocking):



- thread is blocked until message primitive has been performed
- may be blocked for a very long time

Asynchronous vs. Synchronous

- Asynchronous (non-blocking):



- thread gets control back immediately
- thread can run in parallel other activities
- thread cannot reuse buffer for message before message is received
- how to know when to start if blocked on full/empty buffer?
 - poll
 - interrupts/signals
 - ...

Asynchronous vs. Synchronous

- Send semantic:

- Synchronous

- Will not return until data is out of its source memory
 - Block on full buffer

- Asynchronous

- Return as soon as initiating its hardware
 - Completion
 - Require application to check status
 - Notify or signal the application
 - Block on full buffer

- Receive semantic:

- Synchronous

- Return data if there is a message
 - Block on empty buffer

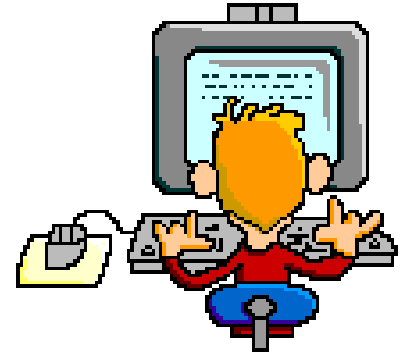
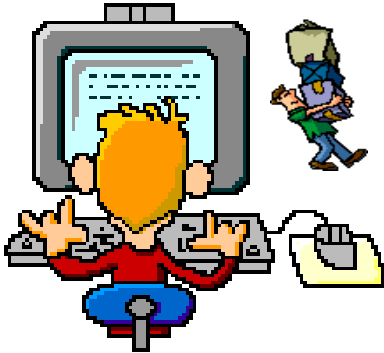
- Asynchronous

- Return data if there is a message
 - Return null if there is no message

Buffering

- No buffering
 - synchronous
 - Sender must wait until the receiver receives the message
 - Rendezvous on each message
- Buffering
 - asynchronous or synchronous
 - Bounded buffer
 - Finite size
 - Sender blocks when the buffer is full
 - Use mesa-monitor to solve the problem?
 - Unbounded buffer
 - “Infinite” size
 - Sender never blocks

Direct Communication



- Must explicitly name the sender/receiver ("`dest`" and "`src`") processes
- A buffer at the receiver
 - More than one process may send messages to the receiver
 - To receive from a specific sender, it requires searching through the whole buffer
- A buffer at each sender
 - A sender may send messages to multiple receivers

Message Passing: Producer-Consumers Problem

```
void producer(void)
{
    while (TRUE) {
        ...
        produce item;
        ...
        send( consumer, item );
    }
}
```

```
void consumer(void)
{
    while (TRUE) {
        recv( producer, item );
        ...
        consume item;
        ...
    }
}
```

Message Passing: Producer-Consumers Problem with N messages

```
#define N 100                                /* number of slots in the buffer */

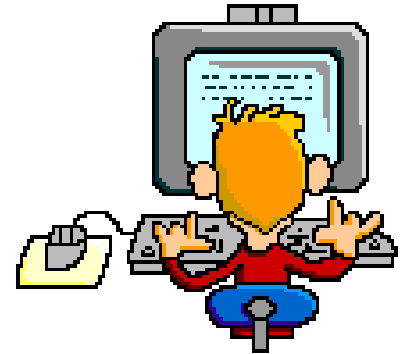
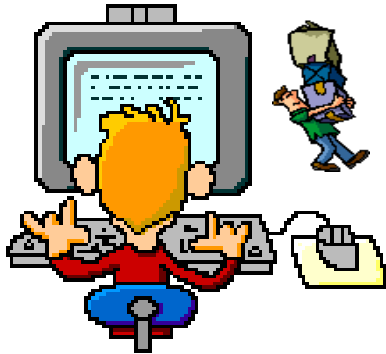
void producer(void)
{
    int item;
    message m;                                /* message buffer */

    while (TRUE) {
        item = produce_item();               /* generate something to put in buffer */
        receive(consumer, &m);              /* wait for an empty to arrive */
        build_message(&m, item);            /* construct a message to send */
        send(consumer, &m);                 /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m);              /* get message containing item */
        item = extract_item(&m);            /* extract item from message */
        send(producer, &m);                 /* send back empty reply */
        consume_item(item);                 /* do something with the item */
    }
}
```

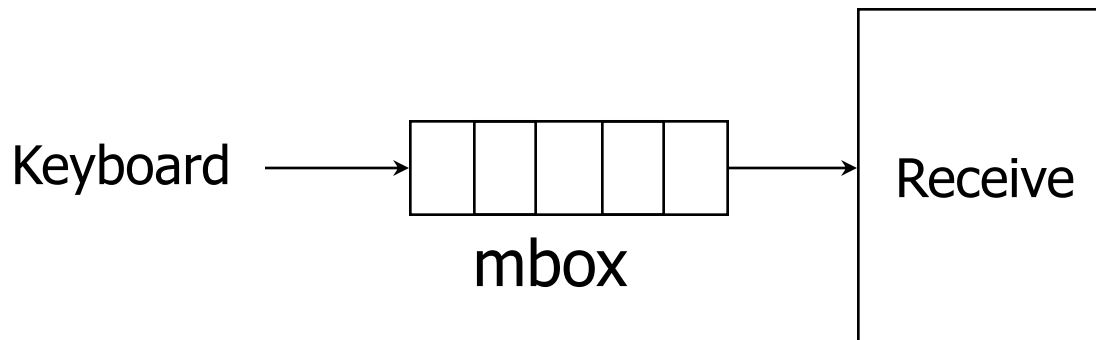
Indirect Communication



- “`dest`” and “`src`” are a shared (unique) mailbox
- Use a mailbox to allow many-to-many communication
 - Requires open/close a mailbox before using it
- Where should the buffer be?
 - A buffer and its mutex and conditions should be at the mailbox

Using Message-Passing

- What is message-passing for?
 - Communication across address spaces
 - Communication across protection domains
 - Synchronization
- Use a mailbox to communicate between a process/thread and an interrupt handler: fake a sender



Process Termination

- P waits for a message from Q, but Q has terminated
 - Problem: P may be blocked forever
 - Solution:
 - P checks once a while
 - Catch the exception and informs P
 - Send ack message

- P sends a message to Q, but Q has terminated
 - Problem: P has no buffer and will be blocked forever
 - Solution:
 - Check Q's state and cleanup
 - Catch the exception and informs P

Message Loss & Corruption

- Unreliable service
 - best effort, up to the user to
- Detection
 - Receiver: Acknowledge each message sent!
 - Sender: Timeout if no ack message from receiver
- Retransmission
 - Sequence number for each message
 - Retransmit a message on timeout
 - Retransmit a message on out-of-sequence acknowledgement
 - Remove duplication messages on the receiver side

Linux Mailboxes

- Messages are stored as a sequence of bytes
- System V IPC messages also have a type
- Mailboxes are implemented as message queues sorting messages according to FIFO
- Can be both blocking and non-blocking (`IPC_NOWAIT`)
- The next slides have some simplified (pseudo) code
 - Linux 2.4.18
 - several parts missing
 - the shown code may block holding the queue lock
 - waiting queues are more complex
 - ...

Linux Mailboxes

- One `msg_queue` structure for each present queue:

```
struct msg_queue {
    struct kern_ipc_perm q_perm;        /* access permissions */
    time_t q_stime;                    /* last msgsnd time */
    time_t q_rtime;                    /* last msgrcv time */
    time_t q_ctime;                    /* last change time */
    unsigned long q_cbytes;            /* current number of bytes on queue */
    unsigned long q_qnum;              /* number of messages in queue */
    unsigned long q_qbytes;            /* max number of bytes on queue */
    pid_t q_lspid;                     /* pid of last msgsnd */
    pid_t q_lrpid;                     /* last receive pid */
    struct list_head q_messages;
    struct list_head q_receivers;
    struct list_head q_senders;
};
```

- Messages are stored in the kernel using the `msg_msg` structure:

```
struct msg_msg {
    struct list_head m_list;
    long m_type;                       /* message type */
    int m_ts;                           /* message text size */
    struct msg_msgseg* next;           /* next pointer */
};
```

NOTE: the message is stored immediately after this structure - no pointer is necessary

Linux Mailboxes

- Create a message queue using the `sys_msgget` system call:

```
long sys_msgget (key_t key, int msgflg)
{
    ...
    create new message queue and set access permissions
    ...
}
```

- To manipulate a queue, one uses the `sys_msgctl` system call:

```
long sys_msgctl (int msqid, int cmd, struct msqid_ds *buf)
{
    ...
    switch (cmd) {
        case IPC_INFO:
            return info about the queue, e.g., length, etc.
        case IPC_SET:
            modify info about the queue, e.g., length, etc.
        case IPC_RMID:
            remove the queue
    }
    ...
}
```

Linux Mailboxes

- Send a message to the queue using the `sys_msgsnd` system call:

```
long sys_msgsnd (int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg)
{
    msq = msg_lock(msqid);
    ...

    if ((msgsz + msq->q_cbytes) > msq->q_qbytes)
        insert message the tail of msq->q_messages *msgp, update msq
    else
        put sender in waiting senders queue (msq->q_senders)

    msg_unlock(msqid);
    ...
}
```

Linux Mailboxes

- Receive a message from the queue using the `sys_msgrcv` system call:

```
long sys_msgrcv (int msqid, struct msgbuf *msgp, size_t msgsz,  
                long msgtyp, int msgflg)  
{  
    msq = msg_lock(msqid);  
    ...  
    search msq->q_messages for first message matching msgtype  
    if (msg)  
        store message in msgbuf *msgp, remove message from msgbuf *msgp, update msq  
    else  
        put receiver in waiting receivers queue (msq->q_receivers)  
  
    msg_unlock(msqid);  
    ...  
}
```

- the `msgtyp` parameter and `msgflg` flag determine which messages to retrieve:
 - = 0: return first message
 - > 0: first message in queue with `msg_msg.m_type = msgtyp`
 - > 0 & `MSG_EXCEPT`: first message in queue with `msg_msg.m_type != msgtyp`

Our Mailbox

- We make it simpler than Linux
 - Deliver messages in FIFO order
- Mailbox \Rightarrow buffer space
 - Finite space
- Main purpose:
 - Send
 - Receive
- Maintenance:
 - Init
 - Open
 - Close
 - Statistics

Linux Pipes

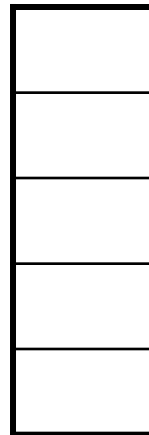
- Classic IPC method under UNIX:

```
> ls -l | more
```

- shell runs two processes `ls` and `more` which are linked via a pipe
- the first process (`ls`) writes data (e.g., using `write`) to the pipe and the second (`more`) reads data (e.g., using `read`) from the pipe

- the system call `pipe(fd[2])` creates one file descriptor for reading (`fd[0]`) and one for writing (`fd[1]`)
 - allocates a temporary inode and a memory page to hold data

```
struct pipe_inode_info {  
    wait_queue_head_t wait;  
    char *base;  
    unsigned int len;  
    unsigned int start;  
    unsigned int readers, writers;  
    unsigned int waiting_readers, waiting_writers;  
    unsigned int r_counter, w_counter;  
}
```



Linux: Mailboxes vs. Pipes

- Are there any differences between a mailbox and a pipe?
 - Message types
 - mailboxes may have messages of different types
 - pipes do not have different types
 - Buffer
 - pipes – one or more pages storing messages contiguously
 - mailboxes – linked list of messages of different types
 - Termination
 - pipes exists only as long as some have open the file descriptors
 - mailboxes must often be closed
 - More than two processes
 - a pipe **often** (not in Linux) implies one sender and one receiver
 - many can use a mailbox

Performance

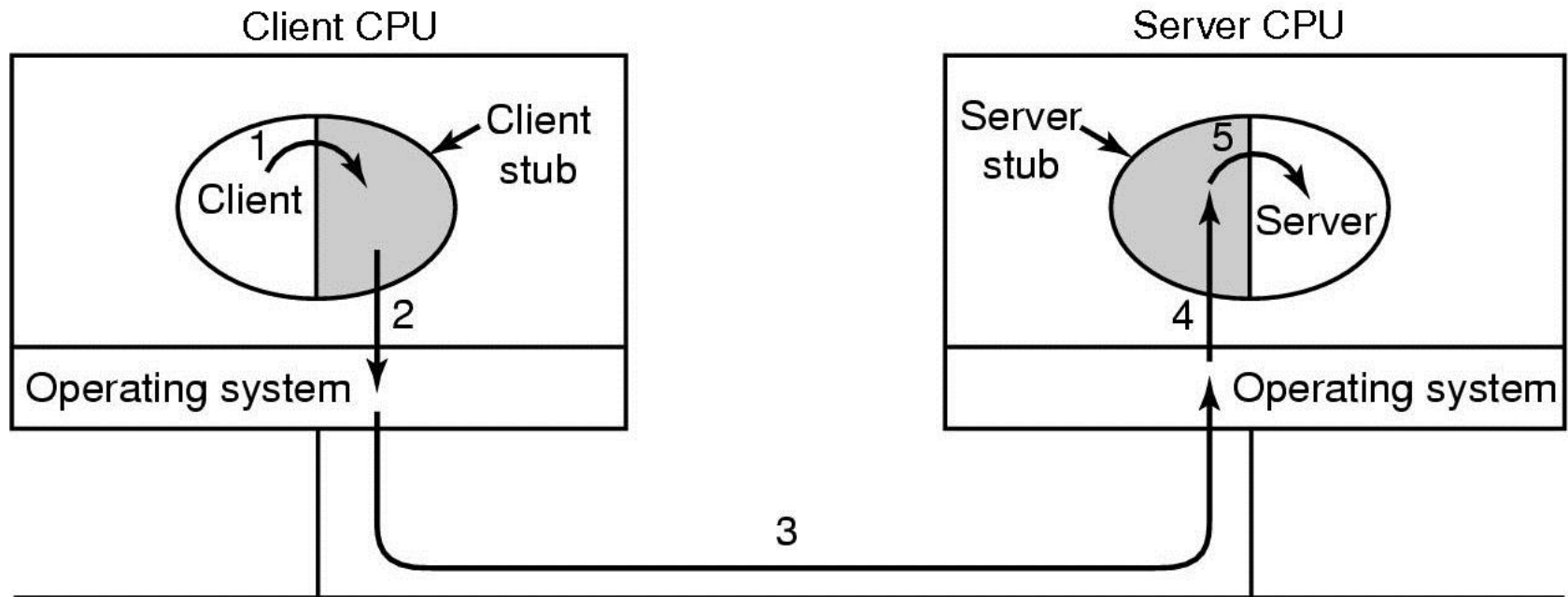
- Performance is an important issue
(at least when sender and receiver is on one machine), e.g.:
 - shared memory and using semaphores
 - mailboxes copying data from source to mailbox and from mailbox to receiver
- Can one somehow optimize the message passing?

Programming Model

- Message passing provides a model for program interaction that spans across separate physical address spaces, but which is different from what is conventionally used within a single address space.
- Other approaches to communicate across address spaces include:
 - Distributed shared memory – Establish address spaces that are shared while still being physically distributed
 - Remote procedure calls – The ability to apply procedure calls (and returns) across address spaces

Remote Procedure Call

- Message passing uses I/O
- Idea of RPC is to make function calls
- Small libraries (stubs) and OS take care of communication



Remote Procedure Call

Implementation Issues:

- Cannot pass pointers - call by reference becomes copy-restore
- Marshaling - packing parameters
- Weakly typed languages - client stub cannot determine size
- Not always possible to determine parameter types
- Cannot use global variables - may get moved to remote machine/protection domain

Summary

- Message passing
 - Move data between processes
 - Implicit synchronization
 - API design is important
- Implementation issues
 - Synchronous method is most common
 - Asynchronous method provides overlapping, but requires careful design considerations
 - Indirection makes implementation flexible
 - Exceptions need to be handled carefully