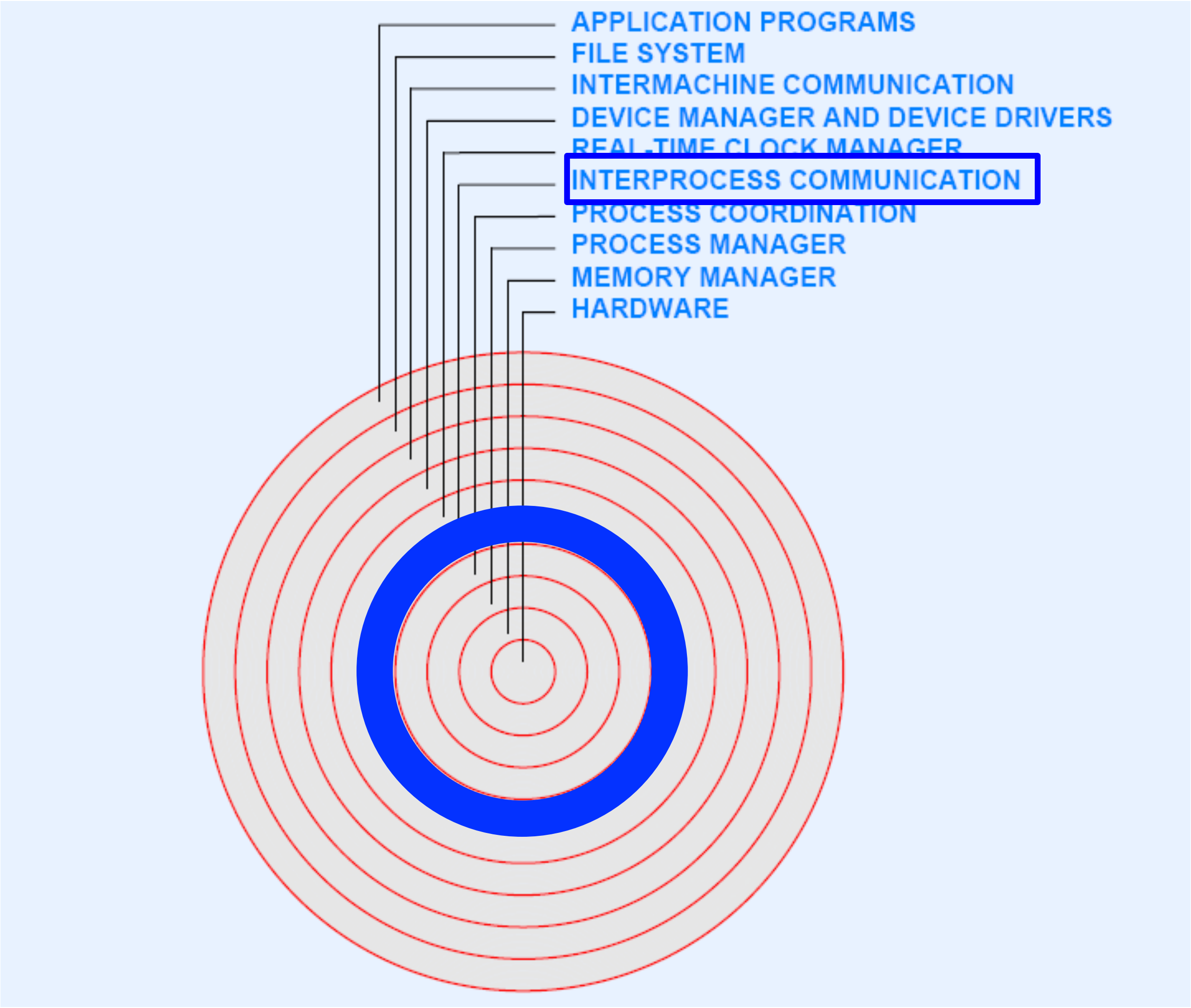
CSCI 8530
Advanced Operating Systems

Part 6

Inter-Process Communication

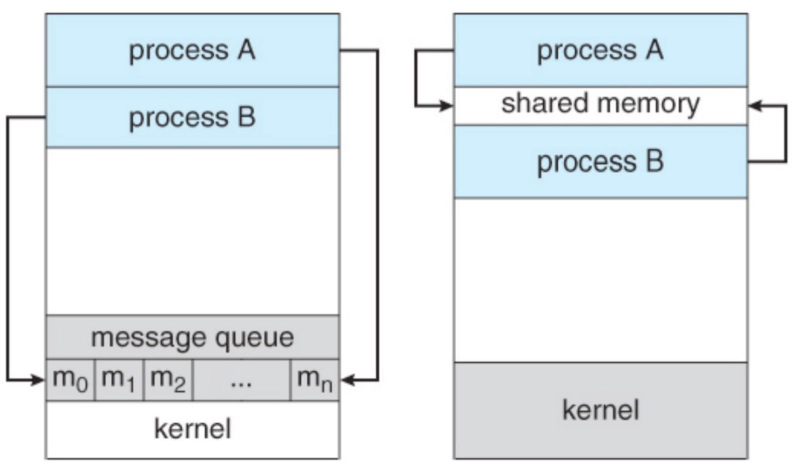
Location of Inter-Process Communication in the Hierarchy



Inter-process Communication

- A process can be of two type: Independent process (not affected by the execution of other processes), Cooperating process.
- A mechanism allows processes to communicate each other and synchronize their action
- Used for
  - Exchange of (nonshared) data
  - Process coordination
- Use these two ways: *Shared Memory, Message passing*

- Ccommunication takes place by way of messages exchanged among the cooperating processes.

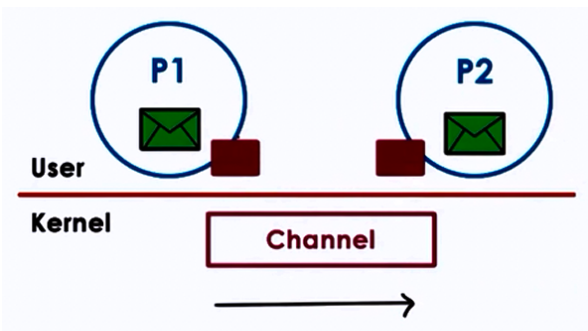


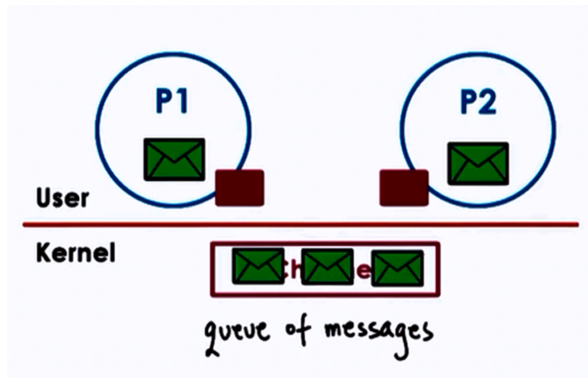
(a) Message Passing (b) Shared Memory

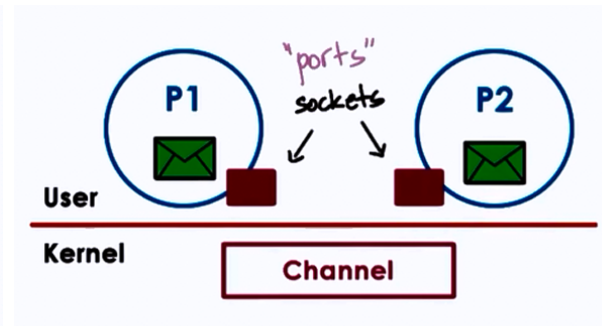
- A region of memory which is shared by cooperating processes gets established.
- Processes can exchange information by reading and writing all the data to the shared region.

https://www.w3schools.in/operating-system-tutorial/interprocess-communication-ipc/

Forms of Message Passing IPC







Pipes
Carry byte stream between 2 process

Message queues
Carry "messages" among processes

Sockets
- send() and recv() : pass message buffers
- socket() : create kernel level socket buffer

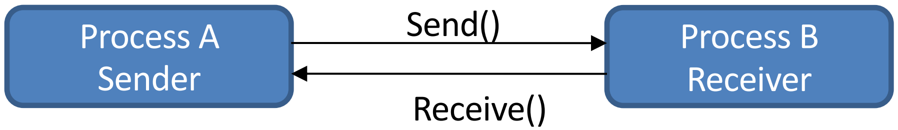
Two Approaches to Message Passing

- Approach #1
  - Message passing is one of many services
  - Messages are separate from I /O and process synchronization services
  - Messages implemented using lower-level mechanisms, such as semaphores
- Approach #2
  - The entire operating system is *message-based*
  - Messages, not function calls, provide the fundamental building block
  - Messages, not semaphores, used for process synchronization

Typical Message-Passing Functions

The following functions and data items are characteristic in a typical message-passing environment:

- *Source* and *Destination* addresses must identify the machine and the process being addressed.
- *message* is a generic data item to be delivered.
- Use two system call
  - send (destination, &message): sends message to destination without blocking; the sender's address is usually included
  - receive (source, &message): receives the next sequential message from source, blocking until it is available.



Design of a Message Passing Facility

- To understand the issues, we will begin with a trivial message passing facility
- We want to allow a process to send a message directly to another process
- In principle,
  - the design should be straightforward
- In practice,
  - many design decisions arise

Message Passing Design Decisions

- Are messages fixed or variable size?
- What is the maximum message size?
- How many messages can be outstanding at a given time?
- Where are messages stored?
- How is a recipient specified?
- Does a receiver know the sender's identity?
- Are replies supported?
- Is the interface synchronous or asynchronous?

Synchronous vs. Asynchronous Interface (1/3)

Can message passing also replace synchronization primitives such as semaphores?

- The answer depends on the implementation of message passing

Synchronous vs. Asynchronous Interface (2/3)

1. Synchronous interface: To receive a message in a synchronous system, a process calls a system function, and the call does not return until a message arrives
   – Blocks until the operation is performed
   – Easy to understand / program
   – Extra processes can be used to obtain asynchrony
   – Using message passing to implement mutual exclusion is possible

Synchronous vs. Asynchronous Interface (3/3)

2. Asynchronous interface: an asynchronous message passing system either requires a process to *poll* or a mechanism that allows OS to stop a process temporarily
   - Process starts an operation
   - Initiating process continues execution
   - Notification
     - Arrives when operation completes
     - May entail abnormal control flow (e.g., software interrupt or "callback" mechanism)
   - Polling can be used to determine status
   - Additional overhead or complexity, but convenient

Two Forms of Message Passing in Xinu

- A completely synchronous paradigm and a partially asynchronous paradigm
- Provide *direct* and *indirect* message delivery:
  - One provides a direct exchange of messages among processes (Ch. 8)
  - The other arranges for messages to be exchanged through rendezvous point (Ch11)

Why is a Message Passing Facility So Difficult to Design?

- Interacts with
  - Process coordination subsystem
  - Memory management subsystem
- Affects user’s perception of system

Message passing system in Xinu

- The message passing facility follows three guidelines:
  - *Limited message size.* The system limits each message to a small, fixed size.
  - *No message queues*. The system permits a given process to store only one unreceived message per process at any time.
  - *First message semantics*. If several messages are sent to a given process before the process receives any of them, only the first message is stored and delivered; subsequent senders do not block.
    - First message semantics: for determining which of several events completes first.

Example Inter-process Message Passing Design (1/3)

- Simple, low-level mechanism
- Direct process-to-process communication
- One-word messages
- Message stored with receiver
- One-message buffer
- Synchronous, buffered reception
- Asynchronous transmission and “reset” operation

Example Inter-process Message Passing Design (2/3)

- Three system calls manipulate messages

```
send(msg, pid);
msg = receive();
msg = recvclr();
```

  - *Send* transmits message to specified process
  - *Receive* blocks until a message arrives
  - *Recvclr:* Remove existing message, if one has arrived, but does not block
    - Receive a message → return the message (like *receive)*
    - No message is waiting→ return OK
- Message stored in *receiver's* process table entry

Example Inter-process Message Passing Design (3/3)

- First-message semantics
  - First message sent to a process is stored until it has been received
  - Subsequent attempts to send fail
- Idiom

```
recvclr();    /* prepare to receive a message */
... /* allow other processes to send messages */
msg = receive();
```

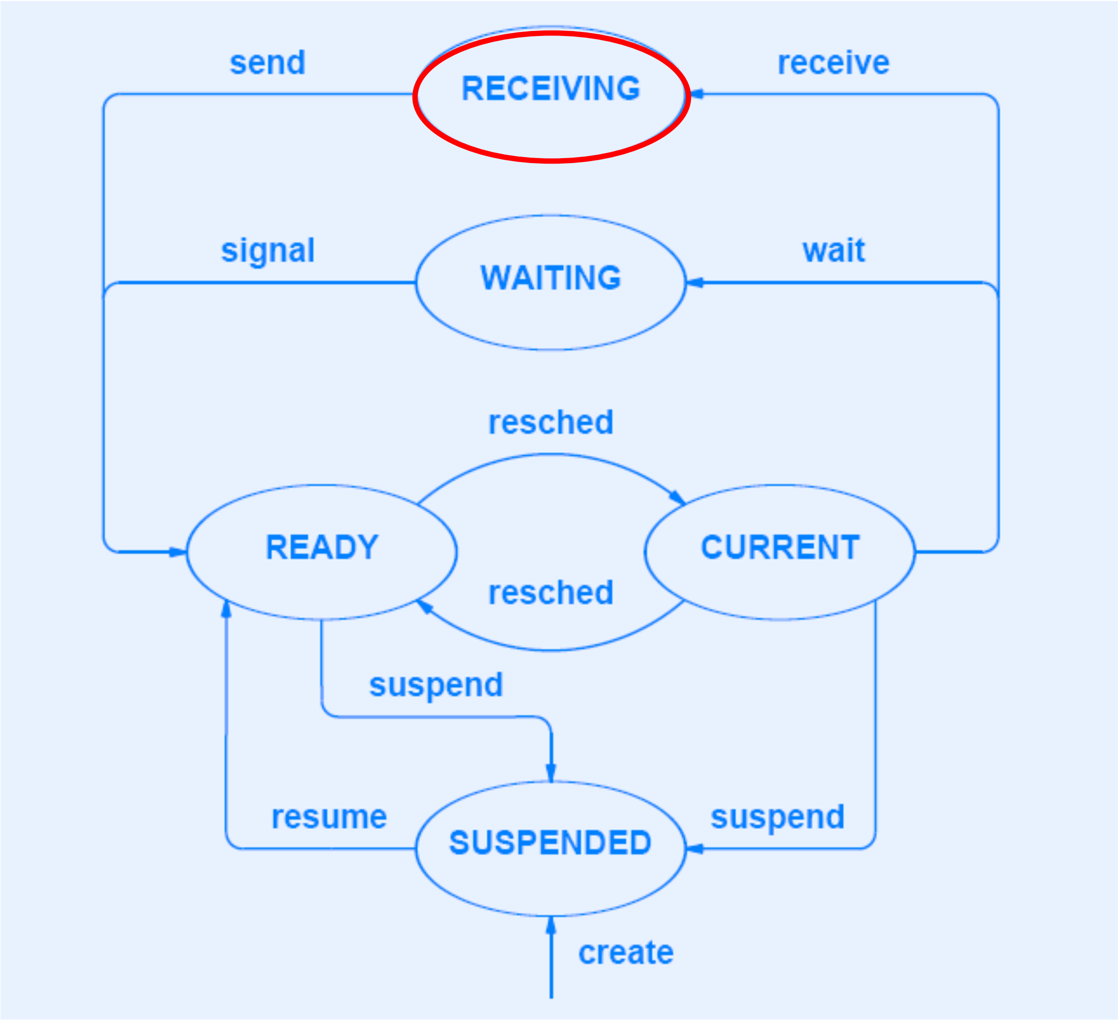
- Above code returns first message that was sent, even if a high priority process sends later

Process State for Message Reception

In what state should a process be while waiting for a message?

- While receiving a message, a process is NOT
  - Executing
  - Ready
  - Suspended
  - Waiting on a semaphore
- Therefore, a new state is needed for message passing
- Named *RECEIVING,* with the symbolic constant PR_RECV
- Entered when *receive* called

State Transitions with Message Passing



The symbolic constant PR_RECV

Xinu Code for Message Transmission (1/2)

A message cannot be stored **in the sender's memory**

```
/* send.c - send */

#include <xinu.h>

/*------------------------------------------------------------------------
 * send - Pass a message to a process and start recipient if waiting
 *------------------------------------------------------------------------
 */
syscall send(
          pid32 pid, /* ID of recipient process */
          umsg32 msg /* Contents of message */
         )
{
          intmask mask; /* Saved interrupt mask */
          struct procent *prptr; /* Ptr to process' table entry */

          mask = disable();
          if (isbadpid(pid)) {
              restore(mask);
              return SYSERR;
          }
          prptr = &proctab[pid];
          if ((prptr->prstate == PR_FREE) || prptr->prhasmsg) {
              restore(mask);
              return SYSERR;
          }
```

Check to ensure the recipient does not have a message outstanding

Xinu Code for Message Transmission (2/2)

Restrictions on the size of messages: The implementation reserve space for one message

```
        prptr->prmsg = msg;             /* Deliver message              */
        prptr->prhasmsg = TRUE;         /* Indicate message is waiting */

        /* If recipient waiting or in timed-wait make it ready */

        if (prptr->prstate == PR_RECV) { /* If the recipient is waiting */
                                        /* for the arrival of a message */
            ready(pid);
        } else if (prptr->prstate == PR_RECTIM) {
            unsleep(pid);
            ready(pid);
        }
        restore(mask);                  /* Restore interrupts */
        return OK;
}
```

Remove the process from the queue of sleeping processes

- Note: we will discuss PR_RECTIM(receive-with-timeout) later

Xinu Code for Message Reception

```
/* receive.c - receive */

#include <xinu.h>

/*------------------------------------------------------------------------
 * receive - Wait for an incoming message and return the message to the caller
 *------------------------------------------------------------------------
 */
umsg32 receive(void)
{
        intmask mask;                   /* Saved interrupt mask         */
        struct procent *prptr;          /* Ptr to process' table entry */
        umsg32 msg;                     /* Message to return            */

        mask = disable();
        prptr = &proctab[currpid];
        if (prptr->prhasmsg == FALSE) {    /* If no message has arrived */
            prptr->prstate = PR_RECV;
            resched();                  /* Block until message arrives */
        }
        msg = prptr->prmsg;             /* Retrieve message             */
        prptr->prhasmsg = FALSE;        /* Reset message flag           */
        restore(mask);
        return msg;
}
```

Determine whether a message is waiting

Xinu Code for Clearing Messages

```
/* recvclr.c - recvclr */

#include <xinu.h>

/*------------------------------------------------------------------------
 * recvclr - Clear incoming message, and return message if one waiting
 *------------------------------------------------------------------------
 */
umsg32 recvclr(void)
{
        intmask mask;                   /* Saved interrupt mask        */
        struct procent *prptr;          /* Ptr to process' table entry */
        umsg32 msg;                     /* Message to return           */

        mask = disable();
        prptr = &proctab[currpid];
        if (prptr->prhasmsg == TRUE) {
            msg = prptr->prmsg;         /* Retrieve message            */
            prptr->prhasmsg = FALSE;    /* Reset message flag          */
        } else {
            msg = OK;
        }
        restore(mask);
        return msg;
}
```

Summary

- Inter-process communication
  - Implemented by message passing
    - A low-level mechanism provides direct communication among processes
    - A high-level mechanism uses rendezvous points.
  - Can be synchronous or asynchronous
- Low-level mechanism in Xinu:
  - Limits the message size to a single word
  - Restricts each process to at most one outstanding message
  - Use first-message semantics
- Synchronous interface is the simplest
- Xinu uses synchronous reception and asynchronous transmission