# 7 Message Passing

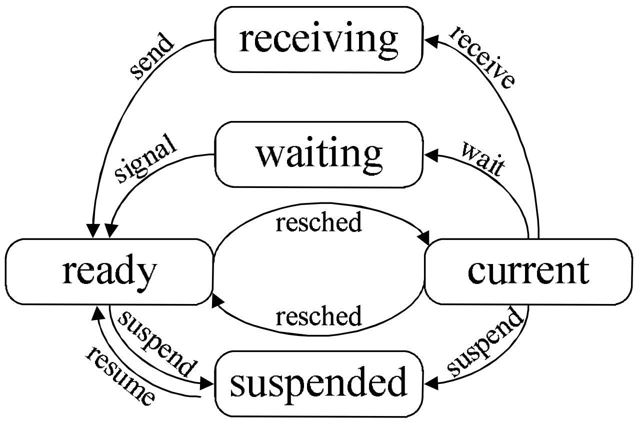
- **Inter-process communication** is an important mechanism in operating systems. Example from Unix: shell-pipeline – IPC between user processes
- Several mechanisms exist:
  - **semaphores** for coordination/synchronization (see chapter 6)
  - **monitors**: mechanism for obtaining mutual exclusion; high-level programming language constructs, similar to some kind of abstract data type or object (no ordinary system call, see [Hoa74], [BrinchHan75])
  - data transmission by using **shared variables**
  - **message passing**: synchronization and data transmission

## 7.1 Message Passing in Xinu

- Message passing is a form of **inter-process communication** and **process synchronization and -coordination** (alternative: shared variables).
- Advantage to synchronization by using semaphores: it can be **asynchronous**!
- Xinu's implementation is using system calls `send` and `receive`.
- Several variants of message passing are possible:
  - **Direct** message passing
  - Message sending and receiving can be **blocking** or **unblocking**
  - **Rendezvous**: send and receive are blocking
  - **Capacity** of the binding (buffer size): what would happen, if buffer is full?
  - Messages of **determined** or **variable size**
  - Is **more than one receiver** possible?
  - **Specified** sending/receiving process
  - ....

# Message Passing in Xinu

- In XINU there are two forms of message passing: either process-to-process (**direct**) or the message are left at **rendezvous** points (see chapter 14).
- System calls in XINU:
  - `send` (asynchronous) takes a message and a process id as arguments and delivers the message to the specified process,
  - `receive` (synchronous) waits for the message to arrive,
  - `recvclr` (asynchronous) never waits for a message to arrive.
- A new process state is needed: **receiving** (`PRREC`).



## Message Passing in Xinu

- Receiving has a **buffer** of size **1** (= one word), so only the **first message** is received and **all other are lost** because sender does not block.
- **Storage place** for a message: in the **process table** entry – not in the **sender's memory** because the sending process might exit before the message is received, but neither in the **recipient's memory** because it poses a security threat!

```
/* proc.h   see page 55 */

...
struct  pentry {
        char     pstate;   /* process state                  */
        ...
        short    pmsg;     /* 1 message sent to this process */
        short    phasmsg;  /* nonzero => msg is valid        */
        ...
};
...
```

## 7.2 Implementation of `send`

- During `send`, interrupts are disabled. Uses the process id to access corresponding process table entry
- **Errors** occur when an **invalid process id** is used or the receiving **buffer is full**.
- Else sending a message consists of **passing the message** and setting the **flag** (`phasmsg`) – if the recipient is **waiting** for a message, then reschedule by calling `ready()`.

# Implementation of `send`

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

#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*------------------------------------------------------------------------
 *  send  --  send a message to another process
 *------------------------------------------------------------------------
 */
SYSCALL send(pid, msg)
int     pid;
int     msg;
{
        struct  pentry  *pptr;          /* receiver's proc. table addr. */
        char    ps;

        disable(ps);
        if (isbadpid(pid) || ( (pptr= &proctab[pid])->pstate == PRFREE)
           || pptr->phasmsg) {
                restore(ps);
                return(SYSERR);
        }
        pptr->pmsg = msg;               /* deposit message              */
        pptr->phasmsg = TRUE;
        if (pptr->pstate == PRRECV)     /* if receiver waits, start it  */
                ready(pid, RESCHYES);
        else if (pptr->pstate == PRTRECV) {
                unsleep(pid);
                ready(pid, RESCHYES);
        }
        restore(ps);
        return(OK);
}
```

## 7.3 Implementation of `Receive`

- **Asynchronous receiving** is implemented by `recvclr`: if the process has a message, `recvclr` returns it, else it returns `OK`:

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

#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*------------------------------------------------------------------------
 *  recvclr  --  clear messages, returning waiting message (if any)
 *------------------------------------------------------------------------
 */
SYSCALL recvclr()
{
        char    ps;
        int     msg;

        disable(ps);
        if ( proctab[currpid].phasmsg ) {       /* existing message?    */
                proctab[currpid].phasmsg = FALSE;
                msg = proctab[currpid].pmsg;
        } else
                msg = OK;
        restore(ps);
        return(msg);
}
```

# Implementation of `Receive`

- **Synchronous receiving** is implemented by `receive`
  – difference to asynchronous receive: if the process has no message, the receiving process changes into the receiving state and calls **`resched`**.

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

#include <conf.h> #include <kernel.h> #include <proc.h>

/*------------------------------------------------------------------------
 *  receive  -  wait for a message and return it
 *------------------------------------------------------------------------
 */
SYSCALL receive() {
        struct  pentry  *pptr;
        int     msg;
        char    ps;

        disable(ps);
        pptr = &proctab[currpid];
        if ( !pptr->phasmsg ) {         /* if no message, wait for one  */
                pptr->pstate = PRRECV;
                resched();
        }
        msg = pptr->pmsg;               /* retrieve message             */
        pptr->phasmsg = FALSE;
        restore(ps);
        return(msg);
}
```