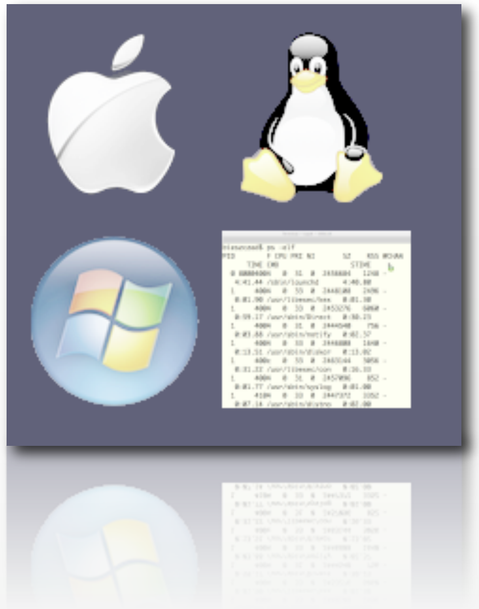


Chapter 4: Inter-Process Communication (IPC)

COMP362 Operating Systems
Prof. AJ Bieszczad

Outline: Processes

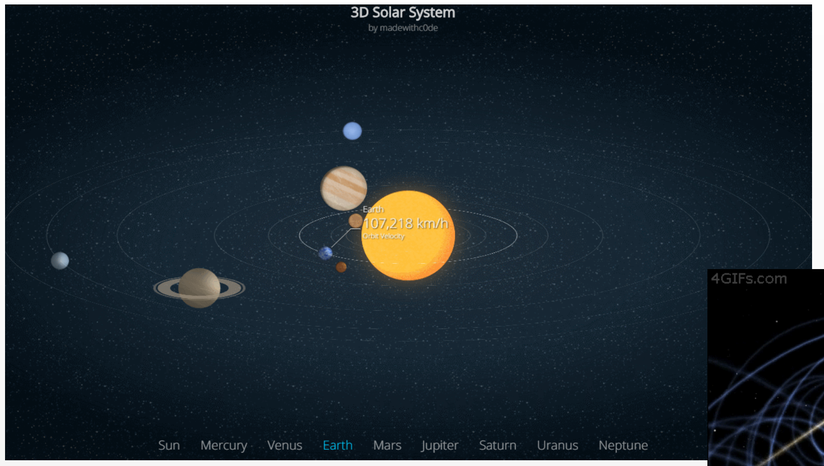
- Cooperating Processes
- Main IPC Models
- Client-Server Systems
- Shared Memory-based IPC
- Kernel-facilitated IPC
- Direct Communication
- Indirect Communication
- Management of IPC Resources

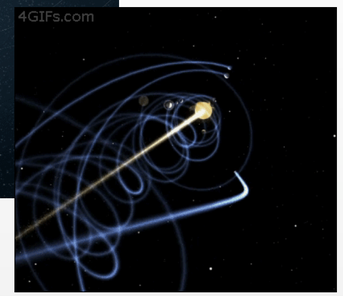
Interprocess Communication (IPC)

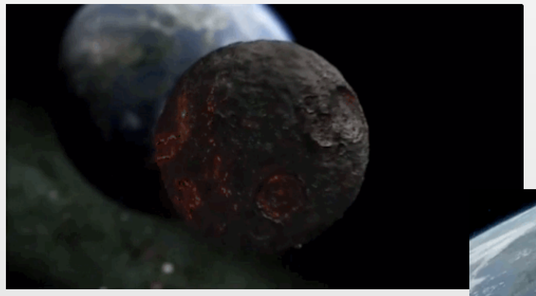
- We may want to use interprocess communication for:
  - **Information sharing**
    - files, shared memory
  - **Computation speedup**
    - e.g., on multi-core processors
  - **Modularity**
    - software engineering
  - **Convenience**
    - e.g., signaling events in parallel activities, in workflows, etc.
      - for example, printing, compiling, building, etc. -- user may stayed informed while doing something else

Example: Dynamic Systems

- Multiple processes collaborate to solve a certain problem
- For example, would we survive if a foreign celestial body enters the system and hits one of the native solar system objects?
  - the gravitational attraction between every pair of celestial bodies is proportional to the distance between them, and vice versa
  - therefore, changing the position of any object disturbs the whole system
- A process may represent a celestial body
  - it receives positions from other objects, calculates the gravitational forces, then its new position, and then sends it back to the others
  - the information is exchanged as long as the position of any object keeps changing
  - the system is considered stable when no object changes the position

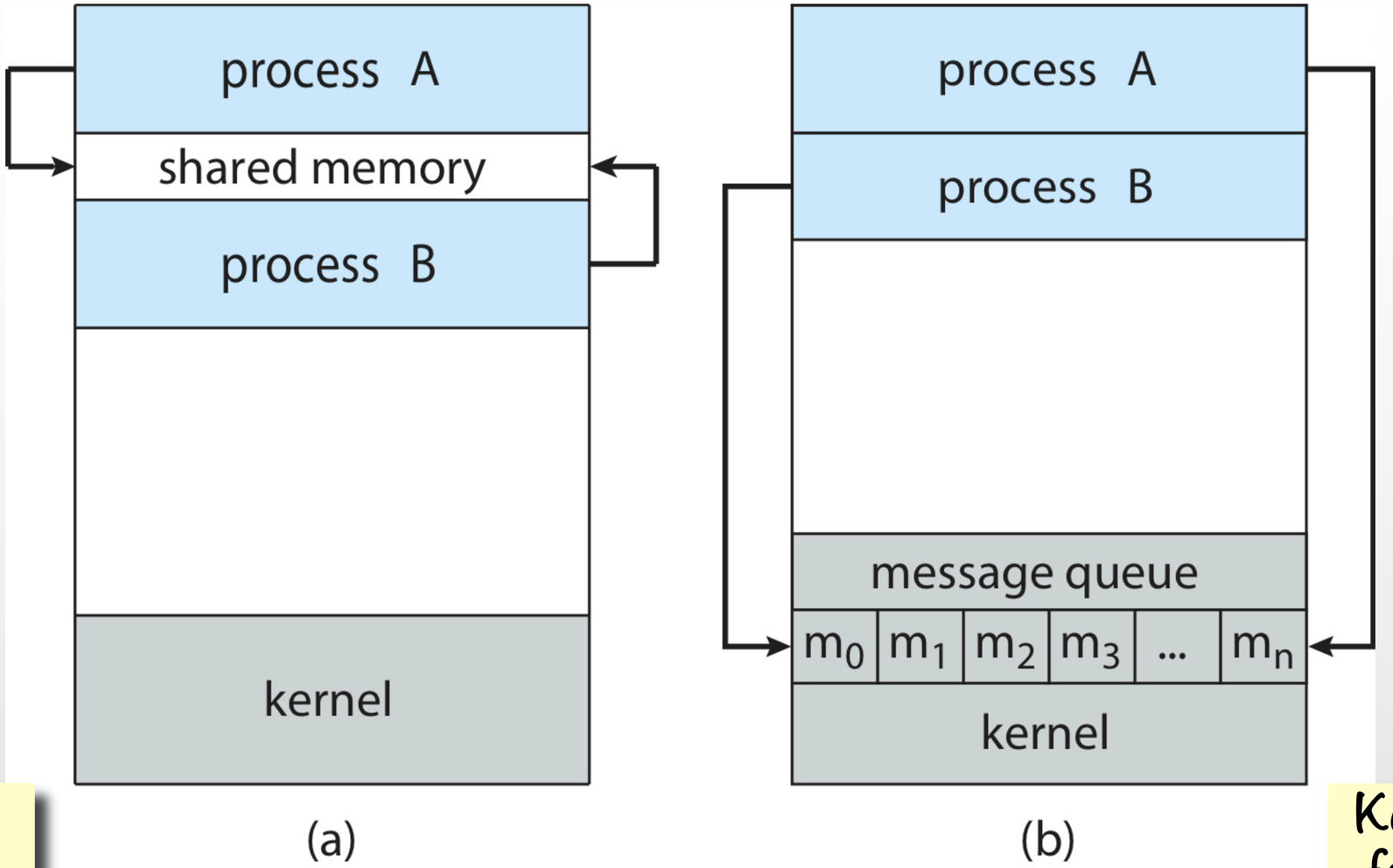






This is a very simplified model!

Two Main IPC Models

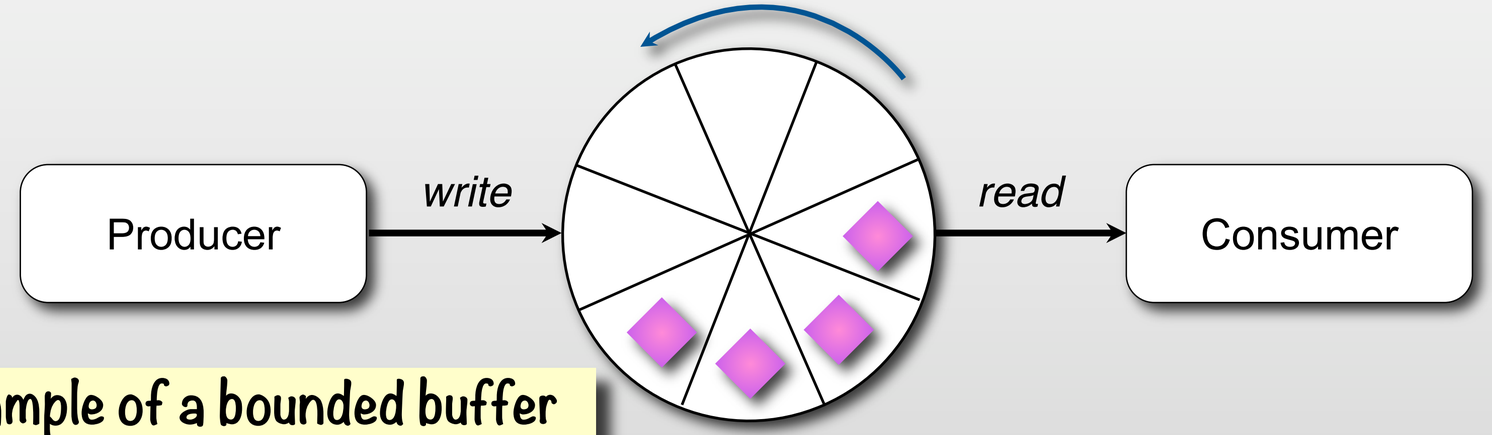


No kernel involvement after the shared space is allocated

Kernel actively facilitates the communication

Shared Memory: Producer-Consumer Paradigm

- Common paradigm for cooperating processes:
  - **producer process** produces information that is consumed by a **consumer process**
- The processes use a buffer:
  - **unbounded-buffer** places no practical limit on the size of the buffer
    - available memory is the only limit
  - **bounded-buffer** assumes that there is a fixed buffer size



An example of a bounded buffer

UNIX Shared Memory

- C Standard Library in UNIX:

Time is sent in this example from "time server" to "time client"

server

```
int main() {
    long now; // time now
    int fd; // shared memory file descriptor
    char *mem_ptr; // pointer to shared memory object

    // create the shared memory object
    if ((fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, S_IWUSR | S_IRUSR)) == -1)
       oops("CANNOT OPEN SHM", EXIT_FAILURE);

    ftruncate(fd, SHM_SIZE); // cut down to the desired size

    // memory map the shared memory object
    mem_ptr = (char *) mmap(0, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    // run a clock for a minute
    printf("Time ticks: ");
    for (int i = 0; i < 60; i++)
    {
        fflush(stdout);
        time(&now);                     // get the time
        strcpy(mem_ptr, ctime(&now)); // write to memory
        sleep(1);                       // wait a sec
        printf(".");
    }
    printf("\n");

    shm_unlink(SHM_NAME); // remove the shared memory object

    return EXIT_SUCCESS;
}
```

shm_POSIX_time_server.c

```
#define SHM_NAME "MY_SHM"
#define SHM_SIZE 64
#define oops(m,x)  { perror(m); exit(x); }
```

shm_POSIX.h

client

```
int main()
{
   int fd; // shared memory file descriptor
   char *mem_ptr; // pointer to shared memory obect

   // open the shared memory object
   if ((fd = shm_open(SHM_NAME, O_RDONLY, 0400)) == -1)
       oops("CANNOT OPEN SHM", EXIT_FAILURE);

   // memory map the shared memory object
   mem_ptr = (char *) mmap(0, SHM_SIZE, PROT_READ, MAP_SHARED, fd, 0);

// read from the shared memory object
   printf("TIME NOW: %s\n", mem_ptr);

   munmap(mem_ptr, SHM_SIZE);

   return EXIT_SUCCESS;
}
```

shm_POSIX_time_client.c

Message System: Overview

- Message system allows processes to communicate with each other without resorting to shared memory (shared variables)

- Kernel provides mechanisms to facilitate the communication
  - Message passing facility provides two operations:
    - **send**
    - **receive**

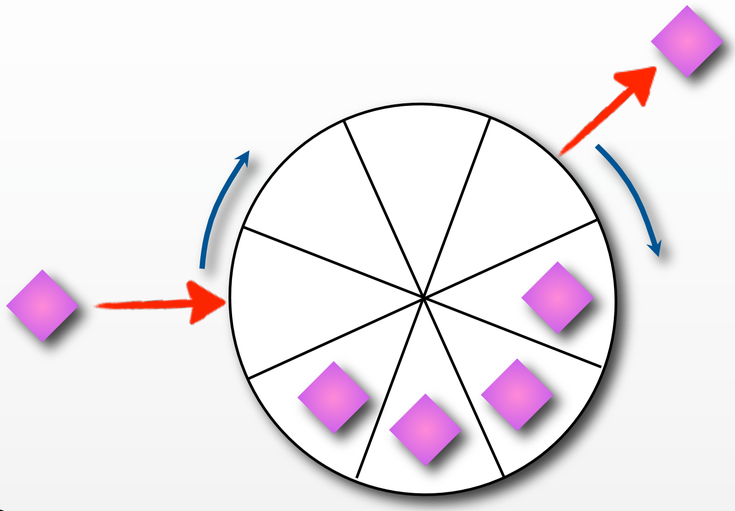
  - If processes P and Q wish to communicate, they need to **establish a communication link** between them before exchanging messages via send/receive

Message System: Implementation Challenges

- Designers of a message system must decide:
  - how are links established
  - whether a link can be associated with more than two processes
  - the number of links that can there be between every pair of communicating processes
  - the capacity of a link
  - whether the size of a message that the link can accommodate is fixed or variable
  - whether the link is unidirectional or bi-directional
  - whether the link is direct or indirect
  - whether the link is synchronous or asynchronous
  - whether the link buffering is automatic or explicit

Message System: Buffering

- Queue of messages attached to the link
- Implemented in one of three ways
  - **zero capacity** – 0 messages
    - sender must wait for receiver
  - **bounded capacity** – finite length of *n* messages
    - sender must wait if link full
  - **unbounded capacity** – infinite length
    - sender never waits
    - available memory is the practical limit however



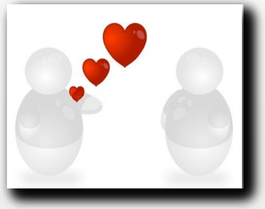
Sample illustration of a bounded buffer with two revolving indices for send (in; green) and receive (out; red)

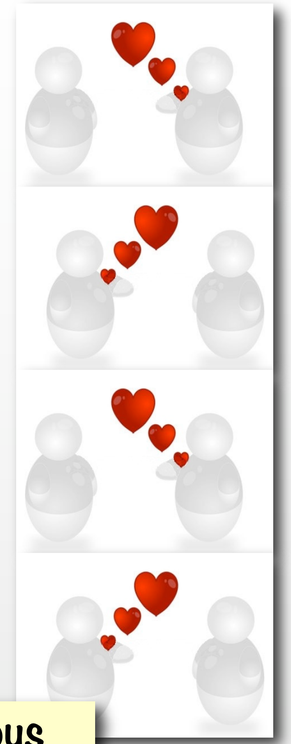
Message System: Synchronization

- Message passing may be either **blocking** or **non-blocking**

- **Blocking** is considered synchronous
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
    - Blocked processes waits in the waiting queue

  - **Rendezvous**
    - communication between blocking sender and receiver





randezvous

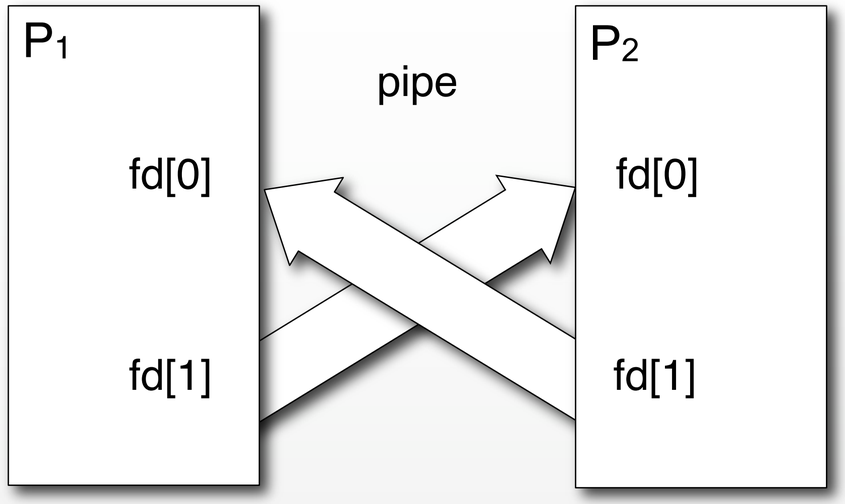
- **Non-blocking** is considered asynchronous
  - **Non-blocking send** has the sender send the message and continue
  - **Non-blocking receive** has the receiver receive a valid message or null (if there is no message)

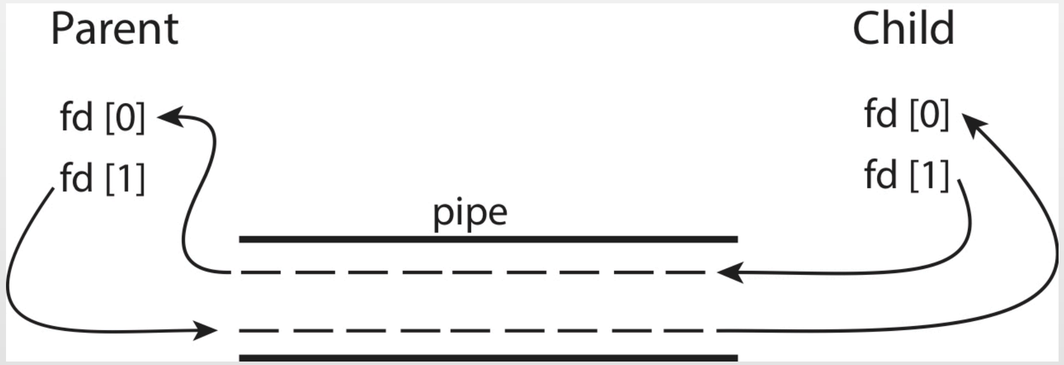
Message System: Direct Communication

- Processes must name each other explicitly:
  - **send**(P, message) – send a message to process P
  - **receive**(Q, message) – receive a message from process Q
- Properties of the direct communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional
  - The communication can be synchronous or asynchronous
    - “blocking” or “non-blocking”

It’s like using an instant messaging system or a phone (with some “call-me-back” stretch for prompting messages from peers)

UNIX (Unnamed) Pipes - Direct IPC





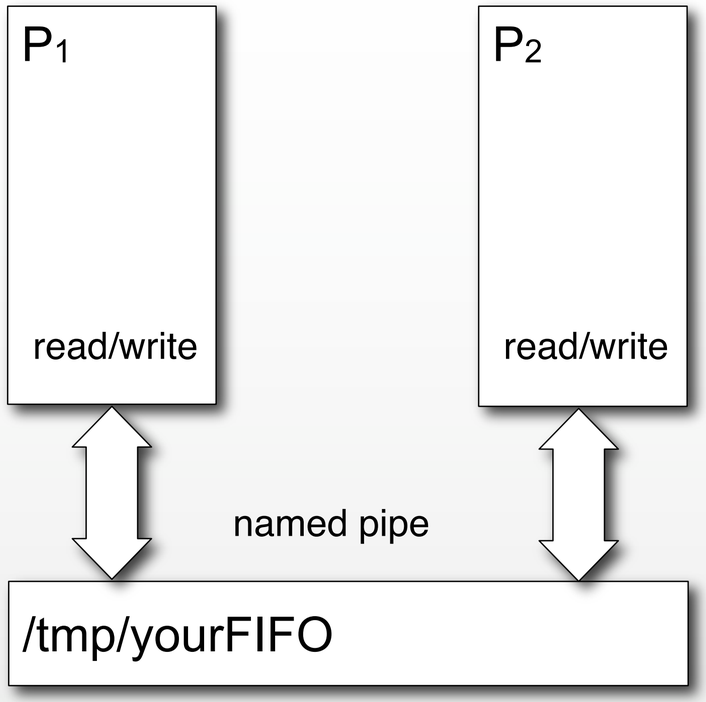
pipe.c

```
int         n, fd[2];
pid_t       pid;
char        line[MAXLINE];
char        line2[MAXLINE];

if (pipe(fd) < 0)
    printf("Cannot create a pipe.");
else if ( (pid = fork()) < 0)
    printf("Cannot fork.");
else if (pid == 0) /* child */
{
    write(fd[1], "Message for the parent\n", 23);
    n = read(fd[0], line, MAXLINE);
    printf("%s", line);
    close(fd[1]);
}
else /* parent */
{
    write(fd[1], "Message for the child\n", 22);
    n = read(fd[0], line2, MAXLINE);
    printf("%s", line2);
    close(fd[0]);
    wait(NULL);
}
```

See also `pipe1.c` and `pipe2.c`

UNIX Named Pipes (FIFOs)




```
if (stat = mkfifo("/tmp/ajsFIFO", 0777) < 0)
   oops("Cannot make FIFO", stat);
// open a named pipe
pipe = open("/tmp/ajsFIFO", O_WRONLY);
while(1) {
     // get a line to send
     printf("Enter line: ");
     fgets(line, MAX_LINE, stdin);
     *strchr(line, '\n') = '\0';
     if (strcmp(line, "quit") == 0)
        break;
     // actually write out the data and close the pipe
     write(pipe, line, strlen(line) + 1);
}
// close the pipe
close(pipe);
```

O_RDWR may be needed

npipe_send.c

Setting / clearing flags

NOTE: if `flag == 00100000`
then `~flag == 11011111`

SET: `11001100 | flag ==> 11101100`

CLEAR: `11101100 & ~flag ==> 11001100`

```
// open a named pipe
pipe = open("/tmp/ajsFIFO", O_RDONLY);
// set the mode to nonblocking
int flags;
flags |= O_NONBLOCK;
//flags &= ~O_NONBLOCK; // default
fcntl(pipe, F_SETFL, flags);
// read the data from the pipe
read(pipe, line, MAX_LINE);
printf("Received line: %s\n", line);
// close the pipe
close(pipe);
```

npipe_receive.c

Message System: Indirect Communication

- Messages are directed to and received from **mailboxes**
  - also referred to as **ports**
    - each mailbox has a unique id
    - processes can communicate only if they share a mailbox
- Properties of the indirect communication link
  - link established only if processes share a common mailbox
  - a link may be associated with many processes
  - each pair of processes may share several communication links
  - link may be unidirectional or bi-directional
- Operations
  - **create** a new mailbox
    - ...and tell others what it is
  - **send** and **receive** messages through mailbox
  - **destroy** a mailbox
- Primitives are defined as:
  - **send**(A, message) – send a message to mailbox A
  - **receive**(A, message) – receive a message from mailbox A

➢ It's like having multiple email accounts and reading them at your own leisure
➢ No one can send anything to you without knowing your address

Message System: Indirect Communication *(cont.)*

- Mailbox sharing dilemma
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?

- Possible solutions
  - constrain links to be associated with at most two processes
    - every message has one sender and one receiver
  - allow only one process at a time to execute a receive operation
  - allow the system to select arbitrarily the receiver
    - sender is notified who the receiver was
  - more complex schema could be used

UNIX POSIX Message Queues: Sender

```c
#include "ipc_POSIX_messg.h"

int main() {
   mqd_t msqid;
   unsigned int type;

   char *msg = calloc(MSG_SIZE, sizeof(char)); // put zeros in the allocated space
   strcpy(msg, "Hello receiver!");

   if ((msqid = mq_open(QUEUE_NAME, O_RDWR | O_CREAT, S_IRWXU | S_IRWXG, NULL)) < 0)
      oops("Error opening a queue.", errno);

   if (mq_send(msqid, msg, strlen(msg) + 1, MSG_TYPE) < 0) // type 1 (arbitrarily)
      oops("Error opening a queue.", errno);

   printf("Sent: \"%s\"\nRun the receiver.\n", msg);

   printf("To continue, press Enter on the keybord...");
   getchar(); // just to stop and wait for the receiver to respond

   if (mq_receive(msqid, msg, MSG_SIZE, &type) < 0)
      oops("Error receiving data.", errno);

   printf("Received: \"%s\" of type %d\n", msg, type);

   mq_close(msqid);
   mq_unlink(QUEUE_NAME);

   exit(EXIT_SUCCESS);
}
```

ipc_POSIX_send.c

```c
#ifndef _IPC_MESSG_H
#define _IPC_MESSG_H

#include <sys/types.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>

#define oops(ermsg,erno)
   {perror(ermsg); exit(erno); }

#define QUEUE_NAME "/myQueue"

#define MSG_SIZE 8192

#define MSG_TYPE 1

#endif
```

ipc_POSIX_messg.h

POSIX Message Queues: Receiver

```c
#include "ipc_POSIX_messg.h"

int main()
{
   mqd_t msqid;
   char msg_rcvd[MSG_SIZE], msg_send[MSG_SIZE];
   unsigned int type;

   if ((msqid = mq_open(QUEUE_NAME, O_RDWR)) < 0) // assuming it already exists
      oops("Error opening a queue.", errno);

   if (mq_receive(msqid, msg_rcvd, MSG_SIZE, &type) < 0)
      oops("Error receiving data.", errno);

   printf("Received: \"%s\" of type %d\n", msg_rcvd, type);

   sprintf(msg_send, "Sender, thank you for your message \"%s\"", msg_rcvd);

   if (mq_send(msqid, msg_send, MSG_SIZE, MSG_TYPE) < 0)
      oops("Error responding.", errno);

   printf("Sent back: \"%s\"\n", msg_send);

   mq_close(msqid);

   exit(0);
}
```

ipc_POSIX_receive.c

```c
#ifndef _IPC_MESSG_H
#define _IPC_MESSG_H

#include <sys/types.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>

#define oops(ermsg,erno)
   {perror(ermsg); exit(erno); }

#define QUEUE_NAME "/myQueue"

#define MSG_SIZE 8192

#define MSG_TYPE 1

#endif
```

ipc_POSIX_messg.h

System Management of IPC Resources

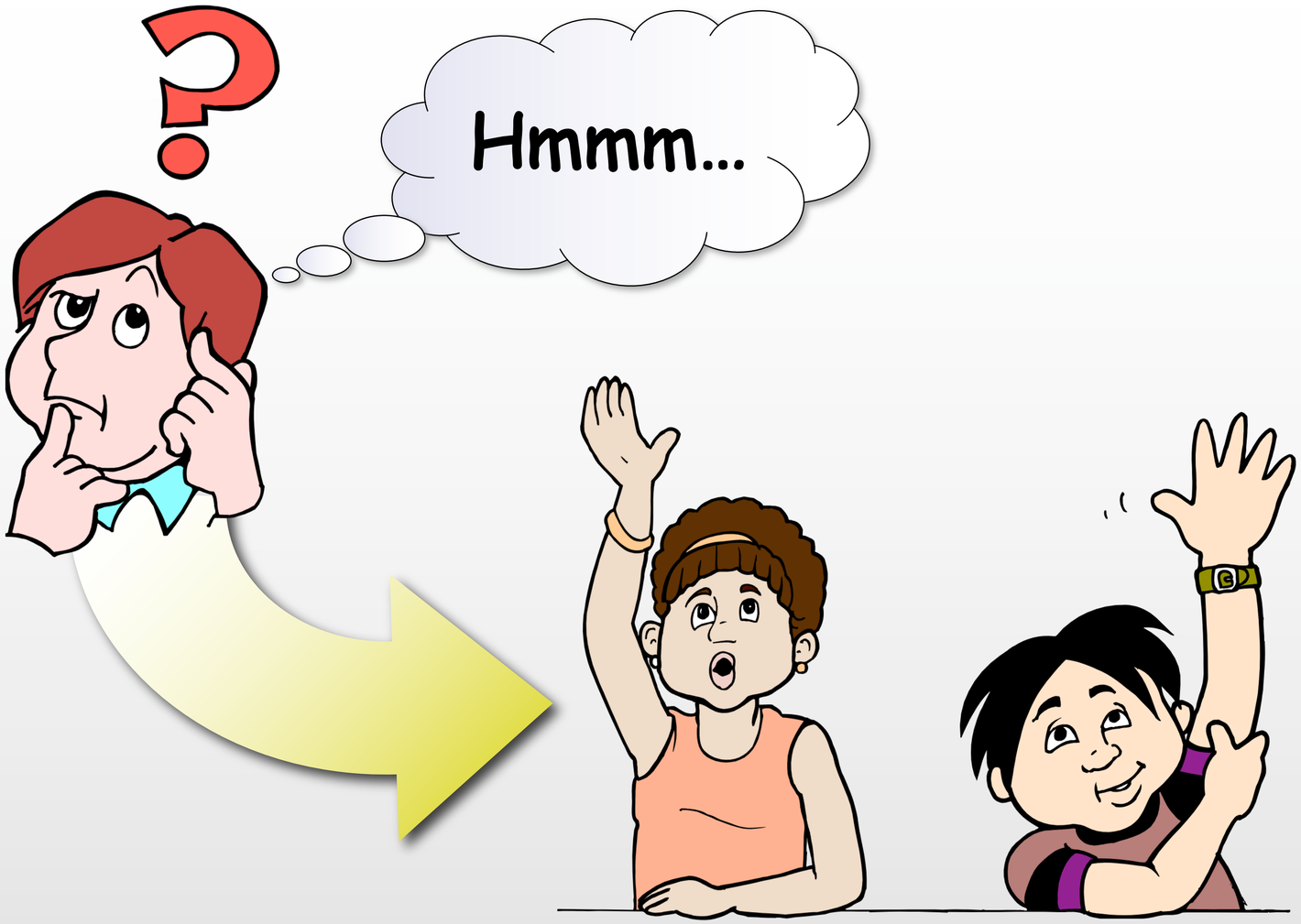
- IPC resources are assigned to the user, and not a process, so they are not returned automatically

- **ipcs** and **ipcrm** are command that can be used to list and remove IPC resources

```
$ man ipcs
$ man ipcrm
$
$ ipcs
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 65536      aj         600        524288     2          dest
0x00000000 524289     aj         600        524288     2          dest
0x00000000 425986     aj         600        524288     2          dest
0x00000000 294915     aj         600        16777216   2
0x00000000 851972     aj         600        134217728  2          dest
0x00000000 950277     aj         600        524288     2          dest
0x00000000 655366     aj         600        524288     2          dest
0x00000000 819207     aj         600        524288     2          dest
0x00000000 1048584    aj         600        524288     2          dest
0x00000000 1146889    aj         600        524288     2          dest
0x00000000 1179658    aj         600        536870912  2          dest
0x00000000 4128779    aj         600        393216     2          dest
0x00000000 5570572    aj         600        393216     2          dest
0x00000000 6848525    aj         600        67108864   2          dest

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0xd20042e8 65536      aj         666        1
0xd20042ea 98305      aj         666        1
0xd20042e9 131074     aj         666        1
0xd200430f 163843     aj         666        1
0xd20042eb 196612     aj         666        1
```



COMP362 Operating Systems

Prof. AJ Bieszczad