W11: INTER-PROCESS COMMUNICATION

CSCE 313 Spring 2017

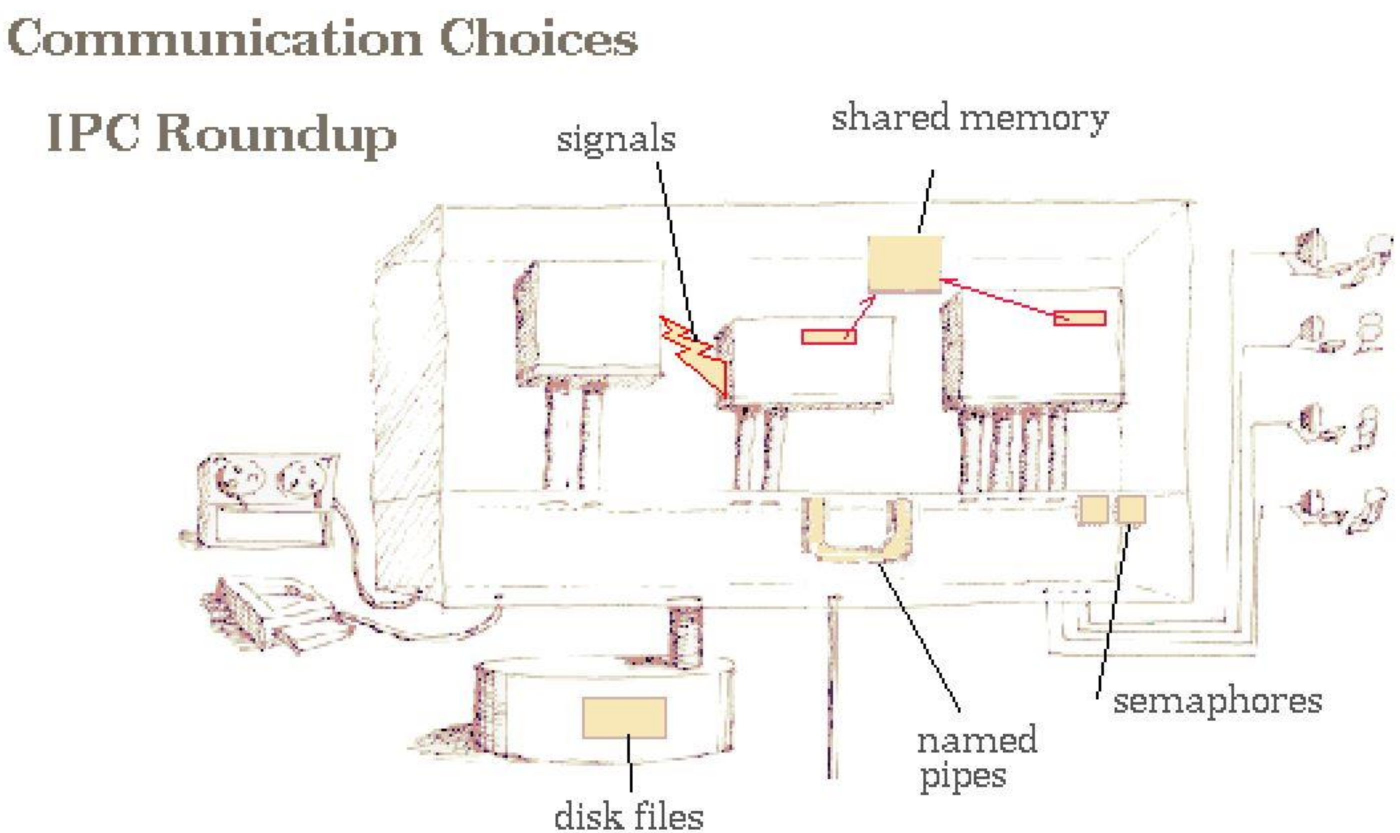
Inter-Process Communication

- IPC classes
  - Pipes and FIFO
  - Message Passing
  - Shared Memory
  - Semaphore Sets
  - Signals
- References:
  - Baseline slides: CSCE-313 Spring'17 Ahmed, CSCE-313 Spring'16 Tyagi & Bettati, and Gu
  - Understanding Unix/Linux Programming, Bruce Molay, Chapters 10, 15
  - Advanced Linux Programming Ch 5
  - Some material also directly taken or adapted with changes from Illinois course in System Programming (Prof. Angrave), UCSD (Prof. Snoeren), and USNA (Prof. Brown)

Inter-Process Communication (IPC)

- A process contains everything needed for execution
  - An address space (defining all the code and data)
  - OS resources (e.g., open files) and accounting information
  - Execution state (PC, SP, registers, etc.)
  - Each of these resources is exclusive to the process
- Yet sometimes processes may wish to cooperate (information sharing, performance, modularity, etc.)
  - But how to communicate? Each process is an island
  - The OS needs to intervene to bridge the gap
  - OS provides system calls to support Inter-Process Communication (IPC)

Inter-Process Communication Landscape



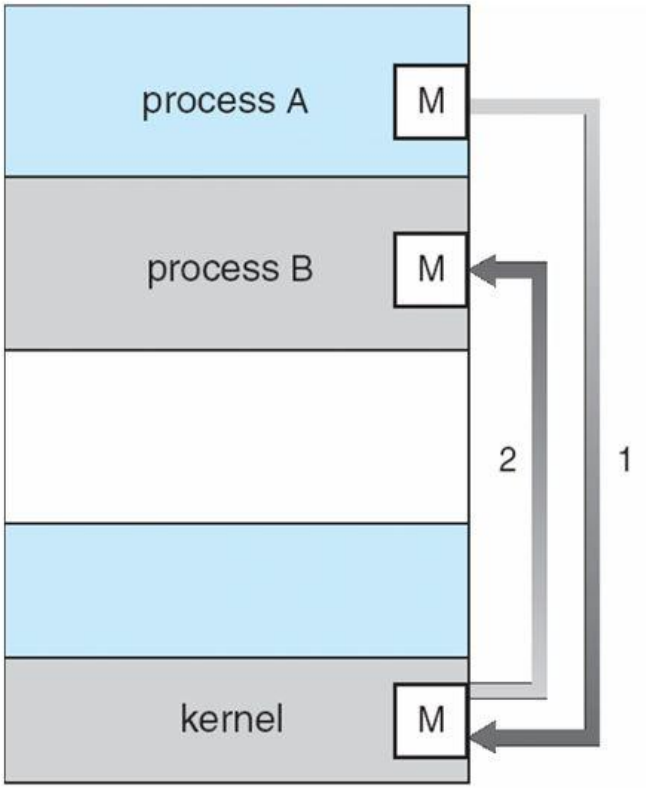
- Rendering from Prof. Farrell (Kent State University)

IPC Motivation

- We have come to know that processes have a limited ability to pass data
  - Parents get one chance to pass everything at fork()
  - But what if the child wants to talk back? What about processes with different ancestry?

IPC at a Glance – Explicit Channel

- <u>Un-named Pipes and Named Pipes (FIFO)</u>
  - Builds a channel between processes and exchange data by reading/writing from/to file descriptors
  - Explicit communication channel

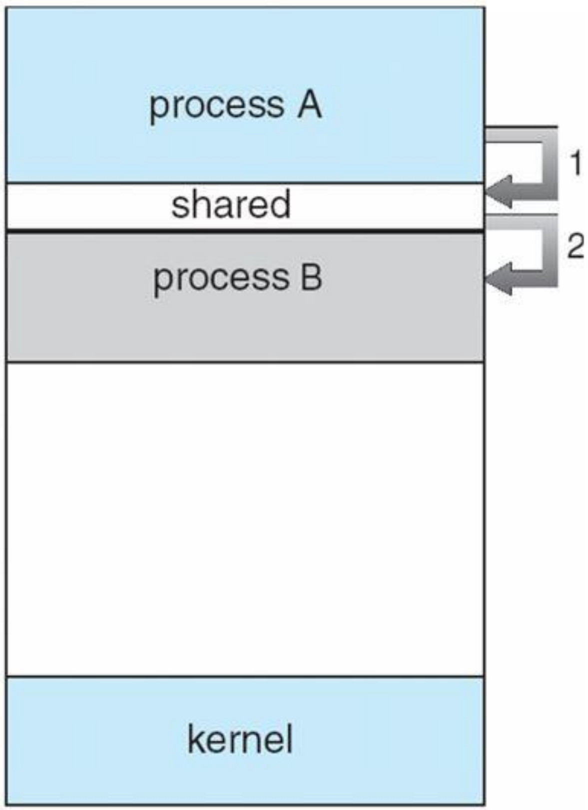


IPC at a Glance – Explicit Channel

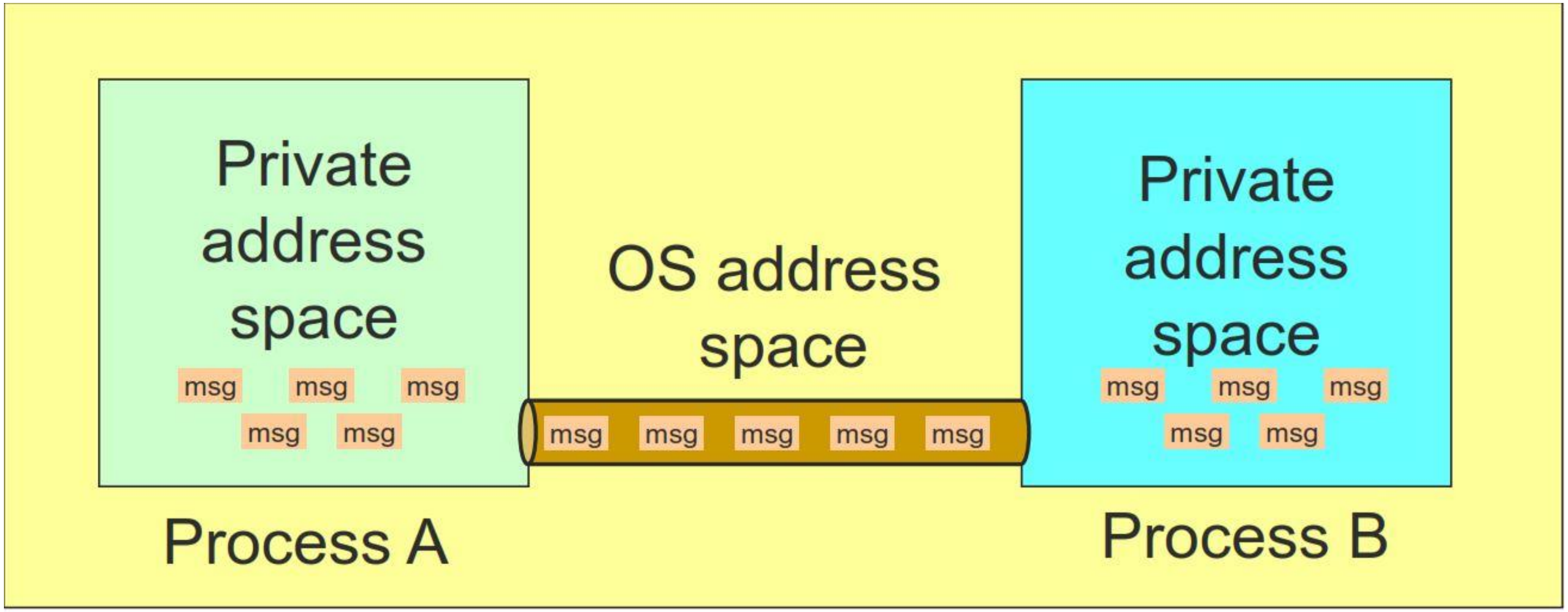
- **Message Passing**: *explicit communication channel provided through* send()/receive() system calls
  - A system call is required
  - Explicit channel

IPC at a Glance – Implicit Channel

- **<u>Shared Memory</u>**: multiple processes can read/write same physical portion of memory; *implicit channel*
  - Implicit channel
  - System call to declare shared region of memory
  - No OS mediation required once memory is mapped



Communication Over a Pipe



Unix Pipes (aka Unnamed Pipes)

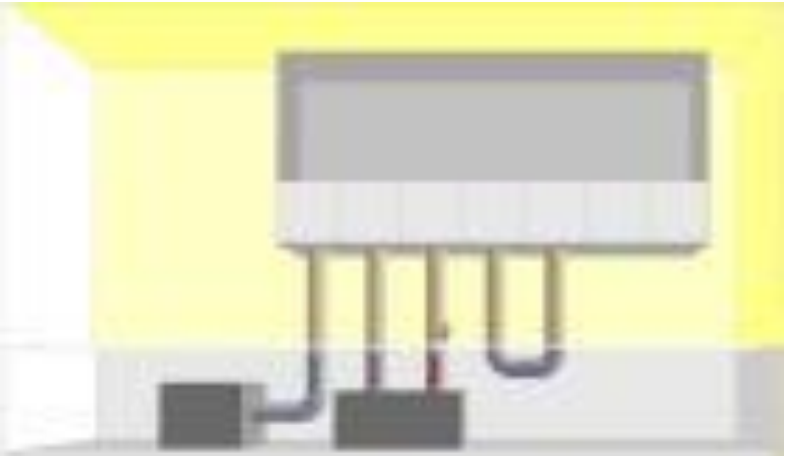
- **#include <unistd.h>**
- **int pipe(int fildes[2]);**
- Returns a pair of file descriptors
  - **fildes[0] is connected to the read end of the pipe**
  - **fildes[1] is connected to the write end of the pipe**
- **Create a message pipe**
  - Anything can be written to the pipe, and read from the other end
  - Data is received in the order it was sent
  - OS enforces mutual exclusion: only one process at a time
  - Accessed by a **file descriptor**, like an ordinary file
  - Processes sharing the pipe must have same parent in common
  - Processes communicating via pipes *must be running on the same host*

Pipe Creation

BEFORE pipe

Process has some usual files open

AFTER pipe



Kernel creates a pipe and sets file descriptors

- BEFORE
  - Shows standard set of file descriptors
- AFTER
  - Shows newly created pipe in the kernel and the two connections to that pipe in the process

IPC Pipe - Method


```c
#include <stdio.h>
#include <unistd.h>

void main ()
{
        char buf [10];
        int fds [2];
        pipe (fds);
        printf ("sending msg: Hi\n");
        write (fds[1], "Hi", 3);
        read (fds[0], buf, 3);
        printf ("Received msg: %s\n", buf);

}
```

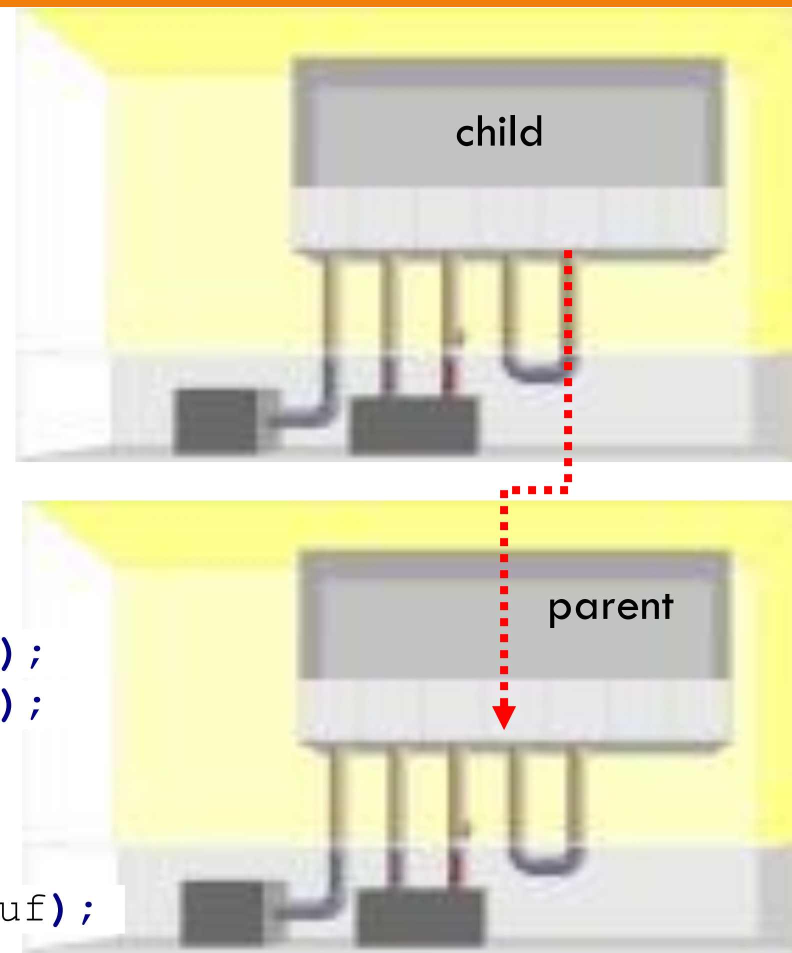
Connects the two fds as pipe

```
compute-linux1 tanzir/code> ./a.out
sending msg: Hi
Received msg: Hi
```

- Is this of any use at all ???

Pipe Between Two Processes

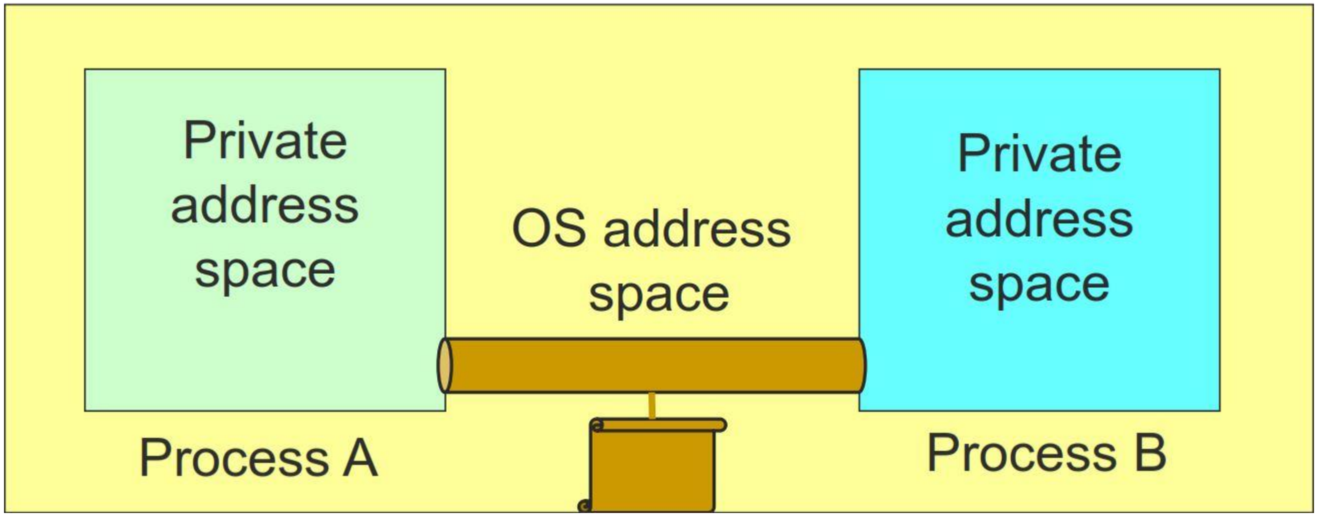

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/stat.h>
#include <fcntl.h>
int main ()
{
   int fds [2];
   pipe (fds); // connect the pipe
   if (!fork()){ // on the child side
      sleep (3);
      char * msg = "a test message";
      printf ("CHILD: Sending %s\n", msg);
      write (fds [1], msg, strlen(msg)+1);
   }else{
      char buf [100];
      read (fds [0], buf, 100);
      printf ("PARENT: Received %s\n", buf);
   }
   return 0;
}
```



IPC- FIFO (named PIPE)

FIFO

- A pipe disappears when no process has it open
- FIFOs (named pipes) are a mechanism that allow for IPC that's similar to using regular files, except that the **kernel takes care of synchronizing reads and writes**, and
- **Data is never actually written to disk** (instead it is stored in buffers in memory) so the overhead of disk I/O (which is huge!) is avoided.



FIFO vs PIPE

- A FIFO is like an unconnected garden hose lying on the lawn
  - Anyone can put one end of the hose to his ear and another person can walk up to the hose and speak into the other end
  - Unrelated people may communicate through a hose
  - Hose exists even if nobody is using it

FIFO

- **It’s part of the file system**
  - It has a name and path just like a regular file.
  - Programs can open it for reading and writing, just like a regular file.
  - However, the name is simply a convenient reference for what is actually just a stream of bytes - no persistent storage or ability to move backwards of jump forward in the stream.

FIFO

- **Works like a Bounded Buffer**
  - Bytes travel in First-In-First-Out fashion: hence the name FIFO.
  - Special Cases:
    - Read **Before** Write: Kernel puts the Reader process to sleep until data is available to read.
    - **Full Buffer**: Writer is put to sleep until a Reader process has read >=1 Byte

FIFO - Problems

- We still need to agree on a name ahead of time – how to communicate that??

```
RequestChannel*rc = new
RequestChannel("control", ..);
```

- Not concurrency safe
  - Like a file used by multiple processes/threads
  - Multiple Writers can cause a race condition

Using FIFO's

- How do I create a FIFO
  - mkfifo (name)
- How do I remove a FIFO
  - rm fifoname or unlink(fifoname)
- How do I listen at a FIFO for a connection
  - open (fifoname, O_RDONLY)
- How do I open a FIFO in write mode?
  - open(fifoname, O_WRONLY)
- How do two processes speak through a FIFO?
  - The sending process uses write and the listening process uses read. When the writing process closes, the reader sees end of file

FIFO DEMO

Writer

```
#define FIFO_NAME "test.txt"
int main(void)
{
  char s[300];
  int num, fd;
  mkfifo(FIFO_NAME, 0666); // create
  printf("Waiting for readers...\n");
  fd = open(FIFO_NAME, O_WRONLY); //open
  if (fd < 0)
    return 0;
  printf("Got a reader--type some 
stuff\n");
  while (gets(s)) {
    if (!strcmp (s, "quit")) break;
    if ((num = write(fd, s, strlen(s)))
== -1)
      perror("write");
    else
      printf("SENDER: wrote %d bytes\n", 
num);
  }
  //unlink (FIFO_NAME);
  return 0;
}
```

Reader

```
int main(void)
{
  char s[300];
  int num, fd;
  printf("waiting for writers...\n");
  fd = open(FIFO_NAME, O_RDONLY);
  printf("got a writer\n");
  do{
    if ((num = read(fd, s, 300)) == -1)
      perror("read");
    else {
      s[num] = '\0';
      printf("RECV: read %d bytes: 
\"%s\"\n", num, s);
    }
  } while (num > 0);
  return 0;
}
```

IPC: Message Passing

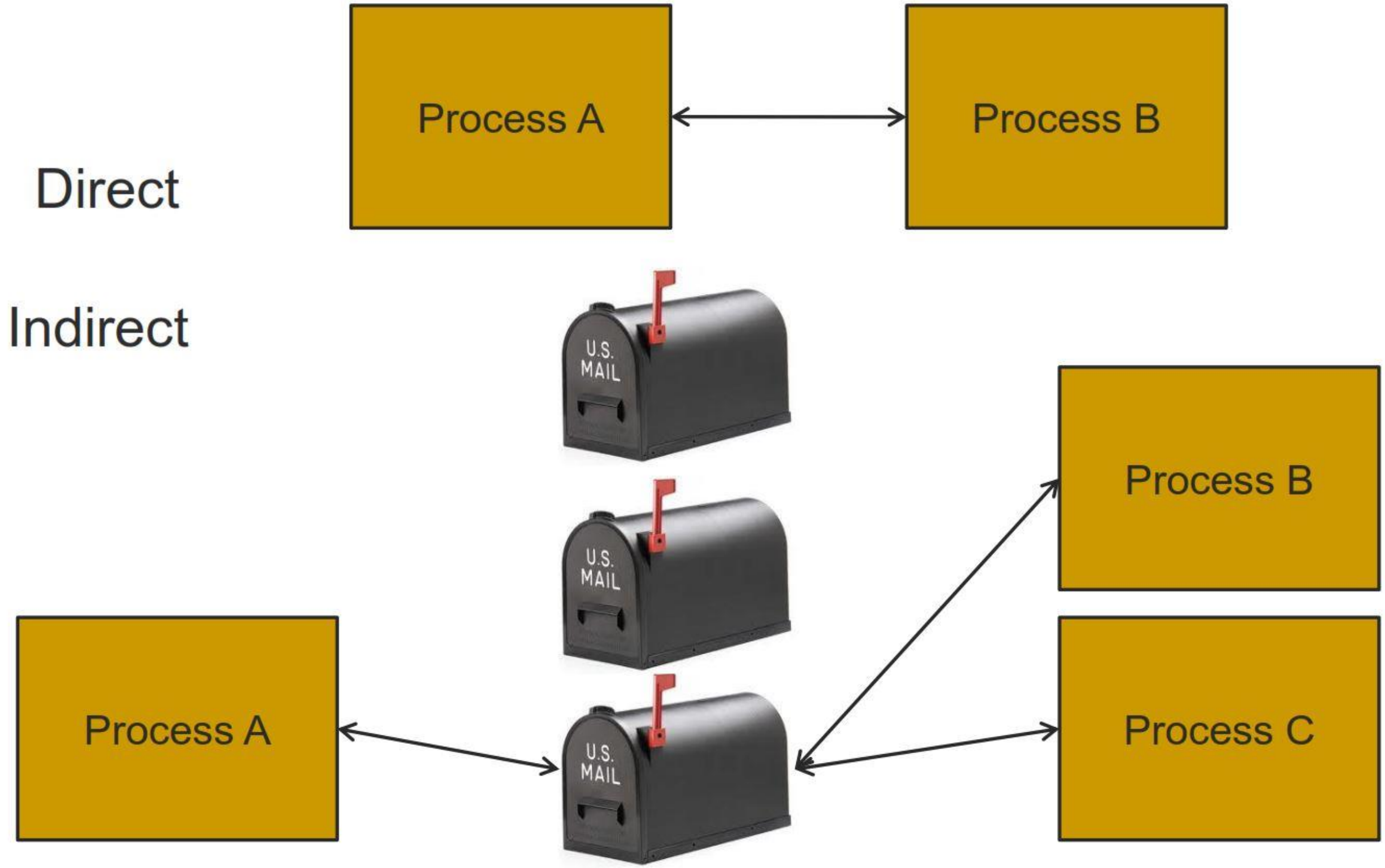
Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- IPC facility provides two operations:
  - **send**(*message*)
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive

Typical Implementation Questions

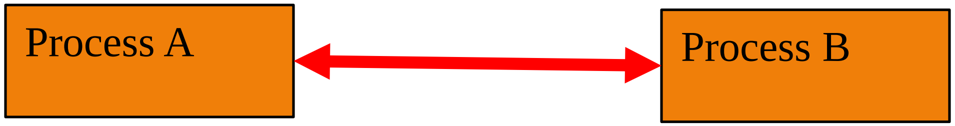
- How is a link established?
- Is a link unidirectional or bi-directional?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Can the message size be fixed or variable?

Message Passing



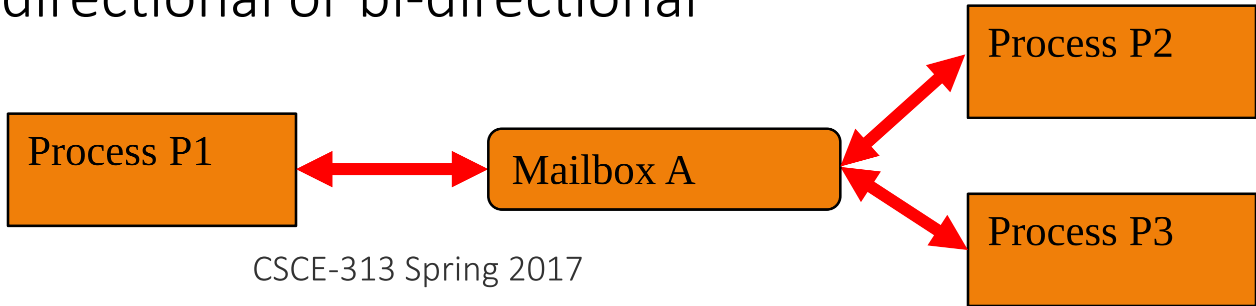
Direct Message Passing

- 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 communication link
  - Links are established automatically (or implicitly) while sending/receiving
  - 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
- Limitation: Must know the name or id of the process



Indirect Message Passing

- Messages are directed to and received from mailboxes (also referred to as ports)
  - Mailbox can be owned by a process or by the OS
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of 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

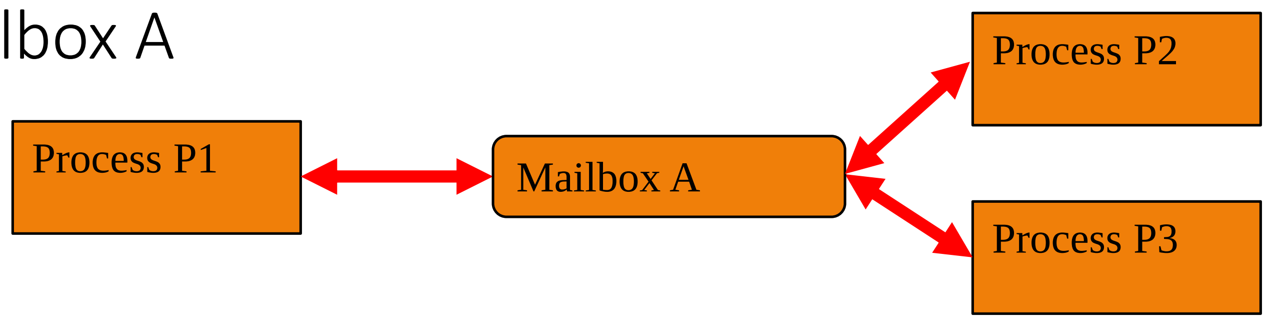


Indirect Message Passing

- Operations
  - create a new mailbox
  - 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



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
- **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

Buffering

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

IPC Object Creation: Message Queues

Object key identifies object across processes. Can be assigned as follows:
-- Create some unknown key
-- Pass explicit key (beware of collisions!)
-- Use file system to consistently hash key (using `ftok`)

```
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
/* create a message queue with given key and flags. */
```

Object id is similar to file descriptor.
-- It can be inherited across `fork()` calls.

Operations on Message Queues


```
#define PERMS (S_IRUSR | S_IWUSR)

int msqid;
if ((msqid = msgget(key, PERMS)) == -1)
    perror("msgget failed);
```

```
struct mymsg { /* user defined! */
    long msgtype;  /* first field must be a long identifier */
    char mtext[1]; /* placeholder for message content */
}
```

```
int msgsnd(int msqid, const void *msgp,
           size_t msgsz, int msgflg)

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz,
               long msgtyp, int msgflg);
```

msgtyp	action
0	remove first message from queue
> 0	remove first message of type `msgtyp` from the queue
< 0	remove first message of lowest type that is less than or equal to absolute value of `msgtyp`

Operations on Message Queues (cont.)


```
int msgctl(int msqid, int cmd, struct msgid_ds *buf)
```

Cmd	description
IPC_RMID	remove the message queue msqid and destroy the corresponding msqid_ds
IPC_SET	Set members of the msqid_ds data structure from buf
IPC_STAT	Copy members of the msqid_ds data structure into buf

Message Queue – Code Example


```c
struct my_msgbuf {
        long mtype;
        char mtext[200];
};
int sender(void)
{
   struct my_msgbuf buf;
   int msqid = msgget(654321, 0644 | IPC_CREAT); // create the msg queue
   while(fgets(buf.mtext, sizeof buf.mtext, stdin) != NULL) {
     int len = strlen(buf.mtext);
     msgsnd(msqid, &buf, len+1, 0);
   }
   msgctl(msqid, IPC_RMID, NULL);  // delete the msg queue
}
int receiver(void)
{
   struct my_msgbuf buf;
   int msqid = msgget(654321, 0644); // connect (not create)
   while(1) {
     msgrcv(msqid, &buf, sizeof buf.mtext, 0, 0);
   }
   printf("Received: %s", buf.mtext);
}
```

IPC: Shared Memory

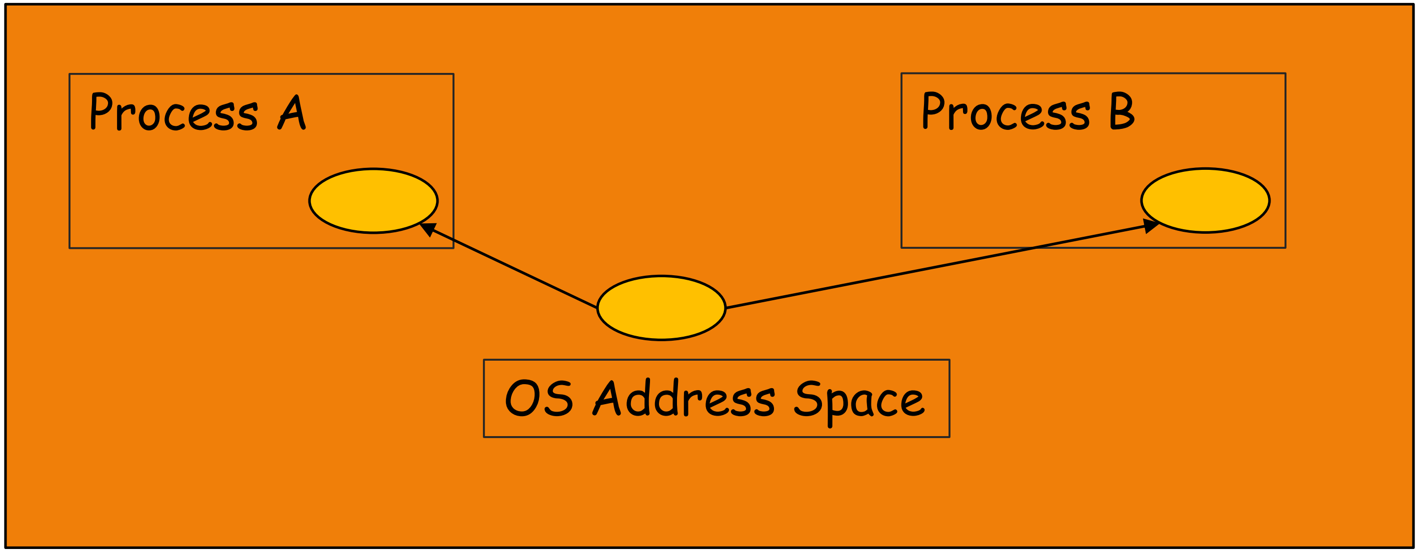
Shared Memory

- How does data travel through a FIFO?
  - ‘write’ copies data from process memory to kernel buffer and then ‘read’ copies data from a kernel buffer to process memory
- If both processes are on the same machine living in different parts of user space, then they may not need to copy data in and out of the kernel
  - They may exchange or share data by using a shared memory segment
  - Shared memory is to processes what global variables are to threads

Shared Memory

- Processes share the same segment of memory directly
  - Memory is mapped into the address space of each sharing process
  - Memory is persistent beyond the lifetime of the creating or modifying processes (until deleted)
- Mutual exclusion **must be provided by** processes using the shared memory

Shared Memory



- Processes request the segment
- OS maintains the segment
- Processes can attach/detach the segment

Facts about Shared Memory Segments

- A shared memory segment lives in memory independent of a process
- A shared memory segment has a name, called a key
- A key is an integer
- A shared memory segment has an owner and permission bits
- Processes may “attach” or “detach” a segment, obtaining a pointer to the segment
- reads and writes to the memory segment are done via regular pointer operations

Shared Memory – POSIX functions

- shmget: create and initialize or access
- shmat: attach memory to process
- shmdt: detach memory from process
- shmctl: control

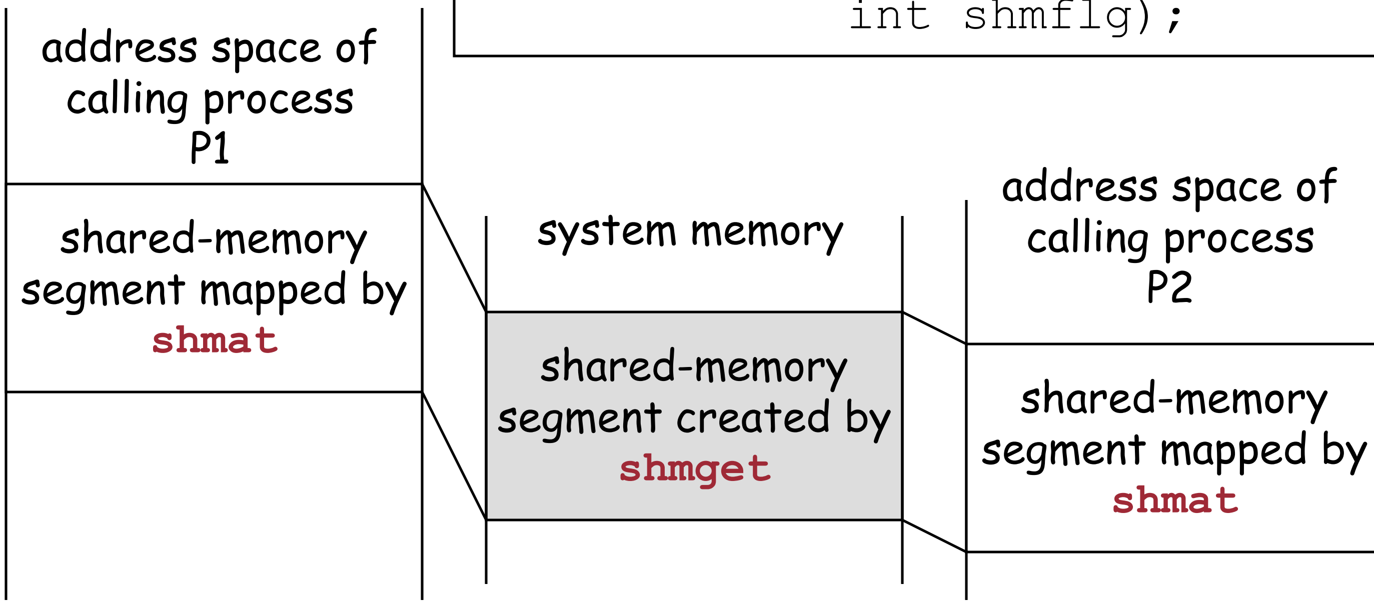
POSIX Shared Memory


```
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

Ok, we have created a shared-memory segment. Now what?

```
void *shmat(int shmid, const void *shmaddr,
                   int shmflg);
```



Shared Memory Example - Client

Understanding Unix/Linux Programming, Bruce Molay


```
#include <stdio.h>
#include <sys/shm.h>
#include <time.h>

#define TIME_MEM_KEY 99/* kind of like a port number */
#define SEG_SIZE ((size_t)100)/* size of segment*/
#define oops(m,x)  { perror(m); exit(x); }

main()
{
  int seg_id;
  char *mem_ptr, *ctime();
  long now;

  /* create a shared memory segment */

  seg_id = shmget( TIME_MEM_KEY, SEG_SIZE, 0777 );
  if ( seg_id == -1 )
    oops("shmget",1);

  /* attach to it and get a pointer to where it attaches */

  mem_ptr = shmat( seg_id, NULL, 0 );
  if ( mem_ptr == ( void *) -1 )
    oops("shmat",2);

  printf("The time, direct from memory: ..%s", mem_ptr);

  shmdt( mem_ptr );/* detach, but not needed here */
}
```

Shared Memory Example (SERVER)


```
/* shm_ts.c : the time server using shared memory, a bizarre application  */

#include <stdio.h>
#include <sys/shm.h>
#include <time.h>

#define TIME_MEM_KEY 99/* like a filename      */
#define SEG_SIZE ((size_t)100)/* size of segment*/
#define oops(m,x)  { perror(m); exit(x); }

main()
{
  int seg_id;
  char *mem_ptr, *ctime();
  long now;
  int n;

  /* create a shared memory segment */

  seg_id = shmget( TIME_MEM_KEY, SEG_SIZE, IPC_CREAT|0777 );
  if ( seg_id == -1 )
    oops("shmget", 1);

  /* attach to it and get a pointer to where it attaches */

  mem_ptr = shmat( seg_id, NULL, 0 );
  if ( mem_ptr == ( void *) -1 )
    oops("shmat", 2);

  /* run for a minute */
  for(n=0; n<60; n++ ){
    time( &now );/* get the time*/
    strcpy(mem_ptr, ctime(&now));/* write to mem */
    sleep(1);/* wait a sec    */
  }

  /* now remove it */
  shmctl( seg_id, IPC_RMID, NULL );
}
```

Understanding Unix/Linux Programming, Bruce Molay

POSIX IPC: Overview

primitive	POSIX function	description

message queues	`msgget`
	`msgctl`
	`msgsnd/msgrcv`
		create or access
		control
		send/receive message

semaphores	`semget`
	`semctl`
	`semop`
		create or access
		control
		wait or post operation

shared memory	`shmget`
	`shmctl`
	`shmat/shmdt`
		create and init or access
		control
		attach to / detach from process

Accessing IPC resources from the shell: ipcs [-a]