CS304 Operating Systems

DR GAYATHRI ANANTHANARAYANAN

gayathri@iitdh.ac.in

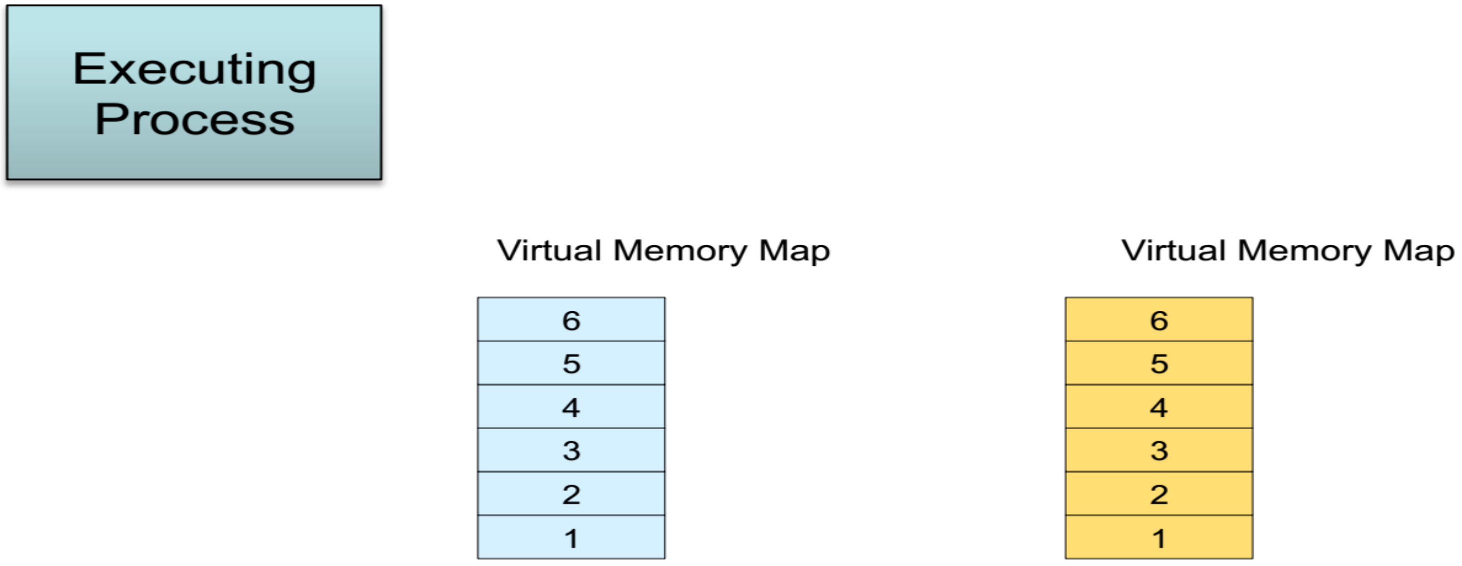
Materials in these slides have been borrowed from textbooks and existing operating systems courses

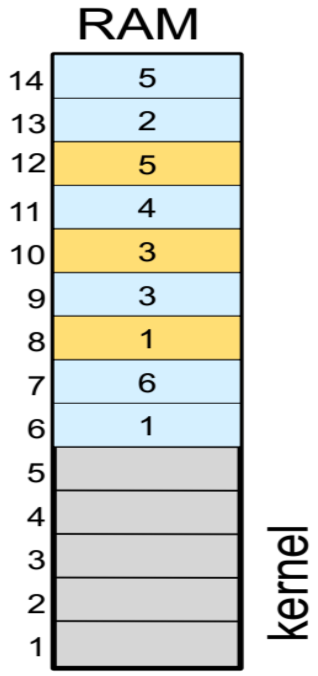
21 Jan 2021 Inter Process Communication

- Processes do not share any memory with each other
- Some processes might want to work together for a task, so need to communicate information
- IPC mechanisms to share information between processes

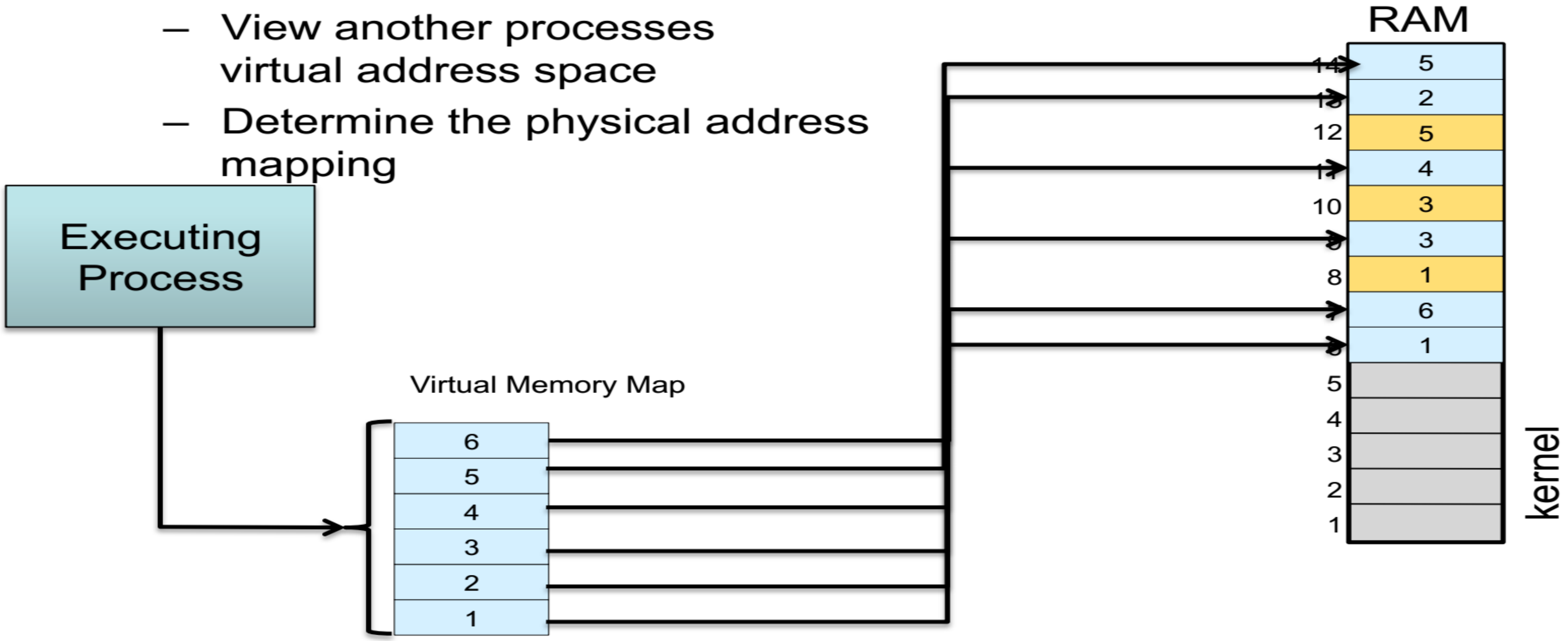
Virtual Memory View

- During execution, each process can only view its virtual addresses,
- It cannot
  - View another processes virtual address space
  - Determine the physical address mapping

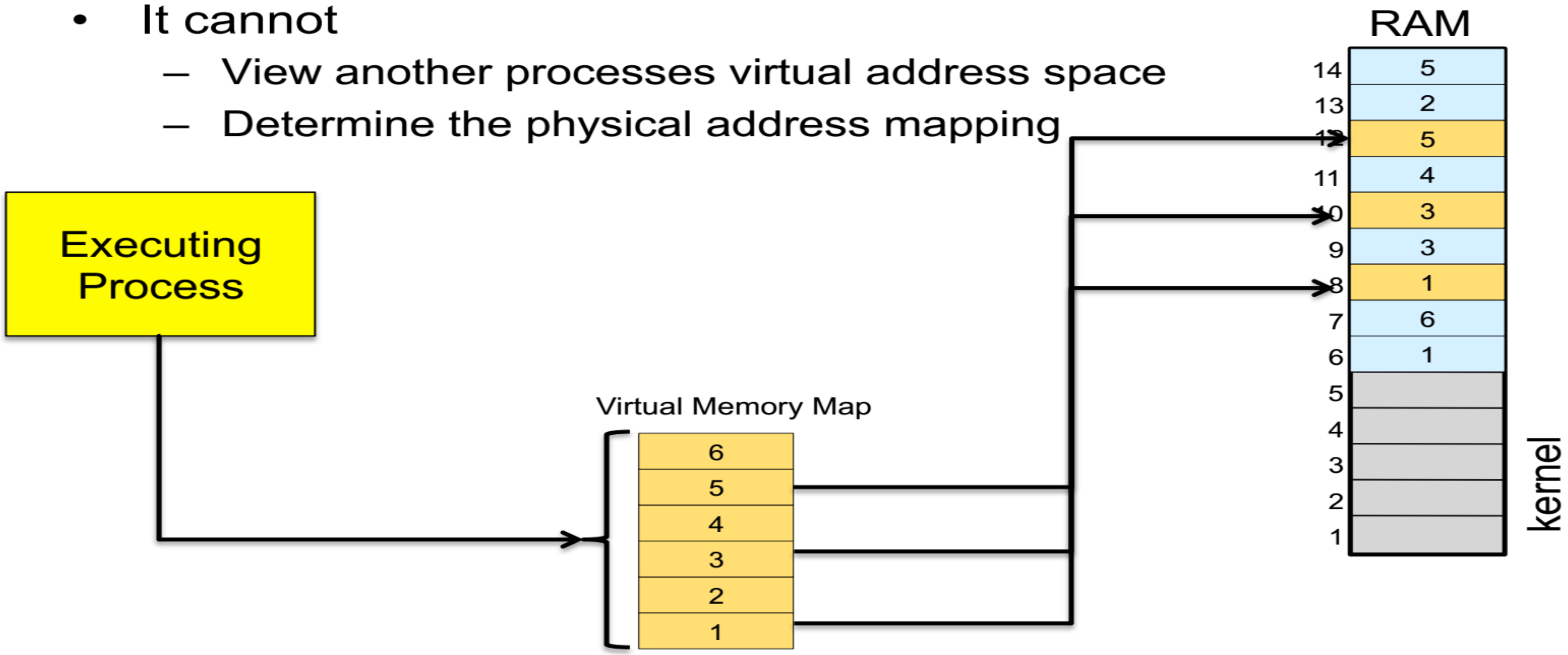




- During execution, each process can only view its virtual addresses
- It cannot
  - View another processes virtual address space
  - Determine the physical address mapping



- During execution, each process can only view its virtual addresses,
- It cannot
  - View another processes virtual address space
  - Determine the physical address mapping

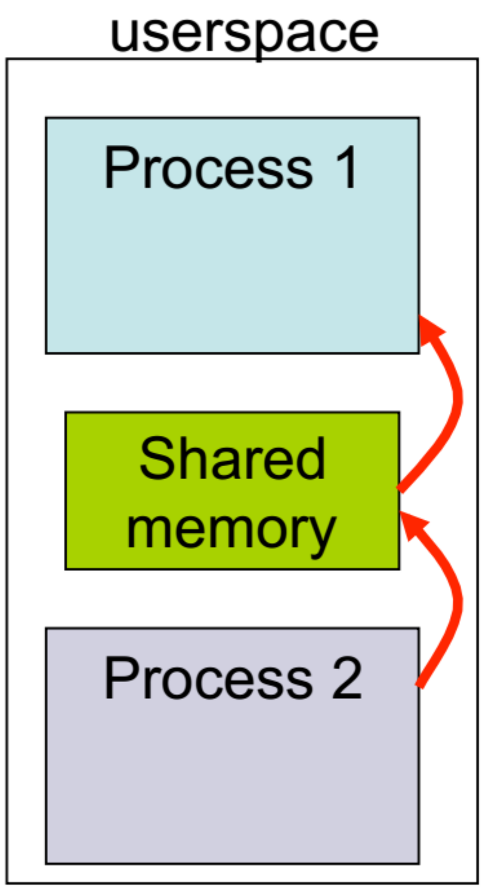


Inter Process Communication

- Advantages of Inter Process Communication (IPC)
  - Information sharing
  - Modularity/Convenience
- 3 ways
  - Shared memory
  - Message Passing
  - Signals

Shared Memory

- One process will create an area in RAM which the other process can access
- Both processes can access shared memory like a regular working memory
  - Reading/writing is like regular reading/writing
  - Fast
- Limitation : Error prone. Needs synchronization between processes



Shared Memory

Processes can both access same region of memory via **`shmget()`** system call

- **`int shmget(key_t key, int size, int shmflg)`**

IPC_CREAT

IPC_EXCL

- By providing same key, two processes can get same segment of memory
- Can read/write to memory to communicate
- Need to take care that one is not overwriting other's data: how?

Shared Memory in Linux

- **int shmget (key, size, flags)**
  - Create a shared memory segment;
  - Returns ID of segment : shmid
  - key : unique identifier of the shared memory segment
  - size : size of the shared memory (rounded up to the PAGE_SIZE)
- **int shmat(shmid, addr, flags)**
  - Attach shmid shared memory to address space of the calling process
  - addr : pointer to the shared memory address space
- **int shmdt(shmid)**
  - Detach shared memory

Example

server.c

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

#define SHMSIZE     27 /* Size of shared memory */

main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;

    key = 5678; /* some key to uniquely identifies the shared memory */

    /* Create the segment. */
    if ((shmid = shmget(key, SHMSIZE, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /* Attach the segment to our data space. */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /* Now put some things into the shared memory */
    s = shm;
    for (c = 'a'; c <= 'z'; c++)
        *s++ = c;
    *s = 0; /* end with a NULL termination */

    /* Wait until the other process changes the first character
     * to '*' the shared memory */
    while (*shm != '*')
        sleep(1);
    exit(0);
}
```

client.c

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

#define SHMSIZE     27

main()
{
    int shmid;
    key_t key;
    char *shm, *s;

     /* We need to get the segment named "5678", created by the server
    key = 5678;

    /* Locate the segment. */
    if ((shmid = shmget(key, SHMSIZE, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /* Attach the segment to our data space. */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /* read what the server put in the memory. */
    for (s = shm; *s != 0; s++)
        putchar(*s);
    putchar('\n');

    /*
     * Finally, change the first character of the
     * segment to '*', indicating we have read
     * the segment.
     */
    *shm = '*';

    exit(0);
```

Signals

• A certain set of signals supported by OS

– Some signals have fixed meaning (e.g., signal to terminate process)

– Some signals can be user-defined

• Signals can be sent to a process by OS or another process (e.g., if you type Ctrl+C, OS sends SIGINT signal to running process)

• Signal handler: every process has a default code to execute for each signal

– Exit on terminate signal

• Some signal handlers can be overridden to do other things

- Asynchronous unidirectional communication between processes
- Signals are a small integer
  - eg. 9: kill, 11: segmentation fault
- Send a signal to a process
  - kill(pid, signum)
- Process handler for a signal
  - sighandler_t signal(signum, handler);
  - Default if no handler defined

Sockets

- Sockets can be used for two processes on same machine or different machines to communicate

– TCP/UDP sockets across machines

– Unix sockets in local machine

- Communicating with sockets

– Processes open sockets and connect them to each other

– Messages written into one socket can be read from another

– OS transfers data across socket buffers

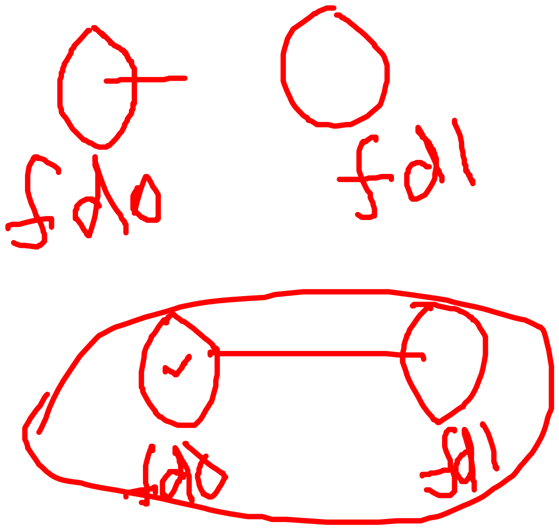
Pipes

- Pipe system call returns two file descriptors
  - Read handle and write handle
  - A pipe is a half-duplex communication
  - Data written in one file descriptor can be read through another
- Regular pipes: both fd are in same process (how it is useful?)
  - Parent and child share fd after fork
  - Parent uses one end and child uses other end
- Named pipes: two endpoints of a pipe can be in different processes
- Pipe data buffered in OS buffers between write and read

fork ()

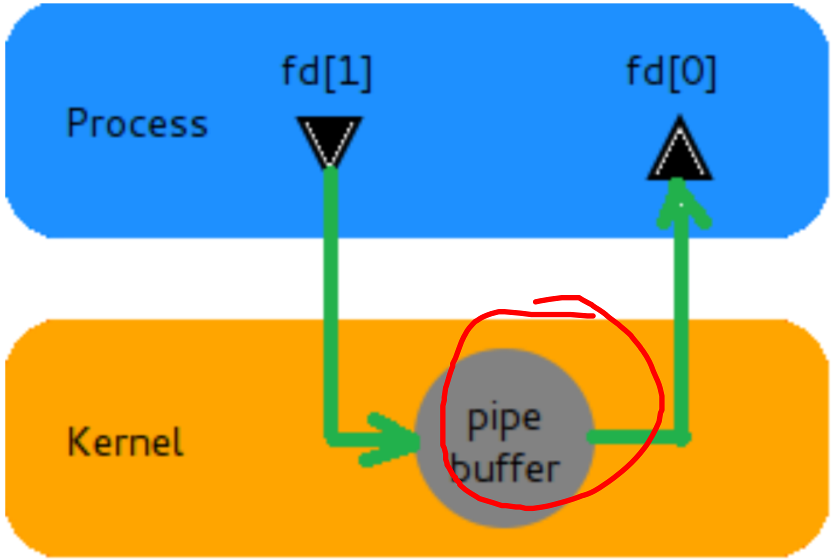
Half duplex

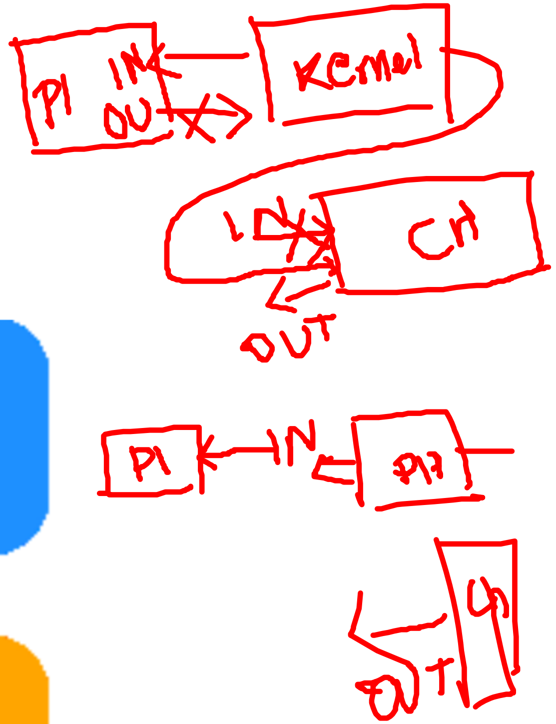
pipe()



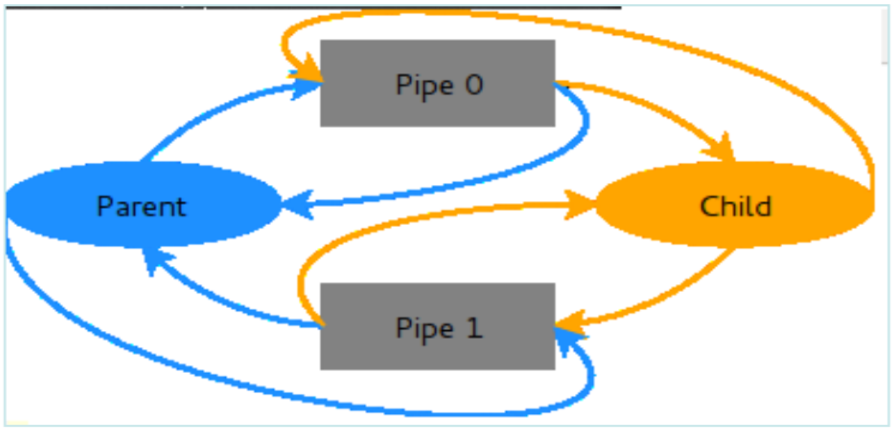
Pipes

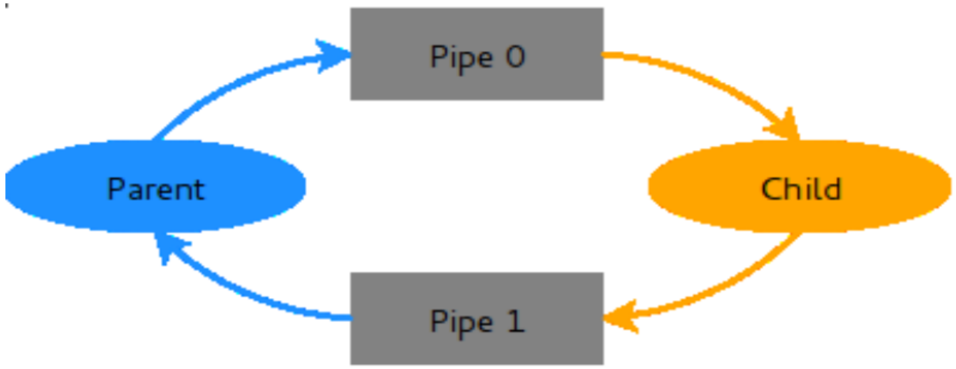
- Always between parent and child
- Always unidirectional
- Accessed by two associated file descriptors:
  - fd[0] for reading from pipe
  - fd[1] for writing to the pipe





Two Way Communication





- Two pipes opened
  pipe0 and pipe1
- Note the unnecessary pipes

- Close the unnecessary pipes

Example [Child process sending a string to Parent]

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

int main(){
  int pipefd[2];
  int pid;
  char recv[32];

  pipe(pipefd);

  switch(pid=fork()) {
  case -1: perror("fork");
           exit(1);
  case 0:                                   /* in child process */
        close(pipefd[0]);                   /* close unnecessary pipefd */
        FILE *out = fdopen(pipefd[1], "w"); /* open pipe descriptor as stream */
        fprintf(out, "Hello World\n");      /* write to out stream */
        break;
  default:                                  /* in parent process */
        close(pipefd[1]);                   /* close unnecessary pipefd */
        FILE *in = fdopen(pipefd[0], "r");  /* open descriptor as stream */
        fscanf(in, "%s", recv);             /* read from in stream */
        printf("%s", recv);
        break;
  }
}
```

dup()
dup2()

Named
FIFOs

Message Queues

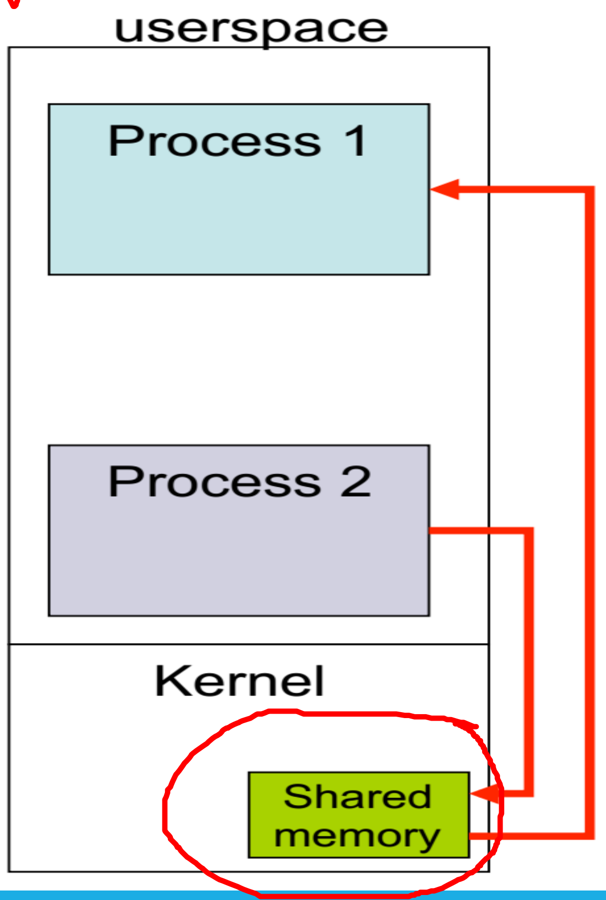
- Mailbox abstraction
- Process can open a mailbox at a specified location
- Processes can send/receive messages from mailbox
- OS buffers messages between send and receive

MB

Message Passing

msgget()
Msgsend()

- Shared memory created in the kernel
- System calls such as send and receive used for communication
  - Cooperating : each send must have a receive
- Advantage : Explicit sharing, less error prone
- Limitation : Slow. Each call involves marshalling / demarshalling of information



Blocking Vs Non-Blocking

- Some IPC actions can block

– Reading from socket/pipe that has no data, or reading from empty message queue

– Writing to a full socket/pipe/message queue

- The system calls to read/write have versions that block or can return with an error code in case of failure

– A socket read can return error indicating no data to be read, instead of blocking