CS304 Operating Systems

DR GAYATHRI ANANTHANARAYANAN

gayathri@iitdh.ac.in

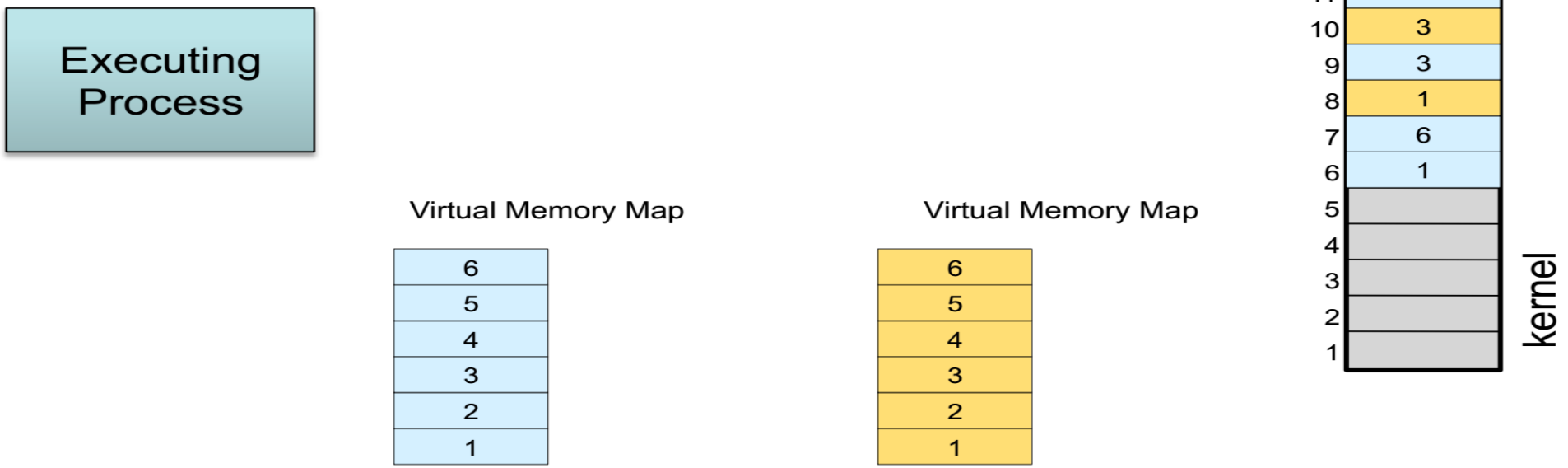
Materials in these slides have been borrowed from textbooks and existing operating systems courses

Inter Process Communication

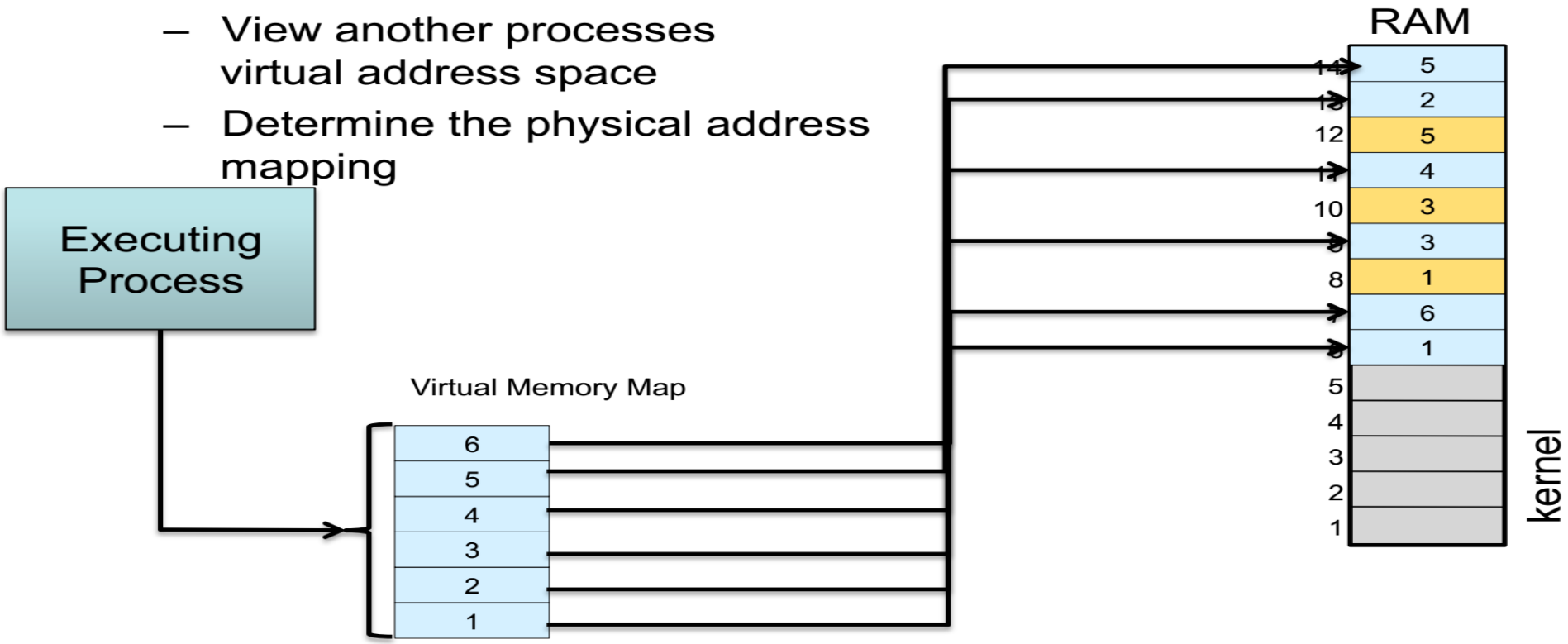
- Processes do not share any memory with each other
- Some processes might want to work together for a task, so need to communicate information
- IPC mechanisms to share information between processes

Virtual Memory View

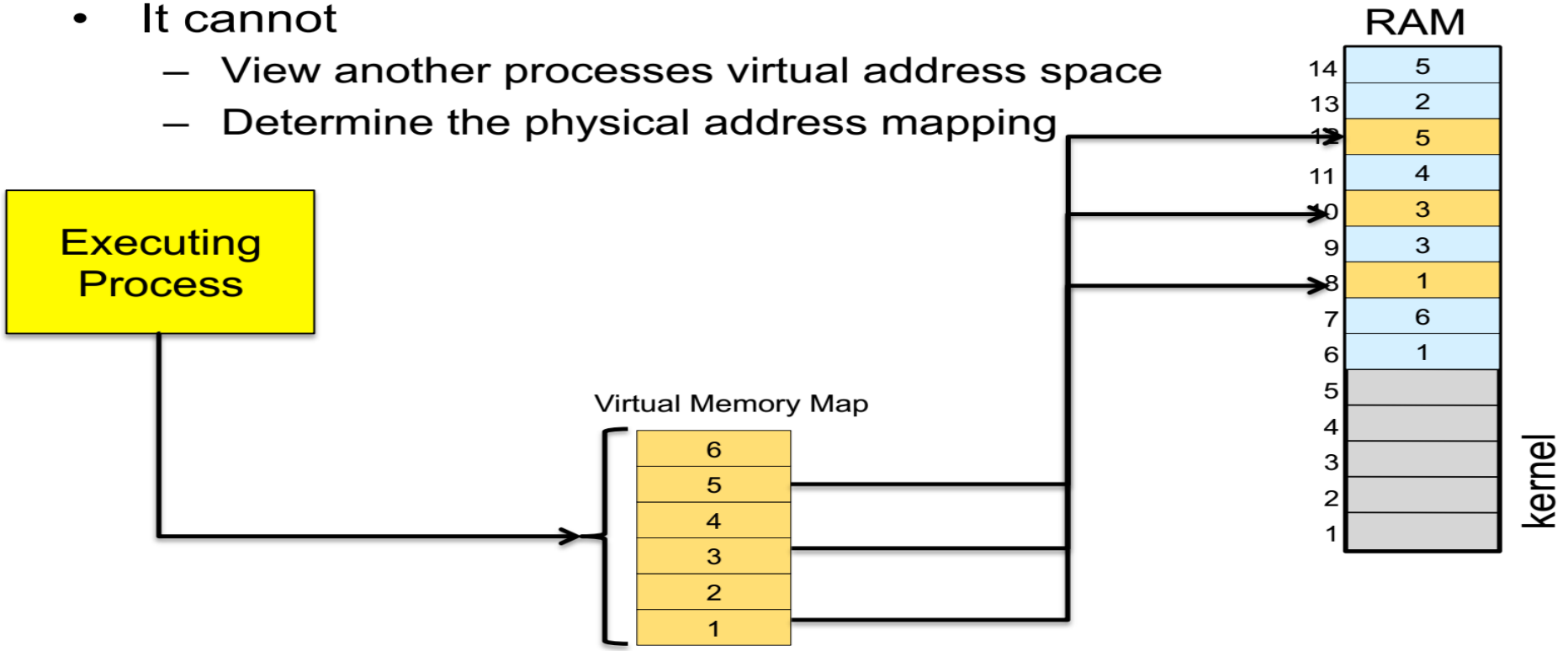
- During execution, each process can only view its virtual addresses,
- It cannot
  - View another processes virtual address space
  - Determine the physical address mapping



- 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



Why processes communicate ?

Higher performance on Multicores

Higher Performance on slow I/O

Modularity in design

Fault Tolerance (usually in distributed systems)

Mechanisms of IPC

1. Pipes

2. Shared Memory

3. Message Passing

4. Signals

Pipes

- Pipe system call returns two file descriptors

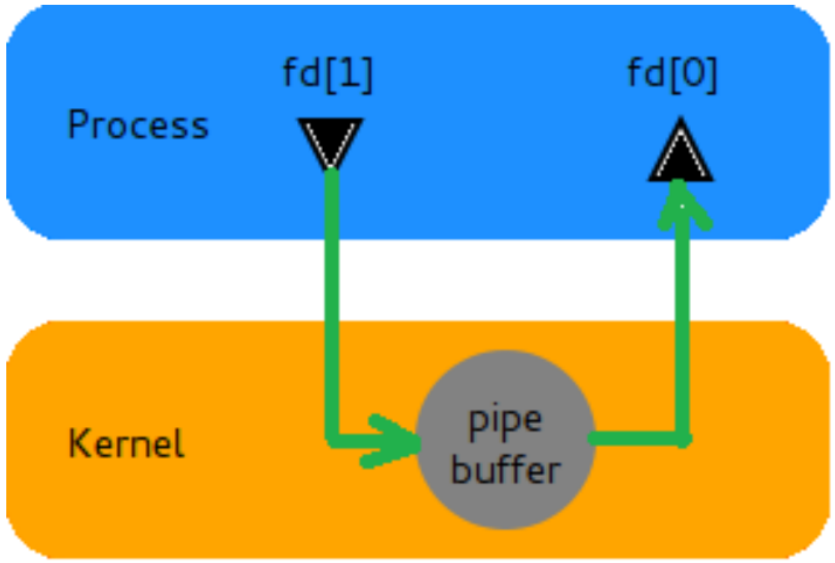
– Read handle and write handle

– A pipe is a half-duplex communication

– Data written in one file descriptor can be read through another

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



Pipes

```
$ ps aux | grep 'zombie' | wc -l
```

Interfacing of various small modular tools with pipes

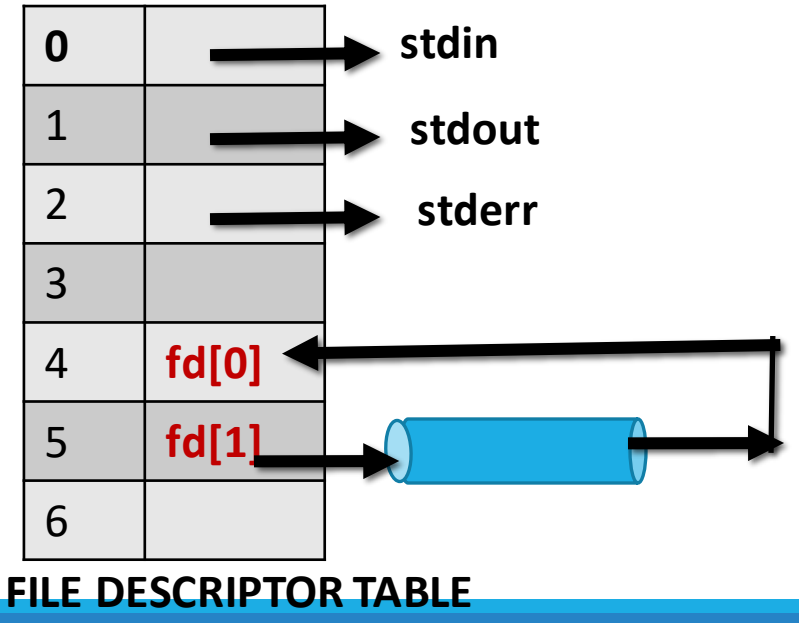
A pipe is a pair of file descriptors one input and another output

```
int fd[2];
pipe(fd);
```

Pipe storage implemented as a buffer in kernel space

fd[0] is set up for reading, fd[1] is set up for writing

Blocking FIFO semantics on read



Pipes and forking

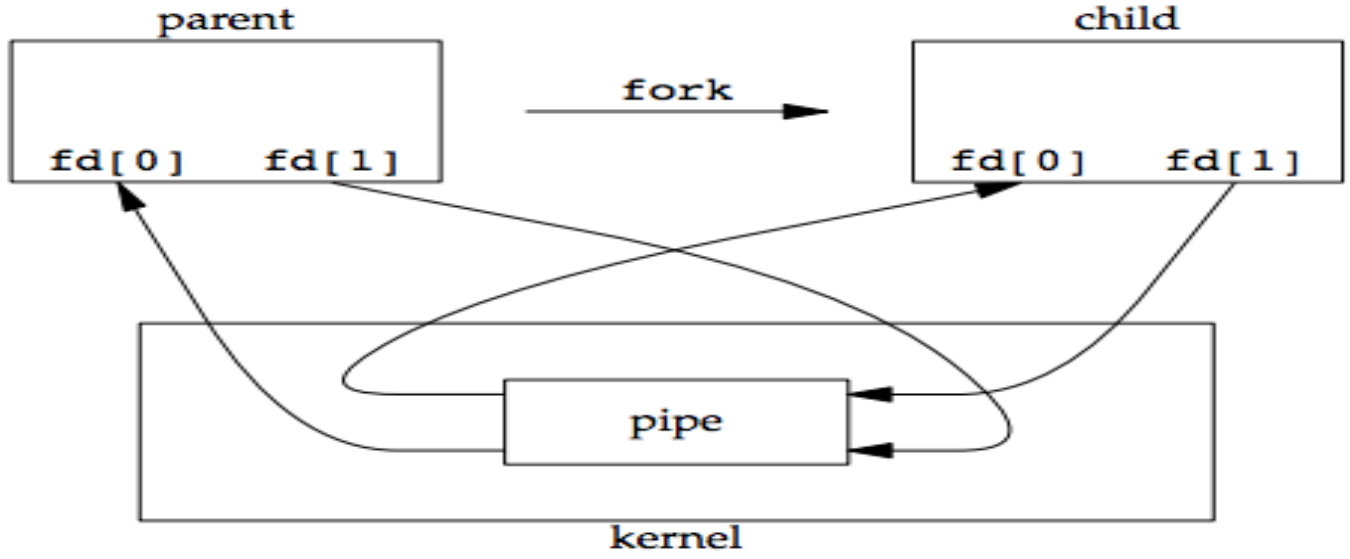
During forking, file description table is copied, and so are pipes

Provides a simple mechanism to communicate between the parent and child.

A pipe in a single process is next to useless. Normally, the process that calls pipe then calls fork, creating an IPC channel from the parent to the child, or vice versa.

The following figure

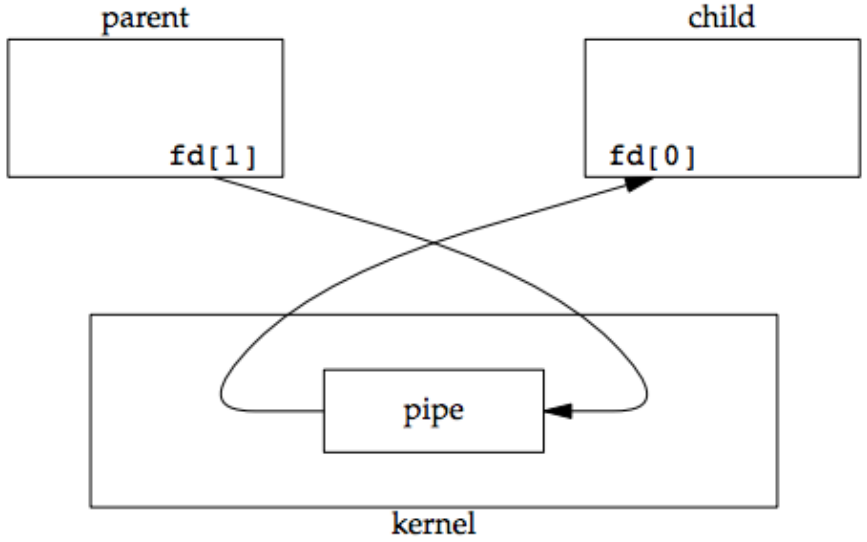
shows this scenario:



What happens after the fork depends on which direction of data flow we want.

For a pipe from the parent to the child, the parent closes the read end of the pipe (*fd[0]*), and the child closes the write end (*fd[1]*).

The following figure shows the resulting arrangement of descriptors.



Example

```
int main(int argc, char *argv[]) {
  int fds[2];
  pipe(fds);
  pid_t pid = fork();
  if (pid == 0) {
    close(fds[1]);
    char buffer[6];
    read(fds[0], buffer, sizeof(buffer));
    printf("Read from pipe bridging processes: %s.\n", buffer);
    close(fds[0]);
    return 0;
  }
  close(fds[0]);
  write(fds[1], "hello", 6);
  waitpid(pid, NULL, 0);
  close(fds[1]);
  return 0;
}
```

How do pipe and fork work together in this example?

- The base address of a small integer array called fds is shared with the call to pipe.
- pipe allocates two descriptors, setting the first to draw from a resource and the second to publish to that same resource.
- pipe then plants copies of those two descriptors into indices 0 and 1 of the supplied array before it returns.
- The fork call creates a child process, which itself inherits a copy of the parent's fds array.

- The reference counts in each of the two open file entries is promoted from 1 to 2 to reflect the fact that two descriptors—one in the parent, and a second in the child—reference each of them.
- Immediately after the fork call, anything printed to fds[1]is readable from the parent's fds[0]and the child's fds[0].
- Similarly, both the parent and child are capable of publishing text to the same resource via their copies of fds[1].

The parent closes fds[0] before it writes to anything to fds[1]to emphasize the fact that the parent has no interest in reading anything from the pipe.

Similarly, the child closes fds[1] before it reads from fds[0] to emphasize the fact that the it has zero interest in publishing anything to the pipe.

It's imperative all write endpoints of the pipe be closed if not being used, else the read end will never know if more text is to come or not.

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;
  }
}
```

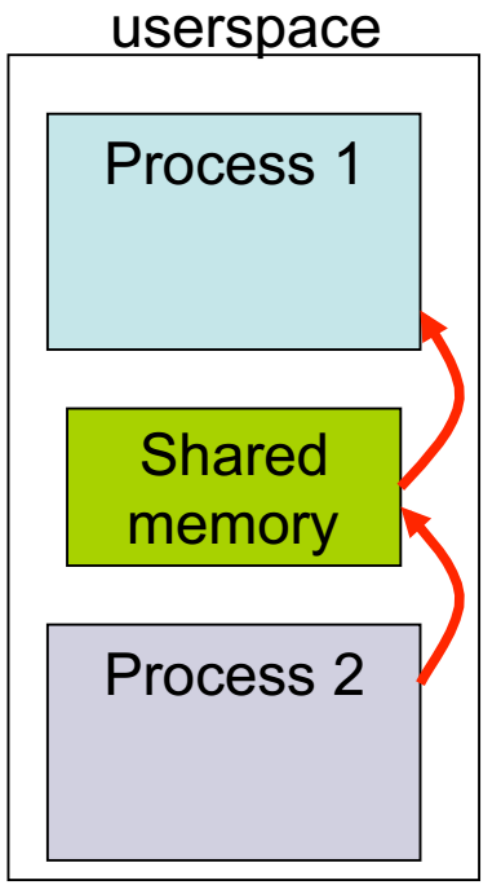
Limitations

Historically, they have been half duplex (data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.

Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

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)`**
- 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)**
  - **At**tach shmid shared memory to address space of the calling process
  - addr : pointer to the shared memory address space
- **int shmdt(shmid)**
  - **D**e**t**ach 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 signal is a small message that notifies a process that an event of some type occurred.
- Signals are often sent by the kernel, but they can be sent from other processes as well.
- A signal handler is a function that executes in response to the arrival and consumption of a signal.
- Eg : Your program in C - unintentionally dereferenced a NULL pointer.
- When that happens, the kernel delivers a signal of type SIGSEGV, informally known as a segmentation fault (or a SEGmentation Violation, or SIGSEGV, for short).
- Unless you install a custom signal handler to manage the signal differently, a SIGSEGV terminates the program and generates a core dump.
- Each signal category (e.g. SIGSEGV) is represented internally by some number (e.g. 11). In fact, C #defines SIGSEGV to be the number 11.

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

Other signal types:

Whenever a process commits an integer-divide-by-zero (and, in some cases, a floating-point divide by zero on older architectures), the kernel hollers and issues a SIGFPE signal to the offending process.

By default, the program handles the SIGFPE by printing an error message announcing the zero denominator and generating a core dump.

When you type ctrl-c, the kernel sends a SIGINT to the foreground process (and by default, that foreground is terminated).

When you type ctrl-z, the kernel issues a SIGTSTP to the foreground process (and by default, the foreground process is halted until a subsequent SIGCONT signal instructs it to continue).

When a process attempts to publish data to the write end of a pipe after the read end has been closed, the kernel delivers a SIGPIPE to the offending process. The default SIGPIPE handler prints a message identifying the pipe error and terminates the program.

Message Queues

- A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. The message queue is called a queue and its identifier is called a queue ID
- Events are asynchronous. When a task wants to send an event to another task, rather than sending it directly to the target task, it passes the event to the operating system message queue and the target task retrieves the event from the head of the message queue when it is ready to process it.

```
int msgget(key_t key, int flag);

int msgctl(int msqid, int cmd, struct msqid_ds *buf);

 int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);

 ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
```

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

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

int main() {
    int msqid = 12345; // Message queue ID
    struct message {
        long type;
        char text[20];
    } msg;

    msg.type = 1;
    strcpy(msg.text, "This is message 1");

    msgsnd(msqid, (void *)&msg, sizeof(msg.text), IPC_NOWAIT);

    strcpy(msg.text, "This is message 2");
    msgsnd(msqid, (void *)&msg, sizeof(msg.text), IPC_NOWAIT);

    return 0;
}
```

```
$ ipcs -q
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x123456   12345      user       644        1024         2
```

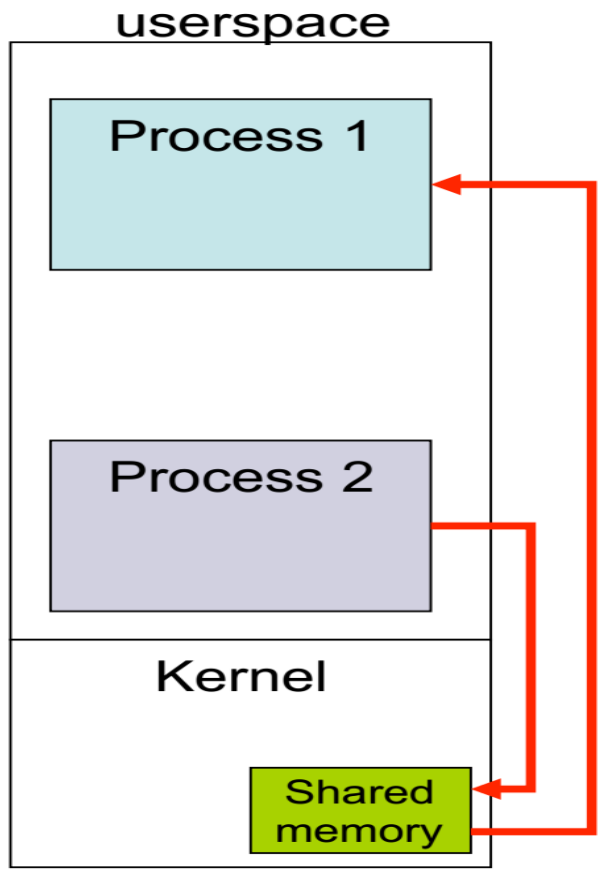
```c
#include <stdio.h>
#include <sys/msg.h>

int main() {
    int msqid = 12345;
    struct message {
        long type;
        char text[20];
    } msg;
    long msgtyp = 0;

    msgrcv(msqid, (void *)&msg, sizeof(msg.text), msgtyp, MSG_NOERROR | IPC_NOWAIT);
    printf("%s \n", msg.text);

    return 0;
}
```

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

Message Passing

- 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