

ECE437/CS481

# M02C: PROCESSES & THREADS INTERPROCESS COMMUNICATIONS

Chapter 3.4-3.5

Xiang Sun

The University of New Mexico

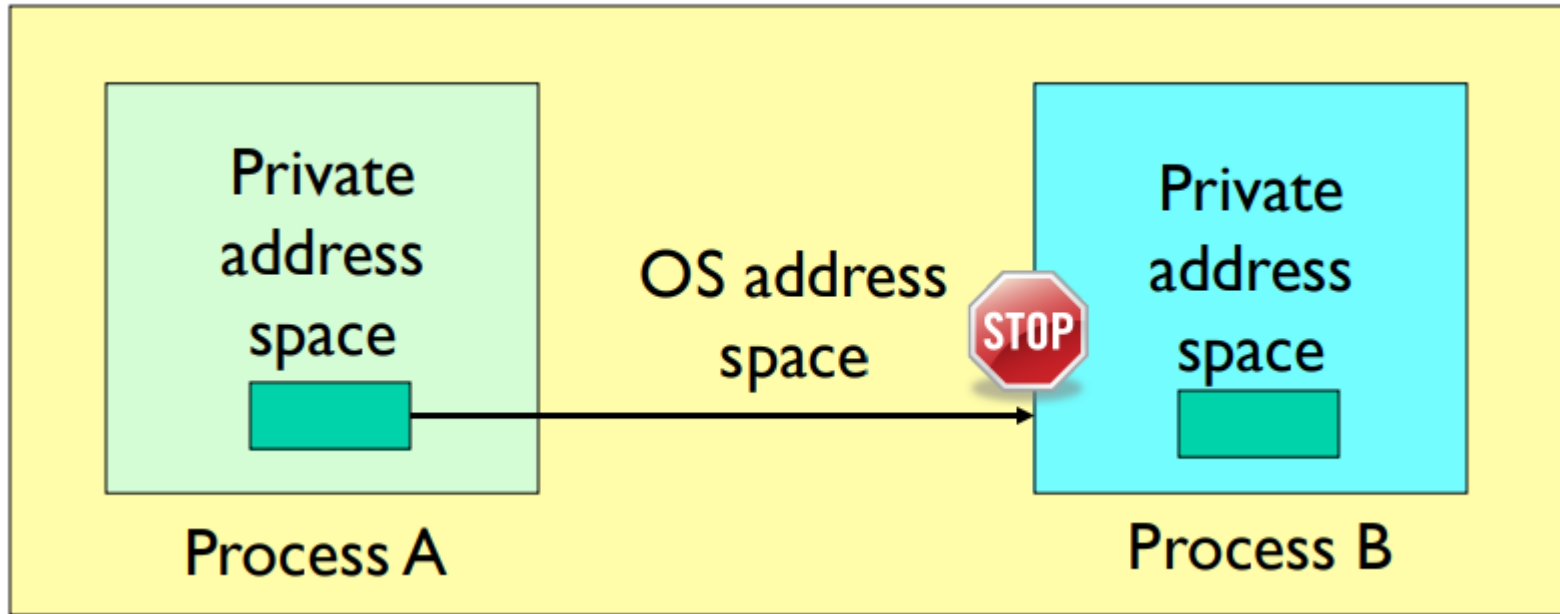
A decorative blue wavy line that spans the width of the slide, starting with a small upward curve on the left, dipping into a V-shape in the center, and then curving back up on the right before continuing as a straight line to the edge.

# Interprocess Communications

- ❑ Processes within a system may be?
  - Independent or Cooperative
- ❑ Cooperative process?
  - Can affect or be affected by other processes, including sharing data
  - Benefits
    - ✓ Information sharing
    - ✓ Computation speedup
    - ✓ Modularity
    - ✓ Convenience
- ❑ Cooperating processes need Interprocess Communication (IPC)
  - Shared memory
  - Message passing

# Interprocess Communications

❑ IPC is not easy

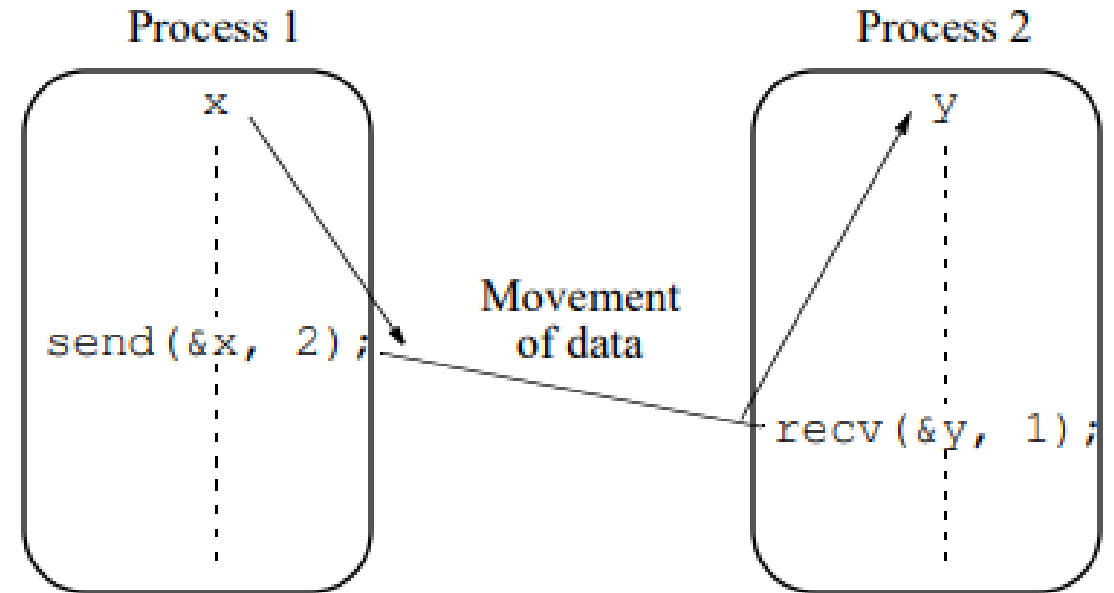


- Each process has a private address space
- No process can write to another process's space
- How can we get data from process A to process B?

# Message Passing

## □ Message Passing

- Establish a link between two processes.
- Exchange messages via send/receive
- Different types of messaging passing:
  - ✓ Synchronous vs. Asynchronous
  - ✓ Direct vs. Indirect

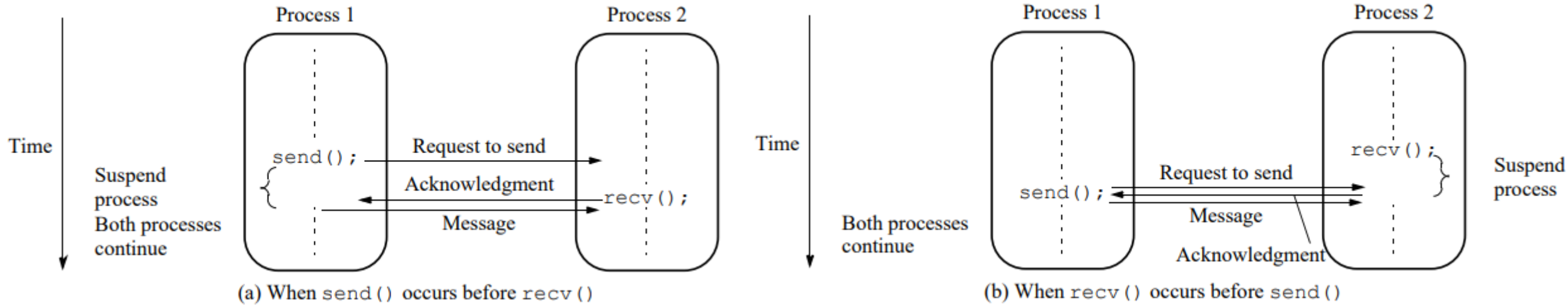


# Message Passing

## □ Synchronous vs. Asynchronous Message Passing

### ➤ Synchronous Message Passing

- ✓ Both source and destination process may suffer from a long suspension time.



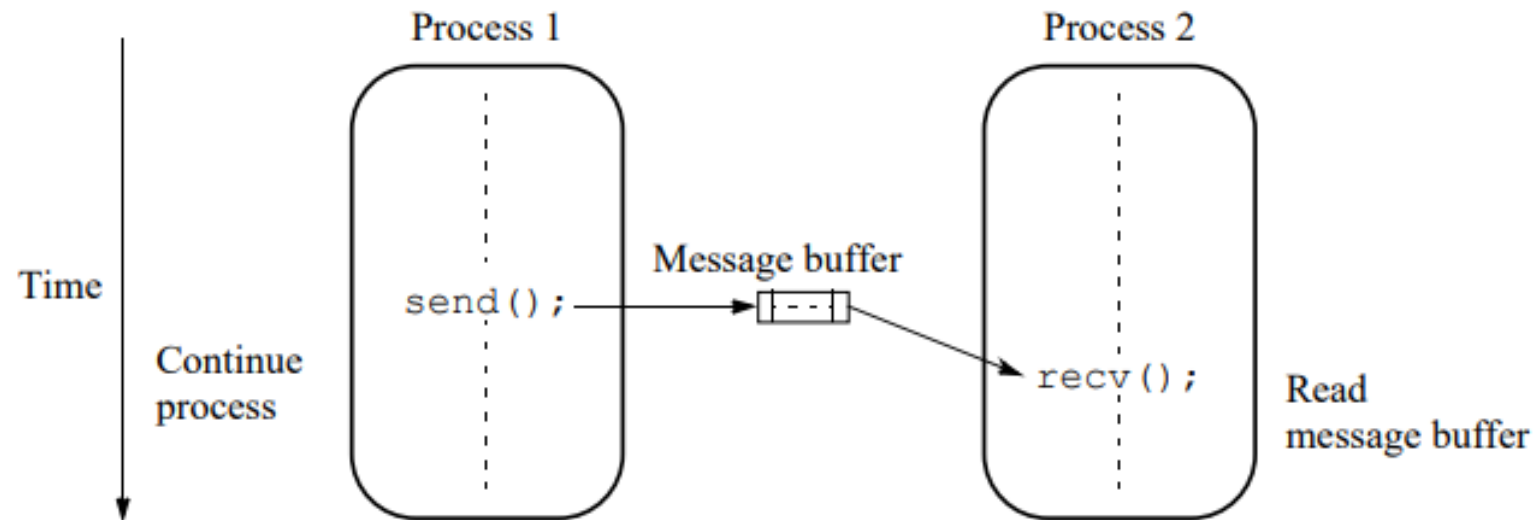
Synchronous `send()` and `recv()` library calls using a three-way

# Message Passing

## □ Synchronous vs. Asynchronous Message Passing

### ➤ Asynchronous Message Passing

- ✓ A message buffer is needed between source and destination to hold message.



## □ Synchronous vs. Asynchronous Message Passing

### Source Process Side

#### ➤ Synchronous

- ✓ Will not return until ack has been received
- ✓ Block on full buffer

#### ➤ Asynchronous

- ✓ Return until the message has been sent out
- ✓ Completion
  1. Require process to check status (POLLING)
  2. Notify or signal the process
- ✓ Block on full buffer

### Destination Process Side

#### ➤ Synchronous

- ✓ Return data if there is message
- ✓ Block on empty buffer

#### ➤ Asynchronous

- ✓ Return data if there is a message
- ✓ Return null if there is no message

# Message Passing

## □ Direct vs. Indirect Message Passing

### ➤ Direct message passing, **PID based**

- ✓ `send (P, message)` - send a message to process P
- ✓ `receive(Q, message)`- receive a message from process Q

### ➤ Indirect message passing, **mailbox based**

- ✓ `send(A, message)` - send a message to mailbox A.
- ✓ `receive(A, message)`-receive a message from mailbox A.
- ✓ Each mailbox has a unique ID.
- ✓ A mailbox can be used for multiple sender-receiver pairs. That is, mailbox allows one-to-one, one-to-many, many-to-one, many-to-many communications.



## □ Direct vs. Indirect Message Passing

### ➤ Problem of many-to-many communications for applying mailbox

- ✓ P1, P2, and P3 share mailbox A.
- ✓ P1 sends two different messages to P2 and P3 via mailbox A, respectively.
- ✓ Who gets the which message?

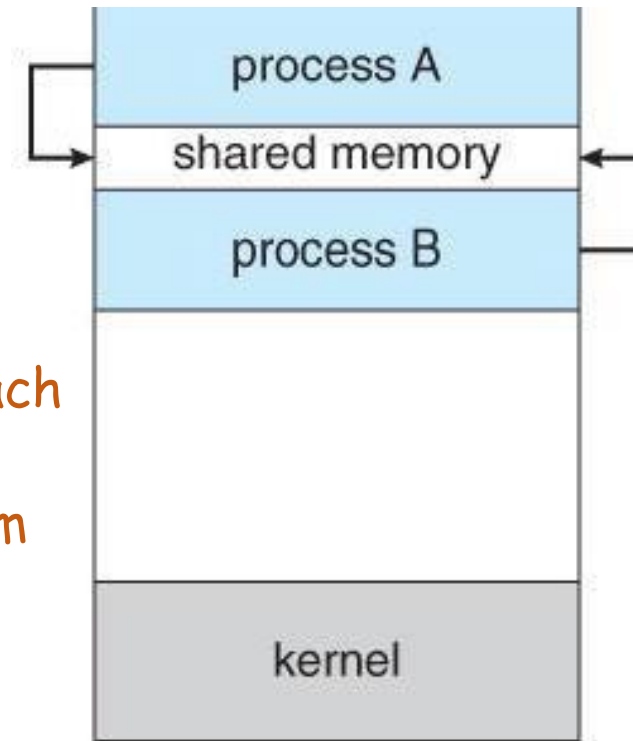
### ➤ Solutions

- ✓ Allow a mailbox to be used by only one sender-receiver pair.
- ✓ Requires open/close a mailbox before using it.

# Shared Memory

## □ Shared Memory

- Data is exchanged by placing it in memory pages shared by multiple processes.
- The shared memory pages **may not be contiguous**.
- Shared memory provides four operations:
  - ✓ `int shmget(key_t key, size_t size, int shmflg)`: get a shared memory segment. `shmget()` returns an identifier for the shared memory segment upon successful completion.
  - ✓ `void *shmat(int shmid, void *shmaddr, int shmflg)`: attach the process to the shared memory segment.
  - ✓ `int shmdt(void *shmaddr)`: detach the process to the from the shared memory segment.
  - ✓ `int shmctl(int shmid, int cmd, struct shmid_ds *buf)`: destroy/delete the shared memory segment (where `cmd=IPC_RMID` and `*buf=NULL`).



# Shared Memory

## ❑ Shared Memory Example—Share\_mem\_wrt

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ 27

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

    key_t key = ftok("shmfile",65);
    /*Create the segment.*/
    if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }
    /*Now we attach the segment.*/
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }
    /*Now put some things into the memory for the other process to read.*/
    s = shm;
    for (c = 'a'; c <= 'z'; c++)
        *s++ = c;
    *s = NULL;

    while (*shm != '*')
        sleep(1);

    exit(0);
}
```

Operation permissions	Octal value
Read by user	00400
Write by user	00200
Read by group	00040
Write by group	00020
Read by others	00004
Write by others	00002

The process waits until the other process changes the first character of our memory to '\*', indicating that some process has read what we put there.

# Shared Memory

## □ Shared Memory Example—Share\_mem\_rd

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ 27

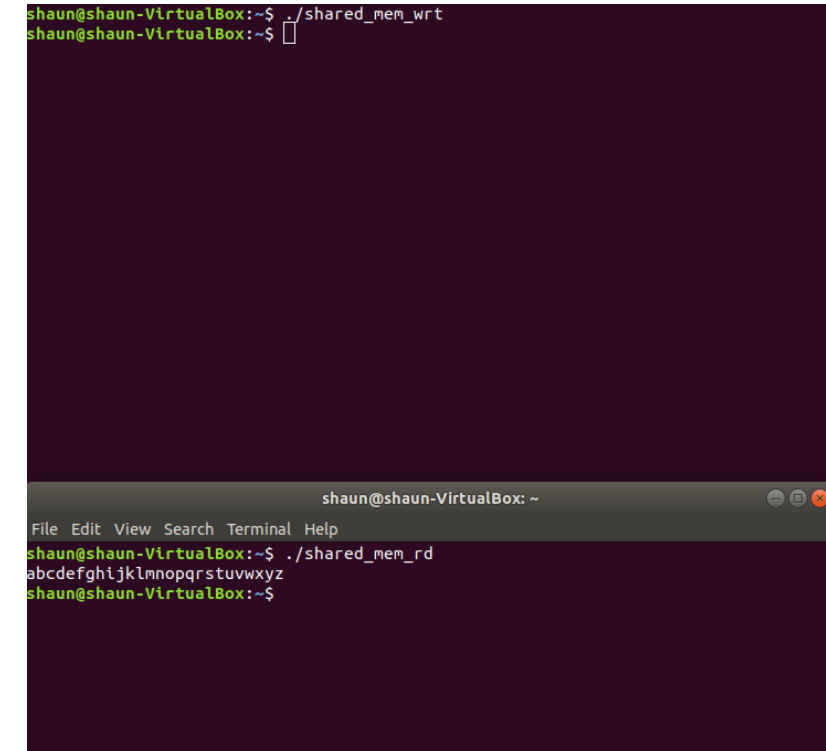
main()
{
    int shmid;
    char *shm, *s;

    key_t key = ftok("shmfile",65);
    //Locate the segment.
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    //Now we attach the segment
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

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

    *shm = '*';
    exit(0);
}
```



```
shaun@shaun-VirtualBox:~$ ./shared_mem_wrt
shaun@shaun-VirtualBox:~$ 
shaun@shaun-VirtualBox: ~
File Edit View Search Terminal Help
shaun@shaun-VirtualBox:~$ ./shared_mem_rd
abcdefghijklmnopqrstuvwxyz
shaun@shaun-VirtualBox:~$
```

Finally, the process changes the first character of the segment to '\*', indicating the process has read the segment.

## ❑ IPC standard—Unix System V and POSIX

- Define semaphore, shared memory, message passing

## ❑ IPC keys

- Keys are members to identify an **IPC object** on a system. An **IPC object** can be
  - ✓ msg—a message queue
  - ✓ sem—a set of semaphores (to be covered later)
  - ✓ shm—a shared-memory segment
- A key allows an IPC object to be shared among different processes.
- The type of a key is determined by the type **key\_t**, which is defined in the system header file `< sys/types.h >`.

❑ `ftok()`: a library function of mapping characters (normally, a file's pathname) into a key.

❑ Synopsis/Syntax:

```
include <sys/types.h>  
key_t ftok(const char *path, int id);
```

- The `ftok()` function returns a key based on `path` and `id` that is usable in subsequent calls to `msgget`, `semget` and `shmget`. The `path` argument **must be?** the pathname of an existing file.
- The `ftok()` function will return the same key value for all paths that name the same file, when called with the same `id` value, and will return different key values otherwise.

## ❑ Unix/Linux Pipe

- Provide processes with a simple way of passing information
  - ✓ Just like a special file that can store a limited amount of data in a **FIFO Manner**
  - ✓ Constant "PIPE\_BUF" defined as **5120**, the max # of bytes a pipe may hold.
- How to communicate:
  - One process writes to the pipe, another process reads from the pipe.
  - If a writing process attempts to write to a **full** pipe, it will block.
  - If a reading process attempts to read from an **empty** pipe, it will block.

## □ Unix/Linux Pipe

➤ **Unnamed pipe:** Piping between the parent & child processes.

✓ before calling `fork()`, use `pipe()` to create an interprocess channel

```
#include <unistd.h>  
int pipe(int fildes[2]);
```

✓ it returns two file descriptors, i.e., `fildes[0]` and `fildes[1]`, for reading and writing, respectively.

```
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

```
ssize_t read(int fildes, void*buf, size_t nbyte)
```



## ❑ Unix/Linux Pipe

### ➤ Unnamed pipe example.

```
#include <unistd.h>
#include <stdio.h>
char msg1[20] = "Hello, world!"; char msg2[20] = "Bye, world!";
main() {
    char inmsg[20]; int pipedes[2]; pid_t pid;
    if (pipe(pipedes) < 0) { perror("pipe call failure"); exit(1); }
    switch (pid=fork()) {
        case -1: perror("fork call failure"); exit(2);
        case 0: // read from the pipe
            read(pipedes[0], inmsg, sizeof(msg1)); printf("%s\n",inmsg);
            read(pipedes[0], inmsg, sizeof(msg2)); printf("%s\n",inmsg); break;
        default: // write to the pipe
            write(pipedes[1], msg1, sizeof(msg1));
            write(pipedes[1], msg2, sizeof(msg2)); wait(NULL); break;
    }
}
```

```
shaun@shaun-VirtualBox:~$ ./pipe_test
Hello, world!
Bye, world!
```

## □ Unix/Linux Pipe

➤ **Named pipe:** Piping between a server process & a client process (non parent-child).

- ✓ A server process creates a file with a **file name**, writes into it. Then, a client process reads from the file using the same **file name**.
- ✓ Use `mkfifo()` to create a named pipe

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
```

## ❑ Unix/Linux Pipe

- Named pipe example: two processes A & B, process A writes msg to a named pipe and process B reads them from the named pipe.

## ✓ Process A

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

main() {
    int fd; extern int errno; char msg1[20] = "Hello, world!";
    if (mkfifo("myfifo", 0666) < 0 && errno != EEXIST) {
        perror("mkfifo failure");
        exit(1);
    }
    if ((fd = open("myfifo", O_WRONLY)) < 0) {
        perror("open failure");
        exit(2);
    }
    write(fd, msg1, sizeof(msg1));
}
```

# Pipe

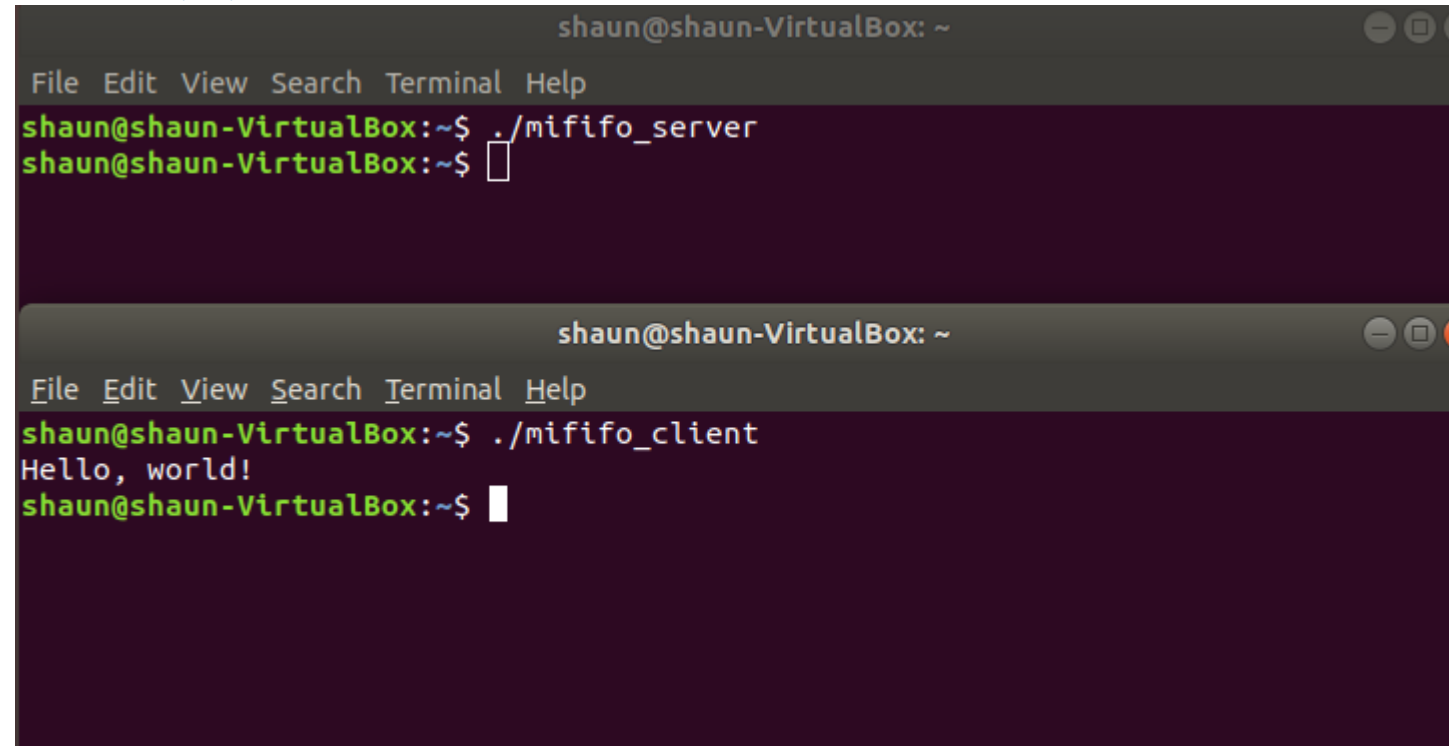
## ❑ Unix/Linux Pipe

- Named pipe example: two processes A & B, process A writes msg to a named pipe and process B reads them from the named pipe.

## ✓ Process B

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>

main() {
    int fd; extern int errno; char inmsg1[20];
    if (mkfifo("myfifo", 0666) < 0 && errno != EEXIST) {
        perror("mkfifo failure");
        exit(1);
    }
    if ((fd = open("myfifo", O_RDWR)) < 0) {
        perror("open failure");
        exit(2);
    }
    read(fd, inmsg1, sizeof(inmsg1));
    printf("%s\n", inmsg1);
}
```



```
shaun@shaun-VirtualBox: ~
File Edit View Search Terminal Help
shaun@shaun-VirtualBox:~$ ./mififo_server
shaun@shaun-VirtualBox:~$

shaun@shaun-VirtualBox: ~
File Edit View Search Terminal Help
shaun@shaun-VirtualBox:~$ ./mififo_client
Hello, world!
shaun@shaun-VirtualBox:~$
```

## □ Unix/Linux signals

### ➤ What is a signal?

- ✓ A short message sent to inform processes of certain events.
- ✓ It is not for data/information exchanging.

### ➤ Who sends a signal to whom

- ✓ Initiated by one process.
- ✓ Interrupt another process(es) or itself.  
Interrupt whatever the process is doing at that particular moment, and force it to respond them immediately.

## □ Unix/Linux signals

### ➤ How to respond?

✓ When a process receives a signal, it can handle it in three ways:

1. By default --- the OS takes care of it
2. User-defined actions --- the process specifies which routine to execute when a certain interrupt signal is received.
3. By ignoring --- no action taken upon receipt of signal.



## □ Unix/Linux signals

### ➤ Different kinds of signals.

- ✓ Each of them are associated with an integer/signal number (1,2,3...).
- ✓ Use the command "kill -l" to see a list of signals.

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL
5) SIGTRAP	6) SIGABRT	7) SIGBUS	8) SIGFPE
9) SIGKILL	10) SIGUSR1	11) SIGSEGV	12) SIGUSR2
13) SIGPIPE	14) SIGALRM	15) SIGTERM	16) SIGSTKFLT
17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU
25) SIGXFSZ	26) SIGVTALRM	27) SIGPROF	28) SIGWINCH
29) SIGIO	30) SIGPWR	31) SIGSYS	34) SIGRTMIN
35) SIGRTMIN+1	36) SIGRTMIN+2	37) SIGRTMIN+3	38) SIGRTMIN+4
39) SIGRTMIN+5	40) SIGRTMIN+6	41) SIGRTMIN+7	42) SIGRTMIN+8
43) SIGRTMIN+9	44) SIGRTMIN+10	45) SIGRTMIN+11	46) SIGRTMIN+12
47) SIGRTMIN+13	48) SIGRTMIN+14	49) SIGRTMIN+15	50) SIGRTMAX-14
51) SIGRTMAX-13	52) SIGRTMAX-12	53) SIGRTMAX-11	54) SIGRTMAX-10
55) SIGRTMAX-9	56) SIGRTMAX-8	57) SIGRTMAX-7	58) SIGRTMAX-6
59) SIGRTMAX-5	60) SIGRTMAX-4	61) SIGRTMAX-3	62) SIGRTMAX-2
63) SIGRTMAX-1	64) SIGRTMAX		

## □ Unix/Linux signals

- Different ways of sending signals—Keyboard  
**Special keys** are interpreted by OS as requests to send signals to the process.

### 1. Ctrl-C

- Send signal **SIGINT** to the running process.
- By default, SIGINT causes the process to immediately **terminate**.

### 2. Ctrl-Z

- send signal **SIGTSTP** to the running process.
- By default, SIGTSTP causes the process to **suspend** execution.



## □ Unix/Linux signals

- Different ways of sending signals—Shell command  
Use various command to send signal to a particular process identified by PID

### 1. `kill -INT 1234 (kill -9 1234)`

- Send signal SIGINT to process with PID=1234.
- By default, SIGINT causes the process to immediately terminate..

### 2. `fg #job`

- Send signal SIGCONT to a previously suspended process.
- By default, SIGCONT causes the process to resume execution.

## ❑ Unix/Linux signals

- Different ways of sending signals—System call  
Use various command to send signal to a particular process identified by PID

### 1. `kill(pid_t, pid, int sig)`

- Use system call "kill" to send signal from process to process.
- e.g. `kill (1234, SIGINT)`.

### 2. `abort()`

- Send `SIGABRT` to itself

### 3. `alarm(int secs)`

- Send `SIGALRM` when the interval expires

## □ Unix/Linux signals

- **Signal handler**—a software interrupt being invoked when the process receives the signal.
  - ✓ When the signal is sent to the process, OS stops the execution of the process, and "forces" it to call the signal handler function.
  - ✓ Signals are not **reliable**
    - ❖ No more than one signal of each type outstanding for a given process.
    - ❖ A signal can be **ignored**.
    - ❖ A signal can be blocked: the signal will not be delivered until it is later **unblocked**.
    - ❖ Two special signals (i.e., **SIGKILL** and **SIGSTOP**) cannot be ignored, blocked, **caught**.

## □ Unix/Linux signals

- **Default signal handler**—OS sets up a set of default signal handlers for your program
- **User-defined signal handler** —Users define their own signal handlers for their programs
  - POSIX signal handling function:

```
int sigaction(int sig, const struct sigaction *act, NULL)
```

1. 1st arg sig specifies any signal in signal(5) **except SIGKILL and SIGSTOP**.
2. if 2nd arg is not NULL, it points to a structure specifying the new action to be taken when delivering sig
3. sigaction is a newer/reliable interface, as opposed to the traditional signal call.