CS2106
Process Management
Inter-Process Communication

Lecture 4

Overview

- Inter-process Communication
  - Motivation
  - Common communication mechanisms
    - Shared memory
    - Message passing
    - Pipe (Unix specific)
    - Signal (Unix specific)

Inter-Process Communication (**IPC**)

- It is hard for cooperating processes to share information
  - Memory space is independent!
  - Inter-Process Communication mechanisms (IPC) is needed

- Two common IPC mechanisms:
  - Shared-Memory and Message Passing

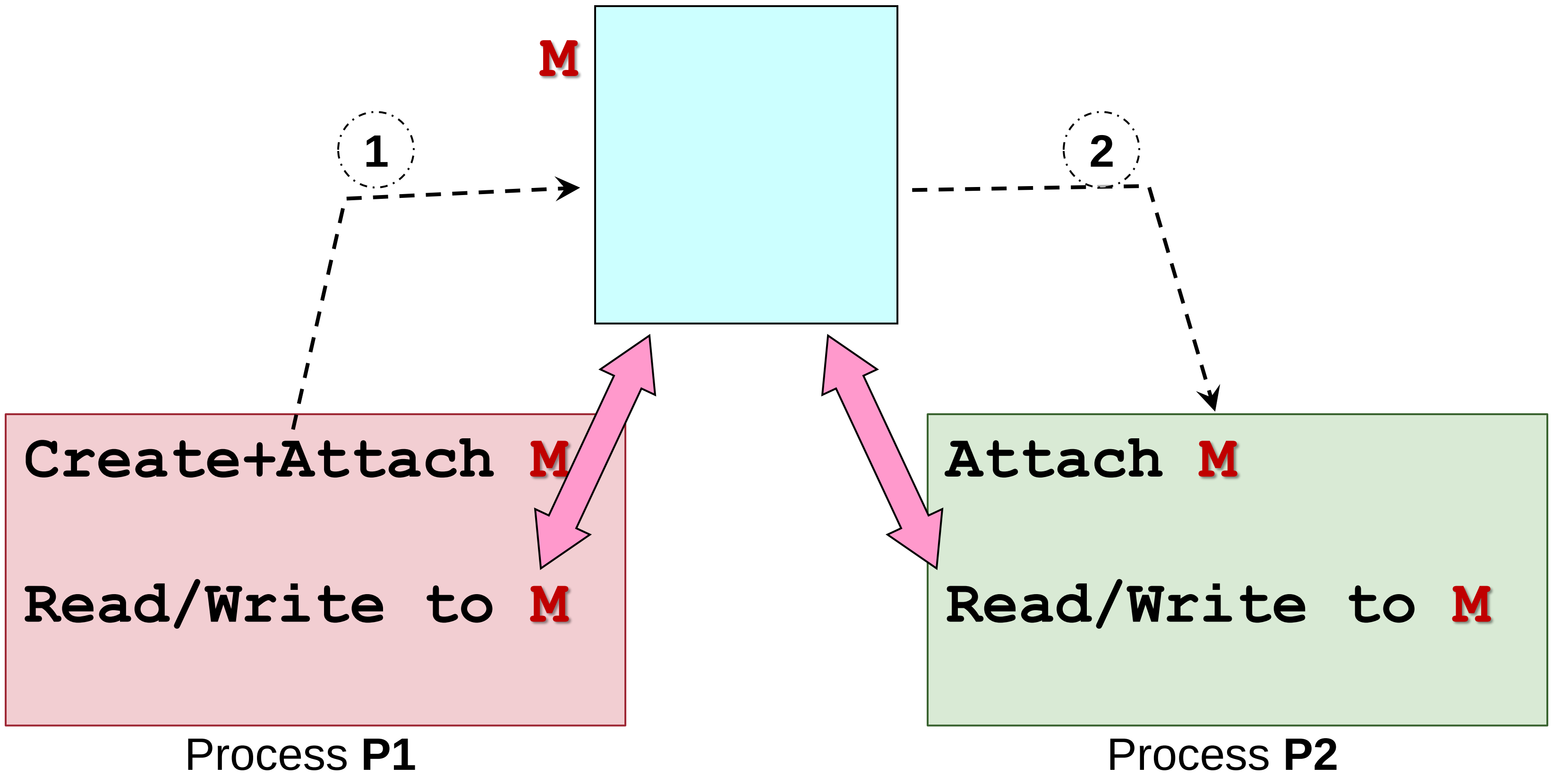
- Two Unix-specific IPC mechanisms:
  - Pipe and Signal

Shared-Memory

- General Idea:
  - Process $\mathbf{P_1}$ creates a shared memory region **M**
  - Process $\mathbf{P_1}$ and $\mathbf{P_2}$ attach memory region **M** to its own memory space
  - $\mathbf{P_1}$ and $\mathbf{P_2}$ can now communicate using memory region M
    - **M** behaves very similar to normal memory region
    - Any writes to the region are visible to the other process

- The same model is applicable to multiple processes sharing the same memory region

Shared-Memory: Illustration



- OS is involved in step 1 and 2 only

Shared-Memory: Pros and Cons

- **Advantages:**
  - Efficient:
    - Only the initial steps (e.g. Create and Attach shared memory region) involves OS
  - Ease of use:
    - Shared memory region behaves the same as normal memory space
    - i.e., information of any type or size can be written easily
- **Disadvantages:**
  - Synchronization:
    - Shared resource → Need to synchronize access (more later)
  - Implementation is usually harder

POSIX Shared Memory in *nix

- Basic steps of usage:
  1. Create/locate a shared memory region **M**
  2. Attach **M** to process memory space
  3. Read from/write to **M**
     - Values written visible to all process that share **M**
  4. Detach **M** from memory space after use
  5. Destroy **M**
     - Only one process need to do this
     - Can only destroy if **M** is not attached to any process

Example: Master program (1/2)

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

int main()
{
    int shmid, i, *shm;

    shmid = shmget( IPC_PRIVATE, 40, IPC_CREAT | 0600);

    if (shmid == -1){
        printf("Cannot create shared memory!\n");
        exit(1);
    } else
        printf("Shared Memory Id = %d\n", shmid);

    shm = (int*) shmat( shmid, NULL, 0 );
    if (shm == (int*) -1){
        printf("Cannot attach shared memory!\n");
        exit(1);
    }
```

The master program create the shared memory region and wait for the "slave" program to produce values before proceeding.

Step 1. Create Shared Memory region.

Step 2. Attach Shared Memory region.

Example: Master program (2/2)

```
    shm[0] = 0;

    while(shm[0] == 0){
        sleep(3);
    }

    for (i = 0; i < 3; i++){
        printf("Read %d from shared memory.\n", shm[i+1]);
    }

    shmdt( (char*) shm);
    shmctl( shmid, IPC_RMID, 0);

    return 0;
}
```

The first element in the shared memory region is used as "control" value in this example (0: values not ready, 1: values ready).

The next 3 elements are values produced by the slave program.

Step 4+5. Detach and destroy Shared Memory region.

Example: Slave program

```
//similar header files
int main()
{   int shmid, i, input, *shm;

    printf("Shared memory id for attachment: ");
    scanf("%d", &shmid);

    shm = (int*)shmat( shmid, NULL, 0);
    if (shm == (int*)-1) {
        printf("Error: Cannot attach!\n");
        exit(1);
    }

    for (i = 0; i < 3; i++){
        scanf("%d", &input);
        shm[i+1] = input;
    }
    shm[0] = 1;

    shmdt( (char*)shm );
    return 0;
}
```

Step 1. By using the shared memory region id directly, we skip ***shmget***() in this case.

Step 2. Attach to shared memory region.

Write 3 values into shm[1 to 3]

Let master program know we are done!

Step 4. Detach Shared Memory region.

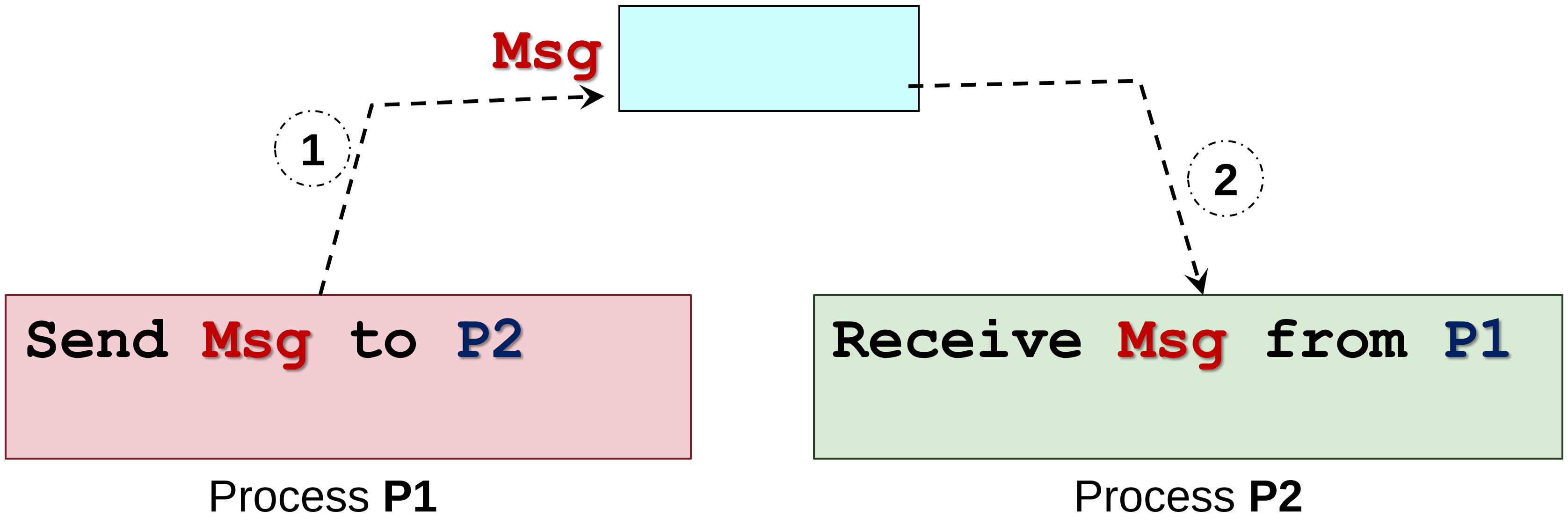
You have 1,023,428 messages waiting…

MESSAGE PASSING

Message Passing

- General Idea:
  - Process $\mathbf{P_1}$ prepares a message **M** and sends it to Process $\mathbf{P_2}$
  - Process $\mathbf{P_2}$ receives the message **M**
  - Message sending and receiving are usually provided as system calls
- Additional properties:
  - **Naming**
    - How to identify the other party in the communication
  - **Synchronization**
    - The behavior of the sending/receiving operations

Message Passing: Illustration



- The **`Msg`** have to be stored in kernel memory space
- Every send/receive operations need to go through OS (i.e., a system call)

Naming Scheme: Direct Communication

- Sender/Receiver of message **explicitly name the other party**
  - Unix: Unix domain socket

- Example:
  - **`Send(`P_2`, Msg)`**: Send Message **`Msg`** to Process P_2
  - **`Receive(`P_1`, Msg)`**: Receive Message **`Msg`** from Process P_1

- Characteristics:
  - One link per pair of communicating processes
  - Need to know the identity of the other party

Naming Scheme: **Indirect Communication**

- Message are sent to / received from message storage:
  - Usually known as ***mailbox*** or ***port***
  - Unix: message queue

- Example:
  - **`Send( MB, Msg )`** : Send Message **`Msg`** to Mailbox **`MB`**
  - **`Receive( MB, Msg )`** : Receive Message **`Msg`** from Mailbox **`MB`**

- Characteristics:
  - One mailbox can be shared among a number of processes

Two Synchronization Behaviors

- **Blocking Primitives** (synchronous):
  - Receive(): Receiver is blocked until a message has arrived
- **Non-Blocking Primitives** (asynchronous):
  - Receive(): Receiver either receive the message if available or some indication that message is not ready yet

Message Passing: Pros and Cons

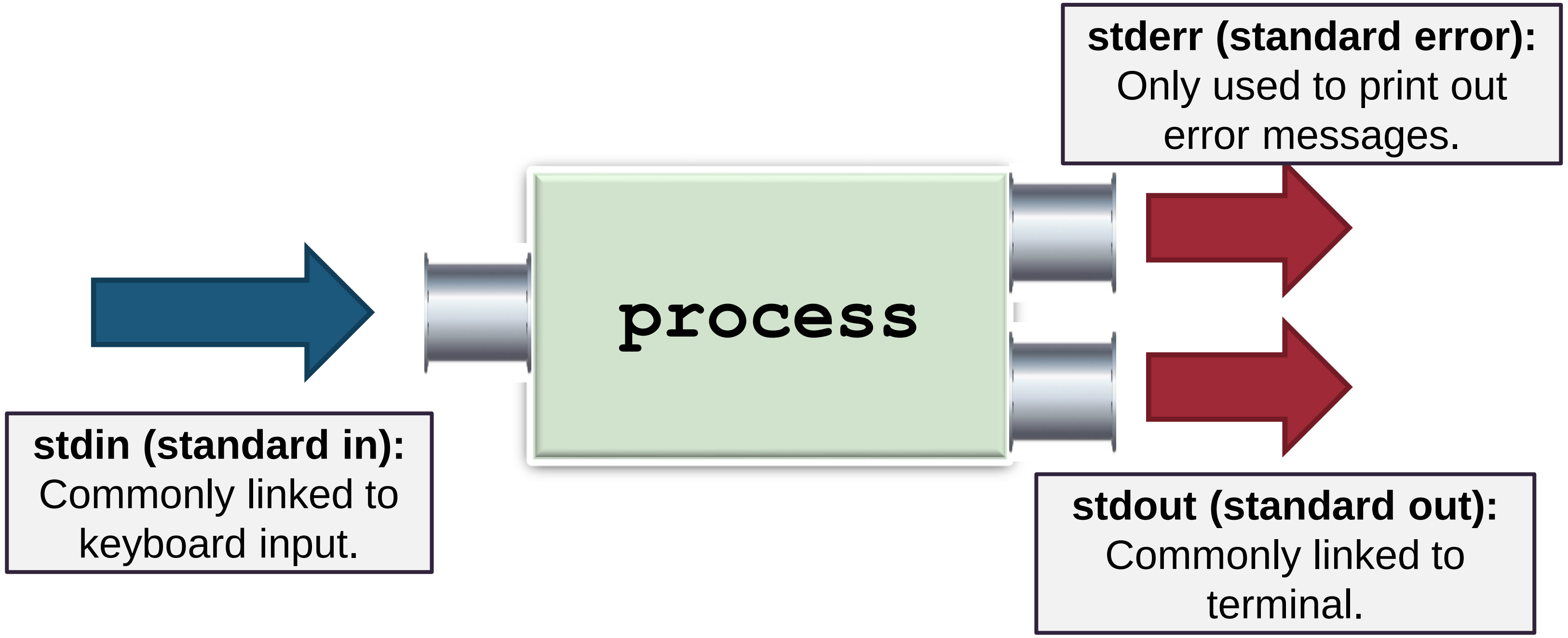
- **Advantages:**
  - Portable:
    - Can easily be implemented on different processing environment, e.g., distributed system, wide area network, etc.
  - Easier synchronization:
    - E.g., when synchronous primitive is used, sender and receiver are implicitly synchronized
- **Disadvantages:**
  - Inefficient:
    - Usually requires OS intervention
    - Extra copying

Plumber needed! Leaking pipes all around!

UNIX PIPES

Pipes: Communication channels

- In Unix, a process has 3 default communication channels:

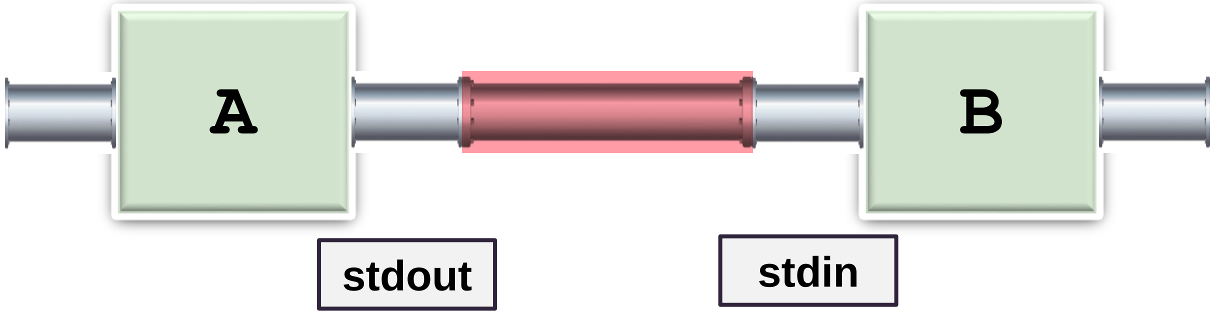


- Example:
  - In a typical C program, printf() uses stdout, scanf() uses stdin.

Piping in Shell

- Unix shell provides the “|” symbol to link the input/output channels of one process to another

- For example (“ A | B”):



- The output of A (instead of going to terminal) directly goes into B as input (as if it come from keyboard)

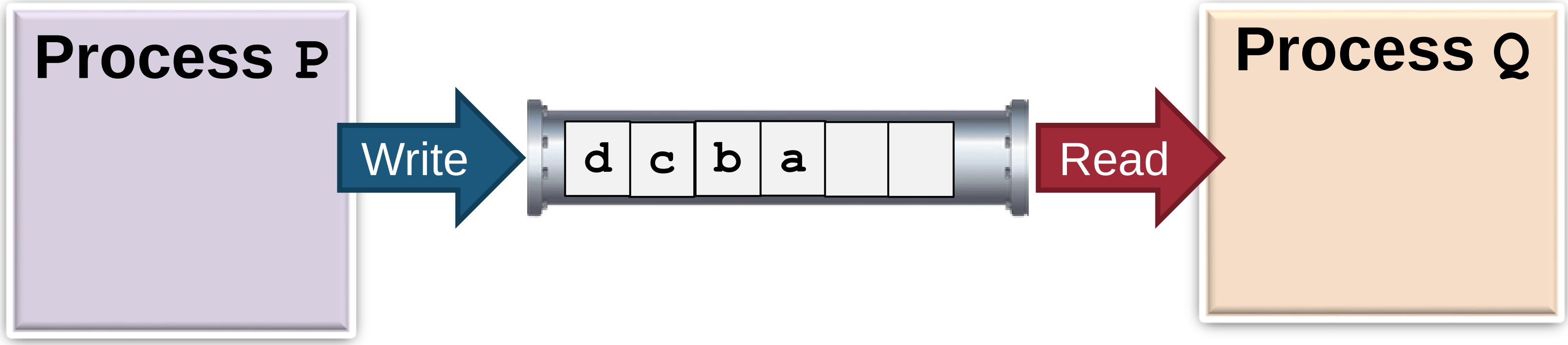
Unix Pipes

- One of the earliest IPC mechanism

- General Idea:
  - A communication channel is created with 2 ends:
    - 1 end for reading, the other for writing
    - Just like a water pipe in the real world

- The piping “|” in shell is achieved using this mechanism internally

Unix Pipes: as an IPC Mechanism



- A pipe can be shared between two processes
- A form of Producer-Consumer relationship
  - P produces (writes) **n** bytes
  - Q consumes (reads) **m** bytes
- Behavior:
  - Like an anonymous file
  - FIFO ➔ must access data in order

Unix Pipes: **Semantic**

- Pipe functions as **circular bounded byte buffer** with **implicit synchronization**:
  - Writers **wait** when buffer is **full**
  - Readers **wait** when buffer is **empty**

- Variants:
  - Can have multiple readers/writers
    - The normal shell pipe has 1 writer and 1 reader
  - Depends on Unix version, pipes may be **half-duplex**
    - **unidirectional**: with one write end and one read end
  - Or **full-duplex**
    - **bidirectional**: any end for read/write

Unix Pipe: **System Calls**

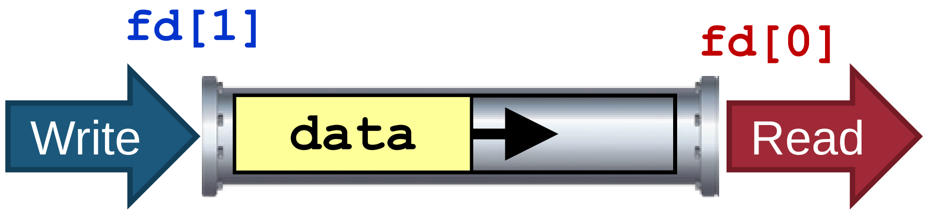
Header File	

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

Syntax

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

- Returns:
  - 0 to indicate success; !0 for errors
  - An array of file descriptors is returned:
    - fd[0] == reading end
    - fd[1] == writing end



Unix Pipes: Example Code

```c
#define READ_END 0
#define WRITE_END 1

int main()
{
   int pipeFd[2], pid, len;
   char buf[100], *str = "Hello There!";

   pipe( pipeFd );
   if ((pid = fork()) > 0) { /* parent */
     close(pipeFd[READ_END]);
     write(pipeFd[WRITE_END], str, strlen(str)+1);
     close(pipeFd[WRITE_END]);
   } else {                    /* child */
     close(pipeFd[WRITE_END]);
     len = read(pipeFd[READ_END], buf, sizeof(buf));
     printf("Proc %d read: %s\n", pid, buf);
     close(pipeFd[READ_END]);
    }
}
```

Unix Pipes: More to explore

- It is possible to:
  - Attach/change the standard communication channels (stdin, stdout, stderr) to one of the pipes
    - Redirect the input/output from one program to another!

- Unix system calls to explore:
  - **`dup()`**
  - **`dup2()`**

- Wikipedia article on **`dup()`** system call has a great program example pssst! pssst!

UNIX SIGNAL

Unix Signal: Quick Overview

- A form of inter-process communication
  - An asynchronous notification regarding an event
  - Sent to a process/thread
- The recipient of the signal must handle the signal by:
  - A default set of handlers OR
  - User supplied handler (only applicable to some signals)
- Common signals in Unix:
  - Kill, Interrupt, Stop, Continue, Memory error, Arithmetic error, etc….

Example: **Custom Signal Handler**

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

void myOwnHandler( int signo )
{
    if (signo == SIGSEGV){
        printf("Memory access blows up!\n");
        exit(1);
    }
}

int main(){
    int *ip = NULL;

    if (signal(SIGSEGV, myOwnHandler) == SIG_ERR)
        printf("Failed to register handler\n");

    *ip = 123;

    return 0;
}
```

User defined function to handle signal. In this example, we handle the "SIGSEGV" signal, i.e., the memory segmentation fault signal.

Register our own code to replace the default handler.

This statement will cause a segmentation fault.

Summary

- Common Inter Process Communication mechanisms:
  - Shared Memory
    - POSIX example
  - Message Passing
  - Unix Pipes
  - Unix Signals

Reference

- Modern Operating System (3rd Edition)
  - Chapter 2.4
- Operating System Concepts (7th Edition)
  - Chapter 5