IPC

Process

- Whenever a command is issued in Linux, a new process is started.
- A program when executed, a special instance is provided by the system to the process. This instance consists of all the services/resources that may be utilized by the process
- Each process is assigned a 5 digit ID number which is known as process ID or PID
- No two processes can exist with the same pid at any point
- Each process also has Parent process id (PPID) that is PID of the process that started it
- A process can exit any time by calling exit() function (stdlib.h needed)

Independant Processes

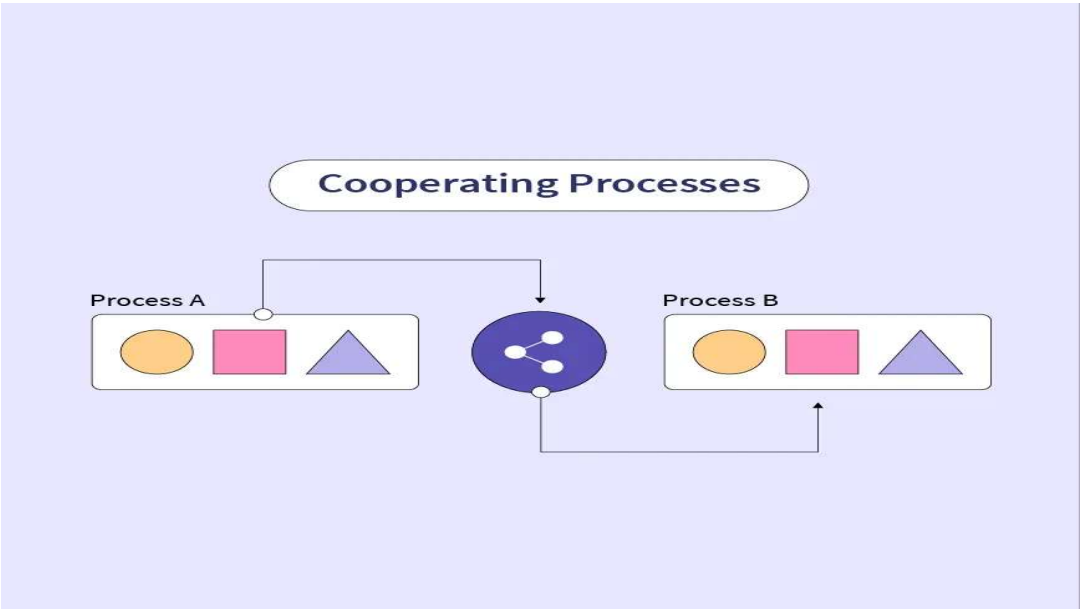
- A process that is independent of the rest of the universe.
- Its state is not shared in any way by any other process.

- Can stop and restart with no bad effects
- Any number of times the process is run with same input, we will get same output (deterministic)
- Example:

    a program to add n natural numbers

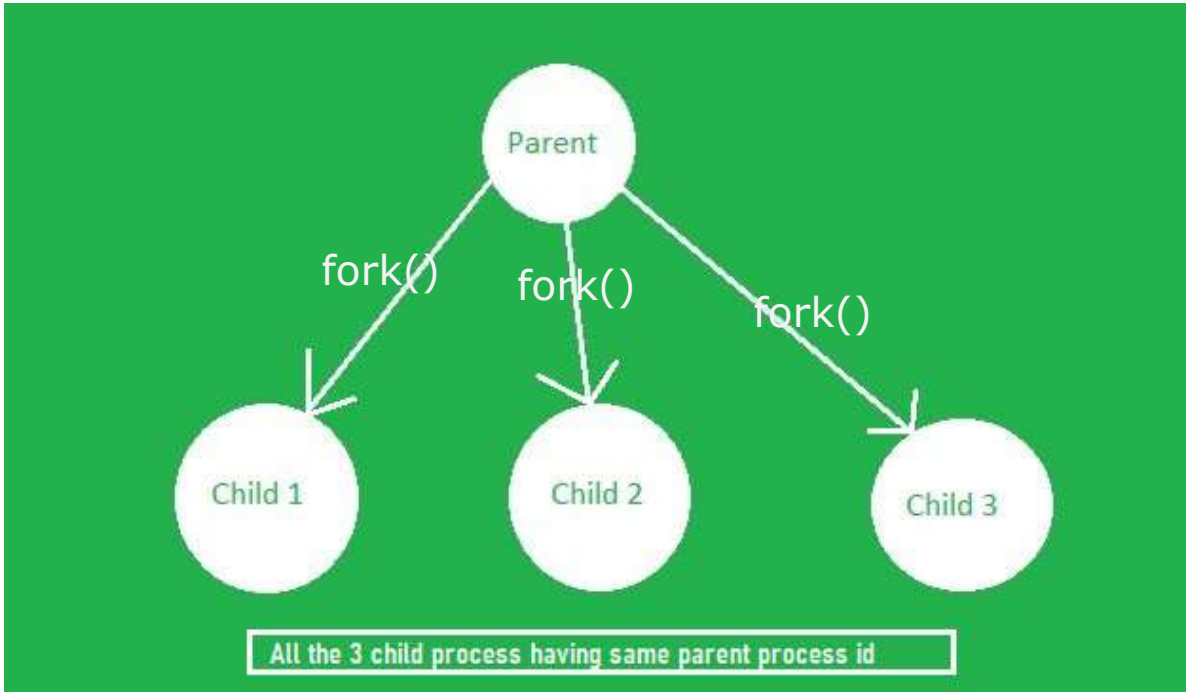
Cooperating Processes

- Cooperating processes are those that share state
- Behavior is *nondeterministic*: depends on relative execution sequence and cannot be predicted
- Example: one process writes "ABC", another writes "CBA".



Parent and Child Processes

- A process can create a child process in Linux with fork() function

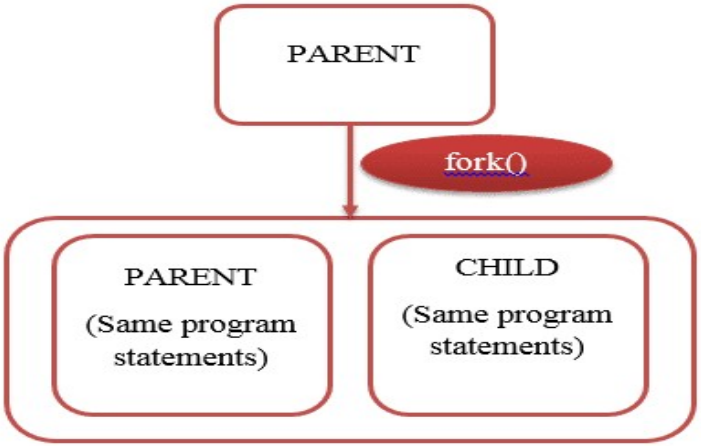


fork() system call

- child process created by calling fork() (unistd.h needed)
- **child process** runs concurrently with the parent process
- a new PCB (Process Control Block) structure allocated for child and most of the parent PCB structure elements are copied into that

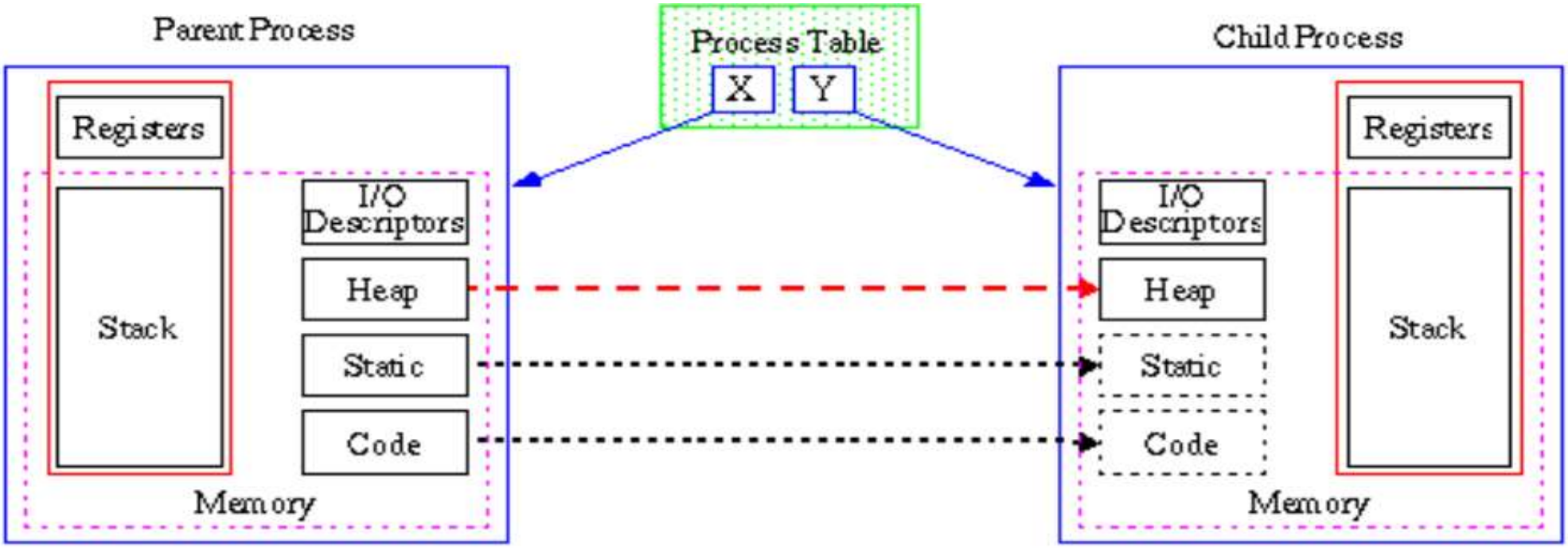
- The fields like parent process ID, process state, process ID of child differs that from the parent

- After a new child process is created, both processes will execute the next instruction following the fork() system call



fork() system call

- To identify the processes we need to use int value returned by for() which is 0 for child and non-zero(child pid) for parent



fork() system call

Zombie Process:

- A process which has finished the execution but still has entry in the process table to report to its parent process is known as a **zombie process**

- The parent process reads the exit status of the child process which reaps off the child process entry from the process table

Orphan Process

- A process whose parent process no more exists (either finished or terminated without waiting for its child process to terminate) is called an orphan process

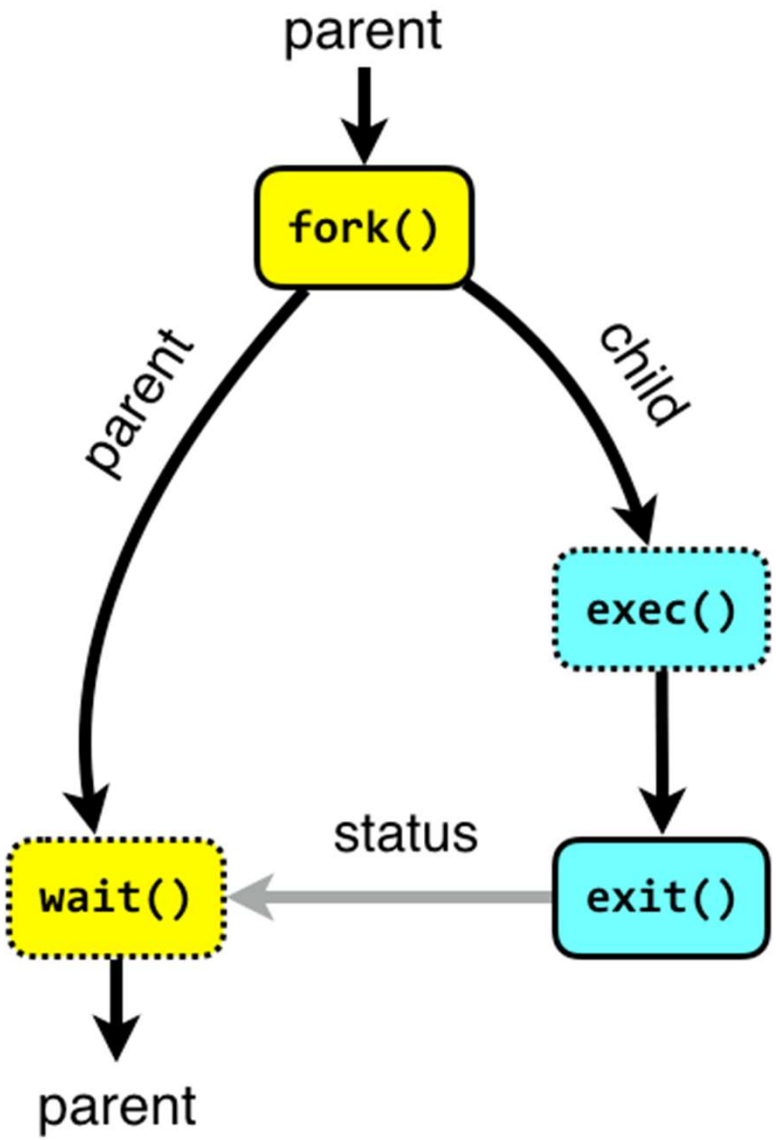
fork() system call sample code

```
int main()
{
int pid;
pid = fork();  // calling fork()
if (pid == 0) {
        printf("Child process with pid: %d, ppid: %d\n", getpid(), getppid());

else
        printf("Parent process with pid: %d, ppid: %d\n", getpid(), getppid());

printf("Process exiting\n");  // this line executed by both parent and child
return 0;
}
```

wait(int *) system call

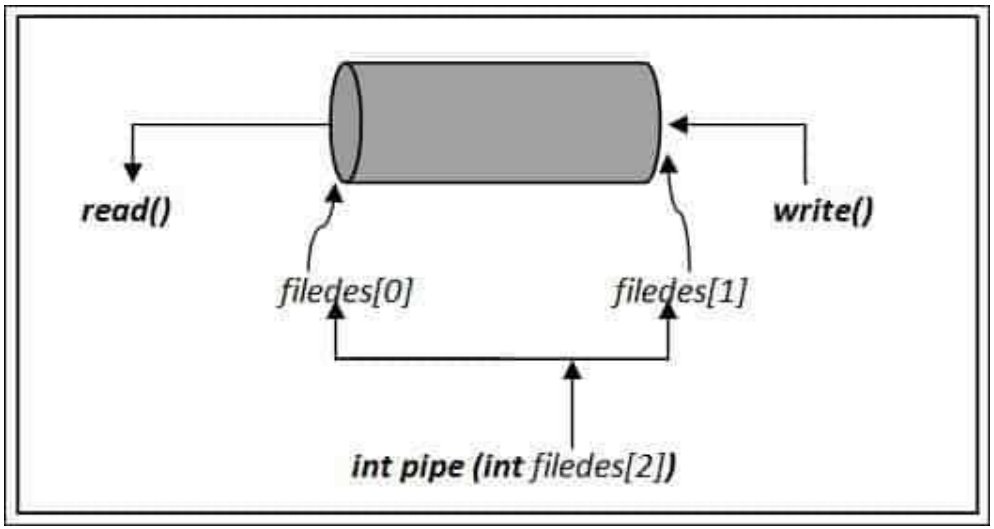
- wait() blocks the calling process until one of its child processes exits
- After the child process terminates, parent ***continues*** its execution after wait system call instruction
- wait() returns pid of child which is terminated
- If no child exists, wait() immediately returns -1
- Child status is stored in pointer of wait argument
- Child process may terminate due to any of these:
    - It calls exit();
    - It returns from main
    - It receives a signal whose default action is to terminate.

wait(int *) system call



pipe() system call

- pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other
- The pipe is a one-way communication only
- We can use a pipe such that One process writes to the pipe, and the other process reads from the pipe
- If a process tries to read before something is written to the pipe, the process is blocked until something is written



pipe() format

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

Parameters :

fd[0] is the fd(file descriptor) for the read end

fd[1] is the fd for the write end of pipe.

Returns :

0 on Success.

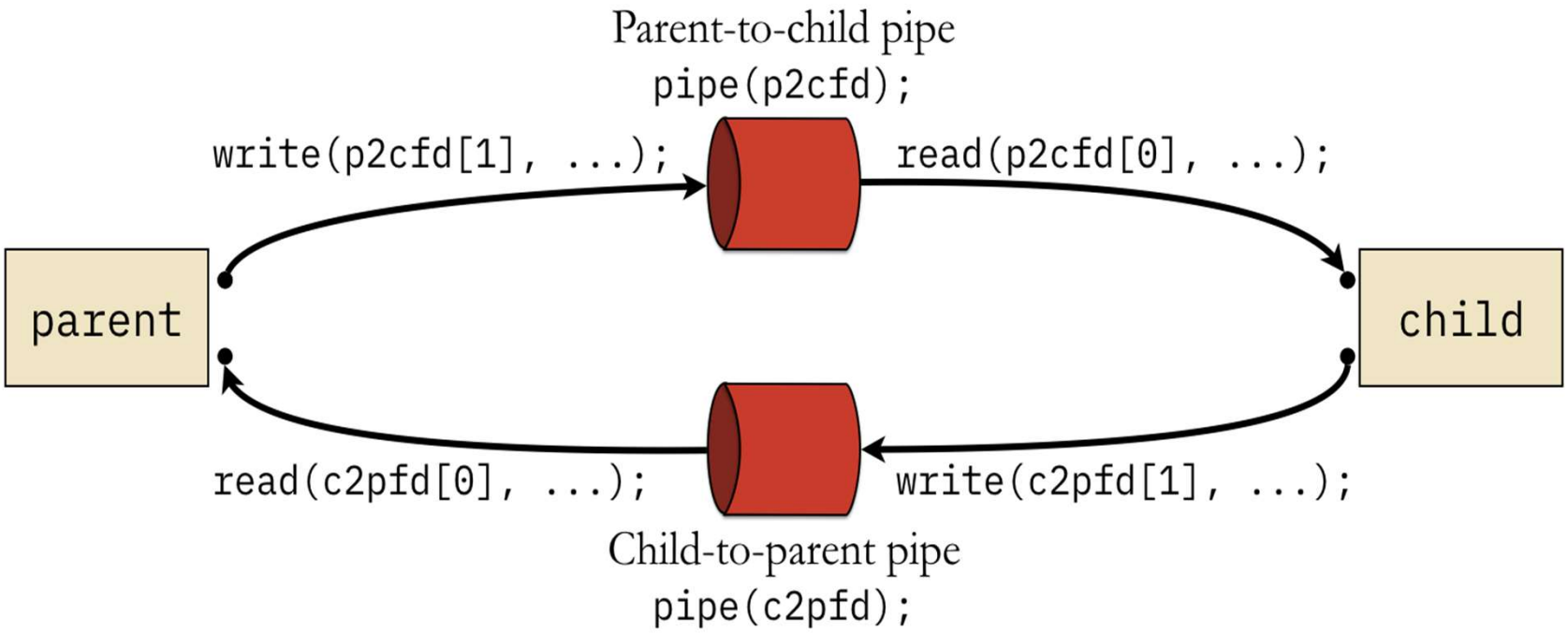
-1 on error

When write end is closed, read returns EOF

Note: Both processes should close unused end after creating pipe

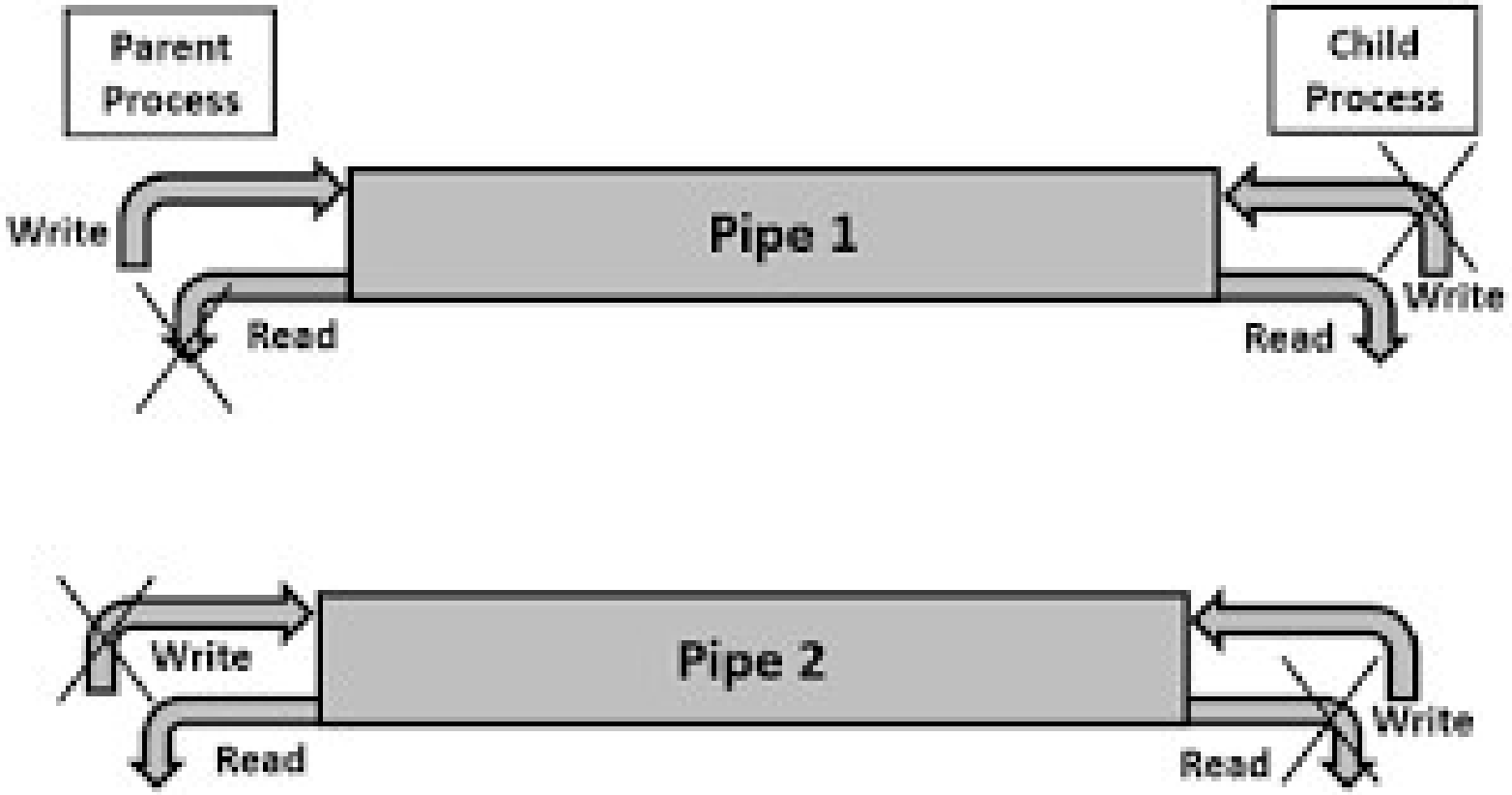
pipe() system call

- For bi-directional data exchange, two pipes created



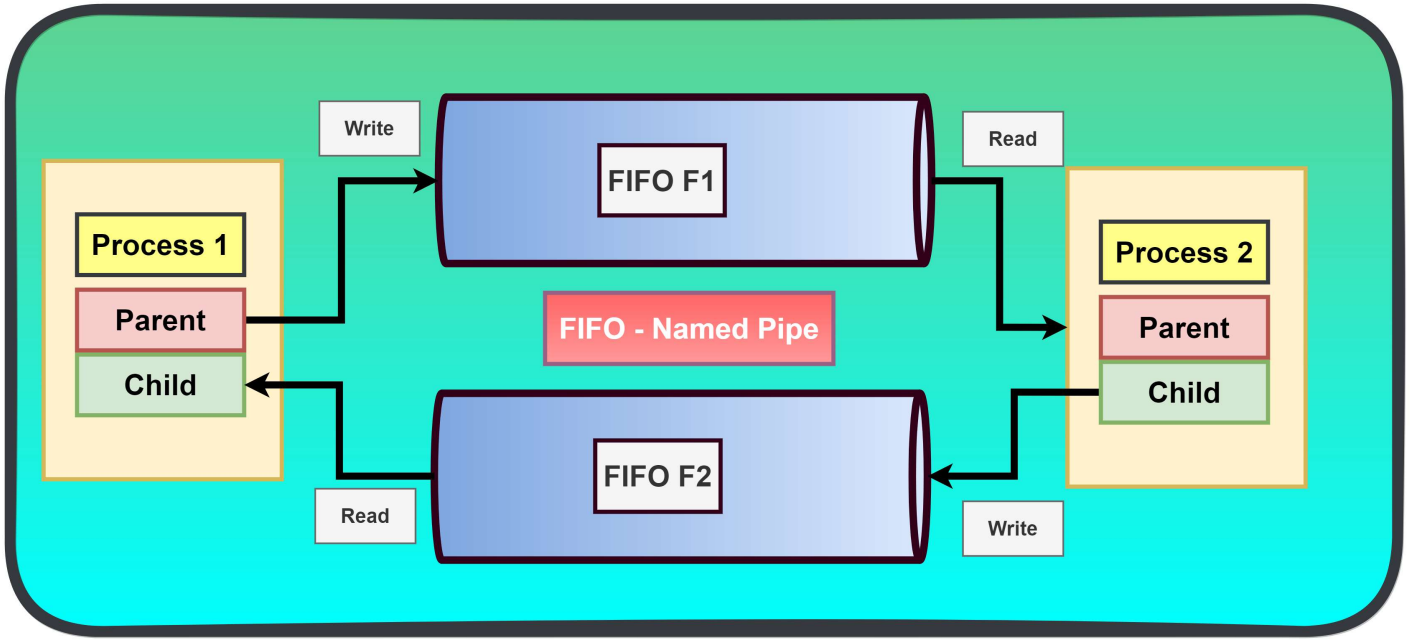
pipe() system call

- For bi-directional data exchange, unused ends of both pipes should be closed on both sides



Named Pipe - FIFO

- FIFO, an acronym for “First In, First Out”, embodies the idea that the first element added is also the first to depart
- named pipes are anchored firmly within the file system (with filename)
- This distinctive feature allows of communication between two unrelated processes



Named Pipe - FIFO - How?

- Create a FIFO by calling makefifo()

  int mkfifo(const char *pathname, mode_t mode);

- mode refers to permissions for the file created (octal value)
- Any process can use open() with modes O_RDONLY, O_WRONLY, O_RDWR etc
- After opening, all file operations like read(), write(), close() can be performed on FIFO

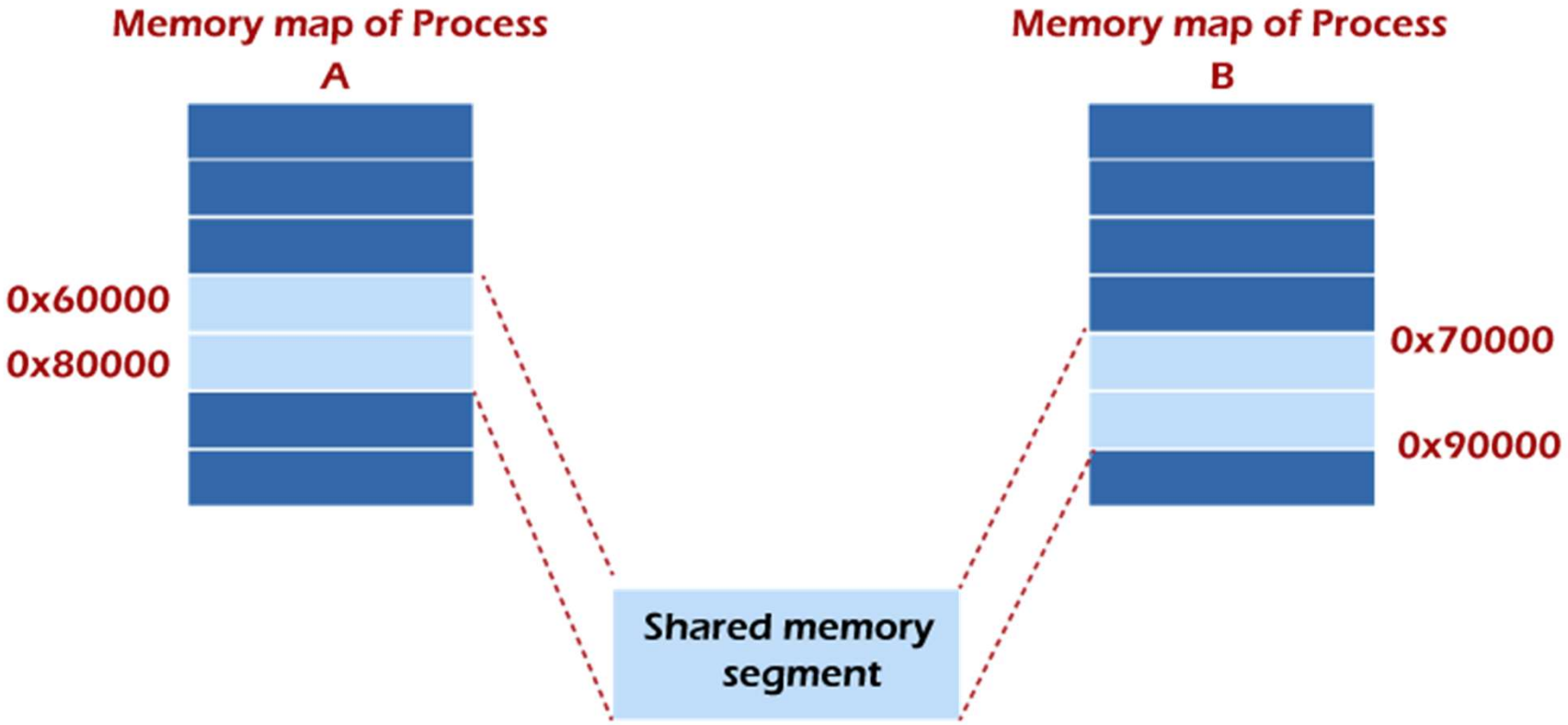
Shared Memory

- Every process has a dedicated address space in order to store data
- If a process wants to share some data with another process, it cannot directly do so since they have different address spaces
- In order to share some data, a process takes up some of the address space as shared memory space
- This shared memory can be accessed by the other process to read/write the shared data

- One process can create SHM (shmget with CREATE option)
- Created shared memory can be attached to its address space(shmat)
- shmat returns the address of the shared memory
- Other process get access to shared memory(shmget)
- Other process can attaches shared memory (shmat)
- Later the shared memory can be detached(shmdt)

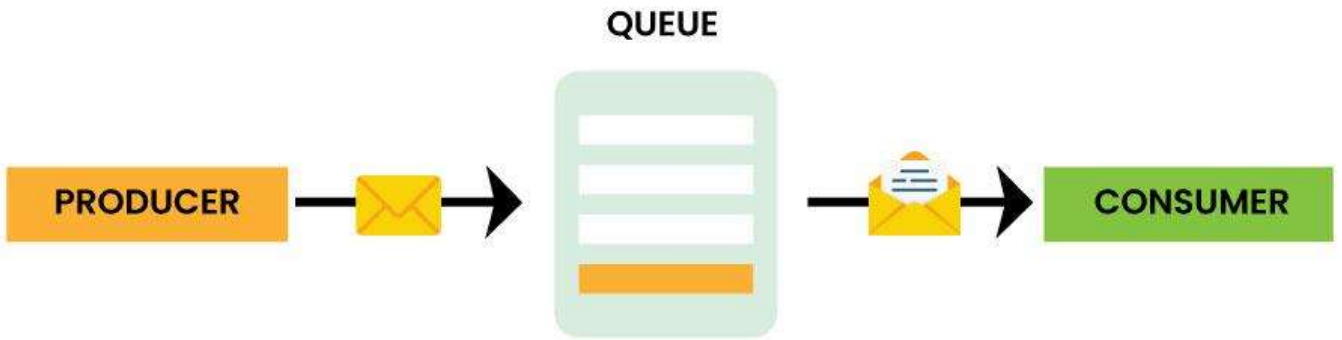
Shared Memory

- Every process has a dedicated address space



Message Queues

- A Message Queue is a form of communication and data transfer mechanism
- For messages sent between various parts, programs, or systems inside a broader software architecture, it serves as a temporary storage and routing system
- Message Queue is implemented as linked list and stored in kernel
- Each Message Queue is identified by MesgQ Id



Message Queue

Message Queue Components

- **Message Producer:** Messages are created and sent to the message queue by the message producer. Any program or part of a system that produces data for sharing can be considered this.

- **Message Queue:** Until the message consumers consume them, the messages are stored and managed by the message queue. It serves as a mediator or buffer between consumers and producers.

- **Message Consumer:** Messages in the message queue must be retrieved and processed by the message consumer. Messages from the queue can be read concurrently by several users.

IPC Message Queues

- A new queue is created or an existing queue is opened by **msgget()**
- New messages are added to the end of a queue by **msgsnd()**
- Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd() when the message is added to a queue
- Messages are fetched from a queue by **msgrcv()**
- We don’t have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field
- **msgctl()** can be used to control message queues. Typically used to destroy the queues

Signals

- A **signal** is a notification to a process indicating the occurrence of an event
- Signal is also called **software interrupt** and is not predictable to know its occurrence, hence it is also called an **asynchronous event**
- Signals are identified as integers
- Signal numbers have symbolic names defined in signal.h

```
#define SIGHUP  1   /* Hang up the process */
#define SIGINT  2   /* Interrupt the process */
#define SIGQUIT 3   /* Quit the process */
#define SIGILL  4   /* Illegal instruction. */
#define SIGTRAP 5   /* Trace trap. */
#define SIGABRT 6   /* Abort. */
```

Signals

- Whenever a signal is raised (either programmatically or system generated signal), a default action is performed
- It is also possible to override the default action and handle the signal or ignore it
- In summary actions performed on signals are:
    - Default action
    - Handle the signal
    - Ignore the signal

signal()

- signal() function can be used to define action
- Ignore the signal – signal(signalnumber,SIG_IGN)

  Eg: signal(SIGINT,SIG_IGN);

- Default action – signal(signalnumber,SIG_DFL)

  Eg: signal(SIGINT,SIG_DFL);

- User defined action = signal(signalnumber, functionname)

  Eg: signal(SIGINT,my_fn);

  my_fn should be void function with int as argument

kill()

- kill() function can be used to send signal to a process
- Format:

  kill(pid,signal)
- Examples:
  - Kill current process:

    kill(getpid(),SIGKILL);
  - Stop child process

    kill(childpid, SIGSTOP);
  - To make child process to continue:

    kill(childpid,SIGCONT);