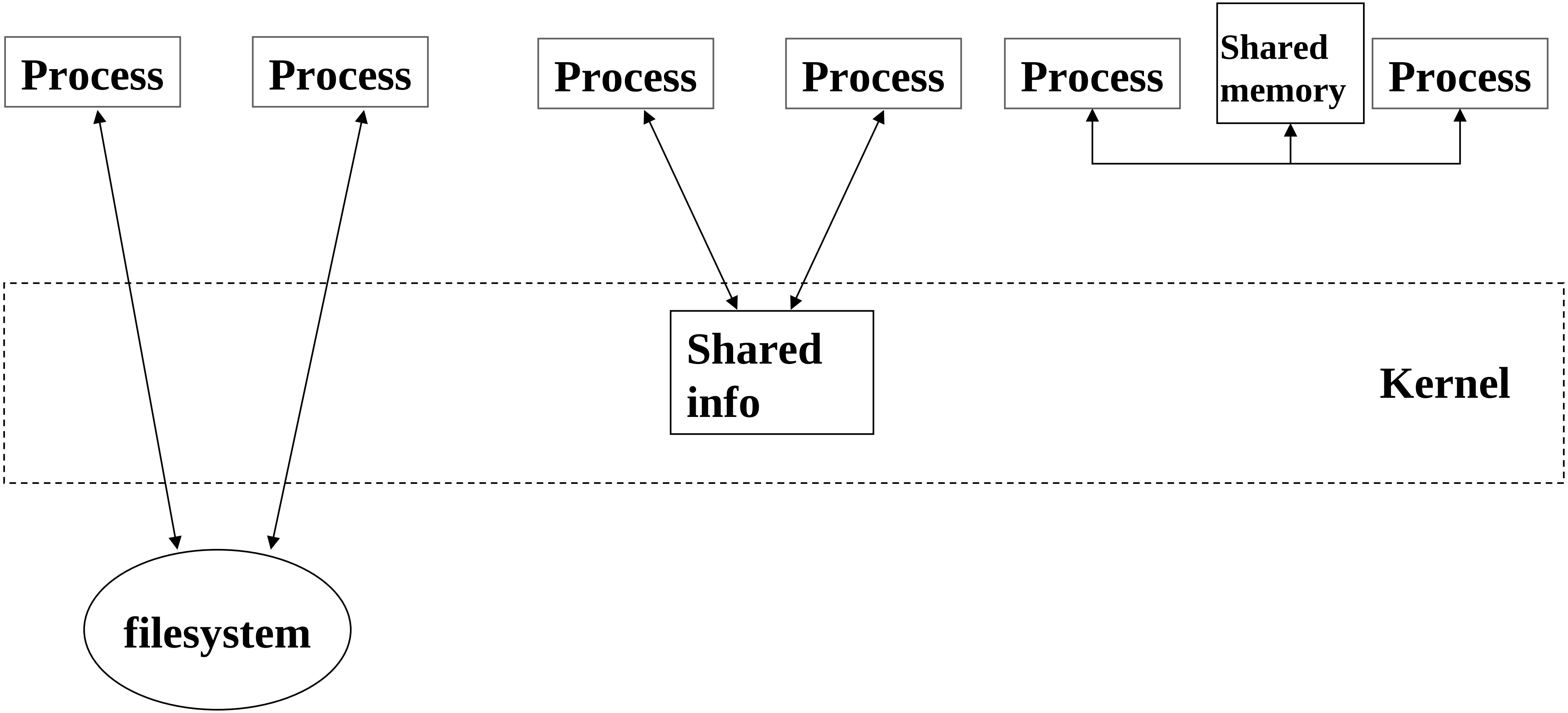
IPC Unix Case Study: Pipes

Unix IPC

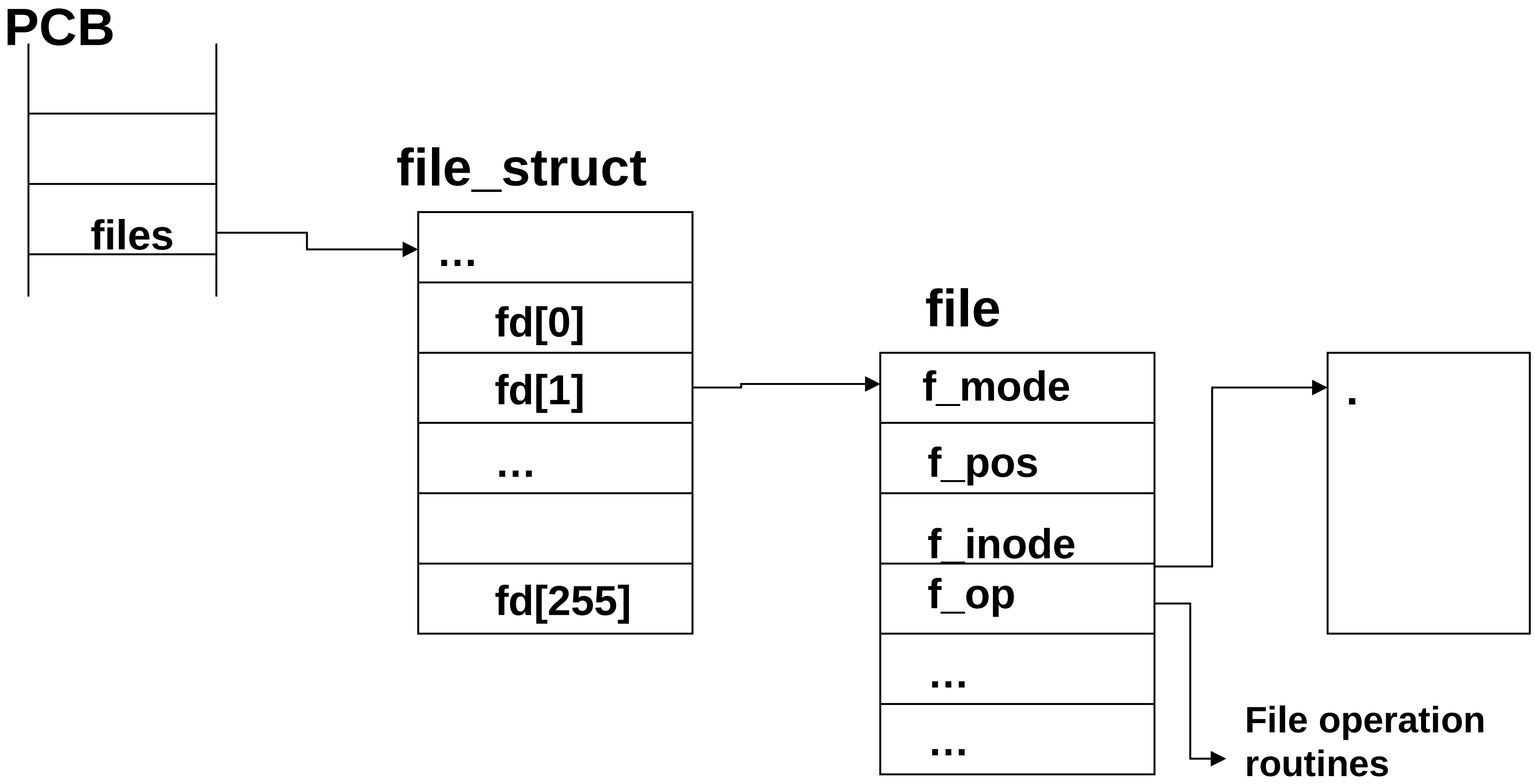

Process
Process
Process
Process
Process
Shared memory
Process
Shared info
Kernel
filesystem

Unix IPC

- Message passing
  - Pipes
  - FIFOs
  - Message queues
- Synchronization
  - Mutexes
  - Condition variables
  - read-write locks
  - Semaphores
- Shared memory
  - Anonymous
  - Named
- Procedure calls
  - RPC
- Event notification
  - Signals

File Descriptors

- The PCB (task_struct) of each process contains a pointer to a file_struct



File Descriptors

- The files_struct contains pointers to file data structures
- Each one describes a file being used by this process.
- f_mode:
  - describes file mode, read only, read and write or write only.
- f_pos:
  - holds the position in the file where the next read or write operation will occur.
- f_inode:
  - points at the actual file

File Descriptors

- Every time a file is opened, one of the free file pointers in the files_struct is used to point to the new file structure.
- Linux processes expect three file descriptors to be open when they start.
- These are known as standard input, standard output and standard error

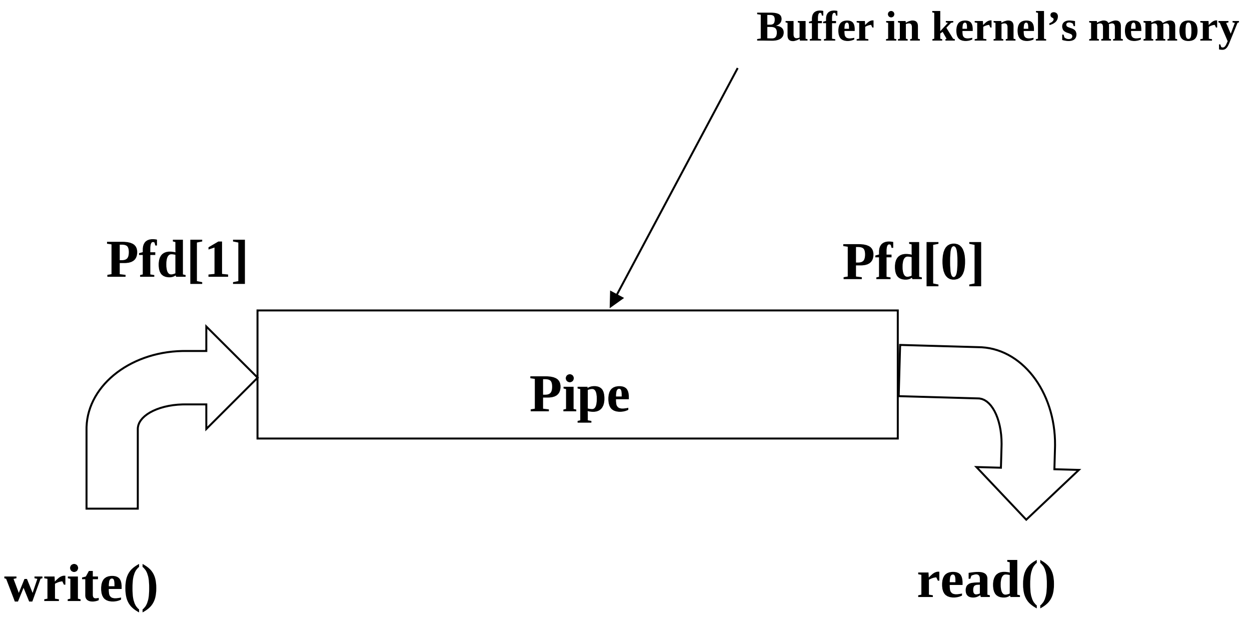
File Descriptors

- The program treat them all as files.
- These three are usually inherited from the creating parent process.
- All accesses to files are via standard system calls which pass or return file descriptors.
- standard input, standard output and standard error have file descriptors 0, 1 and 2.

File Descriptors

- char buffer[10];
- Read from standard input (by default it is keyboard)
  - read(0,buffer,5);
- Write to standard output (by default is is monitor))
  - write(1,buffer,5);
- By changing the file descriptors we can write to files
- fread/fwrite etc are wrappers around the above read/write functions

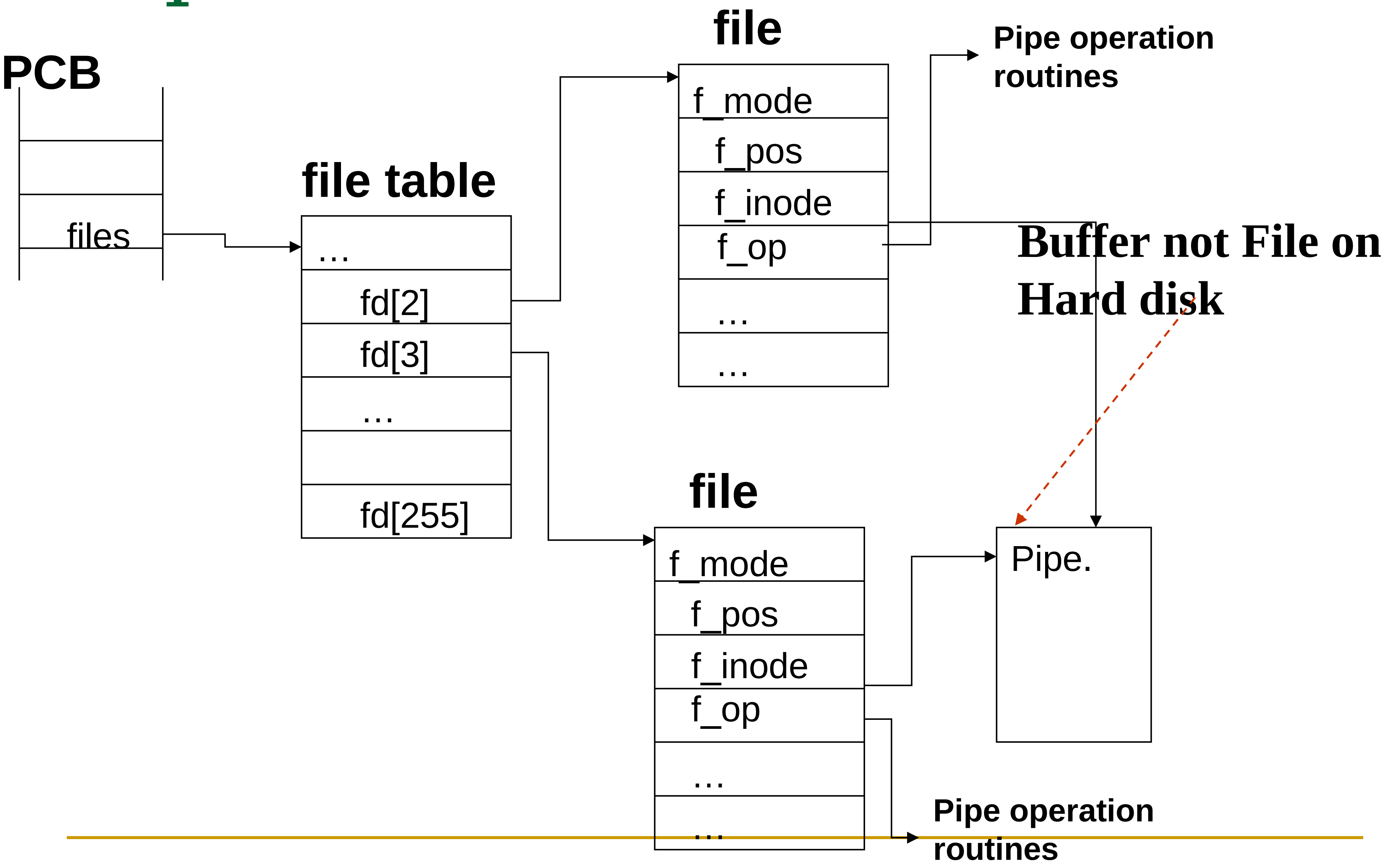
Pipes: Shared info in kernel’s memory



Pipes

- A pipe is implemented using two file data structures which both point at the same temporary data node.
- This hides the underlying differences from the generic system calls which read and write to ordinary files.
- Thus, reading/writing to a pipe is similar to reading/writing to a file

Pipes


PCB
files
file table
…
fd[2]
fd[3]
…
fd[255]
file
f_mode
f_pos
f_inode
f_op
…
…
Pipe operation routines
Buffer not File on Hard disk
file
f_mode
f_pos
f_inode
f_op
…
…
Pipe.
Pipe operation routines

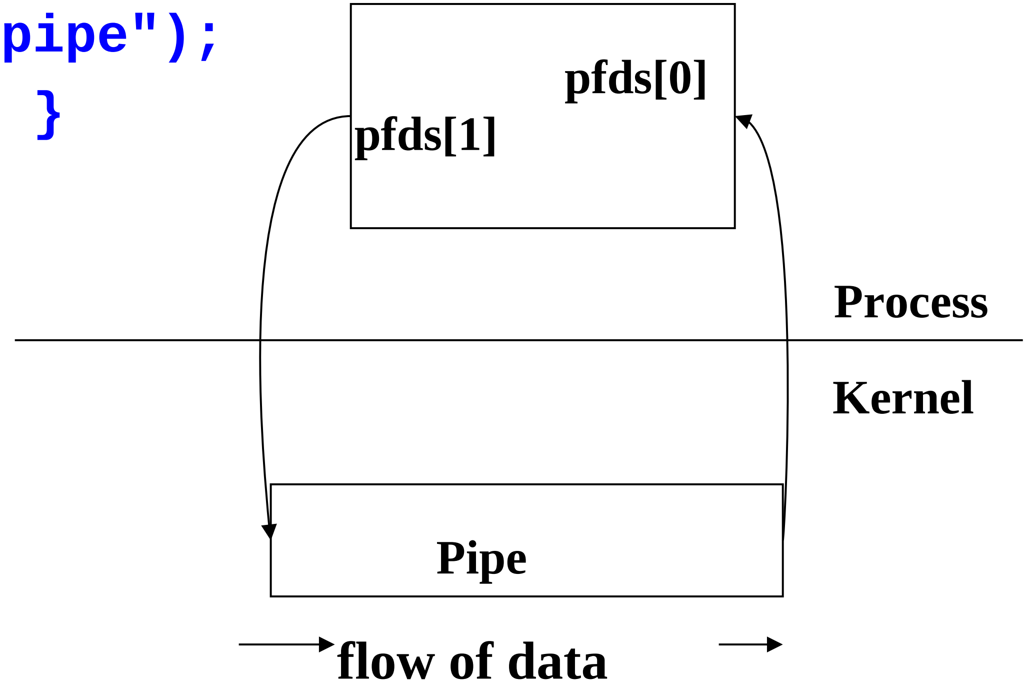
Pipe Creation

- **`#include <unistd.h>`**
- **`int pipe(int`** `filedes`**`[2]);`**
- Creates a pair of file descriptors pointing to a pipe inode
- Places them in the array pointed to by filedes
- filedes[0] is for reading
- filedes[1] is for writing.
- On success, zero is returned.
- On error, -1 is returned

Pipe Creation

```
int main()
{  int pfds[2];

 if (pipe(pfds) == -1)
   {   perror("pipe");
       exit(1); }
 }
```



Reading/Writing from/to a Pipe

- `int read(int filedescriptor, char *buffer, int bytetoread);`
- `int write(int filedescriptor,char *buffer,int bytetowrite);`

Example

```
int main()
{  int pfds[2];
   char buf[30];
   if (pipe(pfds) == -1) {
          perror("pipe");
          exit(1); }
   printf("writing to file descriptor #%d\n", pfds[1]);
   write(pfds[1], "test", 5);
   printf("reading from file descriptor #%d\n",pfds[0]);
   read(pfds[0], buf, 5);
   printf("read %s\n", buf);
}   write(1, "test", 5);????
```

```
    read(0, buf, 5);?????
```

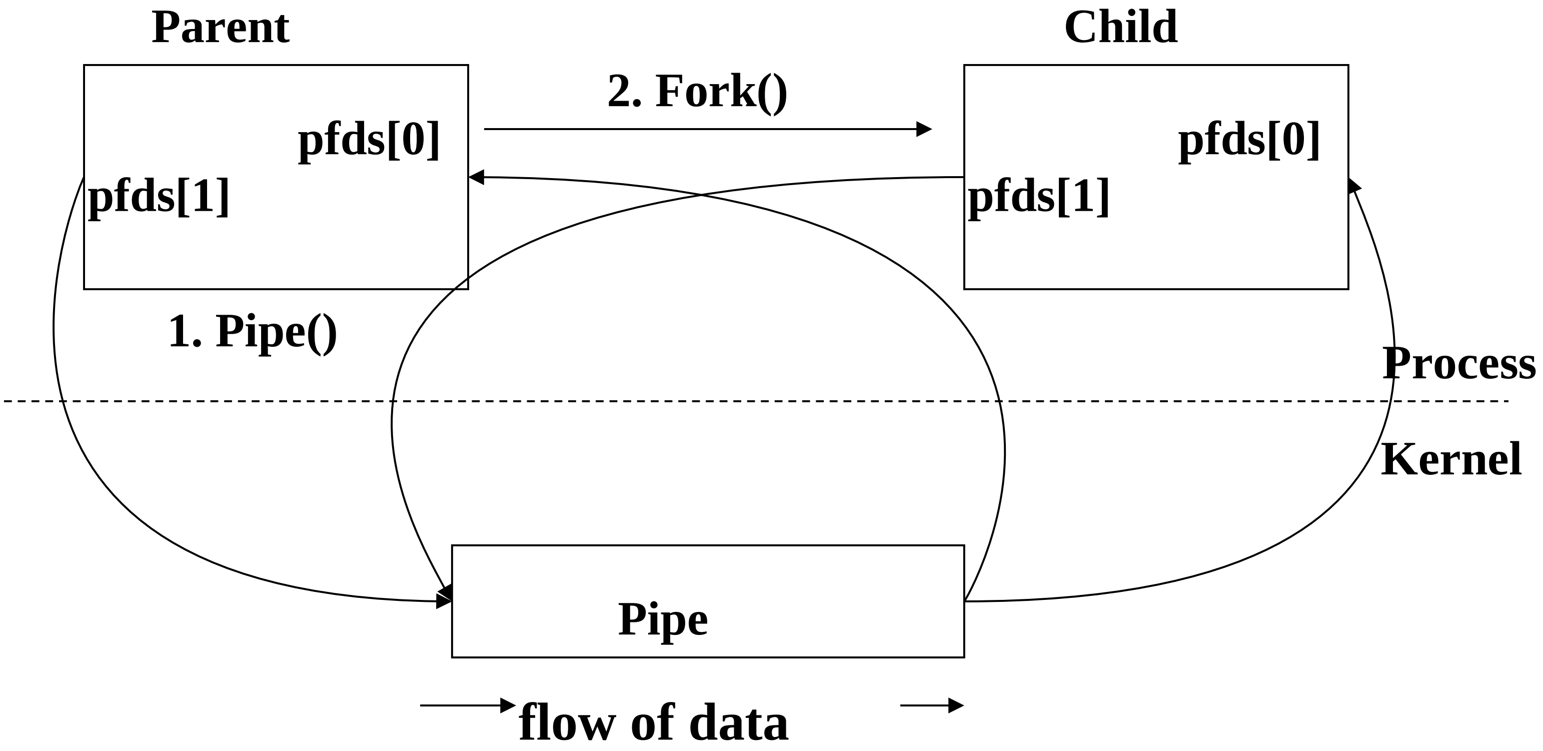
A Channel between two processes

- Remember: the two processes have a parent / child relationship
- The child was created by a fork() call that was executed by the parent.
- The child process is an image of the parent process
- Thus, all the **file descriptors** that are opened by the parent are now available in the child.

A Channel between two processes

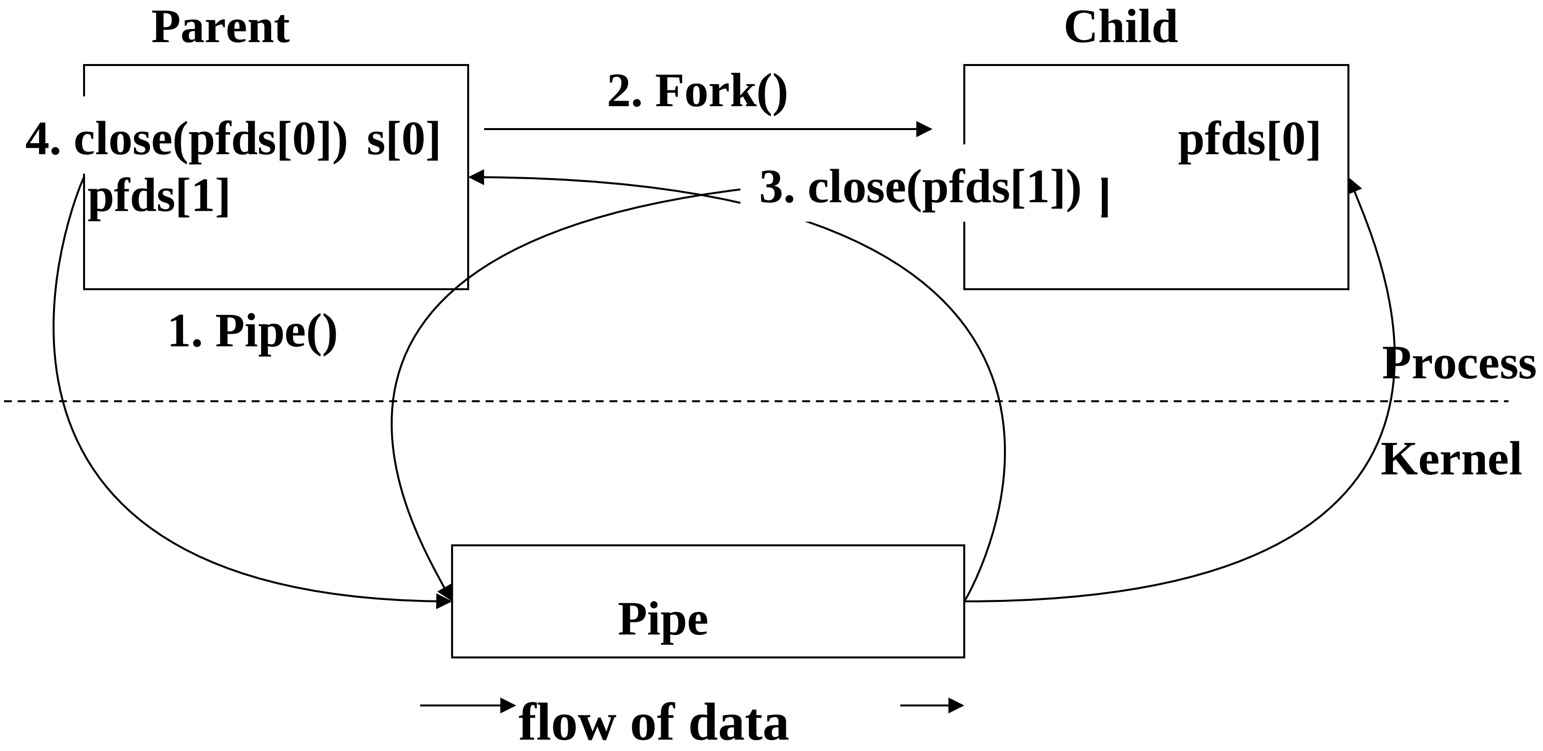
- The **file descriptors** refer to the same I/O entity, in this case a pipe.
- The pipe is inherited by the child
- And may be passed on to the grand-children by the child process or other children by the parent.

A Channel between two processes



A Channel between two processes

- To allow one way communication each process should close one end of the pipe.



Closing the pipe

- The **file descriptors** associated with a pipe can be closed with the close(fd) system call
- How would we achieve two way communication?

An Example of pipes with fork

```
int main() {
    int pfds[2];
    char buf[30];
    pipe(pfds);……………………………………………1
    if (!fork()) ……………………………………………2
   { close(pfds[0]);…………………………………………3
      printf(" CHILD: writing to the pipe\n");
      write(pfds[1], "test", 5);
      printf(" CHILD: exiting\n");
      exit(0);
   }
    else { close(pfds[1]);………………………………………………4
              printf("PARENT: reading from pipe\n");
         read(pfds[0], buf, 5);
         printf( "PARENT: read \"%s\"\n", buf);
         wait(NULL);
     }
  }
```