National University of Computer and Emerging Sciences

Spring 2023

Inter-Process Communication

Interprocess Communication

- A process has access to the memory which constitutes its own address space.
- When a child process is created, the only way to communicate between a parent and a child process is:
  - The variables are replicas
  - The parent receives the exit status of the child
- So far, we've discussed communication mechanisms only during process creation/termination
- Processes may need to communicate during their life time.

Cooperating Processes

- *Independent* process cannot affect or be affected by the execution of another process.
- *Cooperating* process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up:
    - make use of multiple processing elements
  - Modularity
  - Convenience:
    - editing, printing, compiling in parallel
- Dangers of process cooperation
  - Data corruption, deadlocks, increased complexity
  - Requires processes to synchronize their processing

Purposes for IPC

- IPC allows processes to communicate and synchronize their actions without sharing the same address space
  - Data Transfer
  - Sharing Data
  - Event notification
  - Resource Sharing and Synchronization

IPC Mechanisms

- Mechanisms used for communication and synchronization
  - Message Passing
    - message passing interfaces, mailboxes and message queues
    - sockets, pipes
  - *Shared Memory*: Non-message passing systems
  - *Synchronization* – primitives such as semaphores to higher level mechanisms such as monitors
  - *Event Notification* - UNIX *signals*
- We will defer a detailed discussion of synchronization mechanisms and concurrency until a later class
- Here we want to focus on some common (and fundamental) IPC mechanisms

Message Passing

- In a *Message system* there are no shared variables.
- IPC facility provides two operations for fixed or variable sized message:
  - *send(message)*
  - *receive(message)*
- If processes *P* and *Q* wish to communicate, they need to:
  - establish a *communication link*
  - exchange messages via *send and receive*
- Implementation of communication link
  - physical (e.g., memory, network etc)
  - logical (e.g., syntax and semantics, abstractions)

Message Passing Systems

- Exchange messages over a communication link
- Methods for implementing the communication link and primitives (*send*/*receive*):
  1. Direct or Indirect communications (*Naming*)
  2. Symmetric or Asymmetric communications (blocking versus non-blocking)
  3. Buffering
  4. Send-by-copy or send-by-reference
  5. fixed or variable sized messages

Direct Communication – Internet and Sockets

- Processes must name each other *explicitly*:
  - Symmetric Addressing
    - *send (**P**, message)* – send to process P
    - *receive(**Q**, message)* – receive from Q
  - Asymmetric Addressing
    - *send (**P**, message)* – send to process P
    - *receive(**id**, message)* – rx from any; system sets id = sender
- Properties of communication link
  - Links established automatically between pairs
  - processes must know each others ID
  - Exactly one link per pair of communicating processes
- Disadvantage: a process must know the name or ID of the process(es) it wishes to communicate with

Indirect Communication

- Messages are sent to or received from *mailboxes* (also referred to as *ports*).
  - Each mailbox has a unique id.
  - Processes can communicate only if they share a mailbox.
- Primitives:
  - *send(A, message)* – send a message to mailbox A
  - *receive(A, message)* – receive a message from mailbox A
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with more than 2 processes.
  - Each pair of processes may share several communication links.

Indirect Communication- Ownership

- process owns (i.e. mailbox is implemented in user space):
  - only the owner may receive messages through this mailbox.
  - Other processes may only send.
  - When process terminates any “owned” mailboxes are destroyed.
- kernel owns
  - then mechanisms provided to create, delete, send and receive through mailboxes.
  - Process that creates mailbox owns it (and so may receive through it)
  - but may transfer ownership to another process.

Indirect Communication

- Mailbox sharing:
  - P_1, P_2, and P_3 share mailbox A.
  - P_1, sends; P_2 and P_3 receive.
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes.
  - OR Allow only one process at a time to execute a receive operation.
  - OR Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was

Synchronization

- Message passing may be either *blocking* or *non-blocking*.
  - blocking send:
    - sender blocked until message received by mailbox or process
  - nonblocking send:
    - sender resumes operation immediately after sending
  - blocking receive:
    - receiver blocks until a message is available
  - nonblocking receive:
    - receiver returns immediately with either a valid or null message.

Buffering

- All messaging system require framework to temporarily buffer messages.
- These queues are implemented in one of three ways:

1. Zero capacity

    No messages may be queued within the link, requires sender to block until receiver retrieves message.

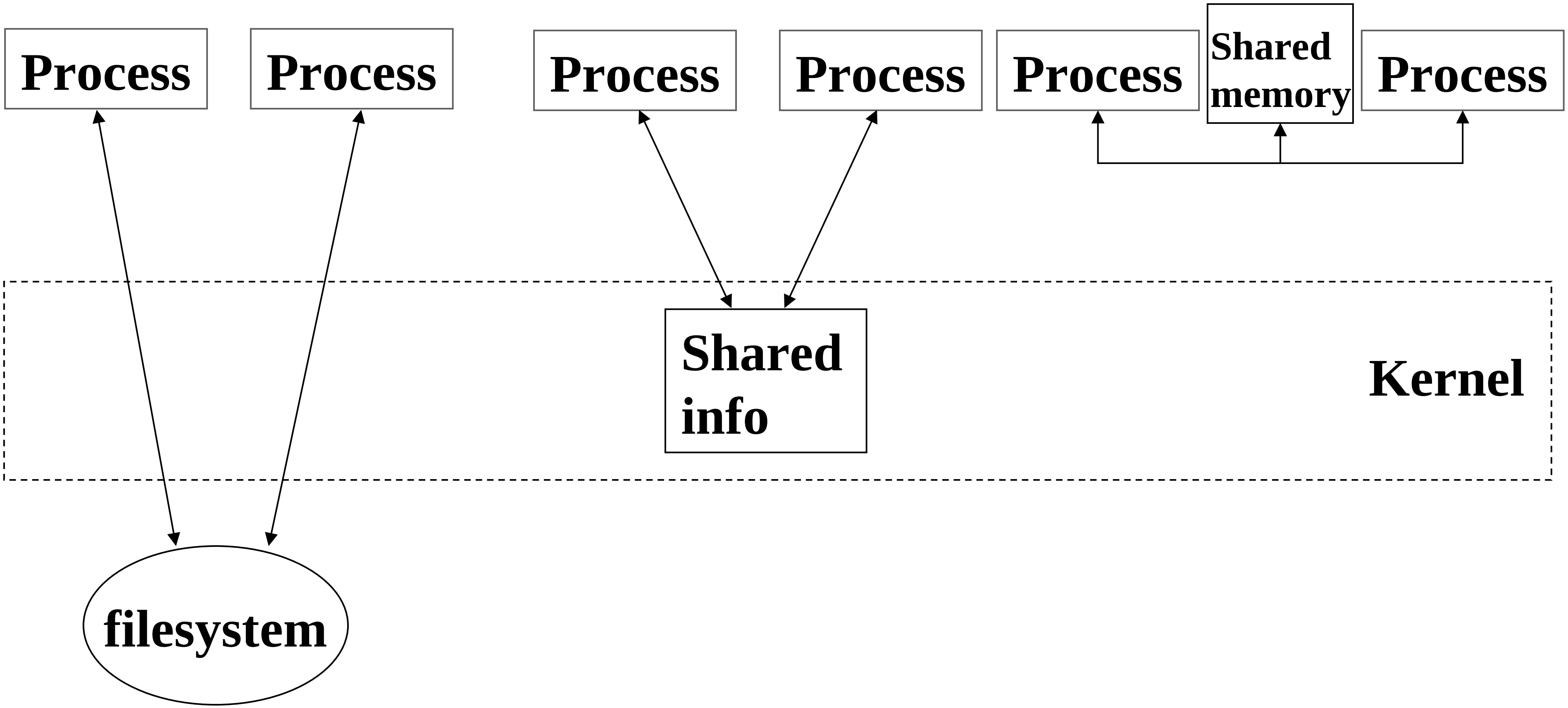
2. Bounded capacity

    Link has finite number of message buffers. If no buffers are available then sender must block until one is freed up.

3. Unbounded capacity

    Link has unlimited buffer space, consequently send never needs to block.

Unix IPC



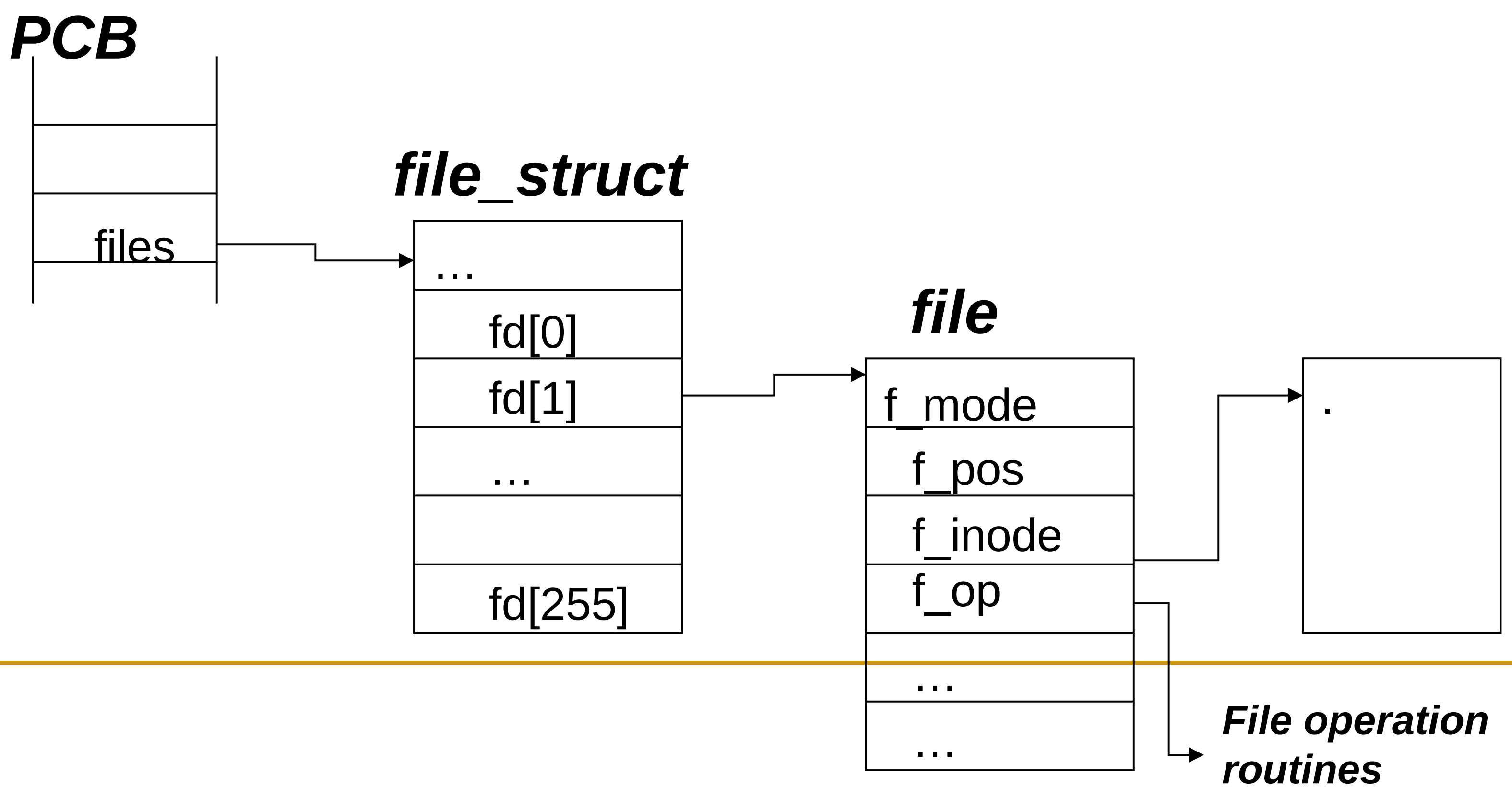
Process
Process
Process
Process
Process
Shared memory
Process
Shared info
Kernel
filesystem

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*

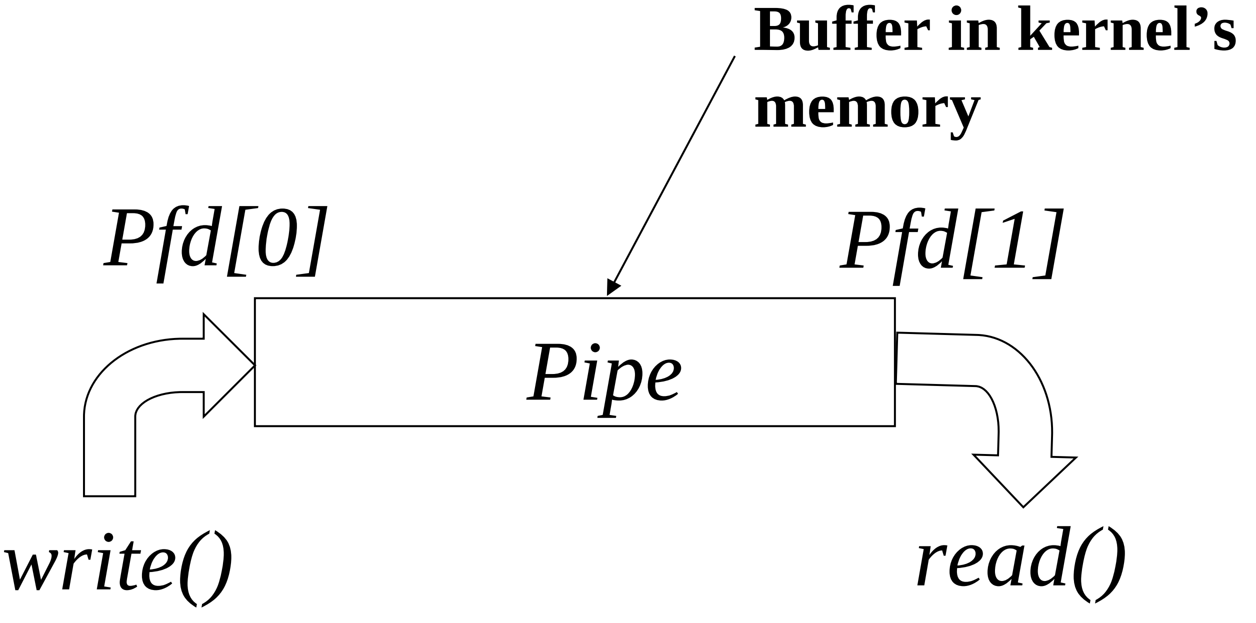
File Descriptors

- 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



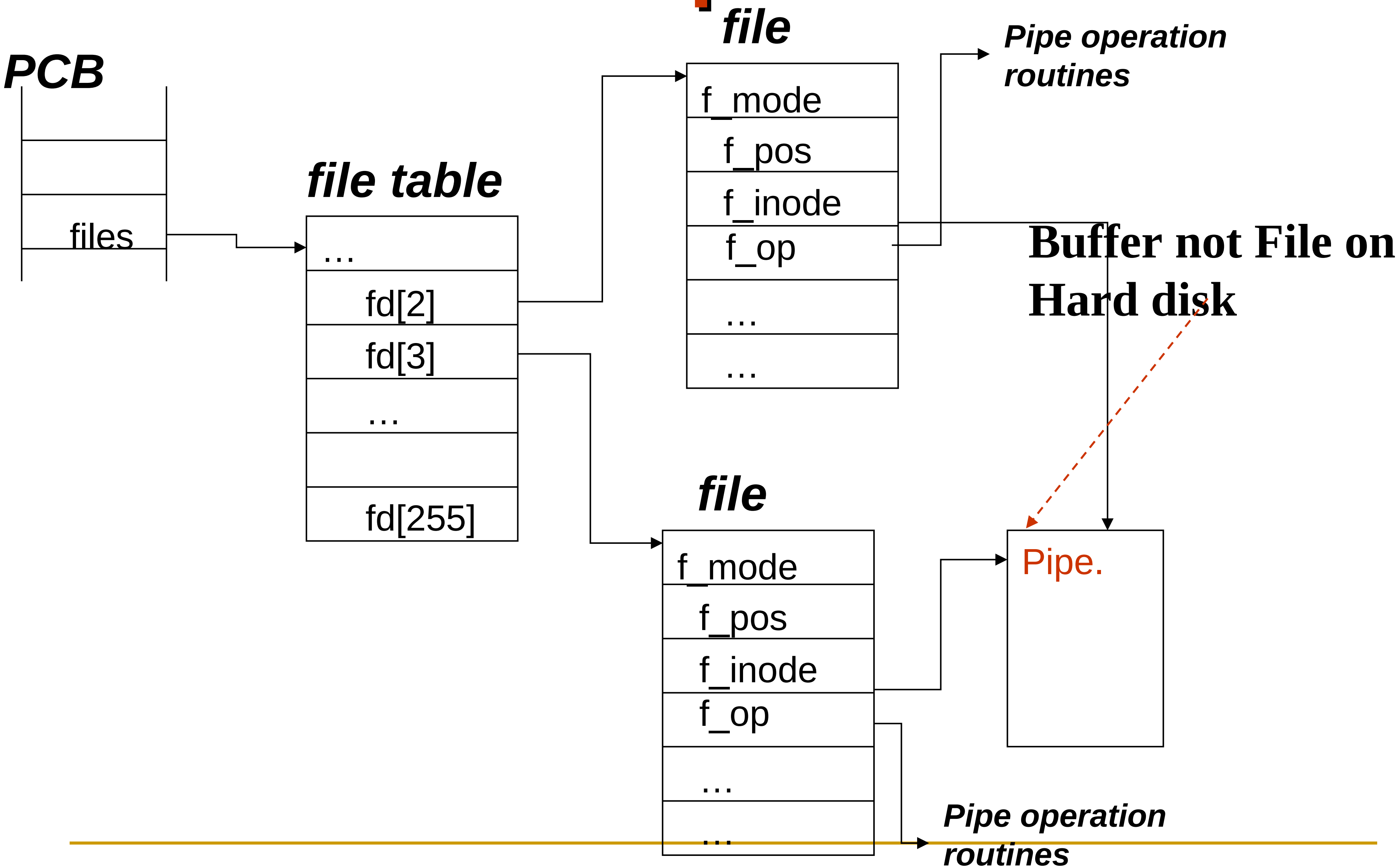
Pipes

- 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

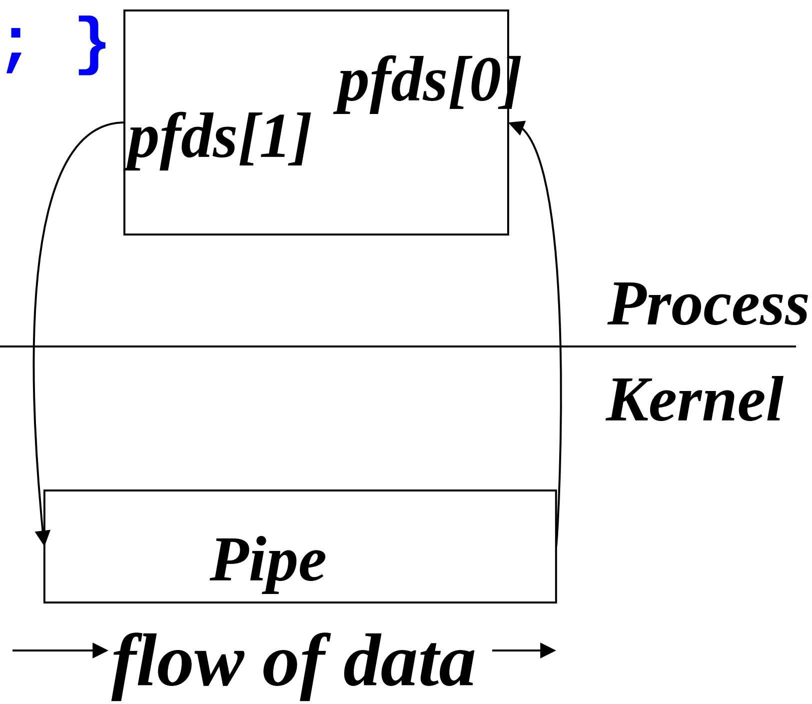

file
Pipe operation routines
PCB
f_mode
f_pos
file table
f_inode
files
Buffer not File on Hard disk
f_op
…
fd[2]
…
fd[3]
…
…
file
fd[255]
Pipe.
f_mode
f_pos
f_inode
f_op
…
…
Pipe operation routines

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);
   write(1, "test", 5);????
  printf("reading from file descriptor #%d\
  n",pfds[0]);
  read(pfds[0], buf, 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.

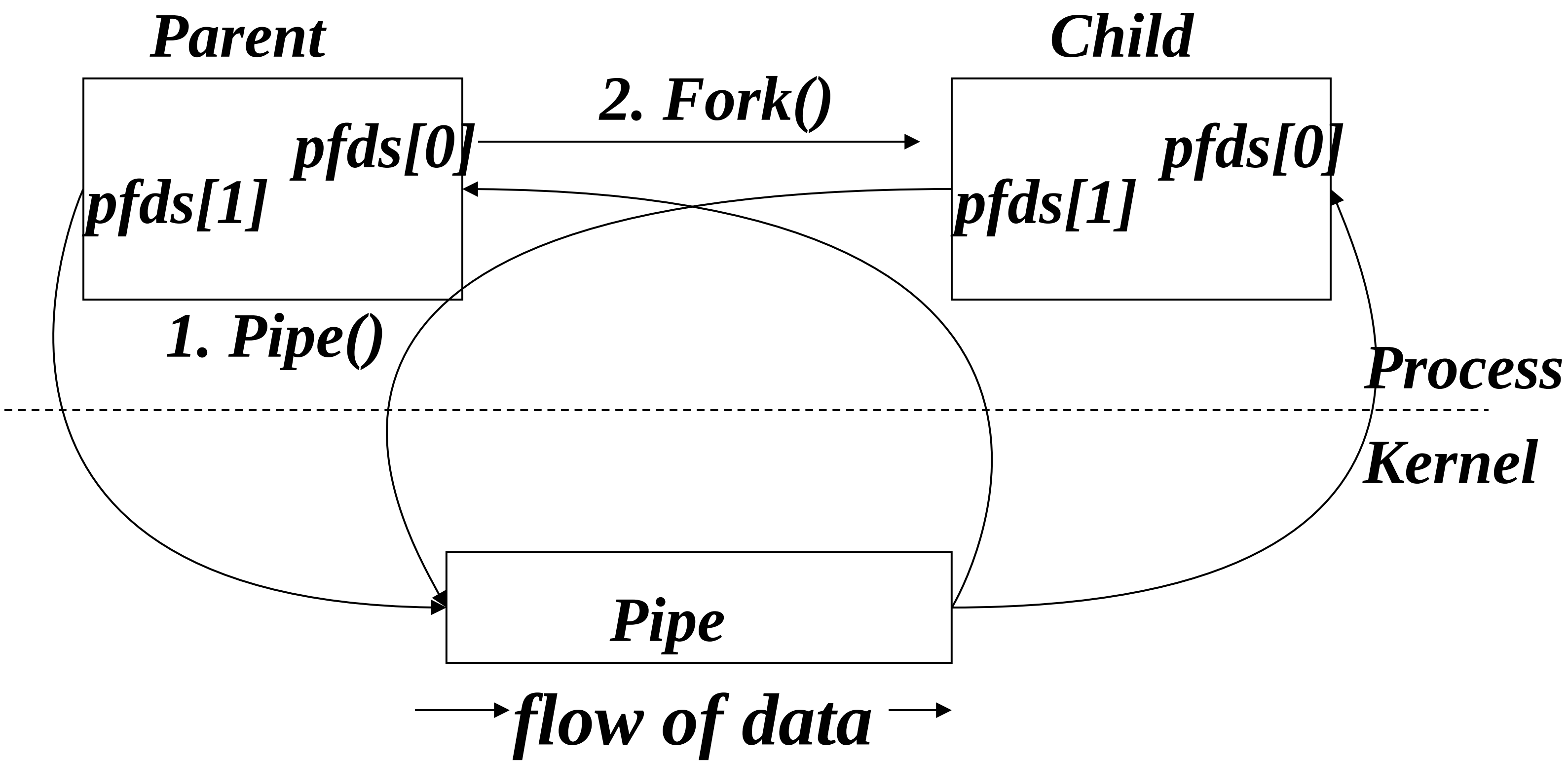
A Channel between two processes

- The **file descriptors** refer to the same I/O entity, in this case a pipe.

- The pipe is inherited by the child

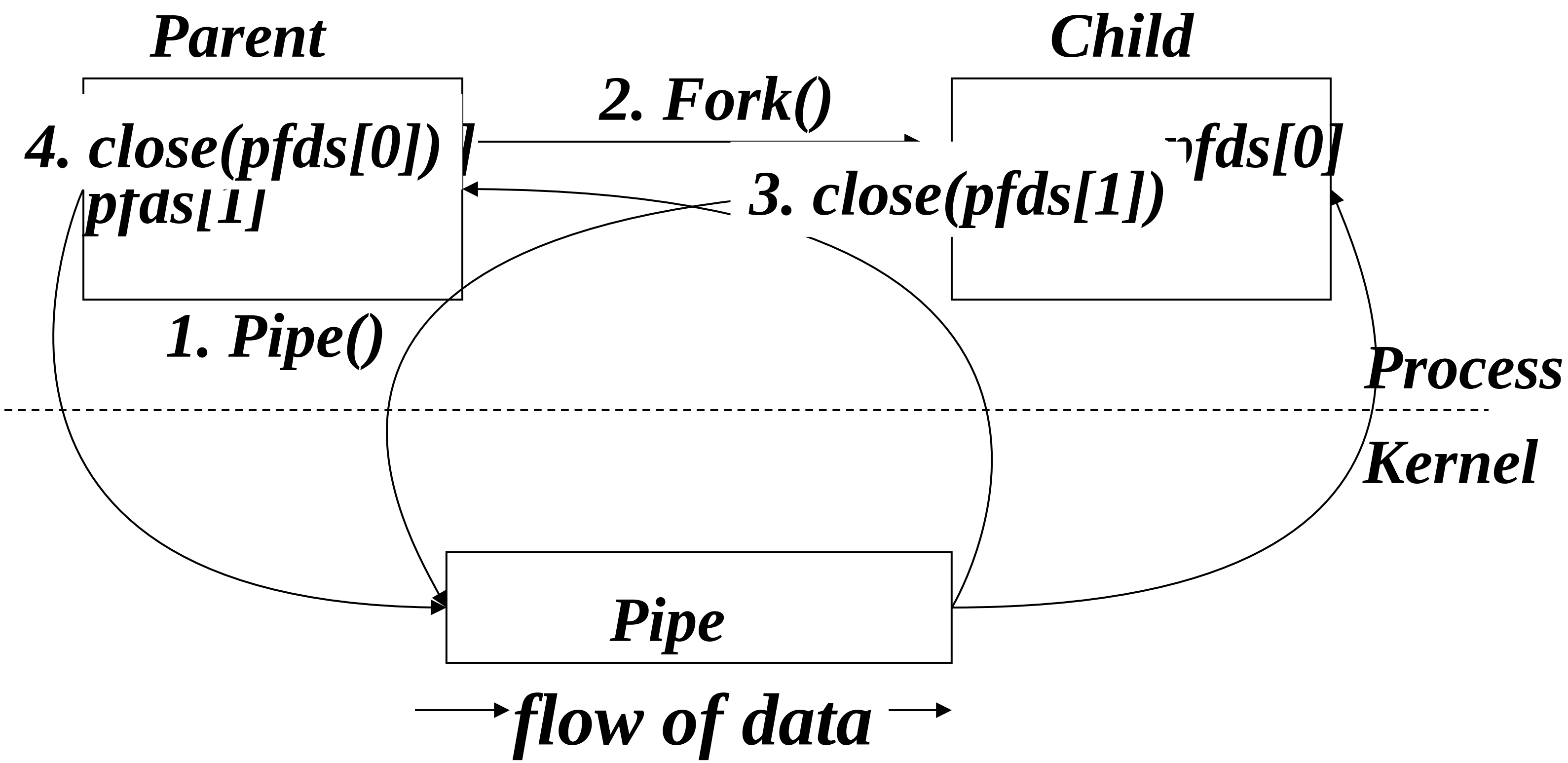
- And may be passed on to the grand-children by the child process or other children by the parent.

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

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);
           }
 }
```