Lecture 04
Inter-Thread and Inter-Process Communication

Dr. Ehtesham Zahoor

Operating systems research

- How to choose a good research paper?

Some review

- Why use threads instead of processes?
- Is there any relation between threads and Docker containers?
- How to create threads?

The Pthreads API

- The most important of thread APIs, in the Unix world, is the one developed by the group known as POSIX.
- POSIX is a standard API supported
- Portable across most UNIX platforms.
- PTHREAD library contains implementation of POSIX standard
- To link this library to your program use ***–lpthread***
  - `gcc MyThreads.c -o MyThreadExecutable - lpthread`

Thread Creation

```
pthread_create( pthread_t *threadid
  ,const pthread_attr_t *attr, void
  *(*start_routine)(void *),void *arg);
```

- This routine creates a new thread and makes it executable.
- Thread stack is allocated and thread control block is created
- Once created, a thread may create other threads.
- Note that an "initial thread" exists by default and is the thread which runs main.
- Returns zero, if ok
- Returns Non-zero if error

Thread Creation

```
pthread_create( pthread_t *threadid
    ,const pthread_attr_t *attr, void
    *(*start_routine)(void *),void *arg);
```

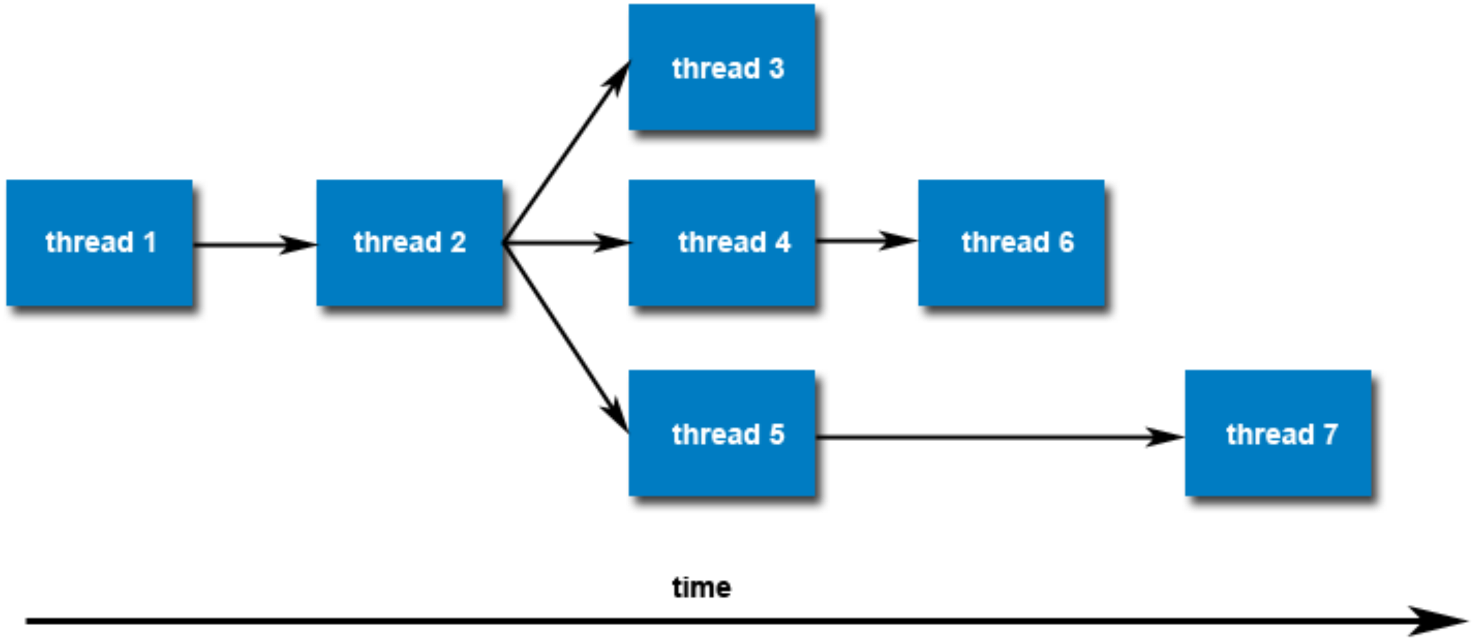
- **threadid**
  - The routine returns the new thread ID via the threadid
  - The caller can use this thread ID to perform various operations
  - This ID should be checked to ensure that the thread was successfully created.
- **attr**
  - used to set thread attributes.
  - NULL for the default values.
- **start_routine**
  - The C routine that the thread will execute once it is created.
- **arg**
  - Arguments are passed to *start_routine* via *arg*.
  - Arguments must be passed by reference as pointers
  - These pointers must be cast as pointers of type void.

Thread Creation

```
#include <pthread.h>
#include <iostream>
using namespace std;


void* PrintHello(void* arg)
{
    cout << "Hello World! " << endl;
    pthread_exit(NULL);
}
int main()
{
   pthread_t threadID;
   pthread_create(&threadID, NULL, PrintHello, NULL);
   cout << "Hello World! " << endl;

   pthread_exit(NULL);
}
```

Thread Creation

- Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.



Passing Arguments

```
#include <pthread.h>
#include <iostream>
using namespace std;


void* PrintHello(void* arg) {
      cout << *(string*) arg  << endl;
    pthread_exit(NULL);
}
int main()
{
   pthread_t threadID;
   string threadArg = "Hello";
   pthread_create(&threadID, NULL, PrintHello, (void*)&threadArg);
   pthread_exit(NULL);
}
```

Passing Arguments To Threads

- The pthread_create() routine permits the programmer to pass one argument to the thread start routine.
- What if we want to pass multiple arguments.
- Create a structure which contains all of the arguments
- Pass a pointer to the structure in the pthread_create() routine.
- Argument must be passed by reference and cast to (void *).

Passing Arguments – The wrong way

```
#include <pthread.h>
#include <iostream>
using namespace std;

void* PrintHello(void* arg)
{
    for (int counter=0; counter<5;counter++) {
      cout << *(string*) arg;
    }
    pthread_exit(NULL);
}
int main()
{
   pthread_t threadID;
   string threadData = "Hello";
   pthread_create(&threadID, NULL, PrintHello, (void*)&threadData);
   threadData = "World";
   pthread_create(&threadID, NULL, PrintHello, (void*)&threadData);

   pthread_exit(NULL);
}
```

The problem?

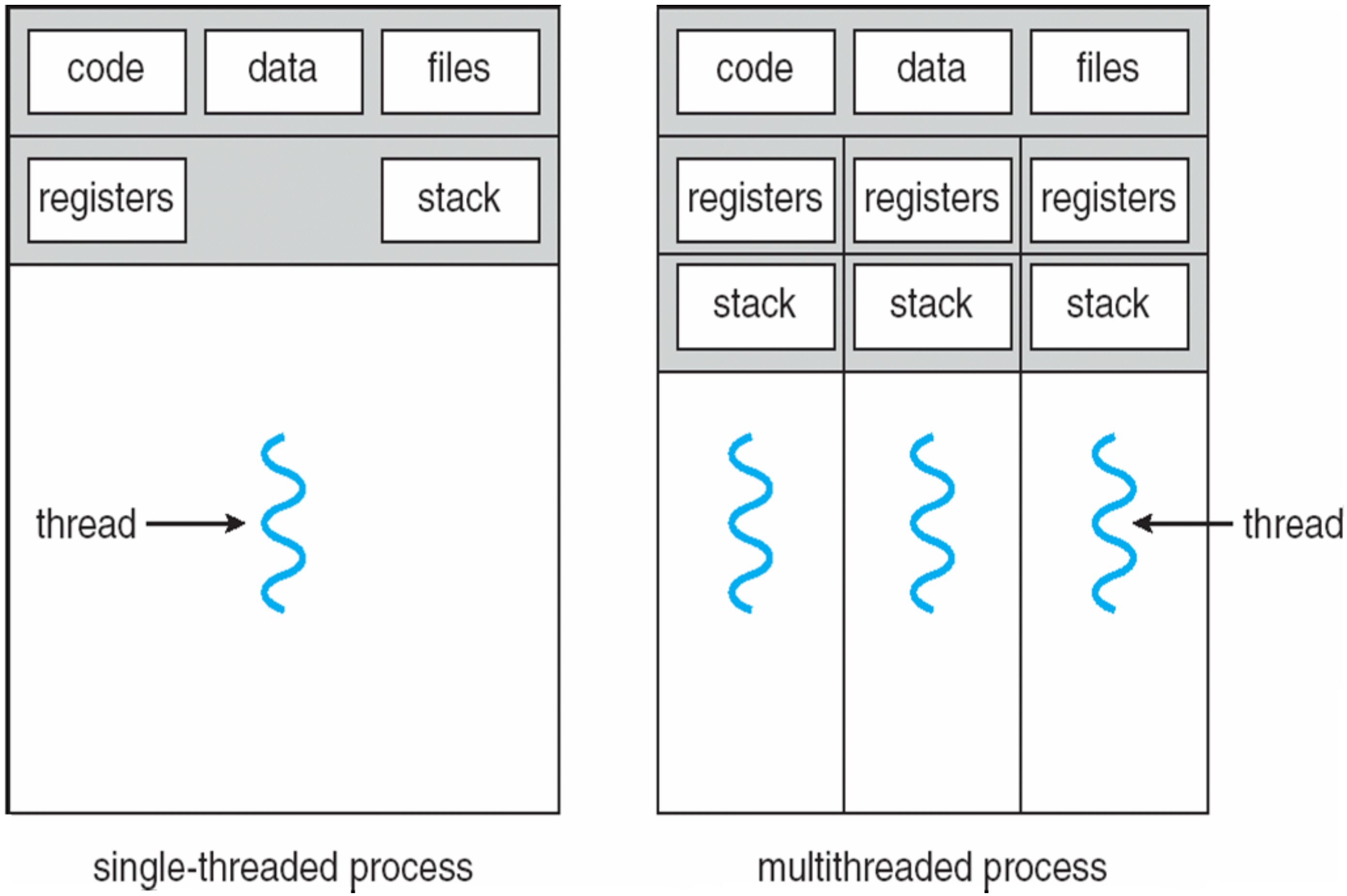
- Threads initially access their data structures in the parent thread's memory space.
- That data structure must not be corrupted/modified until the thread has finished accessing it.

Passing Arguments – better approach

```
#include <pthread.h>
#include <iostream>
using namespace std;
#define NUM_THREADS 3

void* PrintHello(void* arg)
{
    for (int counter=0; counter<2;counter++) {
      cout << *(string*) arg <<endl; }
    pthread_exit(NULL);
}
int main()
{
   pthread_t threadID[NUM_THREADS];
   string threadData[NUM_THREADS] = {"Hello","ez","World"};

   for (int counter=0; counter<NUM_THREADS;counter++) {
      pthread_create(&threadID[counter], NULL, PrintHello,
(void*)&threadData[counter]);
   }
   pthread_exit(NULL);
}
```

Inter-Thread Communication

Inter-Thread Communication



Threads share Global variales!

```
#include <pthread.h>
#include <iostream>
#include <unistd.h>
using namespace std;
#define NUM_THREADS 10

int sharedData = 0;
void* incrementData(void* arg) {
    sharedData++;
    pthread_exit(NULL);
}
int main()
{
   pthread_t threadID;
   for (int counter=0; counter<NUM_THREADS;counter++) {
      pthread_create(&threadID, NULL, incrementData, NULL);
   }
   cout << "ThreadCount:" << sharedData <<endl;
   pthread_exit(NULL);
}
```

What should be the output?

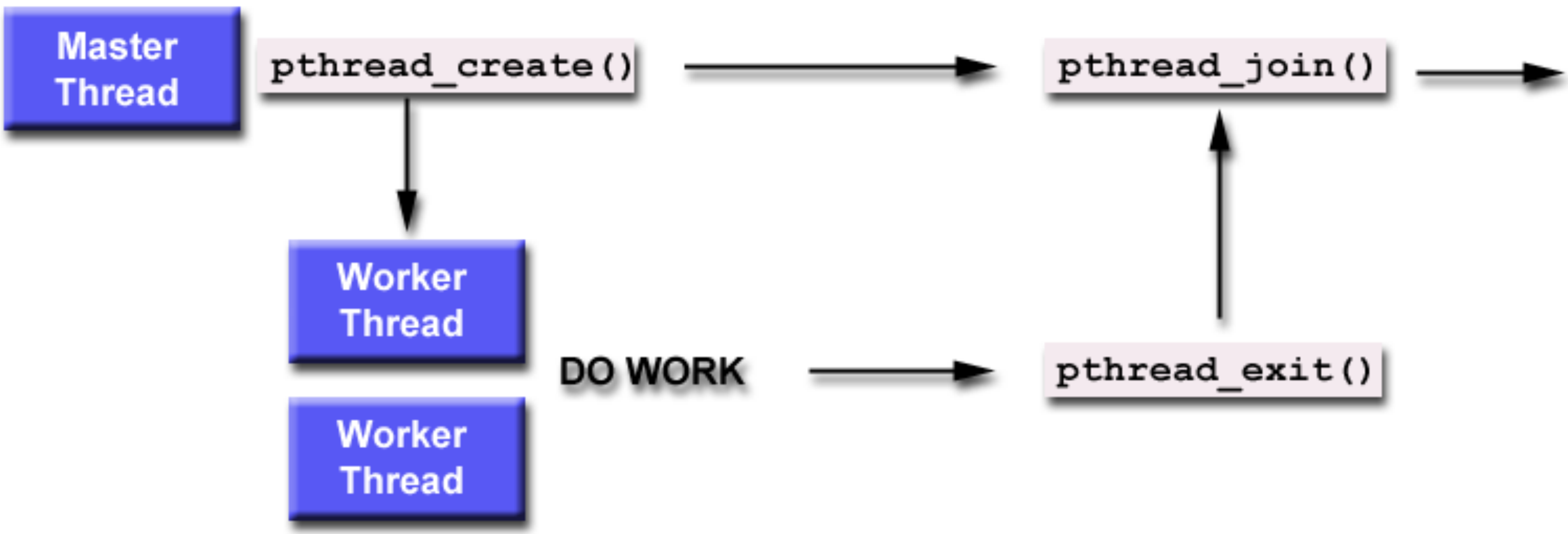
```
>./5globalData
ThreadCount:10

>./5globalData
ThreadCount:8
```

Thread Suspension and Termination

- Similar to UNIX processes, threads have the equivalent of the wait() and exit() system calls
  - pthread_join() Used to block threads
  - pthread_exit() Used to terminate threads

- To instruct a thread to block and wait for a thread to complete, use the **pthread_join()** function.
- Any thread can call join on (and hence wait for) any other thread.

Thread Suspension and Termination



Joining thread

* **Joinable:** on thread termination the thread ID and exit status are saved by the OS.
* Joining a thread means waiting for a thread
* **pthread_join(threadid, status)**
  * "Joining" is one way to accomplish synchronization between threads.
  * subroutine blocks the calling thread until the specified *threadid* thread terminates.
    * The programmer is able to obtain the target thread's termination return status (if specified) in the *status* parameter.
* It is impossible to join a detached thread

ThreadCount: A better implementation

```
#include <pthread.h>
#include <iostream>
#include <unistd.h>
using namespace std;
#define NUM_THREADS 100
int sharedData = 0;
void* incrementData(void* arg)
{
    sharedData++;
    pthread_exit(NULL); }
int main()
{
   pthread_t threadID[NUM_THREADS];
   for (int counter=0; counter<NUM_THREADS;counter++) {
      pthread_create(&threadID[counter], NULL, incrementData, NULL);
   }
   //waiting for all threads
   int statusReturned;
   for (int counter=0; counter<NUM_THREADS;counter++) {
      pthread_join(threadID[counter], NULL);
    }
   cout << "ThreadCount:" << sharedData <<endl;
   pthread_exit(NULL);
}
```

Is the problem solved?

- Unfortunately, not yet :(
- The output from running it with 1000 threads is as below:

```
>./6join
ThreadCount:990
>./6join
ThreadCount:978
>./6join
ThreadCount:1000
>
```

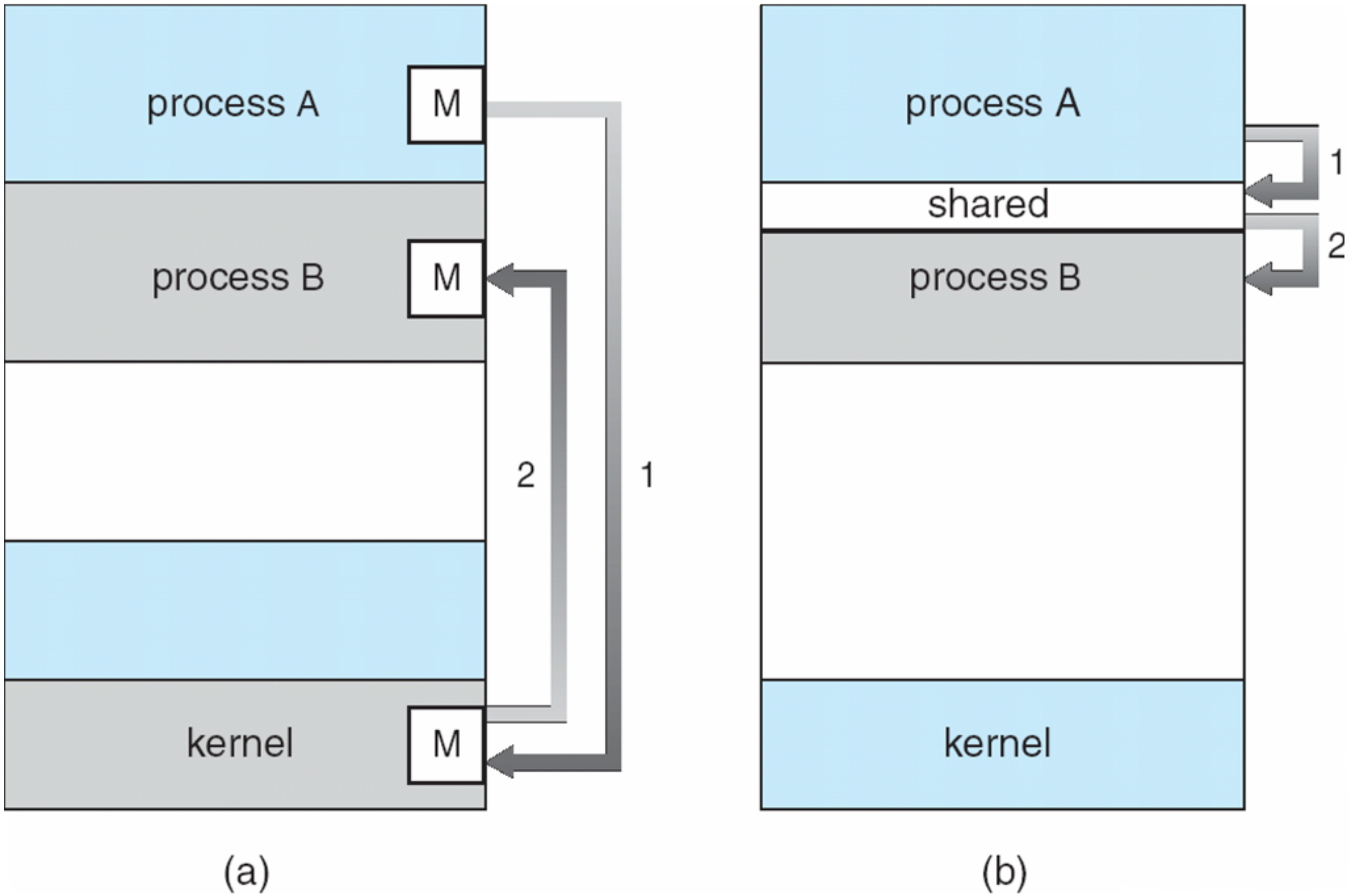
- Reasons? What can be done?
- Lets postpone this discussion till synchronization class.

Inter-Process Communication

Interprocess Communication

- A process has access to the memory which constitutes its own address space.
- So far, we have discussed communication mechanisms only during process creation/termination
  - The parent receives the exit status of the child
- Processes may need to communicate during their life time.

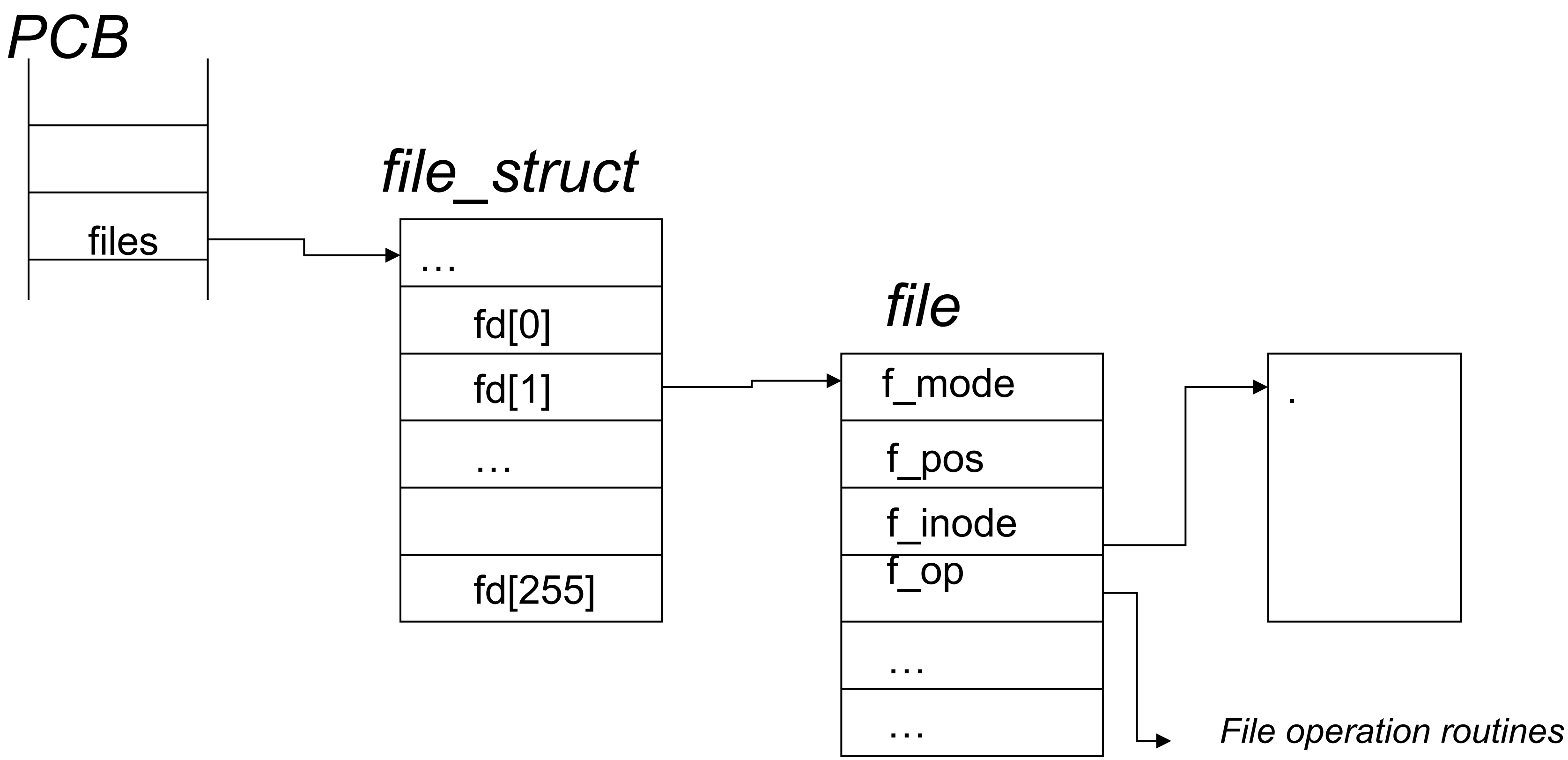
Communications Models



UNIX IPC - Pipes

File Descriptors

- The PCB of each process contains a pointer to a *file_struct, a kernel-resident array data structure containing the details of open files*



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.

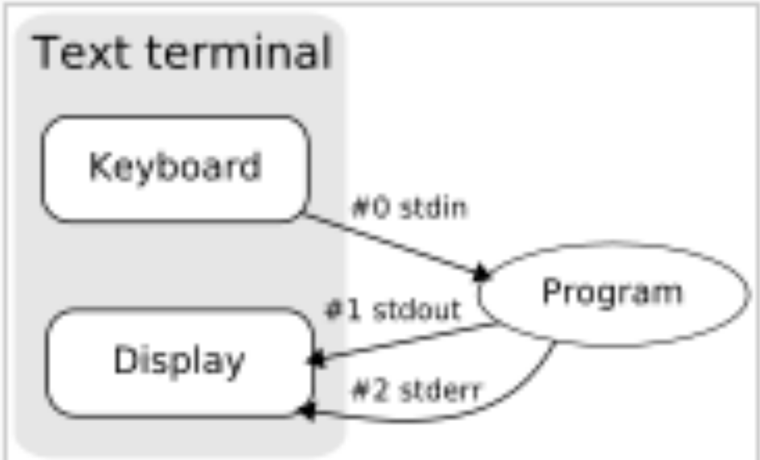
- Linux processes expect three file descriptors to be open when they start.

- 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.

Lecture 04: Inter-Thread and Inter-Process Communication



The file descriptors for input, output, and error

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

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

using namespace std;

int main()
{
char myBuffer[10];
cout << "Please enter your name: \n";
int count = read(0,myBuffer,10);
write(1,myBuffer,count);


}
```

The Unix fact

- When Unix programs do any sort of I/O, they do it by reading or writing to a file descriptor
- A file descriptor is simply an integer associated with an open file

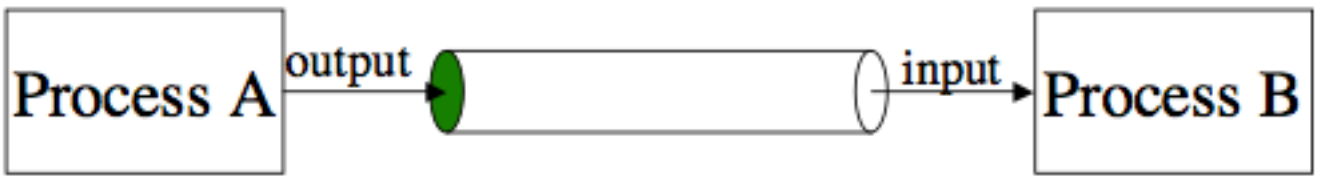
The Unix fact

- A file in Unix can be
  - A network connection
  - A pipe
  - A terminal
  - A real on-the-disk file
  - Or just about anything else

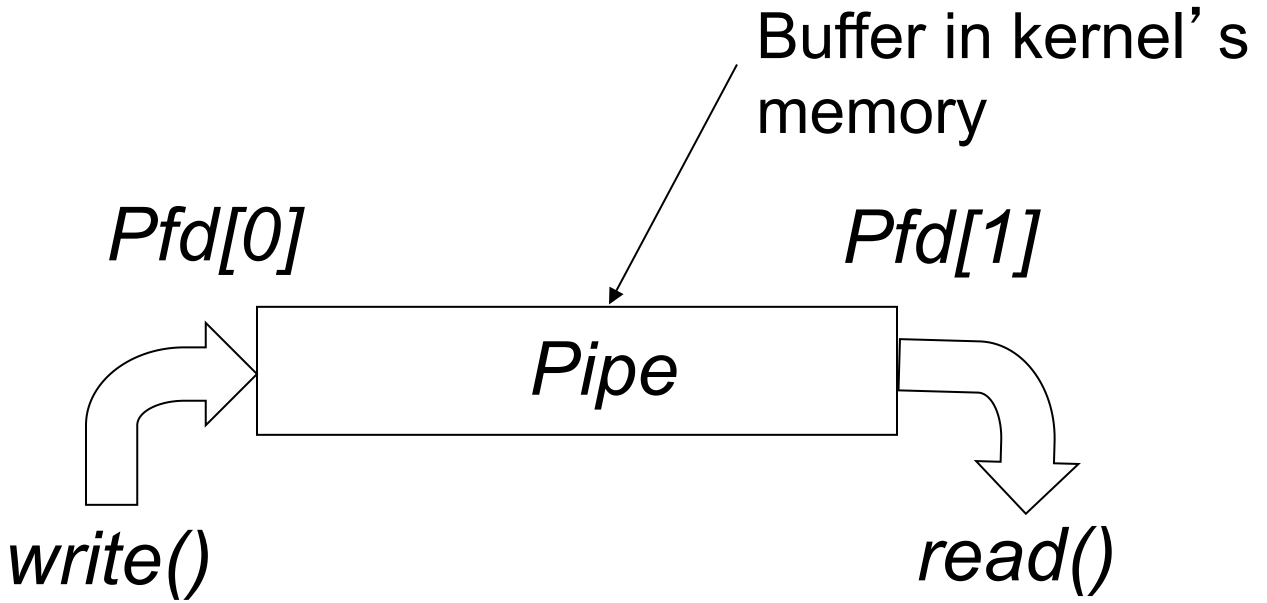
EVERYTHING IN UNIX IS A FILE!

Pipes

- Provides an interprocess communication channel



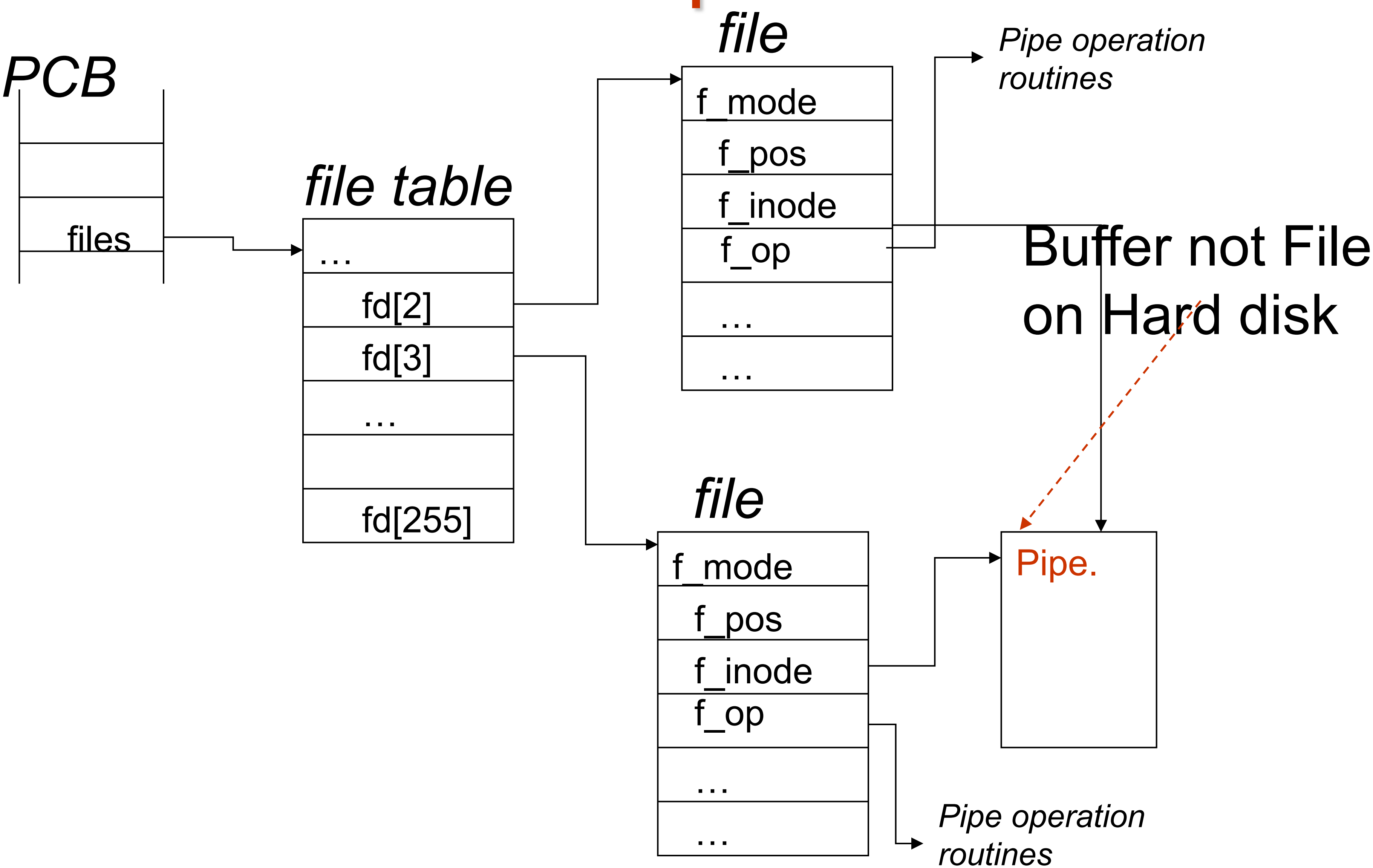
Pipes: Shared info in kernel's memory



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

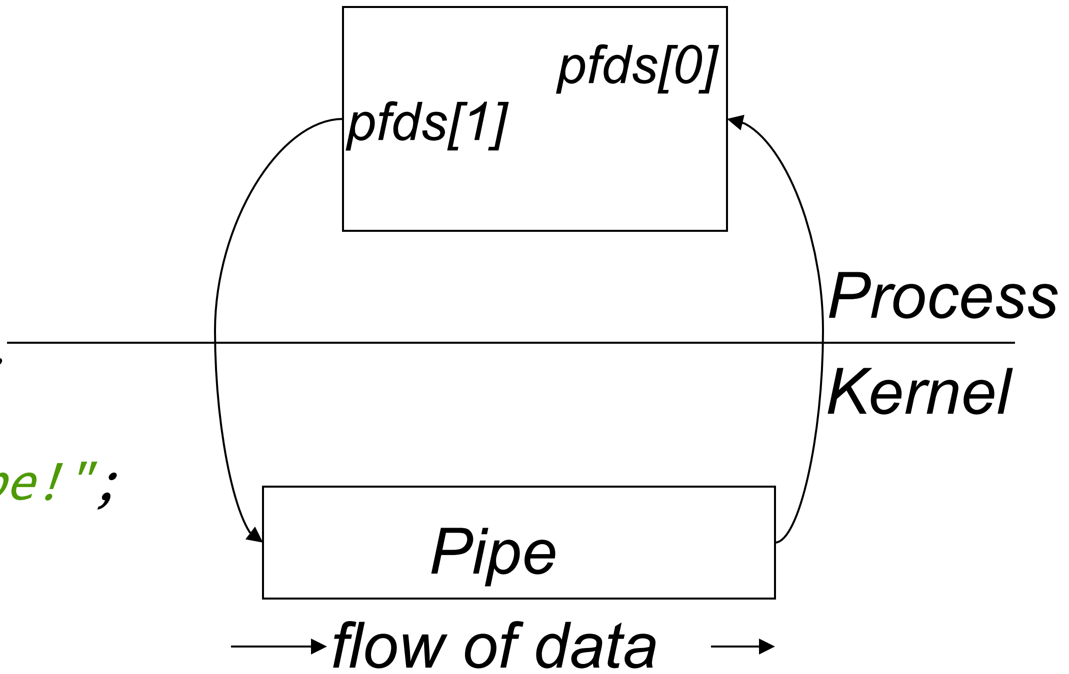

PCB
files
file table
…
fd[2]
fd[3]
…
fd[255]
file
f_mode
f_pos
f_inode
f_op
…
…
Pipe operation routines
Buffer not File on Hard disk
file
f_mode
f_pos
f_inode
f_op
…
…
Pipe.
Pipe operation routines

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

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

using namespace std;

int main()
{
int pfds[2];
if (pipe(pfds) != -1){
cout << pfds[0] << pfds[1];
} else {
cout << "Error creating pipe!";
exit(1);
}


return 0;
}
```



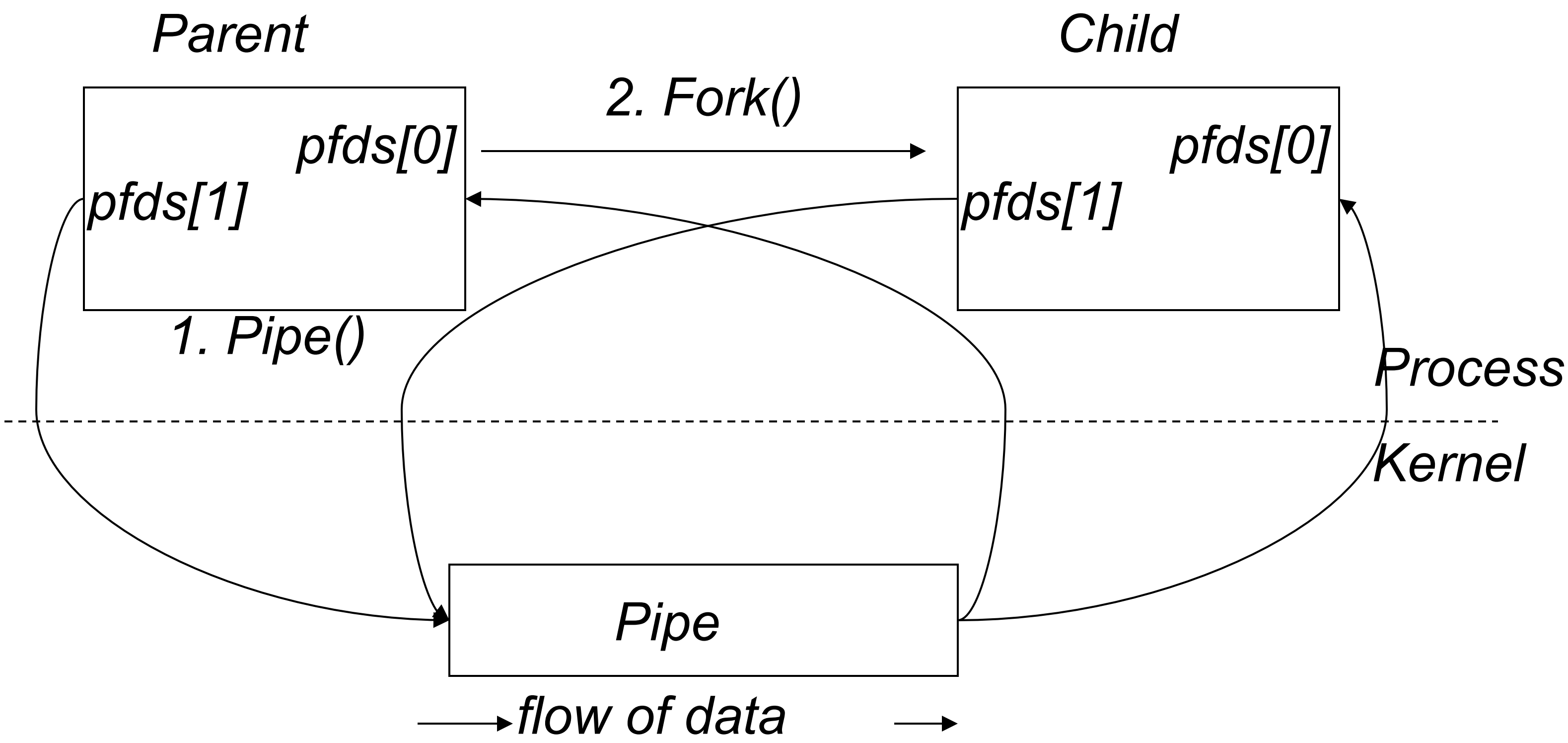
A Channel between two processes

- 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

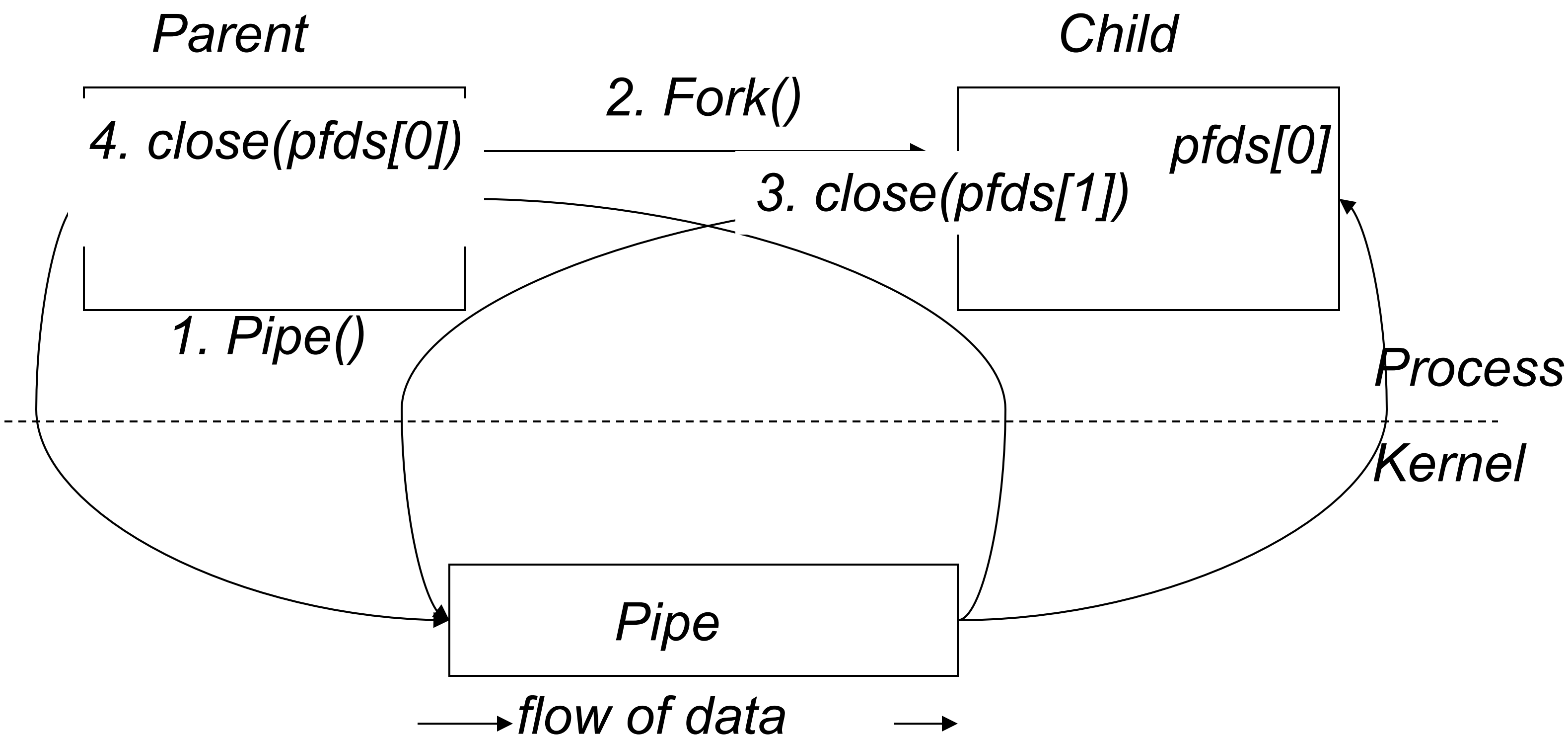
- The **file descriptors** refer to the same I/O entity, in this case a pipe.
- The pipe is inherited by the child
- And may be passed on to the grand-children by the child process or other children by the parent.

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

```
int main()
{
int pfds[2];
char buf[30];
if (pipe(pfds) != -1){
if (!fork()) {
close(pfds[0]);
    cout << "CHILD: writing to the pipe\n";
    write(pfds[1], "Sample Text", 11);
cout << "CHILD: exiting\n";
    exit(0);
  } else {
close(pfds[1]);
     cout << "PARENT: reading from pipe\n";
  read(pfds[0], buf, 11);
  cout << "PARENT has read: " << buf;
  wait(NULL);
}
 } else {
cout << "Error creating pipe!";
exit(1);
}
return 0; }
```