Operating Systems: IPC & Synchronization

Fabrice BOISSIER <fabrice.boissier@epita.fr>

2021-11-09

How to manipulate data in IT

Two visions of the world:

- You know in advance the length of your data
  - Record length, block size, ...
  - Specific OS and languages *(z/OS and JCL, COBOL, ...)*
  - Specific functions *(str**n**len, str**n**cpy, str**n**cmp, ...)*
  - Pretty easy to manage (and somewhat safe)

- You know the separator of each field/record/file/...
  - « , », « ; », « \n », « \0 », « EOF », ... *(CSV format, awk(1), ...)*
  - Hard to manage: your CPU manipulates N bits, buffers of fixed length, ...
  - Way more unsafe...
  - Foundations of modern computing

IPC & Synchronization

- Why divising the work between multiple processes?

- Functional reasons:
  - Multiple organizations are responsible of a precise service....
    ...and a product is using the services of those organizations

- Technical reasons:
  - Multiple machines with one main service on each one
    (We must link them and make them communicate/transfer data)
  - One machine composed of multiple processors or cores of execution

IPC & Synchronization

- How to make processes communicate and share or transfer data from one to another?
  - InterProcess Communications

- What are the technical challenges?
  - Who will read which data at what time?
  - How to wait for the other process to finish reading/writing?
  - How to be sure that the data are written immediately?
  - ...
  - *Synchronization problems*

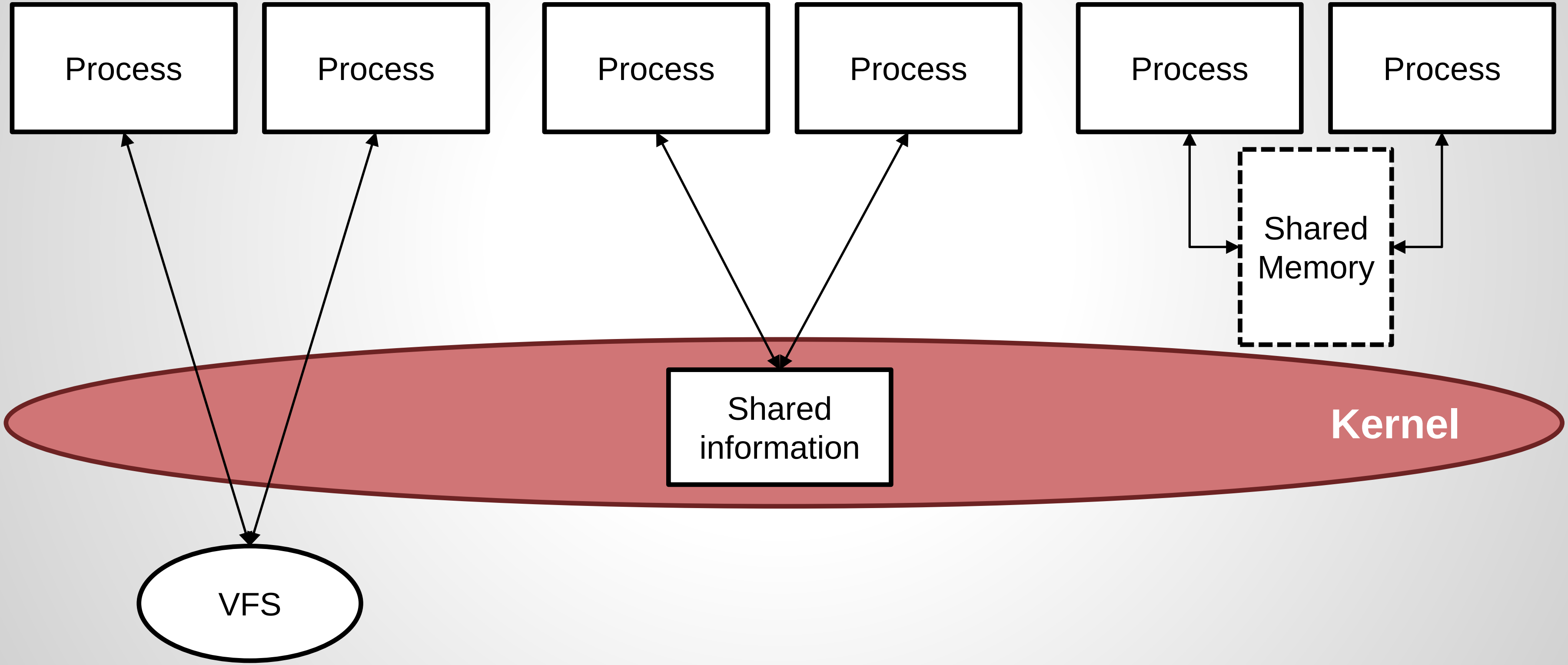
IPC & Synchronization

UNIX Network Programming, Volume 1: The Sockets Networking API - *W. Richard Stevens*

UNIX Network Programming, Volume 2: Interprocess Communications - *W. Richard Stevens*

InterProcess Communication

InterProcess Communication



InterProcess Communication

- File
  - Processes read and write within a file
  - Requires to pass through the VFS and in the kernel (if monolithic)
- Shared informations
  - Pipe, Message queue, Semaphore
  - Processes ask the kernel to share data and manage the accesses to it
- Shared memory
  - Shared memory
  - Memory pages are shared among processes
  - Kernel is invoked « only » when creating/destroying the pages

IPC: Problems & Issues

- Problems
  - Reading and writing data concurrently might corrupt data
    (synchronization in read/write is required)
  - Passing through the kernel is expensive
    (Reduce the number of syscalls)

- Issues
  - How to establish a link (rendez-vous)
  - How many processes per link (1 to 1, 1 to N)
  - How many links per process

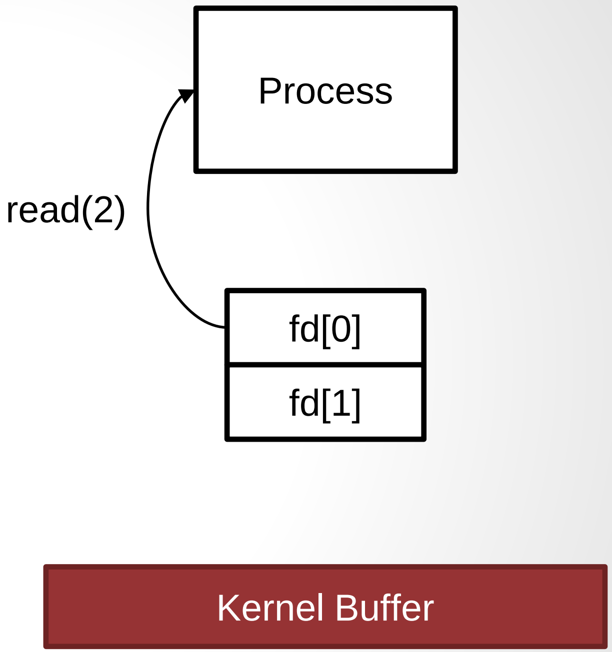
IPC: Pipes (anonymous pipes)

- A buffer is created within the kernel
  - Works exactly like a FIFO
  - Bytes are read in the some order they were written
- Access with read/write through a file descriptor
  - A VFS « file » is created and points to the kernel buffer
  - « Everything is a file »
- Each read « consumes » data
  - Data is not present anymore within the pipe after a read
- *Used with a fork(2)*

IPC: Pipes (anonymous pipes)

int pipe(int fd[2])

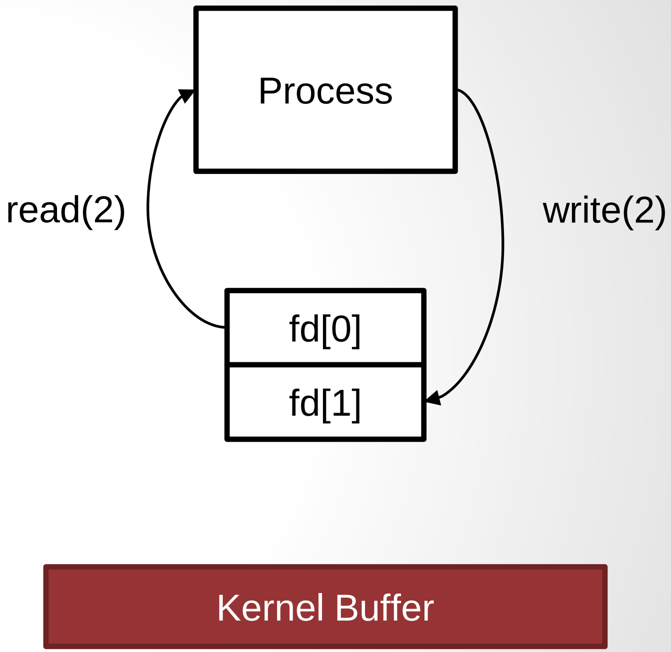
- Read on fd[0]



IPC: Pipes (anonymous pipes)

int pipe(int fd[2])

- Read on fd[0]
- Write on fd[1]

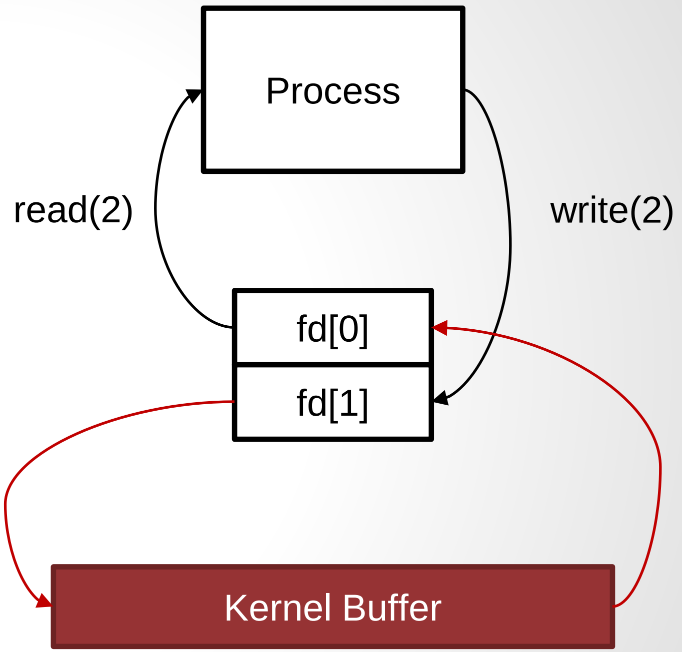


IPC: Pipes (anonymous pipes)

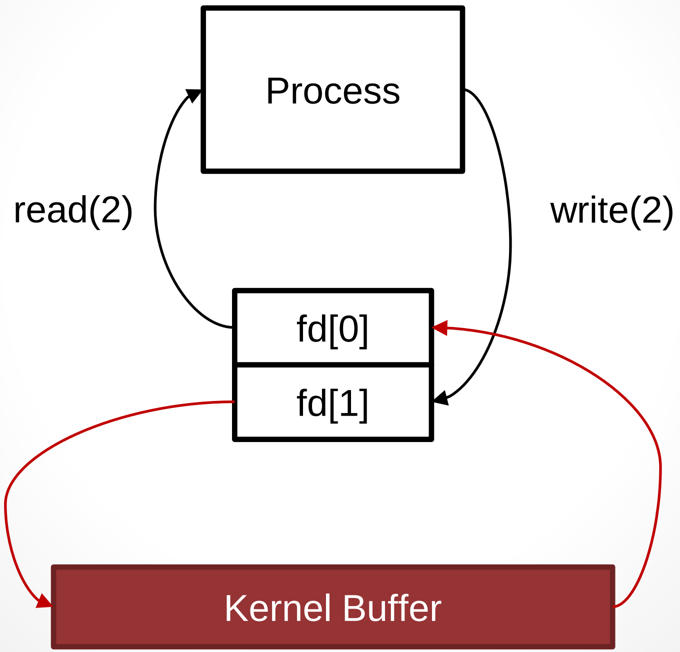
int pipe(int fd[2])

- Read on fd[0]
- Write on fd[1]

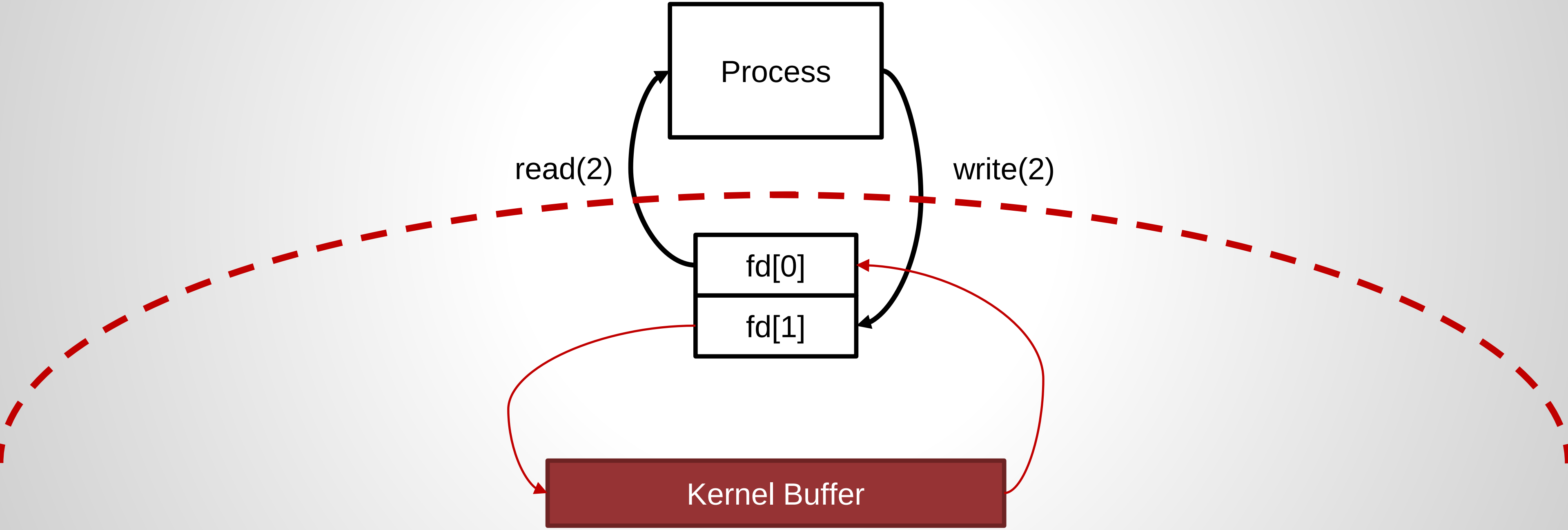
- The kernel manages the transfer of data



IPC: Pipes (anonymous pipes)

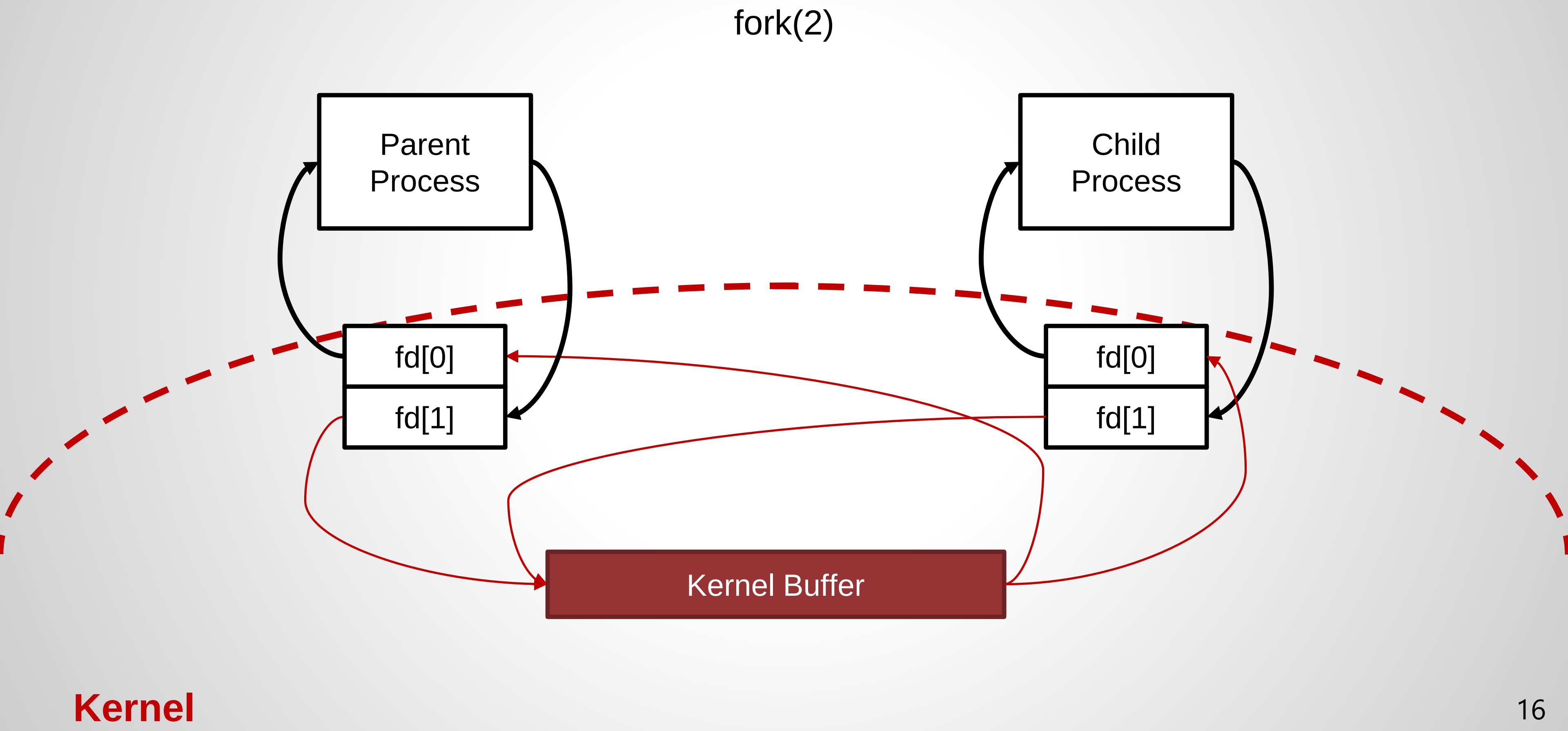


IPC: Pipes (anonymous pipes)

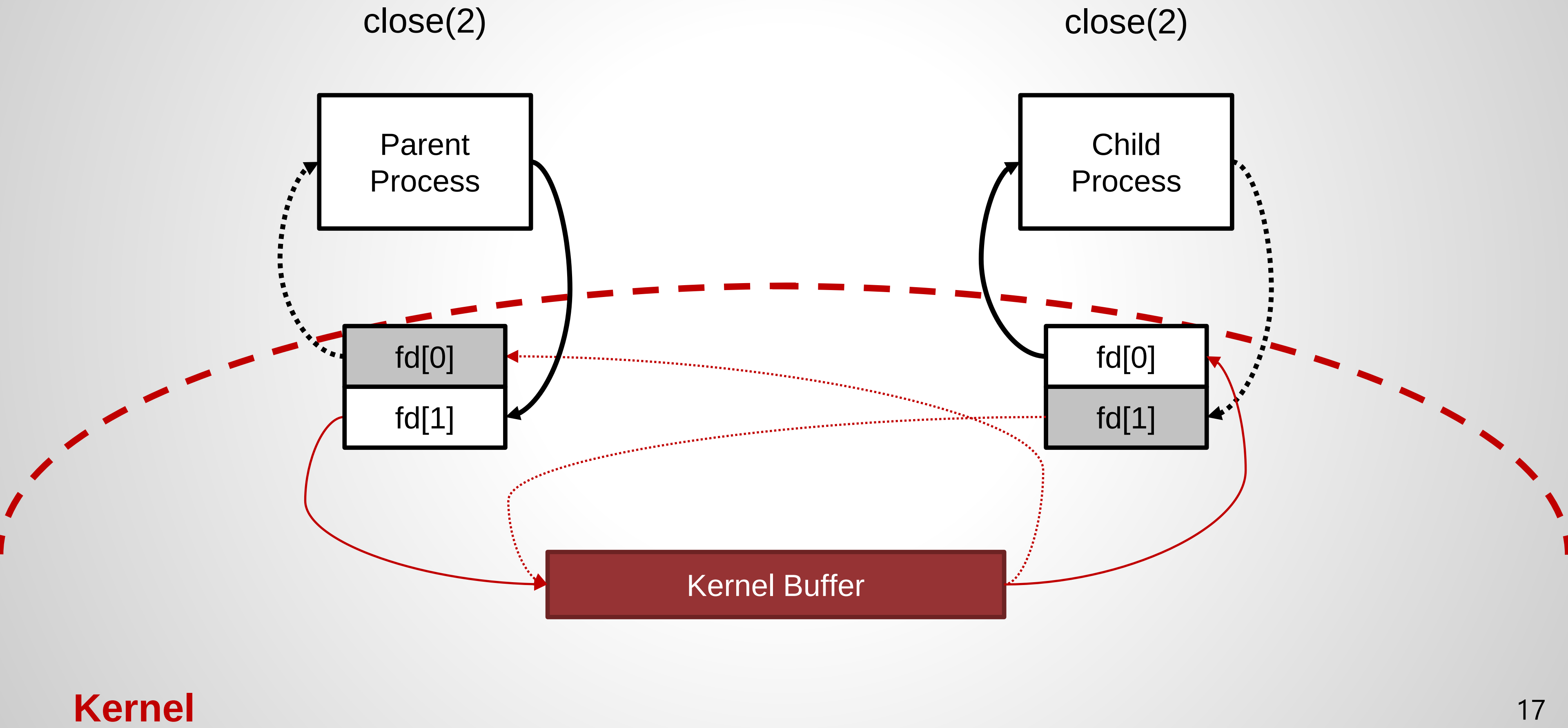


Kernel

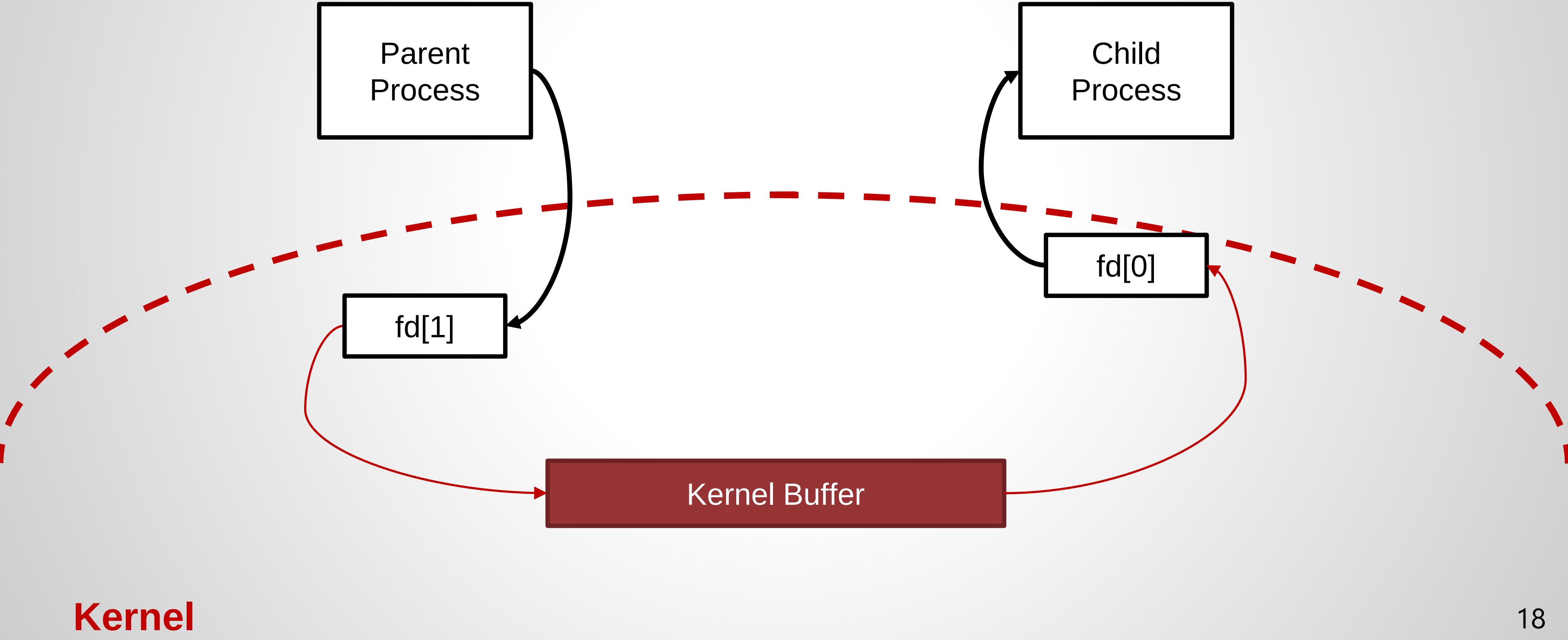
IPC: Pipes (anonymous pipes)



IPC: Pipes (anonymous pipes)



IPC: Pipes (anonymous pipes)



IPC: Pipes (anonymous pipes)

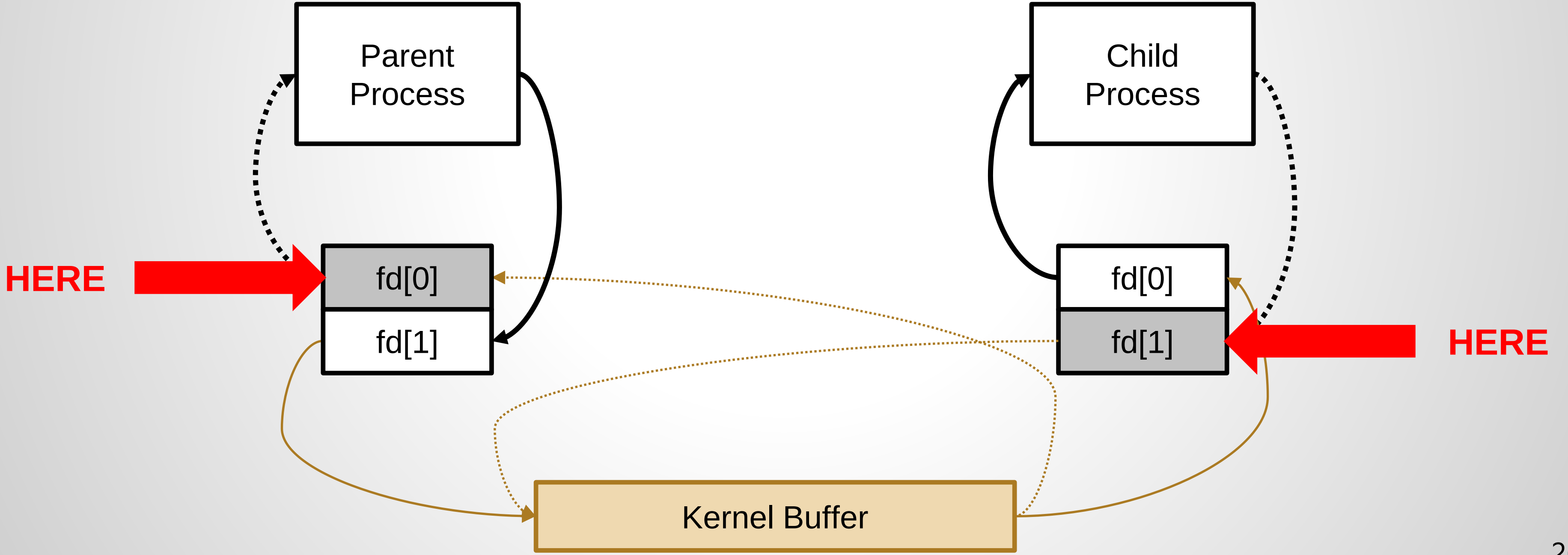
- Kernel manages synchronization
  - *Except if the read/write is asynchronous…*

- When there is no data available, the process that read is put in a waiting state
  - The scheduler is waiting for the ressource to be available before putting back the process within the *ready queue*

- When the buffer is full, the process that writes is put in a waiting state

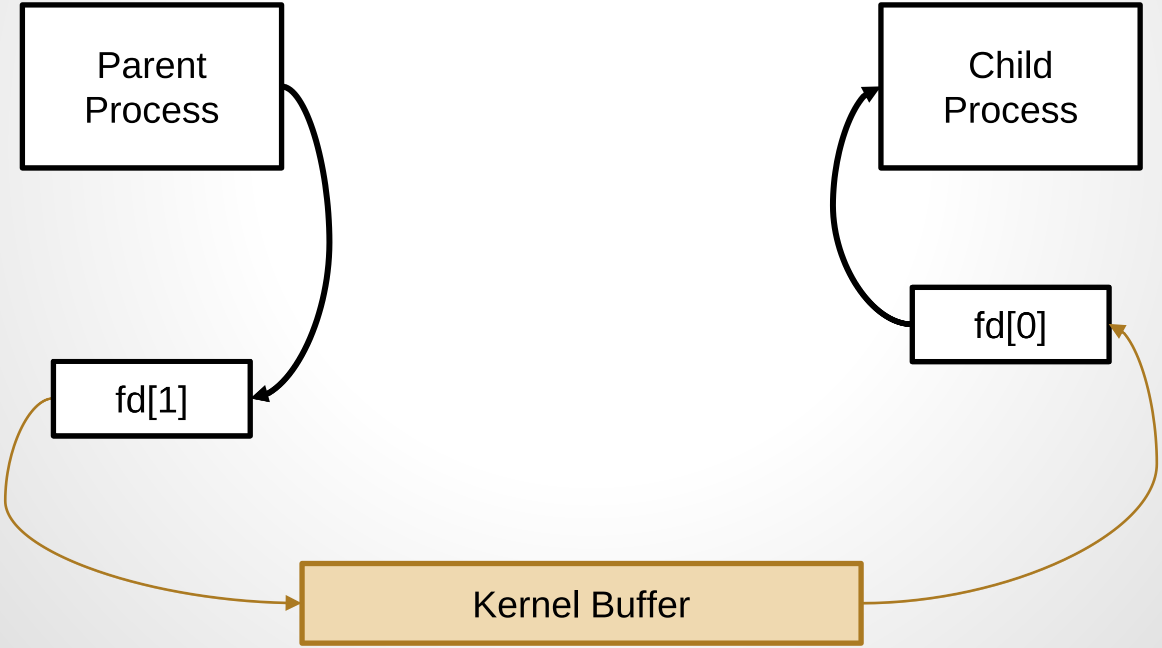
IPC: Pipes (anonymous pipes)

You should close the useless file descriptors BEFORE writing or reading!



IPC: Pipes (anonymous pipes)

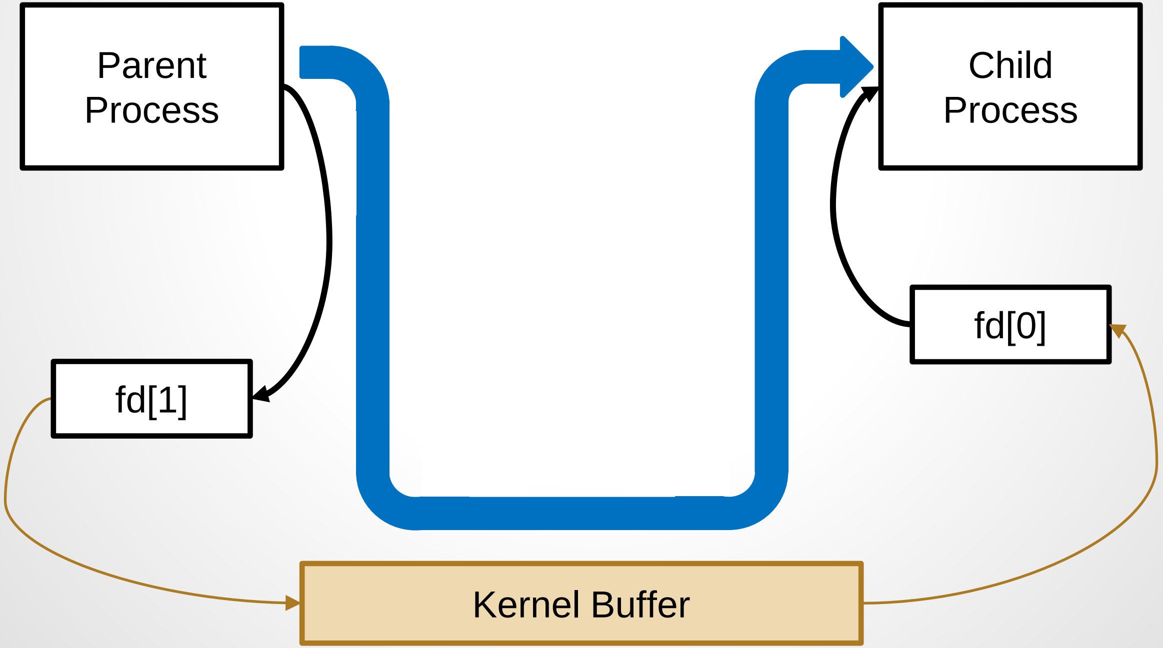
In the pipe context, you should keep a very clean state of your file descriptors



IPC: Pipes (anonymous pipes)

Anonymous pipes are traditionally « unidirectional »

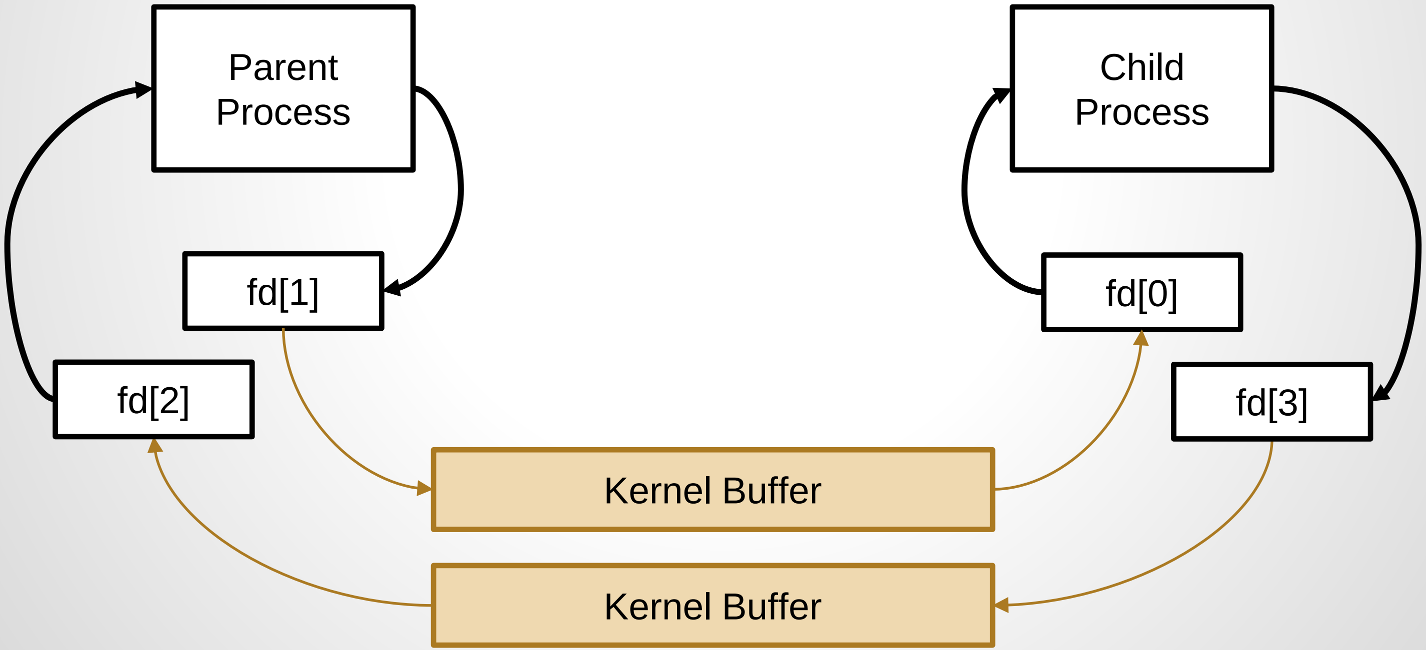
(read very carefully your manual)



IPC: Pipes (anonymous pipes)

For bidirectionnal, you must create 2 pipes

(just create a table for 4 int to put the fd)



IPC: Pipes (anonymous pipes)

- Kernel buffer size:
  - Currently 16 memory pages (64KiB or 65,536 Bytes)…
    …it might be 1 Memory Page (4KiB / 4,096B) in some cases…
    …or it can be dynamically chosen
    (check with « ulimit -a » or fcntl(2) with F_GETPIPE_SZ)
  - Pipe buffer size can be updated with *fcntl(2)* and *F_SETPIPE_SZ*
  - Writing more than the buffer size will block the writing process

- Pipes are linked to the VFS
  - Each fd points to a VFS « file » for keeping read/write cursor state
  - « pipefs » on Linux
    (specific management of the buffers)

IPC: Pipes (anonymous pipes)

- Well, now you know how to use anonymous pipes…
  - See pipe(2) for details

- …but how is implemented the « | » in the shell ?
  - See sh(1) for « some » details…
  - …see strace(1) for much more details

[syscalls & strace: let’s find what’s happening]

- strace(1)

- Traces syscalls (even within childs)
    - Might check for specific syscalls

- Useful for understanding « how » things work…
- …and why they do not work in some cases…
- …or even how to break things.

IPC: Pipes in the shell

sh$ > cat file.txt | grep "hello"

```
strace -qf  -e execve,pipe,dup2,read,write  \
  sh -c 'cat file.txt | grep "hello" ‘
```

- Let’s search for these syscalls
  - « execve », « pipe », « dup2 », « read », « write »

- « -f » even within the childs

```
pipe([3, 4])
[... fork()  fork() ...]
[pid 27597] dup2(3, 0)
[pid 27597] execve("/bin/grep", ["grep", "hello"], ...
[pid 27596] dup2(4, 1)
[pid 27597] read(3, "", 4096)
[pid 27596] execve("/bin/cat", ["cat", "file.txt"], ...
[pid 27596] read(3, "", 4096)
[pid 27597] read(0,  <unfinished ...>
[pid 27596] read(3, "hello world!\n", 131072)
[pid 27596] write(1, "hello world!\n", 13)
[pid 27597] <... read resumed> "hello world!\n", 32768)
[pid 27597] write(1, "hello world!\n", 13)
[pid 27597] read(0,  <unfinished ...>
[pid 27596] read(3, "", 131072)
[pid 27597] <... read resumed> "", 32768)
[pid 27597] +++ exited with 0 +++
[pid 27596] +++ exited with 0 +++
```

IPC: Dup2

(https://toroid.org/unix-pipe-implementation)

- dup2(int oldfd, int newfd)
  - *See dup(2) and dup2(2)*

- Duplicates a file descriptor into another one
  - The VFS « file » pointed in the fd table is exactly the same
  - Therefore, they share the exact same state (lseek, flags, ...)

- « oldfd » is duplicated into « newfd »
  - If « newfd » was opened, it is first closed
  - If « oldfd » is not valid, nothing else is done, and dup2() fails
  - If « oldfd » == « newfd », nothing happens

IPC: Dup2

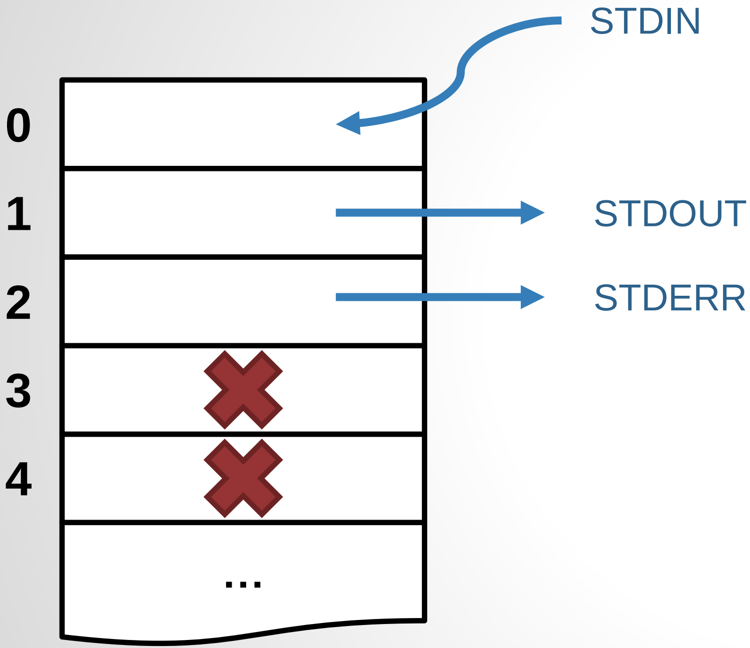
(https://toroid.org/unix-pipe-implementation)

- dup2(int oldfd, int newfd)
  - *See dup(2) and dup2(2)*

- Main usage of dup2(): replace one or more file descriptor entry from STDIN, STDOUT, or STDERR to another VFS file before an exec(2)

- Don’t forget to close the « oldfd » after the duplication
  - If you do an exec(2), it won’t close the « oldfd »…
  - …neither will the next program (except if it is coded to do so)

IPC: Dup2

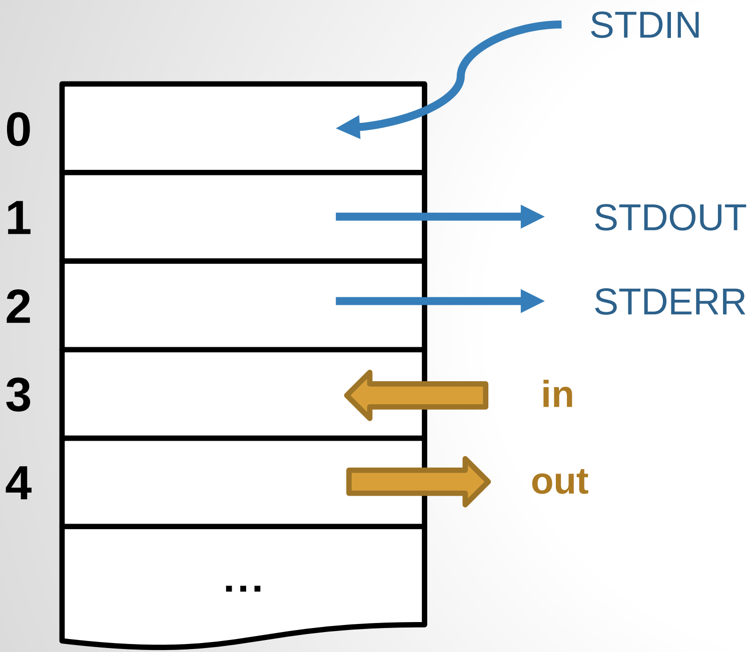


Step 1: *(initial state)*

- The 3 usual file descriptors are opened
  - STDIN
  - STDOUT
  - STDERR

(They are inherited from the parent process)

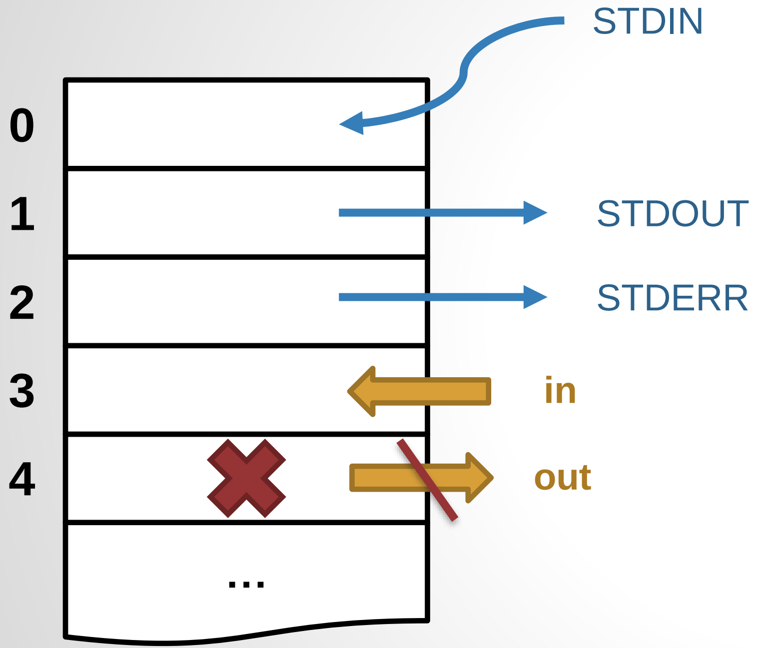
IPC: Dup2



Step 2: pipe(int[2])

- Two file descriptors are reserved in order to access the pipe (the kernel's buffer)

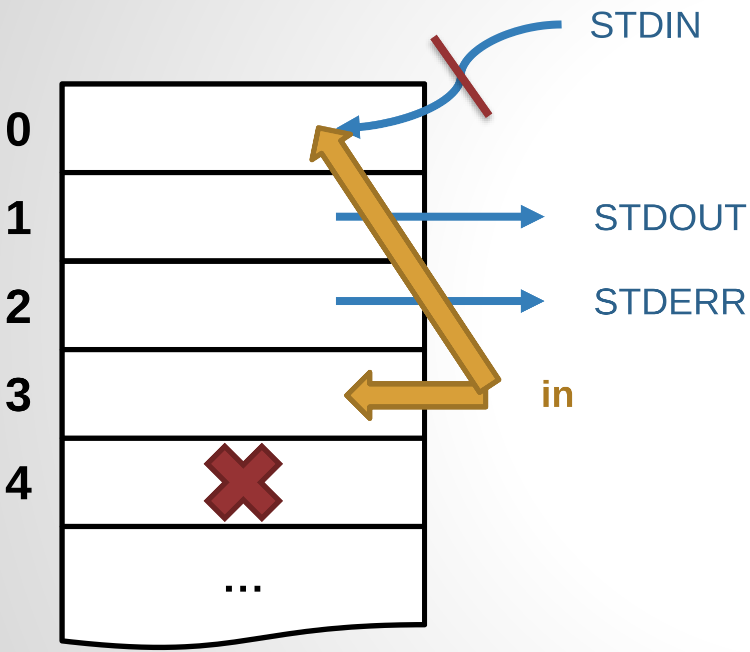
IPC: Dup2



Step 3: close(4)

- One of the file descriptor is closed, in order to keep the pipe unidirectionnal

IPC: Dup2

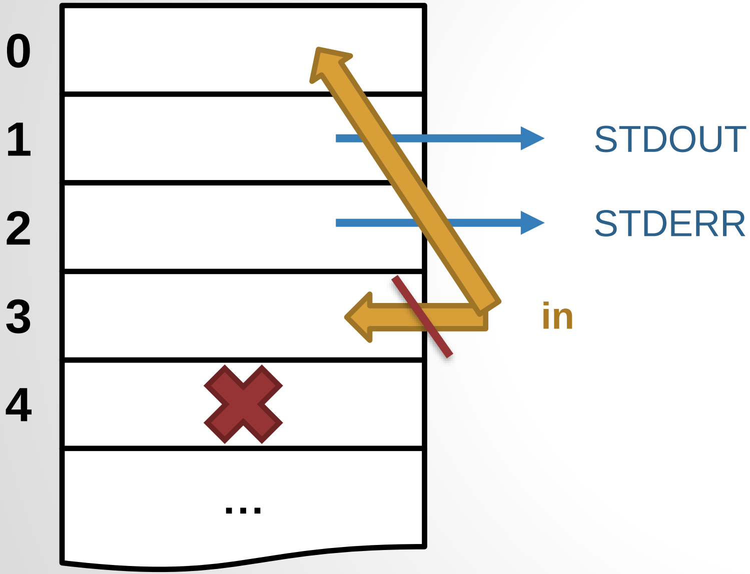


Step 4: dup(3, 0)

- Duplicate the file descriptor number 3 into STDIN

1. fd 0 is first closed
2. fd 3 is redirected to the fd 0

IPC: Dup2

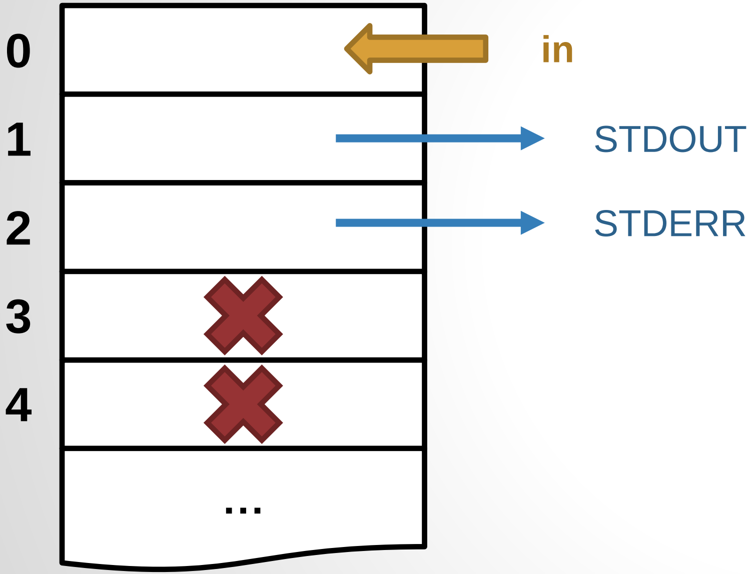


Step 5: close(3)

- Close the old file descriptor (fd 3) because the program that will be loaded with exec() won't close it

  (by inheritance, each process expects only fd 0,1,2 to be opened)

IPC: Dup2



Step 6: *(final state)*

- The next program to be executed will have three opened file descriptors (0 in reading mode, 1 and 2 in writing mode), but the file descriptor 0 is connected to a pipe instead of STDIN

```
pipe([3, 4])
[... fork()  fork() ...]
[pid 27597] dup2(3, 0)
[pid 27597] execve("/bin/grep", ["grep", "hello"], ...
[pid 27596] dup2(4, 1)
[pid 27597] read(3, "", 4096)
[pid 27596] execve("/bin/cat", ["cat", "file.txt"], ...
[pid 27596] read(3, "", 4096)
[pid 27597] read(0,  <unfinished ...>
[pid 27596] read(3, "hello world!\n", 131072)
[pid 27596] write(1, "hello world!\n", 13)
[pid 27597] <... read resumed> "hello world!\n", 32768)
[pid 27597] write(1, "hello world!\n", 13)
[pid 27597] read(0,  <unfinished ...>
[pid 27596] read(3, "", 131072)
[pid 27597] <... read resumed> "", 32768)
[pid 27597] +++ exited with 0 +++
[pid 27596] +++ exited with 0 +++
```

IPC: Pipes (anonymous pipes) and dup2

```
pipe([3, 4])
```

- pipe(2) created file descriptors « 3 » and « 4 »

```
[pid 27597] dup2(3, 0)
```

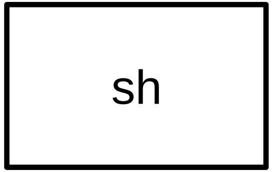
- dup2(2) duplicated fd « 3 » into fd « 0 » (STDIN)

```
[pid 27596] dup2(4, 1)
```

- dup2(2) duplicated fd « 4 » into fd « 1 » (STDOUT)

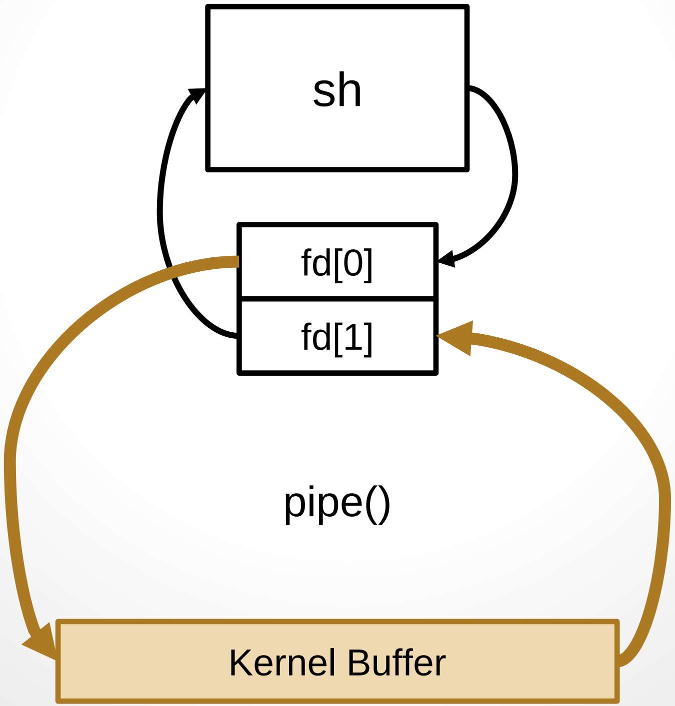
IPC: Pipes in the shell

sh$> cat file.txt | grep "hello"



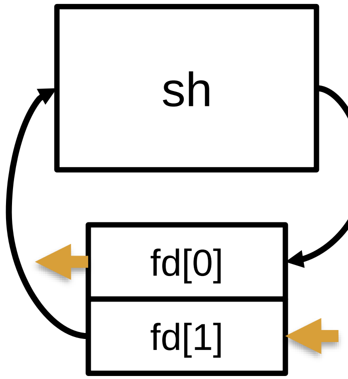
IPC: Pipes in the shell

sh$> cat file.txt | grep "hello"



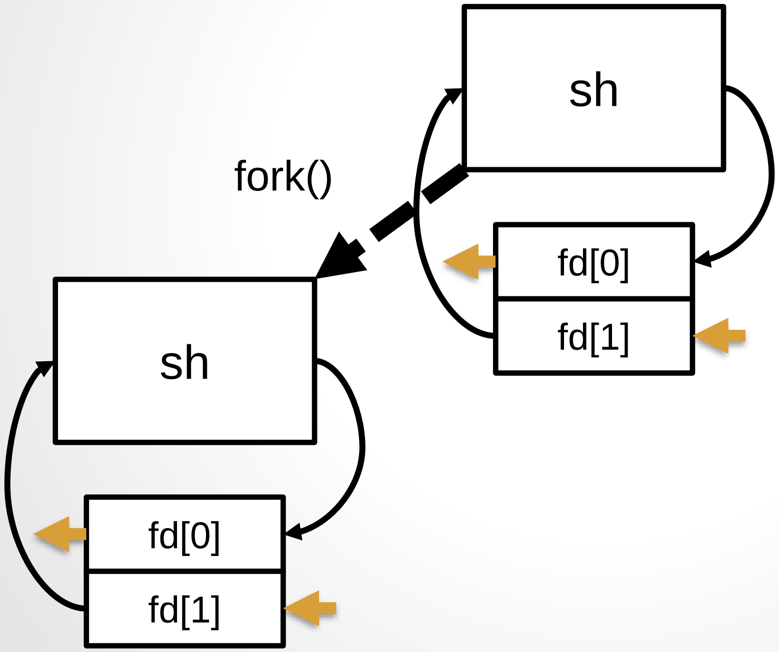
IPC: Pipes in the shell

sh$ > cat file.txt | grep "hello"



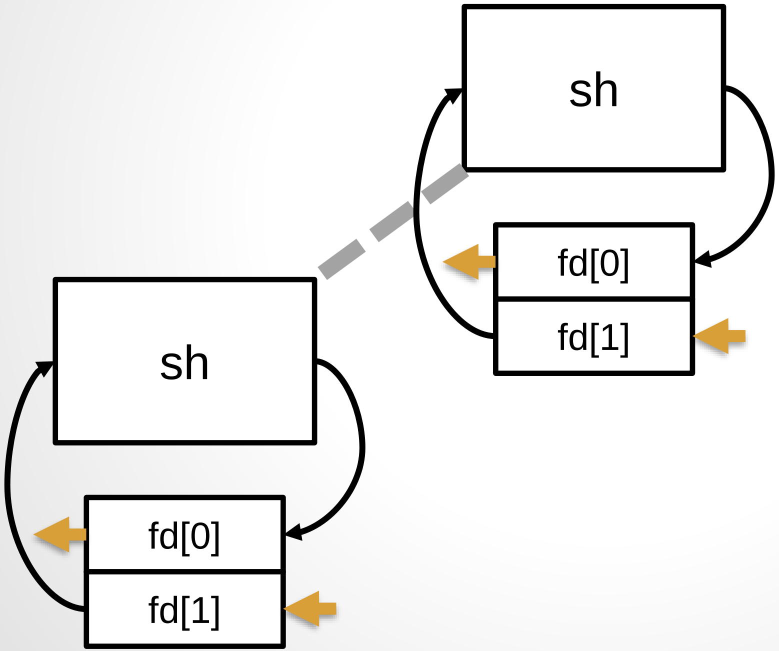
IPC: Pipes in the shell

sh$> cat file.txt | grep "hello"



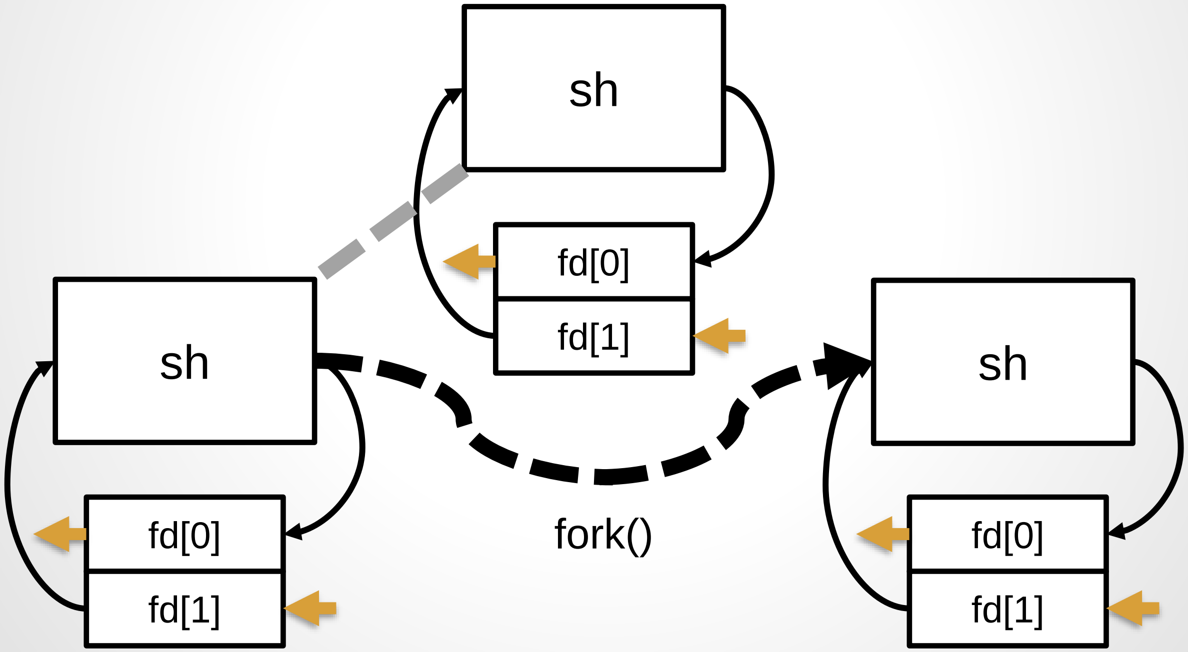
IPC: Pipes in the shell

sh$> cat file.txt | grep "hello"



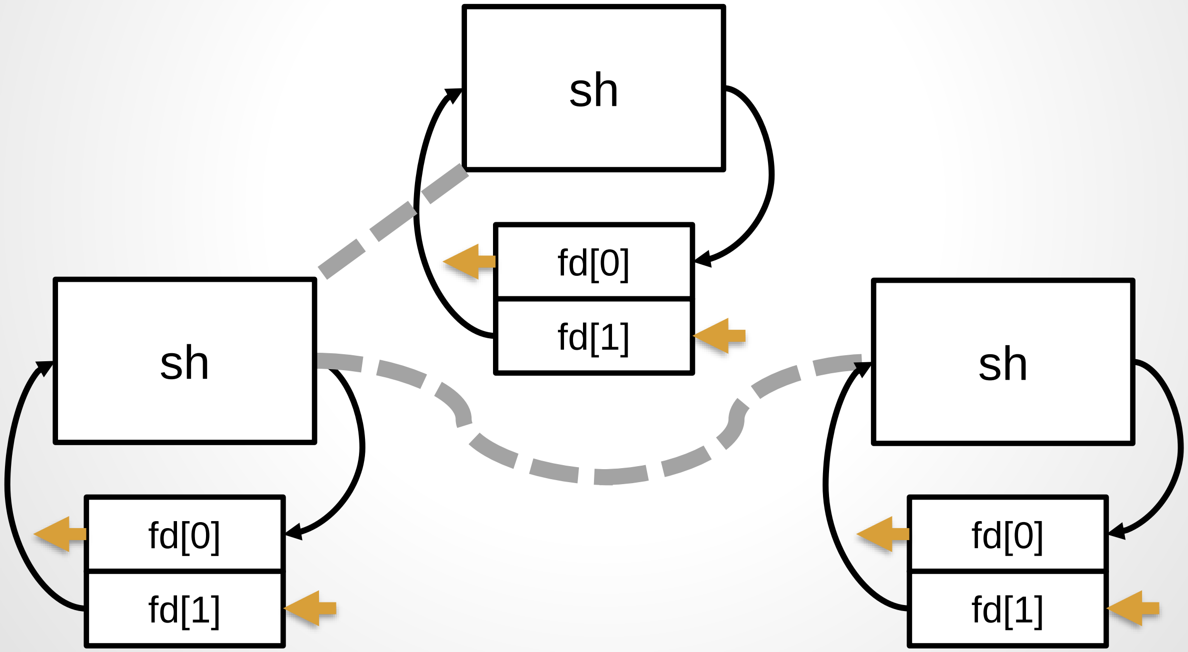
IPC: Pipes in the shell

sh$ > cat file.txt | grep "hello"



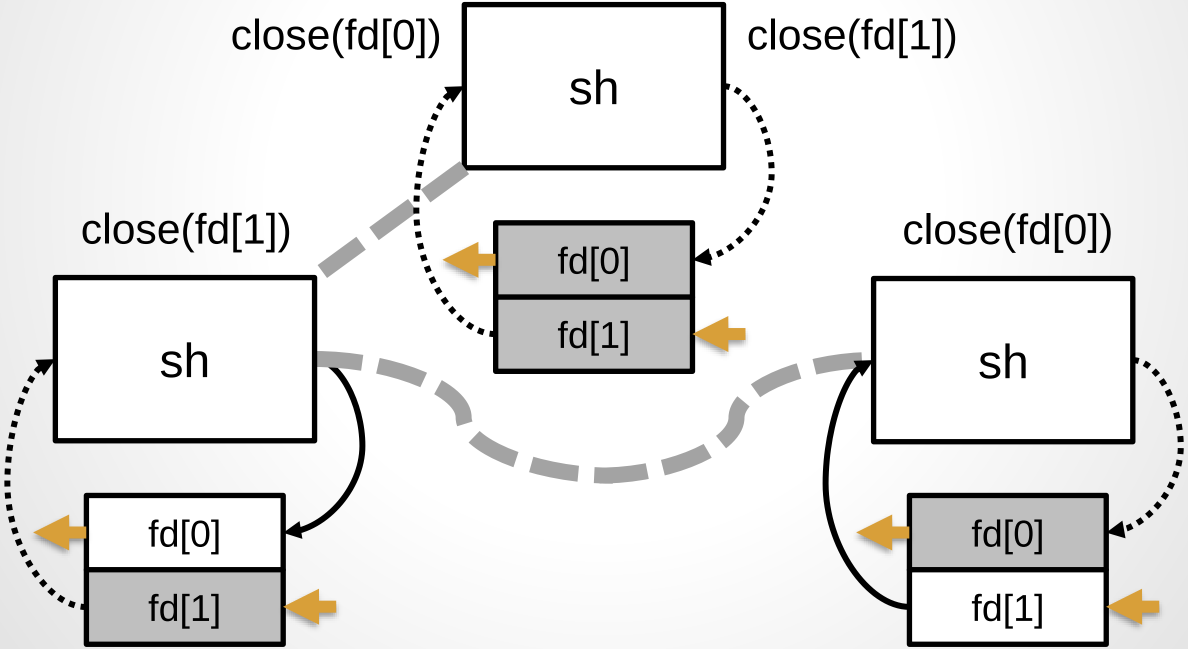
IPC: Pipes in the shell

sh$ > cat file.txt | grep "hello"



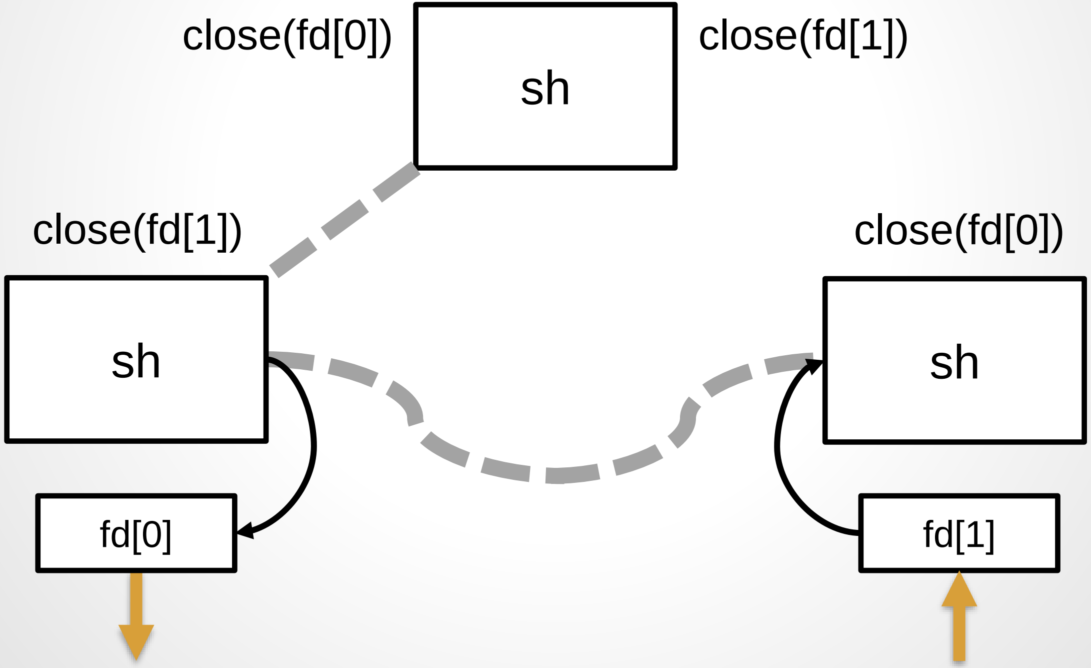
IPC: Pipes in the shell

sh$ > cat file.txt | grep "hello"



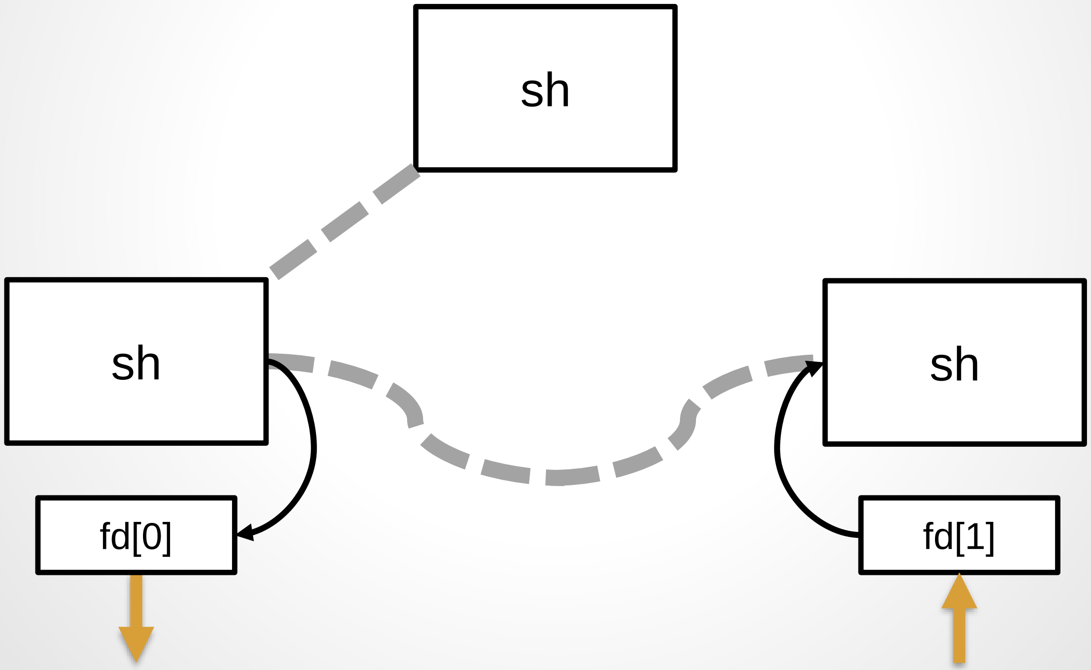
IPC: Pipes in the shell

sh$> cat file.txt | grep "hello"



IPC: Pipes in the shell

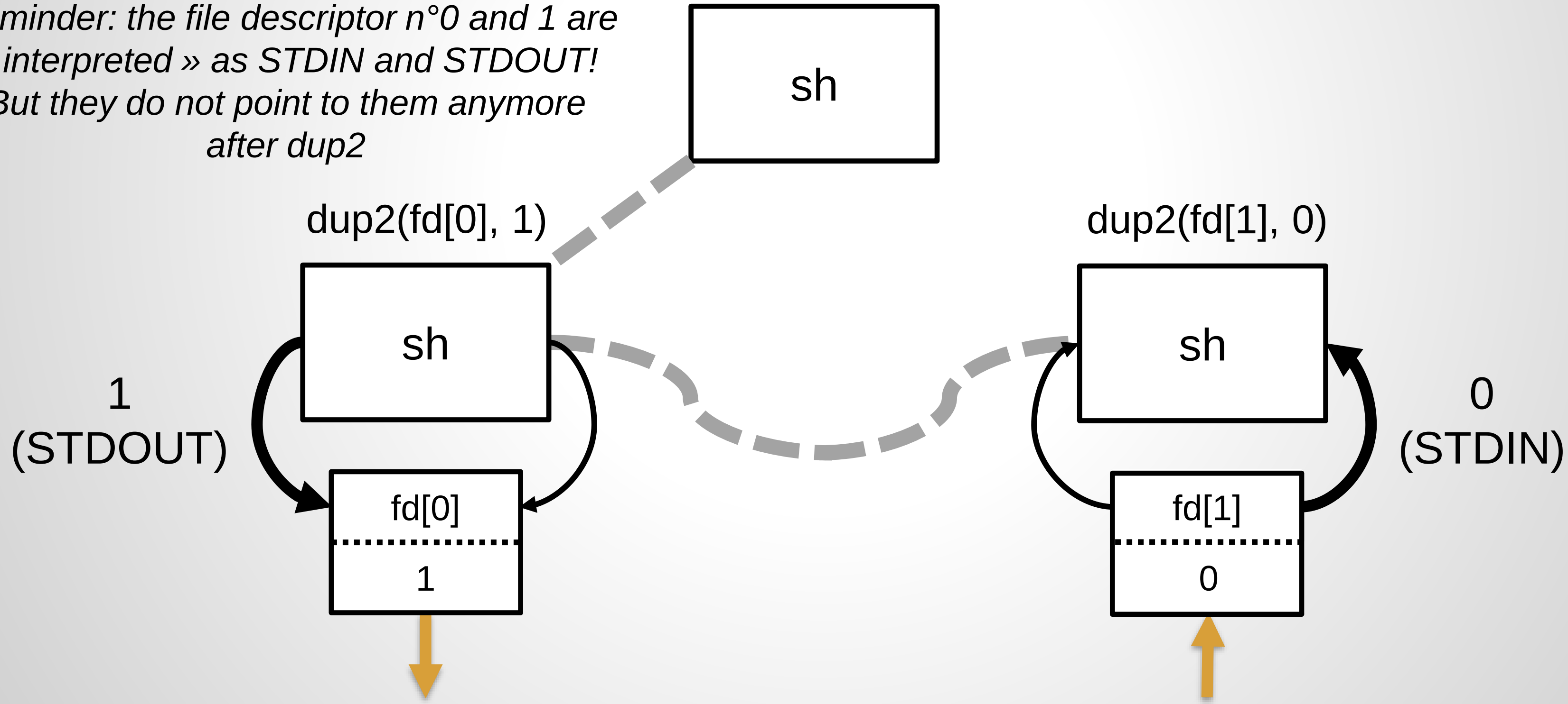
sh$> cat file.txt | grep "hello"



IPC: Pipes in the shell

sh$> cat file.txt | grep "hello"

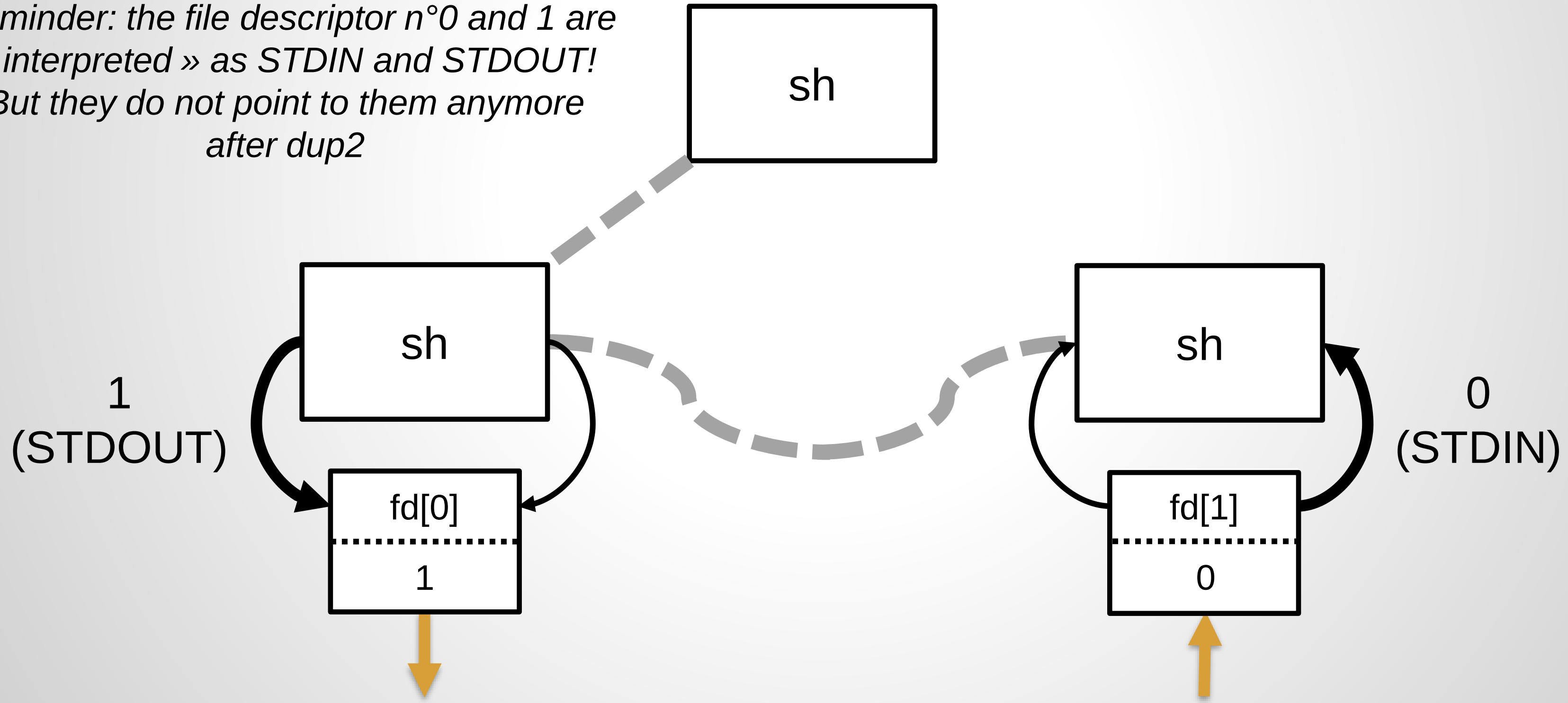
Reminder: the file descriptor n°0 and 1 are « interpreted » as STDIN and STDOUT! But they do not point to them anymore after dup2



IPC: Pipes in the shell

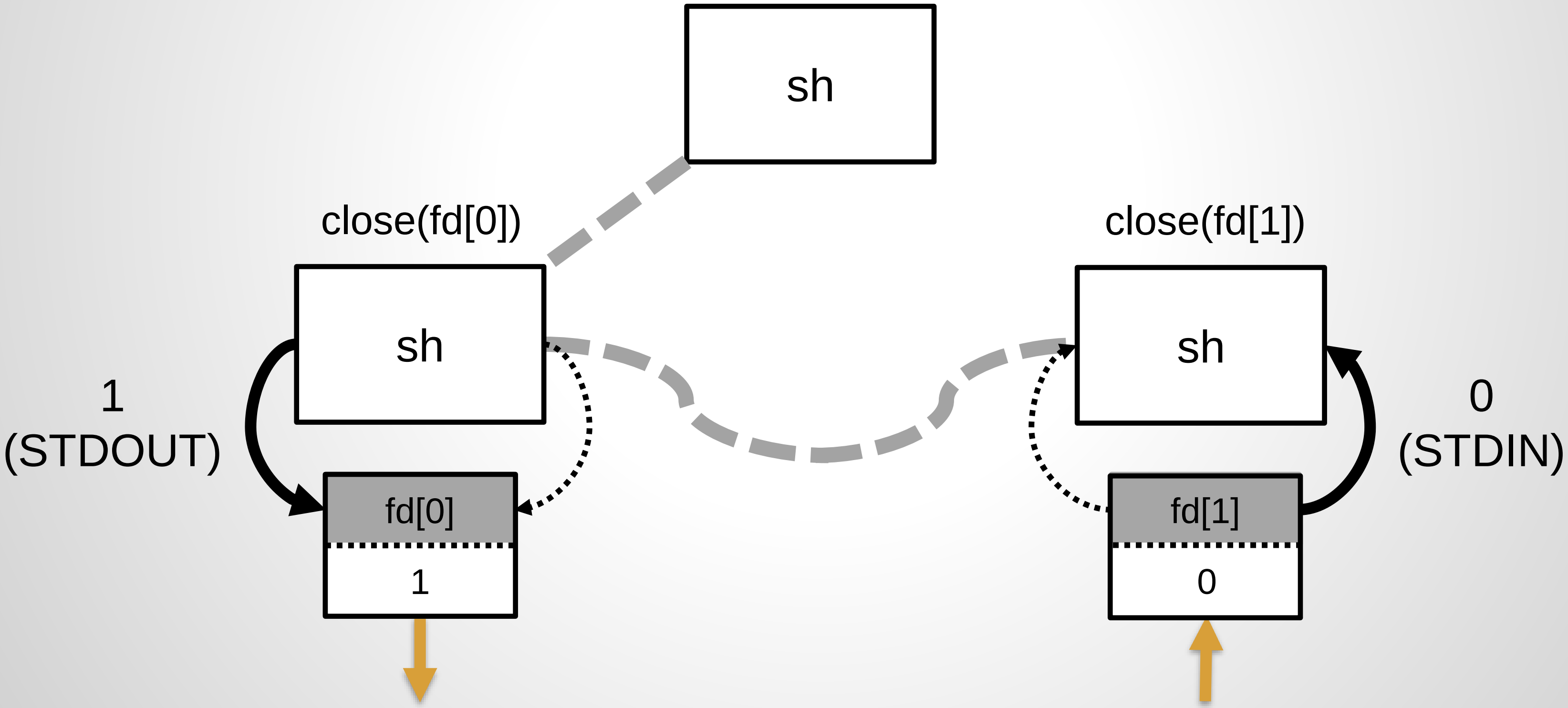
sh$> cat file.txt | grep "hello"

Reminder: the file descriptor n°0 and 1 are « interpreted » as STDIN and STDOUT! But they do not point to them anymore after dup2



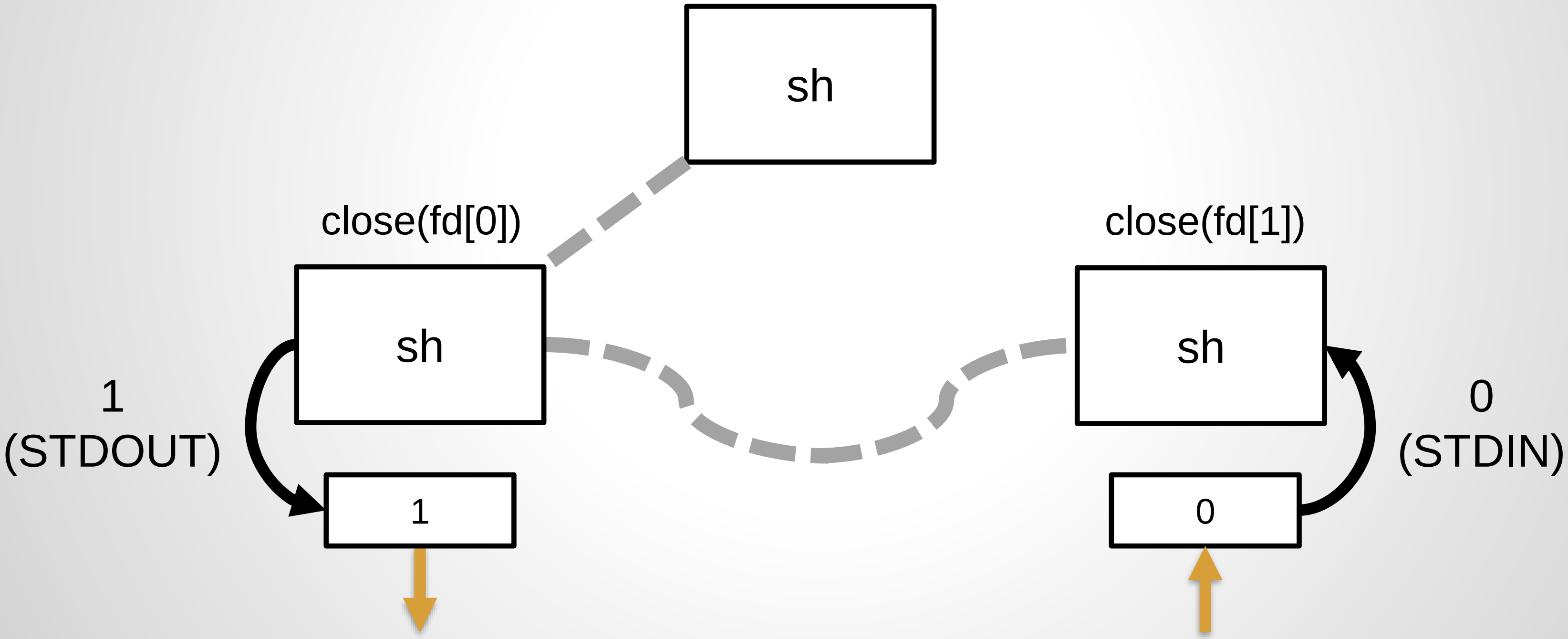
IPC: Pipes in the shell

sh$> cat file.txt | grep "hello"



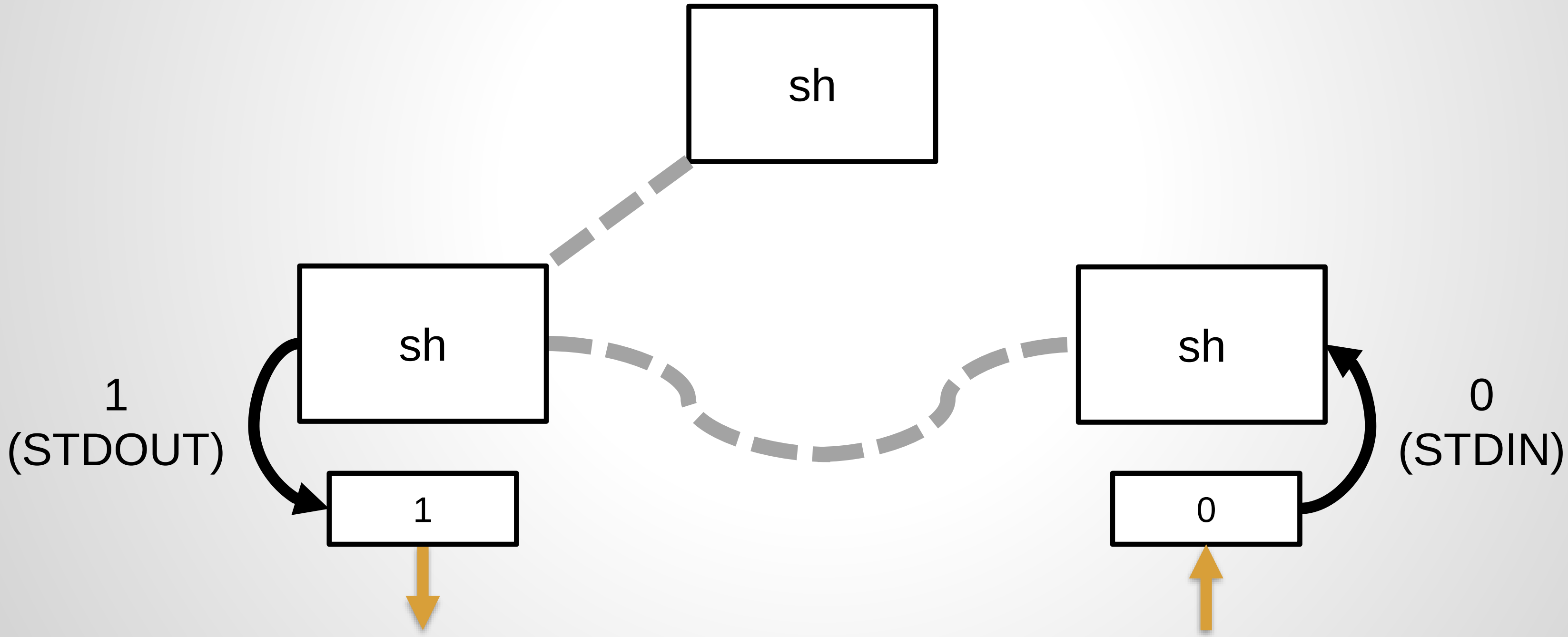
IPC: Pipes in the shell

sh$> cat file.txt | grep "hello"



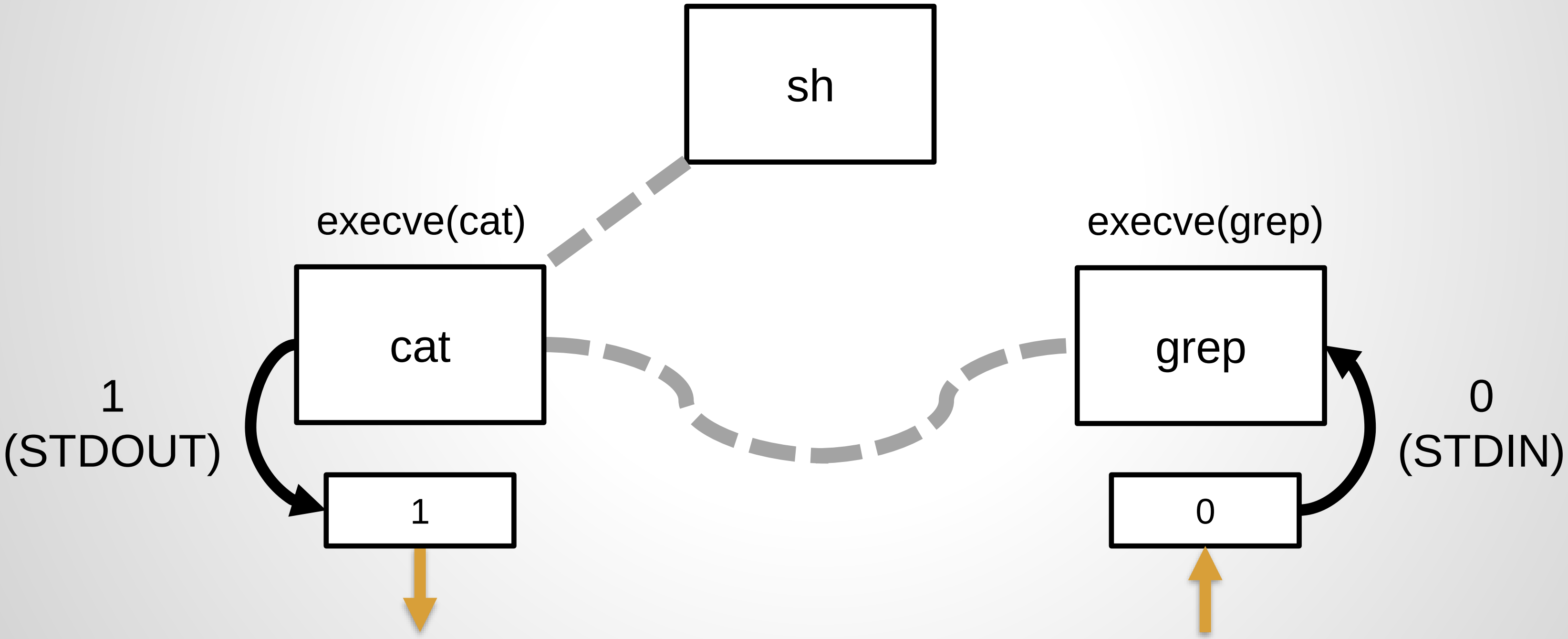
IPC: Pipes in the shell

sh$> cat file.txt | grep "hello"



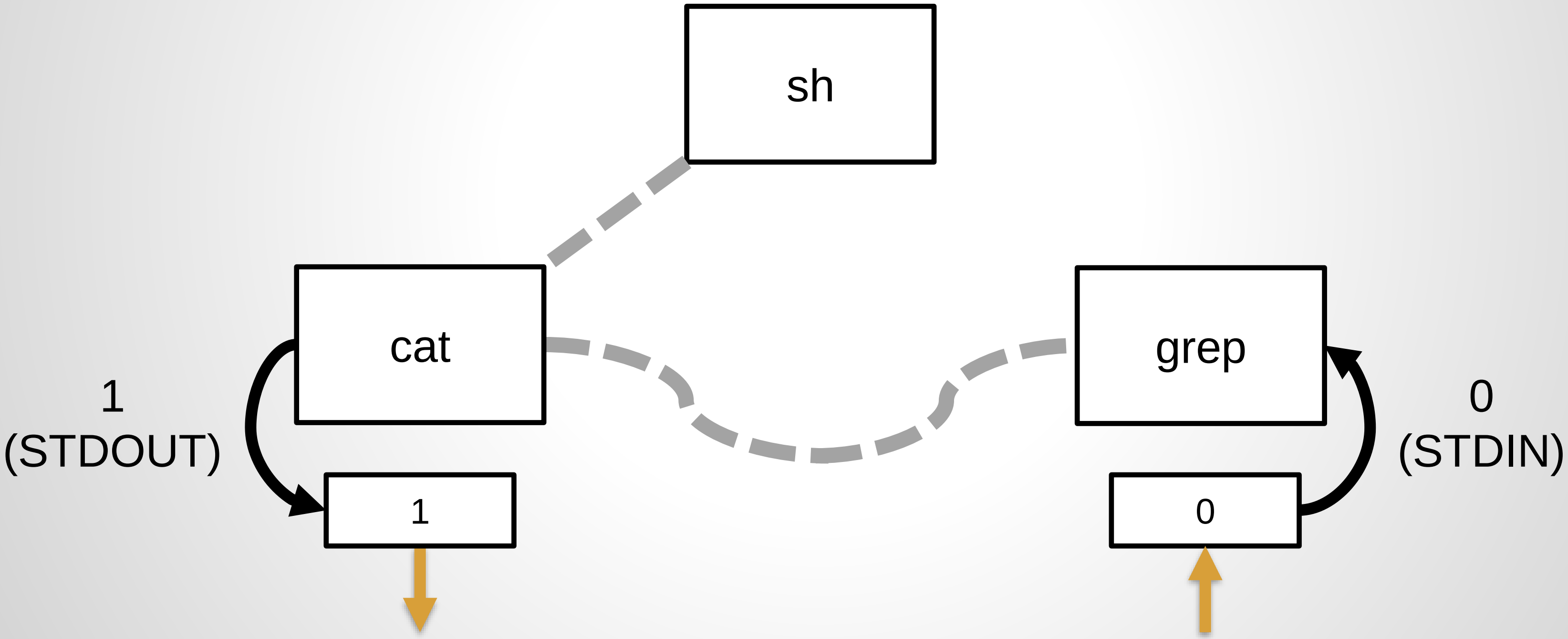
IPC: Pipes in the shell

sh$ > cat file.txt | grep "hello"



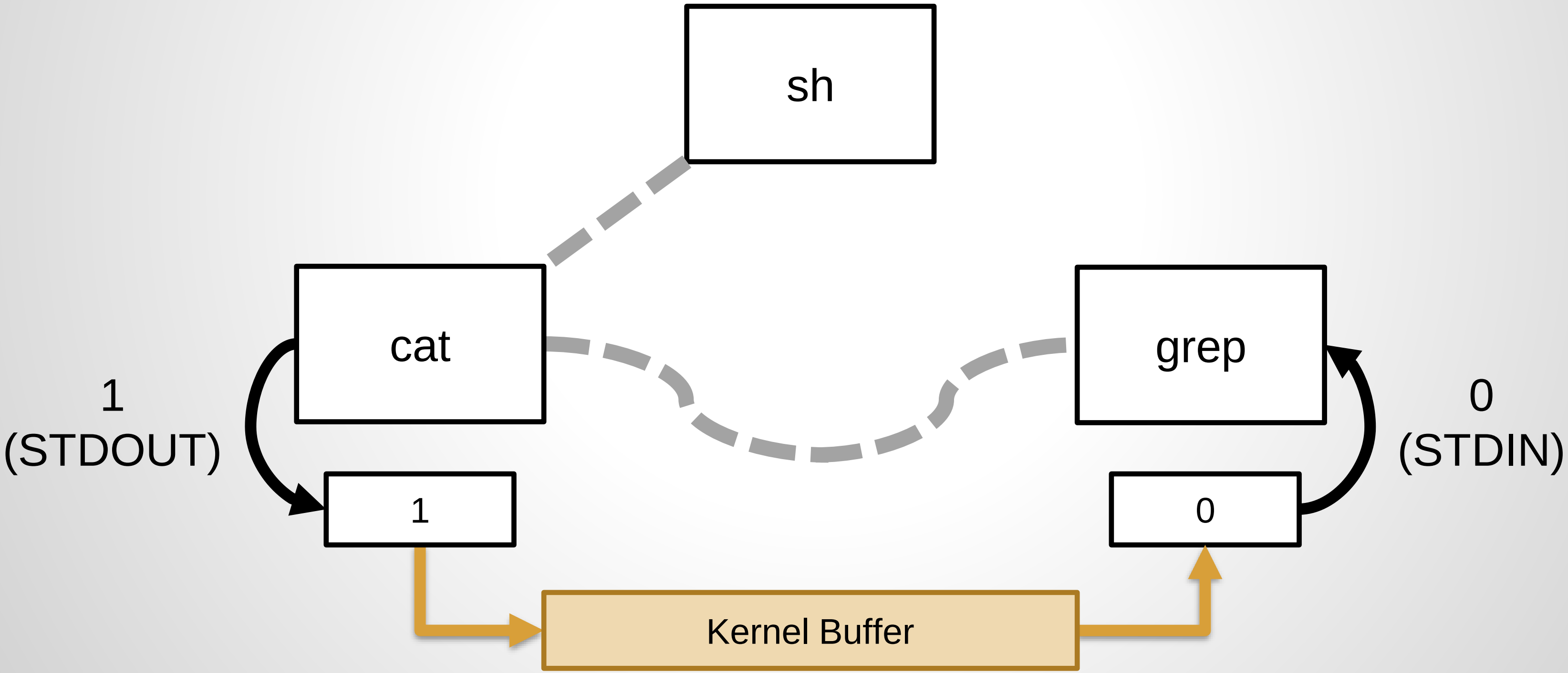
IPC: Pipes in the shell

sh$ > cat file.txt | grep "hello"



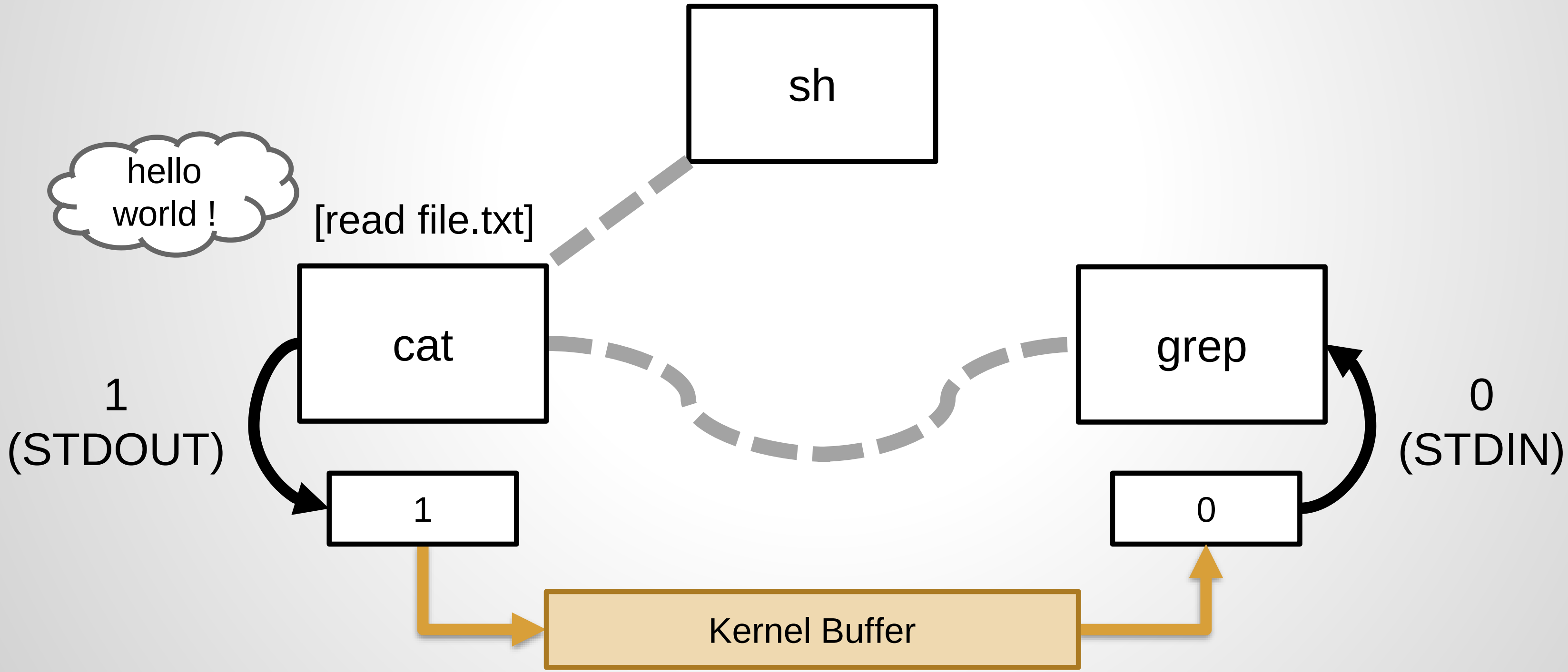
IPC: Pipes in the shell

sh$> cat file.txt | grep "hello"



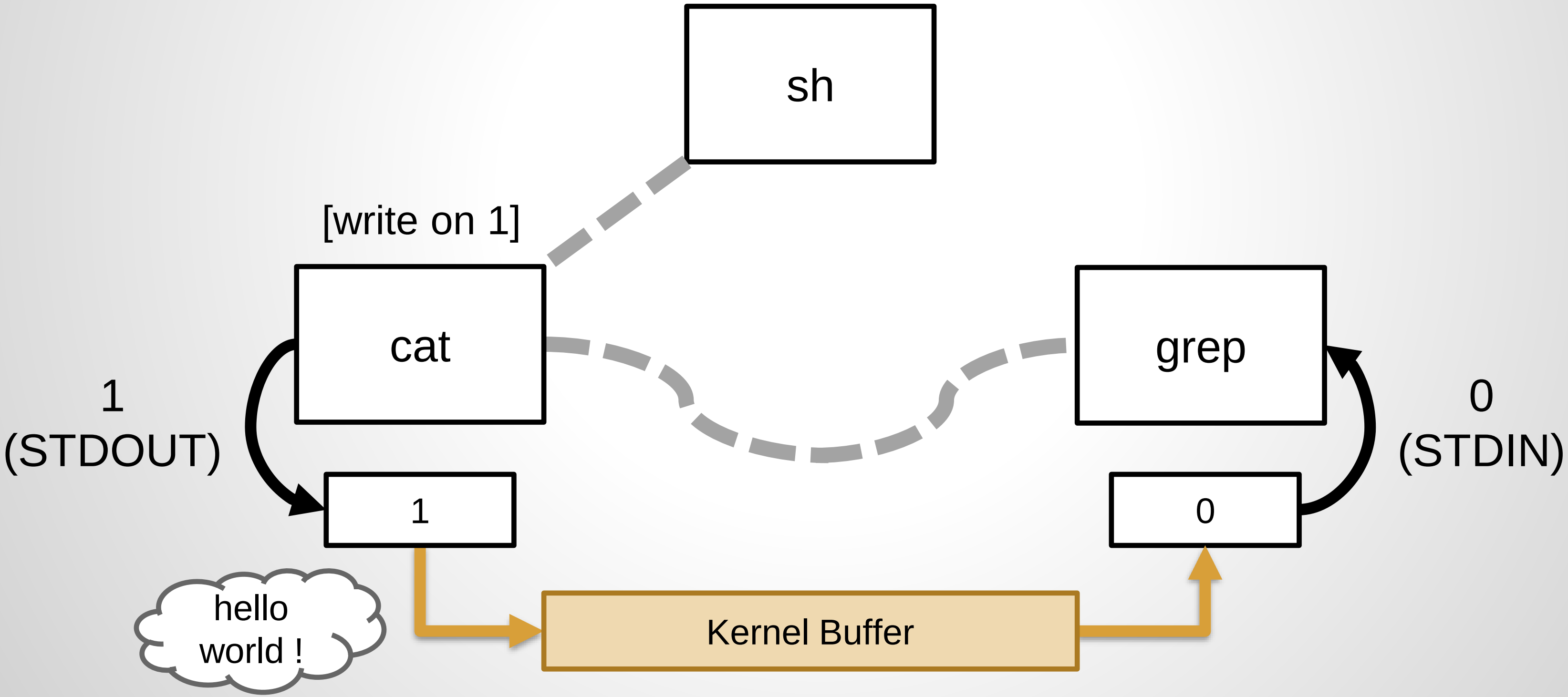
IPC: Pipes in the shell

sh$ > cat file.txt | grep "hello"



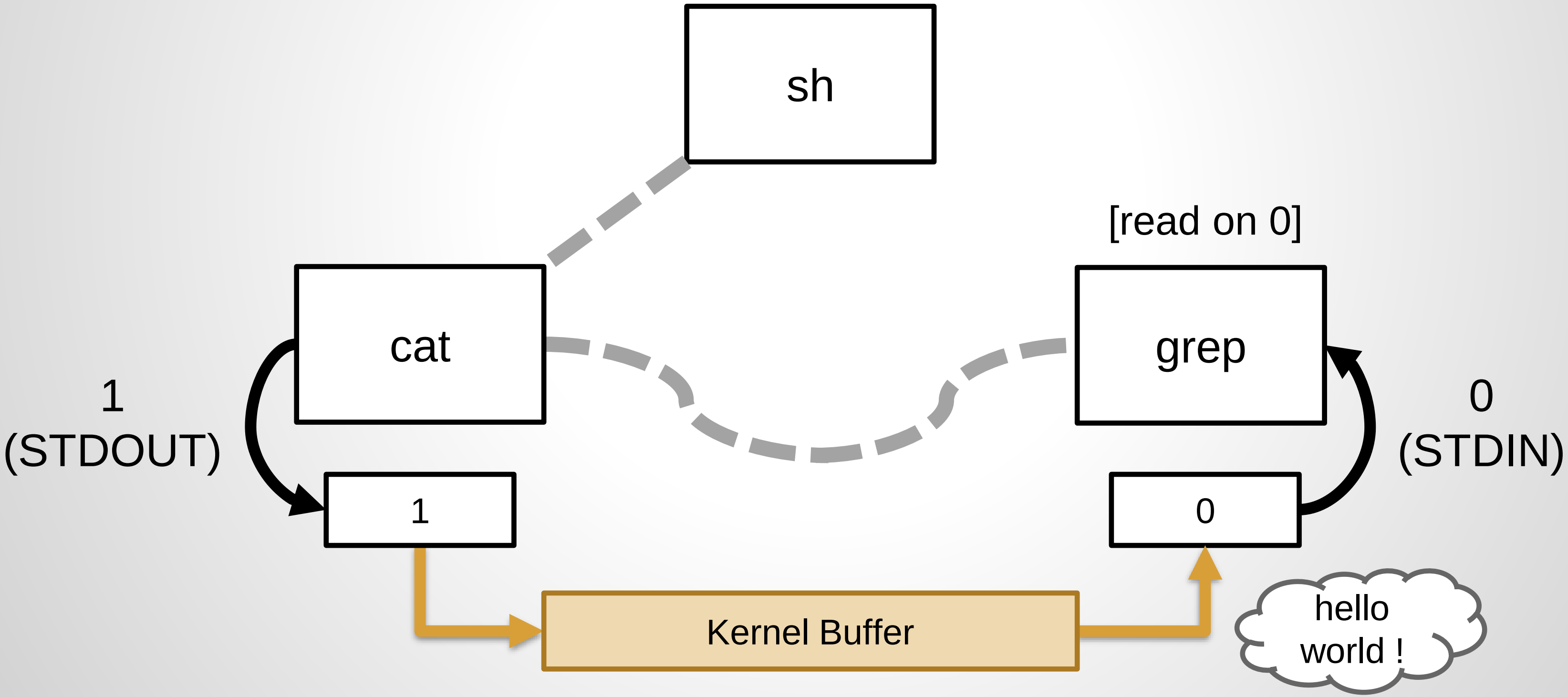
IPC: Pipes in the shell

sh$ > cat file.txt | grep "hello"



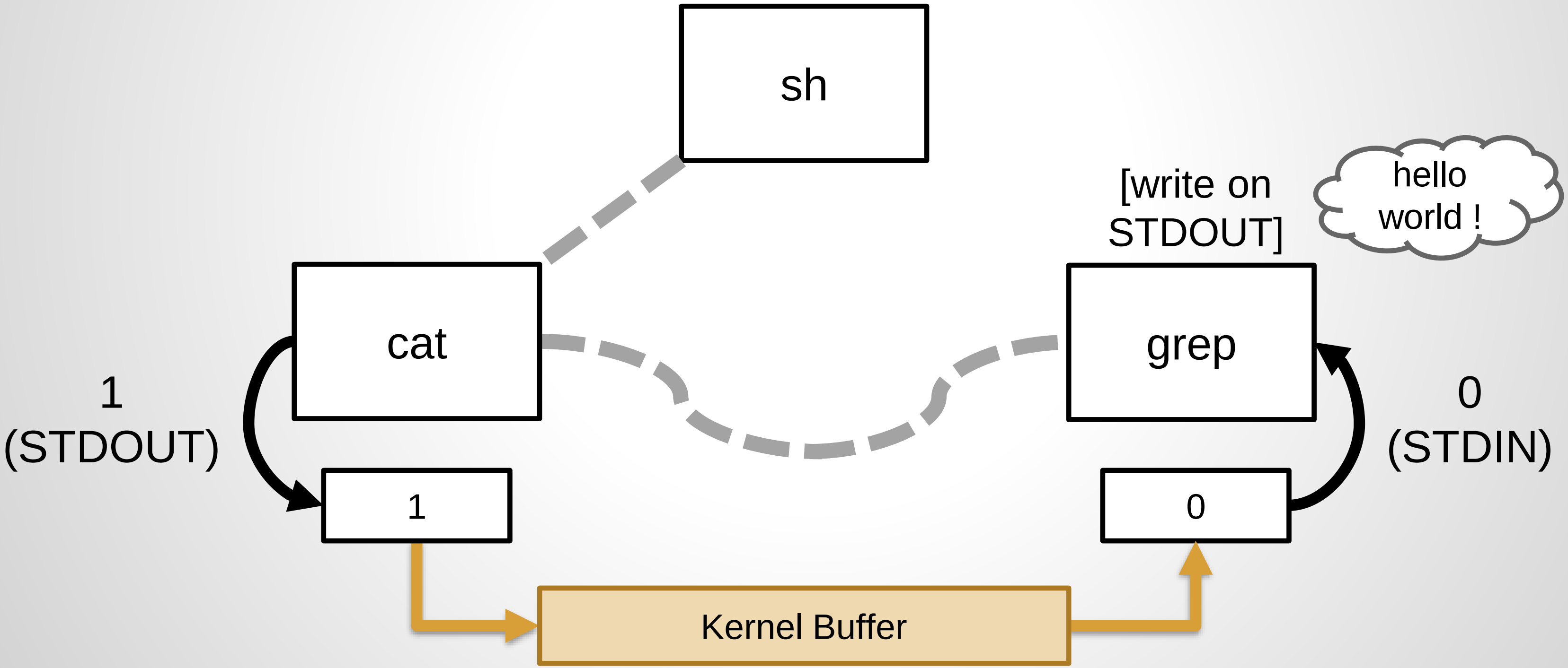
IPC: Pipes in the shell

sh$> cat file.txt | grep "hello"



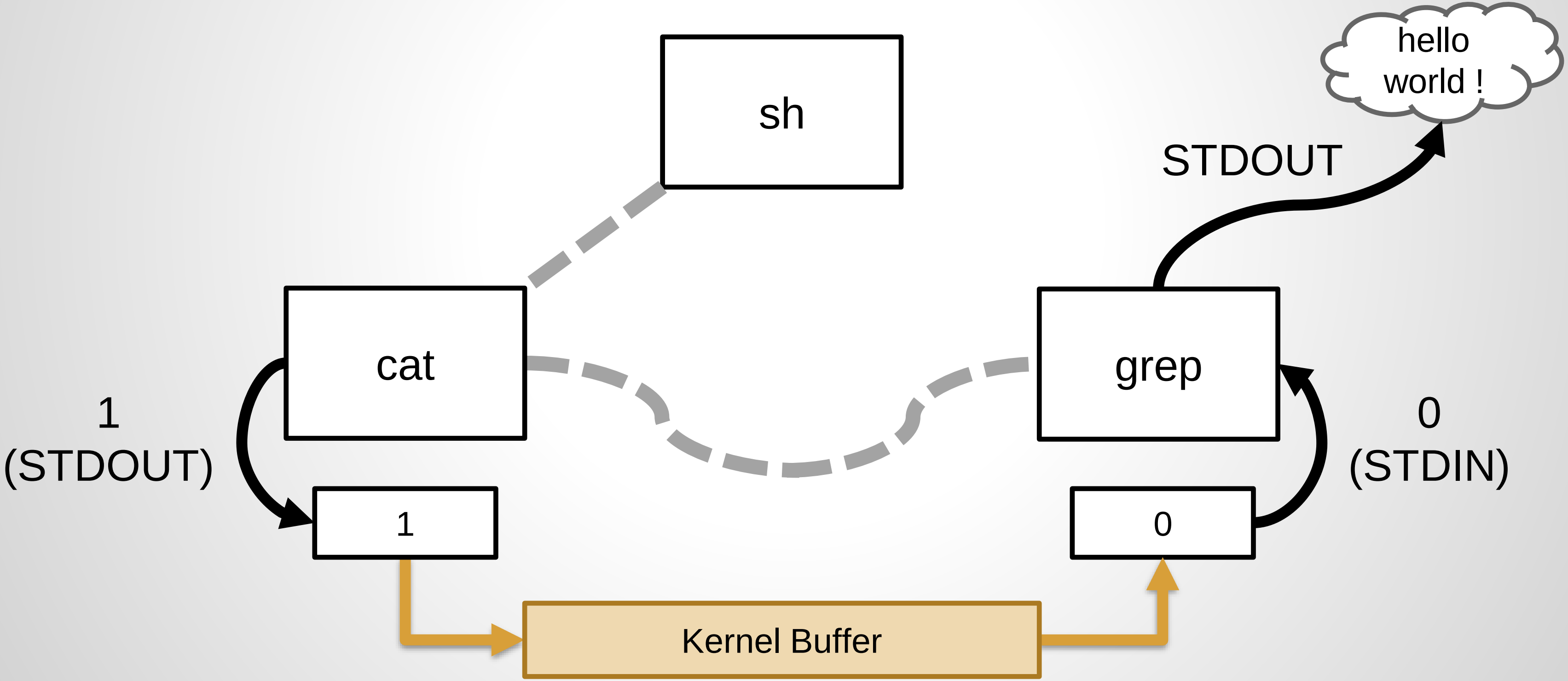
IPC: Pipes in the shell

sh$> cat file.txt | grep "hello"



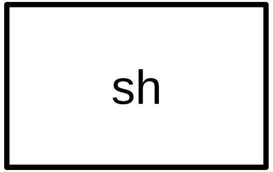
IPC: Pipes in the shell

sh$ > cat file.txt | grep "hello"



IPC: Pipes in the shell

sh$> cat file.txt | grep "hello"



IPC: Named pipes

- Anonymous pipes are created between processes that have a close common parent (or directly between a parent and its child)

- Named pipes are specific files stored on the file system
  - Processes search for a specific pathname in the file tree
  - *See mkfifo(1) and mknod(2)*

- Named pipes can be used for communication between unrelated processes (no close common parent)
  - But each process must know the name of the named pipe

IPC

- Other IPCs require a clear rendez-vous point
  - ftok(2): A « key » is used as the name of the rendez-vous point
  - ipcs(1): prints some of the IPCs currently in use
  - ipcrm(1): removes some active IPCs

- Messages Queues (q)
  - Bi-directionnal communication with messages (type & data)
- Shared Memory (m)
  - Memory shared among processes
- Semaphores (s)
  - Used for synchronization

IPC: Messages Queues

- Mailbox of messages

- A message is a structure with a type, and data
  - Messages have a precise length
- Creation / Destruction: *msgget(2) / msgctl(2)*
  - Defines a clear name for the messages queue (the « key »)
  - Might be restricted to child processes only
- Send / Receive: *msgsnd(2) / msgrcv(2)*
  - Reception is blocking if no message is available
  - The message structure 1st field & msgrcv 4th parameter allow to choose which message get *(see in the manual how to use them smartly)*

IPC: Shared Memories

- Memory shared among multiple processes

- Synchronization is *required*
  - See semaphores and mutexes...

- Creation / Destruction: *shmget(2) / shmctl(2)*
  - Defines a clear name for the shared memory (the « key »)
  - Decides the length of the memory part to share
- Attach / Dettach: *shmat(2) / shmdt(2)*
  - Uses a pointer to the shared memory
  - fork(2) shares the memory between parent and child

IPC: Semaphores *(synchronization)*

- Counter shared among multiple processes
  - Cannot be below 0
  - Must use specific routines to change the value
  - *sem_overview(7): named and unnamed semaphores*

- Creation / Destruction: *sem_init(3) sem_open(3) / sem_destroy(3)*
  - Defines a clear name for the shared memory (the « key »)
  - Defines the maximum value of the counter

- Increase / Decrease: *sem_post(3) / sem_wait(3)*
  - sem_post(3): increases the counter
  - sem_wait(3): decrements the counter OR waits until it can be decreased

Quick overview of the threads

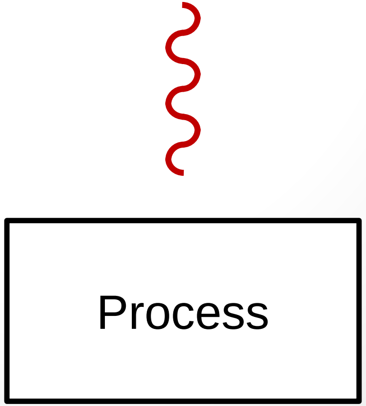
Multithreading

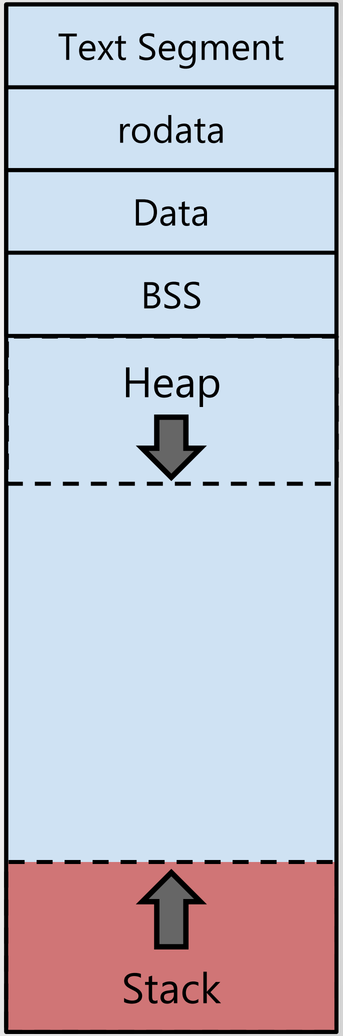
- Problems: How to...
  - Allows parallelism inside a process?
  - Reduces the cost of context switching?

- Solution
  - Thread (lightweight process): state, registers & stack
    Shares other resources
  - Process: group of threads.
    Classical process = process with only 1 thread

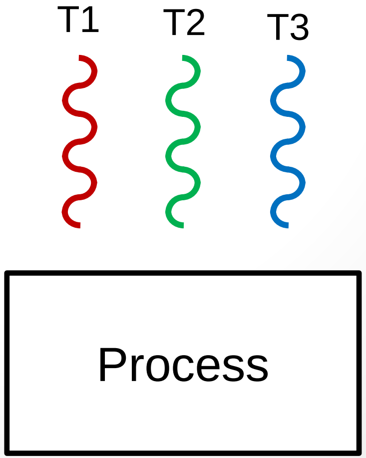
- Functionalities
  - Same as a process: creation, termination, state, etc...
  - New issues: concurrent access on shared resources

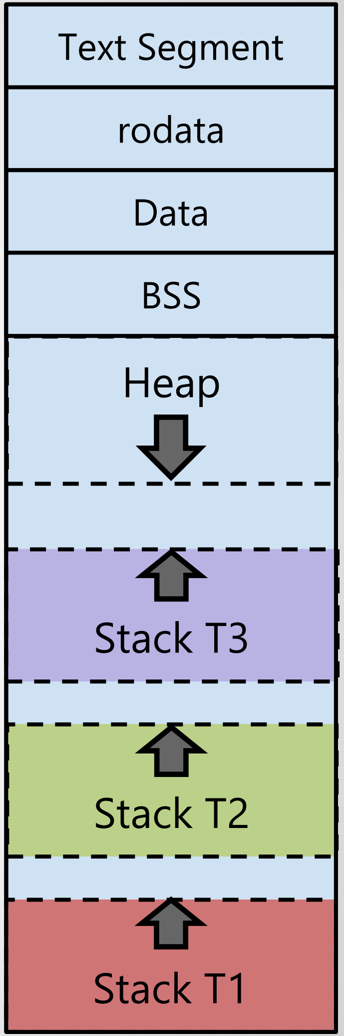
```c
char *myStr;
int i = 0;
int        main(void)
{
const char *var = "Test";
int a = 1337;

i = addition(21, 42);
myStr = malloc(32 * sizeof (char));
return (0);
}
```





```c
char *myStr;
int i = 0;
int main(void)
{
const char *var = "Test";
int a = 1337;

i = addition(21, 42);
myStr = malloc(32 * sizeof (char));
return (0);
}
```

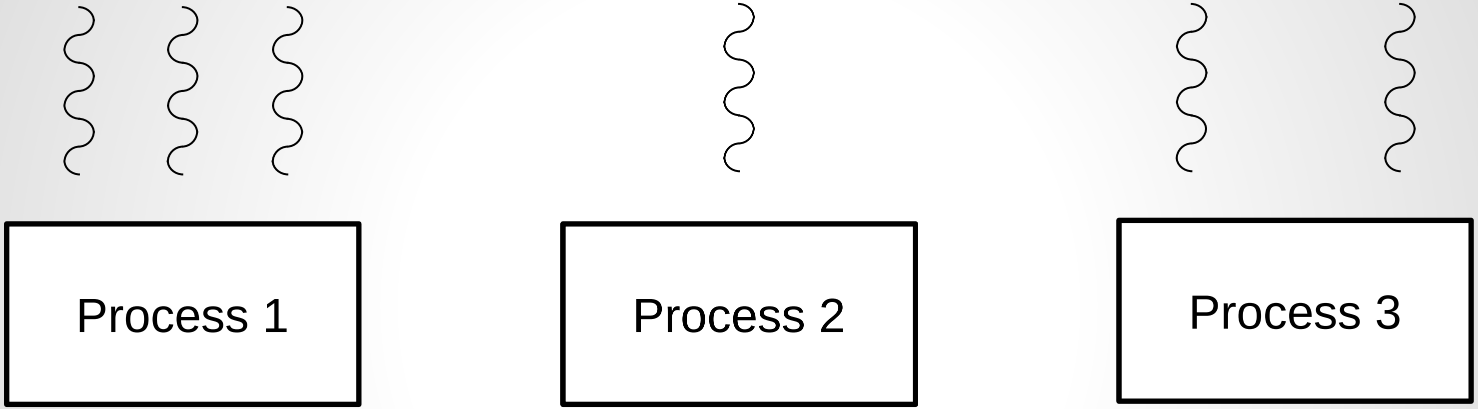




Userland Threads

- Principle
  - Implemented as a library in userland
  - 1 thread table per process (managed by the library)
- Pros
  - Usable on a system without support for threads
  - Fast context switching (no kernel trap)
  - Customizable scheduling algorithm
- Cons
  - Needs for unblocking syscalls
  - Threads can lock the CPU (they need to yield explicitly)
  - Threads are used to alleviate blocking

Userland Threads



The scheduler gives time to « processes », not to their threads. Each process manages itself the thread to execute (*yields* instructions).

Kernel

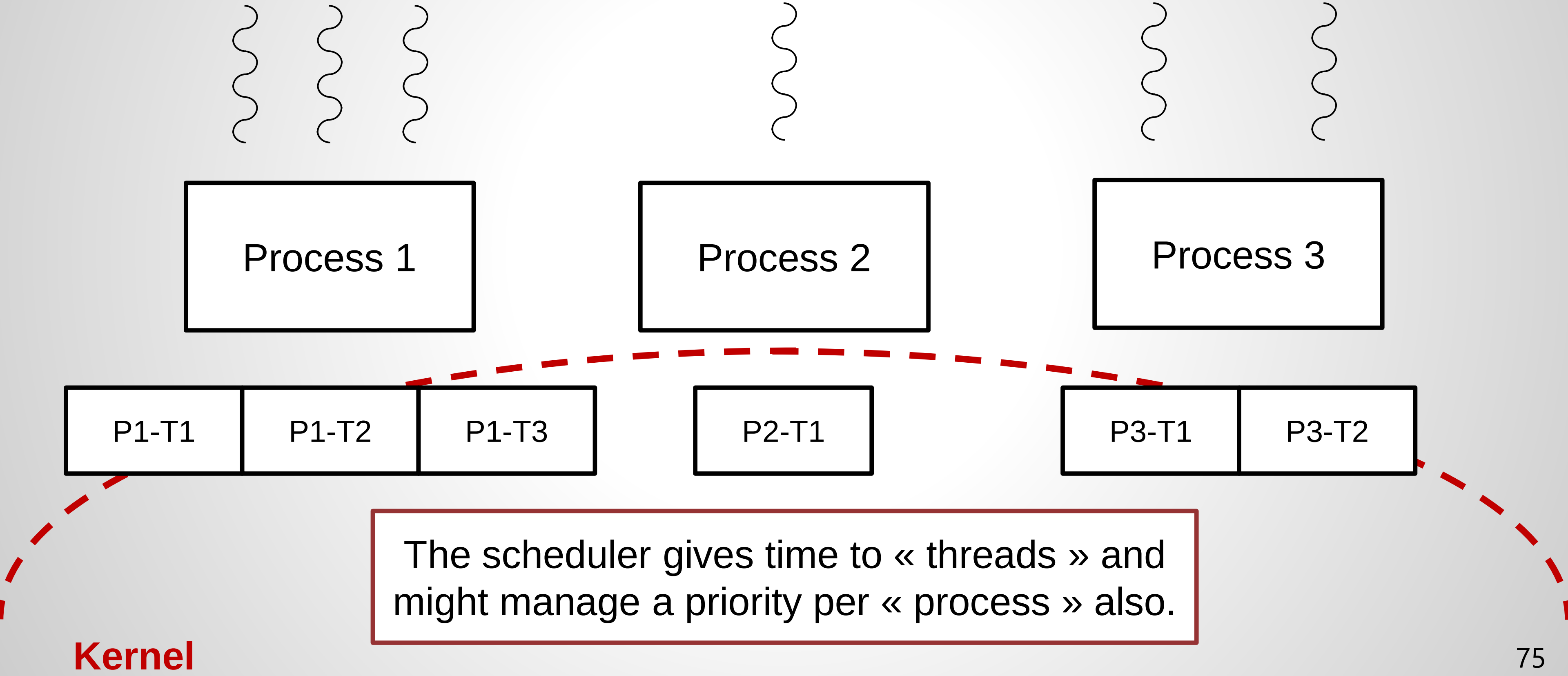
Kernel Threads

- Principle
  - Adds a thread table inside the process table (used by the scheduler)
  - Every blocking call is implemented as a syscall

- Pros
  - Ease to create an application using them
  - No need for non blocking calls

- Cons
  - Creation/deletion/bookkeeping have a cost
  - Interrupt & blocking syscalls

Kernel Threads



The scheduler gives time to « threads » and might manage a priority per « process » also.

Unified API: Pthread

- POSIX API used to run threads
- Simple unified interface for multi threaded environment on POSIX system
- Beware: everything is shared between threads...
  - Except the thread ID in the scheduler => each thread is independent
- pthreads(7)
  - pthread_create(3), pthread_join(3), pthread_yield(3), ...

Where? What?

Per Thread

- Thread ID
- Signal mask
- Errno
- Scheduling policy
- Capabilities
- CPU affinity

Per Process

- Process ID
- Parent Process ID
- Process Group
- User/Group ID
- File descriptors
- umask
- Current directory
- Limits
- ...

Concurrency problems

Concurrency

A ressource is shared among multiple processes…

…they all want to access it simultaneously!

How to manage this case?…

…well, is there only one case of concurrency?

Concurrency

- One ressource to share
  - Available / Used
- How many processes or threads try to access it?
  - Maximum of users
- Usual processing:
  1. Access to the ressource (lock)
  2. Process
  3. Release of the ressource (unlock)

Concurrency

- Mutual exclusion (mutex)
- Ressource allocation
- Producers - Consummers
- Readers - Writers

Mutual exclusion

```
Booking()
{

  if (AvailableSeats > 0)
  {

    /* Processing for booking */

     AvailableSeats -= 1;

  }

}
```

- If 1 process executes this code, *AvailableSeats* will stay at 0 or above
- If 2 processes executes simultaneously this code…
  …In some cases, *AvailableSeats* might fall below 0

Mutual exclusion: critical section

```
Booking()
{

  if (AvailableSeats > 0)
  {

    /* Processing for booking */

     AvailableSeats -= 1;

  }

}
```

- This part is a *critical section*
  - only « one » process (or thread) can execute it if you wish to stay in a coherent state

- 3 properties:
  - Mutual exclusion
  - Bounded wait
  - Progress

Mutual exclusion: critical section

- Mutual exclusion
  - Only one process/thread can be « inside » the critical section
- Bounded wait
  - A process/thread waiting to access the critical section must be able to execute it within a maximum waiting time
- Progress
  - If no process/thread is « inside » the critical section, nothing must block its access, and the processes/threads waiting to enter must decide which one will enter first

Mutual exclusion: Blocking interrupts

```
Booking()
{

  disable_interrupt();

  if (AvailableSeats > 0)
  {
    /* Processing for booking */
      AvailableSeats -= 1;
  }

  enable_interrupt();
}
```

- Blocking interrupts
  - Useful for kernel processing like scheduling or cache protection
- Pros:
  - Cannot interrupt the execution while in the critical section
- Cons:
  - The process must be supervisor
  - All the processes become unaware of interruptions...
  - Do not work on multi core/CPU

Mutual exclusion: Test and Set

```
int *lock = 0;

Booking()
{

  while (TAS(lock) == 1)
  { }
  if (AvailableSeats > 0)
  {
    /* Processing for booking */
      AvailableSeats -= 1;
  }
  (*lock) = 0;
}
```

- Test and Set *(TAS)*
  - Atomic operation
- Pros:
  - The ressource is really protected
- Cons:
  - Busy waiting for others
  - Do not ensure fair access to critical section

Mutual exclusion: Test and Set

```c
int TAS(int *lock)
{
  disable_interrupt();

 int test;

  test = (*lock);
  (*lock) = 1;
  return (test);

  enable_interrupt();
}

// A high level vision of TAS
// (it must be "atomic")
```

```c
int *lock = 0;

Booking()
{

  while (TAS(lock) == 1)
  { }
  if (AvailableSeats > 0)
  {
    /* Processing for booking */
      AvailableSeats -= 1;
  }
  (*lock) = 0;
}
```

Mutual exclusion: Compare and Swap

```
int *lock = 0;

Booking()
{

  key = 1;
  while (key == 1)
    SWAP(lock, key);
  if (AvailableSeats > 0)
  {
    /* Processing for booking */
      AvailableSeats -= 1;
  }
  (*lock) = 0;
}
```

- Compare and Swap *(CAS)*
  - Swaps two values
  - Atomic operation
- Pros:
  - The ressource is really protected
- Cons:
  - Busy waiting for others
  - As all the accesses are in a readable memory

Mutual exclusion: Swap example

```
int SWAP(int *A, int *B)
{
  disable_interrupt();

  int tmp;

  tmp = (*B);
  (*B) = (*A);
  (*A) = tmp;

  enable_interrupt();
}

// A high level vision of CAS
// (it must be "atomic")
```

```
int *lock = 0;

Booking()
{

  key = 1;
  while (key == 1)
    SWAP(lock, key);
  if (AvailableSeats > 0)
  {
    /* Processing for booking */
      AvailableSeats -= 1;
  }
  (*lock) = 0;
}
```

Mutual exclusion: Semaphores

```
InitSemaphore(Mutex, 1);

Booking()
{
  P(Mutex); // Take a token

  if (AvailableSeats > 0)
  {
    /* Processing for booking */
      AvailableSeats -= 1;
  }

  V(Mutex); // Release a token
}
```

- Semaphore
  - Distributes tokens
  - Has a maximum of tokens
  - Managed by the kernel
- Init
  - Initialize counter
- P
  - Distribute tokens
  - Block if no available token
- V
  - Take back tokens

Mutual exclusion: Semaphores

```
P(sem_t sem)
{
  disable_interrupt();

  sem.Count -= 1;
  if (sem.Count < 0)
  {
    // Block current process
    enqueue(sem.Queue, cur_PID);
    cur_PID.state = blocked;
    reschedule = true; // yield
  }

  enable_interrupt();
}
```

```
V(sem_t sem)
{
  disable_interrupt();

  sem.Count += 1;
  if (sem.Count <= 0)
  {
    // 1+ process is waiting
    process = dequeue(sem.Queue);
    process.state = running;
    reschedule = true; // yield
  }

  enable_interrupt();
}
```

Mutual exclusion: Semaphores

```
InitSemaphore(Mutex, 1);

Booking()
{
  P(Mutex); // Take a token

  if (AvailableSeats > 0)
  {
    /* Processing for booking */
      AvailableSeats -= 1;
  }

  V(Mutex); // Release a token
}
```

```
InitSemaphore(sem_t sem, int val)
{
  disable_interrupt();

  // Maximum tokens given
  sem.Count = val;

  // Waiting queue
  sem.Queue = NULL;

  enable_interrupt();
}
```

Mutual exclusion

Lot of other atomic instructions or algorithms:

- Fetch-and-Add
- Dekker’s algorithm
- Peterson’s algorithm
- …

Tips: « mutex » can be a state in which 2+ processes must be mutually excluded in order to execute a critical section…

…or it can be the surname to a simple « lock »

Deadlocks

- 2 processes/threads try to access to 2 ressources that they mutually blocked

- 1st process blocked the 1st ressource and… *[stopped by scheduler]*
- 2nd process blocked the 2nd ressource and try to access the 1st ressource *[is paused by blocking call]*
- 1st process tries to access 2nd ressource *[is paused by blocking call]*

Both processes are blocked and won’t run anymore

(SIGKILL is your sole friend in that case… if you can use it on these processes)

Deadlocks

```
Process 1
{
  P(ressource1);
  P(ressource2);

 /* Usage of ressources */

  V(ressource2);
  V(ressource1);
}
```

```
Process 2
{
  P(ressource2);
  P(ressource1);

 /* Usage of ressources */

  V(ressource1);
  V(ressource2);
}
```

Extended problem to N processes:
Philosopher's diner

Starvation

- The property assuring a process/thread to access a ressource is not met
  - The process/thread trying to access a ressource is « never » able to access it

- Multiple possibles causes
  - Bad logic
  - Deadlock
  - No order is assured when a process is asking for a ressource *(no FIFO on the waiting queue)*
  - …

Readers and Writers problem

- Multiple writers (write data) and readers (read data)

- Writers
  - Only 1 writer can write simultaneously
  - Nobody can read
  - Block the data (file, shared memory, ...)

- Readers
  - Multiple readers can read simultaneously
  - Nobody can write
  - 1^{st} reader must block the data...
  - ...Last reader must wake up the writer *(if they are some waiting)*

Producers and Consumers

- 1 shared buffer and 2 indexes
  - One index for producing, one index for consuming
  - Read and write of data are sequential/in a circular FIFO

- Producers
  - Write in N cells of the buffer from the producing index
  - Cannot « produce » data if the buffer is full
  - Cannot work in the same cell of the consumers
- Consumers
  - Read N cells of the buffer from the consuming index
  - Cannot « consume » data if the buffer is empty
  - Cannot work in the same cell of the producers

Other synchronization tools

- Monitors
  - A construction that uses mutex and other tools in order to be *thread-safe*
  - Thread-safe: allows one ressource to be safely used by multiple threads (no deadlocks can happen, neither

- Barriers
  - Align multiple processes/threads on a specific point during execution
  - Processes/Threads are put in a block state until the last one reaches the barrier