System Programming

Ion Mandoiu
Laurent Michel

Overview

- Motivation
- Process Life-cycle
  - Fork
  - Wait
  - Exec
  - File handles
- Pipes
- FIFO

Motivation

- Processes
    - Historically the core concept in O.S.
    - Represent a *running* executable
        - Code + Processor State + Memory + Resource usage
- Organization in O.S.
    - Processes form a tree rooted at “init”
    - Every process has a parent
    - Every process has 0 - * children

```
init─┬─/usr/sbin/apach───8*[/usr/sbin/apach]
     ├─/usr/sbin/spamd───2*[spamd child]
     ├─acpid
     ├─atd
     ├─bluetoothd
     ├─btsync───6*[{btsync}]
     ├─console-kit-dae───64*[{console-kit-dae}]
     ├─cron
     ├─cupsd
     ├─2*[dbus-daemon]
     ├─dbus-launch
     ├─dhcpd
     ├─dovecot─┬─anvil
     │         ├─auth
     │         ├─config
     │         ├─12*[imap]
     │         ├─18*[imap-login]
     │         └─log
     ├─6*[getty]
     ├─irqbalance
     ├─master─┬─anvil
     │        ├─pickup
     │        ├─qmgr
     │        └─tlsmgr
     ├─mdm───mdm─┬─Xorg───2*[{Xorg}]
     │           └─mdmwebkit───6*[{mdmwebkit}]
     ├─2*[named───6*[{named}]]
     ├─nmbd
     ├─ntpd
     ├─polkitd───{polkitd}
     ├─rpc.idmapd
     ├─rpc.mountd
     ├─rpc.statd
     ├─rpcbind
     ├─rsyslogd───3*[{rsyslogd}]
     ├─saslauthd───4*[saslauthd]
     ├─smbd───2*[smbd]
     ├─squid3───unlinkd
     ├─sshd───sshd───sshd───bash───pstree
     ├─udevd───2*[udevd]
     ├─upstart-file-br
     ├─upstart-socket-
     ├─upstart-udev-br
     ├─xrdp
     └─xrdp-sesman
```

My shell!

Process Life-cycle

- Birth
  - From another process
    - Either a CLI or a GUI
  - A process always starts as a *clone* of its parent process
  - Then the process *upgrades itself* to running a different executable
  - Process *retains access* to the files open by the parent
- Life
  - Process can create children processes
- Death
  - Eventually calls *exit* or *abort* to “suicide”

Birth via Cloning

- A very simple API to do that

```
#include <unistd.h>

pid_t  fork(void);
```

- Child is an exact copy of the parent
- Semantics
  - In the **parent** process:
    - fork returns the process identifier of the child
    - If a failure occurred, it returns -1 (and sets errno)
  - In the **child** process: fork returns 0 (zero)

Cloning effect on resources

- All the files that were open in the parent….
  - Are accessible and shared in the child!
  - Any operation in parent or child moves the file pointer
- In particular
  - the standard file (in/out/err) are accessible in the child

Cloning effect on address space

- The parent and the cloned child
  - Are virtually indistinguishable.
  - All memory is 100% identical.
  - But are distinct copies.
  - Any memory change (stack/heap/static) only affect the caller
  - Thus the parent and his clone can quickly *diverge*

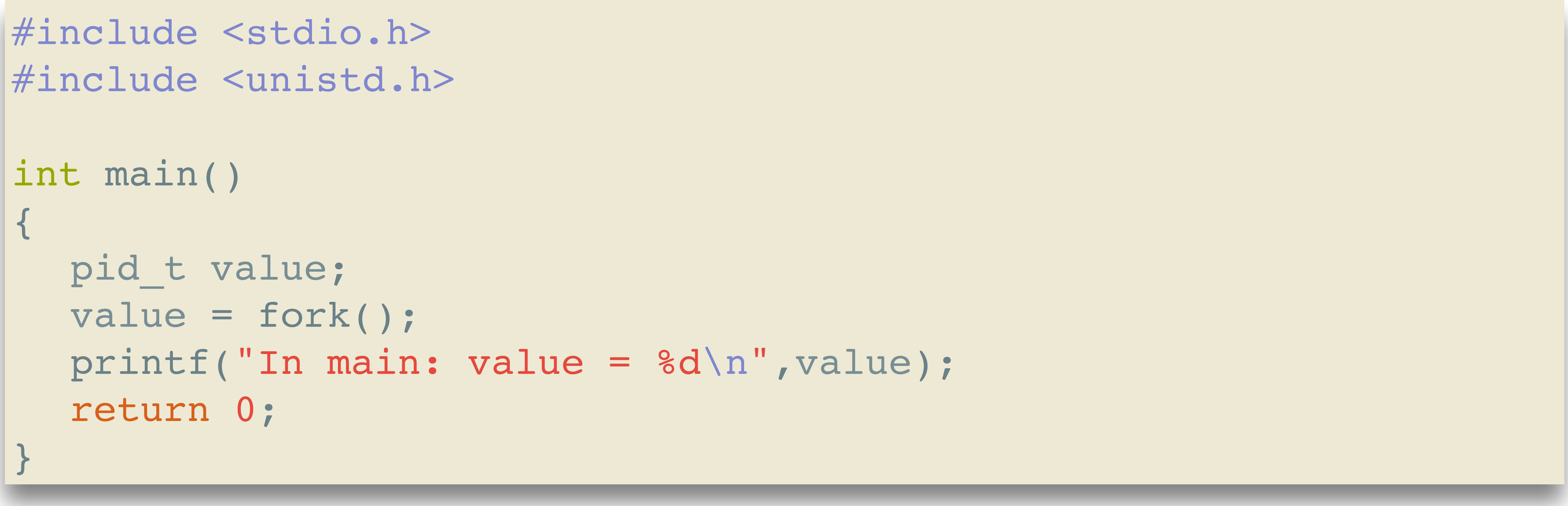
Concurrency

- The parent and the child *both* return from fork
  - This happens concurrently
  - Both can run at the same time on a multicore machine
  - You **cannot** assume as to who “returns first”
  - That is true even on a uni-core. [order chosen by OS]

Usage

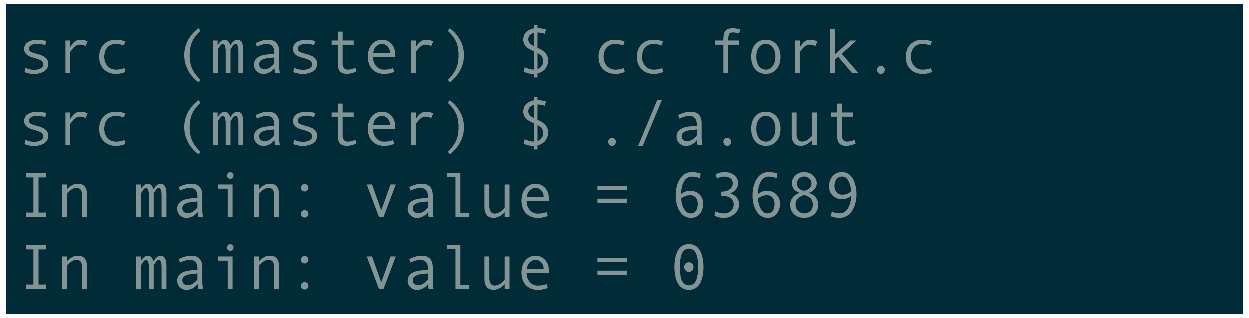
- Typically
  - The parent forks
  - When the fork returns, test the return value
    - If zero: We are the child!
    - If > 0: We are the parent and a child is/will be born!
    - If < 0: We are the parent and the cloning failed [memory?]
  - Branch based on the return value to decide what to do next.

Example



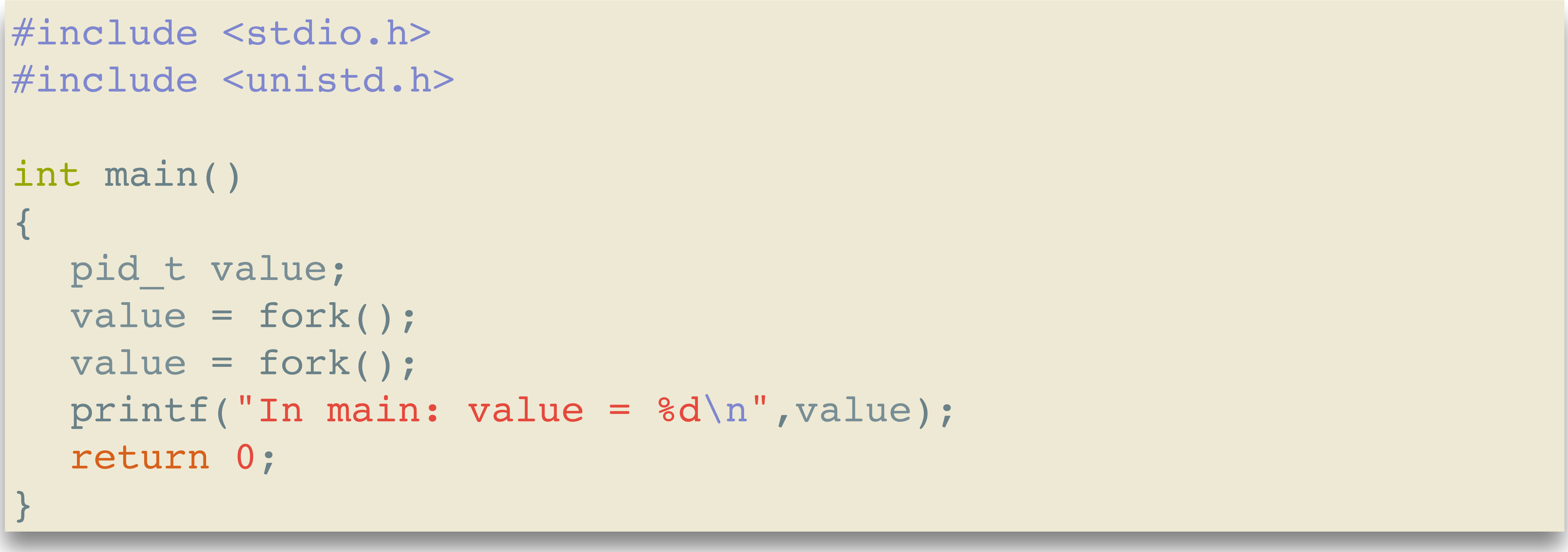
```c
#include <stdio.h>
#include <unistd.h>

int main()
{
   pid_t value;
   value = fork();
   printf("In main: value = %d\n",value);
   return 0;
}
```



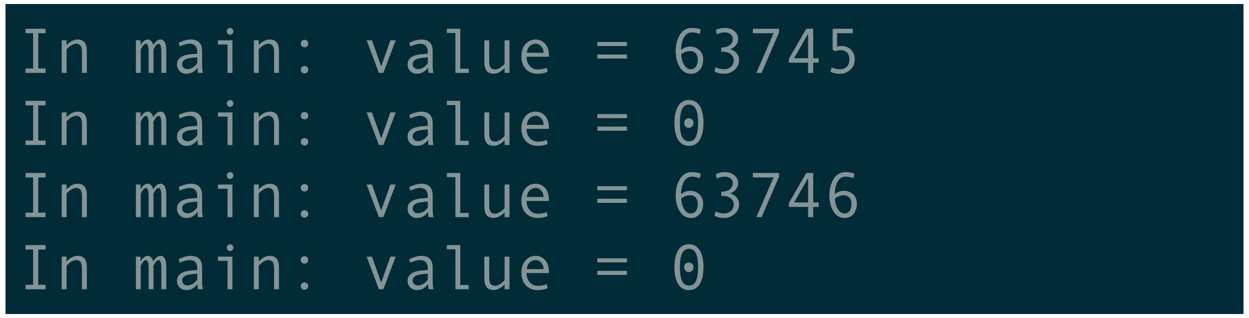
```
src (master) $ cc fork.c
src (master) $ ./a.out
In main: value = 63689
In main: value = 0
```

Example



```c
#include <stdio.h>
#include <unistd.h>

int main()
{
   pid_t value;
   value = fork();
   value = fork();
   printf("In main: value = %d\n",value);
   return 0;
}
```



```
In main: value = 63745
In main: value = 0
In main: value = 63746
In main: value = 0
```

What should the parent do ?

- Depends on application!
  - It could run concurrently and ignore the child
  - It could run concurrently and check back on the child later
  - It could wait until the child is done (dies!)

Example

```c
#include <stdio.h>
#include <unistd.h>
#include <time.h>
int fib(int n) {
   if (n<= 1)
      return n;
   else return fib(n-1) + fib(n-2);
}
int main() {
   pid_t value = fork();
   int i;
   if (value == 0) {
      for(i=0;i < 30;i++)
         printf("fib(%2d) = %d\n",i*5,fib(i*5));
   } else {
      long begin = time(NULL);
      for(i=0;i < 10;i++) {
         sleep(1);
         printf("Elapsed time in parent: %ld\n",time(NULL)-begin);
      }
   }
   return 0;
}
```

What happens ?

- Parent forks
  - On return from fork: test pid value
    - ZERO This is the child, compute a bunch of fib values.
    - >0 This is the parent, sleep in 1s increment
      Report time at each wakeup
      Then exit
- When the parent dies…
  - The child lives on until it finishes its loop
  - The child has been ‘adopted’ by an ancestor (typically, init!)

Waiting on a child

- Useful when the child has a task to do
  - Typical of a shell like bash/ksh/zsh/csh/….
- Two simple APIs to do that

```
 #include <sys/wait.h>

pid_t   wait(int *stat_loc);
pid_t   waitpid(pid_t pid, int *stat_loc, int options);
```

- Purpose
  - Block the calling process until a child dies
  - Reports in *stat_loc
    - the cause of death (check: man wait)
    - the exit status of the child (what he returned from main)
  - return value identifies the dead child

Please note!

- With Operating Systems
  - The child usually dies before the parent
  - The parent attends to the dead processes by waiting on them
- Collecting the dead children…
  - Is called “reaping” and is the responsibility of the parent process

Reaper Dreadnought

Reaping

- When the parent is alive
  - Parent attends to the dead processes by waiting on them
- When the parent is dead
  - Two possibilities
    - Children are adopted by an ancestor process and live on.
      - That’s the default.
      - Adopted by “init” (the first process)
    - Children all die in solidarity
      - Precisely: the children are murdered by the dying parent
      - API: `kill(pid_t pid,int sig)`

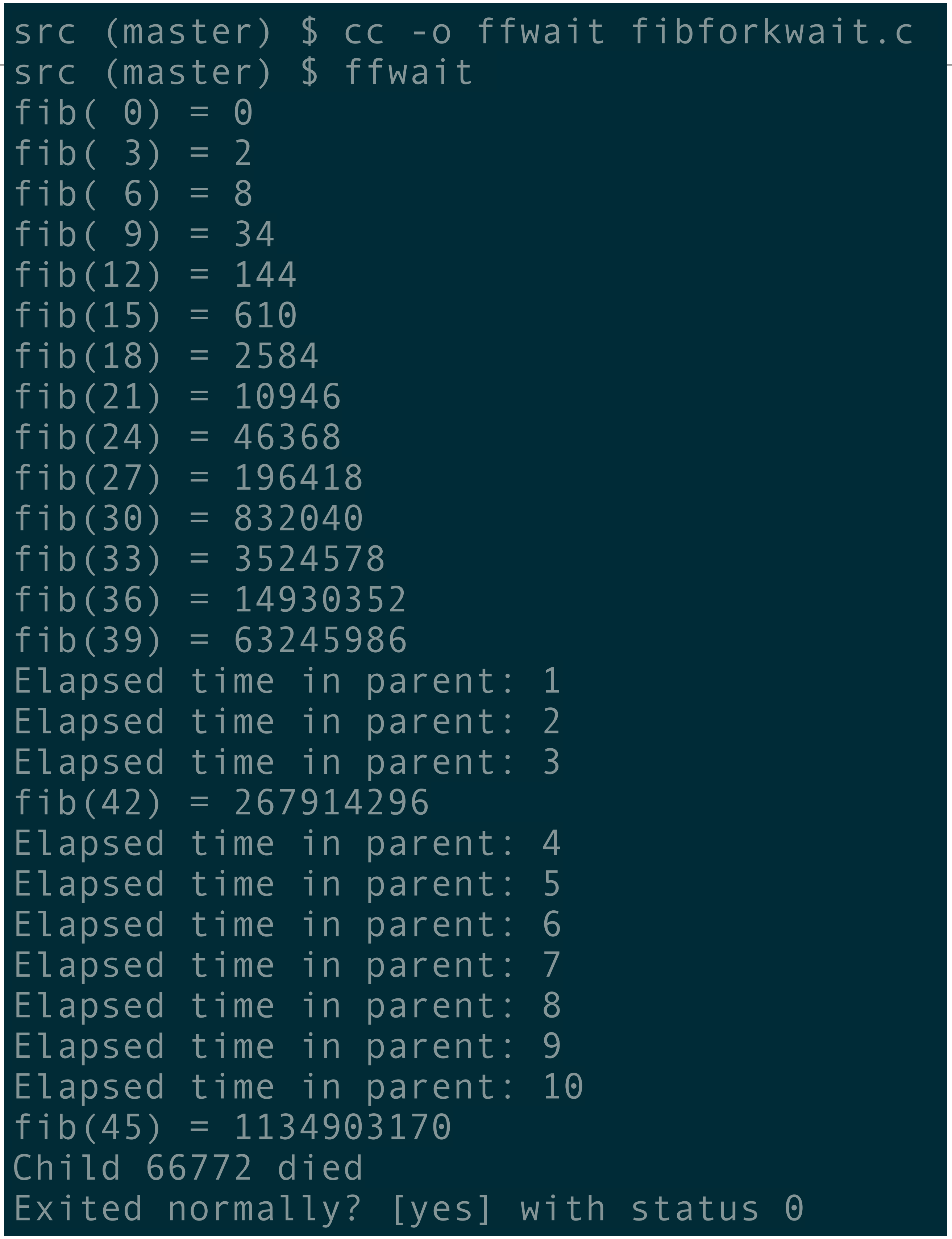
How this is done…

- Terminating the children ?
  - Loop over child processes and send them kill signals
- Letting init adopt children ?
  - Do nothing, that’s the default
- Becoming a (sub)-“reaper” ?
  - A descendant of init can become a Reaper
  - Simply call prctl(PR_SET_CHILD_SUBREAPER,1);
  - (man prctl for details)

Example

```c
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include <sys/wait.h>
int fib(int n)  {
    if (n<= 1)
        return n;
    else return fib(n-1) + fib(n-2);
}
int main() {
    pid_t value = fork();
    int i;
    if (value == 0) {
        for(i=0;i < 16;i++)
            printf("fib(%2d) = %d\n",i*3,fib(i*3));
    } else {
        long begin = time(NULL);
        for(i=0;i < 10;i++) {
            sleep(1);
            printf("Elapsed time in parent: %ld\n",time(NULL)-begin);
        }
        int exitStatus;
        pid_t deadChild = wait(&exitStatus);
        printf("Child %d died\n",deadChild);
        printf("Exited normally? [%s] with status %d\n",
               WIFEXITED(exitStatus) ? "yes" : "no",
               WEXITSTATUS(exitStatus));
    }
    return 0;
}
```

Trace



```
src (master) $ cc -o ffwait fibforkwait.c
src (master) $ ffwait
fib( 0) = 0
fib( 3) = 2
fib( 6) = 8
fib( 9) = 34
fib(12) = 144
fib(15) = 610
fib(18) = 2584
fib(21) = 10946
fib(24) = 46368
fib(27) = 196418
fib(30) = 832040
fib(33) = 3524578
fib(36) = 14930352
fib(39) = 63245986
Elapsed time in parent: 1
Elapsed time in parent: 2
Elapsed time in parent: 3
fib(42) = 267914296
Elapsed time in parent: 4
Elapsed time in parent: 5
Elapsed time in parent: 6
Elapsed time in parent: 7
Elapsed time in parent: 8
Elapsed time in parent: 9
Elapsed time in parent: 10
fib(45) = 1134903170
Child 66772 died
Exited normally? [yes] with status 0
```

Process Upgrades

- Usually….
  - A fresh clone wants to run *different code*
- This can be easily done
  - Child “guts himself”…
    - [meaning it discards the code in its address space]
  - And reloads the process address space with another executable
    - [picked up from the file system of course]
- Note
  - Opened files are *NOT AFFECTED* by the operation.

The `exec` family

- The act of ‘gutting and upgrading’ is done by the child with a system call
    - There is a whole family (variants) of system calls

```
#include <unistd.h>

extern char **environ;

int execl(const char *path,
          const char *arg0, ...
         /*, (char *)0 */);
```

- Check
    - man -S3 execl for variants

Fundamentally

- The call takes as input
  - The path to the executable to load inside our own address space
  - A list of arguments to be passed to the new executable
  - A final NULL pointer to give the “end of argument list”
- Note
  - If successful `execl` does not return!
  - Instead, control is transferred to the main function of the new executable!

Exec example

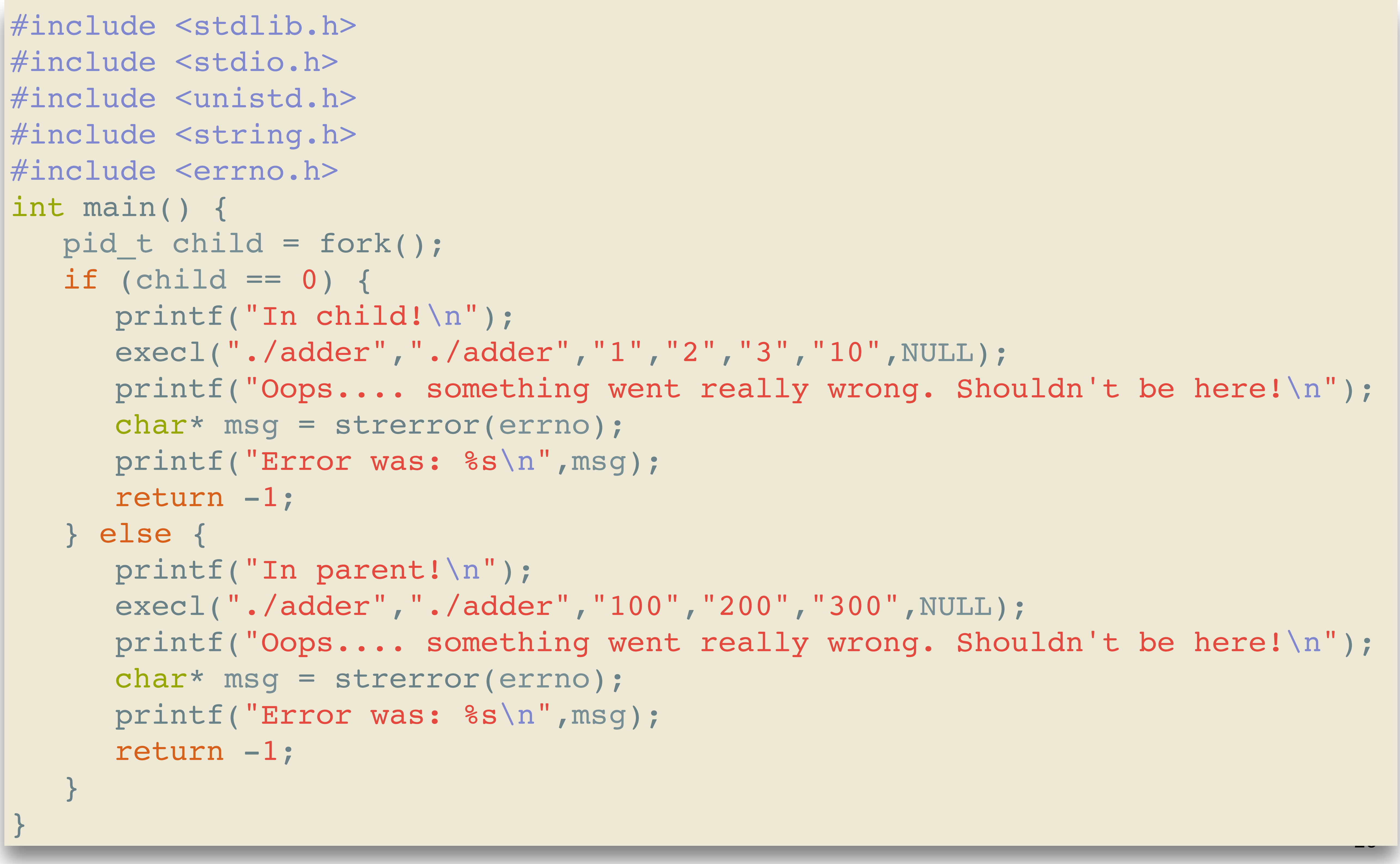
First a child executable

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char* argv[]) {
   int i,value,sum=0;
   for(i=1;i<argc;i++)
      sum += atoi(argv[i]);
   printf("sum is: %d\n",sum);
   return 0;
}
```

This is a simple “adder” program that computes the sum of its integer arguments

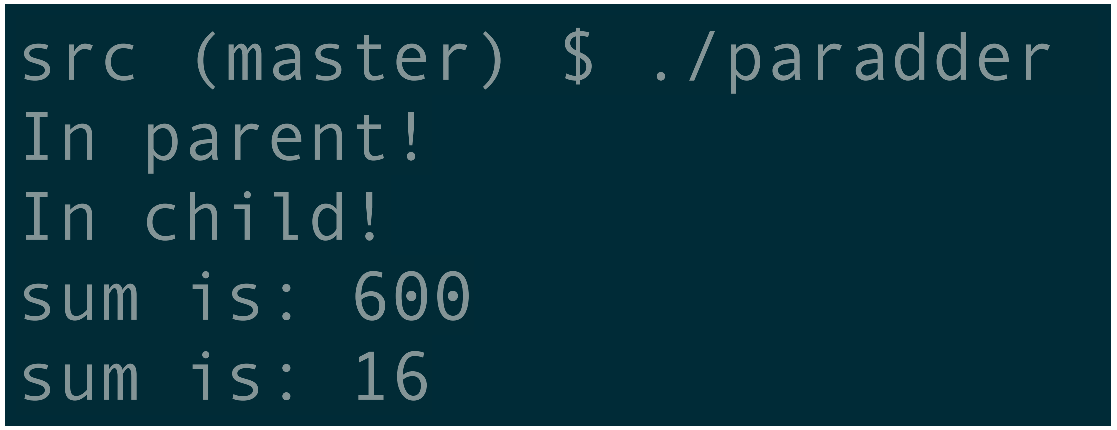
Parent Program



```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
int main() {
   pid_t child = fork();
   if (child == 0) {
      printf("In child!\n");
      execl("./adder","./adder","1","2","3","10",NULL);
      printf("Oops.... something went really wrong. Shouldn't be here!\n");
      char* msg = strerror(errno);
      printf("Error was: %s\n",msg);
      return -1;
   } else {
      printf("In parent!\n");
      execl("./adder","./adder","100","200","300",NULL);
      printf("Oops.... something went really wrong. Shouldn't be here!\n");
      char* msg = strerror(errno);
      printf("Error was: %s\n",msg);
      return -1;
   }
}
```

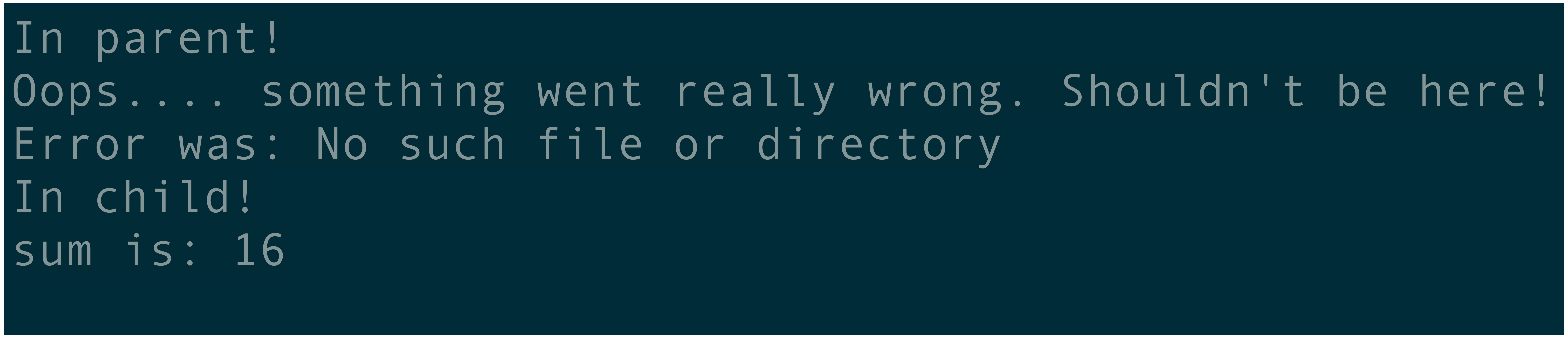
Behavior

- Quite straightforward



```
src (master) $ ./paradder
In parent!
In child!
sum is: 600
sum is: 16
```

- Imagine typo in name of executable



```
In parent!
Oops.... something went really wrong. Shouldn't be here!
Error was: No such file or directory
In child!
sum is: 16
```

FORK / EXEC

AND

INPUT/OUTPUT IMPACTS

(or what happens to file descriptors)
A SysProg Saga

Inheritance

- Whenever a parent *forks….*
  - The child inherits all the parent's file descriptor….
  - Ilustration!

Playing with redirections

- Note
  - A child process inherits any files opened by the parent
- Corollary
  - This can be handy to change where the child…
    - Reads its input from
    - Write its output to!
- How?
  - Simply *change* the files corresponding to
    - stdin (0) | stdout (1) | stderr (2)

Shell redirections

- That’s the embryo of piping
  - Available in your shell

```
$  sort < file.txt   > sorted.txt
```

  - Simple syntax
    - **<**   filename   : Take input from the file *filename*
    - **>**   filename   : Send output to the file *filename*
    - **2>** filename   : Send errors to the file *filename*
  - Implemented with the close/open/dup technique

Demo
Time!

Brief aside

- Remember the IO APIs
  - The “f” family (fopen,fclose, fread, fgetc, fscanf, fprintf,…)
  - All these use a FILE* abstraction to represent a file
  - All these rely on buffering in the C library
  - Support line-ending translation when opening in “text mode”
- UNIX has a lower-level API for file handling
  - Directly mapped to system calls
  - No buffering
  - Raw IO
  - Uses OS level *file descriptors* **[these are just integers]**

Brief aside

- APIs highlight (only some of them)

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

int open(const char *path, int oflag, …);
int close(int fildes);

ssize_t read(int fildes, void *buf, size_t nbyte);
ssize_t write(int fildes, const void *buf, size_t nbyte);

int dup(int fildes);
int dup2(int fildes, int fildes2);
```

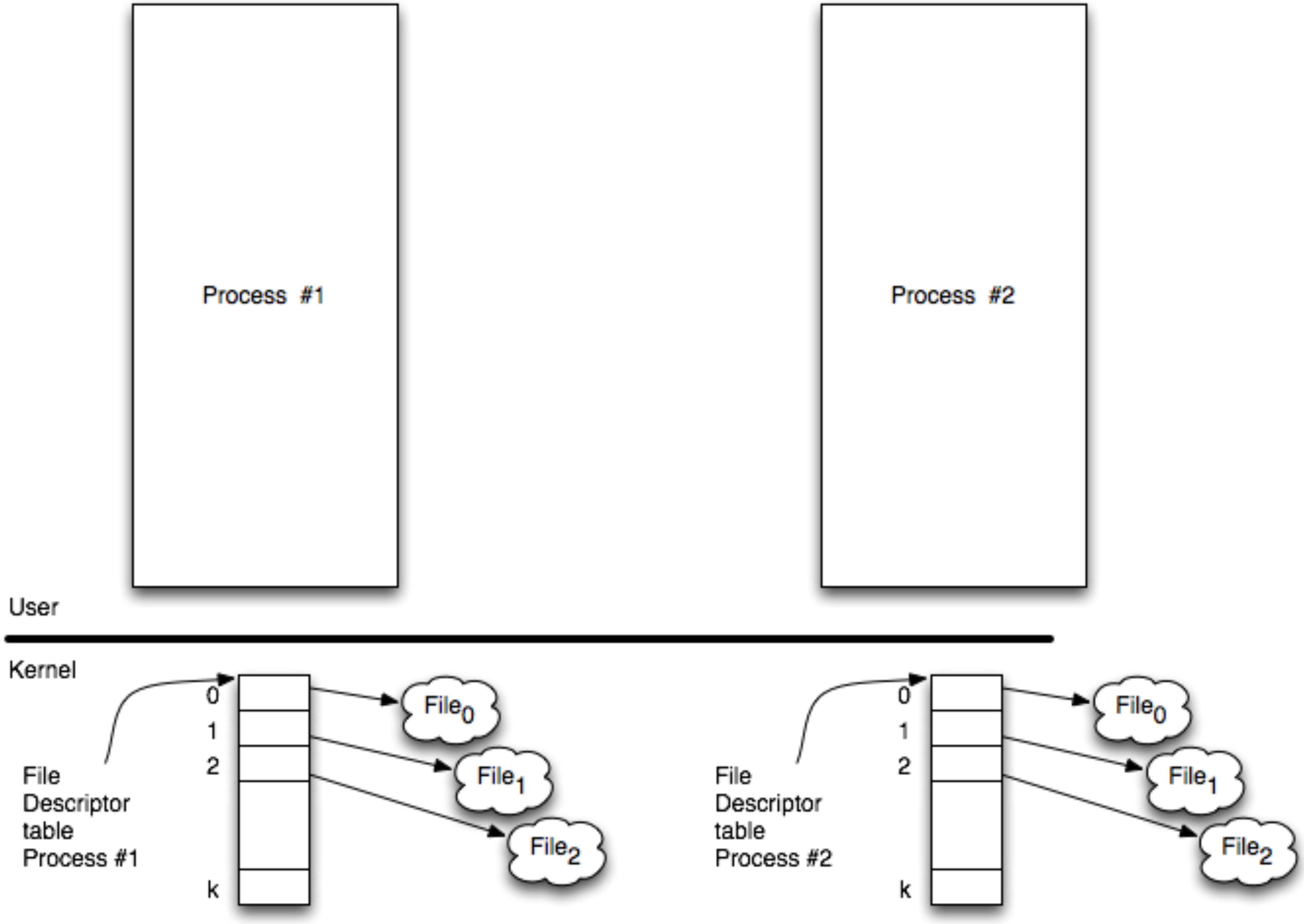
“New” APIs

Brief aside

- The API is perfectly fine for
  - Binary file
  - And setting up redirections ;-)
- Two new “special” APIs
  - dup
  - dup2
- Purpose
  - They duplicate file handles

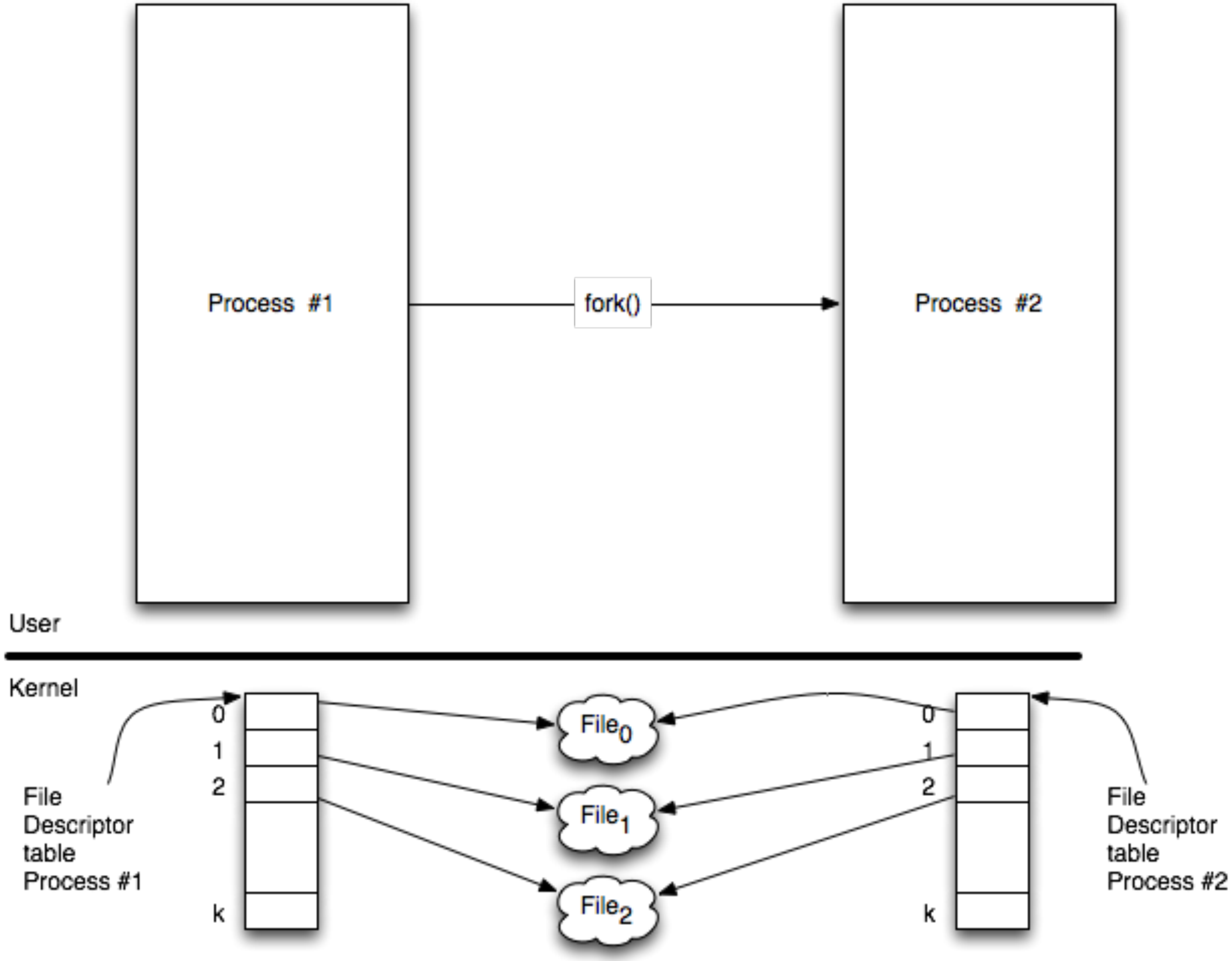
Brief aside (picture!)

Two processes in general



Brief aside (picture!)

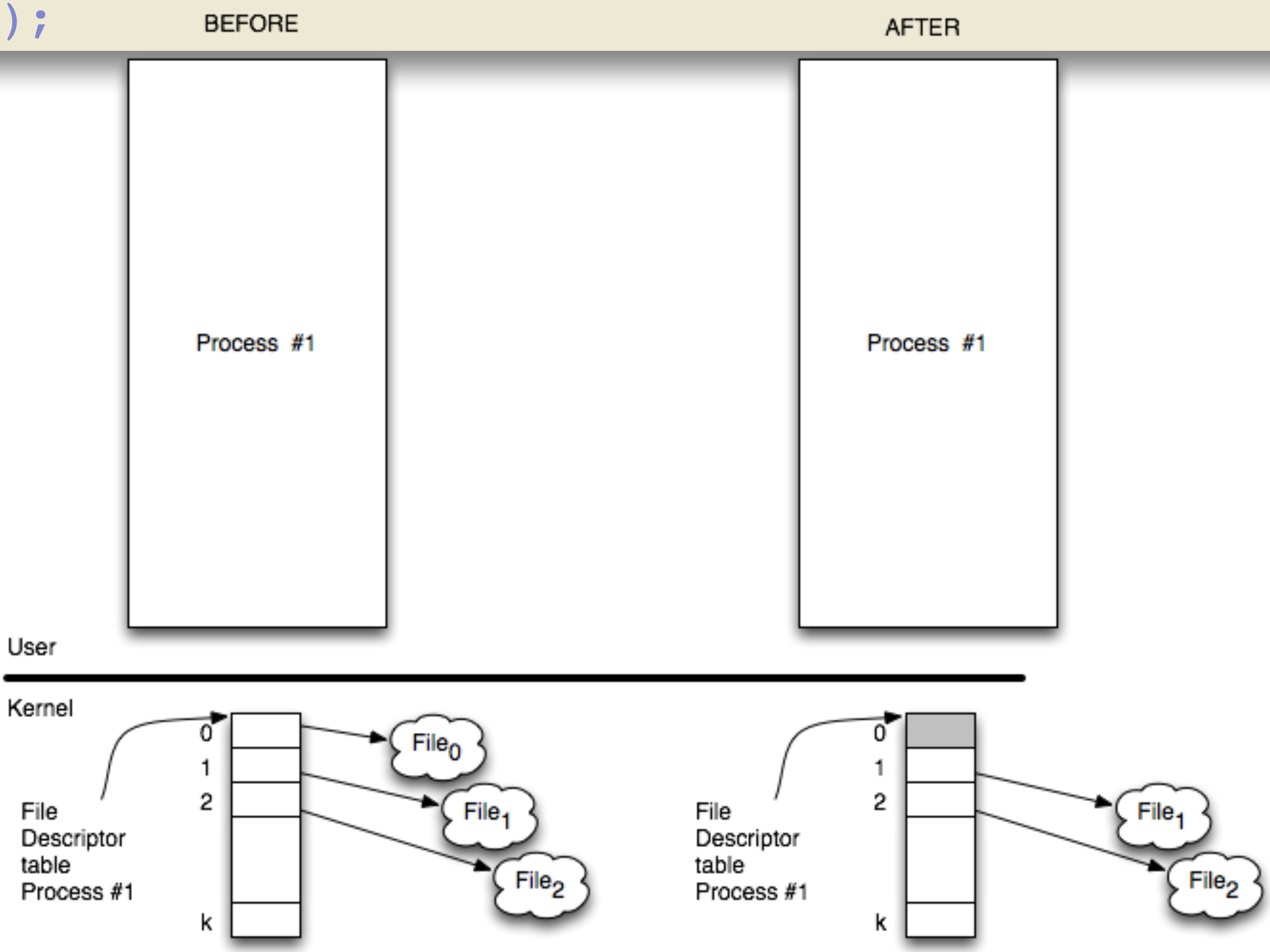
Two processes, process #2 is a fork of #1



What close does

- Consider

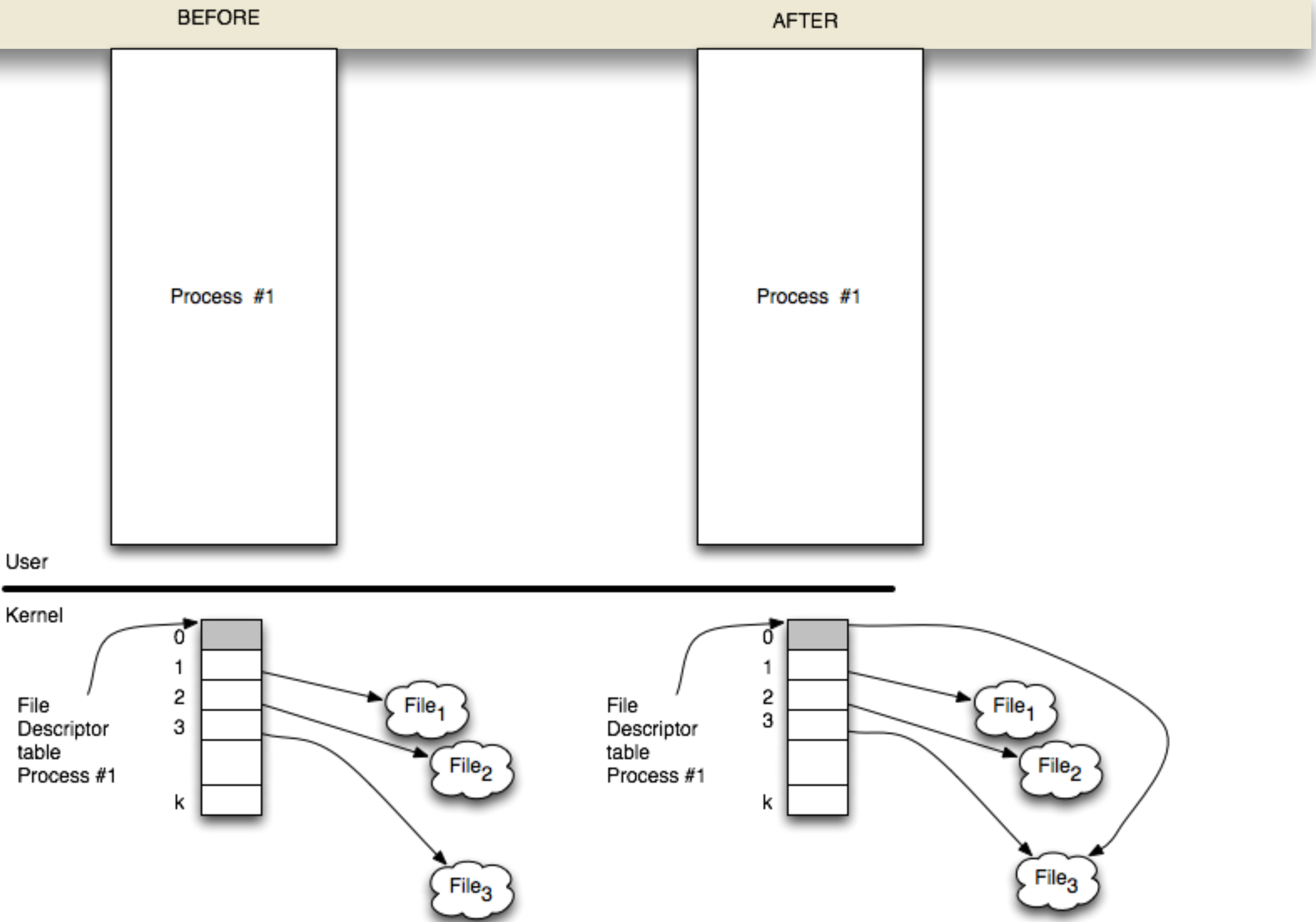
```
close(0);
```



What dup does

- Duplicate a file handle!

```
dup(3);
```



Bottom line

- Purpose of dup
  - Ensure that specific entries of the FDT point to the “correct” file
- Remember

• STDIN	0
• STDOUT	1
• STDERR	2

Key questions

- Who should call dup?
  - The parent?
  - The child?
- When should dup be called ?
  - before execl ?
  - after execl ?

What if?

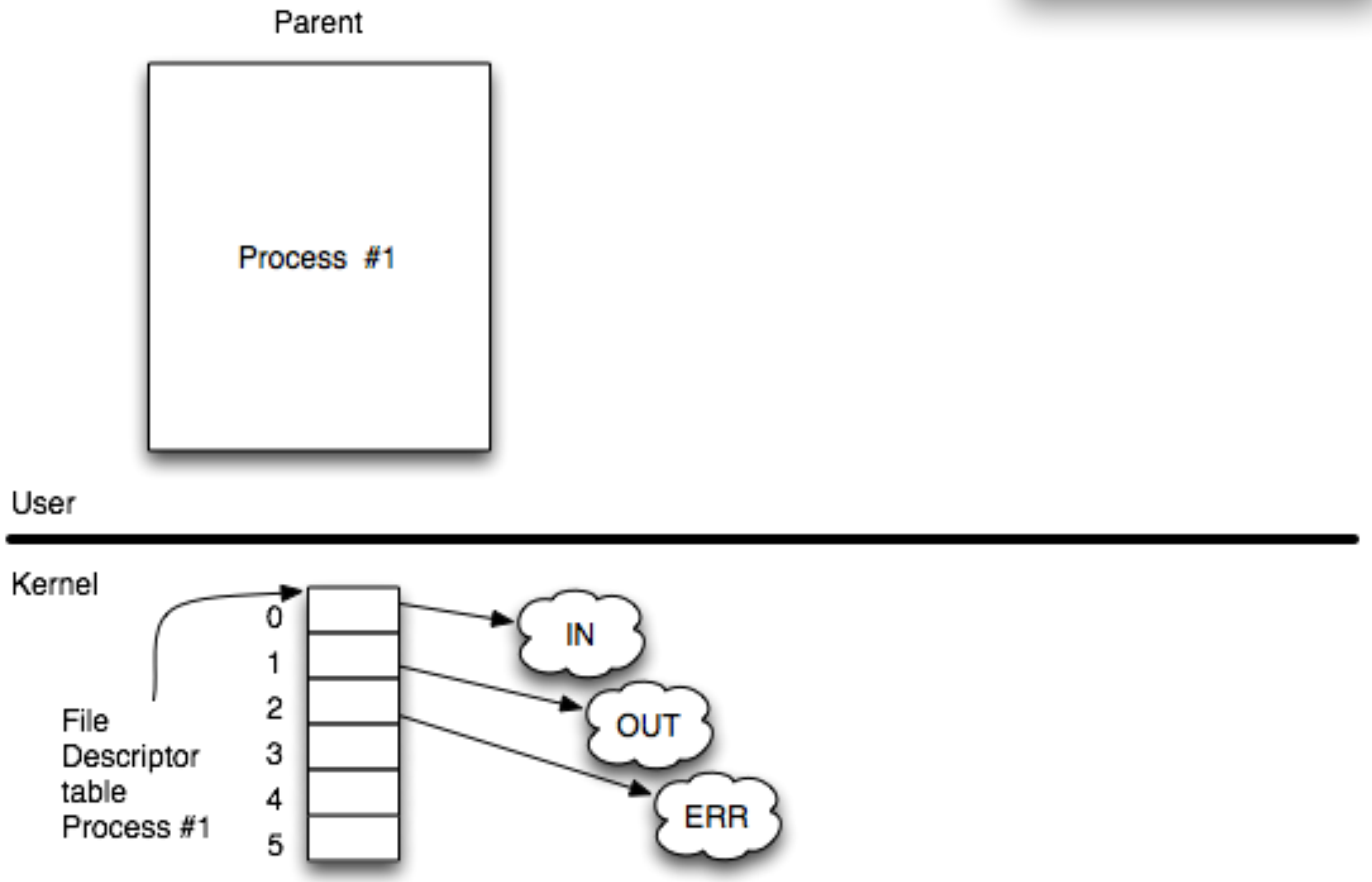
- You call close(0) before forking….
  - Then the child will inherit a closed file as STDIN
  - But the parent *also* looses its STDIN!
- You call close(0) after forking….
  - Then the child inherited the parent’s STDIN
  - The parent’s STDIN is unaffected
  - The clone can close its own and reopen the right file before execl
- Can you call close(0) after execl ?
  - No! That’s too late!

Pipe Demo

Time!

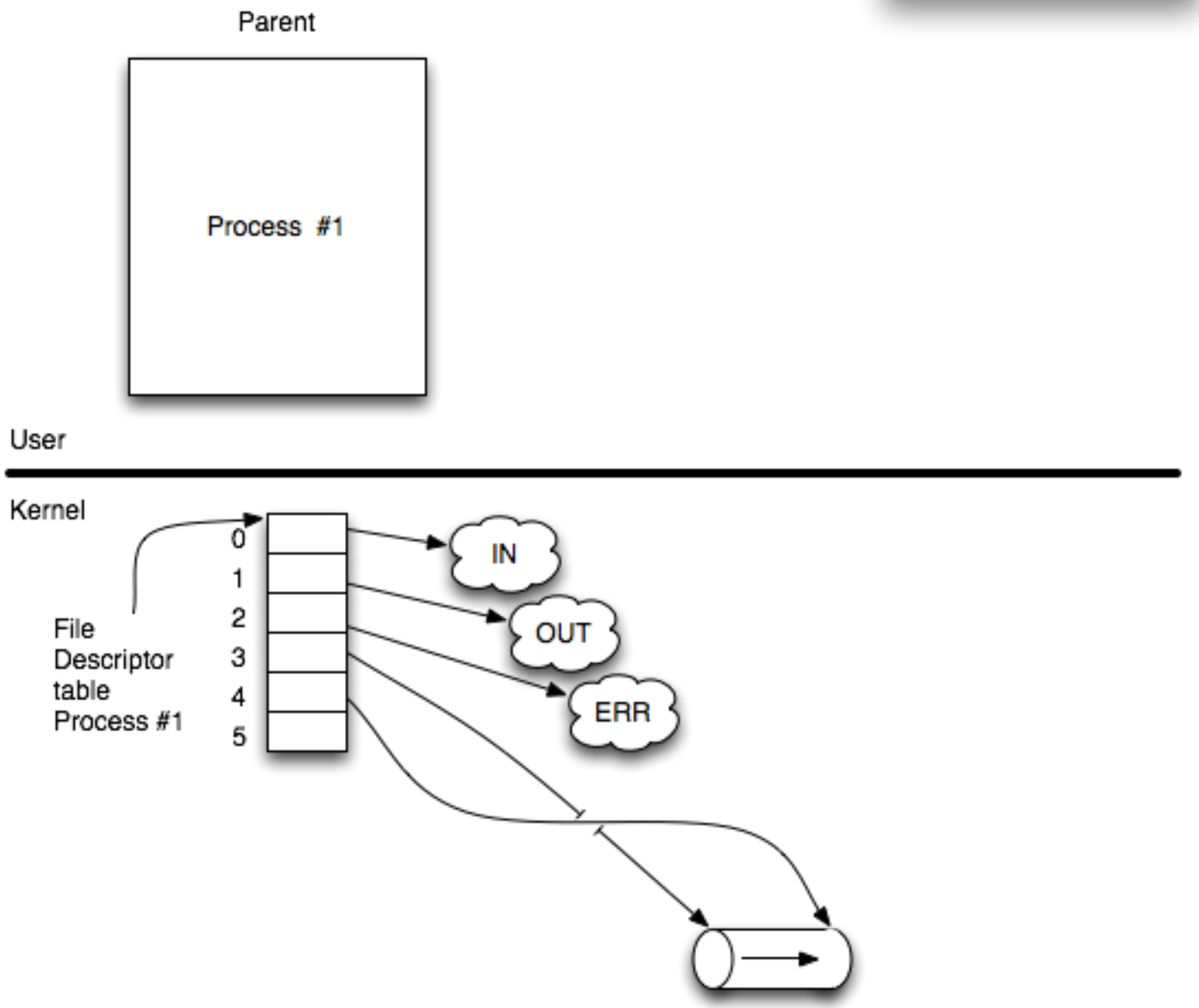
Pipes

```
$ A | B
```


Parent
Process #1
User
Kernel
File
Descriptor
table
Process #1
0
1
2
3
4
5
IN
OUT
ERR

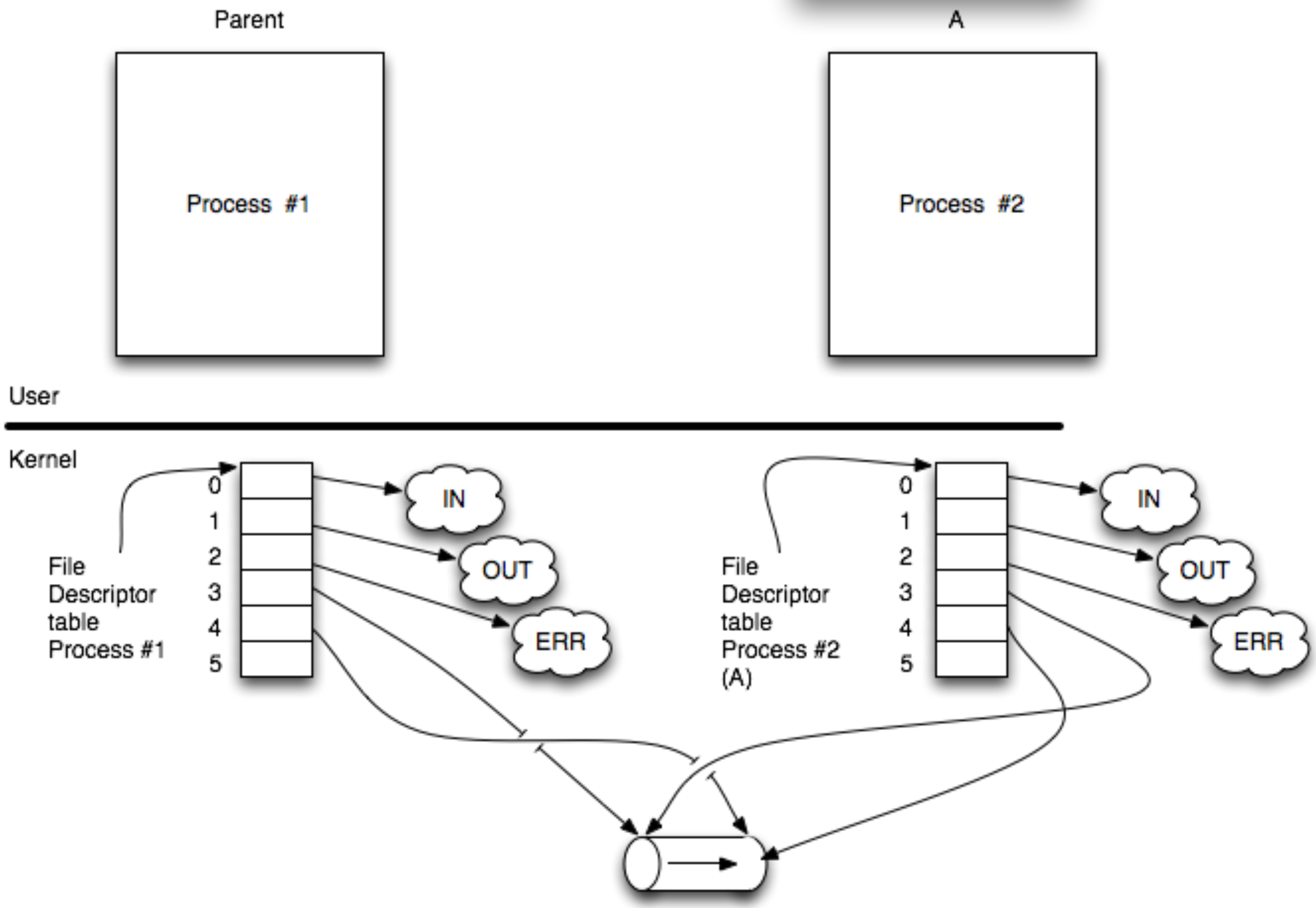
Pipes [creation]

```
$ A | B
```



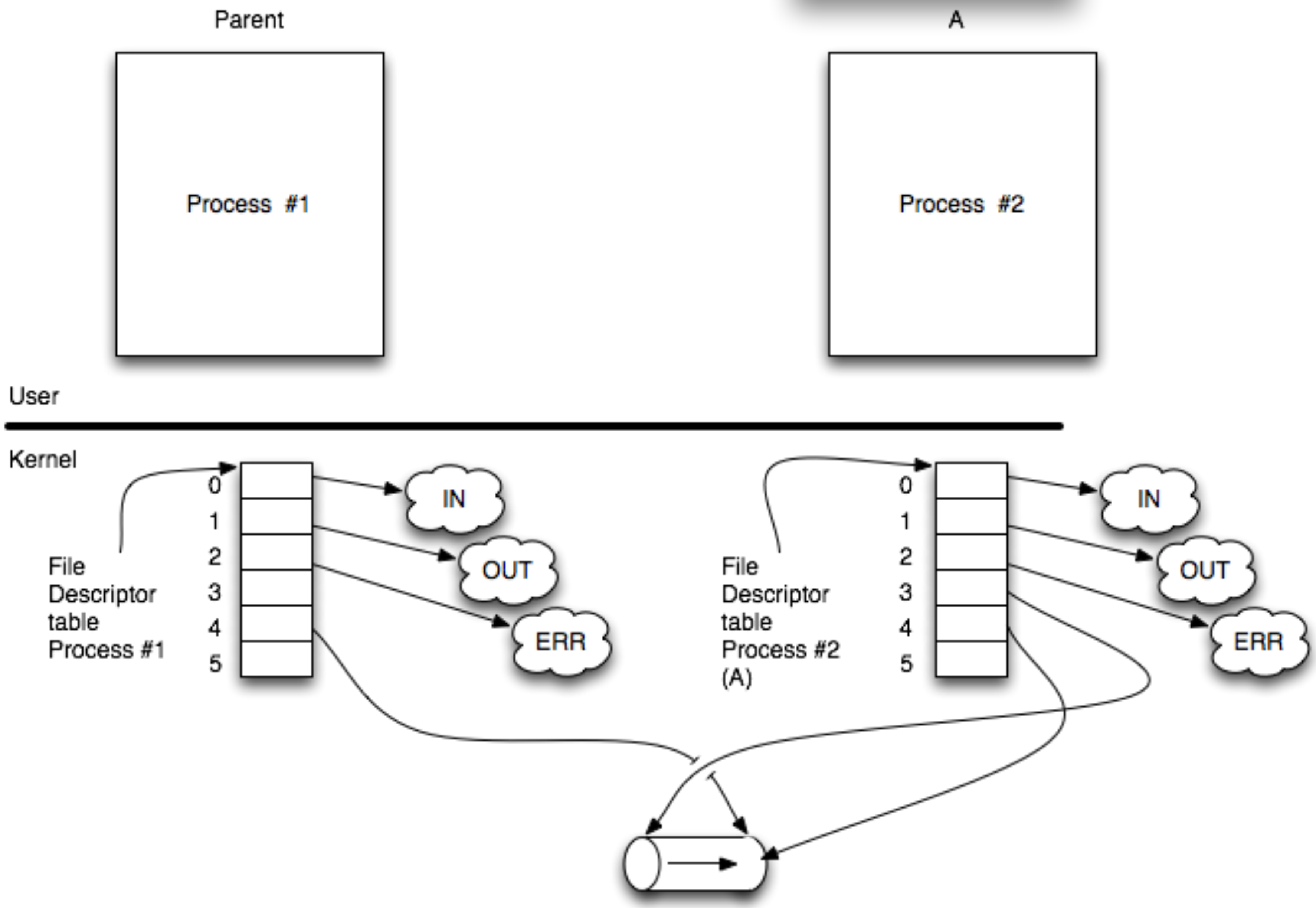
Pipes [forking]

```
$ A | B
```



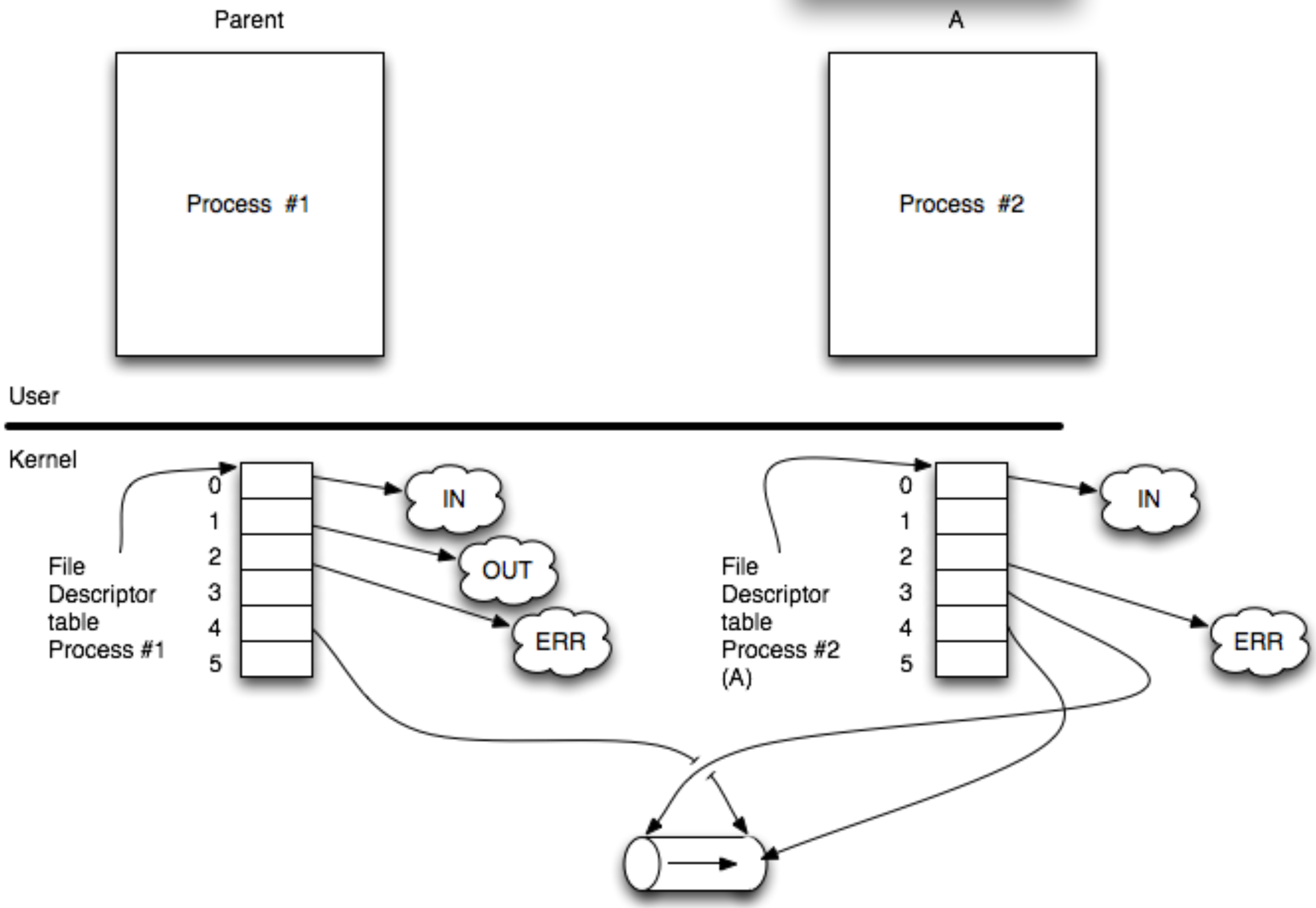
Pipes [cleaning part 1]

```
$ A | B
```


Parent
Process #1
A
Process #2
User
Kernel
File Descriptor table Process #1
0
1
2
3
4
5
IN
OUT
ERR
File Descriptor table Process #2 (A)
0
1
2
3
4
5
IN
OUT
ERR

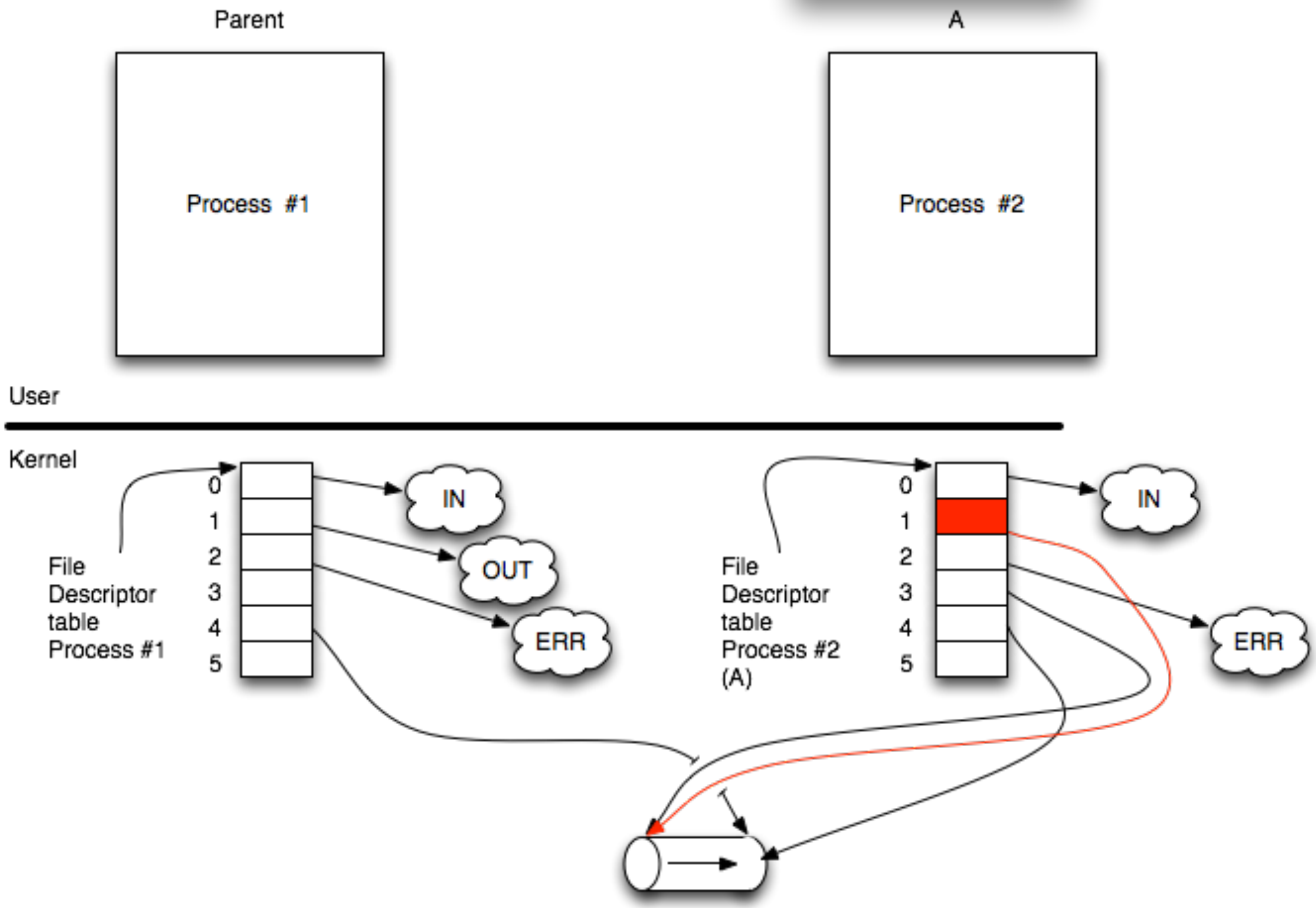
Pipes [setup A]

```
$ A | B
```


Parent
Process #1
A
Process #2
User
Kernel
0
1
2
3
4
5
File Descriptor table Process #1
IN
OUT
ERR
0
1
2
3
4
5
File Descriptor table Process #2 (A)
IN
ERR

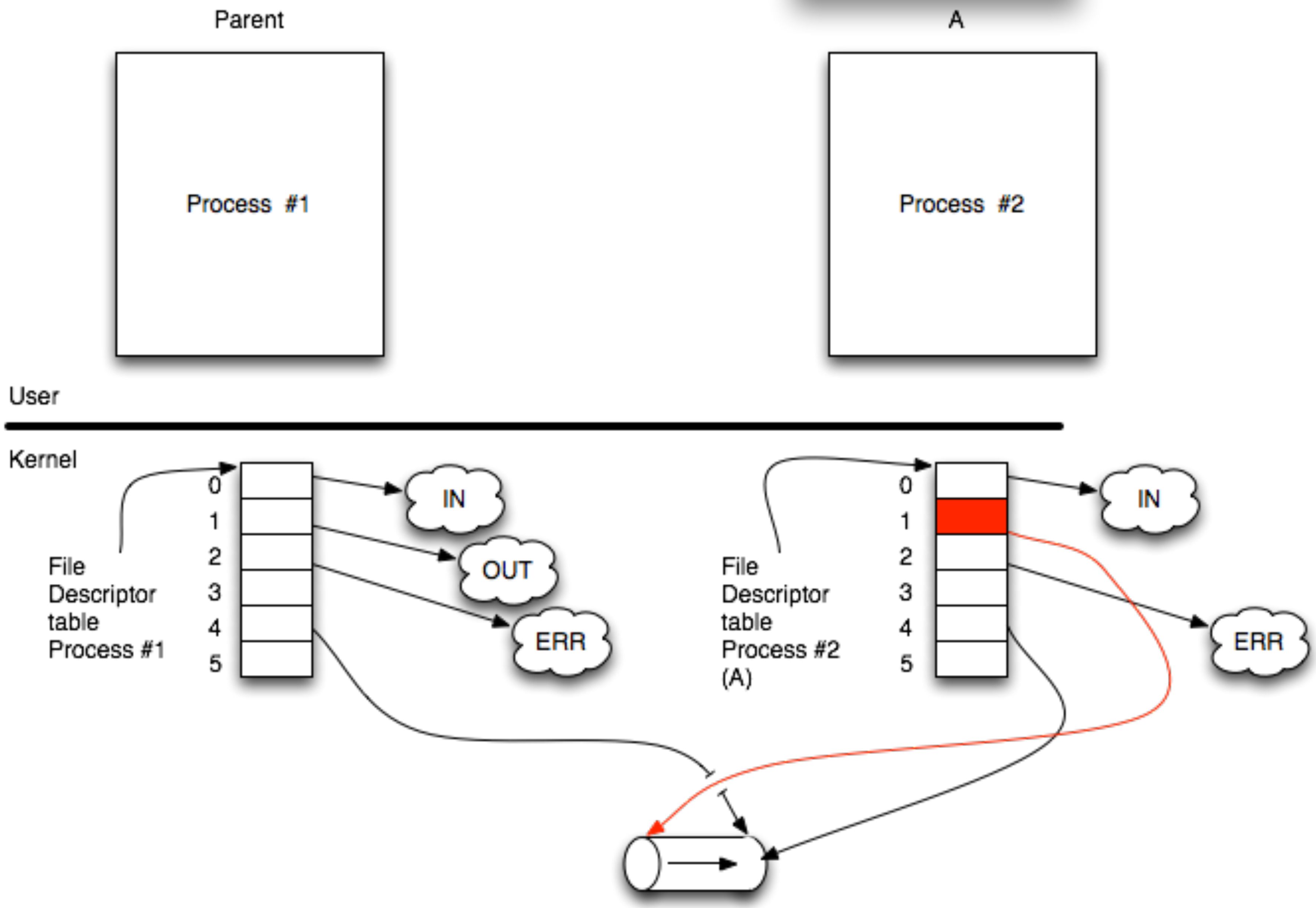
Pipes [setup A]

```
$ A | B
```


Parent
A
Process #1
Process #2
User
Kernel
0
1
2
3
4
5
File Descriptor table Process #1
IN
OUT
ERR
0
1
2
3
4
5
File Descriptor table Process #2 (A)
IN
ERR

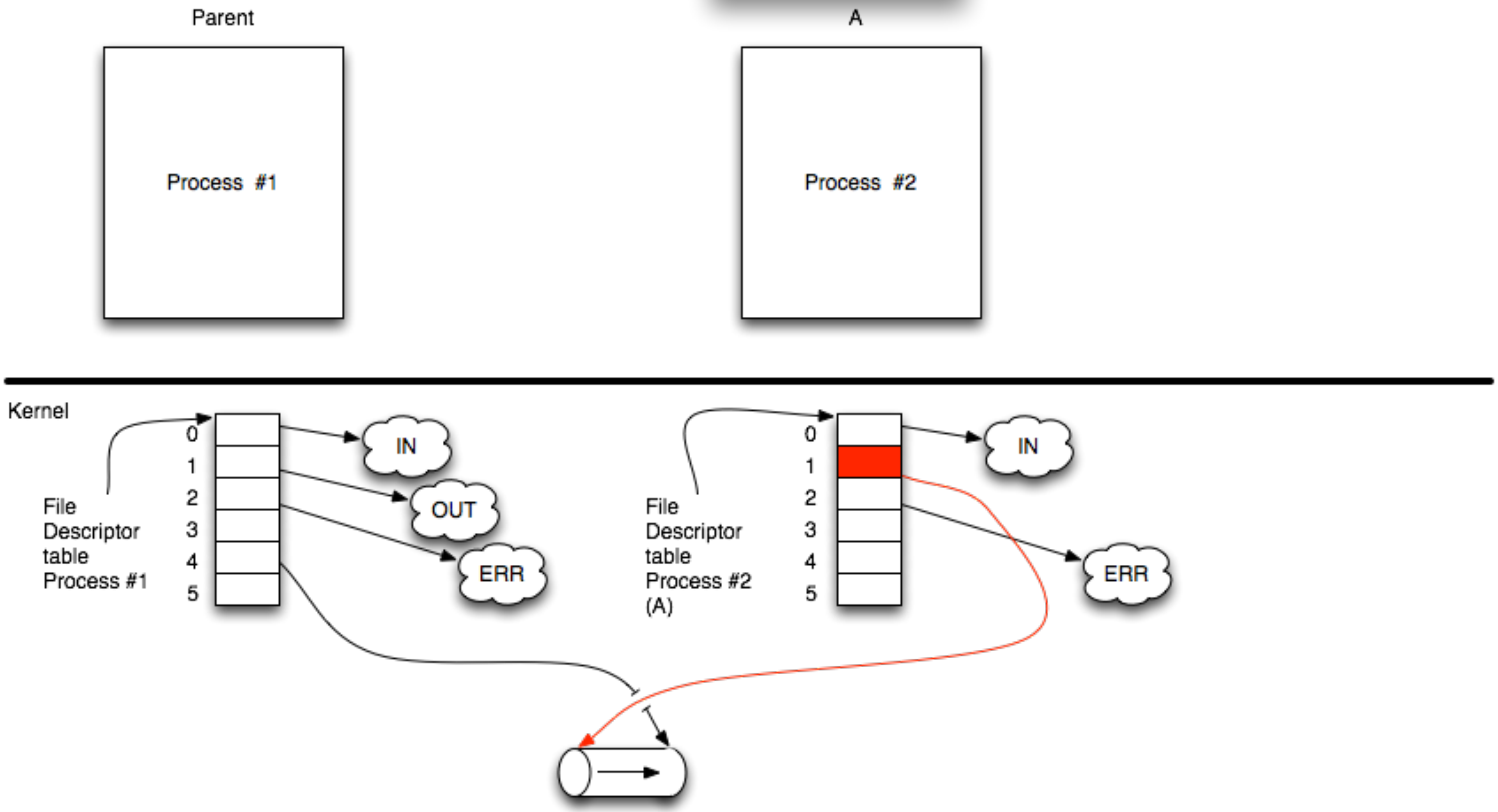
Pipes [setup A]

```
$   A | B
```


Parent
A
Process #1
Process #2
User
Kernel
0
1
2
3
4
5
IN
OUT
ERR
File Descriptor table Process #1
File Descriptor table Process #2 (A)
0
1
2
3
4
5
IN
ERR

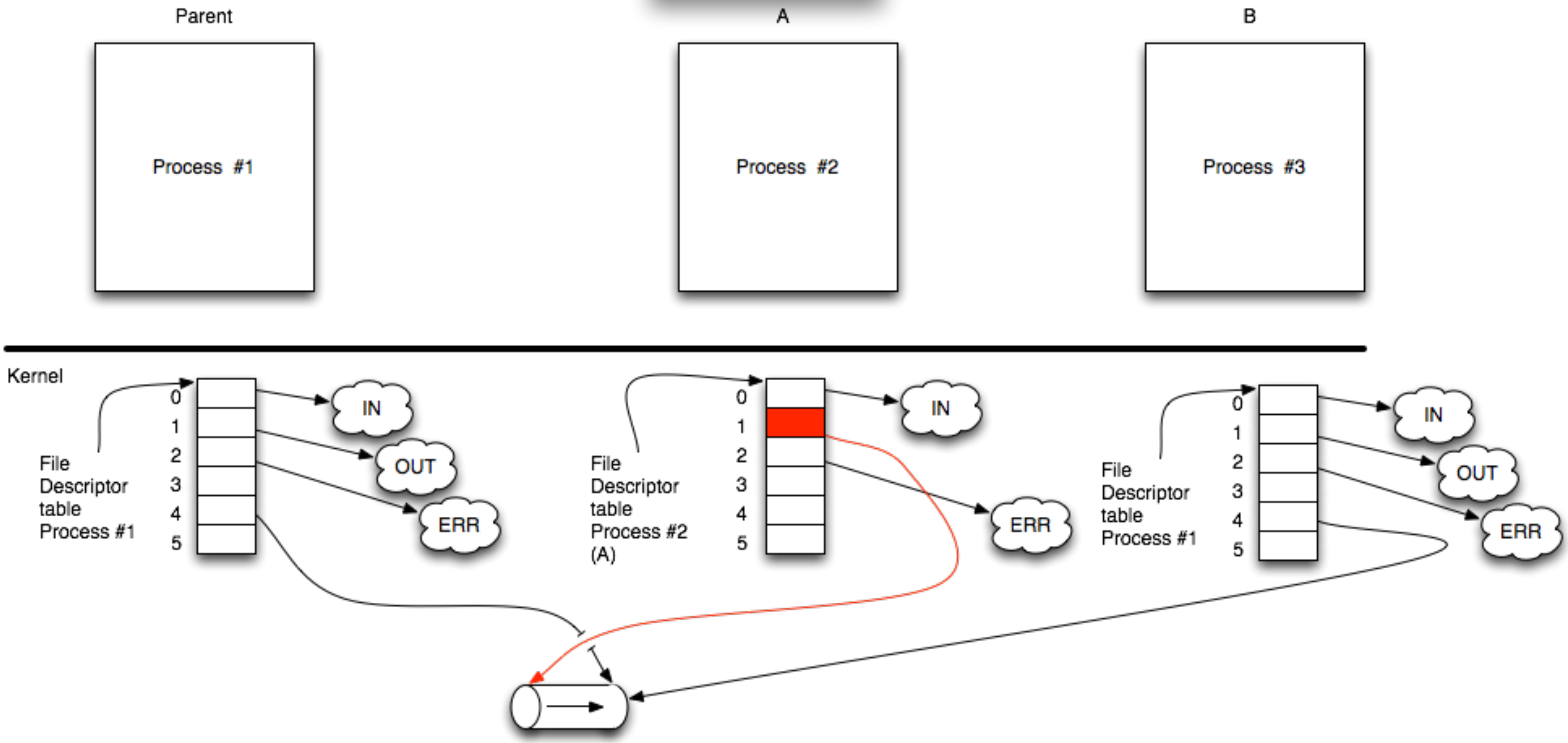
Pipes [setup A]

```
$ A | B
```


Parent
A
Process #1
Process #2
Kernel
File Descriptor table Process #1
0
1
2
3
4
5
IN
OUT
ERR
File Descriptor table Process #2 (A)
0
1
2
3
4
5
IN
ERR

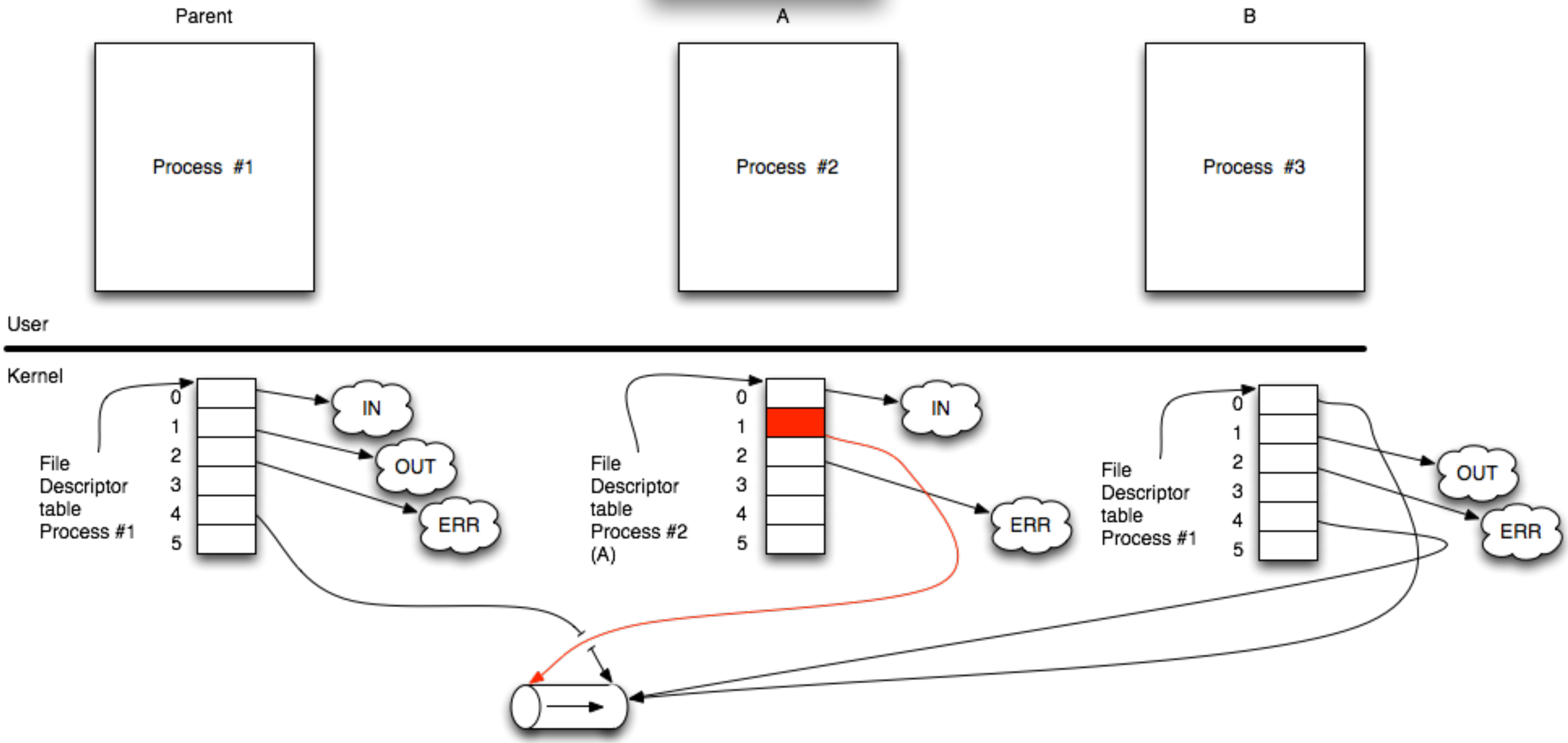
Pipes [fork B from Parent]

```
$ A | B
```


Parent
A
B
Process #1
Process #2
Process #3
Kernel
File Descriptor table Process #1
0
1
2
3
4
5
IN
OUT
ERR
File Descriptor table Process #2 (A)
0
1
2
3
4
5
IN
ERR
File Descriptor table Process #1
0
1
2
3
4
5
IN
OUT
ERR

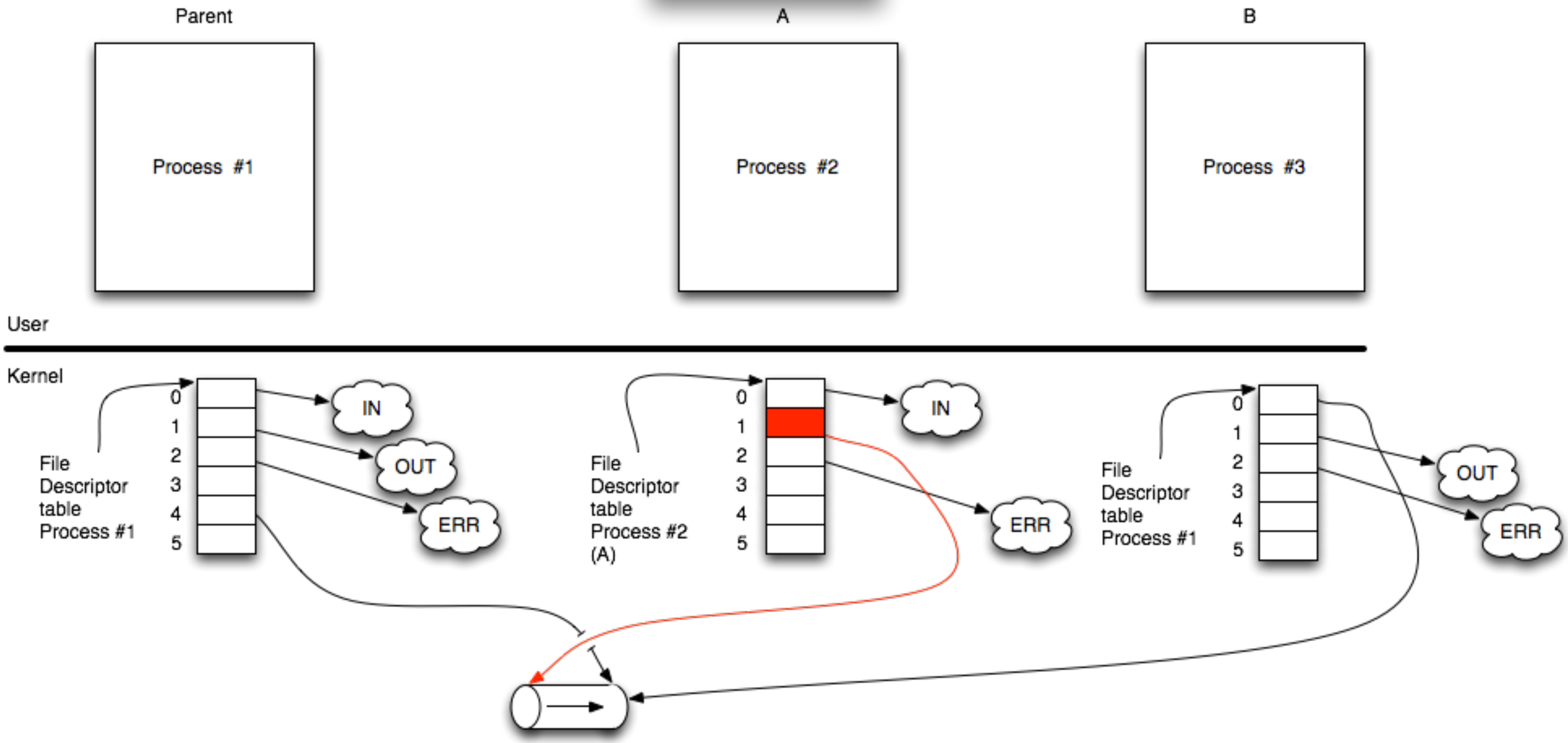
Pipes [setup in B]

```
$ A | B
```


Parent
A
B
Process #1
Process #2
Process #3
User
Kernel
File Descriptor table Process #1
0
1
2
3
4
5
IN
OUT
ERR
File Descriptor table Process #2 (A)
0
1
2
3
4
5
IN
ERR
File Descriptor table Process #1
0
1
2
3
4
5
OUT
ERR

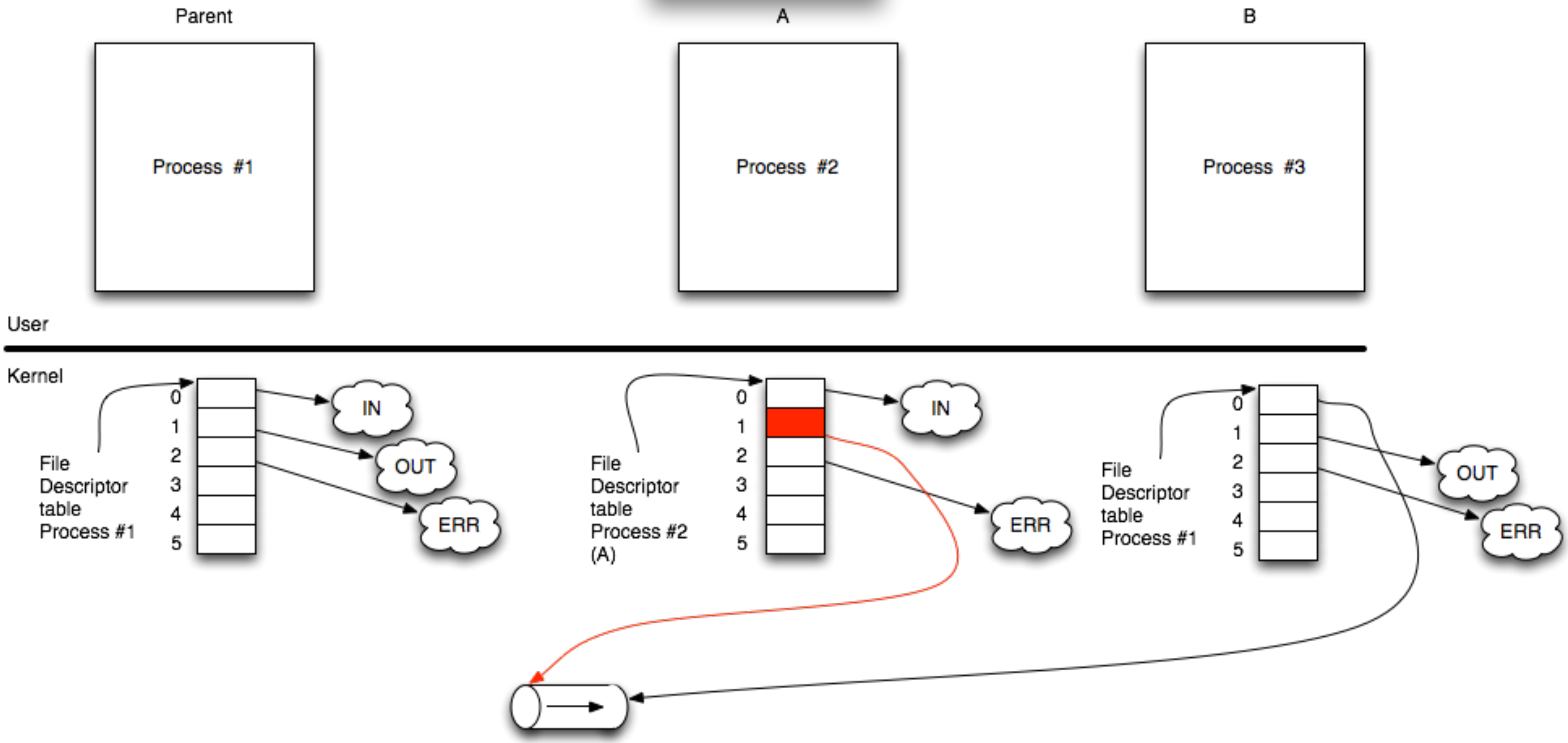
Pipes [setup in B]

```
$ A | B
```


Parent
A
B
Process #1
Process #2
Process #3
User
Kernel
File Descriptor table Process #1
0
1
2
3
4
5
IN
OUT
ERR
File Descriptor table Process #2 (A)
0
1
2
3
4
5
IN
ERR
File Descriptor table Process #1
0
1
2
3
4
5
OUT
ERR

Pipes [cleanup in Parent]

```
$ A | B
```


Parent
A
B
Process #1
Process #2
Process #3
User
Kernel
File Descriptor table Process #1
0
1
2
3
4
5
IN
OUT
ERR
File Descriptor table Process #2 (A)
0
1
2
3
4
5
IN
ERR
File Descriptor table Process #1
0
1
2
3
4
5
OUT
ERR

You can repeat this…

- Once per stage of the pipeline

Remember

- All processes in the pipeline
  - Are running concurrently
  - As soon as data sent in the pipe…
  - The next process picks it up and starts working.
- When the first (source) process dies…
  - Its output is closed
  - But its child keeps reading “remaining inputs”
  - And dies when he has written its last output.

Opened files for a dying process?

- When a process ends
  - All its opened files are automatically closed
- Therefore….
  - Any lingering pipes are closed automatically
- Only danger
  - Not closing properly in the parent (shell) process!