Netzwerkprogrammierung – Network Programming

Interprocess Communication
Signals and Pipes

Jan Krüger
jkrueger@cebitec.uni-bielefeld.de

Alexander Sczyrba
asczyrba@cebitec.uni-bielefeld.de

Interprocess communication

Parent and child processes can communicate via:

- signals
- pipes
- shared memory

Signals

- a technique to pass messages to a running process
- e.g. a *division by zero* causes signal `SIGFPE`
- processes can react to a signal by:
  1. ignoring it
  2. perfoming a default action (`SIGFPE` → terminate)
  3. calling a custom made function (*signal handler*) when signal is triggered

POSIX Signals (excerpt)

name	value	default	meaning
HUP	1	T	Hangup detected
INT	2	T	Interrupt from Keyboard
ILL	4	T	Illegal Instruction
FPE	8	T C	Floating point exception
KILL	9	T !	Termination signal
PIPE	13	T	Write pipe with no readers
TERM	15	T	Termination signal
CHLD	17	I	Child terminated
CONT	18	R	Continue if stopped
STOP	19	S !	Stop process

T: terminate, C: continue, ! : can't be caught, R: resume, S : stop, C :core dump

`os.kill()`

In Python `os.kill()` sends a signal to another process:

```
os.kill(pid, sig)
```

- sends signal `sig` to process `pid`
- `sig` is either the signals int number or its symbolic name (`signal.SIG*`)
- no return value, but error state if `pid` does not exist.

Signalling a process *group:*

`os.killpg()` or negated `pid` in `os.kill()`

Catching signals

- signal handler is a function with exactly two arguments:
  1. signal number
  2. interrupted stack frame
- signal handler must be registered
  `signal.signal(signalnum, handler)`
  - `signalnum:` integer value of the signal
  - `handler:` function to be called (signal handler)
- Note: do not use computationally demanding calls in a signal handler!

Hands on!

1. Write a Python program which infinitly prints „I'm sleeping“ every 5 seconds.
2. How to terminate the above program?
3. Extend your program such that it can react to „keyboard interrupt signals“ (SIGINT). It shall terminate after the THIRD catched SIGINT signals.

Pipes

- Original form of Unix Interprocess Communication (IPC) [1973]
- useful for many scenaries
- Short comming: no identifier, thus only usable by related processes.
- revised 1983, by introducing FIFOs (*named pipes*)
- Pipes and FIFOs are used by the common read and write file operation functions

Pipes in Python

- create a pipe via `os.pipe()`

```
inpipe, outpipe = os.pipe()
```

- write to a pipe:

```
os.write(outpipe, <MESSAGE>)
```

- read from a pipe:

```
os.read(inpipe, <BUFFERSIZE>)

fdHandle = os.fdopen(inpipe)
line = fdHandle.readline()   # a single line
lines = fdHandle.readlines() # all lines
```

Pipes in Python

- pipes are buffered, always! → no real-time output.
- enforcing real time: `os.read(inpipe, 1)`
- remember to close your pipes!

Pipes might be replaced by `os.dup2()`:

```
stdin = sys.stdin.fileno()
stdout = sys.stdout.fileno()
parentStdin, childStdout  = os.pipe()
childStdin,  parentStdout = os.pipe()

if(os.fork() ==0):
    # child process
    os.close(parentStdin)
    os.close(parentStdout)
    os.dup2(childStdin,  stdin)
    os.dup2(childStdout, stdout)
    print("Hallo Elternprozess!")
```

Named Pipes in Python

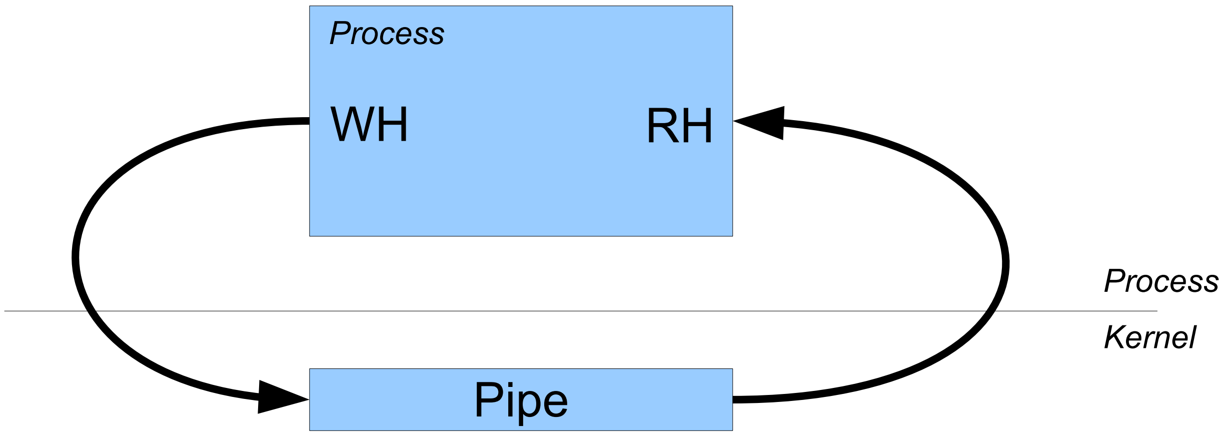
Arbitrary (i.e. unrelated) processes communicate via named pipes.
Named pipes are handled like files by the OS.

```
if not os.path.exists(pipe_name):
    os.mkfifo(pipe_name)

pipeout = os.open(pipe_name, os.O_WRONLY)
os.write(pipeout, 'Number %03d\n' % counter)

pipein = open(pipe_name, 'r')
line = pipein.readline()
```

`os.pipe()`

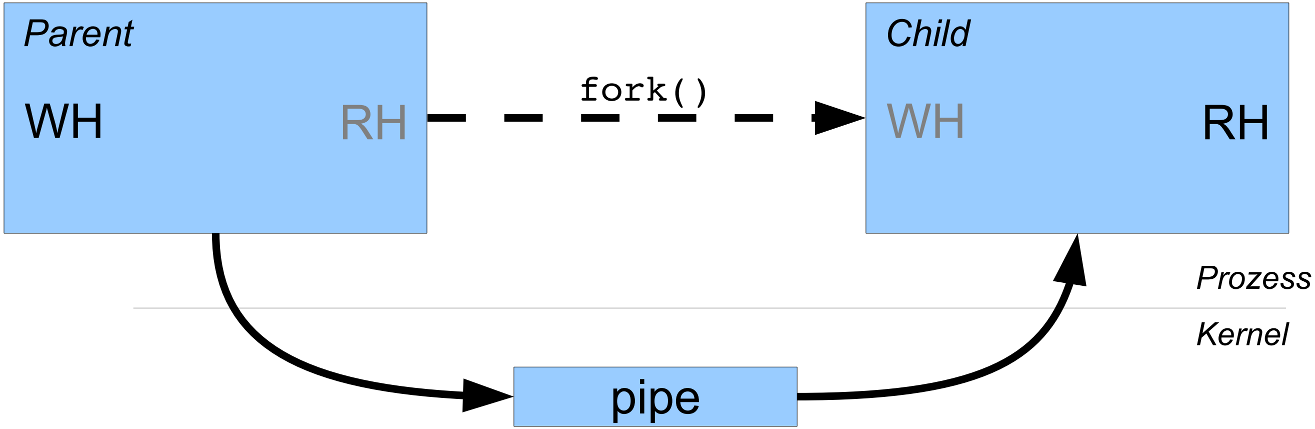
```
pipein, pipeout = os.pipe()
```

- Opens a pair of file descriptors, which are connected by a pipe.



`os.pipe()`

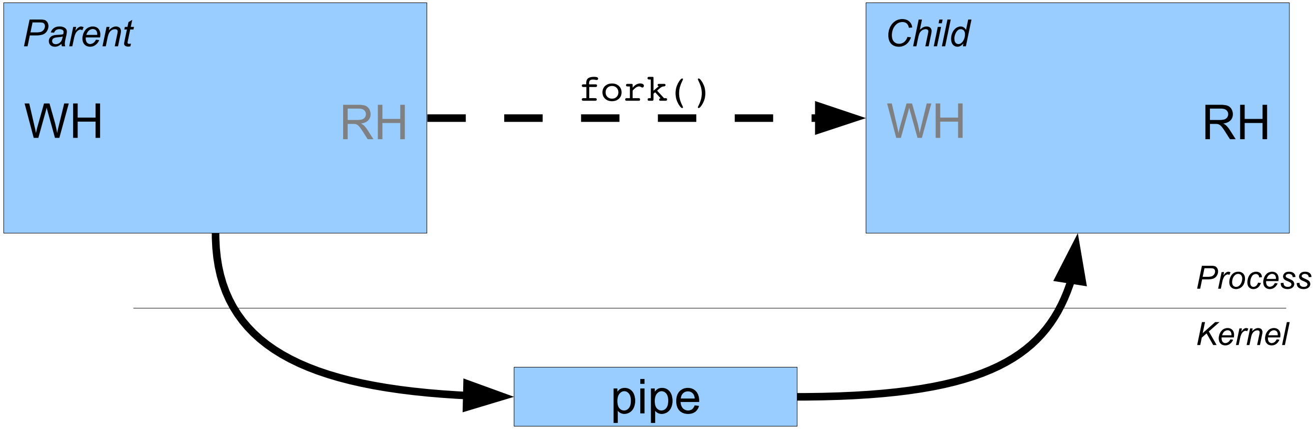
- usually in combination with a fork()
- parent opens first file handle and closes the other
- child acts reversly: closes first file handle and opens the other



Hands on!

Write a program, which

1. opens a pipe,
2. forks,
3. sends a message from the parent to the child,
4. which ultimately prints the message.



Bidirectional Pipes

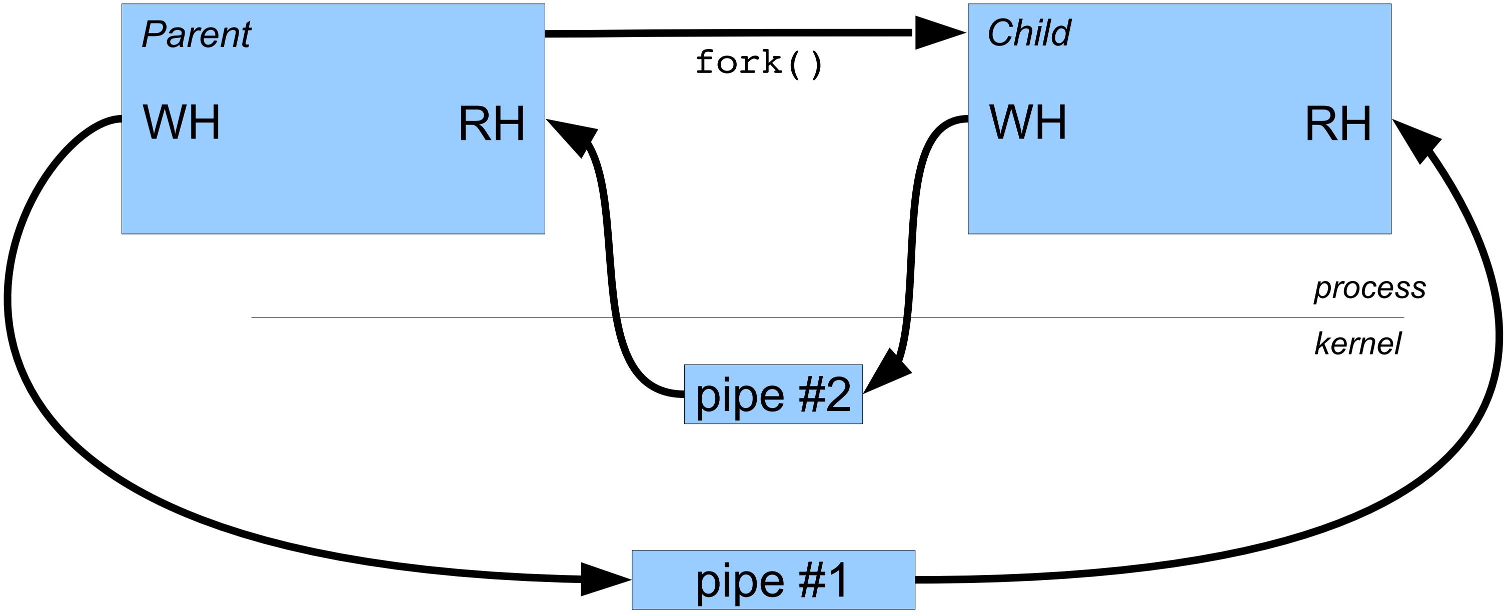
- presented pipes were *half-duplex* or *unidirectional.*
- passing information in one direction
- bi-directional communication requires two pipes → one per direction

Hands on!

Write a program, that forks and can bi-directionally communicate between parent and child. Do so by:

1. creating two pipes A and B
2. `os.fork()`
3. in parent
    1. close reader of pipe A
    2. close writer of pipe B
4. in child
    1. close writer of pipe A
    2. close reader of pipe B
5. print messages from the parent in the child and vice versa!

Hands on!


Parent
WH
RH
fork()
Child
WH
RH
process
kernel
pipe #2
pipe #1