Systems Calls - IPCs I

Alain NINANE – UCL

12 Avril 2016

System Calls - Recalls

- We have seen system calls about
  - writing/reading data
  - redirection of data to/from files
  - managing files
  - managing processes
- Process communicates between them with
  - pipes: exchange of data
  - *signals*: change of state
    - CTRL Z, fg, bg, ...
    - CTRL C to kill a program

System Calls - IPCs

- Kernel level implementation
  - Inter Process Communications (IPC I)
    - pipes -> pipes
    - commands -> signals
  - Inter Process Communications (IPC II)
    - System V IPCs
      - messages queues
      - shared memory
      - semaphores
  - Inter Process Communications (IPC III)
    - BSD network communication
      - sockets

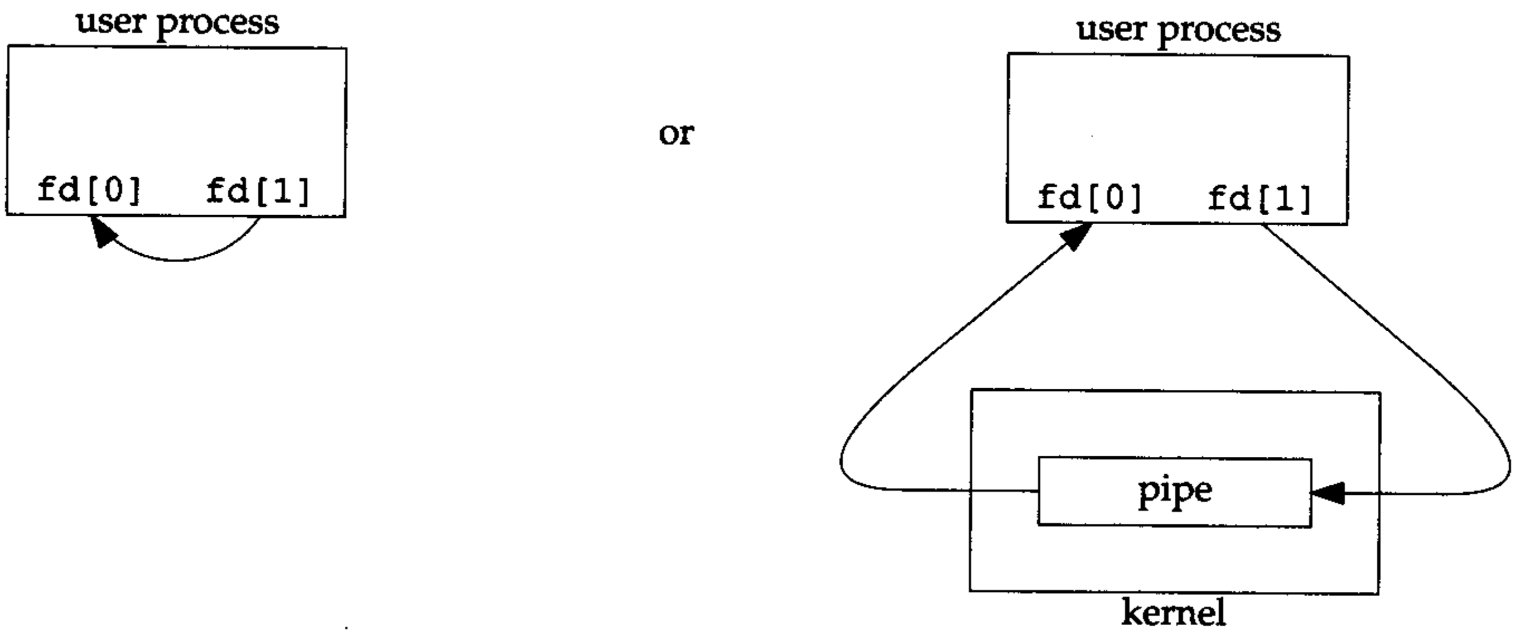
Pipes - Introduction (I)

- #include <unistd.h>
- int pipe(int filedes[2]);
  - Returns 0 or -1
  - Errno: EMFILE, ENFILE, EFAULT

- Returns two file descriptors
  - filedes[0]: file descriptor opened for reading
  - filedes[1]: file descriptor opened for writing

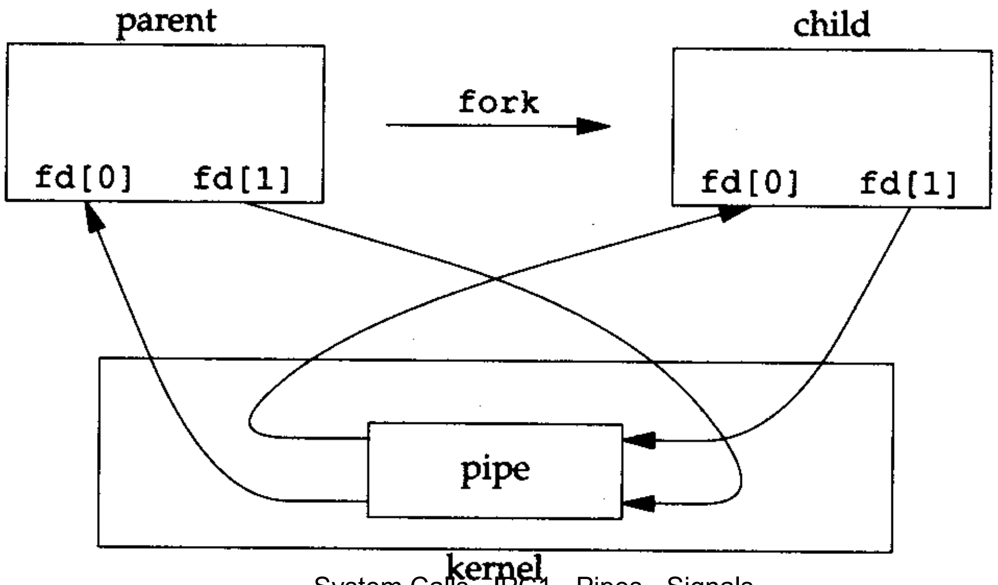
Pipes - Introduction (II)

- Pretty much useless inside a single process



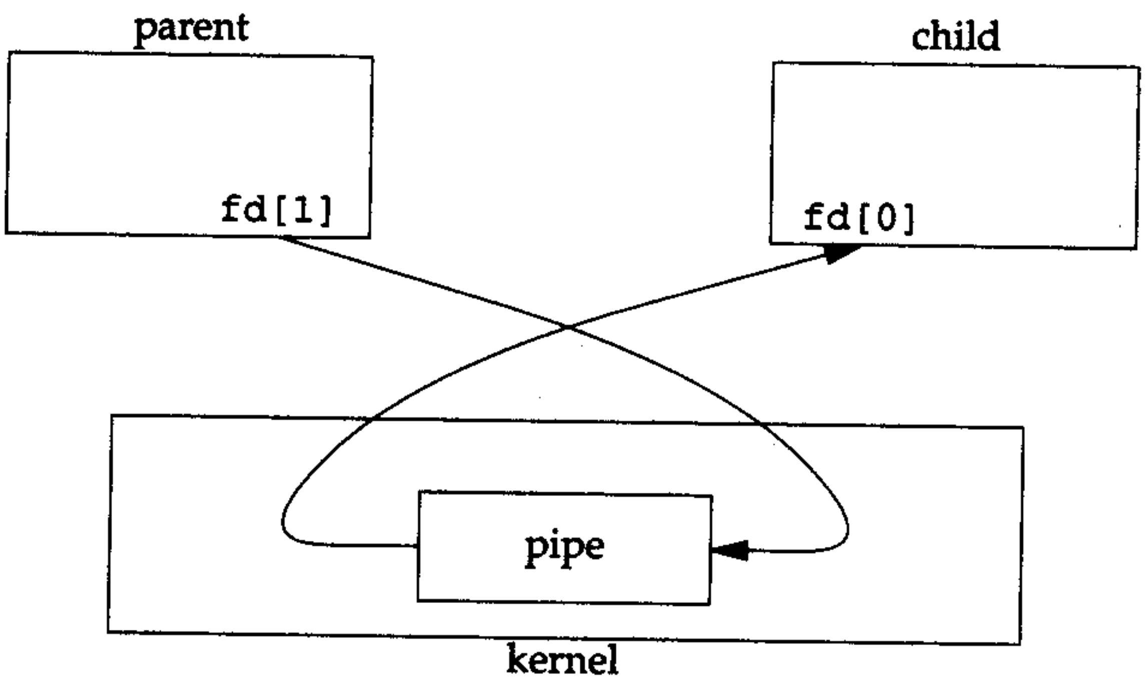
Pipes - Introduction (III)

- Pipes are more useful when combined with a fork() system
  - Recall:
    - Fork() duplicates the process



Pipes - Introduction (IV)

- The complete picture
  - Both processes close one end



Pipes - Demo test01

- The parent process will writes to the child
  - pipes(fdp[2])
  - fork()
  - parent
    - close the reader
    - write data - with/without fflush() !
  - child
    - close the writer
    - read data
  - Both processes: use of fdopen()

Pipes - General Remarks (I)

- Pipes are half-duplex. Data flows in one direction.
- Pipes can only be used between processes with a common ancestor.
  - Pipes must be set up before fork()
- Synchronization achieved by:
  - blocking a reader when pipe is empty
  - blocking a writer when the pipe is full
  - but ... see later !!!!!

Pipes - General Remarks (II)

- Many pipes may be set up between two or more processes; a process can be both reader and writer.
- Both processes must agree about the size and format of the exchanged messages.
- **Demo: test02**
  - The parent send a token (an integer) to the child
  - The child increments the token and send it back to the parent who prints it.
  - Needs two pipes, one for reading, one for writing

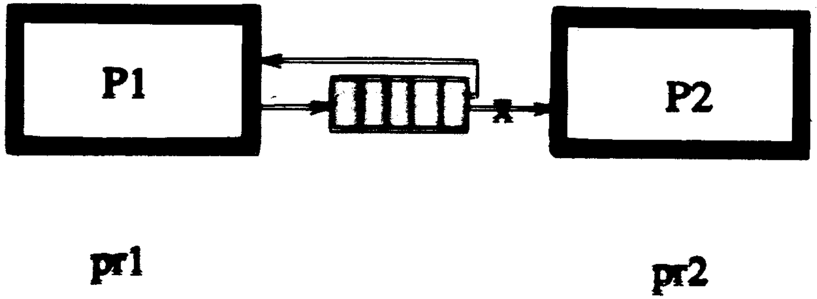
Pipes - Dup’ed with stdio

- file descriptors obtained by pipes can be reassigned to stdin/stdout descriptors
- int fdp[2];
- ret = pipe(fdp)
- Writer
  - close(fdp[0])
  - dup2(fdp[1],1); // close stdout - fdp[1] new stdout
- Reader
  - close(fdp[1]);
  - dup2(fdp[0],0); // close stdin - fdp[0] new stdin
- **Demo: test04**

Pipes - Special Situations (I)

- Synchronization okay but ....
  - read() on a pipe with no writers
    - returns EOF
  - write() on a pipe with no readers
    - sends SIGPIPE
- Each process MUST close the unused file descriptors otherwise deadlocks may (read WILL) occur.
- Loi de Murphy : If there is less than 0.000000001 % probability that a problem can occur, there is a 100 % probability that it will occur. So... allways be safe !

Pipes - Special Situations (II)



- Parent P1 “forgot” to close reader side
- Child P2 terminates
- Parent P1 continues to write data to pipe
  - But doesn’t read them ...
  - Deadlock occurs !!
- **Demo: test05**
  - Parent enters deadlock when pipe full
  - Child seen as a zombie

Pipes - To Another Process

- File descriptors correctly closed
- Child can be overlaid by another process
  - simple example: /bin/more
  - **demo: test06**
  - argv[0] is MyOwnMore ... !
  - close(1) on parent process to allows the child to exits.

Pipes - popen() stdio library

- The process of sending data to another process can be simplified by fct popen() from the stdio library
  - #include <stdio.h>
  - FILE *popen(const char *cmd, const char *type);
  - int pclose(FILE *stream);
- **Demo: test07**
  - type is “r” or “w”

Pipes - named pipes

- Pipes seen as anonymous
- Pipes must have a common ancestor
  - pipes created **before** fork()
- There exists “**Named Pipes**”
  - have a name
  - connect independent processes
  - created my mkfifo or mknod
- **Demo**:
  - pipe: mkfifo /tmp/bar
  - process 1 : send beer names to the bar
  - process 2 : sort beers

Interrupts - A Definition

- Hardware interrupts
  - An electrical signal is sent by a device to the µProcessor to stop executing current process and do something different (i.e. handle interrupts).
  - Interrupts handler is selected by the µProcessor by reading the *interrupt vector* from the device
  - The execution state (the context) is saved before executing the interrupt handler and restored afterwards.

Signals - A Definition

- Signals are *Software* Interrupts
- Signals may be sent to a process:
  - from another process
    - e.g. kill something
  - from its controlling terminal
    - e.g. hit CTRL C or CTRL Z
  - from itself
  - from the system
    - e.g. accessing an invalid memory pointer or divide by 0

Signals - Actions

- Default action on signal for processes:
  - terminate process
  - depending on signal: produce a *core dump* file.
- A process may decide to ignore signals
- A process may decide to catch signals
  - i.e. handle the software interrupts by itself
- A signal is sometimes sent to a process group
  - i.e. a group of processes from the same parent
- A signal can only be sent to a process with the same user identification (uid) - exception for uid=0 (root)

Signals - List (I - incomplete)

SIGHUP	1	Hangup/Reload	Terminate
SIGINT	2	Keyboard interrupt	Terminate
SIGILL	4	Illegal Instruction	Core dumped
SIGFPE	8	Arithmetic exception	Core Dumped
SIGKILL	9	Kill	Terminate
SIGBUS	10	Bus error	Core Dumped
SIGSEGV	11	Segmentation Violation	Core Dumped
SIGPIPE	13	Write a pipe (no reader)	Terminate

Signals - List (II - incomplete)

SIGALRM	14	Alarm clock	Terminate
SIGTERM	15	Software termination	Terminate
SIGSTOP	17	Stop	Suspended
SIGTSTP	18	Stop (from keyboard)	Suspended
SIGCONT	19	Continue after stop	Discarded
SIGCHLD	20	Child status has changed	Discarded
SIGUSR1	30	User defined signal 1	Terminate
SIGUSR2	31	User defined signal 2	Terminate

Signals - Note

- Some signals can be generated by the keyboard
  - CTRL C: SIGINT
  - CTRL \: SIGQUIT (with core dump file)
- Signals are handled only when a process
  - is active
  - is waiting for I/O on a slow device
  - is waiting for I/O on a pipe
- If signals arrives during I/O on a pipe or a slow device, the I/O terminates prematurely
- Some syscalls terminates with ret -1 and errno INTR.

Signals - kill (I)

- Signals are sent with the kill() system call
- #include <signal.h>
- int kill(int pid, int sig)
  - pid = destination process or process group
  - sig = the signal name or number (man 7 signal)
- returns 0: OK
- returns -1: ERROR
  - EINVAL: invalid signal number;
  - ESRCH: pid does not exist
  - EPERM: permission denied

Signals - kill (II)

- sig = 0: no action performed (validity check)
- pid > 0:   signal sent to process
- pid = 0: signal sent to all processes of the caller's process group
- pid = -1: signal sent to all processes (except init and other users processes)
- pid < -1: signal sent to all processes of the process group (-pid)

Signals - kill (III)

- How to obtain pid ?
  - returns value from fork()
  - getpid()
    - my own pid
  - getppid()
    - my parent’s pid

Signals - from the shell

- kill -NAME pid
- kill -number pid
- to get pid ...
  - top
  - ps
    - ps xu
    - ps aux
    - ps alx
    - .....

Signals - signal (I)

- Handling of signals
  - #include <signal.h>
  - typedef void (*sighandler_t)(int);
  - sighandler_t signal(int signum, sighandler_t handler);
    - signum: signal number
    - handler:
      - SIG_DFL: default signal specific handler
      - SIG_IGN: ignore the signal
      - handler: user specified

Signals - handling (I)

- queuing: only one signal of every type can be registered for a process
  - several signals of the same type to one process
  - process handle only one - others are lost
- fork: all signals settings are preserved in the child process
- exec:
  - signals to be ignored are still ignored
  - signals to be caught: default action

signals - handling (II)

- Different behaviour between BSD and SYSV
  - SYSV:
    - when signals is handled, the handlers is reset to default
  - BSD:
    - do not reset the handler to default but **blocks** signals until end of handler
    - blocks means
      - not discard: one is queued
      - delivered later

signals- SYSV behaviour (I)

- Example of a handler to remove tmp files
  - main()
  - {   signal(SIGINT,cleanTmp);
    - creat(tempfile,mode);
    - ...
    - unlink(tempfile); // normal exit
    - exit(0);
  - }
  - cleanTmp()
  - {   .... lot of code ....
    -     unlink(tempfile);
    -     exit(1);
  - }

signals - SYSV behaviour (II)

- A window of vulnerability exists
  - If a SIGINT signal occurs in cleanTmp() before calling unlink() ?
    - signal handler for SIGINT reset to default
    - process terminates before removing (unlink) the file
- Solution:
  - use sigaction() which allows to mask signals during the treatment of the handler.
  - use BSD behaviour which block signals during handler processing

demo - test08

- floating point exception
  - integer divide by 0
- core dump file
  - a copy of the process memory
  - new linux: core not written
    - ulimit -c size
- program compiled with debug
  - cc -g -o test08 test08.c
- core dump analyze
  - gdb test08 core.xxx

demo - test09

- Handlers
  - SIGFPE (divide by 0)
  - SIGINT (CTRL C)
- BSD vs SYSV behaviour
  - #define _XOPEN_SOURCE 1

    SYSV behaviour

timers - SIGALRM

- #include <unistd.h>
- unsigned int alarm(unsigned int seconds);
  - sends the SIGALRM after seconds ...
- **Demo: test10**
  - alarm must be re-enabled !
  - shows load-distribution between processors.