



Use the FreeBSD System Calls

by Gregory HOLLAND
bsdforever@hotmail.fr



Overview 1/2

- 1. Introduction
 - What's a System Call ?
 - What's a “Standard” ?
 - Portability issue

 - 2. Process Management System calls
 - fork
 - wait
-
-



Overview 2/2

- 3. File Management System calls
 - open
 - close
 - read
 - write
 - 4. Interprocess Communication
 - IPC principles
 - pipe
 - redirection
 - 5. Conclusion
-
-

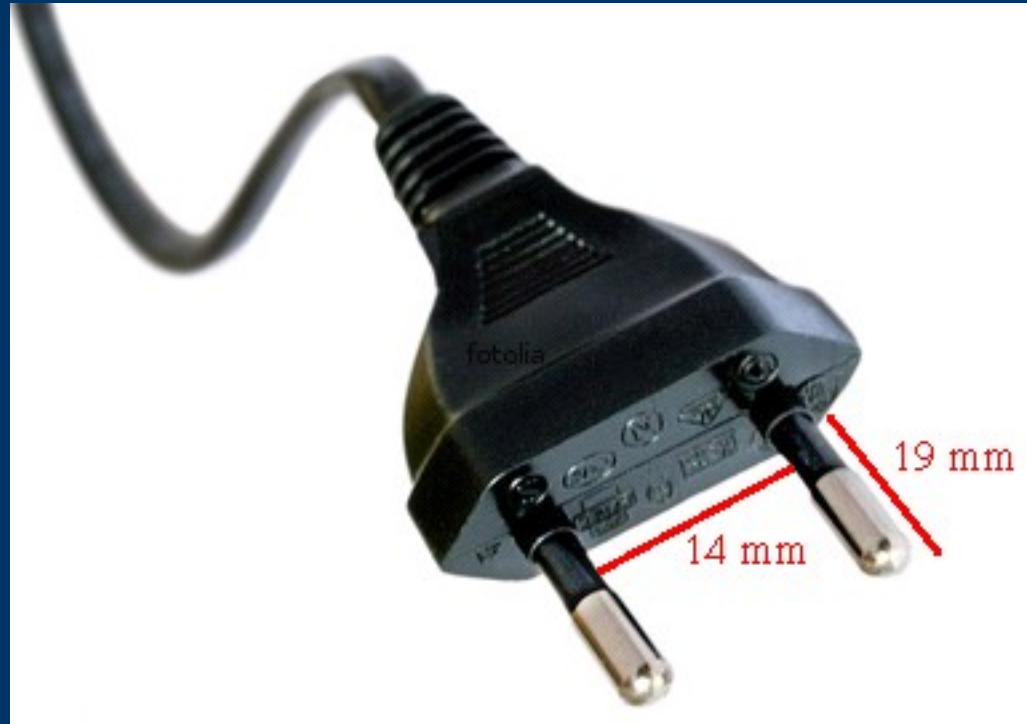
1.1 What's a “System Call” ?



- Interfaces to the kernel
- A unique entry point to the kernel
- Limited set for each Unix kernel
- Most of [Unix] kernels are compliant to a “Standard”



1.2 What's a "Standard" ?



- IEC 60083 standard specifies 14 mm between pins
 - You can build a non-standard plug with 20 mm between pins if you want...
-
-

1.3 Several 'differents' standards

- POSIX (also referred as “IEEE 1003.1”)
 - IEEE Std 1003.1-1990 (POSIX.1)
 - “IEEE Std 1003.2-1992 (POSIX.2) and its subsequent amendments, combined with the core volumes of the Single UNIX Specification, Version 2”
- Single Unix Specification
 - Based on earlier work by IEEE and The Open Group

- ...



1.4 What's "IEEE 1003.1" looks like

NAME

sys/types.h - data types

The Open Group Base Specifications Issue 6

IEEE Std 1003.1, 2004 Edition

Copyright © 2001-2004 The IEEE and The Open Group, All Rights reserved.

SYNOPSIS

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

DESCRIPTION

The `<sys/types.h>` header shall include definitions for at least the following types:

blkcnt_t

Used for file block counts.

blksize_t

Used for block sizes.

clock_t

[XSI]  Used for system times in clock ticks or CLOCKS_PER_SEC; see [<time.h>](#). 

...

All of the types shall be defined as arithmetic types of an appropriate length, with the following exceptions:

[XSI] 

key_t



...

Additionally:

- **mode_t** shall be an integer type.
- **size_t** shall be an unsigned integer type.

...

The type **ssize_t** shall be capable of storing values at least in the range [-1, {SSIZE_MAX}].

...





1.5 Portability

- Programs ...
 - are no more “Standard C programs”
 - are UNIX C programs
 - are portables across O.S. that are compliant to the same standard
-
-

1.6 List of the System Calls



Complete list in the file
“/usr/include/sys/syscall.h”

```
/*  
 * System call numbers.[...]  
 */  
#define SYS_syscall 0  
#define SYS_exit 1  
#define SYS_fork 2  
#define SYS_read 3  
#define SYS_write 4  
#define SYS_open 5  
#define SYS_close 6  
#define SYS_wait4 7  
...
```

1.7 System calls categories



Processes Management

fork, wait, exec, alarm, getpid, ...

I/O

open, creat, read, write, lseek, dup, close, ...

File/Directory Management

link, unlink, mount, umount, stat, ...

Signaling

sigaction, sigpending, sigsuspend, ...

Protection

chmod, chown, getuid, setuid, umask, ...

Misc

time

1.8 General considerations 1/2

- [bsdgreg@bsdmania ~]\$ man 2 sys_call_name
- [bsdgreg@bsdmania ~]\$ man 3 standard_C_function_name
- Sections of a man page :
 - NAME
 - LIBRARY
 - SYNOPSIS
 - DESCRIPTION
 - **RETURN VALUES**
 - **ERRORS**
 - SEE ALSO
 - **STANDARDS**
 - HISTORY



1.8 General considerations 2/2

- On success
 - return value ≥ 0
- On error
 - the call return -1
 - existing errors are in `<sys/errno.h>`
 - variable “errno” is set, but not reset between SysCalls

void perror(const char *string);

use errno to find the right error message
and write it to the standard error file descriptor

char *strerror(int errnum);

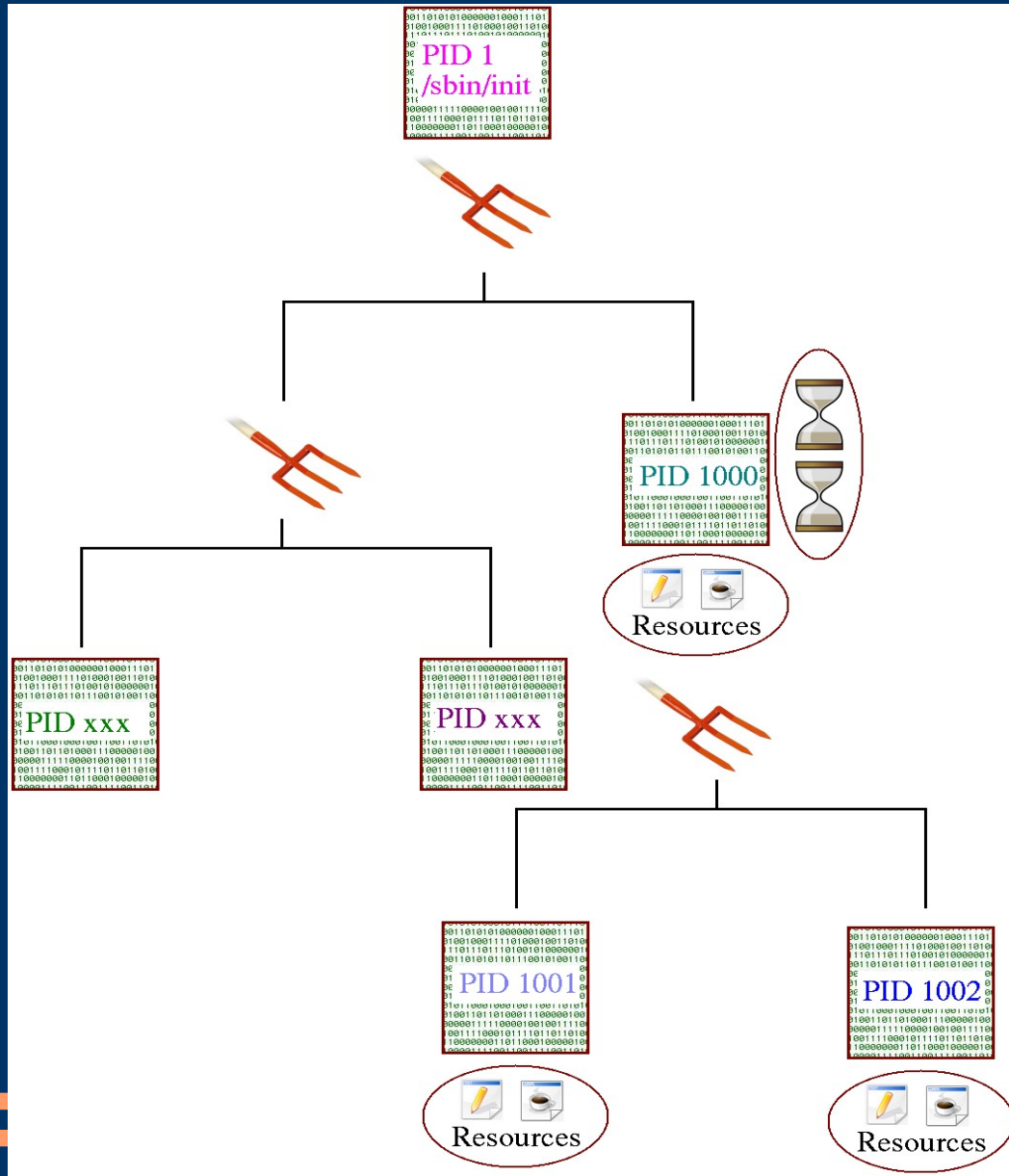
returns a pointer to the corresponding message string

2. Processes Management SysCalls

- Create a process :
`pid_t fork(void);`
- Wait for the termination of a process :
`pid_t wait(int *status);`
- Execute another program :
`exec...`



2.1 What does 'fork' do ?



2.2 *fork() shared 'objects'*



```
pid_t fork(void);
```

- `fork()` returns ...
 - 0 to the child process
 - the process ID of the child process to the parent
 - Child process has ...
 - a unique PID
 - a different parent process ID
 - its own copy of the parent's descriptors
(file pointers are shared between child and parent)
 - resource utilization counter set to 0
 - Not shared :
 - all interval timers are cleared
-
-



2.3 wait()

```
pid_t wait(int *status);
```

- suspends execution of its calling process until
 - status information is available for a terminated child process
 - a signal is received
-
-

2.4 *wait()* - the 'status'



```
pid_t wait(int *status);
```

- **WIFEXITED(status)**
 - True if the process terminated normally by `_exit(2)` or `exit(3)`
 - **WIFSIGNALED(status)**
 - True if the process terminated due to receipt of a signal
 - **WIFSTOPPED(status)**
 - ...
-
-

2.5 fork() and wait() - example

```
01 int main() {
02     int pid;
03     int status;
04
05     if ((pid = fork()) < 0) {
06         perror("fork failed");    exit(1);
07     }
08
09     if (pid) {    //father
10         if (wait(&status) < 0) {
11             perror("wait error");    exit(1);
12         }
13
14         if (WIFEXITED(status)) {
15             printf("Child terminated !");
16         }
17     } else {    //child
18         sleep(2);
19     }
20
21     return 0;
22 }
```





3. File Management

- Unix philosophy : everything is a file !

<u>Example</u>	<u>Rights</u>	<u>Comment</u>
- /dev/ad12s1	crw-r-----	a slice !
- /dev/acd0	crw-r-----	IDE cd-rom drive
- /dev/psm0	crw-rw-rw-	mouse on PS/2
- /bin/lis	-r-xr-xr-x	a command file
- a symbolic link	lrwxr-xr-x	
- a pipe	prw-r--r--	
- ...		

3.1 Main System Calls



```
int open(const char *path, int flags, ...);  
returns a "File Descriptor"
```

```
int close(int d);  
d is a "File Descriptor"
```

```
ssize_t read(int d, void *buf, size_t nbytes);  
returns the number of bytes actually read
```

```
ssize_t write(int d, const void *buf, size_t nbytes);  
return the number of bytes which were written
```

3.2 Open modes



```
int open(const char *path, int flags, ...);
```

```
const char *path  
    "/tmp/myFile"  
    "/dev/null"
```

```
int flags
```

O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_NOFOLLOW	do not follow symbolic links (FreeBSD add'on)
O_CREAT	create file if it does not exist
...	

```
other:
```

```
0666
```

return value : the smallest file descriptor available

3.3 read / write



```
ssize_t read(int d, void *buf, size_t nbytes);  
ssize_t write(int d, const void *buf, size_t nbytes);
```

int d

- File Descriptor (perhaps this one returned by open)

void *buf

- buffer where data can be read/written
- “const” if data are written => data are not modified

size_t nbytes

- number of byte to read/write

“size_t” ? IEEE 1003.1-2004 : “Used for sizes of objects.”

3.4 example : open / write / close

```
01 int main() {
02     int fd;
03     int res;
04     char *string;
05     char buf_read[101];
06
07     string = "BSD : The Power to Serve";
08
09     if ((fd = open("myfile", O_RDWR | O_CREAT, 0666)) < 0) {
10         perror("open error");
11         exit(1);
12     }
13
14     if ((res = write(fd, string, strlen(string))) < 0) {
15         perror("write error");
16         exit(1);
17     }
18
19     if (close(fd) < 0) {
20         perror("close error");
21         exit(1);
22     }
23     ... to be continued...
```



3.5 example : open / read / close

```
24  if ((fd = open("myfile", O_RDONLY)) < 0) {
25      perror("open error");
26      exit(1);
27  }
28
29  if ((res = read(fd, buf_read, 100)) < 0) {
30      perror("read error");
31      exit(1);
32  }
33
34  buf_read[strlen(string)] = '\0';
35  printf("read : %s", buf_read);    /* print "BSD : The Power to Serve" */
36                                  /* to the standard output */
37
38  if (close(fd) < 0) {
39      perror("close error");
40      exit(1);
41  }
42 }
```



4. Interprocess Communication

Different ways :

- Pipes
- Signals
- BSD behaviour :
 - Sockets
- System V behaviour :
 - Message queues
 - Shared memories
 - Semaphores



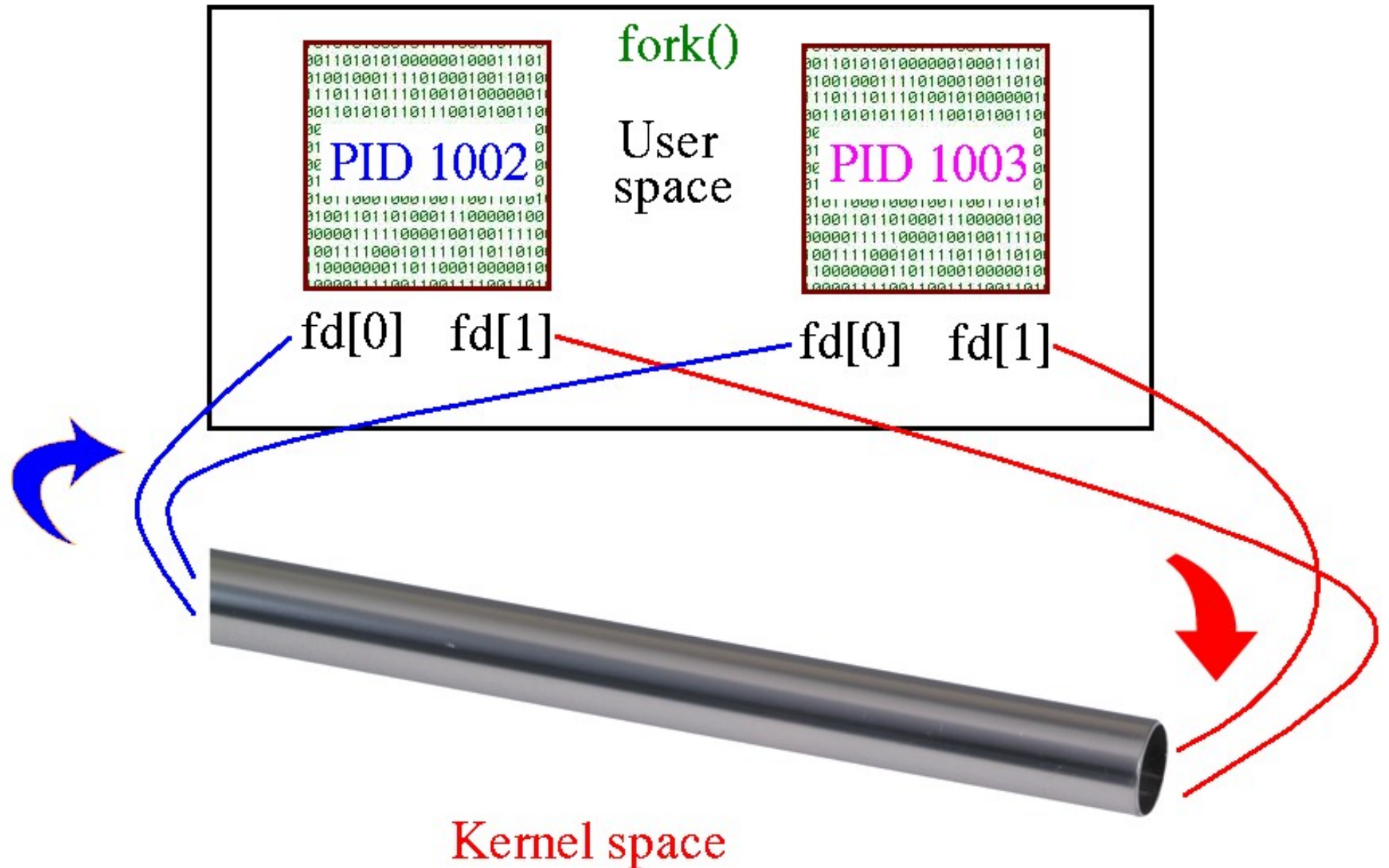
4.1 IPC - pipes



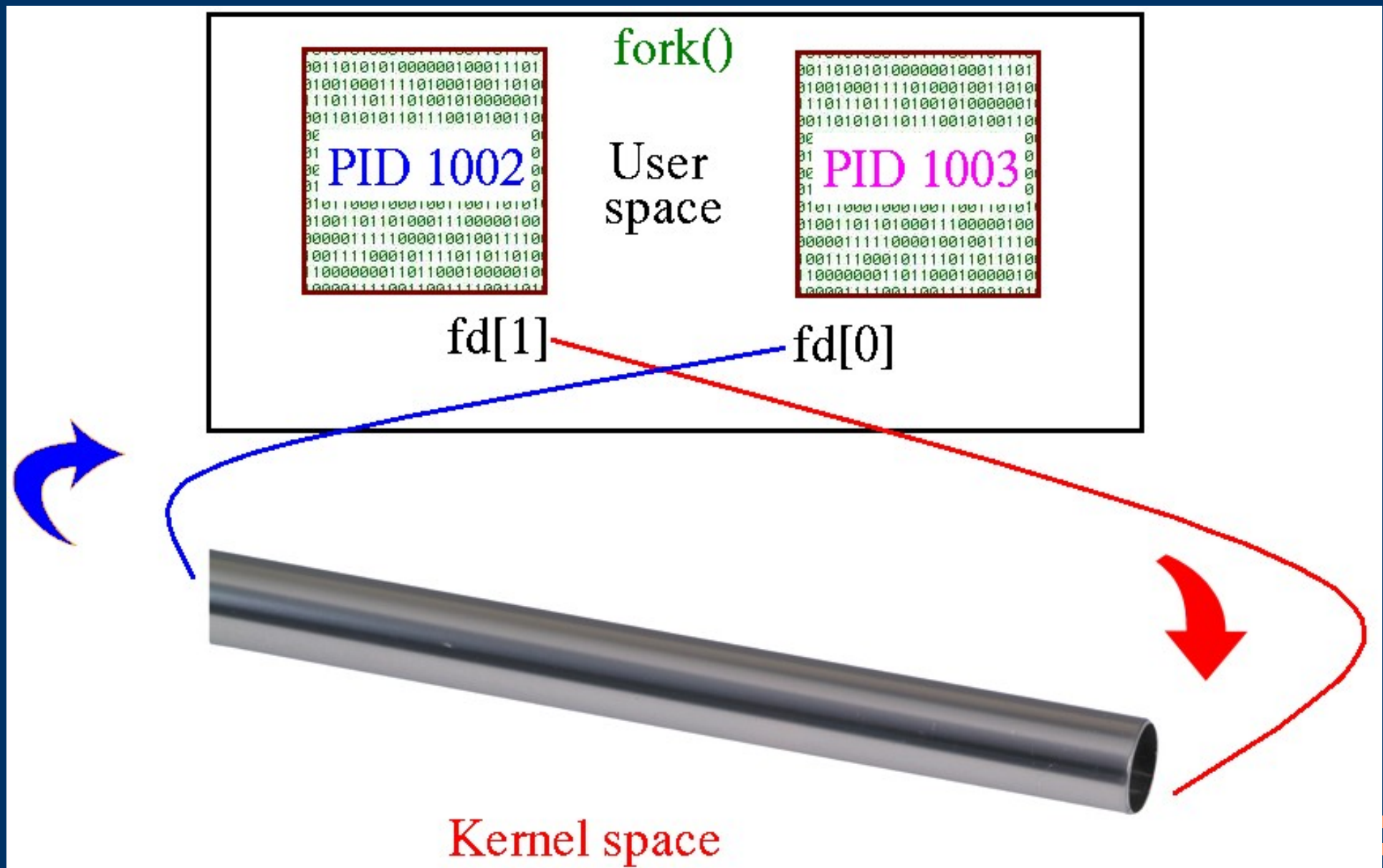
- “Classical” way to do UNIX IPC
 - Half duplex
 - Need a common ancestor
 - Blocking
 - block the reader when the pipe is empty
 - block the writer when the pipe is full
 - Read on a pipe with no writer : returns EOF
 - Write on a pipe with no reader : sends SIGPIPE
 - Pipe's size :

```
[bsdgreg@bsdmania ~]$ ulimit -a | grep pipe  
pipe size          (512 bytes, -p) 1
```
-
-

4.2 IPC - pipes



4.3 IPC - 1 reader and 1 writer closed



4.4 IPC : pipes 1/2



```
01 int main() {
02     int pid, res;
03     int fd[2];
04
05     if (pipe(fd) < 0) {
06         perror("pipe error");    exit(1);
07     }
08
09     if ((pid = fork()) < 0) {
10         perror("fork error");    exit(1);
11     }
12
13     if (pid) {
14         char *string;
15
16         string = "Ba Bu Bi Bo Be...SD !";
17
18         if (close(fd[0]) < 0) {
19             perror("close pipe error");    exit(1);
20         }
21
22         if (write(fd[1], string, strlen(string) + 1) < 0) {
23             perror("write error");    exit(1);
24         }
25     } ...
```

4.5 IPC : pipes 2/2



```
26 } else {
27     char buffer[100];
28
29     sleep(2);
30
31     if (close(fd[1]) < 0) {
32         perror("close pipe error");
33         exit(1);
34     }
35
36     if (read(fd[0], buffer, 100) < 0) {
37         perror("read error");
38         exit(1);
39     }
40
41     printf("read : %s\n", buffer);
42 }
43
44 return 0;
45 }
```

4.6 Redirection

- Each process has existing file descriptors :
 - FD 0 stdin
 - FD 1 stdout
 - FD 2 stderr

- Do you remember the “open” system call ?

```
int open(const char *path, int flags, ...);
```

return value : the smallest file descriptor available





4.7 Redirection

- `int dup(int oldd);`
duplicate an existing file descriptor and returns its value to the calling process
 - `int dup2(int oldd, int newd);`
 - Atomic SysCall for :
`close(fd2);`
`fd2 = dup(fd1);`
 - Atomic : no signals can happened between the two operations
-
-

4.8 Redirection : example

```
01 int main() {
02     int new_stdout, old_stdout;
03     char *texte;
04
05     if ((old_stdout = dup(1)) < 0) {           //save original stdout
06         perror("error dup\n");             exit(1);
07     }
08
09     //create a new stdout
10     if ((new_stdout = open("outfile", O_RDWR | O_CREAT, 0666)) < 0) {
11         perror("can't open new stdout\n");  exit(1);
12     }
13
14     if (dup2(new_stdout, 1) < 0) {           //make redirection
15         perror("can't replace stdout\n");   exit(1);
16     }
17
18     printf("printf on stdout\n");           //print on new stdout (outfile)
19
20     fprintf(stdout, "fprintf on stdout\n");
21
22     texte = "fprintf on old stdout\n";       //write to old stdout
23     if (write(old_stdout, texte, strlen(texte)) < 0) {
24         perror("can't write to old stdout\n"); exit(1);
25     }
26
27     return 0;
28 }
```

5. Conclusion



- System Calls let you go inside the kernel
- There is a jungle of standards but they share common points
- IPC is the power of UNIX and FreeBSD
- Mechanism like redirection are easy to implement



That's all, folks !

- You can contact me at bsdforever@hotmail.fr
 - for job proposition as freelance programmer
 - for any question
- Sources
 - FreeBSD man pages
 - A. TANENBAUM :
Operating System : design & implementation
 - J-C JAUMAIN “System” course (H.E.B.)
 - Dr. A. NINANE “Unix” course (U.C.L.)
 - Wikipedia
- Thanks to
 - Dr. A. PIROTTE (U.C.L.)
 - IEEE committee for their approval of using
a modified part of the IEEE Std 1003.1-2004
 - D. SEUFFERT (BSD community)

