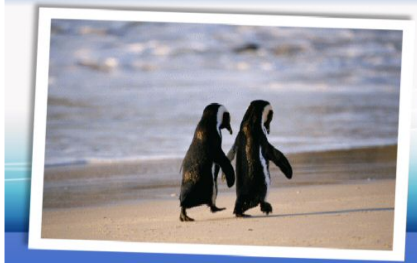

Systems Calls Input/Output (I)

Alain NINANE – RSSI UCL

23 Février 2016

System Calls

- Interfaces to the kernel
  - Written and used in C programs
  - Can be used from other languages programs (Fortran, ...)
    - (UNIX allows mixing of languages ... !)
- Limited set for each Unix kernel
- Kernel entry point is unique
  - System calls are indexes in a ‘C’ switch statement
    - MacOSX: /usr/include/sys/syscall.h
    - Linux: /usr/include/sys/syscall.h
      /usr/include/bits/syscall.h
      /usr/include/asm/unistd.h
  - Systems calls are a finite set
    - Unless rewriting/upgrading the kernel :-)
  - Number depends on the type of unix/linux kernel

System Calls

- I/O
  - open, creat, read, write, lseek, dup, ioctl, close, ...
- File Management
  - link, unlink, mknod, mount, umount, stat, fctnl, ...
- Protection
  - access, chmod, chown, getuid, setuid, umask, ...
- Processes
  - exec, fork, alarm, signal, chdir, getpid, brk, times
- Misc
  - acct, time

Portability issues (I)

- Because of UNIX diversity
  - Programs using syscalls directly
    - are no more “Standard C programs”
    - are UNIX C programs
    - non-portable across UNIX’es
  - Syscalls annoyances
    - Header files
    - Types of arguments
- Importance of man pages
  - man 2 syscall_name

Portability Issues (II)

- Header files ...
- E.g. read()
  - macosx
    - #include <unistd.h>
  - linux
    - #include <sys/types.h>
    - #include <sys/uio.h>
    - #include <unistd.h>

Portability Issues (III)

- Type of arguments ...
- E.g. signal()
  - macosx
    - typedef void (***sig _t**) (int);
    - **sig_t** signal(int sig, **sig_t** func);
  - linux
    - typedef void (***sighandler_t**)(int);
    - **sighandler_t** signal(int signum, **sighandler_t** handler);

Good Prog. Practices (I)

- Use Compilation Directives
  - CPP - C Pre Processor
  - CPP is called before real compilation to resolves directives
- Directives
  - Example in a C program
    - #ifdef MYMACOSX
      - ....
    - #endif
  - User defined
    - cc -DMYMACOSX test1.c
  - Pre defined by compiler
    - cc test1.c

Good Prog. Practices (II)

```
#ifdef __MACH__
    Code for Apple UNIX
#elif __alpha
    Code for Digital/Compaq/HP UNIX
#elif __linux__
    Code for Linux
#else
#error “Unsupported O/S”
#endif
```

CPP - Predefinitions (I)

__MACH__	Kernel Mach (Apple ...)
__alpha	Digital Alpha 64 bit CPU
__linux__	A Linux system
__gnu_linux__	Linux gcc compiler
__i386	An Intel like 386 proc.
__x86_64	An Intel 64 bit proc.

CPP - Predefinitions (II)

- How to get them ... ?
  - Experience ...
- Linux, MacOSX ....
  - cpp -dM someFile.c ...
- Other Unixe’s
  - man cc
  - man gcc
  - man cpp

Autoconfiguration

- Unix Source Code Distr./Install.
  - tar zcvf code.tar.gz
  - Uncompression process
    - unzip - untar
- Compilation
  - ./configure
    - generates a Makefile with correct cpp definitions
  - Compilation
    - make
    - make install

Standards

- Standards developed around UNIX
  - POSIX: IEEE 1003.1
- Primitive system data types
  - Hides implementation details
  - typedef’s in <sys/types.h>

ino_t	file serial number
off_t	file size
size_t	i/o size
time_t	system time
pid_t	process id

Errors handling (1 - errno)

- All syscalls returns -1 upon error

```
int fd;
fd = open("myfile",O_RDWR);
if( fd == -1 )
{   fprintf(stderr,"open error !\n");
    return -1;
}
... continue reading ...
```

- Return code MUST allways be tested
- errno: external global integer
  - Give details about the error
  - #include <errno.h>
  - #include <sys/errno.h>
  - errno NOT cleared by "next" syscall

Error handling (II - errno)

```
#include <errno.h>
extern int errno;
int fd;
if( (fd = open(“myfile”,O_RDWR) == -1 )
{   switch( errno )
    {           case EPERM:
                            ...
                break;
                case ENOENT:
                            ...
                break;
                default:
                            ...
    }
    return -1;
}
```

Error handling (III - perror)

- #include <errno.h>
- void perror(const char *s)
  - s is not NULL
    - print on the standard error:
      - s: clear_text_errno_message
  - s is NULL
    - print on the standard error:
      - clear_text_errno_message

Error handling (IV - example)

```
#include        <errno.h>
int fd;

if( (fd = open(“myfile”,O_RDWR)) < 0 )
{       perror(“open myfile”);    /* open failure */
        return -1;
}
... access granted to file ...

% ./a.out
open myfile: Permission denied
```

I/O System Calls (I)

- UNIX provides a uniform interface for performing I/O operations on ressources
- Ressources can be ...
  - files
  - terminals
  - pipes
  - tapes
  - network sockets
  - ... other devices ...

I/O System Calls (II)

- Accessing the ressource
  - open(), creat(), close(), ...
- R/W data from/to the ressource
  - read(), readv(), write(), writev(), ...
- Controlling ressource
  - lseek(), ioctl(), ...
- Special
  - select(), dup(), ...
- *Network*
  - *socket(), bind(), accept(), ...*

I/O System Calls (III)

- Ressource identified by a “small integer”, the file descriptor
  - the file descriptor is a program abstraction to access the ressource
- I/O Systems Calls are non buffered
  - each system call implies a kernel and device operation

File Descriptor (I)

- Opened files designed by a file descriptor (fd)
- fd is a “small” integer used to perform I/O
- fd returned by
  - open()
  - creat()
  - *socket()*
- int fd = open(...);
- read(fd, ...);

File Descriptor (II)

- For each process, the kernel maintains a table of open files
- Illustration:
  - command lsof = list open files
- Demo
  - ./testfd
  - ls -l /proc/PID/fd
  - /usr/sbin/lsof -p PID

File Descriptor (III)

- Observation
  - fd for “messages” file is 3
  - there exists fd 0, 1 and 2
    - connected to my “terminal”

Fd	Purpose	Initial Device
0	standard input	keyboard
1	standard output	terminal
2	standard error	terminal

File Descriptor (IV)

- Pre defined for each process
  - /* Read from standard input */
    - int ret = read(0,buffer,len);
  - /* Write on the standard output */
    - int ret = write(1,buffer,len);
  - /* Write on the standard error */
    - int ret = write(2,buffer,len);

Open() system call (I)

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

int open(const char *path, int flags, mode_t mode);
int open(const char *path, int flags);
int creat(const char *path, mode_t mode);
```

- Opens a file for reading or writing
- Returns the fd or -1
- path: string designating the ressource in the file system

Open() system call (II)

- flags

O_RDONLY	read only
O_WRONLY	write only
O_RDWR	read/write
O_CREAT	creat if ! exist
O_APPEND	pointer set to eof
O_TRUNC	if exists - truncate
O_EXCL	creat fails if exists

Open() system calls (III)

- flags
  - lot of non-portable flags
  - on linux ... man 2 open
    - O_SYNC
    - O_NOFOLLOW (free bsd)
    - O_DIRECTORY (linux)
    - O_LARGEFILE
      - Allows > 2 GB files on some 32 bit systems

Open() system call (IV)

- mode
  - Argument used with O_CREAT
  - UNIX file mode (or'ed with umask)
- errors (specified by errno)
  - EACCES
    - access not allowed
  - ENOENT
    - file does not exist or missing path component
  - ….

Open() system call (V)

```
int fd;
int flags = O_CREAT|O_TRUNC|O_WRONLY;

if( (fd = open(“../file”,flags)) == -1 )
{   perror(“open file”);
        exit(1);
}
```

- **Discussion: why a write only file ?**

Creat() & other system call

- int creat (const char *path, mode_t mode)
  - simple replacement for open with O_CREATE
- Directories
  - int mkdir(const char *path, mode_t mode);
- Special files
  - int mknod(const char *path, mode_t mode, dev_t dev);

Read() system call (I)

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t len);
```

fd: file descriptor of the opened file to read
buf: address of the pointer where input must be copied
len: maximum number of bytes to read

Returns:

ret : the number of bytes actually read.

0 : EOF is reached

-1 : an error occured (see errno or perror())

Warning:

ret may be lower than requested len

Read() system call (II)

- ret < len
  - ret = -1
    - error condition (see errno)
  - ret = 0
    - no more date to read
  - 0 < ret && ret < len
    - file size smaller than len
    - input is a terminal and user hit CTRL-D (EOF)
    - input is a raw device (e.g. a tape drive, a terminal)
    - input is a network socket

Read() system call

- Allways be prepared to read fewer data then expected
- Even if input is a “standard” file
  - std file can easily be substituted by
    - a pipe
    - a network socket
    - ...

Handling of incomplete reads

```
#define LEN     8192
char buffer[LEN];

int remaining = LEN;
int current_p =   0;

while( (ret = read(fd,&buffer[current_p],remaining)) > 0 )
{       remaining -= ret;
        current_p += ret;
}

if( ret )
{       perror("read");
        exit(1);
}
else
{       printf("eof reached !");
}
```

Write() system call

```
#include <unistd.h>
ssize_t write(int fd, void *buf, size_t len);
```

fd: file descriptor of the opened file to write
buf: address of the pointer where output must be read
len: maximum number of bytes to write

Returns:

ret : the number of bytes actually written.
0 : EOF is reached
-1 : an error occured (see errno or perror())

Warning:

ret may be lower than requested len

Some words about files

- Files are seen as sequence of bytes
  - no records
  - physical blocks not visible
- Current file offset tight to file descriptor
- Blocks devices (raw disk i/o, tapes) are seen as sequence of blocks
  - managing unit is the block
  - whole blocks are read/written at once

Lseek() system call

To move the pointer into the file ...

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, int offset, off_t whence);
```

fd: file descriptor of the opened file

offset: number of bytes to be moved by the pointer

whence: starting point

- SEEK_SET: beginning of file;
- SEEK_CUR: from the current position;
- SEEK_END: from the end of file;

Ioctl() system call

- for all other i/o operations
  - e.g. move a tape to next file
  - operations described in the driver (4) part of man
  - e.g. man st

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

int ioctl(int fd, MTIOCTOP, (struct mtop *)mt_cmd);

/* Structure for MTIOCTOP - mag tape op command: */
struct mtop
{   short  mt_op;    /* operations defined below */
    int    mt_count; /* how many of them */
};

MTBSF       Backward space over mt_count filemarks.
MTERASE     Erase tape.
MTFSF       Forward space over mt_count filemarks.
```

Close() system call

- int close(int fd)
  - Dissociates the “small integer” fd from the file
  - Returns 0 or -1
  - All files are closed on program exit.