System and Device Programming

The UNIX File System

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

File System

- The file system is one of the most visible aspects of an OS
- It provides mechanisms to save data (permanently)
- It includes management of
  - Files
  - Direttories
  - Disks and disk partitions

Files

- Information is store for a long period of time
  - Independently from
    - Termination of programs/processes, power supply, etc.
- From the logical point of view a file is
  - A set of correlated information
    - All information (i.e., numbers, characters, images, etc.) are stored in a (electronic) device using a **coding system**
  - Contiguous address space

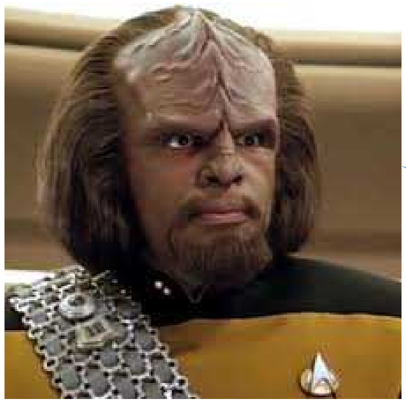
How is this information encoded?

What is the actual organization of this space?

ASCII encoding

- De-facto standard
  - ASCII, American Standard Code for Information Interchange
    - Originally based on the English alphabet
    - 128 characters are coded in 7-bit (binary numbers)
  - Extended ASCII (or high ASCII)
    - Extension of ASCII to 8-bit and 255 characters
    - Several versions exist
      - ISO 8859-1 (ISO Latin-1), ISO 8859-2 (Eastern European languages), ISO 8859-5 for Cyrillic languages, etc.

128 total characters
32 not printable
96 printable



The alphabet of the Klingom language is not supported by Extended ASCII

ASCII encoding

ASCII Table

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	|
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(	88	58	1011000	130	X					
41	29	101001	51	)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[					
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135	]					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	_					

(from commons.wikimedia.org)

128 total
32 non printable
96 printable chars

ASCII encoding

Extended ASCII Table

ASCII control characters

DEC	HEX	Simbolo ASCII	
00	00h	NULL	(carácter nulo)
01	01h	SOH	(inicio encabezado)
02	02h	STX	(inicio texto)
03	03h	ETX	(fin de texto)
04	04h	EOT	(fin transmisión)
05	05h	ENQ	(enquiry)
06	06h	ACK	(acknowledgement)
07	07h	BEL	(timbre)
08	08h	BS	(retroceso)
09	09h	HT	(tab horizontal)
10	0Ah	LF	(salto de linea)
11	0Bh	VT	(tab vertical)
12	0Ch	FF	(form feed)
13	0Dh	CR	(retorno de carro)
14	0Eh	SO	(shift Out)
15	0Fh	SI	(shift In)
16	10h	DLE	(data link escape)
17	11h	DC1	(device control 1)
18	12h	DC2	(device control 2)
19	13h	DC3	(device control 3)
20	14h	DC4	(device control 4)
21	15h	NAK	(negative acknowle.)
22	16h	SYN	(synchronous idle)
23	17h	ETB	(end of trans. block)
24	18h	CAN	(cancel)
25	19h	EM	(end of medium)
26	1Ah	SUB	(substitute)
27	1Bh	ESC	(escape)
28	1Ch	FS	(file separator)
29	1Dh	GS	(group separator)
30	1Eh	RS	(record separator)
31	1Fh	US	(unit separator)
127	20h	DEL	(delete)

ASCII printable characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
32	20h	espacio	64	40h	@	96	60h	`
33	21h	!	65	41h	A	97	61h	a
34	22h	"	66	42h	B	98	62h	b
35	23h	#	67	43h	C	99	63h	c
36	24h	$	68	44h	D	100	64h	d
37	25h	%	69	45h	E	101	65h	e
38	26h	&	70	46h	F	102	66h	f
39	27h	'	71	47h	G	103	67h	g
40	28h	(	72	48h	H	104	68h	h
41	29h	)	73	49h	I	105	69h	i
42	2Ah	*	74	4Ah	J	106	6Ah	j
43	2Bh	+	75	4Bh	K	107	6Bh	k
44	2Ch	,	76	4Ch	L	108	6Ch	l
45	2Dh	-	77	4Dh	M	109	6Dh	m
46	2Eh	.	78	4Eh	N	110	6Eh	n
47	2Fh	/	79	4Fh	O	111	6Fh	o
48	30h	0	80	50h	P	112	70h	p
49	31h	1	81	51h	Q	113	71h	q
50	32h	2	82	52h	R	114	72h	r
51	33h	3	83	53h	S	115	73h	s
52	34h	4	84	54h	T	116	74h	t
53	35h	5	85	55h	U	117	75h	u
54	36h	6	86	56h	V	118	76h	v
55	37h	7	87	57h	W	119	77h	w
56	38h	8	88	58h	X	120	78h	x
57	39h	9	89	59h	Y	121	79h	y
58	3Ah	:	90	5Ah	Z	122	7Ah	z
59	3Bh	;	91	5Bh	[	123	7Bh	{
60	3Ch	<	92	5Ch	\	124	7Ch	|
61	3Dh	=	93	5Dh	]	125	7Dh	}
62	3Eh	>	94	5Eh	^	126	7Eh	~
63	3Fh	?	95	5Fh	_			

theASCIIcode.com.ar

Extended ASCII characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
128	80h	Ç	160	A0h	á	192	C0h	└	224	E0h	Ó
129	81h	ü	161	A1h	í	193	C1h	┴	225	E1h	ß
130	82h	é	162	A2h	ó	194	C2h	┬	226	E2h	Ô
131	83h	â	163	A3h	ú	195	C3h	├	227	E3h	Ò
132	84h	ä	164	A4h	ñ	196	C4h	─	228	E4h	õ
133	85h	à	165	A5h	Ñ	197	C5h	┼	229	E5h	Õ
134	86h	å	166	A6h	ª	198	C6h	ã	230	E6h	µ
135	87h	ç	167	A7h	º	199	C7h	Ã	231	E7h	þ
136	88h	ê	168	A8h	¿	200	C8h	╚	232	E8h	Þ
137	89h	ë	169	A9h	®	201	C9h	╔	233	E9h	Ú
138	8Ah	è	170	AAh	¬	202	CAh	╩	234	EAh	Û
139	8Bh	ï	171	ABh	½	203	CBh	╦	235	EBh	Ù
140	8Ch	î	172	ACh	¼	204	CCh	╠	236	ECh	ý
141	8Dh	ì	173	ADh	¡	205	CDh	═	237	EDh	Ý
142	8Eh	Ä	174	AEh	«	206	CEh	╬	238	EEh	¯
143	8Fh	Å	175	AFh	»	207	CFh	¤	239	EFh	´
144	90h	É	176	B0h	░	208	D0h	ð	240	F0h	
145	91h	æ	177	B1h	▒	209	D1h	Ð	241	F1h	±
146	92h	Æ	178	B2h	▓	210	D2h	Ê	242	F2h	‗
147	93h	ô	179	B3h	│	211	D3h	Ë	243	F3h	¾
148	94h	ö	180	B4h	┤	212	D4h	È	244	F4h	¶
149	95h	ò	181	B5h	Á	213	D5h	ı	245	F5h	§
150	96h	û	182	B6h	Â	214	D6h	Í	246	F6h	÷
151	97h	ù	183	B7h	À	215	D7h	Î	247	F7h	¸
152	98h	ÿ	184	B8h	©	216	D8h	Ï	248	F8h	°
153	99h	Ö	185	B9h	╣	217	D9h	┘	249	F9h	¨
154	9Ah	Ü	186	BAh	║	218	DAh	┌	250	FAh	·
155	9Bh	ø	187	BBh	╗	219	DBh	█	251	FBh	¹
156	9Ch	£	188	BCh	╝	220	DCh	▄	252	FCh	³
157	9Dh	Ø	189	BDh	¢	221	DDh	¦	253	FDh	²
158	9Eh	×	190	BEh	¥	222	DEh	Ì	254	FEh	■
159	9Fh	ƒ	191	BFh	┐	223	DFh	▀	255	FFh	

From ASCII to Unicode

- C was originally developed in an English-speaking environment
  - 7-bit ASCII was sufficient
  - 8-bt ASCII become the most common encoding
- Unfortunately software for international use must be able to represent more characters
  - A variety of multi-byte encoding schemes have been internationally used for long time to represent non-Latin alphabets and non-alphabetic Chinese, Japanese, Korean, etc.

From ASCII to Unicode

- In 1994 ISO-C standardized two ways of representing large character sets
  - Wide characters
    - Same width is used for every character is a character set
    - UTF-16 and UTF-32 are implemented in wchar_t (at least 16 or 32 bit wide)
  - Multi-byte characters
    - Each character may be represented by one or several bytes
    - UTF-8 uses from 1 to 4 bytes to represent a character as wchar_t, char16_t, char32_t
- C provides standard functions fo convert formats

Unicode

- An industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems
  - The most recent version is Unicode 9.0
    - from June 2016, ISO/IEC 10646:2014 plus Amendments 1 & 2
  - The current version is 6.3, using 110,187 of the available 1.1 million code points
    - Covers 100 scripts and multiple symbol sets

The first version started out with 65536 codes, encoded in 16 bits
The second 2 more than 1.1 millions

Unicode

- Unicode is a superset of ASCII
  - The numbers 0–128 have the same meaning in ASCII and Unicode
- Because Unicode characters don't generally fit into one 8-bit byte, there are numerous ways of storing Unicode characters in byte sequences
  - Unicode can be implemented by different encoding
  - An encoding maps (possibly a subset of) the codes to sequences of values in some fixed-size range
  - Known encodings
    - UCS (now obsolete) and UTF

Unicode

- UTF encodings
  - UTF-1, UTF-7 obsolete versions
  - UTF-8
    - An 8-bit, **variable-width** encoding
    - Uses from one to four 8-bit units
    - The first 128 characters coincides with ASCII
  - UTF-16
    - A 16-bit, **variable-width** encoding
    - Uses one or two 16-bit units
  - UTF-32
    - A 32-bit, **fixed-width** encoding
    - Easy indexing (fixed-width) but space inefficient

Minimum 8 bits

Minimum 16 bits

Exactly 32 bits

Unicode problems

- A few problems still remain
  - In UTF-16/32 encodings the order of the bytes depend on the **endianness** of the machine that created the text stream
    - Big Endian
      - The most significant byte (the "big end") of the data is placed at the byte with the lowest address
      - 0x12345678 → 12 34 56 78
    - Little Endian
      - The least significant byte (the "little end") of the data is placed at the byte with the lowest address
      - 0x12345678 → 78 56 34 12

The other bytes are placed in order in the next three bytes in memory

32 bits

Increasing Adress

32 bits

Increasing Adress

Unicode

- Which UTF choice is the "best" one? Which one is used on the current system?
  - One counter-measure is the definition of a BOM (Byte Order Mark)
  - BOM = a special code-point (U+FEFF, zero width space) at the beginning of a text stream that indicates how the rest of the stream is encoded
  - It indicates both the UTF encoding and the endianess and is neutral to a text rendering engine
  - Unfortunately it is optional and many programmers claim their right to omit it, so accidents are still pretty common

Text and binary files

- A file is basically a sequence of bytes written one after the other on a physical device
  - Each byte includes 8 (or more) bits, with possible values 0 or 1
  - As a consequence all files are binary
- However, most people classify files in two categories
  - Text files (or ASCII)
  - Binary files

C sources, C++, Java, Python, etc.

Executables, Word, Excel, etc.

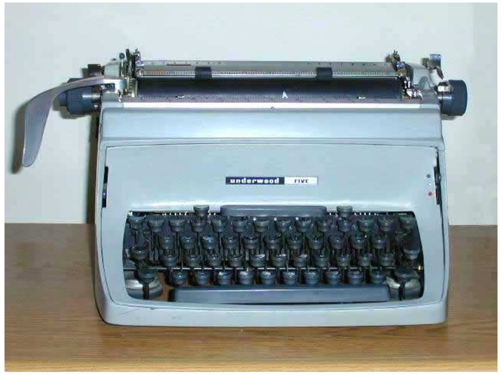
Remark:
The UNIX/Linux kernel does not distinguish between binary and textual files

Text files

- Files consisting of data encoded in ASCII (or Unicode)
  - Sequence of 0 and 1, which (in groups of 8 or more bits) codify ASCII (or Unicode) symbols
- ASCII files
  - Are stored as a sequence of binary values, i.e., a sequence of 1's and 0's
  - Are basically binary files, because they store binary numbers
  - **Are binary files that store ASCII (Unicode) codes**

Text files

- Text file are usually line-oriented
  - A newline is a set of bytes which convince the computer to go at the beginning of the next row
    - In UNIX/Linux and Mac OSX a newline is represented by a single character
      - Line Feed (go to next line, LF, 10_{10})
    - In Windows a newline is represented by two characters (as former mechanical typewriters)
      - Line Feed (go to next line, LF, 10_{10})
      - Carriage Return (push the carriage at the beginning of the line, CR , 13_{10})



Binary Files

- A sequence of 0 and 1, not “byte-oriented”
- The smallest unit that can be read/write is the bit
  - Non easy the management of the single bit
    - It's difficult to edit a binary file as individual bits should be edited
  - They usually include every possible sequence of 8 bits, which do not necessarily correspond to printable characters, new-line, etc.

Binary Files

- Why do people use binary files anyway?
  - Compactness
    - Example
      - Number 100000_{10}
      - Text/ASCII format
        - 6 characters, i.e., 6 bytes
      - Binary format
        - 100000_{10} is an integer value and it can be stored using 4 bytes

Example

"ciao"

'c' 'i' 'a' 'o'

99_{10} 105_{10} 97_{10} 111_{10}

01100011_2 01101001_2 01100100_2 01101111_2

String
Textual or binary file

"231"

'2' '3' '1'

50_{10} 51_{10} 49_{10}

00110010_2 00110011_2 00110001_2

Integer number
Text file

"231"

"231_{10}"

11100111_2

Integer number
Binary file

Serialization

- In the context of data storage, **serialization** is the process of translating **data structure** or **objects** into a format that can be stored as a single entity
  - The process of serializing an object is also called **marschalling** an object
- The opposite operation, extracting a data structure from a series of bytes, is **deserialization**
  - Deserialization is also called **unmarschalling**

Serialization

- Using serialization
  - A structure can be stored in a file (or transmitted across a network connection link) as a unique entity
    - Manipulating single fields is **not** required !
  - When it is reconstructed (or received) later the same serialization format must be used to create a semantically identical clone of the original object

```
struct mys {
  int id;
  char name[L];
  ...
} s;
```

Serialization

- Serialization breaks the opacity of an abstract data type (ADT) by potentially exposing private implementation details
  - Trivial implementations which serialize all data members may violate encapsulation
  - For complex objects, such as those that uses references, this process is not straightforward
- Several languages directly support object serialization (or object archival)

```
struct mys {
  int id;
  char *name;
  ...
} s;
```

Example

```
struct mys {
  int id;
  long int rn;
  char n[L], s[L];
  int mark;
} s;
...
_tprintf(_T("%d %ld %s %s"),
  s.id, s.rn, s.n, s.s, s.mark);
hOut = GetStdHandle (STD_OUTPUT_HANDLE);
WriteFile (hOut, &s, sizeof(struct mys),
  &nOut, NULL);
```

Binary manipulation of a structure as a unique object

Text write on standard output

Binary write on standard output

Example

```
struct mys {
  int id;
  long int rn;
  char n[L], c[L];
  int mark;
} s;
```

Text:
Single fields
Characters on 8 bits (ASCII)

```
1 100000 Romano Antonio 25

† Romano ÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ
Antonio ÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ

??? †?R?o?m?a?n?o???ÌÌÌÌÌÌÌÌÌÌÌÌÌ
ÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ
ÌÌA?n?t?o?n?i?o???ÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ
ÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ
```

Binary:
Entire structure
Ctr on 8 bits (ASCII)

Binary:
Engire structure
Ctr on 16 bits (UNICODE)
(note file size)

ISO C Standard Library

- I/O operations with ANSI C can be performed through different categories of functions
  - Character by character
    - getc, fgetc, putc, fputc
  - Row by row
    - gets, fgets, puts, fputs
  - Formatted I/O
    - scanf, fscanf, printf, fprint
  - Binary I/O
    - fread, fwrite

ISO C Standard Library

- The I/Oc standard is "fully buffered"
  - Each I/O is done only when the I/O buffer is full
  - Each "**flush**" operation writes the I/O buffer on the I/O device

```
#include <stdio.h>

void setbuf (FILE *fp, char *buf);

int fflush (FILE *fp);
```

The standard error is never buffered

For concurrent process, use
setbuf (stdout, 0);
fflush (stdout);

POSIX Standard Library

- I/O in UNIX can be entirely performed with only **5** functions
  - open, read, write, lseek, close
- This type of access
  - Is part of POSIX and of the Single UNIX Specification, but not of ISO C
  - It is normally defined with the term "unbuffered I/O", in the sense that each read or write operation corresponds to a system call

System call open()

- In the UNIX kernel a "file descriptor" is a non-negative integer
- Conventionally (also for shells)
  - Standard input
    - 0 = STDIN_FILENO
  - Standard output
    - 1 = STDOUT_FILENO
  - Standard error
    - 2 = STDERR_FILENO

These descriptors are defined in the headers file **unistd.h**

System call open()

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

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

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

- It opens a file defining the permissions
- Return value
  - The descriptor of the file, on success
  - The value -1, on error

System call open()

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

- It can have 2 or 3 parameters
  - The **mode** parameter is optional
- **Path** indicates the file to open
- **Flags** has multiple options
  - Can be obtained with the OR bit-by-bit of constants defined in the header file **fcntl.h**
  - One of the following three constants is mandatory
    - O_RDONLY open for read-only access
    - O_WRONLY open for write-only access
    - O_RDWR open for read-write access

System call open()

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

- Optional constants
  - O_CREAT creates the files if not exist
  - O_EXCL error if O_CREAT is set and the file exists
  - O_TRUNC remove the content of the file
  - O_APPEND append to the file
  - O_SYNC each write waits that the physical write operation is finished before continuing
  - ...

System call open()

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

- **Mode** specifies permission access
  - S_I[RWX]USR rwx --- ---
  - S_I[RWX]GRP --- rwx ---
  - S_I[RWX]OTH --- --- rwx

When a file is created, actual permissions are obtained from the **umask** of the user owner of the **process**

System call read()

```
#include <unistd.h>

int read (int fd, void *buf, size_t nbytes);
```

- Read from file **fd** a number of bytes equal to **nbytes**, storing them in **buf**
- Returned values
  - The number of read bytes, on success
  - The value -1, on error
  - The value 0, in the case of EOF

System call write()

```
#include <unistd.h>

int write (int fd, void *buf, size_t nbytes);
```

- Write **nbytes** bytes from **buf** in the file identified by descriptor **fd**
- Returned values
  - The number of written bytes (i.e., normally **nbytes),** in the case of success
  - The value -1, on error

System call write()

- Remarks
  - Function **write** writes on the system buffer, not on the disk
    - fd = open (file, O_WRONLY | O_SYNC);
  - O_SYNC forces the sync of the buffers, but only for ext2 file systems

```
int write (int fd, void *buf, size_t nbytes);
```

Examples

```
float data[10];

if ( write(fd, data, 10*sizeof(float))==(-1) ) {
  fprintf (stderr, "Error: Write %d).\n", n);
}
```

Write a data array (of float values)

```
struct {
  char name[L];
  int n;
  float avg;
} item;

if ( write(fd,&item,sizeof(item)))==(-1) ) {
  fprintf (stderr, "Error: Write %d).\n", n);
}
```

Write a structured item (with 3 fields) in binary form

System call lseek()

```
#include <unistd.h>

off_t lseek (int fd, off_t offset, int whence);
```

- The current position of the file offset is associated to each file
  - This position indicates the one from which the next read/write operation starts
  - The system call lseek assigns the value **offset** to the file offset

System call lseek()

- Whence specifies the interpretation of offset
  - If whence==SEEK_SET
    - The offset is evaluated from the beginning of the file
  - If whence==SEEK_CUR
    - The offset is evaluated from the current position
  - If whence==SEEK_END
    - The offset is evaluated from the end of the file

The value of **offset** can be positive or negative

It is possible to leave "holes" in a file (filled with zeros)

```
off_t lseek (int fd, off_t offset, int whence);
```

System call lseek()

- Return value
  - New offset, on success
  - -1, on error

```
off_t lseek (int fd, off_t offset, int whence);
```

System call close()

```
#include <unistd.h>

int close (int fd);
```

- It closes the file of descriptor **fd**
  - Notice that, all the open files are closed automatically when the process terminates
- Return value
  - The value 0, on success
  - The value -1, on error

Example

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

#define BUFFSIZE 4096

int main(void) {
  int nR, nW, fdR, fdW;
  char buf[BUFFSIZE];
  fdR = open (argv[1], O_RDONLY);
  fdW = open (argv[2], O_WRONLY | O_CREAT | O_TRUNC,
                       S_IRUSR | S_IWUSR);
  if ( fdR==(-1) || fdW==(-1) ) {
    fprintf (stdout, “Error Opening a File.\n“);
    exit (1);
  }
```

Example

```
while ( (nR = read (fdR, buf, BUFFSIZE)) > 0 ) {
    nW = write (fdW, buf, nR);
    if ( nR!=nW )
      fprintf (stderr,
        "Error: Read %d, Write %d).\n", nR, nW);
  }

  if ( nR < 0 )
    fprintf (stderr, "Write Error.\n");

  close (fdR);
  close (fdW);

  exit(0);
}
```

Error check on the last reading operation

This program works indifferently on text and binary files

File system management

- The POSIX standard provides a set of functions to perform the manipulation of directories
  - The function **stat**
    - Allows to understand the type of "entry" (file, directory, link, etc.)
    - This operation is permitted using the C data structure returned by the function, i.e. **struct stat**

    Returned data structure
  - Some other functions to manage the file system
    - getcwd, chdir
    - mkdir, rmdir
    - opendir, readdir, closedir

    Positioning

    Creation Cancellation

    Visit / Inspection

stat ()

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

int stat (const char *path, struct stat *sb);
int lstat (const char *path, struct stat *sb);
int fstat (int fd, struct stat *sb);
```

Path to return information about

Returned data structure

- The function **stat** returns a reference to the structure **sb** (**struct stat**) for the file (or file descriptor) passed as a parameter
- Return value
  - The value 0, on success
  - The value -1, on error

stat ()

- The function
  - **lstat** returns information about the symbolic link, not the file pointed by the link (when the path is referred to a link)
  - **fstat** returns information about a file already opened (it receives the file descriptor instead of a path)

```
int stat (const char *path, struct stat *sb);
int lstat (const char *path, struct stat *sb);
int fstat (int fd, struct stat *sb);
```

stat ()

```
struct stat {
  mode_t st_mode;       /* file type & mode */
  ino_t st_ino;         /* i-node number */
  dev_t st_dev;         /* device number */
  dev_t st_rdev;        /* device number */
  ...
};
```

- The second argument of **stat** is the pointer to the structure **stat**
- The field **st_mode** encodes the file type

stat ()

```
struct stat {
  mode_t st_mode;        /* file type & mode */
  ino_t st_ino;          /* i-node number */
  dev_t st_dev;          /* device number */
  dev_t st_rdev;         /* device number */
  ...
};
```

- Some macros allow to understand the type of the file
  - **S_ISREG** regular file, **S_ISDIR** directory, **S_ISBLK** block special file, **S_ISCHR** character special file, **S_ISFIFO** FIFO, **S_ISSOCK** socket, **S_ISLNK** symbolic link

Example

Check the directory entry type

```
struct stat buf;
...
if (lstat(argv[i], &buf) < 0) {
  fprintf (stdout, "lstat error.\n");
  exit(1);
}
if       (S_ISREG(buf.st_mode))  ptr = "regular";
else if (S_ISDIR(buf.st_mode))  ptr = "directory";
else if (S_ISCHR(buf.st_mode))  ptr = "char special";
else if (S_ISBLK(buf.st_mode))  ptr = "block special";
else if (S_ISFIFO(buf.st_mode)) ptr = "fifo";
else if (S_ISLNK(buf.st_mode))  ptr = "symbolic link";
else if (S_ISSOCK(buf.st_mode)) ptr = "socket";
     printf("%s\n", ptr);
}
```

Allow to understand if it is a directory !

getcwd () and chdir ()

```
#include <unistd.h>

char *getcwd (char *buf, int size);

int chdir (char *path);
```

Dimension of buf

Get Current Working Directory

Change Directory

- Get (change) the path of the **working directory**
- Returned values
  - getcwd
    - The buffer buf on success; NULL on error
  - chdir
    - 0 on success; -1 on error

Example

How to use getcwd and chdir

```
#define N 100

char name[N];

if (getcwd (name, N) == NULL)
  fprintf (stderr, "getcwd failed.\n");
else
  fprintf (stdout, "dir %s\n", name);

if (chdir(argv[1]) < 0)
  fprintf (stderr, "chdir failed.\n");
else
  fprintf (stdout, "dir changed to %s\n", argv[1]);
```

mkdir () and rmdir ()

```
#include <unistd.h>
#include <sys/stat.h>

int mkdir (const char *path, mode_t mode);

int rmdir (const char *path);
```

See system call open

- **mkdir** creates a new (empty) directory
- **rmdir** deletes a directory (if it is empty)
- Returned values
  - 0 on success
  - -1 on error

opendir (), dirent () and closedir ()

```
#include <dirent.h>

DIR *opendir (
  const char *filename
);

struct dirent *readdir (
  DIR *dp
);

int closedir (
  DIR *dp
);
```

Open a directory for reading
Return value:
The pointer to the directory, on success
The NULL pointer, on error

Proceed with the reading of the directory. Return value:
The pointer to the directory, on success
The NULL pointer, on error or at the end of the reading operation

Terminate the reading
Return value:
0, on success
-1, on error

dirent structure

```
struct dirent {
  inot_t d_no;
  char d_name[NAM_MAX+1];
  ...
}
```

- The structure **dirent** (**DIR ***) returned by **readdir**
  - Has a format that depends on the specific implementation
  - It contains at least the following fields
    - The i-node number
    - The file name (null-terminated)

Example

Structure for lstat

Visit a directory and print its content

```
#define N 100
...
struct stat buf;
DIR *dp;
char fullName[N];
struct dirent *dirp;
int i;
...
if (lstat(argv[1], &buf) < 0 ) {
  fprintf (stderr, "Error.\n"); exit (1);
}
if (S_ISDIR(buf.st_mode) == 0) {
  fprintf (stderr, "Error.\n"); exit (1);
}
if ( (dp = opendir(argv[1])) == NULL) {
  fprintf (stderr, "Error.\n"); exit (1);
}
```

Directory "handle"

Structure for readdir

Ask information about the path in argv[1]

If it is not a directory, the program terminates

Otherwise, the directory is open

Example

```
i = 0;
while ( (dirp = readdir(dp)) != NULL) {
  sprintf (fullName, "%s/%s", argv[1], dirp->d_name);
  if (lstat(fullName, &buf) < 0 ) {
    fprintf (stderr, "Error.\n"); exit (1);
  }
  if (S_ISDIR(buf.st_mode) == 0) {
    fprintf (stdout, "File %d: %s\n", i, fullName);
  } else {
    fprintf (stdout, "Dir  %d: %s\n", i, fullName);
  }
  i++;
}
if (closedir(dp) < 0) {
  fprintf (stderr, "Error.\n"); exit (1);
}
```

Read the directory (iterating over all entries)

Request information about the entry fullName

Display data

Closure and termination