Il File-System

I file in ambiente Linux

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

File System

- Il file-system è uno degli aspetti più visibili di un sistema operativo
- Fornisce i meccanismi per la memorizzazione (permanente) dei dati
- Include la gestione di
  - File
  - Direttori
  - Dischi e partizioni di dischi

I file

- Memorizzano informazioni a lungo termine
  - In maniera indipendente da
    - Terminazione del programma/processo, alimentazione, etc.
- Dal punto di vista logico un file può essere visto come
  - Insieme di informazioni correlate
    - Le informazioni (tutte, i.e., numeri, caratteri, immagini, etc.) sono memorizzate su un dispositivo (elettronico) utilizzando un **sistema di codifica**
  - Spazio di indirizzamento contiguo

Come sono codificate tali informazioni?

Qual è l'organizzazione effettiva di tale spazio?

Codifica ASCII

- De-facto standard
  - ASCII, American Standard Code for Information Interchange
    - Basato originariamente sull'alfabeto inglese
    - Codifica 128 caratteri in 7-bit (numeri binari)
  - Extended ASCII (or high ASCII)
    - Estensione dell'ASCII a 8-bit e 255 caratteri
    - Ne esistono diverse versioni
      - ISO 8859-1 (ISO Latin-1), ISO 8859-2 (Eastern European languages), ISO 8859-5 for Cyrillic languages, etc.

128 caratteri totali
32 non stampabili
96 stampabili

La lingua Klingom non è presente in Extended ASCII

Tabella ASCII Estesa

The ASCII code

American Standard Code for Information Interchange

www.theasciicode.com.ar

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	ò	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	

Codifica Unicode

- Standard industriale
  - Codifica, rappresenta, e permette di manipolare in maniera consistente il testo scritto nella maggioranza delle lingue
    - Prima versione [1991]
      - Possedeva 65,536 codici, con una codifica su 16 bits
    - Versione attualmente utilizzata [2016]
      - Unicode 9.0, ISO/IEC 10646:2014 con emendamenti 1 & 2
      - Codifica 110,187 simboli tra gli 1.1 milioni possibili
      - Copre 100 scripts e simboli multipli inclusi gli emoji

1,114,112 caratteri da 0x000000 a 0x10FFFF

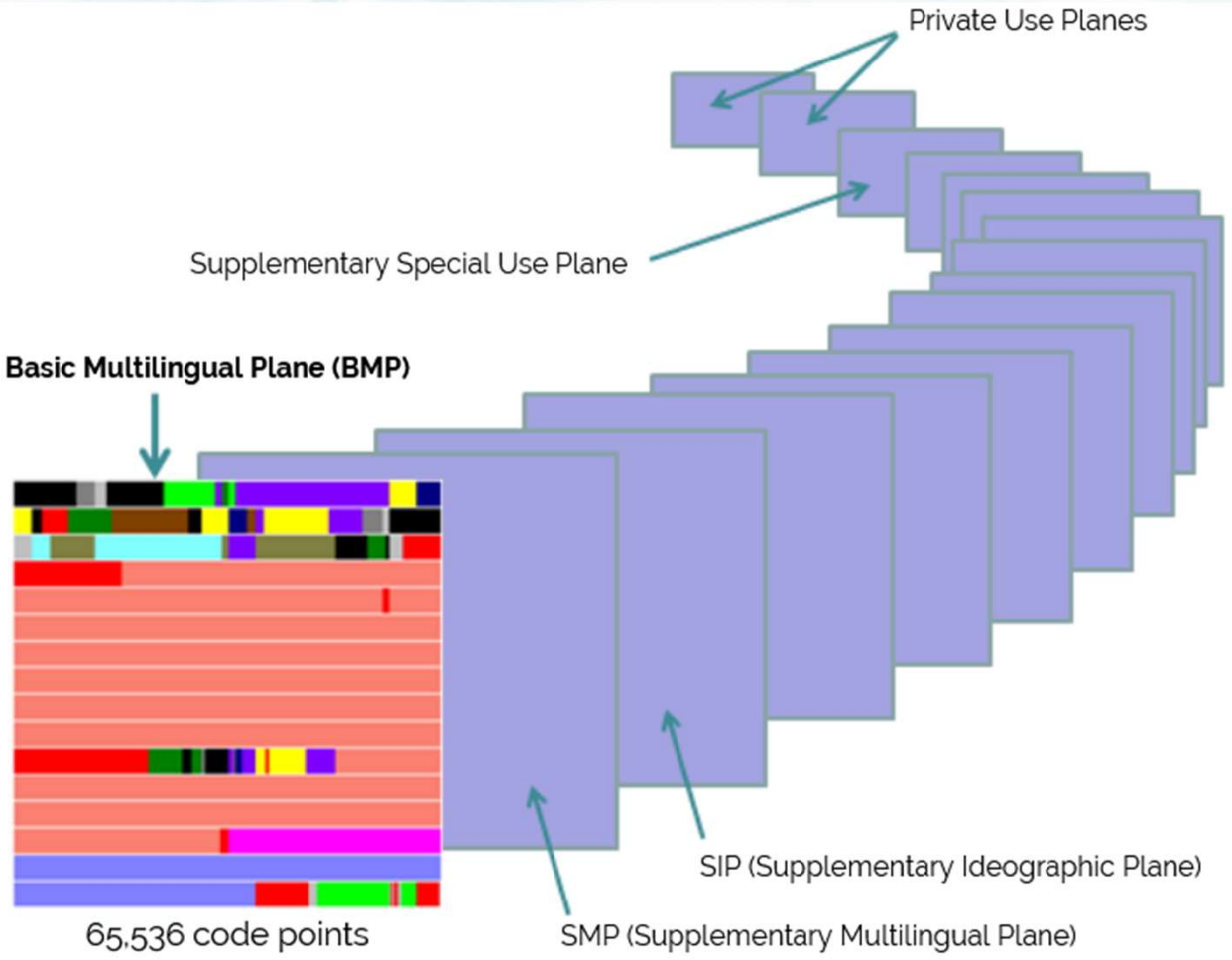
Codifica Unicode

- I simboli unicode sono solitamente memorizzati su 4 byte
  - Memorizzare sempre 4 byte per ciascun carattere è dispendioso, quindi unicode può essere definito con differenti forme di codifica
  - Nel 1994 ISO-C ha standardizzato due forme
    - Caratteri multi-byte
      - UTF-8 e UTF-16
      - Utilizzano da uno a quattro byte
    - Wide characters
      - UTF-32
      - Stessa ampiezza per ogni carattere
      - **Facile** da codificare (fixed-width) ma **inefficient** in termini di spazio

Gli encoding UCS, UTF-1 e UTF-7 sono obsoleti

Codifica Unicode

In UTF-16 il primo insieme di codici include 65,536 posizioni. Questi costituiscono il Basic Multilingual Plane (BMP), i.e., caratteri da 0x0000 a 0xFFFF. Il BMP include la maggiornaze dei caratteri poiù comuni.



I caratteri unicode includono spazio per circa un millione di codici addizionali. Tali codici sono riferiti come caratteri supplementary.

Codifica Unicode

- UTF-8

Il più populare. Utilizzato in oltre il 90% dei website sul World Wide Web cosi come nei più moderni sistemi operativi

- Codifica a 8-bit, di lunghezza **variabile**
- Utilizza da 1 a 4 unità di 8-bit
  - 1 byte representa i caratteri ASCII
  - 2 byte rappresentano caratteri in diversi alfabeti distinti
  - 3 byte per la parte rimanente del BMP
  - 4 byte per i caratteri supplementari
- Per mantenere la conpatibilià a ritroso, i primi 128 caratteri unicode coincidono con quelli ASCII

C fornisce funzioni standard per convertire i vari formati

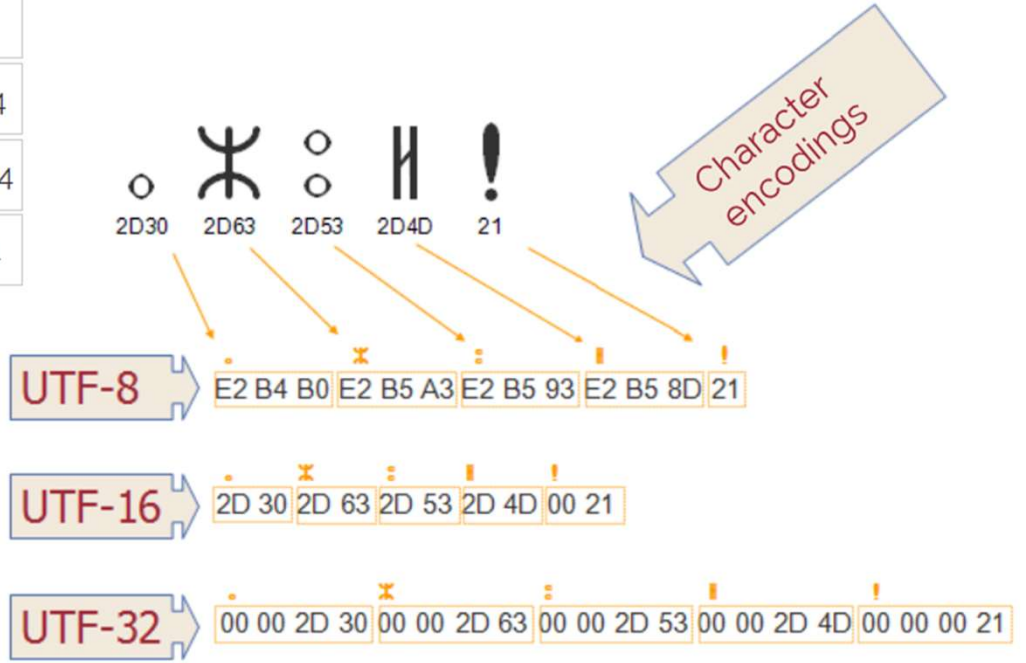
Un esempio

0xC0, 0xC1, 0xF5, 0xFF non possonoo essere codici UTF-8 validi

UTF-8

00000000 – 0000007F	0xxx xxxx			
00000080 – 000007FF	110x xxxx	10xx xxxx		
00000800 – 0000FFFF	1110 xxxx	10xx xxxx	10xx xxxx	
00010000 – 001FFFFF	1110 xxxx	10xx xxxx	10xx xxxx	10xx xxxx

	A	א	好	不
Code point	U+0041	U+05D0	U+597D	U+233B4
UTF-8	41	D7 90	E5 A5 BD	F0 A3 8E B4
UTF-16	00 41	05 D0	59 7D	D8 4C DF B4
UTF-32	00 00 00 41	00 00 05 D0	00 00 59 7D	00 02 33 B4



Problemi

- Anche Unicode presenta alcuni problemi
  - L'ordine dei byte dipende dall'**endianness** dell'architettura che genera il flusso dati

Gli altri byte sono disposti nei byte successivi al primo

- Big Endian
  - Il byte più significativo (the "big end") dei dati è disposto nell'indirizzo più basso
  - 0x12345678 → 12 34 56 78

32 bits

Increasing Adress

- Little Endian
  - Il byte meno significativo (the "little end") dei dati è disposto nel byte più basso
  - 0x12345678 → 78 56 34 12

32 bits

Increasing Adress

Problemi

- Dato un file quale encoding è stato utilizzato per memorizzarlo?
  - Una contro-misura è definite tramite il BOM (Byte Order Mark)
  - Il BOM è un codice speciale (U+FEFF) scritto zll'inizio del file per indicare il tipo di codifica utilizzata per il resto dei dati
  - Indica tanto la codifica UTF quanto la endianess dei dati
  - Sfortunatamente è un carattere opzionale e molte app si erogano il diritto di ometterlo

File di testo e file binari

- Un file è sostanzialmente una serie di byte scritti uno dopo l'altro
  - Ogni byte include 8 bit, il cui valore è 0 oppure 1
  - Quindi di fatto tutti i file sono binari
- Normalmente però si distinguono
  - File di testo (o ASCII)
  - File Binari

Sorgenti C, C++, Java, Perl, etc.

Eseguibili, Word, Excel, etc.

Osservazione:
Il kernel UNIX/Linux non distingue tra file di testo e binari

File di testo (o ASCII)

- File che consiste in dati codificati in ASCII
  - ASCII: numeri su 8 bit, sequenza di 0 e 1
  - Sono però sequenze di 0 e 1 che codificano dei codici ASCII
- I file di testo di solito solo "line-oriented"
  - Newline: spostamento sulla riga successiva
    - UNIX/Linux e Mac OSX
      - Newline = 1 carattere
      - Line Feed (go to next line, LF, 10_{10})
    - Windows
      - Newline = 2 caratteri
      - Line Feed (go to next line, LF, 10_{10})
        + Carriage Return (go to beginning of the line, CR, 13_{10})

Binary Files

- Una sequenza di 0 e 1 non "byte-oriented"
- La più piccolo unità di lettura/scrittura è il bit
  - Difficile la gestione di bit singoli
  - Include ogni possibile sequenza di 8 bit e non necessariamente questi corrispondono a caratteri stampabili, new-line, etc.

Binary Files

- Vantaggi
  - Compattezza (minore dimensione media)
    - Esempio: Il numero intero 100000_{10} occupa 6 caratteri (i.e., 6 byte) in format testuale e 4 byte se codificato su un intero (short)
  - Facilità di modificare il file
    - Un intero occupa sempre lo stesso spazio
  - Facilità di posizionarsi sul file
    - Struttura a record fissi
- Svantaggi
  - Portabilità limitata
  - Impossibilià di utilizzare un editor standard

Esempio

Una stringa in un file testo o binario

"ciao"

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

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

01100011_2 01101001_2 01100100_2 01101111_2

Un numero intero in un file testo

"231"

'2' '3' '1'

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

00110010_2 00110011_2 00110001_2

Un numero intero (di un byte) in un file binario

"231"

"231_{10}"

11100111_2

Example

```
FILE *fp;
int fd;
char sv[] = "This is a string";
int iv = 10;
float fv = 15.55;

fp = fopen ("my_file_1.txt", "w");
fprintf (fp, ...);
fclose (fp);

fd = open ("my_file_1.bin", O_WRONLY|O_CREAT|O_TRUNC,
  S_IRUSR|S_IWUSR);
write (fd, ...);
close (fd);
```

ASCII file

Binary file

Example

ASCII file

```
fprintf (fp, "%s", sv);
fprintf (fp, "%d", iv);
fprintf (fp, "%f", fv);
```

T = 54 hex

```
This is a string
```

```
> hexdump -C my_file_1.txt
00000000  54 68 69 73 20 69 73 20  61 20 73 74 72 69 6e 67
00000010
```

Memory addresses

Binary file

```
write (fd, sv, strlen (sv));
write (fd, &iv, sizeof (int));
write (fd, &fv, sizeof (float));
```

Same content

```
> hexdump -C my_file_1.bin
00000000  54 68 69 73 20 69 73 20  61 20 73 74 72 69 6e 67
00000010
```

Example

ASCII file

```
fprintf (fp, "%s", sv);
fprintf (fp, "%d", iv);
fprintf (fp, "%f", fv);
```

T = 31 hex

10

```
> hexdump -C my_file_2.txt
00000000  31 30
00000002
```

Memory addresses

Binary file

```
write (fd, sv, strlen (sv));
write (fd, &iv, sizeof (int));
write (fd, &fv, sizeof (float));
```

```
> hexdump -C my_file_2.bin
00000000  0a 00 00 00
00000004
```

0000-1010 0000-0000 0000-etc.
Litte endian = Least significant value is stored first

0a = 0000-1010 = one byte

Example

ASCII file

```
fprintf (fp, "%s", sv);
fprintf (fp, "%d", iv);
fprintf (fp, "%f", fv);
```

T = 31 hex

```
15.550000
```

```
> hexdump -C my_file_3.txt
00000000  31 35 2e 35 35 30 30 30  30
00000009
```

Memory addresses

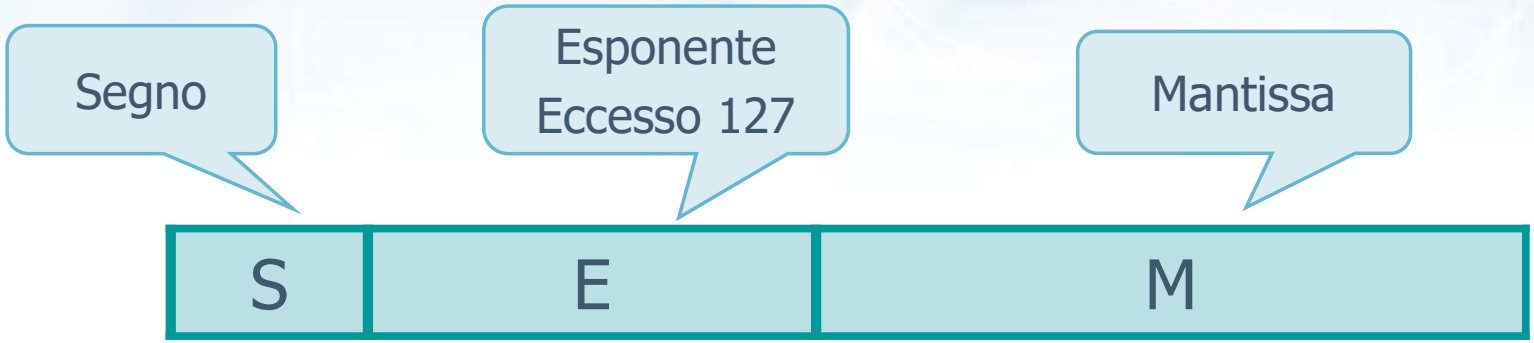
Binary file

```
write (fd, sv, strlen (sv));
write (fd, &iv, sizeof (int));
write (fd, &fv, sizeof (float));
```

The IEEE 754 notation for floating point numbers plus litte endian

```
> hexdump -C my_file_3.bin
00000000  cd cc 78 41
00000004
```

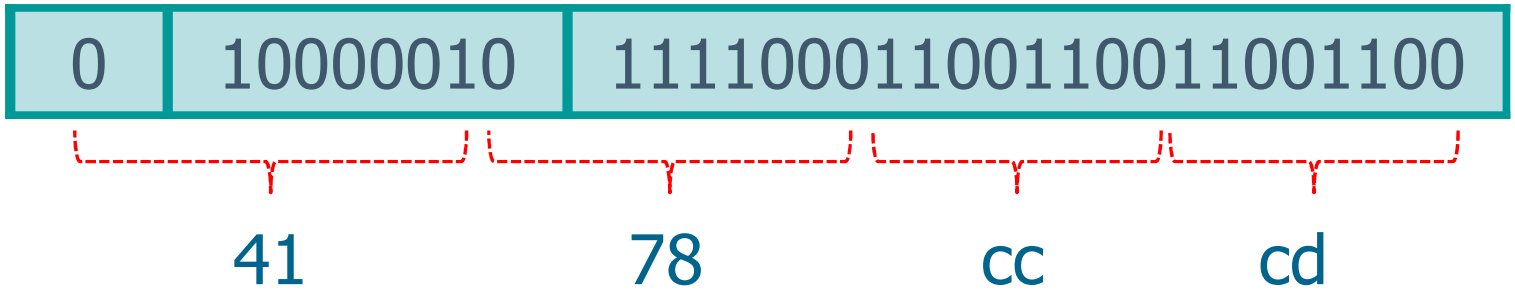
Example



- Segno
  - $N \geq 0 \rightarrow 0$
  - $N \leq 0 \rightarrow 1$
- Esponente eccesso 127
  - $E_{rappresentato} = e_{reale} + 127$
- Mantissa normalizzata in binario puro su 24 bit
  - $M = 1 \ . \ c_{-1} \ c_{-2} \ c_{-3} \ \ldots \ c_{-22} \ c_{-23}$

Example

- $N = 15.55_{10}$
  - $15.0_{10} = 1111_2$
  - $0.55_{10} = 10001100110011 \ldots_2$
  - $15.55_{10} = 1111.10001100110011 \ldots_2$
    $= 1.11110001100110011 \ldots \cdot 2^3$
  - $S = 1$
  - $E = 3+127 = 130 = 10000010_2$
  - $M = 11110001100110011$



Serializzazione

- Processo di traduzione di una struttura (e.g., C struct) in un formato memorizzabile
  - Utilizzando la serializzazione una struttura può essere memorizzata o trasmessa (sulla rete) come un'unica entità
  - Quando la sequenza di bit viene letta lo si fa in accordo con la serializzazione effettuata e la struttura viene ricostruita in maniera identica
- Alcuni linguaggi supportano la serializzazione mediante operazioni di R/W su file
  - Java, Python, Objective-C, Ruby, etc.

Esempio

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

Dump in C della memoria centrale su disco

Testo:
Campi singoli
Caratteri su 8 bit (ASCII)

```
1 100000 Romano Antonio 25

† Romano ÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ
Antonio ÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ

??? †?R?o?m?a?n?o???ÌÌÌÌÌÌÌÌÌÌÌÌ
ÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ
ÌÌA?n?t?o?n?i?o???ÌÌÌÌÌÌÌÌÌÌÌÌ
ÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ
```

Binario:
Ctr su 8 bit (ASCII)

Binario:
Ctr su 16 bit (UNICODE)
N.B. Dimensione file

ISO C Standard Library

- L'I/O ANSI C si può effettuare attraverso diverse categorie di funzioni
  - Un carattere alla volta
  - Una riga alla volta
  - I/O formattato
  - R/W diretto

ISO C Standard Library

- Lo standard I/O è "fully buffered"
  - L'operazione di I/O avviene solo quando il buffer di I/O è pieno
  - L'operazione di "flush" indica la scrittura del buffer su I/O

```
#include <stdio.h>

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

int fflush (FILE *fp);
```

Lo standard error non è mai buffered

Per processi concorrenti, usare
setbuf (stdout, 0);
fflush (stdout);

Apertura e chiusura di un file

```
#include <stdio.h>

FILE *fopen (char *path, char *type);

FILE *fclose (FILE *fp);
```

- Metodi di accesso
  - r, rb, w, wb, a, ab r+, r+b, etc.
  - Il kernel UNIX non differenzia file di testo (ASCII) da file binari
    - "b" durante l'apertura di un file non ha effetto, e.g. "r"=="rb", "w"=="wb", etc.

I/O a caratteri

```
#include <stdio.h>

int getc (FILE *fp);
int fgetc (FILE *fp);

int putc (int c, FILE *fp);
int fputc (int c, FILE *fp);
```

- Valore di ritorno
  - Un carattere in caso di successo
  - EOF in caso di errore oppure fine file
- La funzione
  - **getchar** è equivalente a **getc (stdin)**
  - **putchar** è equivalente a **putc (c, stdout)**

I/O a righe

```
#include <stdio.h>

char gets (char *buf);
char *fgets (char *buf, int n, FILE *fp);

int puts (char *buf);
int *fputs (char *buf, FILE *fp);
```

- Valore di ritorno
  - buf (gets/fgest) o un valore non negative (puts/fputs) in caso di successo
  - NULL (gets/fgets) o EOF (puts/fputs) per errori o fine file
- Occorre le righe siano delimitate dal "new-line"

I/O formattato

```
#include <stdio.h>

int scanf (char format, …);
int fscanf (FILE *fp, char format, …);

int printf (char format, …);
int fprintf (FILE *fp, char format, …);
```

- Elevata duttilità nella manipolazione di dati
  - Formati (caratteri, interi, reali, etc.)
  - Conversioni

I/O binario

```
#include <stdio.h>

size_t fread (void *ptr, size_t size,
  size_t nObj, FILE *fp);

size_t fwrite (void *ptr, size_t size,
  size_t nObj, FILE *fp);
```

- Ogni operazione di I/O (singola) opera su una oggetto aggregato di dimensione specifica
  - Con getc/putc occorrerebbe scorrere tutti i campi della struttura
  - Con gets/puts non sarebbe possibile visto che terminerebbero l'operazione sui byte NULL o i new-line

I/O binario

- Spesso utilizzate per gestire file binari
  - R/W di intere strutture mediante una singola operazione
    - Una fwrite effettua il dump su file della struttura così come essa è memorizzata su file
  - Potenziali problemi nel gestire architetture diverse
    - Compatibilità sul formato dei dati (e.g., interi, reali, etc.)
    - Offset differenti per i campi di una struttura

```
#include <stdio.h>

size_t fread (void *ptr, size_t size,
  size_t nObj, FILE *fp);

size_t fwrite (void *ptr, size_t size,
  size_t nObj, FILE *fp);
```

I/O binario

- Valore di ritorno
  - Numero di oggetti letti/scritti
  - Se il valore di ritorno non corrisponde al parametron nObj
    - Si ha avuto un errore
    - Si è raggiunta la fine del file

Utilizzare ferror e feof per distinguire i due casi

```
#include <stdio.h>

size_t fread (void *ptr, size_t size,
  size_t nObj, FILE *fp);

size_t fwrite (void *ptr, size_t size,
  size_t nObj, FILE *fp);
```

POSIX Standard Library

- L'I/O UNIX si può effettuare interamente attraverso solo **5** funzioni
  - open, read, write, lseek, close
- Tale tipologia di accesso
  - Fa parte di POSIX e della Single UNIX Specification ma non di ISO C
  - Si indica normalmente con il termine di "unbuffered I/O" nel senso che ciascuna operazione di read o write corrisponde a una system call al kernel

System call open ()

- Nel kernel UNIX un "file descriptor" è un intero non negativo
- Per convenzione (anche nelle shell)
  - Standard input
    - 0 = STDIN_FILENO
  - Standard output
    - 1 = STDOUT_FILENO
  - Standard error
    - 2 = STDERR_FILENO

Descrittori definiti nel file di header **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);
```

- Apre un file dato il path, definendone le modalità di accesso e i permessi
- Valore di ritorno
  - Il descrittore del file in caso di successo
  - Il valore -1 in caso di errore

Da controllare **sempre** !

System call open ()

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

- Parametri
  - Può avere 2 oppure 3 parametri
    - Il parametro **mode** è opzionale
  - **Path** indica il file da aprire
  - **Flags** ha molteplici opzioni
    - Si ottiene mediante l'OR bit-a-bit di costanti presenti nel file di header **fcntl.h**
    - Una delle tre seguenti costanti è obbligatoria
      - 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
);
```

- Le seguenti costanti sono invece opzionali
  - O_CREAT crea il file se non esiste
  - O_EXCL errore se O_CREAT è settato e il file esiste
  - O_TRUNC rimuove il contenuto del file
  - O_APPEND appende al file
  - O_SYNC ogni write attende che l'operazione di scrittura fisica sia terminata prima di proseguire
  - ...

System call open ()

- **Mode** specifica i diritti di accesso
  - S_I[RWX]USR rwx --- ---
  - S_I[RWX]GRP --- rwx ---
  - S_I[RWX]OTH --- --- rwx

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

I permessi con cui viene effettivamente creato un file sono modificati dall'**umask** dell'utente proprietario del processo (and dei diretti del processo e del file)

System call read ()

```
#include <unistd.h>

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

- Legge dal file **fd** un numero di byte uguale a **nbytes**, memorizzandoli in **buf**
- Valori di ritorno
  - Il numero di byte letti in caso di successo
  - Il valore -1 in caso di errore
  - Il valore 0 in caso di EOF

System call read ()

- Il valore ritornato è inferiore a **nbytes**
  - Se la fine del file viene raggiunta prima di **nbytes** byte
  - Se la **pipe** da cui si sta leggendo non contiene **nbytes** bytes

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

System call write ()

```
#include <unistd.h>

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

- Scrive **nbytes** byte contenuti in **buf** nel file di descrittore **fd**
- Valori di ritorno
  - Il numero di byte scritti in caso di successo, cioè normalmente **nbytes**
  - Il valore -1 in caso di errore

System call write ()

- Osservazioni
  - write scrive sui buffer di sistema, non sul disco
    - fd = open (file, O_WRONLY | O_SYNC);
  - O_SYNC forza la sincronizzazione dei buffer, ma solo sul file system ext2

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

Esempi: File R/W

```
float data[10];
if (write(fd, data, 10*sizeof(float))==(-1)) {
   fprintf (stderr, "Error on Write\n");
  }
}
```

Scrittura
del vettore data (di float)

```
struct {
  char name[L];
  int n;
  float avg;
} item;
if (write(fd,&item,sizeof(item)))==(-1)) {
   fprintf (stderr, "Error on Write\n");
  }
}
```

Scrittura della struttura item
(con 3 campi)

System call lseek ()

```
#include <unistd.h>

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

- Ogni file ha associata una posizione corrente del file offset
  - Tale posizione indica la posizione di partenza della successiva operazione di read/write
  - La system call lseek assegna un nuovo valore (**offset**) al file offset

System call lseek ()

- Whence specifica l'interpretazione dell'offset
  - Se whence==SEEK_SET
    - L'offset è valutato dall'inizio del file
  - Se whence==SEEK_CUR
    - L'offset è valutato dalla posizione corrente
  - Se whence==SEEK_END
    - L'offset è valutato dalla fine del file

Il valore di **offset** può essere positivo o negativo

È possibile lasciare "buchi" in un file (riempiti con zeri)

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

System call lseek ()

- Valore di ritorno
  - Il nuovo offset, in caso di successo
  - Il valore -1, in caso di errore

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

System call close ()

```
#include <unistd.h>

int close (int fd);
```

- Chiude il file di descrittore **fd**
  - Tutti i file sono chiusi automaticamente quando il processo termina
- Valore di ritorno
  - Il valore 0, in caso di successo
  - Il valore -1, in caso di errore

Esempio: File R/W

```
#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);
  }
```

Esempio: File R/W

```
  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, "Read Error.\n");

  close (fdR);
  close (fdW);

  return (0);
}
```

Controllo errore sull'ultima operazione di lettura

Opera indifferentemente su file di testo e binari