



A.A. 2021-2022



LABORATORIO DI SISTEMI OPERATIVI

Università degli studi di Napoli

Federico II



Valentino Bocchetti



N86003405



vale.bocchetti@studenti.unina.it

Indice

1 Introduzione al corso	25
1.1 Professori	25
1.2 Crediti	25
1.3 Informazioni generali	25
1.4 Esame	25
1.4.1 Modalità di Esame	25
1.4.2 Propedeuticità	26
1.5 Prerequisiti	26
1.6 Obiettivi del corso	26
1.7 Programma di Massima UNIX	26
1.8 Testi di riferimento	26
2 Lezione del 27-09	26
2.1 Il sistema UNIX	26
2.1.1 Cenni storici	26
2.1.2 Il SO UNIX	27
2.1.3 Sistemi multi-utenti	28
2.1.4 Standard	28
3 Lezione del 29-09	29
3.1 UNIX Shell	29
3.2 Ciclo di esecuzione Shell	30
3.3 Shell interattiva	30

3.4	Classi di comandi	30
3.5	Formato dei comandi	31
3.6	Gestione delle directory	31
3.7	File System	31
3.7.1	Caratteristiche	31
3.7.2	File UNIX	32
3.7.3	Organizzazione dei file	32
3.7.4	Directory in UNIX	32
3.7.5	Files Sinonimi	32
3.7.6	Pathname	32
3.7.7	Home directory	32
3.7.8	Pathname relativi	33
3.7.9	File speciali	33
3.7.10	Attributi di un file	33
3.7.11	File system montabile	33
3.7.12	File nascosti (dotfiles)	33
3.7.13	Protezione di un file	34
3.7.14	Identificazione utente	34
3.7.15	Gruppi	34
3.7.16	Permessi	34
3.7.17	Permessi iniziali	34
3.7.18	chmod	34
3.7.19	chown	34

4 Lezione del 01-10	35
4.1 Introduzione a Android	35
4.1.1 Riferimenti	35
4.2 Android Software Developer Kit (SDK)	35
4.3 Android Stack	35
4.3.1 System & user apps	36
4.3.2 Java API framework	36
4.3.3 Android runtime	36
4.3.4 C/C++ libraries	36
4.3.5 Hardware Abstraction Layer (HAL)	37
4.3.6 Linux Kernel	37
4.4 Cos'è un'applicazione Android	37
4.5 Sfide di sviluppo Android	37
4.6 App building blocks	37
4.7 Android SDK	37
4.7.1 ADB - Android Debug Bridge	38
5 Lezione del 04-10	38
5.1 Comando link	38
5.2 Comando rm	38
5.3 Shell expansion	38
5.4 Variabili di shell predefinite	38
5.5 Variabili	39
5.6 Comando echo	39

5.7	Quoting	39
5.8	File standard	39
5.9	Ridirezione Stn Input	40
5.10	Comando cat	40
5.11	Ridirezione Stn Error	40
5.12	Ridirezione	40
5.13	Pipe ()	41
5.14	Command substitution	41
5.15	Metacaratteri	41
5.16	Quoting	41
5.17	Metacaratteri di shell	42
5.18	Comando wc (word count)	42
5.19	Comandi concatenabili	42
5.20	Esercitazione	42
5.21	Comando sort	43
5.22	Comando head & tail	43
5.23	Word splitting	43
6	Lezione del 06-10	43
6.1	Comando grep e le espressioni regolari	43
6.2	Espressioni regolari	44
6.3	Extendend regular expression	44
6.4	Processi Bash	45
6.4.1	Tabella processi	45

6.4.2	Attributi dei processi	45
6.4.3	Comando ps	46
6.4.4	Terminazione di un processo	46
6.4.5	Controllo dei processi	46
6.4.6	Jobs e Processi	47
6.5	Monitoraggio memoria	47
6.6	Script shell	47
6.7	Variabili predefinite negli script	47
6.8	Exit status	47
6.9	Operatori sui comandi	47
6.10	Comando if	48
7	Lezione del 08-10	48
7.1	XML	48
7.2	Documento XML	49
7.2.1	Elementi in un file XML	49
7.2.2	Elementi e Attributi	49
7.2.3	Documenti XML come alberi ordinati	49
7.2.4	Caratteri speciali in XML	50
7.3	Documenti XML ben costruiti	50
7.4	Namespace Syntax	50
7.4.1	Struttura	50
7.4.2	Namespace di default	51
7.5	Definizione dei tipi di documento	51

7.6	Elementi di un app Android	51
7.7	Descrizione file XML - metadati e struttura dell'app (Android Manifest)	52
7.8	Permessi	52
7.9	SDK manifest	52
7.10	Schemi supportati	52
7.11	Application priority stack	53
7.12	Application	53
7.13	Activity	53
7.14	Activity lifecycle	54
7.15	Log	54
8	Lezione del 11-10	55
8.1	Espressioni condizionali	55
8.2	Sostituzione aritmetica con \$((...))	55
8.3	Sostituzione aritmetica	56
8.4	Ciclo while	56
8.5	Ciclo for	56
8.6	Case statement	57
8.7	Script interattivi	57
8.8	Ciclo until	57
8.9	sed e awk	57
8.9.1	sed (Stream Editor)	57
8.9.2	awk	59
8.9.3	Struttura di un programma awk	59

8.9.4	Operatori e Predicati	60
8.9.5	Variabili per awk	61
8.9.6	Awk - array	61
9	Lezione del 13-10	61
9.1	Linguaggio C	61
9.2	Compilazione	61
9.3	Memo	62
9.4	Operatori	62
9.5	Funzioni e prototipi	62
9.6	Esempio 1	63
9.7	Esempio 2	63
9.8	Esempio 3	64
9.9	Esempio 4 - scanf	64
9.10	File in UNIX	64
9.10.1	Inodo	64
9.10.2	Accesso e creazione di file	65
9.11	I/O di basso livello	65
9.12	Descrittori di file	66
9.13	La funzione open	66
9.14	Chiamata di sistema open	66
9.15	I flag di open	67
9.16	I permessi per open	67
9.17	Esempi di open	68

9.18 creat	68
9.19 Implementazione nel kernel	68
9.20 Accesso di un file da più processi	69
9.21 Trattare gli errori	69
9.22 Close	69
9.23 L'offset	69
9.24 lseek	69
10 Lezione del 15-10	70
10.1 Risorse (XML - Java)	70
10.2 Outline	70
10.3 Layout	71
10.4 A simple XML document	72
10.5 Elementi DTD	72
10.6 Attributi DTD	72
10.7 Linking DTD e XML docs	73
10.8 Internal DTD	73
10.9 Schema XML	74
10.10 XPath	74
10.11 Elementi di XPath	75
11 Lezione del 18-10	75
11.1 Comando cut	75
11.2 Comando uniq	75
11.3 Comando find	75

11.4	read	76
11.5	Write	76
11.6	Esempio	77
11.7	Condivisione di file	78
11.8	Accesso ad un file	78
11.9	Programma A	79
11.10	Programma B	79
11.11	Duplicazione di File descriptor	80
11.12	Open e Dup	80
11.13	Ottenere info su File	80
11.14	Struttura stat	80
11.15	Tipo di file	81
11.16	Accesso ai file	82
11.17	Funzione access	82
11.18	chmod fchmod	82
11.19	chown	83
11.20	Lista di Ambiente	83
11.21	Variabili di Ambiente	84
11.22	Layout in memoria di un programma	84
11.23	Esecuzione di un programma	85
12	Lezione del 20-10	85
12.1	Identificazione di Processi	85
12.2	Ottenere pid e ppid	86

12.3 Identificazione dei processi	86
12.4 Creazione di processi	86
12.5 Padri e figli	87
12.6 Creazione di Processi-fork	87
12.7 Fork e File	88
12.8 Fork e file	88
12.9 Ciclo di vita del processo	88
12.10 Terminazione di un processo	88
12.11 Processi zombie	89
12.12 Aspettare un figlio	89
12.13 Stato del figlio	90
12.14 Stato di uscita del Figlio	90
12.15 Azioni del kernel a terminazione di processo	91
12.16 Valori di ritorno a terminazione del processo	91
12.17 Status <code>wait/waitpid</code>	91
12.18 Esecuzione di programmi	91
12.19 La famiglia di system call <code>exec</code>	92
12.20 Esempio di utilizzo di <code>exec1</code>	92
12.21 Esecuzione di Programmi	93
12.22 Chiamata alla system call <code>exec</code>	93
12.23 Famiglia system call <code>exec</code>	93
12.24 Ambiente di un processo	94
12.25 Current working directory e root directory	94
12.26 Race Condition	95

12.27 Funzione vfork()	95
12.28 La funzione system	95
13 Lezione del 22-10	95
13.1 ScrollView	95
13.2 Elementi Grafici	96
13.3 TextView	96
13.4 EditText	96
13.5 Button	96
13.6 Parametri di layout	97
13.7 Assegnare UI	97
13.8 findViewById	97
13.9 Listener	98
13.10 Constraint Layout	98
13.11 Intent	98
13.12 Intent-filter	98
13.13 Intent implicito ed esplicito	99
13.13.1 <code>getAction()</code>	99
13.14 Passaggio di Parametri	100
14 Lezione del 25-10	100
14.1 Esercitazione 1	100
14.2 Esercitazione 2	101
15 Lezione del 27-10	101
15.1 Segnali ed allarmi	102

15.2 Caratteristiche dei Segnali	102
15.3 Generazione di segnali	102
15.4 Azioni associate a segnali	103
15.5 Alcuni segnali	103
15.6 Catturare i segnali	103
15.7 Esempio	104
15.8 Inviare segnali	105
15.9 Inviare segnali da shell (<code>kill</code>)	105
15.10 Inviare segnali in C	105
15.11 Impostare una sveglia	106
15.12 Esempio	106
15.13 Insieme di segnali	107
15.14 Signal mask (<code>sigprocmask</code>)	107
16 Lezione del 29-10	108
16.1 Fragment	108
16.2 Ciclo di vita del fragment	109
16.3 UI con Fragments	109
16.3.1 UI con Fragments - Inflate	109
16.4 Aggiungere Fragment all'Activity	110
16.5 Fragment a RunTime	110
17 Lezione del 03-11	111
17.1 IPC - Pipe e FIFO	111
17.2 Pipe	111

17.2.1	La funzione pipe	112
17.2.2	Utilizzo della pipe	112
17.2.3	Leggere e scrivere sulle pipe	113
17.2.4	Pipe tra 2 programmi	114
17.2.5	<code>popen</code> e <code>pclose</code>	115
17.3	FIFO (named pipe)	116
17.3.1	<code>mkfifo</code>	117
17.3.2	FIFO - open e write	117
17.3.3	Operazioni e modalità	118
18	Lezione del 05-11	118
18.1	Livelli di visibilità	118
18.2	Shared Preferences	119
18.3	Accesso a dati su file	119
18.3.1	Salvare dati su un file	119
18.3.2	Recuperare dati da un file	119
18.3.3	Accesso a un file statico	120
18.3.4	Cache files	120
18.3.5	File su memoria esterna	120
18.3.6	Cartelle pubbliche	120
18.3.7	Controllare lo stato	121
18.4	Database (<code>android.database.sqlite</code>)	121
18.4.1	Classe Contenitore (<code>Contract</code>)	121
18.4.2	Creazione del DB	121

18.4.3 DB Helper (<code>android.database.sqlite.SQLiteOpenHelper</code>)	122
18.4.4 Creazione classe Helper	122
18.4.5 Helper <code>onCreate</code> e <code>onUpgrade</code>	122
18.4.6 Inserimento nel DB	123
18.4.7 Leggere dal DB	123
18.4.8 Il cursor	124
19 Lezione del 08-11	124
19.1 I Thread	124
19.2 Processo Multi-Thread	125
19.3 Identificare i Thread	126
19.4 Creazione dei thread	126
19.5 Funzione <code>pthread_create</code>	127
19.6 Creare un thread	127
19.7 Esempio	127
19.8 Trattare gli errori	128
19.9 Risorse condivise	128
19.10 Terminazione di un thread	129
19.11 Aspettare la terminazione di un thread	129
19.12 Esempio - <code>create, join</code>	130
19.13 Condivisione memoria	130
19.14 Variabili globali	131
19.15 Esempio - variabili globali, locali, allocazione dinamica	131
19.16 Cancellare un thread	132

19.17 La funzione <code>pthread_detach</code>	132
19.18 Thread e fork	133
19.19 Thread e segnali	133
19.20 Inviare un segnale a un thread	133
19.21 Attributi di un thread	133
19.22 Gestione Attributi	134
19.23 Detached State	134
19.24 Impostare detached state	134
19.25 Esempio	134
19.26 Cancellabilità	135
19.27 Impostare la cancellabilità	135
20 Lezione del 10-11	135
21 Lezione del 12-11	135
21.1 Sensori HW	135
21.2 Sensori disponibili e simulati	136
21.3 Framework	136
21.4 Utilizzare i sensori	136
21.5 Sensor Service	137
21.6 Tipi di sensori	137
21.7 Sensor	137
21.8 SensorEventListener	137
21.9 registerListener	138
21.10 SensorEvent	138

21.11 Registrare un suono	138
21.12 GPS	139
21.13 Text to Speech (TTS)	139
22 Lezione del 15-11	139
22.1 Sincronizzazione di thread POSIX	139
22.2 Sincronizzazione	140
22.3 Esempio	140
22.4 Mutex POSIX	140
22.5 I mutex	141
22.6 Inizializzare e distruggere un mutex	141
22.7 Usare i mutex	141
22.8 Esempio	142
22.9 Sincronizzazione	142
22.10 Attributi mutex	143
22.11 Tipologie di mutex	143
22.12 Lock per tipologia	144
22.13 Unlock per tipologia	144
22.14 Attributi mutex	144
22.15 Deadlock	145
22.16 Limiti dei mutex	145
22.17 Variabili di condizione	145
22.18 Inizializzare e distruggere una condition variable	146
22.19 Inizializzazione	146

22.20 Usare una condition variable	147
22.21 Attendere una condition variable	147
22.22 Inizializzare e distruggere una condition variable	147
22.22.1 Segnalazione	147
22.23 Timed Wait & Broadcast	148
22.24 Attributi	148
22.25 Dati privati di un thread	149
22.26 Thread Specific Data e Chiavi	149
22.27 Funzioni per TSD	149
22.28 Creare una chiave	149
22.29 Usare una chiave	150
23 Lezione del 17-11	150
23.1 I socket	150
23.2 API per IPC	150
23.3 Socket e descrittori	151
23.4 Domini e Stili di Comunicazione	151
23.4.1 Dominio	151
23.4.2 Stili di comunicazione	151
23.5 Indirizzamento	152
23.5.1 Indirizzi	152
23.6 Comunicazione Client-Server	153
23.7 Server	154
23.8 Funzione socket	154

23.8.1 Valori e Tipi	154
23.8.2 Creare un socket locale	155
23.8.3 Funzione bind	155
23.9 Indirizzi locali	156
23.10 Funzione listen	156
23.11 Funzione accept	156
23.12 Struttura di un server	157
23.13 Connettersi ad un server	157
23.14 Funzione close	157
23.15 Struttura di un client	158
23.16 Leggere da una socket	158
23.17 Scrivere su una socket	158
23.18 Indirizzi TCP/IP	159
23.19 Ordine byte	160
23.20 Specificare indirizzi IP	161
23.21 Impostare indirizzo	161
23.22 Indirizzi TCP/IP per il server	161
23.23 Indirizzi TCP/IP per il client	162
23.24 Schema della connessione	162
23.25 Trasmissione dati	162
23.26 Ricezioni dati	163
23.27 Leggere e scrivere su socket	163
24 Lezione del 19-11	163

24.1 Presentazione del progetto	163
25 Lezione del 22-11	164
25.1 send() e sendto()	164
25.2 recv() e recvfrom()	164
25.3 Comunicazione TCP/IP	165
25.3.1 Apertura Comunicazione TCP/IP	165
25.3.2 Chiusura comunicazione	166
25.4 Server a Programmazione Concorrente	166
25.4.1 Coda di Connessione	167
25.5 Opzioni Socket	167
25.6 Opzione SO_REUSEADDR	168
25.7 Opzioni SO_SNDTIMEO, SO_RCVTIMEO	168
25.8 Nomi di dominio	168
25.8.1 Stampare nome dominio	169
25.9 Risoluzione indirizzi	169
25.10 Lookup.c	170
26 Lezione del 24-11	170
26.1 Formato dei dati	170
26.2 Lettura dei dati	171
26.3 Scrittura dati	171
26.4 Buffer output	171
26.5 Scrittura completa	171
26.6 Lettura completa	172

26.7 Comunicazione senza connessione socket UDP	172
26.7.1 Funzione sendto()	173
26.7.2 Funzione recvfrom()	173
26.7.3 Uso di connect()	174
26.7.4 Comunicazione senza connessioni	174
26.7.5 Funzioni bloccanti	175
26.8 Select	176
26.8.1 Parametri Select	176
26.8.2 Timeout	176
26.8.3 Impostazione degli insiemi	177
27 Lezione del 26-11	177
27.1 Classe contenitore (Contract)	177
27.2 Room	178
27.2.1 Creazione di un database Room	178
27.2.2 Room - Entity	178
27.2.3 Room - DAO	179
27.2.4 Room - Database	179
27.3 Content Provider	180
27.3.1 Accedere ai dati	180
27.3.2 Content Provider nel Manifest	180
27.3.3 Content URI	181
27.3.4 Creazione dei Content Provider	181
27.3.5 Interrogare Il Content Provider con ContentResolver	181

28 Lezione del 29-11	182
29 Lezione del 01-12	182
29.1 AsyncTask (deprecati)	182
29.1.1 Esempio	182
29.2 Modificare la UI - Handler	182
29.2.1 Handler	183
29.2.2 Metodi Handler Runnable	183
29.2.3 Metodi Handler Messaggi	183
29.2.4 Esempio	183
29.3 Service	184
29.3.1 Tipi di servizi	184
29.3.2 Creazione di un service	184
29.3.3 Ciclo di vita	185
29.3.4 Creazione di un Service	185
30 Lezione del 06-12	185
30.1 Servizi Started	185
30.2 IntentService	185
30.2.1 Esempio	186
30.2.2 Metodi	186
30.3 Bound Service	186
30.3.1 Avviare un bound service	187
30.3.2 Scollegarsi da un bound service	187
30.4 IBinder onBind(Intent intent)	187

30.4.1 Classe Binder	188
30.5 Usare Messenger	188
30.6 Ciclo di vita di un bound service	189
30.7 Service management	189
31 Lezione del 10-12	189
32 Lezione del 13-12	190
32.1 Architettura Client Server	190
32.2 Protocolli di comunicazione	190
32.3 Socket	190
32.3.1 Socket client e server	191
32.3.2 Leggere da una Socket	191
32.3.3 Scrivere da una Socket	191
32.3.4 Esempio	191
32.4 HTTP (HyperText Transfer Protocol)	193
32.4.1 Flusso di utilizzo	193
32.4.2 Esempio	193
32.4.3 Gestione della risposta	193
32.4.4 Richieste POST	193
32.4.5 Performance	194
32.5 Android Design Support Library	194
32.5.1 Nel build grandle	194
32.5.2 Navigation View	194
33 Lezione del 15-12	195

33.1 ListView (deprecati in favore di RecycleView)	195
33.1.1 Esempio - CursorAdapter	195
33.1.2 Visualizzazione custom	196
33.2 ActionBar	196
33.2.1 onOptionsItemSelected()	196
33.2.2 Affordance di navigazione	197
33.2.3 ToolBar	197

1 Introduzione al corso

1.1 Professori

- ▶ Giovanni Scala → UNIX;
 - ◊ Ricevimento Mercoledì 14:00-16:00 (inviare una mail 24 ore prima).
 - ◊ Il ricevimento verrà effettuato per il momento su Teams.
- ▶ Francesco Cutugno → Android.

1.2 Crediti

4 + 4 CFU (UNIX + Android).

Lunedì e Mercoledì → UNIX;

Venerdì → Android.

1.3 Informazioni generali

Le registrazioni vengono effettuate dal professore in caso di necessità.

1.4 Esame

In presenza, eccetto casi certificati (COVID - studente fragile).

- ▶ Prova scritta su scripting e System Programming;
- ▶ Progetto Linux (Architettura Client-Server);
- ▶ Progetto Android.

1.4.1 Modalità di Esame

Il progetto consiste in:

- ▶ Realizzazione di un SW con allegata relazione (assegnato a metà corso);
- ▶ Discussione sul progetto.

Gruppi di max 2 studenti (1 in caso di studenti-lavoratori).

Tutti i membri del gruppo discutono il progetto, l'esame può essere svolto in 2 momenti separati.

Il progetto va consegnato prima del progetto.

1.4.2 Propedeuticità

- ▶ SO1;
- ▶ Algebra;

In ogni caso fare riferimento alla guida dello studente.

1.5 Prerequisiti

- ▶ SO1;
- ▶ Programmazione II;
- ▶ Linguaggio C.

1.6 Obiettivi del corso

Strumenti e metodologie per la gestione del Sistema e per lo sviluppo di applicazioni in ambiente UNIX (e Android):

- ▶ Gestione del SO;
- ▶ Programmazione C + scripting;
- ▶ Android.

1.7 Programma di Massima UNIX

- ▶ Comandi Linux
 - ◊ Gestione di file e directory, processi, compilazione dei programmi, . . .;
- ▶ Shell scripting;
- ▶ Programmazione avanzata in C in UNIX

1.8 Testi di riferimento

Advanced Programming in the UNIX environment (Almeno la seconda edizione).

Expert C Programming.

2 Lezione del 27-09

2.1 Il sistema UNIX

2.1.1 Cenni storici

Il SO è lo strato SW che controlla le risorse HW e fornisce l'ambiente nel quale vengono eseguiti i programmi.

In senso più largo si considera parte del SO anche altro SW.

Tutti i SO forniscono servizi.

- ▶ 1965 → Bell Laboratory + MIT lavorano assieme per la creazione di un nuovo SO: **MULTICS** (Multiplexer Information and Computing Service).
- ▶ 1969 → AT&T abbandona il progetto, ma Thompson, Ritchie, Canaday e Cllroy progettano e implementano la prima versione del FS UNIX insieme ad alcune utility.
 - ◊ Il nome UNIX indica Uniplexed Information and Computing Service.
- ▶ 1973 → UNIX viene riscritto in C (grazie a Dennis Ritchie). Caratteristiche:
 - ◊ Architettura semplice ed elegante;
 - ◊ Ambiente di programmazione (in C);
 - ◊ Indipendenza dall'HW.

In particolare, SO basato su **kernel**:

- ▶ Il nucleo del SO è l'unica porzione che deve essere adattata all'HW;
- ▶ tutte le altre funzioni poggiano sul nucleo (indipendenti dall'HW).

2.1.2 Il SO UNIX

Layer:

- ▶ Kernel;
 - ◊ System calls (vengono rimappate in funzioni definite dentro opportune librerie);
 - Shell;
 - Library routines;
 - Applicazioni

Il kernel gestisce le risorse essenziali:

- ▶ CPU;
- ▶ Memoria;
- ▶ Periferiche.

Il kernel è l'unico programma ad essere eseguito in modalità privilegiata, con il completo accesso HW.

Gli altri programmi vengono eseguiti in modalità protetta.

Il kernel si occupa:

- ▶ Gestione della CPU;
- ▶ Gestione della memoria (memoria virtuale → consente di assegnare ad ogni processo uno spazio di indirizzi "virtuale" che il kernel rimappa automaticamente sulla memoria disponibile).
- ▶ Periferiche → vengono viste attraverso un'interfaccia astratta che permette di trattarle come fossero file, secondo il concetto per cui "everything is a file".

2.1.3 Sistemi multi-utenti

Il kernel **Linux/UNIX** nasce fin dall'inizio come sistema multutente.

È un sistema con la presenza di utente privilegiato **root** (con UID 0) e X utenti non privilegiati.

2.1.4 Standard

SUS (Single UNIX Specification)

- ▶ Basato su specifiche precedenti dell'IEEE e di The Open Group;
- ▶ Famiglia di standard per SO;
- ▶ Necessaria per trademark UNIX.

Portable Operating System Interface (POSIX)

- ▶ IEEE 1003 e ISO/IEC 9945
- ▶ Famiglia di standard per SO;
- ▶ Interfaccia di SO portabile per UNIX;
- ▶ Alcuni sistemi unix based sono POSIX-certified (es MacOS X).

Standard:

- ▶ Base definition volume → Termini generali, concetti e interfacce;
- ▶ Systems Interfaces Volume → Definizione di funzioni di servizi di sistemi e sotto-routine;
- ▶ Shell and Utilities Volume → Definizione di interfacce di ogni applicazione per la shell di comandi;
- ▶ Rationale Volume.

3 Lezione del 29-09

3.1 UNIX Shell

È l'interprete dei comandi. Si pone tra utente e SO. Nei sistemi UNIX qualsiasi operazione può essere eseguita da una sequenza di comandi shell

Tra le shell più utilizzate ricordiamo:

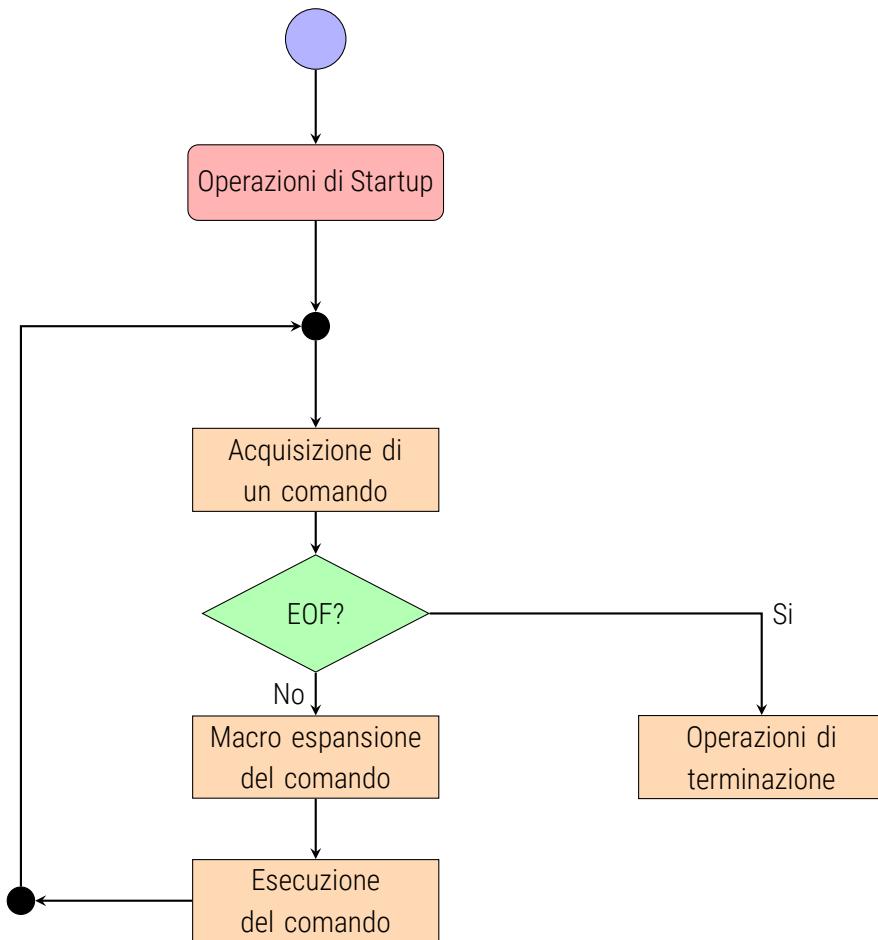
Bourne shell	<code>/bin/sh</code>	(Bell Lab)
Bourne-again shell	<code>/bin/bash</code>	(Linux)
Cshell	<code>/bin/csh</code>	(Berkeley)
Korn shell	<code>/bin/ksh</code>	(Bell Labs)
TENEX C Shell	<code>/bin/tcsh</code>	(BBN tech)

Il file `/etc/shell` contiene l'elenco delle shell installate dall'amministratore e disponibili a tutti gli utenti

È il più delle volte eseguito in maniera interattiva. Presenta:

- ▶ Gestione del "main command loop";
- ▶ Analisi sintattica;
- ▶ Esecuzione di comandi ("built-in") e programmi in linguaggio di shell (script);
- ▶ Gestione dello standard I/O e dello standard error `stderr`;
- ▶ Gestione dei processi da terminale

3.2 Ciclo di esecuzione Shell



3.3 Shell interattiva

- ▶ Comunicazione tra utente e shell avviene tramite comandi o script;
- ▶ Nome comando built-in;
- ▶ Nome di un file eseguibile;
- ▶ Nome di script, cioè un file ASCII presente nel sistema dotato di permessi di esecuzione.

3.4 Classi di comandi

- ▶ Reperire informazioni su comandi, file, . . .;
- ▶ Gestire FileSystem;
- ▶ Operare su file e directory;
- ▶ Elaborare testi;
- ▶ Sviluppare software;

- ▶ Operare in remoto;
- ▶ Amministrare in remoto
 - ◊ Utenti e gruppi;
 - ◊ Dispositivi;
 - ◊ Software.

3.5 Formato dei comandi

Gli argomenti possono essere:

- ▶ Opzioni o flag (-);
- ▶ Parametri

Sono separati da almeno un separatore (di default il carattere spazio)

L'ordine delle opzioni in genere è irrilevante;

L'ordine dei parametri è in genere rilevante.

NB UNIX è CASE SENSITIVE.

3.6 Gestione delle directory

- ▶ `mkdir` → crea una directory;
- ▶ `rmdir` → rimuove una directory vuota;
- ▶ `pwd` → stampa a video la directory corrente;
- ▶ `du` → informazioni sullo spazio occupato di una determinata cartella.

3.7 File System

3.7.1 Caratteristiche

- ▶ Struttura gerarchica;
- ▶ File senza struttura (byte stream);
- ▶ Protezione da accessi non autorizzati;
- ▶ Semplicità di struttura.

On a UNIX system, everything is a file; if something is not a file, it is a process.

3.7.2 File UNIX

Tipi principali:

- ▶ File ordinari;
- ▶ Directory;
- ▶ File speciali

Il sistema assegna biunivocamente a ciascun file un identificatore numerico, detto *i-number* (index number), che gli permette di rintracciarlo nel FS.

File ordinari

Sono sequenze di byte (byte stream). Possono contenere qualsiasi informazione possibile. Il sistema non ne impone struttura specifica.

3.7.3 Organizzazione dei file

UNIX permette di raggruppare i file in cartelle, dette **Directories**, organizzate in una unica struttura gerarchica.

3.7.4 Directory in UNIX

Sequenze di bytes come i file ordinari.

Almeno 2 entry:

- ▶ La directory stessa indicata con .
- ▶ La directory padre indicata con ..

3.7.5 Files Sinonimi

Un file può avere più filename (ma sempre un solo **i-name**).

3.7.6 Pathname

Ogni file viene identificato univocamente specificando il suo pathname.

3.7.7 Home directory

Solitamente ad ogni utente viene assegnata dall'utente root una directory proprietaria che ha come nome lo username del proprietario.

Per denotare la propria home directory si può utilizzare il simbolo ~

3.7.8 Pathname relativi

Ogni file può essere identificato univocamente specificando solamente il suo pathname relativo alla working directory.

3.7.9 File speciali

Ogni device di I/O viene visto, a tutti gli effetti come un file (**file speciale**)

Richieste di lettura/scrittura da/a files speciali causano operazioni di I/O

Vantaggi:

- ▶ Trattamento uniforme di files e devices.
- ▶ In UNIX i programmi non sanno se operano su un file o su un device.

3.7.10 Attributi di un file

Per ogni file UNIX mantiene le seguenti informazioni:

- ▶ Tipo;
- ▶ Posizione;
- ▶ Dimensione;
- ▶ Numero di links;
- ▶ Proprietario;
- ▶ Permessi;
- ▶ Creazione;
- ▶ Modifica;
- ▶ Accesso.

3.7.11 File system montabile

Un FS UNIX è sempre unico, ma può avere parti residenti su device rimovibili:

- ▶ Montare prima di potervi accedere;
- ▶ Smontare prima di rimuovere il supporto.

3.7.12 File nascosti (dotfiles)

Sono tutti quei file che iniziano con un punto. Di default su File Manager e comandi come `ls` (senza flags) non sono visibili.

3.7.13 Protezione di un file

A ciascun file sono associati alcuni attributi:

- ▶ Proprietario;
- ▶ Gruppo;
- ▶ Permessi.

Questi 3 attributi, sono assegnati dal sistema al file al momento della sua creazione;

Il proprietario può successivamente modificare tali attributi con appositi comandi (`chown`, `chgrp` e `chmod`).

3.7.14 Identificazione utente

Ogni utente viene identificato da uno `username` assegnato dall'amministratore del sistema. Ad esso corrisponde biunivocamente uno `userid` numerico, assegnato dal sistema.

Username e user-id sono **pubblici**.

3.7.15 Gruppi

Ogni utente può far parte di uno o più gruppi, definiti dall'amministratore di sistema.

Ogni gruppo è identificato da un group-name di al più 8 caratteri, associato biunivocamente a un group-id numerico.

3.7.16 Permessi

Permissions	Links	Size	User	Group	
<code>drwx-----</code>	27	-	valentino	users	Directory (d), Lettura (r)
<code>drwxr-xr-x</code>	3	-	root	root	Scrittura (w), Esecuzione (x) Links (l)

3.7.17 Permessi iniziali

File ordinari non eseguibili → 666;

File ordinari eseguibili e directory → 777;

3.7.18 `chmod`

Attribuisce i permessi a filename e directory, attraverso notazioni in ottale o parametri.

3.7.19 `chown`

Consente di modificare il proprietario e/o gruppo primario per 1 o più file.

4 Lezione del 01-10

4.1 Introduzione a Android

Android è un' ecosistema, un progetto open-source nato nel 2009, dall'intento comune di aziende (come Google) di progettare un Sistema operativo.

Si basa su un kernel Linux.

Presenta una User interface su touch screen (tecnologia che si appoggia su brevetti proprietari HTC):

- ▶ Gesture come swiping, tapping, pinching;
- ▶ Tastiera virtuale per i caratteri, numeri ed emoji;
- ▶ Supporto per Bluetooth, Controller USB e periferiche.

Presenta un parco di sensori sempre più ampio (molti dei quali embedded).

4.1.1 Riferimenti

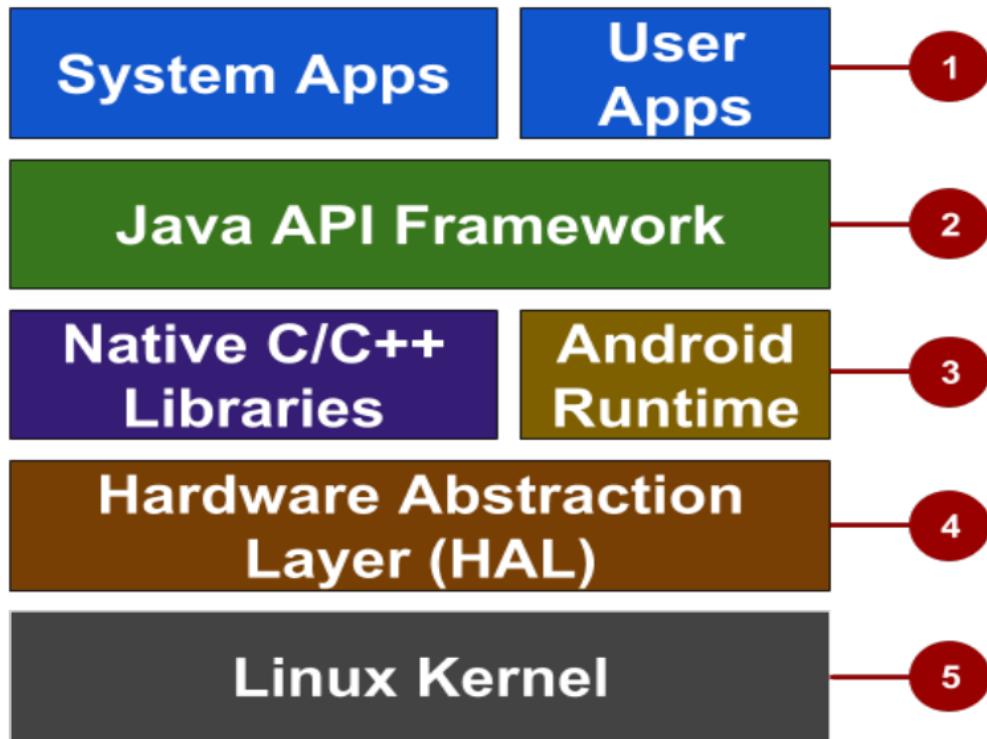
- ▶ [Android](#)
- ▶ Professional Android - Ed. Wrox;
- ▶ Android 9 - Massimo Carli

4.2 Android Software Developer Kit (SDK)

- ▶ Tool di development (debugger, monitor, editor);
- ▶ Librerie:
- ▶ Emulatori;
- ▶ Documentazione (developers.android.com);
- ▶ Codice sample.

4.3 Android Stack

1. App di sistema e di utente;
2. Android OS API in Java Framework;
3. Expose native APIs run apps;
4. Expose device hardware capabilities;
5. Linux Kernel.



4.3.1 System & user apps

Le app di sistema non hanno uno status speciale (app a tenuta stagna → 2 app non possono colloquiare internamente tra loro).

Le app di sistema offrono key capabilities per i developer.

4.3.2 Java API framework

Tutto il set di feature del SO Android è disponibile attraverso le API scritte in JAVA:

- ▶ Gerarchia di classi per la creazione della UI;
- ▶ Manager delle notifiche;
- ▶ Manager delle activity per il ciclo di vita e navigazione

4.3.3 Android runtime

È lo spazio in cui vengono gestiti i run delle single app. È possibile generare solo una singola istanza per app.

4.3.4 C/C++ libraries

Danno accesso al alle componenti e i servizi di sistema di Android.

4.3.5 Hardware Abstraction Layer (HAL)

Interfaccia standard che espone le capacità hardware del device come librerie.

4.3.6 Linux Kernel

- ▶ Threading e low-level memory-management;
- ▶ Security features;
- ▶ Drivers.

4.4 Cos'è un'applicazione Android

- ▶ Vengono scritti in Java e XML (meta-linguaggio);
- ▶ Usano l'Android Software Development Kit (SDK);
- ▶ Usano librerie Android e l'Application framework Android;
- ▶ Eseguite dall'ART (Android Runtime Virtual Machine).

Il manifest di Android è scritto in XML (il più delle volte si è obbligati a editarlo manualmente, senza far uso dall'interfaccia grafica).

4.5 Sfide di sviluppo Android

A differenza di IOS, Android presenta una serie di sfide da gestire:

- ▶ Una molteplicità di devices su cui girare con dimensioni e risoluzioni schermo molto diverse tra loro;
- ▶ Performance → bisogna rendere le app responsive;
- ▶ Compatibilità → deve girar bene anche su vecchie versioni;
- ▶ Marketing → conoscere il mercato e il proprio user.

4.6 App building blocks

- ▶ Risorse → layouts, immagini, strings, colori in XML e file media;
- ▶ Componenti → activities, servizi e classi di aiuto in codice Java;
- ▶ Manifest → informazioni per app per il runtime;
- ▶ Build Configuration → versione APK nel config file di Grandle.

4.7 Android SDK

Set completo per lo sviluppo in ambiente Android:

- ▶ Tools → componenti di sviluppo e debug;
- ▶ Platform Tools → componenti per comunicare con il sistema Android (adb);
- ▶ API → comprende le API di ogni versione Android.

4.7.1 ADB - Android Debug Bridge

Consente la comunicazione con il sistema Android.

Costituito da 3 componenti:

- ▶ Client → invia i comandi, eseguito sulla macchina che sviluppa;
- ▶ Daemon → è sul dispositivo Android dove girerà l'applicazione;
- ▶ Server → Gestisce le comunicazioni tra daemon e client.

5 Lezione del 04-10

5.1 Comando `link`

Associa il nuovo nome (link) al file esistente, che non può essere una directory es:

```
ln name1 name2
```

Nel caso in cui fosse una directory avremo → `name2/name1`

Il comando `ls` permette di visualizzare anche il tipo (se sia un link o meno) con il flag `-l`

È possibile eseguire link simbolici (invece degli hard-link) con il flag `-s`.

Per forzare il linking (andando quindi a eliminare un eventuale file pre-esistente) si usa il flag `-f`

5.2 Comando `rm`

Permette la rimozione dei file indicato. Se il file indicato è una directory è necessario utilizzare il flag `-r` (ricorsivo); In caso contrario otteniamo un errore.

5.3 Shell expansion

Elenco di tutte le espansioni possibili nella shell → [Manuale Bash](#)

5.4 Variabili di shell predefinite

Esistono delle variabili di shell predefiniti (le variabili di ambiente).

Per listare le variabili è possibile utilizzare il comando `env` o `printenv`. Nel caso si conosca il nome di una variabile di ambiente è possibile ottenerne il valore con il comando:

```
echo $VARIABILE
```

Tra le variabili di ambiente ricordiamo:

- ▶ `$HOME` → indica la home dell'utente;
- ▶ `$PATH` → indica il path di ricerca degli eseguibili;
- ▶ `$PS1` → indica la stringa di prompt (default `$` per l'utente normale e `#` per l'utente root);
- ▶ `$HOSTNAME` → nome del PC;
- ▶ `$USER` → nome dell'utente.

5.5 Variabili

`Scrittura/definizione` → `a=3` (senza spazi).

`Lettura` → `${a}` (preferibile rispetto al semplice `$a` per evitare il fenomeno di globbing).

5.6 Comando echo

Stampa a video una linea di testo

5.7 Quoting

È importante notare che in base al tipo di apici comporta un diverso comportamento → con i singoli apici è possibile ignorare alcuni dei meta-caratteri (ad es il `$`).

5.8 File standard

Normalmente, un programma (comando) opera su più file. In UNIX esiste il concetto di file standard:

- ▶ Standard input → il file da cui il programma acquisisce i suoi input (codice 0);
- ▶ Standard output → il file da cui il programma produce i suoi output (codice 1);
- ▶ Standard error → il file da cui il programma produce i suoi errori (codice 2);

Normalmente:

- ▶ standard input → tastiera;
- ▶ Standard output/error → terminale.

È ovviamente possibile dirottare questi canali su file o altri canali.

Si utilizzano le parentesi angolari → < ; > ; << ; >>

Se il file non esiste viene creato.

Se il file non esiste, viene riscritto (>) o accodato (>>)

5.9 Ridirezione Stn Input

```
command arg1 ... argn < file
```

Il file **file** viene rediretto sullo standard input del comando.

5.10 Comando cat

Concatena i file e li scrive sullo standard output, a meno di mancati argomenti → in questo caso scrive lo standard input sullo standard output.

es:

```
cat <<EOF
Caro amico,
leggi questa lettera
EOF
```

5.11 Ridirezione Stn Error

```
comando argomenti 2 > file
```

```
comando argomenti 2 >> file
```

```
1 echo "ciao!"
2 - bash: echo: command not found
3 echo "ciao!" 2 > /dev/null
```

5.12 Ridirezione

```
comando (codiceA)>& (codiceB)
```

Redirige il canale A sul canale B:

```
comando > file 2>&1
```

5.13 Pipe (|)

Pipeline di 2 o più comandi.

Lo standard output di `comando1` diventa l'input per il `comando2`

```
com1 [arg ...] | com2 [arg ...] | ...
```

5.14 Command substitution

Il pattern `$(comando)` viene sostituito con l'output del comando.

5.15 Metacaratteri

La shell riconosce alcuni caratteri speciali, chiamati metacaratteri, che possono comparire nei comandi.

Quando l'utente invia un comando, la shell lo scandisce alla ricerca di metacaratteri che processa in modo speciale.

Il metacarattere `*` nel pathname è un'abbreviazione per un nome di file (Filename Expansion).

- ▶ `*` → stringa di 0 o più caratteri;
- ▶ `?` → singolo carattere;
- ▶ `[]` → singolo carattere tra quelli elencati;
- ▶ `{ }` → stringa tra quelle elencate

5.16 Quoting

Il meccanismo del **quoting** è utilizzato per inibire l'effetto dei metacaratteri. Questi perdono il loro significato speciale e la shell li tratta come caratteri ordinari;

Il metacarattere `\` (escape) inibisce l'effetto speciale del metacarattere che segue.

Tutti i metacaratteri presenti in una stringa racchiusa tra singoli apici perdono l'effetto speciale.

I metacaratteri per l'abbreviazione del pathname presenti in una stringa racchiusa tra doppi apici perdono l'effetto speciale (ma non tutti i metacaratteri della shell)

5.17 Metacaratteri di shell

Simbolo	Significato	Esempio d'uso
>	Ridirezione dell'output	<code>ls > temp</code>
>>	Ridirezione dell'output (append)	<code>ls >> temp</code>
<	Ridirezione dell'input	<code>wc -l < text</code>
<<delim	Ridirezione dell'input da riga di comando (here document)	<code>wc -l <<delim</code>
*	Wildcard (stringa di 0 o più caratteri ad eccezione del punto)	<code>ls *.c</code>
?	Wildcard (singolo carattere ad eccezione del punto)	<code>ls ?.c</code>
[...]	Wildcard (singolo carattere tra quelli elencati)	<code>ls [a-zA-Z].bak</code>
{...}	Wildcard (le stringhe specificate all'interno delle parentesi)	<code>ls {prog.doc}*txt</code>

5.18 Comando wc (word count)

Fornisce il numero dei codici di interruzione di riga (in pratica il numero di righe) delle parole o dei caratteri contenuti in file. Senza opzioni fornisce, nell'ordine suddetto, ciascuna delle precedenti informazioni.

Tra le opzioni ricordiamo:

- ▶ `-c` → emette solo il numero complessivo di caratteri di file;
- ▶ `-w` → emette solo il numero complessivo di parole in file;
- ▶ `-l` → emette solo il numero complessivo di righe in file;

5.19 Comandi concatenabili

Solo a inizio pipe → `echo`, `ls`, ...

Anche al centro → `wc`, `sort`, `uniq`, `grep`, `cat`, `head`, `tail`, ...

Solo a fine pipe → `less` (paginatore interattivo).

5.20 Esercitazione

```
1 #!/usr/bin/env bash
2 set -e
3
4 # Creazione della directory e salvataggio dei dati in un file
```

```

5   mkdir EsercitazioneLS0-1 && echo "$USER $hostname" > EsercitazioneLS0-1/provaFile.txt
6
7   a=$(cat EsercitazioneLS0-1/provaFile.txt)
8
9
10  a="${a} $(basename EsercitazioneLS0-1/provaFile.txt)"
11
12  echo "$a"

```

5.21 Comando sort

Permette di riordinare o fondere insieme il contenuto di file passati come parametri. In assenza di opzioni che definiscano diversi criteri di ordinamento, si utilizza quello alfabetico.

5.22 Comando head & tail

Mostrano le prime e ultime X righe di un file

5.23 Word splitting

L'ultima fase prima di eseguire un comando consiste nella suddivisione in parole. La variabile IFS (internal field separator) definisce i separatori (di default è lo spazio)

Come effetto collaterale, il word splitting sostituisce i newline con spazi.

6 Lezione del 06-10

6.1 Comando grep e le espressioni regolari

Se l'opzione extglob è attiva (usando il comando built-in shopt), abbiamo un'estensione per il matching pattern.

```
grep [opzioni] pattern [nomefile]
```

Stampa le righe del file che corrispondono al pattern (espressione regolari).

Senza un file specificato, legge dallo standard input (utile quindi con la pipe).

Flags:

- ▶ **-v** → stampa le righe che non corrispondono al pattern;
- ▶ **-c** → visualizza il numero delle occorrenze della stringa nel file
- ▶ **-i** → case insensitive;

- ▶ `-n` → numero di riga.

6.2 Espressioni regolari

% Definizione

Una espressione regolare è un pattern che descrive un insieme di stringhe

L'elemento atomico delle espressioni regolari è il carattere:

- ▶ Un carattere è una espressione regolare che descrive se stesso.

La maggior parte dei caratteri sono *espressioni regolari*.

- ▶ `.` → rappresenta un qualunque carattere;
- ▶ `exp*` → 0 o più occorrenze di exp;
- ▶ `^exp` → exp all'inizio del rigo;
- ▶ `exp$` → exp alla fine del rigo;
- ▶ `[a-z]` → un carattere nell'intervallo specificato;
- ▶ `[^a-z]` → un carattere fuori dall'intervallo specificato;
- ▶ `\<exp` → exp all'inizio di una parola;
- ▶ `exp/>` → exp alla fine di una parola;
- ▶ `exp{N}` → exp compare N volte;
- ▶ `exp{N,}` → exp compare almeno N volte;
- ▶ Classi di caratteri.

Molti comandi per l'elaborazione di testi di UNIX consentono la definizione di espressioni regolari, ossia di schemi per la ricerca di testo basati sull'impegno di metacaratteri:

- ▶ Solitamente, i metacaratteri usanti da tali comandi non coincidono con i metacaratteri impiegati dalla shell per identificare i nomi dei file.

La concatenazione di espressioni regolari è una espressione regolare

6.3 Extendend regular expression

- ▶ `exp+` → 1 o più occorrenze di exp;
- ▶ `exp?` → 0 o più occorrenze di exp;
- ▶ `exp1|exp2` → exp1 oppure exp2;

- ▶ `(exp)` → equivalente a `exp`, serve a stabilire l'ordine di valutazione

In `grep` questi comandi vanno preceduti dall'escape `\;`; con `egrep` si possono usare direttamente.

6.4 Processi Bash

I processi sono programmi in esecuzione. Lo stesso programma può corrispondere a diversi processi.

Ciascun processo può generare nuovi processi (figli).

Ogni processo ha:

- ▶ PID;
- ▶ PPID.

Tranne il processo `init`, che ha PID 1 e nessun PPID (è il primo che parte all'avvio).

La nascita di un nuovo processo, può avvenire soltanto attraverso una richiesta da parte di un altro processo già esistente.

6.4.1 Tabella processi

Il kernel gestisce una tabella che tiene traccia di tutti i processi attivi.

6.4.2 Attributi dei processi

Ogni processo ha associato 2 utenti:

- ▶ `Real user` → utente che ha lanciato il processo;
- ▶ `Effective user` → utente che determina i diritti del processo.

Quando un processo apre un file, vale l'effective user.

Di solito i 2 utenti coincidono.

Se invece un file eseguibile ha il bit `set user ID` impostato, il corrispondente processo avrà:

- ▶ Real user → utente che ha lanciato il processo;
- ▶ Effective user → utente proprietario dell'eseguibile.

Ogni processo ha anche 2 gruppi associati:

- ▶ Real group;
- ▶ Effective group.

6.4.3 Comando ps

Il comando **ps** fornisce i processi presenti nel sistema.

```
ps [selezione] [formato]
```

Selezione:

- ▶ niente → processi lanciati dalla shell corrente;
- ▶ **-u** **USER** → processi dell'utente indicato;
- ▶ **-a** → tutti i processi

Formato:

- ▶ Niente → PID, terminale, ora di esecuzione, comando;
- ▶ **-f** → (full) aggiunge UID, PPID, argomenti;
- ▶ **-F** → anche altro;
- ▶ **-o** → visualizza i campi specificati.

6.4.4 Terminazione di un processo

Per arrestare un processo in esecuzione si può utilizzare:

- ▶ C-c dal terminale stesso su cui il processo è in esecuzione;
- ▶ Comando **kill** seguito dal nome del PID (eventuali parametri) o con il loro job-id (preceduto da una %).

6.4.5 Controllo dei processi

Normalmente, la shell aspetta che ogni comando termini (comando in foreground)

Aggiungendo al comando **&** possiamo mandare il processo in background → il comando può comunque scrivere su standard output.

Per identificare i comandi possiamo utilizzare il job number (valido a livello della shell aperta) e il loro PID.

I processi in background sono eseguiti in una sottoshell, in parallelo al processo padre (la shell) e non sono controllati da tastiera.

I processi in background sono utili per eseguire task in parallelo che non richiedono controllo da tastiera.

6.4.6 Jobs e Processi

Un job di shell rappresenta tutti i processi che vengono generati da un comando impartito dalla shell stessa.

Un comando impartito, attraverso una shell può generare più di un processo.

6.5 Monitoraggio memoria

Il comando `top` fornisce informazioni sulla memoria utilizzata dai processi, che vengono aggiornate ad intervalli di X secondi.

6.6 Script shell

Uno script di shell *BASH* è un file di testo che inizia con:

```
#!/usr/bin/env bash
```

E che abbia permessi di esecuzione.

Non esiste differenza tra quello che si può scrivere al prompt e quello che si può scrivere in uno script.

Si utilizza `#!/usr/bin/env bash` per indicare che il file è uno script. Il resto dice a bash qual è l'interprete per questo script.

6.7 Variabili predefinite negli script

- ▶ `$0` → il nome dello script stesso (`argv[0]`);
- ▶ `$1..$9` → parametri da riga di comandi;
- ▶ `$#` → numero di parametri ricevuti;
- ▶ `$*` → tutti i parametri in una stringa singola;
- ▶ `$@` → tutti i parametri in stringhe separate;
- ▶ `$!` → process ID del processo corrente;
- ▶ `$?` → exit status dell'ultimo comando eseguito.

6.8 Exit status

Ogni comando restituisce un intero detto *exit status* al chiamante:

- ▶ 0 → terminazione regolare;
- ▶ Diverso da 0 → terminazione irregolare.

6.9 Operatori sui comandi

- ▶ `cmd1; cmd2` → esegue il primo comando e poi il secondo;

- ▶ `cmd1 && cmd2` → esegue cmd1 e in caso di terminazione regolare esegue il secondo;
- ▶ `cmd1 || cmd2` → esegue il comando cmd2 solo se il primo comando ha una terminazione irregolare.

In tutti e 3 i casi, l'exit status complessivo è quello dell'ultimo comando eseguito.

6.10 Comando if

Come test usa l'exit status del comando.

7 Lezione del 08-10

7.1 XML

Nasce agli inizi degli anni '90, con lo scopo di espandere le potenzialità di HTML.

XML non è:

- ▶ Un sostituto di HTML (ma questo può essere generato da XML → XML è un meta-linguaggio (rappresentazione astratta di un numero ∞ infinito di linguaggi di programmazione))
- ▶ Un formato di presentazione, ma XML può essere convertito;
- ▶ Un linguaggio di programmazione;
- ▶ Un protocollo di trasferimento sulla rete (ma per molti anni è stato utilizzato per questo);
- ▶ Un DBMS (ma XML può essere salvato su DB).

XML è un meta linguaggio di markup per **documenti di testo/dati** → permette di definire linguaggi per rappresentare **documenti/dati**

```

1  <article>
2    <author> Gerhard Weikum </author>
3    <title> The Web in 10 Year </title>
4  </article>

```

% Parser

Processo di lettura di un file XML che fornisce una user application interface per l'accesso al documento

Un file XML è di facile lettura e molto espressivo (la semantica assieme ai dati).

Tutti i più moderni compilatori offrono librerie per la navigazione di file XML.

Attributi di un tag in un file XML → ulteriori descrittori dei dati.

7.2 Documento XML

In un file XML troviamo:

- ▶ Elementi;
- ▶ Attributi che definiscono gli elementi;
- ▶ Altro ...

Per ogni elemento possiamo dichiarare 1 o più attributi.

7.2.1 Elementi in un file XML

Tags (definibili liberamente) → article, title, author (con tag di apertura e chiusura).

Gli elementi:

- ▶ Hanno un nome e un contenuto;
- ▶ Possono avere nesting;
- ▶ Possono essere vuoti;

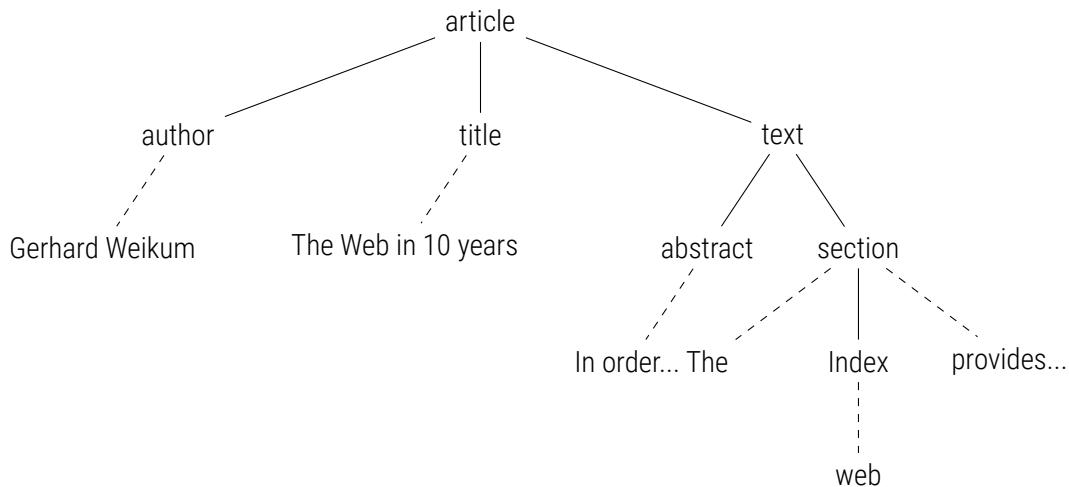
Ogni documento XML ha esattamente una root e una forma di albero.

Elementi con un parente comune sono ordinati

7.2.2 Elementi e Attributi

Gli elementi possono avere attributi che hanno un nome e un valore.

7.2.3 Documenti XML come alberi ordinati



7.2.4 Caratteri speciali in XML

Alcuni caratteri speciali devono essere evasi attraverso entità:

- ▶ < → <
- ▶ & → &
- ▶ > → >
- ▶ " → "
- ▶ ' → &apos.

7.3 Documenti XML ben costruiti

Un documento XML per essere ben costruito deve rispettare le seguenti regole:

- ▶ Ogni tag di inizio deve averne uno di fine;
- ▶ Gli elementi possono essere nestati, ma non sovrapposti;
- ▶ Deve essere presente una sola root;
- ▶ I valori degli attributi devono essere quotati;
- ▶ Un elemento non dovrebbe avere molteplici attributi con lo stesso nome;
- ▶ Commenti e istruzioni non dovrebbero apparire all'interno dei tag;
- ▶ Evitare di usare caratteri non evasi.

7.4 Namespace Syntax

La semantica delle descrizioni degli elementi è ambigua. Pertanto il contenuto dovrebbe essere definito differentemente → La rinomina potrebbe essere impossibile

7.4.1 Struttura

1 `<dbs:book xmlns dbs="http://www-dbs/dbs">`

Dove:

- ▶ **dbs** → prefix e abbreviazione dell'URI;
- ▶ **xmlns** → segnale per l'inizio dichiarazione della definizione;
- ▶ **http://www-dbs/dbs** → URI unico per identificare il namespace

es:

```
1 <dbs:book xmlns dbs="http://www-dbs/dbs>
2   <dbs:description>...</dbs:description>
3   <dbs:text>
4     <dbs:formula>
5       <mathml:math xmlns:mathml="http://www.w3.org/1998/Math/MathML"> ... </mathml:math>
6     </dbs:formula>
7   </dbs:text>
8 </dbs:book>
```

7.4.2 Namespace di default

Potrebbe essere settato per gli elementi e il suo contenuto (ma non per i suoi attributi).

Possono essere sovrascritti negli elementi specificando il namespace (usando il prefix o il namespace di default).

7.5 Definizione dei tipi di documento

Spesso XML è troppo flessibile:

- ▶ Molti programmi possono processare solo una limitata porzione delle applicazioni XML;
- ▶ Per lo scambio dei dati, il formato deve essere fixato;
- ▶ Il DTD (Document Type Definition) per stabilire il vocabolario per una applicazione XML.

Un documento è valido rispettando il DTD se è conforme alla sue regole specificate.

Molti parser XML possono essere configurati per la validazione.

7.6 Elementi di un app Android

Activity → il livello di presentazione dell'app. La UI è costruita attorno a una o più activities. Usano *Fragments* e *Views* per organizzare e mostrare le informazioni.

Service → sono in esecuzione senza UI e comunicano con gli altri oggetti. Utili per eseguire task asincroni (es comunicazione di rete).

Intent → elementi per il *message-passing* tra le applicazioni.

Content Provider → gestiscono la persistenza dei dati dell'app. Sono usati per gestire dati tra le applicazioni.

Broadcast Receiver → riceve intent dal SO.

Widget → componenti visive delle applicazioni che possono essere aggiunti sulla home-screen. Sono una variazione dei Broadcast Receiver.

Notification → abilitano l'app all'invio di alert all'utente senza togliere il focus all'app in esecuzione.

7.7 Descrizione file XML - metadati e struttura dell'app (Android Manifest)

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"> <!-- Indica il namespace
   utilizzato -->
3
4     <!-- Internet is required. READ_LOGS are to ensure that the Logcat is transmitted-->
5     <uses-permission android:name="android.permission.INTERNET"/>
6     <uses-permission android:name="android.permission.READ_LOGS"/>
7
8     <application
9         android:allowBackup="true"
10        android:name=".ACRAHandler"<!-- Activates ACRA on startup -->
11        android:icon="@drawable/ic_launcher"
12        android:label="@string/app_name"
13        android:theme="@style/AppTheme" >
14            <!-- Activities -->
15        </application>
16    </manifest>
```

7.8 Permessi

Sintassi:

```
1 <uses-permission android:name="string" [android:maxSdkVersion="integer"] />
2 <!-- Esempio per i Contatti -->
3 <uses-permission android:name="android.permission.READ_CONTACTS"/>
```

7.9 SDK manifest

Sintassi

```
1 <uses-sdk android:minSdkVersion="integer" android:targetSdkVersion="integer" [android:maxSdkVersion=
  "integer"]/>
```

7.10 Schemi supportati

Sintassi

```
1 <supports-screens android:resizeable=["true" | "false"]
2   android:smallScreens=["true" | "false"]
3   android:normalScreens=["true" | "false"]
4   android:largeScreens=["true" | "false"]
```

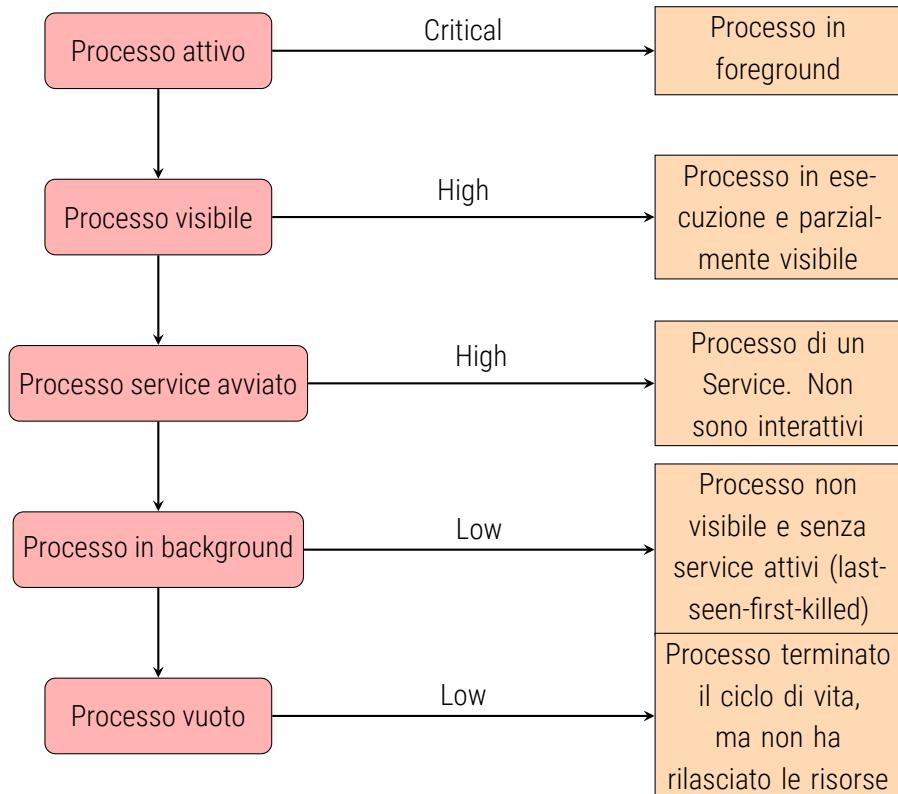
```

5     android:xlargeScreens=["true" | "false"]
6     android:requiresSmallestWidthDp="integer"
7     android:compatibleWidthLimitDp="integer"
8     android:largestWidthLimitDp="integer">

```

NB → `android:resizeable` è deprecato.

7.11 Application priority stack



7.12 Application

```

1 <application android:icon="@drawable/icon" android:name="MyApplication">
2   <!-- android:name="MyApplication" specifica la classe dell'app -->

```

7.13 Activity

```

1 public class MyActivity extends Activity {
2     @Override
3     public void onCreate(Bundle savedInstanceState) { // Metodo obbligatorio
4
5         super.onCreate(savedInstanceState); // Chiamata alla superclasse (salva lo stato dell'activity)
6         setContentView(R.layout.main); // Collega la classe alla sua interfaccia grafica

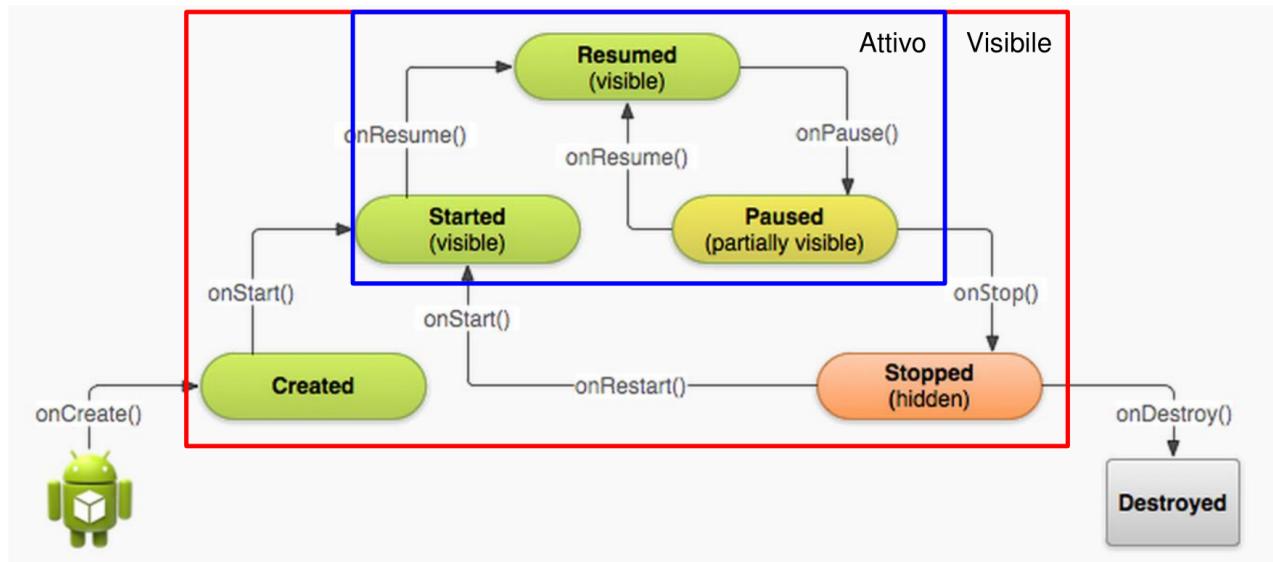
```

7 }
8 }

Metodi:

- ▶ onRestart();
- ▶ onStart();
- ▶ onResume();
- ▶ onPause();
- ▶ onStop();
- ▶ onDestroy();

7.14 Activity lifecycle



7.15 Log

1 `private static final String TAG = "MyActivity";`

Diversi livelli di log (dal meno importante al più importante)

- ▶ Verbose (v);
- ▶ Debug (d);
- ▶ Info (i);
- ▶ Warning (w);

- ▶ Error (e);
- ▶ Assert (wtf).

Presentano 2 parametri:

- ▶ Tag : String;
- ▶ Message : String.

8 Lezione del 11-10

8.1 Espressioni condizionali

Il comando `test exp` valuta `exp` come espressione condizionale:

- ▶ Termina con exit status 0 se l'espressione è vera

`test exp = [exp]`

Operatori ammessi:

- ▶ Su stringhe → `=`, `!`, `-Z`;
- ▶ Su interi → `-lt`, `-le`, `-eq`, `-ne`, `-ge`, `-gt`;
- ▶ Operatori unari su nomi di file → `-e`, `-f`, `-r`, `-w`, `-x`

```

1 if [ -z "$1" ]; then
2   echo "Questo script richiede un argomento"
3   exit 1
4 fi

```

8.2 Sostituzione aritmetica con `$((...))`

- ▶ `$((exp))` valuta `exp` come espressione aritmetica;
- ▶ `$((exp))` viene sostituito dalla shell con il valore di `exp`;
- ▶ Solo aritmetica con numeri interi.

8.3 Sostituzione aritmetica

Operatori:

- ▶ Aritmetici;
- ▶ Elevamento a potenza;
- ▶ bit-a-bit;
- ▶ Booleani.

8.4 Ciclo while

```
1 while comando
2   do
3     sequenza
4     conadi
5   done
6 # es
7 while true
8   do
9     echo "Inserisci il nome di un file da visualizzare"
10    echo "q per uscire:"
11    read nome_file
12    if [ nome_file == "q" ]; then
13      break
14    else
15      cat "$nome_file"
16    fi
17 done
```

8.5 Ciclo for

```
1 for var in lista variabile
2   do
3     sequenza comandi
4   done
5
6 # es
7 for a in 1 2 3
8
9 for a in "$@"
10
11 for a in "$*"
```

8.6 Case statement

```
1 case stringa in
2     stringa caso 1) lista comandi 1;;
3     stringa caso 2) lista comandi 1;;
4 ...
5 esac
6
7 # es
8
9 case "$1" in
10    --status)
11        print_status
12        ;;
13    *)
14        show_menu
15        ;;
16 esac
```

8.7 Script interattivi

È possibile scrivere script interattivi attraverso il comando `read`

8.8 Ciclo until

Esegue la lista di comandi finché il comando è vero

8.9 sed e awk

8.9.1 sed (Stream Editor)

Editor non interattivo di file di testo (1974 - Bell Labs, come evoluzione di grep)

Editor di linea che non richiede l'interazione con l'utente.

Può filtrare l'input che riceve da un file o una pipe. Di default non effettua modifica

Legge i file specificati, o lo standard input se non specificato

L'input è scritto sullo standard output.

Alcuni comandi:

- ▶ `a\` → append di testo al di sotto della riga corrente;
- ▶ `c\` → Modifica il testo della riga corrente;

- ▶ **d** → Cancella la riga corrente.

La forma del comando sed è la seguente:

```
[address [, address]] function [arguments]
```

Sed è una macchina a registri:

1. Copia ciclicamente una linea di input in un pattern space;
2. Applica tutti i comandi con address selezionati dal pattern space;
3. Copia il pattern space nello standard output, aggiungendo newline;
4. Cancella il pattern space.

L'indirizzo non è richiesto, ma se specificato deve essere:

- ▶ Un numero;
- ▶ Un carattere \$ per l'ultima linea di input;
- ▶ Un address di contesto (espressione regolare preceduta o seguita da un delimitatore).

Una linea di comando senza indirizzo seleziona ogni pattern space.

Una linea di comando con un indirizzo seleziona ogni pattern space dato dall'address

Una linea di comando con 2 indirizzi seleziona un range esclusivo del primo pattern space tra i 2 primi indirizzi

Un indirizzo può essere:

- ▶ N-esima riga;
- ▶ Ultima riga;
- ▶ Espressione regolare fra match
- ▶ Range (dato da una coppia di indirizzi begin e cont separati da , e inizia dalla prima linea che fa match con begin (inclusa) e si estende fino a che cont fa match)

Se non si specificano azioni, sed stampa sullo standard output le linee di input, lasciandole inalterate.

Ricerca e Sostituzione

es.

```
1 sed -e 's/erore/errore/g'
```

NB: Le espressioni regolari sono greedy → si cerca di matchare la stringa più lunga possibile

8.9.2 awk

Linguaggio per l'elaborazione di modelli orientato a campi (prende il nome dai loro creatori)

Linguaggio di scripting:

- ▶ POSIX complaint;
- ▶ gawk è la sua implementazione ben documentata;
- ▶ Strumento ideato per processare file di testo strutturati in *record* (definibili dall'utente), farne report;
- ▶ Sintassi simil-C;
- ▶ Molto usato per scriptini "one liner".
- ▶ [Riferimenti](#)

Il vantaggio di awk è la suddivisione la linea in una sequenza di campi delimitati da separatori (per modificare il delimitatore si utilizza il flag **-F**).

La variabile **\$0** contiene la riga in esame.

Sintassi:

```
awk [options] file
```

8.9.3 Struttura di un programma awk

È costituito da una sequenza di regole:

```
pattern { action }
```

Dove pattern è uno tra:

- ▶ **BEGIN**, prima di processare l'input;
- ▶ **END**, dopo aver processato l'input;
- ▶ **boolexpr**, fa match se è vera;
- ▶ **/regex/**, fa match se la regex fa match;

- ▶ **bpat**, **epat**, dal record che fa match con **bpat** a quello che fa match con **epat**

Default:

- ▶ Action → `print $0;`
- ▶ Pattern → true.

```
awk 'length($0) > 80' data
```

```
awk '{print $2}' data
```

es.

```
1 df -H /dev/disk/by-label/root \
2 | awk '/\dev/ {print "Spazio Totale: " $2, "\nSpazio Occupato: " $3, "\nSpazio Rimanente: " $4,
      "\nSpazio utilizzato: " $5}'
```

8.9.4 Operatori e Predicati

Operatori aritmetici:

- ▶ Somma;
- ▶ Sottrazione;
- ▶ Prodotto;
- ▶ Esponente;
- ▶ Divisione;
- ▶ Resto

Gli operatori sono anche disponibili in versione `<op>`

Predicati disponibili per numeri e stringhe:

- ▶ Ordinamento;
- ▶ Uguaglianza;
- ▶ Università;
- ▶ Match con regex;
- ▶ Match negato.

8.9.5 Variabili per awk

awk usa molte variabili, alcune editabili, altre read-only:

- ▶ La variabile **FS** (Field separator) → identifica il separatore di input (default tab e spazi);
- ▶ La variabile **OFS** (Output field separator) → identifica il separatore di output (default spazio);
- ▶ La variabile **OFS** (Output record separator) → identifica il separatore per i record (di default carattere di nuova riga);
- ▶ La variabile **NR** (Number record) → identifica la riga interessata.

8.9.6 Awk - array

Array associativi (anche multidimensionali) → presentano una chiave per accedere ai loro elementi.

9 Lezione del 13-10

9.1 Linguaggio C

I sorgenti devono essere creati utilizzando un editor di testo.

Per gcc, il suffisso del nome del file identifica **la/le operazioni/e** che il compilatore esegue:

- ▶ **file.c** → Codice sorgente C che deve essere preprocessato
 - ◊ **file.C** → codice sorgente C++;
- ▶ **file.h** → Header file che non deve essere né compilato né utilizzato per l'operazione di link.

Utilizzeremo lo standard [C99](#).

Contenuto del file:

- ▶ I file **.c** contengono direttive per il pre-processore e codice sorgente;
- ▶ I file **.h** possono contenere direttive per il pre-processore, ...

È opportuno predisporre una certa modularità del programma (semplifica lo sviluppo, il debugging ed il riuso del codice).

L'accorpamento delle funzioni avviene a tempo di compilazione.

9.2 Compilazione

Sui sistemi Linux avviene attraverso il comando **gcc**. Con l'opzione **-Wall** mostra tutti i Warning (useremo anche **-Wextra** e **-Werror**)

Per creare l'eseguibile, gcc attraversa varie fasi:

- ▶ Pre-processing;
- ▶ Compilazione;
- ▶ Linking

Normalmente il compilatore esegue sia la compilazione che il linking (con il flag `-c` otteniamo solo la fase di compilazione → otteniamo file oggetto, che possono essere linkati tra loro).

Il programma `make` automatizza questo processo.

9.3 Memo

In ogni applicazione esiste sempre un'unica funzione main:

- ▶ Unica per l'applicazione;
- ▶ Non unica in ogni file.

Le firme standard della funzione main sono:

```
int main(int argc, char *argv[])
int main(void)
```

Il secondo caso lo si ha nel caso in cui ci si aspetti di non avere parametri da riga di comando Consente di ottenere i parametri passati all'applicazione dalla linea di comando

9.4 Operatori

- ▶ Operatori relazionali → `> >= < <= = !`;
- ▶ Operazioni logici → `&& || !`;
- ▶ Operatori di incremento e decremento → `k++ ++k k-- --k`;
- ▶ Operatori orientati ai bit → `& (and) !(or) ^ (xor) << (shift a sx) >> (shift a destra) ~ (complemento a 1)`

9.5 Funzioni e prototipi

È opportuno dichiarare sempre le funzioni utilizzate all'interno di ogni file → semplifica la correzione degli errori legati ai tipi (solitamente vanno messi nei file header)

La firma di un metodo include:

- ▶ Tipo del valore restituito della funzione;

- ▶ Nome della funzione;
- ▶ Elenco dei parametri con rispettivi tipi.

9.6 Esempio 1

```

1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main(void){
5     printf("Hello World!");
6     sleep(5);
7     // Aspetta prima 5 secondi e poi stampa "Hello World!"
8     // Nel caso volessimo eseguirle in ordine aggiungiamo \n
9 }
```

Alcune funzioni di I/O standard utilizzano il buffering:

- ▶ Utilizzano un'area di memoria per memorizzare le informazioni prima di inviarle al device;
- ▶ Riduce il numero di *system call effettive* e migliora le performance.

Esistono 3 tipi di buffering:

- ▶ **Completo** → L'I/O effettivo avviene solo al riempimento del buffering
 - ◊ Non utilizzato per device interattivi come schermo e tastiera;
- ▶ **Buffering a linea** → L'I/O viene eseguito non appena viene inserito nel buffer un carattere newline \n (o al termine del processo);
 - ◊ Utilizzato da `printf` e `scanf`;
- ▶ **Senza buffering** → L'I/O avviene immediatamente
 - ◊ Utilizzato ad esempio per lo standard error.

9.7 Esempio 2

```

1 #include <stdio.h>
2
3 int main(void){
4     int x,y;
5     x=0;
6     y=0;
7 }
```

```

8     while(x=y) x=x+1;
9
10    printf("x=%d, y=%d\n", x,y);
11 }
```

Avremo che entrambi le variabili sono uguali a zero (il while è un confronto per assegnamento).

9.8 Esempio 3

```

1 #include <stdio.h>
2
3 int main(void){
4     int x,y;
5     x=0;
6     for(y=0; y<99; y++){
7         x= x+1;
8         printf("x=%d, y=%d\n", x,y);
9     }
}
```

9.9 Esempio 4 - scanf

```

1 #include <stdio.h>
2 int main(void){
3
4     int x;
5     printf ("Inserisce un intero:");
6     scanf("%d", x); // ERRORE -> scanf si aspetta un indirizzo e non una variabile
7     scanf("%d", &x); // MODO CORRETTO;
8 }
```

9.10 File in UNIX

Il kernel di UNIX vede tutti i file come flussi non formattati di byte → l'interpretazione è lasciato alle applicazioni. La struttura dei file è composta:

- ▶ Nome;
- ▶ Inode;
- ▶ Dati.

9.10.1 Inodo

Le informazioni contenute all'interno di ogni i-node residente in un file system sono le seguenti:

- ▶ Modo del file;
- ▶ Contatore link;
- ▶ ID Proprietario;
- ▶ ID Gruppo;
- ▶ Dimensione;
- ▶ Indirizzi;
- ▶ Ultimo Accesso;
- ▶ Ultima Modifica;
- ▶ Ultima modifica dello stato.

9.10.2 Accesso e creazione di file

Quando un processo si riferisce ad un file per nome (path assoluto o relativo) il kernel suddivide il path componenti per componente controllando i permessi di accesso alle directory relative.

In caso di esito positivo, il kernel:

- ▶ Ritrova l'inodo del file, se il file esiste e il processo chiede di accedervi;
- ▶ Assegna un inodo non usato, se il processo chiede di creare un nuovo file.

In entrambi i casi il kernel restituisce al processo un intero non negativo (descrittore del file), che resta associato al file fino a quando il file non viene rilasciato.

Attraverso i descrittori il kernel accede ai file e ne permette le elaborazioni da parte dei processi.

9.11 I/O di basso livello

5 chiamate di sistema fondamentali:

- ▶ open;
- ▶ read;
- ▶ write;
- ▶ lseek;
- ▶ close

Il kernel associa un *file descriptor* ad ogni file aperto:

- ▶ Il file descriptor è un intero;
- ▶ Quando un file viene aperto con *open*, la funzione *open* restituisce il file descriptor associato al file.

Le costanti simboliche **STDIN_FILENO** (0), **STDOUT_FILENO** (1), e **STDERR_FILENO** (2) sono definite in **unistd.h**.

9.12 Descrittori di file

Quando si vuole leggere o scrivere su un file si passa come argomento a read e write il descrittore ritornato da open.

Per convenzione il descrittore 0 viene associato allo standard input, 1 allo standard output e 2 allo standard error.

Questi numeri possono essere sostituite dalle costanti definite nell'header `<unistd.h>`.

9.13 La funzione open

```
1 #include <sys/types.h> // data types
2 #include <sys/stat.h> // data returned by stat()
3 #include <fcntl.h> // file control option
4
5 int open(const char *pathname, int oflag, ... /* mode_t mode */);
```

Restituisce il descrittore del file (-1 in caso di errore).

Permette di aprire sia un file esistente che di crearne uno nuovo

`pathname` è il pathname (assoluto o relativo) del file.

`oflag` permette di specificare le opzioni mediante costanti definite da `<fcntl.h>`, combinate con | (or bit-a-bit).

`mode` è il modo del file ed è un parametro opzionale, utilizzato solo nel caso di creazione del file ("..." modo ISO C per dire variabile).

9.14 Chiamata di sistema open

`oflag` può assumere diversi valori (definiti nell'header `<fcntl.h>`):

- ▶ `O_RDONLY` → apri solo in lettura;
- ▶ `O_WRONLY` → apri solo in scrittura;
- ▶ `O_RDWR` → apri in lettura e scrittura.

Solo una delle precedenti costanti può essere specificata, con una combinazione OR di:

- ▶ `O_APPEND` → esegue un append dalla fine del file per ciascuna write;
- ▶ `O_CREAT` → crea il file se non esiste;
- ▶ `O_EXCL` → se utilizzato insieme a `O_CREAT`, ritorna un errore se il file esiste;
- ▶ `O_TRUNC` → se il file esiste e aperto con successo `write-only/read-write`.

9.15 I flag di open

O_RDONLY	Apre il file in sola lettura. Questa opzione non può essere combinata con O_WRONLY e O_RDWR
O_WRONLY	Apre il file in sola scrittura. Questa opzione non può essere combinata con O_RDONLY e O_RDWR
O_RDWR	Apre il file in lettura e scrittura. Questa opzione non può essere combinata con O_RDONLY e O_WRONLY
O_APPEND	L'offset del file è posto alla fine del file prima di ogni chiamata a write
O_CREAT	Crea il file se esso non esiste. Richiede che open sia utilizzata con l'argomento mode, per la specifica del modo del file
O_EXCL	Genera un errore se è stata specificata anche l'opzione O_CREAT e se il file già esiste
O_TRUNC	Tronca il file a zero byte se esso esiste ed è stato aperto in sola scrittura o in lettura-scrittura
O_NCTTY	Se pathname si riferisce ad un terminale, non lo alloca come terminale di controllo per il processo.
O_NONBLOCK	Se pathname si riferisce ad una FIFO o ad un file di dispositivo, definisce la modalità nonblocking per l'apertura e l' I/O sul file
O_SYNC	Specifica che ogni chiamata write deve attendere per il completamento dell' I/O sul dispositivo fisico.

9.16 I permessi per open

S_IRWXU	Permesso di lettura, scrittura ed esecuzione per il proprietario
S_IRUSR	Permesso di lettura per il proprietario
S_IWUSR	Permesso di scrittura per il proprietario
S_IXUSR	Permesso di esecuzione per il proprietario
S_IRWXG	Permesso di lettura, scrittura ed esecuzione per il gruppo
S_IRGRP	Permesso di lettura per il gruppo
S_IWGRP	Permesso di scrittura per il gruppo
S_IXGRP	Permesso di esecuzione per il gruppo
S_IRWXO	Permesso di lettura, scrittura ed esecuzione per gli altri
S_IROTH	Permesso di lettura per gli altri
S_IWOTH	Permesso di scrittura per gli altri
S_IXOTH	Permesso di esecuzione per gli altri

9.17 Esempi di open

```
1 open("prova.txt", O_RDONLY); // accede al file read-only
2 open("prova.txt", O_RDONLY | O_CREAT, S_IRWXY); // accede al file read-only, se non esiste lo crea e
   setta i permessi
3 open("prova.txt", O_RDWR | O_CREAT | O_EXCL, S_IRWXY);
```

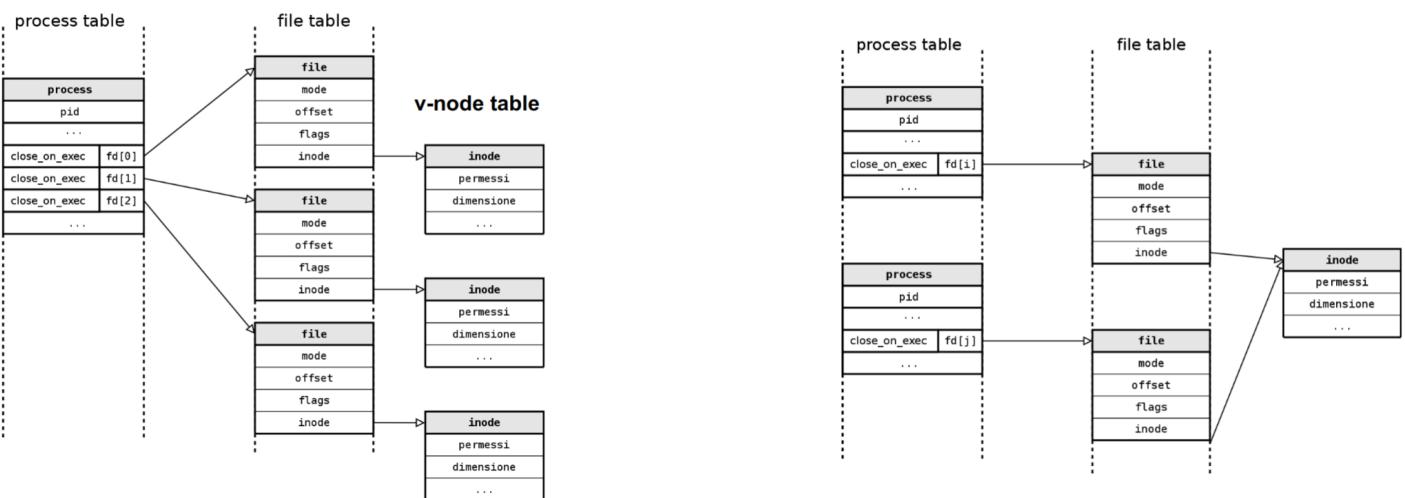
9.18 creat

```
1 #include <fcntl.h>
2
3 // Il nuovo file è aperto solo per scrittura
4 int creat(const char *pathname, mode_t mode);
5
6 // ANALOGO A
7 open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

9.19 Implementazione nel kernel

Il kernel usa 2 strutture dati indipendenti per gestire file aperti:

- ▶ Ogni processo mantiene la lista dei propri file descriptor (chiave astratta per accedere ai file, in POSIX un intero);
- ▶ Ogni file descriptor punta ad un elemento della file table;
- ▶ La file table contiene
 - ◊ La modalità di apertura del file;
 - ◊ Le opzioni con cui è stato aperto il file;
 - ◊ L'offset corrente;
 - ◊ L'inode corrispondente.



9.20 Accesso di un file da più processi

9.21 Trattare gli errori

Molte system call restituiscono `-1` in caso di errore.

Per avere maggiori informazioni, si usa la variabile globale `errno` (error number).

La funzione `perror(const char *)` stampa la stringa passata come parametro, e poi un messaggio in base al valore corrente di `errno`:

```
1 int fd = open("prova.txt", O_RDONLY);
2 if (fd < 0)
3     perror("errore di open");
4
5 int fd;
6 if (( fd=open("prova.txt", O_RDONLY)) < 0)
7     perror("errore di open");
```

9.22 Close

```
1 #include <unistd.h>
2 int close(int filedes) // accetta un unico parametro (descrittore del file)
```

Chiudere il file identificato da `filedes` e precedentemente aperto con `open`.

Restituisce 0 in caso di successo, `-1` in caso di errore

9.23 L'offset

Ad ogni file aperto è associato un intero, detto offset, che rappresenta la posizione (espressa in numero di byte dall'inizio del file) in cui verrà effettuata la prossima operazione di I/O.

L'offset è inizializzato a 0 da `open` (a meno che non venga specificato con `O_APPEND`).

Le operazioni di `read` e `write` incrementano il valore dell'offset di un numero di byte pari al numero di byte `letti/scritti`.

9.24 lseek

```
1 #include <sys/types.h>
2 #include <unistd.h>
3
4 off_t lseek(int filedes, off_t offset, int whence);
```

Modifica l'offset corrente del file. Restituisce il nuovo valore dell'offset, o `-1` in caso di errore.

Il valore del parametro **offset** è interpretato in base al parametro **whence**:

- ▶ **SEEK_SET** → L'offset corrente è posto a **offset** byte dall'inizio del file;
- ▶ **SEEK_CUR** → L'offset corrente è incrementato di **offset** byte
 - ◊ Il valore del parametro **offset** può essere sia positivo che negativo;
- ▶ **SEEK_END** → L'offset è posto a **offset** byte dalla fine del file
 - ◊ Il valore del parametro **offset** può essere sia positivo che negativo.

Per conoscere l'offset corrente, è sufficiente eseguire:

```
1 off_t currpos;  
2  
3 currpos = lseek(filedes, 0, SEEK_CUR);
```

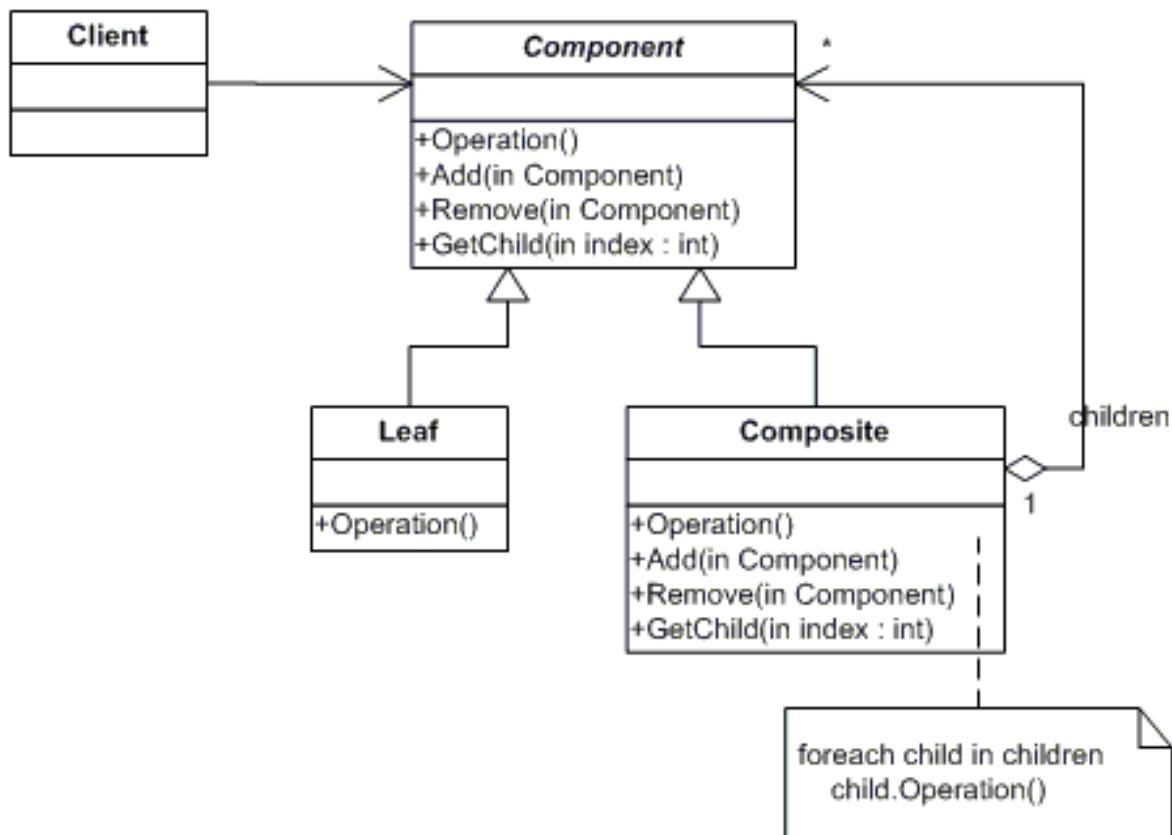
10 Lezione del 15-10

10.1 Risorse (XML - Java)

```
1 @string/nome_stringa @dimen/nome_dimensione  
  
1 getResources().getString(id_stringa)
```

10.2 Outline

View e Layout → implementazione del composite Pattern:

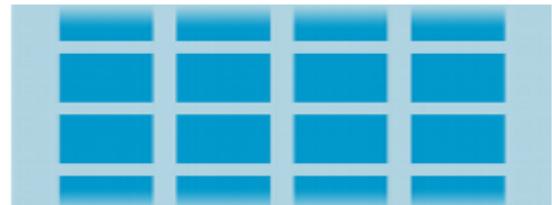


View e ViewGroup:

- ▶ ViewGroup → Contenitore invisibile che può contenere altri ViewGroup e View;
- ▶ View → elemento base della UI.

10.3 Layout

- ▶ LinearLayout → Allinea ogni View in orizzontale o verticale;
- ▶ RelativeLayout → La posizione delle View è definita in relazione alle View presenti;
- ▶ GridLayout → Usa una griglia per posizionare le View.



10.4 A simple XML document

Concede di definire tag liberamente:

```
1 <article>
2   <author> Gerhard Weikum </author>
3   <title> The Web in 10 Year </title>
4   <text>
5     <abstract> In order to evolve ... </abstract>
6     <section number="1" title="Introduction">
7       The <index>Web</index> provides the universal...
8     </section>
9   </text>
10 </article>
```

10.5 Elementi DTD

```
1 <!-- L'ordine deve essere rispettato -->
2 <!ELEMENT article (title, author+, text)> <!-- il + indica 1 o più -->
3 <!ELEMENT title (#PCDATA)> <!-- non hanno altri figli se non il markup-->
4 <!ELEMENT author (#PCDATA)>
5 <!ELEMENT test (abstract, section*, literature?)> <!-- il * indica 0 o più, il ? 0 o nessuno -->
6 <!ELEMENT abstract (#PCDATA)>
7 <!ELEMENT section (#PCDATA|index)+> <!-- | indica l'OR -->
8 <!ELEMENT literature (#PCDATA)>
9 <!ELEMENT index (#PCDATA)>
```

Ogni dichiarazione di elemento per ogni tipo di elemento.

Permette aritmetica arbitraria usando le parentesi.

Gli elementi possono:

- ▶ Essere mischiati;
- ▶ Avere contenuti vuoti;
- ▶ Possono avere contenuti arbitrari.

10.6 Attributi DTD

Per ogni elemento dichiarato deve essere presente una lista di attributi:

```
1 <!ATTRLIST section number CDATA #REQUIRED
2   title CDATA #REQUIRED> <!-- CDATA indica una stringa -->
```

In questo modo dichiariamo 2 attributi per ogni sezione, indicandone anche lo stato:

- ▶ #REQUIRED → valore richiesto per ogni istanza;
- ▶ #IMPLIED → è opzionale;
- ▶ #FIXED default → valore di default;
- ▶ default → ha un valore di default, modificabile.

Anche gli attributi presentano le loro espressioni regolari.

```

1  <!ATTRLIST publication type (journal|inproceedings) #REQUIRED
2      pubid ID #REQUIRED>
3  <!ATTRLIST cite cid IDREF #REQUIRED>
4  <!ATTRLIST citation ref IDREF #IMPLIED
5      cid ID #REQUIRED>
6
7  <publications>
8      <publication type="journal" pubid="Weikum01">
9          <author> Gerhard Weikum </author>
10         <text> In the Web of 2010, XML <cite cid="12"/> ... </text>
11         <citation cid="12" ref="XML98"/>
12         <citation cid="15" ... </citation>
13     </publication>
14     <publication type="inproceedings" pubid="XML98">
15         <text>XML, the extended Markup Language, ... </text>
16     </publication>
17 </publications>

```

10.7 Linking DTD e XML docs

```

1  <!DOCTYPE article SYSTEM "http://www-dbs/article.dtd">

```

10.8 Internal DTD

```

1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE article [
3      <!ELEMENT article (title, author+, text)>
4      ...
5      <!ELEMENT index (#PCDATA)>
6  ]>
7  <article>
8      ...
9  </article>

```

Entrambe le modalità possono essere mixate

10.9 Schema XML

DTD mette a disposizione tipi molto semplici (comportava una forte limitazione) → Si utilizzano pertanto gli schemi XML:

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3
4 <xs:element name="shiporder">
5   <xs:complexType>
6     <xs:sequence>
7       <xs:element name="orderperson" type="xs:string"/>
8       <xs:element name="shipto">
9         <xs:complexType>
10        <xs:sequence>
11          <xs:element name="name" type="xs:string"/>
12          <xs:element name="address" type="xs:string"/>
13          <xs:element name="city" type="xs:string"/>
14          <xs:element name="country" type="xs:string"/>
15        </xs:sequence>
16      </xs:complexType>
17    </xs:element>
18    <xs:element name="item" maxOccurs="unbounded">
19      <xs:complexType>
20        <xs:sequence>
21          <xs:element name="title" type="xs:string"/>
22          <xs:element name="note" type="xs:string" minOccurs="0"/>
23          <xs:element name="quantity" type="xs:positiveInteger"/>
24          <xs:element name="price" type="xs:decimal"/>
25        </xs:sequence>
26      </xs:complexType>
27    </xs:element>
28  </xs:sequence>
29  <xs:attribute name="orderid" type="xs:string" use="required"/>
30 </xs:complexType>
31 </xs:element>
32
33 </xs:schema>
```

Gli schemi XML sono applicazioni.

10.10 XPath

Path language → permette di identificare parti del documento XML (per future elaborazioni).

Opera su una rappresentazione di albero del documento.

I risultati sono un set di attributi.

10.11 Elementi di XPath

È una locazione che consiste in step separate da /:

```
1 /article/text/abstract <!-- seleziona tutti gli elementi abstract -->
```

Il primo / indica sempre l'elemento di root.

Ogni step viene valutato nel contesto di un nodo di un albero (context node).

I possibili step son:

- ▶ Child element x: → Seleziona tutti i figli con il nome x;
- ▶ Attributi @x → Seleziona tutti gli attributi con il nome x;
- ▶ Wildcards * (tutti i figli), @* (tutti gli attributi);
- ▶ Match multipli separati da | : x | y | z

11 Lezione del 18-10

11.1 Comando cut

Permette di "spezzettare" stringhe di testo:

```
1 echo "galileo" | cut -b 1-3 # ottengo gal
2
3 uname -a | cut -f 1-3 -d ' ' # Ottengo Linux PC 5.14.12
```

11.2 Comando uniq

Permette di eliminare le ripetizioni.

È possibile contare le ripetizioni di una stessa occorrenza utilizzo il flag -c

```
1 sort nomi.txt | uniq
2
3 sort nomi.txt | uniq -c
```

11.3 Comando find

Permette la ricerca di file o cartelle in una directory specificata (in base ai parametri specificati)

```
1 find . -type d -name "*dir*" # cerca una directory che matcha nella directory corrente
2 find . -type f -name "*file*" # cerca un file che matcha nella directory corrente
```

11.4 read

```
1 #include <unistd.h>
2
3 ssize_t read(int filedes, void *buf, size_t nbytes);
```

Restituisce:

- ▶ Il numero di byte effettivamente letto;
- ▶ 0 se ci troviamo alla fine di un file;
- ▶ -1 in caso di errore.

L'operazione di lettura avviene partendo dall'offset corrente del file:

- ▶ L'offset viene incrementato opportunamente.

Il numero di byte letti può essere diverso dal parametro di nbytes quando:

- ▶ Il numero di byte ancora presenti nel file è inferiore ad nbytes;
- ▶ La lettura avviene da un terminale (si legge una riga alla volta);
- ▶ La lettura avviene da un buffer di rete (nbytes superiore);
- ▶ La lettura avviene da una pipe o una FIFO;
- ▶ L'operazione è interrotta da un segnale.

11.5 Write

```
1 #include <unistd.h>
2
3 ssize_t write(int filedes, void *buf, size_t nbytes);
```

Restituisce:

- ▶ Il numero di byte effettivamente scritti;
- ▶ -1 in caso di errore.

L'operazione di scrittura avviene partendo dall'offset corrente del file:

- ▶ L'offset viene incrementato opportunamente.

11.6 Esempio

- ▶ Apre un file;
- ▶ Scrive il buf1;
- ▶ Sposta l'offset;
- ▶ Scrive il buf2

```
1 #include <stdio.h> // perror
2 #include <unistd.h> // perror
3 #include <errno.h> // write, lseek, close, exit
4 #include <sys/types.h>
5 #include <sys/stat.h> // open
6 #include <fcntl.h> // open
7
8 char buf1[] = "abcdefghijkl";
9 char buf2[] = "ABCDEFGHIL";
10
11 int main(void){
12     int fd;
13     if ((fd = open("file.hole", O_RDWR| O_CREAT, S_IRWXU)) < 0)
14         perror("open error");
15
16     if (write(fd, buf1, 10) != 10)
17         perror("buffer write error");
18     // L'offset ora è 10
19
20     if(lseek(fd, 20, SEEK_SET) == -1 )
21         perror("lseek error");
22     // L'offset ora è 20
23
24     if (write(fd, buf2, 10) != 10)
25         perror("buffer 2 write error");
26     // L'offset ora è 30
27
28     close(fd);
29     return 0;
30 }
```

Possiamo utilizzare il comando `hexdump` per mostrare a video il contenuto in:

- ▶ esadecimali;
- ▶ ottale;
- ▶ decimale;

► ASCII.

```

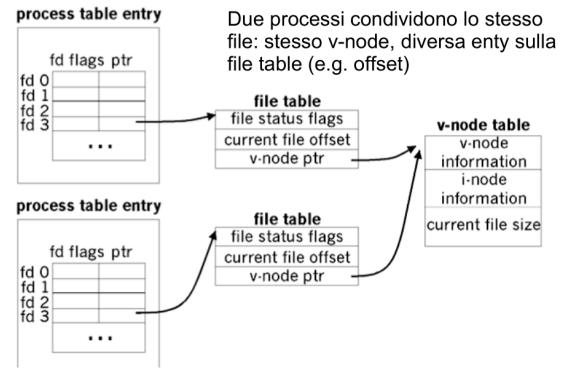
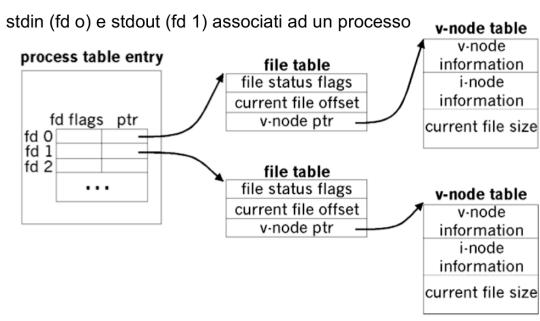
1 hexdump -c file.hole
2
3 # Ottengo:
4 0000000 a b c d e f g h i l \0 \0 \0 \0 \0 \0
5 00000010 \0 \0 \0 \0 A B C D E F G H I L
6 0000001e

```

11.7 Condivisione di file

Il kernel utilizza 3 strutture dati per la gestione dell' I/O:

1. Ciascun processo ha un elemento nella tabella dei processi → Tale elemento è un vettore di descrittori di file aperti, ciascuno con un puntatore ad un elemento della tabella del file;
2. Il kernel possiede una tabella per ciascun file aperto con i flag di stato del file, l'offset corrente ed un puntatore ad un elemento della tabella del v-node;
3. Ciascun file aperto (o device) ha una struttura v-node → contiene informazioni sul tipo di file e sulle funzioni che operano su di esso (informazioni in i-node).



11.8 Accesso ad un file

Quando un processo accede ad un file mediante una write, l'elemento della tabella del file relativo all'offset viene aggiornato e se necessario viene aggiornato l'i-node.

Se il file è aperto con `O_APPEND` un flag corrispondente è messo nella tabella del file (ogni write alla fine del file).

Una chiamata a `Iseek` modifica solo l'offset corrente del file e non viene eseguita nessuna operazione di I/O.

Se si chiede di posizionarsi alla fine del file, il valore corrente dell'offset nella tabella del file viene preso dal campo della tavola di i-node che descrive la dimensione.

11.9 Programma A

```
1 strcpy(string, "aaaaaaaaaa\n");
2 fd = open("testfile", O_RDWR | O_CREAT | O_APPEND, S_IRUSR | S_IWUSR);
3
4 if (fd < 0){
5     perror("Errore di apertura");
6     exit(1);
7 }
8 do {
9     if (write(STDOUT_FILENO, "Comando:", 8) < 8){
10         perror("write error");
11     }
12
13     input=getchar();
14     __fpurge(stdin);
15     string[0]=input;
16     write(fd, string, 10);
17     lseek(fd,(off_t) 3, SEEK_SET); // sposta l'offset a "INIZIO" file
18     if(write(STDOUT_FILENO, "Eseguito\n", 9) < 9)
19         perror("write error");
20
21 } while (input != "f");
22
23 close(fd);
```

11.10 Programma B

```
1 strcpy(string, "bbbbbbbbbb\n");
2 fd = open("testfile", O_RDWR | O_CREAT , S_IRUSR | S_IWUSR);
3
4 lseek(fd, 0, SEEK_END); // sposta l'offset a FINE file
5
6 do {
7     if(write(STDOUT_FILENO, "Comando:", 8) < 8)
8         perror("write error");
9     input=getchar();
10    __fpurge(stdin);
11    string[0] = input;
12    if(write(fd, string, 10) < 10)
13        perror("write error");
14    if(write(STDOUT_FILENO, "Eseguito\n", 9 ) <9 )
15        perror("write error su stdout");
16 } while(input != "f");
17 close(fd);
```

11.11 Duplicazione di File descriptor

```
1 #include <unistd.h>
2 int dup (int filedes); // Ritorna un file descriptor che punta allo stesso file indirizzato da
   filedes
3 int dup2 (int filedes, int filedes2); // Prende in input filedes2 e il file descriptor da unire
   nella duplicazione
```

Il valore ritornato da `dup` è il minimo file descriptor non utilizzato

Se `filedes2` è aperto, `dup2` chiude il file prima di duplicare il descrittore `filedes`. Se `filedes = filedes2` ritorna `filedes` e non chiude

`dup2` è una operazione atomica.

11.12 Open e Dup

```
1 fd1 = open("file", O_RDONLY);
2 fd2 = dup(fd1)
```

11.13 Otttenere info su File

```
1 int stat( const char *file_name, struct stat *buf);
2 int lstat( const char *file_name, struct stat *buf);
3 int fstat(int filedes, struct stat *buf);
```

Restituiscono 0 in caso di successo, -1 in caso contrario

Queste system call prendono in input un puntatore ad una struttura stat che conterrà le informazioni sul file.

- ▶ `stat` e `lstat` prendono in input il nome del file
 - ◊ `lstat` da informazioni sui link simbolici (info su link simbolici, non sul file linkato).
- ▶ `fstat` prende in input il file descriptor del file → il file deve essere aperto.

11.14 Struttura stat

```
1 struct stat{
2     mode_t st_mode; // file type e permessi
3     uid_t st_uid; // user ID del proprietario
4     gid_t st_gid; // group ID del proprietario
5     ino_t st_ino; // inode number
6     dev_t st_dev; // device number (FS)
7     dev_t st_rdev; // device type (if inode device)
```

```

8   nlink_t st_nlink; // numero di link
9   off_t st_size; // dimensione totale in bytes
10  time_t st_atime; // ultimo accesso
11  time_t st_mtime; // ultima modifica
12  time_t st_ctime; // ultimo cambio
13  blksize_t st_block; // blocksize per I/O
14  blkcnt_t st_blocks; // numero di blocchi allocati
15 }

```

11.15 Tipo di file

- ▶ File regolari
 - ◊ Contiene dati di qualche tipo;
 - ◊ Comprende anche i file eseguibili;
- ▶ Directory file
 - ◊ Contiene nomi e puntatori a inode;
 - ◊ È necessario utilizzare system call specifiche per manipolarlo;
- ▶ Block Special file → rappresenta particolari device (es dischi);
- ▶ Character Special File → rappresentano particolari device (es scheda audio);
- ▶ FIFO (o pipe) → utilizzati per la comunicazione tra processi;
- ▶ Socket → utilizzati per comunicazione tra processi;
- ▶ Link Simbolici (hard e soft).

Il tipo di file associato a un pathname o un file descriptor è codificato nel campo **st_mode** della struttura **stat**.

Per interpretare **st_mode** si utilizzano le seguenti macro:

- ▶ **S_ISREG(m)** → regular file;
- ▶ **S_ISDIR(m)** → directory;
- ▶ **S_ISCHR(m)** → Character device;
- ▶ **S_ISBLK(m)** → block device;
- ▶ **S_ISFIFO(m)** → FIFO;
- ▶ **S_ISLNK(m)** → link simbolico;
- ▶ **S_ISSOCK(m)** → socket.

Le macro prendono come argomento il campo **st_mode**.

11.16 Accesso ai file

Il campo `st_mode` codifica i permessi di accesso ai file.

Per accedere ad un file occorre: → Avere diritto di esecuzione su TUTTE le directory nel path (il permesso di lettura permette di leggerne il nome ma non di aprirlo)

Per creare un file in una directory → Permessi di scrittura sulla directory.

Per cancellare un file in una directory → Permessi di scrittura sulla directory.

L'accesso ai file è regolato dalla seguente sequenza:

- ▶ Se l'effective user del processo è 0 (superuser) → OK;
- ▶ Se l'effective user del processo coincide con l'owner del file
 - ◊ Controlla i permessi per l'utente ed in caso nega l'accesso;
- ▶ Se l'effective group del processo è uguale al group del file
 - ◊ Controlla i permessi del gruppo ed in caso nega l'accesso;
- ▶ Altrimenti controlla i permessi per gli altri.

11.17 Funzione access

```
1 #include <unistd.h>
2 int access(const char *pathname, int mode); // Restituisce 0 in caso di successo, -1 in caso contrario
```

Controlla i permessi di accesso ad un file in base ad UID e GID reali → normalmente valgono quelli effettivi.

Il parametro mode può assumere i valori:

- ▶ `R_OK`, `W_OK`, `X_OK` → lettura, scrittura o esecuzione;
- ▶ `F_OK` → esistenza.

11.18 chmod fchmod

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 int chmod(const char *path, mode_t mode);
4 int fchmod(int fildes, mode_t mode);
```

Consentono di modificare i permessi di accesso ai file:

- ▶ `chmod` prende in input un pathname;
- ▶ `fchmod` prende in input un file descriptor (il file deve essere aperto).

Il parametro *mode* può essere una combinazione delle seguenti costanti:

- ▶ `S_ISUID`, `S_ISGID`, `S_ISVTX` → set user id exe, group id exe, saved text;
- ▶ `S_IRWXU`, `S_IRUSR`, `S_IWUSR`, `S_IXUSR` → owner;
- ▶ `S_IRWXG`, `S_IRGRP`, `S_IWGRP`, `S_IXGRP` → group;
- ▶ `S_IRWXO`, `S_IROTH`, `S_IWOTH`, `S_IXOTH` → others.

11.19 chown

```

1 #include <sys/types.h>
2 #include <unistd.h>
3 int chown(const char *path, uid_t owner, gid_t group);
4 int fchown(int fd, uid_t owner, gid_t group);
5 int lchown(const char *path, uid_t owner, gid_t group);

```

Modificano il campo `st_uid` `st_gid` del file indicato dal pathname (chown e lchown) o dal file descriptor (fchown).

Se il parametro *owner* o *group* è uguale a `-1`, il campo corrispondente non viene modificato.

In molti sistemi, solo un processo del superuser può modificare il campo `st_uid`.

Un processo può modificare il gruppo se:

- ▶ È owner del file e
- ▶ Il parametro *group* è uguale all'effective GID del processo o ad uno dei gruppi alternativi.

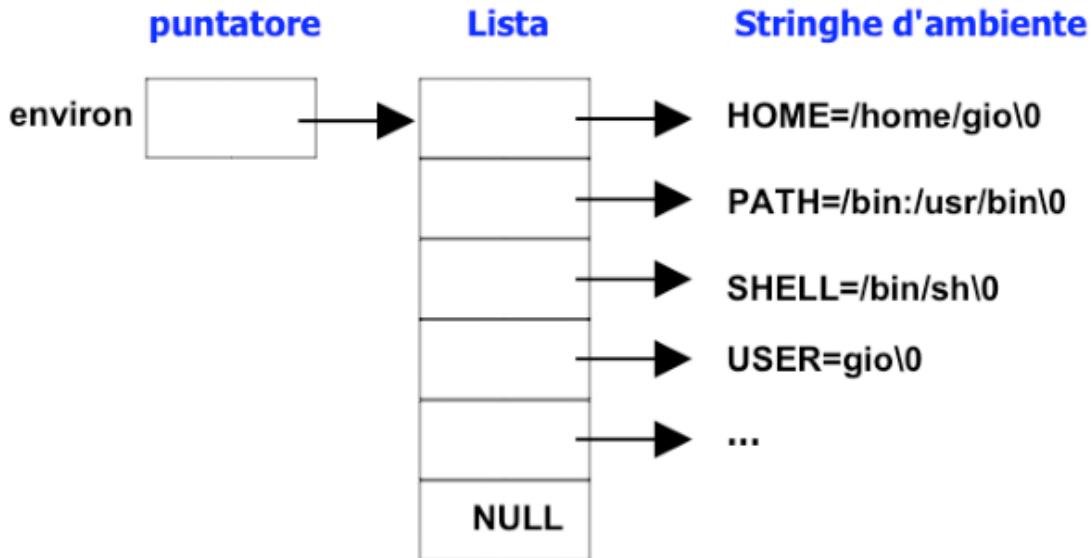
11.20 Lista di Ambiente

Serve alle applicazioni per definire il contesto in cui operano.

È un array di puntatori a carattere, in cui l'ultimo puntatore punta a NULL. Ogni puntatore contiene l'indirizzo di una stringa del tipo *nome = valore* detta **stringa di ambiente**.

L'indirizzo dell'array di puntatori è contenuto nella variabile globale **environ**:

```
extern char **environ
```



11.21 Variabili di Ambiente

Le variabili di ambiente sono definite e modificate operando sulle stringhe d'ambiente, grazie ad opportune funzioni. Le più importanti sono:

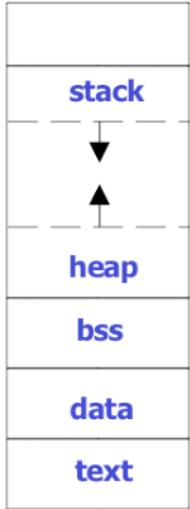
```

1 #include <stdlib.h>
2 /*
3  getenv restituisce il puntatore alla stringa valore,
4  associata al nome della variabile nome nella stringa di
5  ambiente nome=valore. Se nome non esiste restituisce il puntatore nullo
6 */
7 char *getenv(const char *name);
8 /*
9  putenv aggiunge alla lista d'ambiente la stringa string della forma
10 nome=valore. Se nome esiste ne aggiorna il valore
11 */
12 int putenv(char *string)
13

```

11.22 Layout in memoria di un programma

Questo segmento dell'area di memoria riserva ad un programma serve per la memorizzazione degli argomenti della linea di comando e delle variabili di ambiente.



Lo stack serve a memorizzare le variabili locali e le informazioni relative alle chiamate di funzioni

Viene implementato con una coda LIFO, per permette l'uso annidato e ricorsivo delle funzioni).

Lo heap è dove viene effettuata l'allocazione dinamica di funzioni

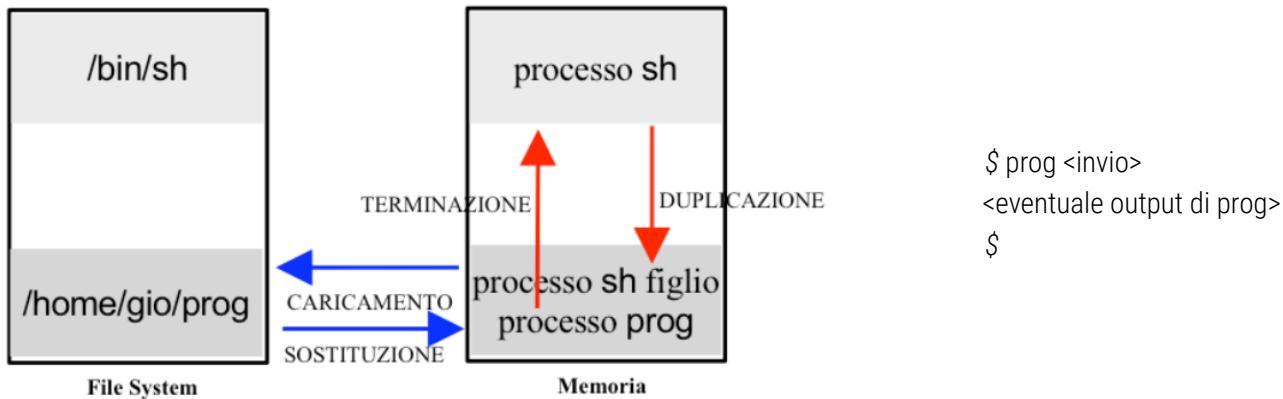
Viene implementata sempre con una coda LIFO, ed è gestita a basso livello dalla syscall **sbrk**

Le variabili globali non inizializzate, analogamente ai puntatori a variabili globali, vengono memorizzati nel segmento **bss**, detto anche **uninitialized data segment**. I dati in tale segmento sono inizializzati dal kernel come valori numerici pari a 0 o puntatori nulli prima che il programma vada in esecuzione.

L'area data contiene le variabili globali ed inizializzate del programma.

L'area text contiene le istruzioni macchina relative al programma. Questa area è in sola lettura, in quanto è condivisa tra tutti i processi che eseguono lo stesso codice binario.

11.23 Esecuzione di un programma



12 Lezione del 20-10

12.1 Identificazione di Processi

Ogni processo ha un pid ed un Parent's pid (ppid) → i processi quindi formano un albero.

Init è la radice dell'albero:

- ▶ viene lanciato direttamente dal kernel;
- ▶ Non ha padre;
- ▶ Ha pid 1.

Quando un processo termina, i suoi figli diventano figli di init.

12.2 Ottenerne pid e ppid

```
1 pid_t getpid(void);
2 pid_t getppid(void);
```

Restituiscono pid e ppid, rispettivamente.

Non possono fallire.

Di solito `pid_t` è sinonimo di `int pid_t`.

12.3 Identificazione dei processi

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 pid_t getpid(void); // Identificativo del processo
4 pid_t getppid(void); // Identificativo del genitore
5
6 uid_t getuid(void); // Credenziali utente reale
7 uid_t geteuid(void); // Credenziali utente effettivo
8
9
10 gid_t getgid(void); // Credenziale gruppo reale
11 gid_t getegid(void); // Credenziale gruppo effettivo
```

12.4 Creazione di processi

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 pid_t fork(void);
4 pid_t vfork(void); // Da usare con exec, lavora nello spazio del genitore e aspetta il figlio
```

L'unico modo per istruire il kernel a creare un nuovo processo è di chiamare la funzione `fork` (`vfork`) da un processo esistente.

Il processo creato è detto figlio. Il processo chiama `fork` (`vfork`) viene detto genitore.

Ogni chiamata a `fork` (`vfork`) ha 2 ritorni:

- ▶ Al genitore viene restituito l'identificativo del figlio;
- ▶ Al figlio viene restituito l'identificativo 0.

es.

```

1 pid_t pid;
2
3 if ((pid=fork()) < 0)
4     perror("fork"), exit(1);
5 else if (pid !=0){
6     // codice del padre
7 }
8 else {
9     // codice del figlio
10}

```

12.5 Padri e figli

Il figlio procede indipendentemente dal padre:

- ▶ Memoria → il figlio ottiene una copia nuova della memoria del padre (variabili globali e locali);
- ▶ File aperti → I descrittori vengono copiati come con `dup`, i processi condividono l'offset;
- ▶ Segnali → Per ogni segnale, il figlio continua ad avere la stessa reazione del padre.

12.6 Creazione di Processi-fork

Ad una chiamata `fork` il kernel esegue le operazioni seguiti:

- ▶ Alloca uno spazio nella tabella dei processi per il figlio;
- ▶ Assegna un PID al figlio, unico nel sistema;
- ▶ Fa una copia dell'immagine del genitore, ad eccezione dei segmenti di memoria condivisi;
- ▶ Incrementa i contatori del file del genitore, per registrare che anche il figlio possiede tali file;
- ▶ Assegna al processo figlio lo stato READY;
- ▶ Restituisce il PID del figlio al genitore e il PID 0 al figlio;
- ▶ A seconda della routine di allocazione può
 - ◊ Rimanere nel genitore;
 - ◊ Trasferire il controllo al figlio;
 - ◊ Trasferire il controllo ad un altro processo.

12.7 Fork e File

Una caratteristica della chiamata `fork` è che tutti i descrittori che sono aperti nel processo parent sono duplicati nel processo child.

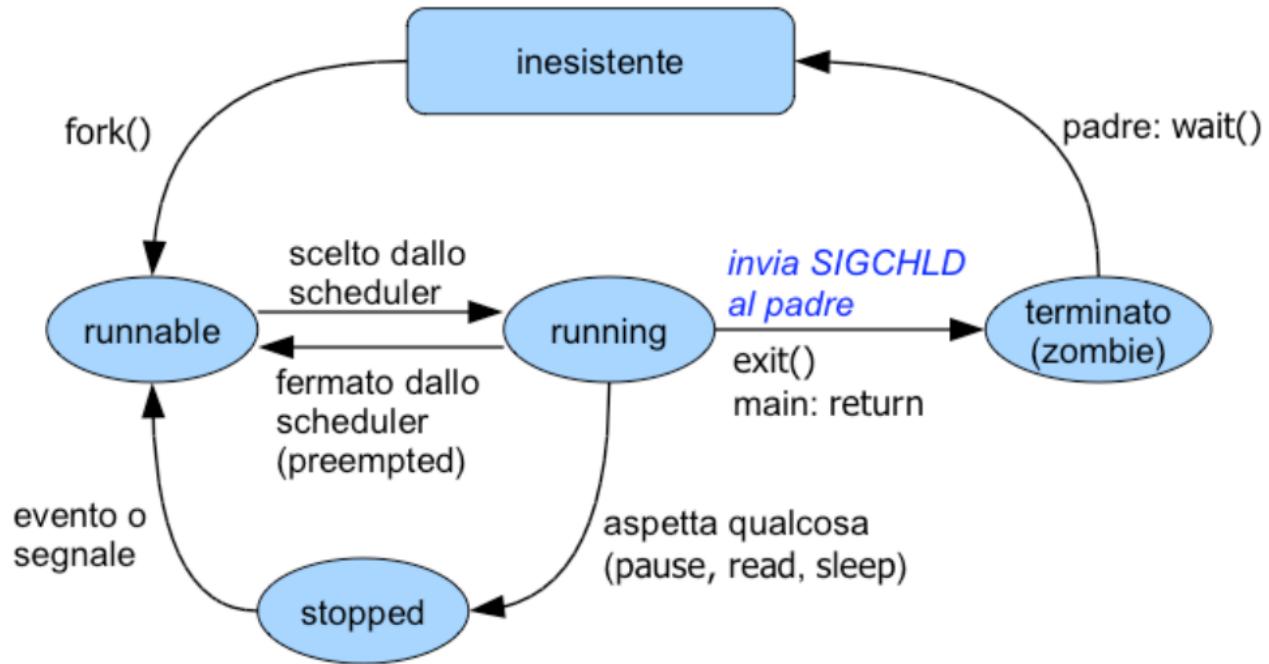
Prima e dopo il fork:



12.8 Fork e file

Padre e figlio condividono lo stesso file offset.

12.9 Ciclo di vita del processo



12.10 Terminazione di un processo

Viene invocato il segnale di `SIGCHLD` al padre.

Il processo diventa uno *zombie* finché il padre non chiama una `wait/waitpid`.

Esistono 3 modi per terminare in modo NORMALE:

- ▶ Eseguire un return da main (è equivalente a chiamare `exit`);
- ▶ Chiamare la funzione `exit`
 - ◊ `void exit(int status);`
 - Invoca di tutti gli exit handlers che sono stati registrati;
 - Chiude di tutti gli I/O stream standard;
 - È specificato in ANSI C;
 - ◊ Chiamare la system call `_exit`
 - `void _exit(int status);`
 - Ritorna al kernel immediatamente;
 - È chiamata come ultima operazione da exit;
 - È specificata nello standard POSIX.1

Esistono 2 modi per terminare in modo ANORMALE:

- ▶ Quando un processo riceve certi segnali
 - ◊ Generati dal processo stesso;
 - ◊ Generati da altri processi;
 - ◊ Generati dal kernel;
- ▶ Chiamando `abort`
 - ◊ `void abort();`
 - ◊ La chiamata `abort` costituisce un caso speciale del primo caso dei 3 sopra elencati, in quanto genera il segnale SIGABT.

12.11 Processi zombie

Nel caso in cui il processo padre termini prima del figlio, questo viene *adottato* dal processo `init` (PID 1), in quanto il kernel vuole evitare che un processo diventi orfano.

Quando un processo termina, il kernel esamina la tabella dei processi per vedere se aveva figli → in tal caso il PPID di ogni figlio viene posto a 1.

12.12 Aspettare un figlio

¹ `pid_t wait(int *status);`

Blocca il processo finché un figlio termina → non blocca se c'è un figlio zombie.

Restituisce il pid del processo terminato:

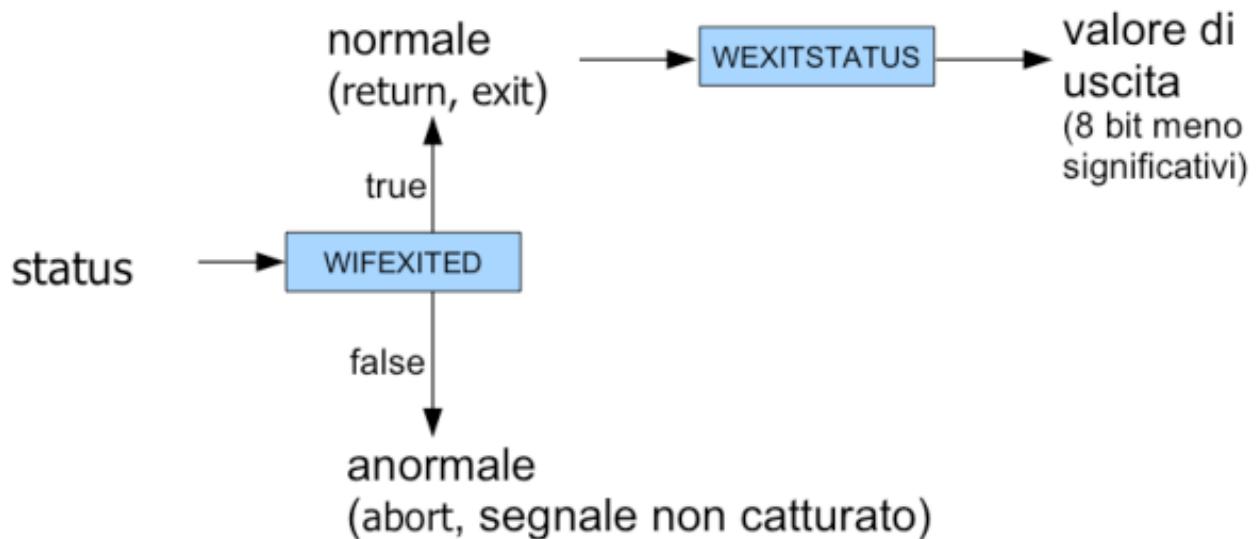
- -1 in caso di errore (ad es se un processo non ha figli).

status contiene il valore di uscita del figlio (nel caso non ci interessi, passiamo NULL).

```
1 pid_t waitpid(pid_t pid, int *status, int options);
2 // Come wait, ma aspetta un figlio specifico
3 // options può essere lasciato a zero
```

Entrambe sono utilizzate per avere informazioni sullo stato del processo.

12.13 Stato del figlio



12.14 Stato di uscita del Figlio

```
1 int status;
2
3 wait(&status);
4
5 if (WIFEXITED(status))
6     printf("valore di uscita: %d\n", WEXITSTATUS(status));
7 else
8     printf("Terminazione anomala\n");
```

12.15 Azioni del kernel a terminazione di processo

Qualsiasi sia lo status di uscita dei processi, il kernel effettua le seguenti operazioni:

- ▶ Rimozione della memoria utilizzata dal processo
- ▶ Chiusura dei descrittori aperti

12.16 Valori di ritorno a terminazione del processo

- ▶ Exit status → valore che viene passato ad `exit` (e `_exit`) e che notifica il padre su come è terminato il processo;
- ▶ Termination status
 - ◊ Valore che viene generato dal kernel nel caso di una terminazione **normale/anormale**;
 - ◊ Nel caso si parli di terminazione anormale, si specifica la ragione per questa terminazione anormale;
 - ◊ L' `exit status` è convertito dal kernel in `termination status` quando alla fine viene chiamata `_exit`.

12.17 Status `wait/waitpid`

Se `statloc` non è NULL, l'intero cui esso punta rappresenta lo **stato di terminazione** del processo atteso, secondo una codifica che dipende dall'implementazione.

POSIX prevede che tale stato possa essere rilevato grazie ad opportune macro definite in `<sys/wait.h>`

Macro	Descrizione
<code>WIFEXITED(status)</code>	Vera se il figlio è terminato normalmente; in tal caso <code>WEXITSTATUS(status)</code> restituisce lo stato di uscita
<code>WIFSIGNALED(status)</code>	Vera se il figlio è terminato a causa di un segnale in tal caso <code>WTERMSIG(status)</code> ne restituisce il numero
<code>WIFSTOPPED(status)</code>	Vera se il figlio è in stato di stop; allora il numero del segnale di stop è dato da <code>WSTOPSIG(status)</code>
<code>WIFCONTINUED(status)</code>	Vera se il figlio ha ripreso l'elaborazione dopo uno stato di stop

12.18 Esecuzione di programmi

Avviene attraverso la chiamata ad una delle funzioni exec.

Le funzioni `exec` non generano un nuovo processo, ma modificano il layout in memoria del processo chiamante per l'esecuzione del nuovo programma.

Quando un processo chiama una delle funzioni `exec`, le aree di memoria `text`, `data`, `heap` e `stack` relative a processo corrente vengono sostituite in base al nuovo programma.

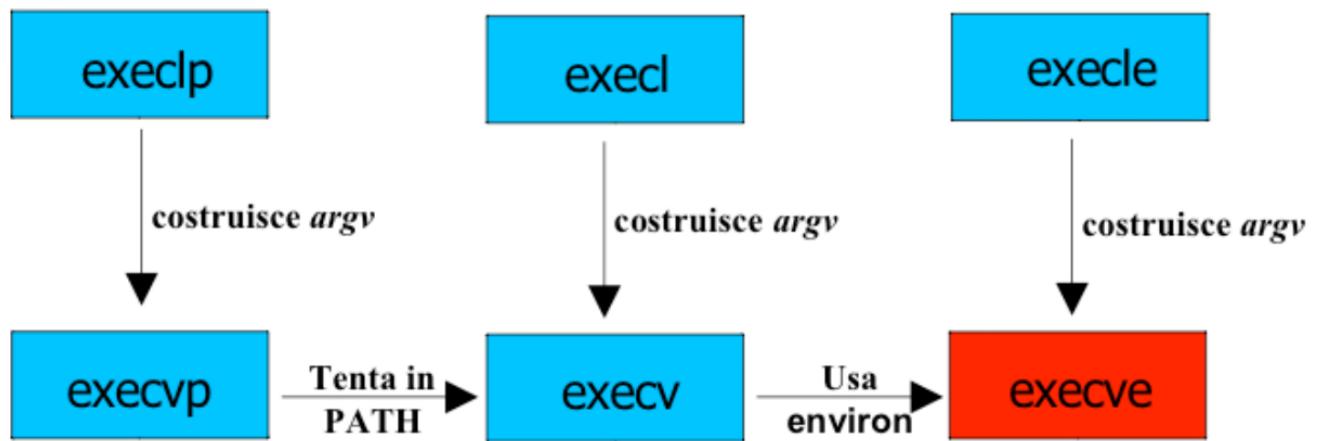
Il nuovo programma incomincia la propria esecuzione a partire dalla funzione `main`.

12.19 La famiglia di system call `exec`

Se `fork` fosse l'unica primitiva per creare nuovi processi, la programmazione in ambiente UNIX sarebbe ostica, dato che si potrebbero creare soltanto copie dello stesso processo.

La famiglia di primitive `exec` può essere utilizzata per superare tale limite in quanto le varie system call `exec` permettono di iniziare l'esecuzione di un altro programma sovrascrivendo la memoria del processo chiamante

In realtà tutte le funzioni chiamano in ultima analisi `execve` che è l'unica vera system call della famiglia. Le differenze sono su come vengono passati i parametri.



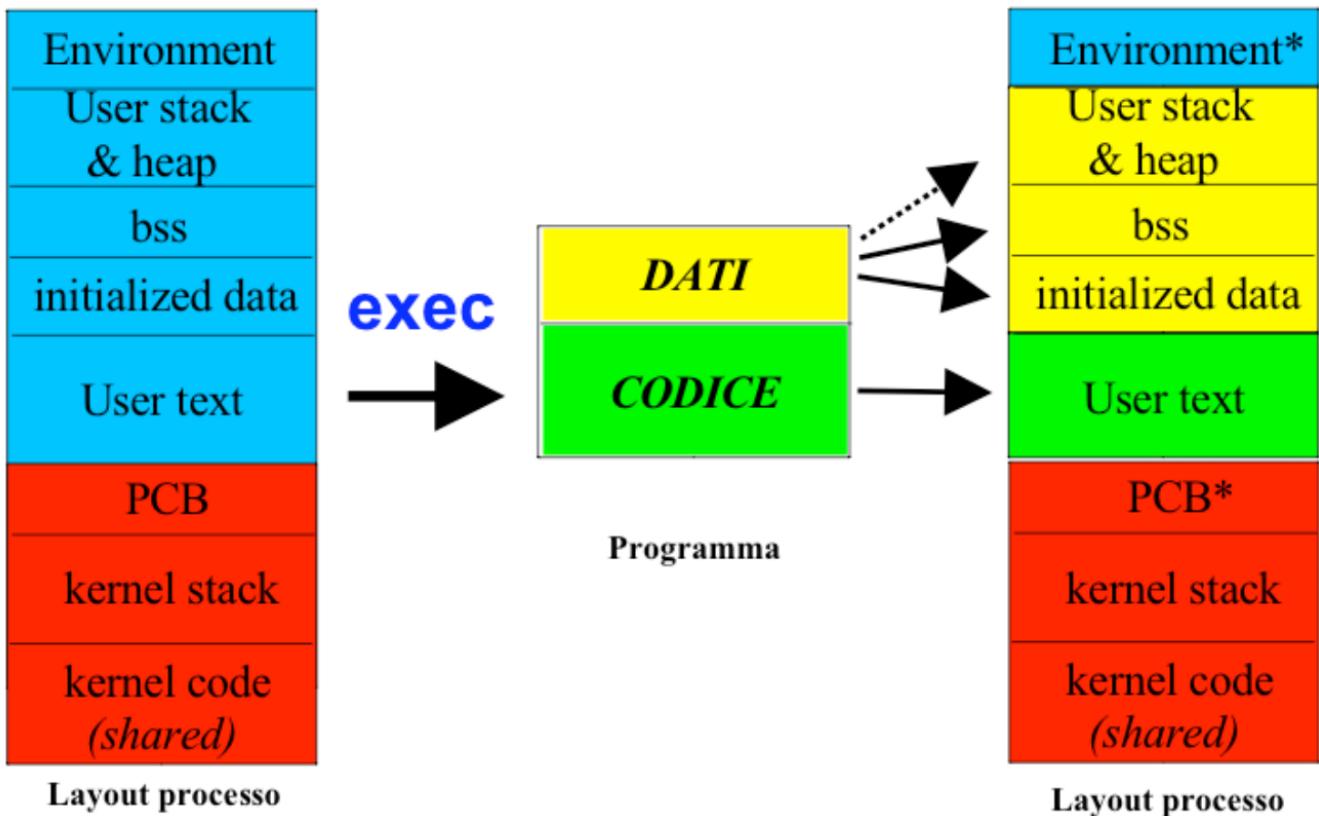
12.20 Esempio di utilizzo di `execl`

```
1 #include <stdio.h>
2 #include <unistd.h>
3 main(){
4     printf("Esecuzione di ls\n");
5     execl("/bin/ls", "ls", "-l", (char *)0);
6     perror("La chiamata di execl ha gerato un errore\n");
7     exit(1);
8 }
```

- ▶ `execl` elimina il programma originale sovrascrivendolo con quello passato come parametro;

- ▶ Quindi le istruzioni che seguono una chiamata `exec1` verranno eseguite soltanto in caso si verifichi un errore durante l'esecuzione di quest'ultima ed il controllo ritorni al chiamante.

12.21 Esecuzione di Programmi



12.22 Chiamata alla system call `exec`

Quando un processo chiama una delle system call `exec`:

- ▶ Il processo viene rimpiazzato completamente da un nuovo programma;
- ▶ Il nuovo programma inizia a partire dalla sua funzione main;
- ▶ Il process ID non cambia.

12.23 Famiglia system call `exec`

```

1 int execl(char *pathname, char *arg0, ...);
2 int execv(char *pathname, char *argv[]);
3 int execle(char *pathname, char *arg0, ..., char* envp[]);
4 int execve(char *pathname, char *argv[], char* envp[]);
5 int execlp(char *filename, char *arg0, ...);

```

```
6 int execvp(char *filename, char *argv[]);
```

12.24 Ambiente di un processo

Insieme di stringhe (terminate da \0).

Rappresentato da un vettore di puntatori a caratteri terminato da un puntatore nullo.

Ogni puntatore (che non sia quello nullo) punta ad una stringa della forma → *identificatore* = *valore*

Per accedere all'ambiente da un programma C, è sufficiente aggiungere il parametro `envp` a quelli del main

```
1 #include <stdio.h>
2 main(int argc, char **argv, char **envp){
3     while(*envp);
4     printf("%s\n", *envp++);
5 }
```

Oppure usare la variabile globale:

```
1 extern char **environ;
```

L'ambiente di default di un processo coincide con quello del processo padre.

Per specificarne uno diverso è necessario usare una delle 2 varianti della famiglia di `exec`, memorizzando in `envp` l'ambiente desiderato:

```
1 execle(path, arg0, arg1, ..., argn, (char *)0, envp);
2 execve(path, argv, envp);
```

Esiste una funzione della libreria standard che consente di cercare in `environ` il valore corrispondente ad una specifica variabile d'ambiente:

```
1 #include <stdlib.h>
2 char *getenv(const char *name);
```

12.25 Current working directory e root directory

Ad ogni processo è associato una current working directory che viene ereditata dal processo padre.

La chiamata di sistema seguente permette di cambiarla:

```
1 #include <unistd.h>
2 int chdir(const char *path);
```

Ad ogni processo inoltre è associata una root directory che specifica il punto di inizio del FS visibile a processo stesso. Per modificarla:

```
1 #include <unistd.h>
2 int chroot(const char *path);
```

12.26 Race Condition

Si verifica quando più processi elaborano dati **condivisi** e l'effetto dell'insieme di siffatte elaborazioni dipende dall'ordine in cui i processi sono eseguiti.

L'ordine in cui vengono elaborate le istruzioni per un genitore e un figlio dopo un fork in generale dipende dallo scheduler e dal carico del sistema.

Una chiamata fork può causare una race condition, se le istruzioni relative al genitore ed al figlio **operano su dati condivisi**.

Il rilevamento **in base a run** di una race condition del tipo suddetto può essere molto difficile.

12.27 Funzione vfork()

```
1 #include <unistd.h>
2 pid_t vfork(void);
```

Simile a **fork**, ma non copia lo spazio di indirizzamento → viene eseguito nello spazio del genitore finché il figlio non esegue **exec** o **exit** (fin a quel momento il figlio viene eseguito per primo).

vfork utilizzato con **exec** per l'esecuzione di un programma.

12.28 La funzione system

```
1 int system(char* command);
```

Esegue un comando, aspettando la sua terminazione.

È una funzione della libreria standard definita in ANSI C (non è una system call, anche se svolge una funzione analoga alla **fork+exec**).

È comodo per eseguire un comando **UNIX** da un programma C.

13 Lezione del 22-10

13.1 Scrollview

Contiene una lista di View che possono essere scrollate dall'utente

13.2 Elementi Grafici

- ▶ Riempiono i Layout;
- ▶ Permettono l'interazione con l'utente;
- ▶ Possono generare eventi.

13.3 TextView

- ▶ Visualizza stringhe sulla schermata;
- ▶ Utili a spiegare l'interfaccia;
- ▶ Il testo può essere selezionabile o meno

```
1 <TextView  
2     android:id="@+id/myTextView" <!-- utile per richiamarlo nel codice java -->  
3     android:layout_width="wrap_content" <!-- Grande quanto il suo contenuto -->  
4     android:layout_height="wrap_content"  
5     android:text="@string/string_code" <!-- Fa riferimento ad una risorsa (il contenuto) -->  
6     android:textColor="@android:color/black"  
7     android:textSize="20sp"/>
```

13.4 EditText

- ▶ Permette l'inserimento di dati;
- ▶ Eredita da TextView.

```
1 <EditText  
2     android:id="@+id/myEditText" <!-- utile per richiamarlo nel codice java -->  
3     android:layout_width="wrap_content" <!-- Grande quanto il suo contenuto -->  
4     android:layout_height="wrap_content"  
5     android:ems="19" <!-- Fa riferimento ad una risorsa (il contenuto) -->  
6     android:hint="@string/string_code" <!-- Analogico ad un placeholder -->  
7     android:inputType="number"/>
```

13.5 Button

Realizzano l'interazione con l'utente (eredita da TextView).

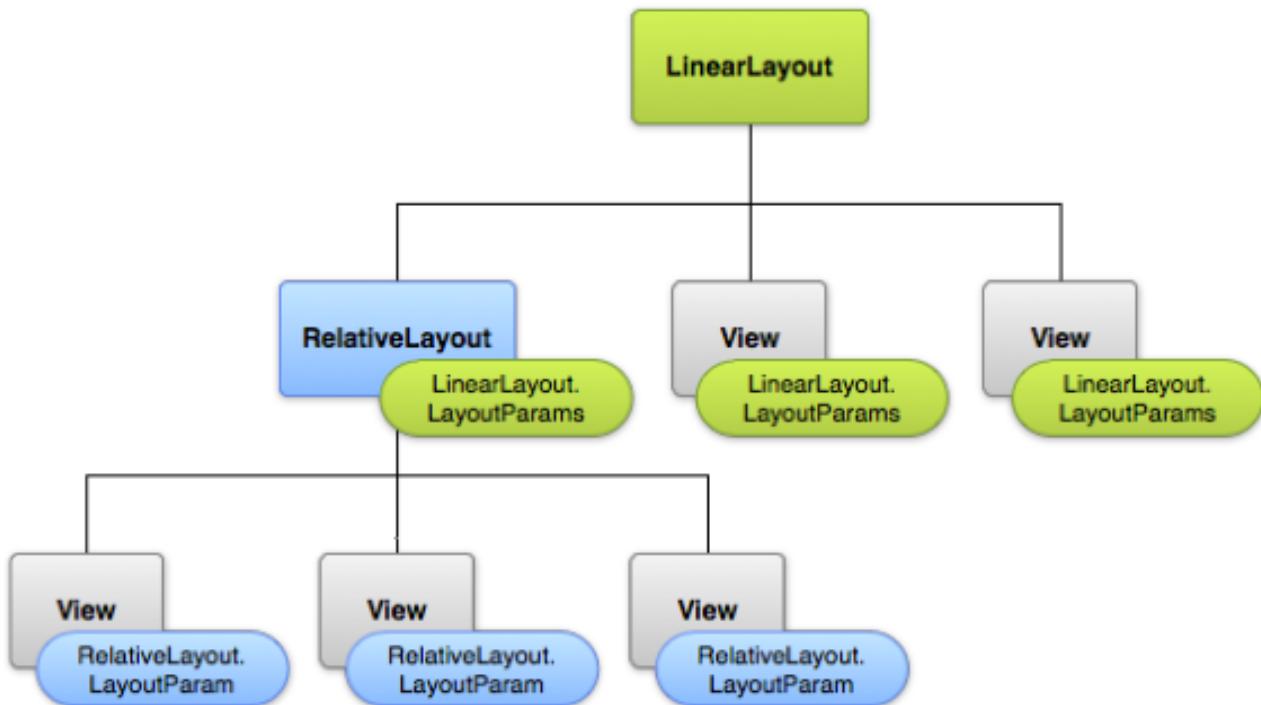
```
1 <Button  
2     android:id="@+id/button"  
3     android:layout_width="wrap_content"  
4     android:layout_height="wrap_content"  
5     android:text="@string/string_code"
```

6

```
    android:onClick="OnClick"/> <!-- richiede di specificare il contesto (ne riduce la portabilità)
-->
```

13.6 Parametri di layout

`match_parent` VS `wrap_content`



13.7 Assegnare UI

```
@Override
protected void onCreate(Bundle savedInstanceState){
    super.onCreate(savedInstanceState);

    // Metodo per invocare una view o un layout
    // ID del layout da caricare
    setContentView(R.layout.activity_main);
}
```

13.8 findViewById

Recuperare gli elementi grafici del layout nelle Activity.

```

1 View findViewById(int reference)
2
3 // View -> superclasse
4 // reference -> ID (R.id.submit_button)

```

Stesso metodo per tutti i tipi di elementi.

13.9 Listener

L'interazione genera eventi che possono essere catturati

```

1 final Button button = (Button) findViewById(R.id.button_id);
2 button.setOnClickListener(new View.OnClickListener(){
3     public void onClick(View v){
4         // Perform action on click
5     }
6 });

```

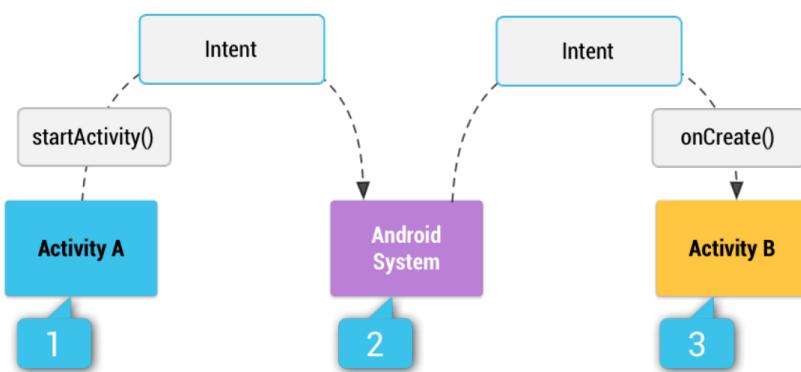
13.10 Constraint Layout

Simile a RelativeLayout, ma più flessibile.

Non richiede ViewGroup innestati.

Risulta meglio integrato nel Layout Editor di Android Studio.

13.11 Intent



Permettono di scambiare messaggi tra diverse activity.
Possono richiamare altre applicazioni.

13.12 Intent-filter

Si specificano nel manifest:

```

1 <activity>

```

```
2 <intent-filter/>
3 <intent-filter>
4   <action android:name="" />
5   <category android:name="" />
6 </intent-filter>
7 </activity>
```

13.13 Intent implicito ed esplicito

```
1 Intent intenzione = new Intent(AZIONE, DATI);
2 startActivity(intenzione);
```

Dove AZIONE:

- ▶ Intent.ACTION_MAIN;
- ▶ Intent.ACTION_VIEW;
- ▶ Intent.ACTION_DIAL;
- ▶ Intent.ACTION_EDIT.

Dove DATI → Uri.parse(A:B) Android cerca le app in grado di eseguire l'azione (intent resolution)

Per un intent esplicito Android lancia l'activity specificata con eventuali parametri.

```
1 Intent nuovaPagina = new Intent(this, Pagina1.class);
2 // this -> Classe/Contesto di partenza
3 // Pagina1.class -> Classe di destinazione (Activity)
4 startActivity(nuovaPagina);
```

13.13.1 `getAction()`

```
1 public void onCreate(Bundle savedInstanceState){
2 ...
3   Intent myIntent = getIntent(); // Recupera l'intent che ha creato l'activity
4
5   // con getAction ottengo l'azione che è stata richiesta
6   // Può restituire NULL (in caso di Intent esplicito)
7   if (Intent.ACTION_VIEW.equals(myIntent.getAction())){
8     // Activity lanciata da qualcuno con ACTION_VIEW
9   }
10 ...
11 }
```

13.14 Passaggio di Parametri

```
1 public void onCreate(Bundle savedInstanceState){  
2     ...  
3     Intent next = new Intent(this, OtherActivity.class);  
4     next.putExtra("key", "value"); // Non si è obbligati a usare stringhe!  
5     startActivity(next);  
6     ...  
7 }
```

```
1 public void onCreate(Bundle savedInstanceState){  
2     ...  
3     Intent next = new Intent(this, OtherActivity.class);  
4     MyClass par = new MyClass(); // Implementa Serializable  
5     next.putExtra("key", par); // Non si è obbligati a usare stringhe!  
6     startActivity(next);  
7     ...  
8 }
```

```
1 public void onCreate(Bundle savedInstanceState){  
2     ...  
3     Intent next = new Intent(this, OtherActivity.class);  
4     if (next.getStringExtra("key")){  
5         String val = next.getStringExtra("key");  
6     }  
7     ...  
8 }
```

```
1 public void onCreate(Bundle savedInstanceState){  
2     ...  
3     Intent next = new Intent(this, OtherActivity.class);  
4     if (next.getStringExtra("key")){  
5         MyClass m = (MyClass) next.getSerializableExtra("key");  
6     }  
7     ...  
8 }
```

14 Lezione del 25-10

14.1 Esercitazione 1

Lettura di un file e scrittura a byte alterni

```
1 #include <stdio.h>  
2 #include <stdlib.h>
```

```

3  ||#include <sys/types.h>
4  ||#include <errno.h>
5  ||#include <unistd.h>
6  ||#include <fcntl.h>
7  ||#include <sys/stat.h>
8
9  #define BUFSIZE 256;
10 ||int main(int argc, char* argv[]){
11   ||char buffer[BUFSIZE]; // in questo esercizio è sprecato (leggiamo un carattere alla volta)
12   ||if (argv[1] == NULL){
13     ||printf ("Error: Usage %s <filename>\n", argv[0]);
14     ||exit(1); // per una maggiore portabilità conviene utilizzare le macro relative ad exit
15   }
16   ||int fd, n;
17   ||if((fd= open(argv[1], O_RDONLY)) == -1){
18     ||perror("Open Error");
19     ||exit(2);
20   }
21   ||while((n = read(fd, buffer, 1)) != 0){
22     ||if (n == -1){
23       ||perror ("Read Error");
24       ||exit(3);
25     }
26     ||if (write(STDOUT_FILENO, buffer, 1) == -1){
27       ||perror ("Write Error");
28       ||exit(4);
29     }
30
31     ||if (lseek(fd, 1, SEEK_CUR) == -1){
32       ||perror ("Seek Error");
33       ||exit(5);
34     }
35   }
36   ||printf("\n");
37   ||close(fd);
38
39   ||return 0;
40 }
```

14.2 Esercitazione 2

15 Lezione del 27-10

15.1 Segnali ed allarmi

Un segnale è un interrupt SW.

La terminologia corretta è *exception* mentre *interrupt* è usata solo per gli interrupt HW.

Consente la comunicazione asincrona tra processi e/o tra device e processo.

Ogni segnale ha un proprio nome:

- ▶ Tutti i nomi cominciano per /SIG/M
- ▶ Definiti in `<signal.h>`;
- ▶ Associati ad interi positivi.

Vengono inviati in modo asincrono → non è possibile sapere quando il processo riceverà un segnale.

15.2 Caratteristiche dei Segnali

Ogni segnale ha un identificatore (inizia con *SIG*)

Numero segnali → 15-40, a seconda delle versioni di UNIX:

- ▶ POSIX (18);
- ▶ Linux (38).

I nomi simbolici corrispondono ad un intero positivo (in `bit/signum.h`).

15.3 Generazione di segnali

- ▶ Pressioni di tasti speciali sul terminale;
- ▶ Eccezioni HW
 - ◊ Divisione per 0;
 - ◊ Riferimento non valido a memoria;
 - ◊ L'interrupt viene generato dall'HW, e catturato dal kernel (questo invia il segnale al processo in esecuzione);
- ▶ System call kill
 - ◊ Permette di spedire un segnale ad un altro processo;
 - ◊ Limitazione → uid del processo che esegue `kill` deve essere lo stesso del processo a cui spedisce il segnale, oppure 0 (utente root).
- ▶ Condizioni SW
 - ◊ Eventi asincroni generati dal SW del SO, non dall'HW della macchina.

15.4 Azioni associate a segnali

È possibile indicare al kernel l'azione da interpretare quando un segnale è generato per un processo:

- ▶ Ignora → valida per quasi tutti i segnali tranne **SIGKILL** e **SIGSTOP**;
 - ◊ Motivo → permette a root di terminare processi;
 - ◊ Segnali HW → comportamento non definito in POSIX se ignorati.
- ▶ Catch del segnale → Indicare una procedura da eseguire (signal handler, che gestisce il problema nel modo più opportuno)
 - ◊ **SIGCHLD** → esegui le operazioni associate alla terminazione di un figlio;
 - ◊ **SIGINT** → (C-c);
 - ◊ Non è possibile intercettare SIGKILL o SIGSTOP.
- ▶ Default → Eseguire l'azione di default
 - ◊ Per molti segnali **critici**, l'azione di default consiste nel terminare il processo;
 - ◊ Può essere generato un file di core (eccetto quando bit set-user-id e set-group-id settati e **uid/gid** sono diversi da **owner/group** o mancano di permessi in scrittura per la directory il core file è troppo grande).

15.5 Alcuni segnali

SIGABRT	(Terminazione, core)	Generato da syscall abort() terminazione anormale
SIGNALRM	(Terminazione)	Generato da un timer settato con la syscall alarm() o la funzione settimer()
SIGBUS	(Non POSIX, terminazione core)	Indica un HW fault (definito da SO)
SIGCHILD	(Default: ignore)	Quando un processo termina, SIGCHILD viene spedito al processo parent. Questo deve definire un signal handler che chiama wait() o waitpid()
SIGFPE	(Terminazione, core)	Eccezione aritmentica, come divisioni per 0
SIGHUP	(Terminazione)	Inviato ad un processo se il terminale connesso viene disconnesso

15.6 Catturare i segnali

Un handler (gestore) è una funzione del tipo:

```
1 void funzione(int num_segnale){  
2     printf("%d", num_segnale);  
3 }
```

Una volta che l'handler termina si torna al punto in cui il programma era stato interrotto

```
1 typedef void (*sighandler_t)(int);  
2 sighandler_t signal(int signum, sighandler_t handler);
```

Restituisce l'impostazione precedente, cioè uno dei seguenti valori:

- ▶ L'indirizzo dell'handler precedente;
- ▶ **SIG_DFL** → Reazione di default;
- ▶ **SIG_IGN** → Ignorare il segnale;
- ▶ **SIG_ERR** → Errore.

`signal(SIGINT, foo)` imposta la funzione `foo` come handler del segnale `SIGINT`.

Si può anche richiedere di ignorare il segnale:

```
signal(SIGINT, SIG_IGN)
```

Ottobre ritornare alla reazione di default:

```
signal(SIGINT, SIG_DFL)
```

15.7 Esempio

```
1 void foo(int num_segnale);  
2 int main(void);  
3 int main(void){  
4     signal(SIGUSR1, foo);  
5     signal(SIGUSR2, foo);  
6     if (signal(SIGKILL, foo) == SIG_ERR) // Errore Certo!  
7         perror("Impossibile intercettare SIGKILL");  
8     for(;;){pause();}  
9 }  
10 void foo(int num_segnale){  
11     if (num_segnale == SIGINT) // Impossibile bloccare con C-c  
12         printf("Segnale INT %d\n", num_segnale);
```

```

13
14 if (num_segnale == SIGUSR1)
15   printf("Segnale USR1 %d\n", num_segnale);
16
17 if (num_segnale == SIGUSR2)
18   printf("Segnale USR2 %d\n", num_segnale);
19 }
```

15.8 Inviare segnali

I segnali si inviano ai processi (o a gruppi di processi identificati da un process group).

Un processo si individua usando il suo PID, che è un intero non negativo:

- ▶ I PID 0 e 1 sono riservati al sistema.

Per visualizzare i PID dei propri processi correnti, possiamo usare *ps -u* e leggere la seconda colonna.

Per inviare un segnale, bisogna averne il permesso (si possono inviare segnali solo ai propri processi).

15.9 Inviare segnali da shell (`kill`)

- ▶ `kill -INT 127` → Invia il segnale SIGINT al processo il cui pid è 127;
- ▶ `kill -1` → Elenca tutti i segnali ed il loro valori numerici;
- ▶ `kill 127` → Equivale a `kill -TERM 127`.

15.10 Inviare segnali in C

```

1 int kill(pid_t pid, int sig);
2
3 // Esempio
4 kill(127, SIGINT); // Invia il segnale SIGINT al processo il cui pid è 127
```

Restituisce 0 in caso di successo e -1 in caso di errore.

Il parametro `pid` può assumere i seguenti valori:

- ▶ $pid > 0$ → Identifica il processo con $PID = pid$;
- ▶ $pid = 0$ → Tutti i processi con group ID pari al group ID del process che esegue la kill;
- ▶ $pid < 0$ → Tutti i processi con group id pari al valore assoluto di pid;
- ▶ $pid = -1$ → Inviato a tutti i processi del sistema per cui il processo che esegue la kill ha il permesso di inviare un segnale.

Il parametro **sig** può assumere i seguenti valori:

- ▶ **sig>0** → È un intero specificato in `<signal.h>`
- ▶ **sig=0** → È utilizzato per verificare se il processo ha i permessi per inviare un segnale **al/i processo/i** specificati da **pid**
 - ◊ Nessun segnale viene inviato;
 - ◊ Utile per verificare l'esistenza di un processo;
 - ◊ NB → UNIX ricicla i pid!

15.11 Impostare una sveglia

```
1 unsigned int alarm(unsigned int seconds);  
2  
3 // Esempio  
4 alarm(30); // prenota un segnale SIGALRM, che sarà inviato tra 30 secondi  
5  
6 alarm(0); // cancella la prenotazione precedente
```

Restituisce 0 se non è presente nessuna sveglia già prenotata, altrimenti restituisce il tempo rimanente perché la vecchia sveglia suonasse.

Esiste un'unica sveglia per processo.

Può trascorrere un tempo indefinito da quando il kernel genera il segnale fino all'esecuzione dell'handler.

NB: L'azione di default di SIGALRM è la terminazione → Definire l'handler prima di eseguire l'alarm

15.12 Esempio

```
1 void foo(int num_segnale);  
2 int main(void){  
3     int n = 0;  
4     int buff[100];  
5     alarm(5);  
6     signal(SIGALRM, foo);  
7  
8     while( n<= 0){  
9         printf("Digitare qualcosa:\n");  
10        alarm(1);  
11        if((n=read(STDIN_FILENO, buf, 10)) < 0)  
12            perror("Read error");  
13        alarm(0);  
14    }  
15}
```

```

16
17 void foo(int num_segnale){
18     alarm(1);
19     printf("Vuoi muoverti a digitare qualcosa???\n");
20 }
```

15.13 Insieme di segnali

```

1 #include <signal.h>
2
3 // Ritornano 0 in caso di successo, -1 in caso di errore
4 int sigemptyset(sigset_t *set); // Tutti i segnali sono esclusi in *set
5 int sigfillset(sigset_t *set); // Tutti i segnali sono inclusi in *set
6 int sigaddset(sigset_t *set, int signum); // Aggiunge il segnale signum a *set
7 int sigdelset(sigset_t, int signum); // Rimuove signum da *set
8
9 // Ritorna 1 se vero, 0 se falso, -1 in caso di errore
10 int sigismember(const sigset_t *set, int signum); // Verifica se signum appartiene a *set
```

In alcuni casi è necessario definire un insieme di segnali:

- ▶ Per indicare al kernel quali segnali bloccare;

Non è possibile rappresentare tutti i segnali in un unico intero → Troppi segnali disponibili.

Per questo motivo POSIX definisce il tipo **sigset_t**.

15.14 Signal mask (**sigprocmask**)

```

1 #include <signal.h>
2
3 // Ritorna 0 in caso di successo, -1 in caso di errore
4 int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

La signal mask identifica un insieme di segnali bloccati dal processo.

sigprocmask consente di leggere, modificare o eseguire entrambe le operazioni sulla maschera dei segnali.

Se **oldset** non è nullo, conterrà la vecchia maschera.

Se **set** non è nullo, la nuova maschera viene calcolata in base ai parametri **set** e **how**.

how può assumere i valori:

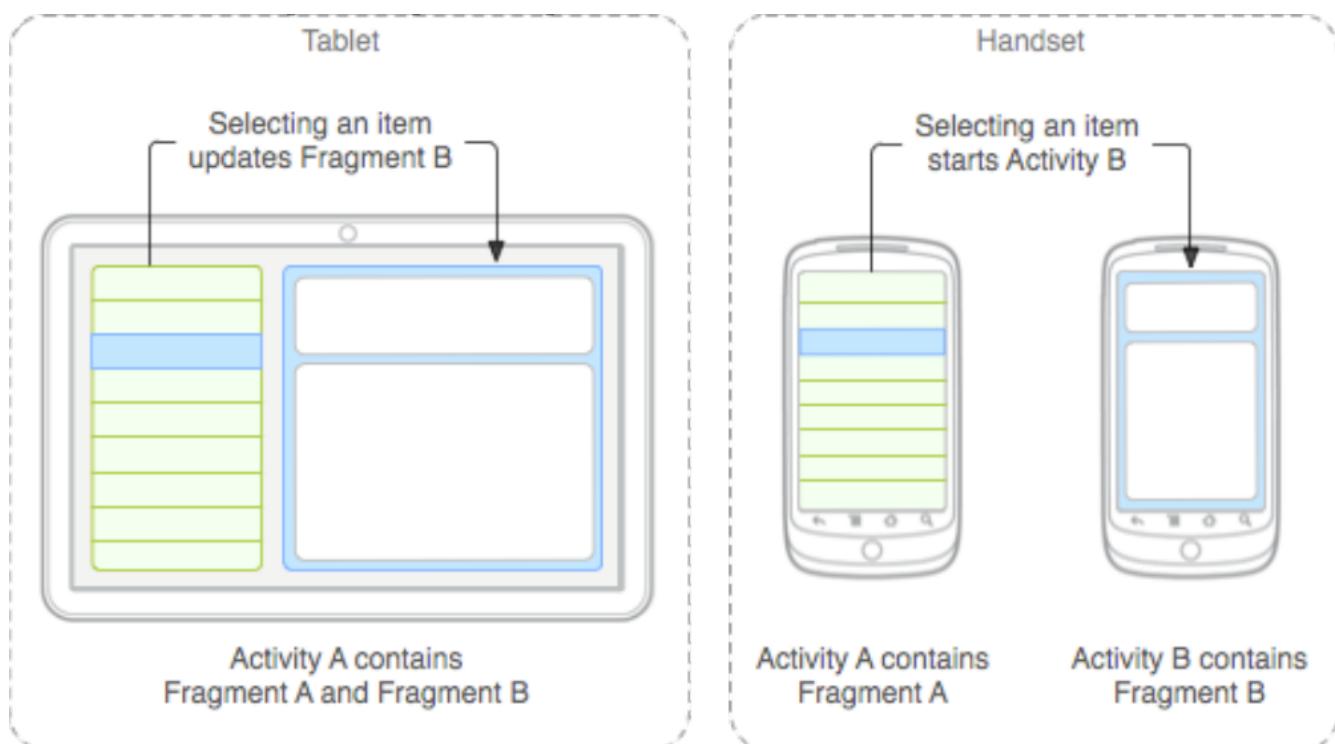
- ▶ **SIG_BLOCK** → La nuova maschera è l'unione (in senso algebrico) tra (l'insieme rappresentato dal) la vecchia maschera e la nuova;
- ▶ **SIG_UNBLOCK** → La nuova maschera è l'intersezione (in senso algebrico) tra la vecchia maschera e la nuova;
- ▶ **SIG_SETMASK** → La nuova maschera è uguale alla maschera definita dal parametro **sig**.

16 Lezione del 29-10

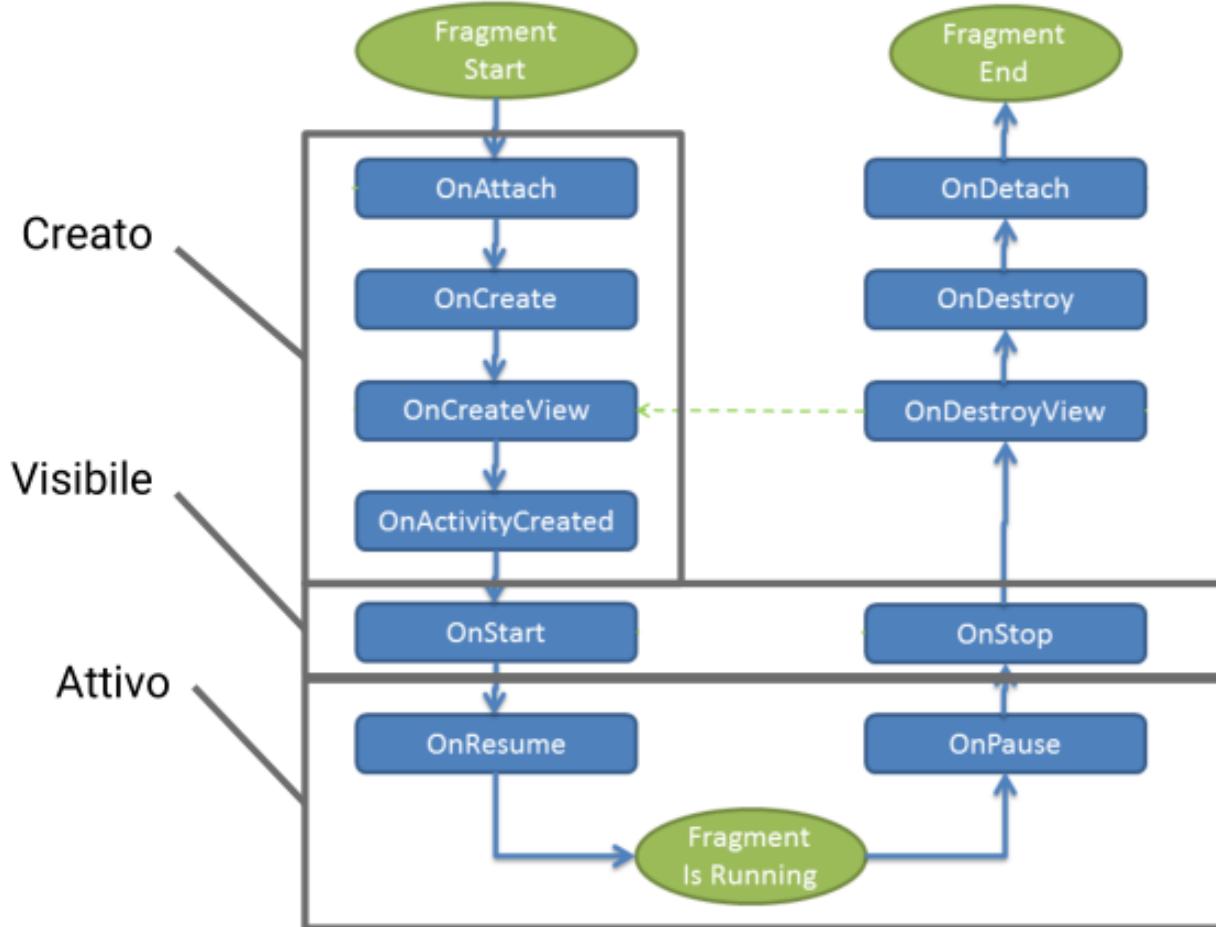
16.1 Fragment

Scompongono l'interfaccia in parti più piccole.

Hanno un loro ciclo di vita



16.2 Ciclo di vita del fragment



16.3 UI con Fragments

Contribuisce con il suo layout all'activity. Il metodo usato per caricare il layout del fragment è:

```
1 public View onCreateView( // Ritorna la root del layout del Fragment
2     LayoutInflater inflater, // Oggetto per l'inserimento del layout del fragment
3     ViewGroup container, // ViewGroup del layout dell'Activity
4     Bundle savedInstanceState // Bundle di dati nel caso in cui il fragment sia
      resumed
5 )
```

16.3.1 UI con Fragments - Inflate

Nel metodo `onCreateView` viene utilizzato il `LayoutInflater` con il metodo `inflate`.

```
1 || inflater.inflate(
```

```

2     R.layout.example_fragment, // id del layout del fragment
3     container, // ViewGroup del layout dell'Activity
4     false // Booleano per indicare se attaccare o meno il layout del fragment al view
5     group
)

```

16.4 Aggiungere Fragment all'Activity

```

1 <LinearLayout xmlns:android="http://schemas.com"
2     android:orientation="horizontal"
3     android:layout_width="match_parent"
4     android:layout_height="match_parent">
5     <!-- Il sistema alla creazione del layout sostituisce il tag fragment con onCreateView() --&gt;
6     &lt;fragment android:name="ArticleListFragment"
7         android:id="@+id/list"
8         android:layout_weight="1"
9         android:layout_width="0dp"
10        android:layout_height="match_parent"/&gt;
11    &lt;fragment android:name="ArticleReaderFragment"
12        android:id="@+id/viewer"
13        android:layout_weight="2"
14        android:layout_width="0dp"
15        android:layout_height="match_parent"/&gt;
16 &lt;/LinearLayout&gt;
</pre>

```

16.5 Fragment a RunTime

Per inserire fragment e modificarli è necessario il `FragmentManager` che richiama il `FragmentTransaction` per aggiungere, modificare, rimuovere fragment e/o eseguire altre azioni.

FrameLayout come contenitore.

```

1 <FrameLayout
2     xmlns:android="http://schemas.android.com/apk/res/android"
3     android:id="@+id/fragment_container"
4     android:layout_width="match_parent"
5     android:layout_height="match_parent"/>

```

```

1 if(findViewById(R.id.fragment_container) != null){
2     HeadlinesFragment firstFragment = new HeadlinesFragment();
3     firstFragment.setArguments(getIntent().getExtras());
4     getSupportFragmentManager().beginTransaction().add(R.id.fragment_container, firstFragment).commit
5 }

```

17 Lezione del 03-11

17.1 IPC - Pipe e FIFO

Interprocess Communication (IPC):

- ▶ Per cooperare, i processi hanno bisogno di comunicare;
- ▶ I segnali sono un primo modo di farlo
 - ◊ Il messaggio consiste nel *numero di segnale* ed, eventualmente, le informazioni in `siginfo_t`.
- ▶ Vogliamo ovviamente trasmettere informazioni arbitrarie.

17.2 Pipe

Forniscono un meccanismo attraverso il quale l'output di un processo diviene l'input per un altro processo.

Sono canali di comunicazione a senso unico tra 2 processi.

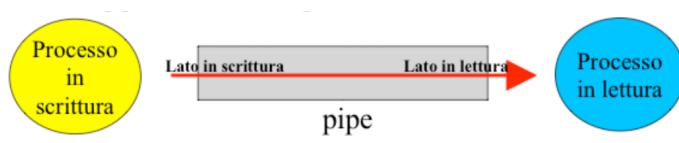
I processi devono essere imparentati → tipicamente, un padre e un figlio.

Un processo scrive sulla pipe (usando `write`). Un altro processo legge dalla stessa pipe (usando `read`)

Caratteristiche:

- ▶ La più vecchia e la più usata forma di Interprocess Communication utilizzata in UNIX;
- ▶ Limitazioni
 - ◊ Sono **half-duplex** (comunicazione in un solo senso);
 - ◊ Utilizzabili solo tra processi con un *antenato* comune. Per superarle
 - Gli *stream pipe* (**full-duplex**);
 - FIFO (*named pipe*) possono essere utilizzate tra più processi;
 - *Named stream pipe* → *stream pipe + FIFO*

Le pipe tra processi, realizzate attraverso la chiamata di sistema `pipe` o la funzione `popen` della libreria `STDIO`, sono del tutto analoghe → i dati immessi da un processo nella pipe con successive scritture sono accodati nell'attesa che un altro processo le legga; la coda è gestita con criterio **FIFO**.



Una pipe ha una dimensione finita il cui valore è definito dalla costante `PIPE_BUF`
Soltanto un numero massimo di byte può essere scritto o letto ogni volta.

Una pipe è subordinato all'organizzazione gerarchica dei processi → affinché 2 processi possano comunicare in pipeline è necessario un antenato comune che abbia predisposto una pipe a questo fine.

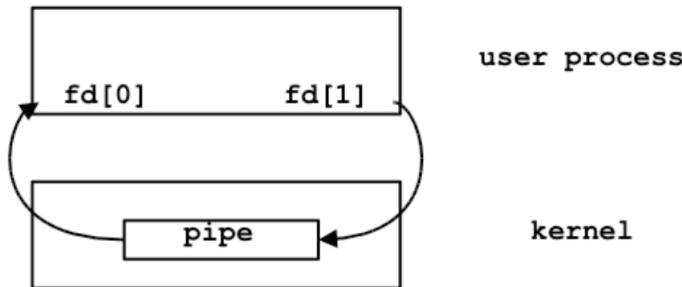
17.2.1 La funzione pipe

```
1 #include <unistd.h>
2 int pipe(int filedes[2]);
```

L'argomento `filedes` è costituito da 2 descrittori di file:

- ▶ `filedes[0]` è aperto in lettura e rappresenta il lato in lettura della pipe;
- ▶ `filedes[1]` è aperto in scrittura e rappresenta il lato in scrittura della pipe;
- ▶ L'output di `filedes[1]` è l'input per `filedes[0]`.

Restituisce 0 in caso di successo, -1 altrimenti.



L'output di `filedes[1]` (estremo di write del pipe) è l'input di `filedes[0]` (estremo di read del pipe).

```
1 int fd[2];
2
3 if(pipe(fd) < 0){
4     perror("pipe"), exit(1);
5 }
6 /*
7
8 Dopo la sua esecuzione:
9 fd[0] è il descrittore per leggere dalla pipe
10 fd[1] è il descrittore per scrivere dalla pipe
11
12 */
13
```

17.2.2 Utilizzo della pipe

Tipicamente, un processo crea una pipe e poi chiama una fork.

Chiamata `read`

Se l'estremo di write è aperto:

- ▶ Restituisce i dati disponibili, ritornando il numero di byte;
- ▶ Successive chiamate si bloccano fino a quando nuovi dati non saranno disponibili.

Se l'estremo di write è chiuso:

- ▶ Restituisce i dati disponibili, ritornando il numero di byte;
- ▶ Successive chiamate ritornano 0, per indicare la fine del file

Chiamata `write`

Se l'estremo di read è aperto:

- ▶ I dati in scrittura vengono bufferizzati fino a quando non saranno letti dall'altro processo.

Se l'estremo di read è chiuso:

- ▶ Viene generato un segnale **SIGPIPE**
 - ◊ **ignorato/catturato** → `write` ritorna -1 e `errno=EPPIPE`;
 - ◊ Azione di default → terminazione.

17.2.3 Leggere e scrivere sulle pipe

1. All'inizio una pipe è vuota;
2. `write` aggiunge dati alla pipe;
3. `read` legge e *rimuove* dati dalla pipe
 - ▶ Non si possono leggere più volte gli stessi dati da una pipe;
 - ▶ Non si può chiamare `lseek` su una pipe;
 - ▶ I dati si ottengono in ordine FIFO;
4. Una pipe con un'estremità chiusa si dice rotta (*broken*).

Scrivere → `write` aggiunge i suoi dati alla pipe:

- ▶ Se la pipe è rottata, viene generato il segnale **SIGPIPE** e write restituisce un errore.

Leggere → **read(fd[0], buf, 100)**:

- ▶ Meno di 100 bytes nella pipe → **read** legge l'intero contenuto della pipe;
- ▶ Più di 100 bytes nella pipe → **read** legge i primi 100 bytes;
- ▶ Pipe vuota → **read** si blocca in attesa di dati;
- ▶ Pipe vuota e rottata → **read** restituisce 0.

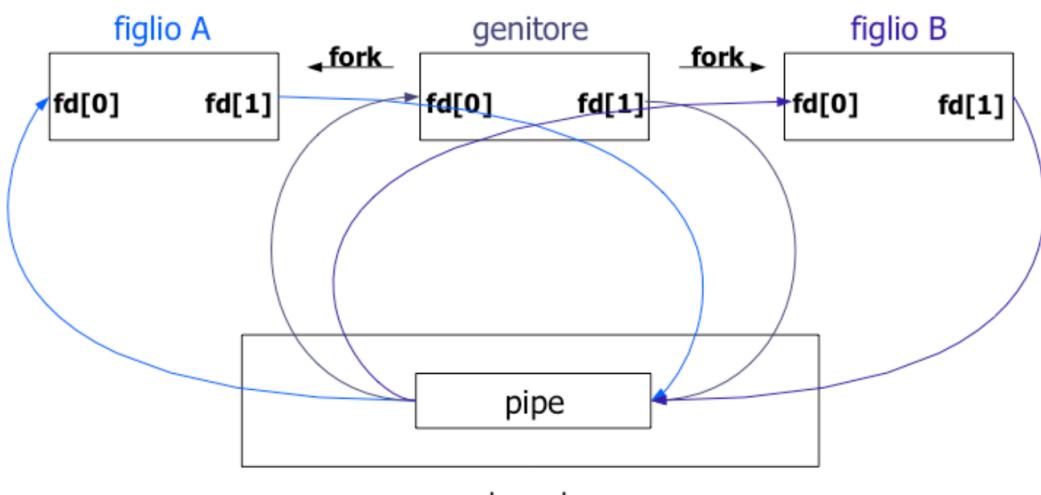
17.2.4 Pipe tra 2 programmi

Per associare lo **stdout** o lo **stdin** di un programma, rispettivamente, al lato in scrittura o a quello in lettura di una pipe, si può utilizzare la funzione seguente:

```

1 #include <unistd.h>
2 int dup2(int filedes, int filedes2);
3
4 /*
5
6 1. Duplica il descrittore di file filedes, nel nuovo descrittore filedes2;
7 2. Se filedes2 è aperto, esso viene chiuso prima della duplicazione.
8
9 Restituisce il nuovo descrittore filedes2 in caso di successo, -1 altrimenti
10 */
11

```



Pipe half-duplex dopo due chiamate a fork

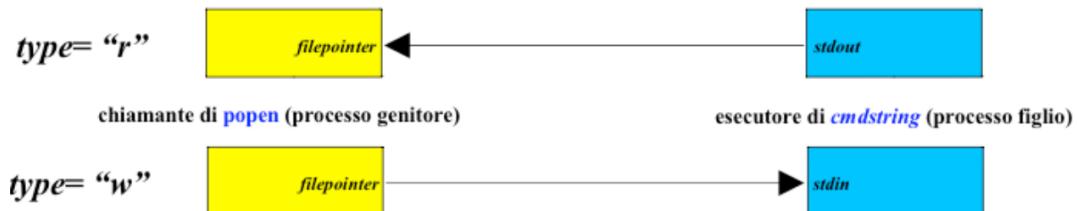
17.2.5 `popen` e `pclose`

L'esecuzione di un comando da parte di un processo in modo che quest'ultimo possa riceverne l'output o inviargli l'input è un'operazione molto comune, per la quale è dunque preferibile avere delle funzioni di più alto livello.

La libreria standard di IO fornisce le funzioni `popen` e `pclose`, che consentono al programmatore di evitare la azioni esplicite di creazione di una pipe, generazione di un figlio, chiusura del lato della pipe non utilizzato da parte dei processi scrittore e lettore, `exec` di una shell per l'esecuzione del comando e, infine, attesa della terminazione di quest'ultimo.

Più precisamente, `popen` crea una pipe, un figlio, chiude i lati non utilizzati della pipe ed esegue il comando; mentre `pclose` attende che l'esecuzione del comando sia terminata e chiude la pipe.

```
1 #include <stdio.h>
2
3 FILE *popen(const char *cmdstring, const char *type);
4
5 int pclose(FILE *filepointer)
6
7 /*
8
9 La funzione popen esegue il comando cmdstring e restituisce un puntatore a file per il processo
10 chiamante.
11 Se type è uguale a "w" il puntatore a file è connesso allo stdin del comando.
12 Se invece è uguale a "r" tale puntatore è collegato allo stdout
13
14 */
```



Il comando `cmdstring` è eseguito come `sh -c cmdstring`, ossia la shell espande i caratteri speciali.

`fp=popen("ls *.c", "r")` (eseguita da bash)

La funzione `popen` restituisce un puntatore a file in caso di successo, il puntatore `NULL` in caso di errore.

La funzione `pclose` chiude lo stream di I/O standard e attende che il comando si terminato, restituendo lo stato di terminazione della shell.

Se la shell non può essere eseguita, lo stato di terminazione restituito da `pclose` equivale all'esecuzione da parte della shell di `exit(127)`.

17.3 FIFO (named pipe)

Le pipe possono essere utilizzate solo se 2 processi hanno un antenato comune → un processo crea la pipe e qualche discendente la usa.

Le FIFO possono essere utilizzate per consentire la comunicazione tra 2 processi arbitrari → devono condividere solo il *nome* della FIFO.

Una volta creati, esistono nel file system fintanto che non vengono esplicitamente cancellati.

Possono essere usati da processi che non hanno un antenato comune.

I file FIFO possono essere creati in 2 modi:

- ▶ Attraverso la shell, con il comando `mkfifo`;
- ▶ All'interno di un programma, con la chiamata alla funzione `mkfifo`.

Un volta creato un file FIFO, su di esso si possono effettuare le operazioni usuali di I/O su file.

Dopo la creazione della FIFO, può essere usata come un file:

- ▶ Utilizzando `open`, `read`, `write`, `close`;
- ▶ Non `lseek`.

È possibile che più processi scrivano sulla stessa FIFO:

- ▶ Se il numero di byte scritti sulla FIFO è inferiore a `PIPE_BUF`, le scritture sono atomiche.

L'utilizzo di `O_NONBLOCK` consente di non bloccare le operazioni di `open/read/write` (prestare attenzione ad errori e `SIGPIPE`).

Come per le pipe, se si esegue una write su di un file FIFO che nessun processo ha aperto in lettura, è generato il segnale `SIGPIPE`.

È comune la situazione in cui più processi scrivono su di uno stesso file FIFO → affinché i dati non si mischino è necessario utilizzare operazioni atomiche di scrittura

Come per la pipe, la costante `PIPE_BUF` stabilisce il massimo numero di byte che possono essere scritti in maniera atomica in un file FIFO.

17.3.1 mkfifo

```
1 int mkfifo(char* pathname, mode_t mode);
2
3 /*
4
5     Crea un FIFO dal pathname specificato
6     La specifica dell'argomento mode è identica a quella di open, creat
7
8 */
```

Una volta creato un FIFO, le normali chiamate open, read, write, close, possono essere utilizzate per leggere in FIFO.

Il FIFO può essere rimosso utilizzando **unlink**.

Le regole per i diritti di accesso si applicano come se fosse un file normale

17.3.2 FIFO - open e write

Chiamata **write**

Se nessun processo ha aperto il file in lettura viene generato un segnale **SIGPIPE**:

- ▶ ignorato/catturato → **write** ritorna -1 e **errno=EPPIPE**;
- ▶ Azione di default → terminazione.

Atomicità:

- ▶ Quando si scrive su un pipe, la costante **PIPE_BUF** (in genere pari a 4096) specifica la dimensione del buffer del pipe;
- ▶ Chiamate **write** di dimensione inferiore a **PIPE_BUF** vengono eseguite in maniera atomica;
- ▶ Chiamate **write** di dimensione superiore a **PIPE_BUF** vengono eseguite in maniera non atomica;
- ▶ La presenza di più scrittori può causare interleaving tra chiamate.

Chiamata **open**

File aperto senza flag **O_NONBLOCK**:

- ▶ Se il FIFO è aperto in sola lettura, la chiamata si blocca fino a quando un altro processo non apre il FIFO in scrittura;
- ▶ Se il FIFO è aperto in sola scrittura, la chiamata si blocca fino a quando un altro processo non apre il FIFO in lettura;

File aperto con flag O_NONBLOCK:

- ▶ Se il FIFO è aperto in sola lettura, la chiamata ritorna immediatamente;
- ▶ Se il FIFO è aperto in sola scrittura, e nessun altro processo lo ha aperto in lettura, la chiamata ritorna un messaggio di errore.

17.3.3 Operazioni e modalità

Operazione corrente	Status del descrittore complementare	Comportamento modalità bloccante	Comportamento modalità non bloccante
apertura FIFO in sola lettura	FIFO aperta in scrittura	ritorna con successo	ritorna con successo
	FIFO chiusa in scrittura	blocca finchè la FIFO è aperta in scrittura	ritorna con successo
apertura FIFO in sola scrittura	FIFO aperta in lettura	ritorna con successo	ritorna con successo
	FIFO chiusa in lettura	blocca finchè la FIFO è aperta in lettura	ritorna con l'errore ENXIO
lettura da pipe o FIFO vuote	pipe o FIFO aperta in scrittura	blocca finchè sono immessi dati o il lato in scrittura viene chiuso	ritorna con l'errore EAGAIN
	pipe o FIFO chiusa in scrittura	ritorna col valore 0 (fine del file)	ritorna col valore 0 (fine del file)
scrittura su pipe o FIFO	pipe o FIFO aperta in lettura	Se #byte<= PIPE_BUF, scrive in modo atomico, bloccando se non c'è spazio disponibile. Se #byte>PIPE_BUF scrive in modo non atomico.	Se #byte<=PIPE_BUF, scrive in modo atomico, ritornando con EAGAIN se non c'è spazio disponibile. Se #byte>PIPE_BUF e non c'è almeno 1 byte disponibile, ritorna con EAGAIN; altrimenti scrive solo ciò che può.
	pipe o FIFO chiusa in lettura	genera SIGPIPE	genera SIGPIPE

18 Lezione del 05-11

18.1 Livelli di visibilità

Android offre diversi livelli di visibilità/accesso dei dati:

- ▶ Nascosti;
- ▶ Visibili da altre app;
- ▶ Accessibili da altre app;
- ▶ Utilizzo di Database esterni

I dati vengono salvati:

- Sul dispositivo → /data/data/<package>;
- SdCard → Android/data/<package>.

18.2 Shared Preferences

Informazioni semplici, memorizzate in una **Map**. Ogni applicazione ha le sue SharedPreferences:

```

1 getSharedPreferences();
2 getPreferences();
3 SharedPreferences.editor
4 commit();

1 public class Calc extends Activity{
2     public static final String PREFS_NAME = "MyPrefsFile";
3
4     protected void onCreate(Bundle state){
5         /*
6             Accetta 0 o MODE_PRIVATE, MODE_WORLD_READABLE, MODE_WORLD_WRITEABLE
7             NB: MODE_WORLD_READABLE e MODE_WORLD_WRITEABLE sono deprecati e non più usabili in Android
8                 7.0
9         */
10        SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
11        boolean silent = settings.getBoolean("silentMode", false);
12        setSilent(silent);
13
14    protected void onStop(){
15        SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
16        SharedPreferences.Editor editor = settings.edit();
17        editor.putBoolean("silentMode", mSilentMode)
18            editor.commit();
19    }
20}

```

18.3 Accesso a dati su file

18.3.1 Salvare dati su un file

Un'app fornisce file ad altri tramite **FileProvider**.

18.3.2 Recuperare dati da un file

```

1 String FILENAME = "hello_file";
2 byte[] b = new byte[256];
3

```

```
4 FileInputStream fis = openFileInput(FILENAME);  
5     file.read(b, 0, 256);  
6     fis.close();
```

18.3.3 Accesso a un file statico

File di progetto, salvato in `/res/raw`

```
1 openRawResource() // Restituisce un InputStream  
2  
3 // Passando R.raw.<filename>
```

18.3.4 Cache files

Alcuni dati possono essere salvati su cache. Sono gestiti diversamente dai normali file.

`File getCacheDir()` → apre la directory per i file cache

Android potrebbe eliminare la cache per salvare spazio (Android consiglia di occupare circa 1MB).

18.3.5 File su memoria esterna

Android supporta una memoria esterna (in genere esiste anche solo virtuale).

Accedere alla memoria esterna richiede:

```
1 android.permission.WRITE_EXTERNAL_STORAGE  
2 android.permission.READ_EXTERNAL_STORAGE
```

È importante ricordare che non bisogna salvare qui file utili per l'utente.

18.3.6 Cartelle pubbliche

Accedere a directory condivise con altre app

```
1 File getExternalStoragePublicDirectory(String type)  
2  
3     Environment.DIRECTORY_MUSIC  
4     Environment.DIRECTORY_PICTURES  
5     Environment.DIRECTORY_RINGTONES  
6     Environment.DIRECTORY_DOWNLOADS
```

18.3.7 Controllare lo stato

È buona norma controllare lo stato della memoria per essere sicuri che sia accessibile.

```
1 String Environment.getExternalStorageState();
```

Possibili stati:

- ▶ MEDIA_UNKNOWN;
- ▶ MEDIA_CHECKING;
- ▶ MEDIA_MOUNTED_READ_ONLY;
- ▶ MEDIA_REMOVED;
- ▶ MEDIA_NOFS;
- ▶ MEDIA_SHARED;
- ▶ MEDIA_UNMOUNTABLE;
- ▶ MEDIA_UNMOUNTED;
- ▶ MEDIA_MOUNTED;
- ▶ MEDIA_BAD_REMOVAL.

18.4 Database (android.database.sqlite)

18.4.1 Classe Contenitore (Contract)

Utile per specificare lo schema del DB. Nel livello più alto della classe si inseriscono dati globali. Saranno presenti classi interne per ogni tabella.

```
1 public final class MyDB{  
2     private MyDB();  
3     public static class ClientTab implements BaseColumns{  
4         public static final string TABLE_NAME = "Cliente";  
5         public static final string ID = "ID";  
6         public static final string NOME = "nome";  
7         public static final string COGNOME = "cognome";  
8         // ...  
9     }  
10 }
```

18.4.2 Creazione del DB

```
1 public static final String CREATE_T_CLIENTE = "CREATE TABLE" +  
2     ClienteTab.TABLE_NAME + " (" + ClienteTab.ID + " INTEGER PRIMARY KEY," +  
3     ClienteTab.NOME + " TEXT," +
```

```

4     ClienteTab.COGNOME + " TEXT ");
5
6 public static final String DELETE_T_CLIENTE =
7     "DROP TABLE IF EXISTS" + ClienteTab.TABLE_NAME;

```

Scriviamo sempre nella classe contract le varie query per la creazione e cancellazione delle tabelle

18.4.3 DB Helper (`android.database.sqlite.SQLiteOpenHelper`)

Set di API per leggere e scrivere sul DB:

- ▶ `getWritableDatabase()`;
- ▶ `getReadableDatabase()`;

Entrambe ritornano un oggetto `SQLiteDatabase`.

18.4.4 Creazione classe Helper

```

1 public class MyDBHelper extends SQLiteOpenHelper{
2     public static final int DATABASE_VERSION = 1;
3     public static final String DATABASE_NAME = "mydb.db";
4
5     public MyDBHelper(Context context){
6         super(context, DATABASE_NAME, null, DATABASE_VERSION);
7     }
8     // ...
9 }

```

Il nome del DB è il nome del file che lo contiene. La versione è utile per l'upgrade.

Si devono/possono poi sovrascrivere i metodi `onCreate()`, `onUpgrade()`, `onDowngrade()`, `onOpen()`.

18.4.5 Helper `onCreate` e `onUpgrade`

```

1 public void onCreate(SQLiteDatabase db){
2     db.execSQL(MyDB.CREATE_T_CLIENTE);
3 }
4 public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion){
5     db.execSQL(MyDB.DELETE_T_CLIENTE);
6     onCreate(db);
7 }

```

Nel metodo `onCreate` eseguiamo le istruzioni SQL presenti nella classe contract (`MyDB`).

Per accedere al DB:

```
1 Classe_Helper helper = new Classe_Helper(getApplicationContext);
```

18.4.6 Inserimento nel DB

```
1 SQLiteDatabase db = myDBHelper.getWritableDatabase(); // Richiamiamo il DB
2
3 // Utilizziamo un ContentValues per creare coppie chiave-valore
4 ContentValues values = new ContentValues();
5 values.put(MyDB.ClienteTab.NOME, nome);
6 values.put(MyDB.ClienteTab.COGNOME, cognome);
7
8 /*
9  * Inseriamo con il metodo insert:
10 * 1. Nome della tabella in cui inserire;
11 * 2. nullColumnHack;
12 * 3. Valori.
13 */
14 long newRowId = db.insert(MyDB.ClienteTab.TABLE_NAME, null, values);
```

18.4.7 Leggere dal DB

```
1 SQLiteDatabase db = myDBHelper.getReadableDatabase(); // Richiamiamo il DB
2
3 // Utilizziamo vari oggetti String per specificare la query da eseguire
4 String[] projection = {
5     MyDB.ClientTab.ID,
6     MyDB.ClientTab.NOME,
7     MyDB.ClientTab.COGNOME
8 };
9
10 String sortOrder = MyDB.ClientTab.COGNOME + " DESC";
11
12 /*
13  * Argomenti per la query:
14  * 1. Nome della tabella da leggere;
15  * 2. Colonne del risultato;
16  * 3. Colonne per la clausola where
17  * 4. Valori per la clausola where
18  * 5. Colonne per il group by;
19  * 6. Colonne per having;
20  * 7. Order by.
21 */
```

```

22
23 Cursor cursor = db.query(
24         MyDB.ClientTab.TABLE_NAME,
25         projection,
26         null,
27         null,
28         null,
29         null,
30         sortOrder);

```

18.4.8 I cursor

Con i cursor possiamo muoverci all'interno del risultato della query:

- ▶ `moveTo<posizione>;`
- ▶ `get<Tipo>`

```

1 while(curso.moveToNext()){
2     long itemId = cursor.getLong(cursor.getColumnIndexOrThrow(MyDB.ClientTab.ID));
3     String nome = cursor.getString(cursor.getColumnIndexOrThrow(MyDB.ClientTab.NOME));
4     String cognome = cursor.getString(cursor.getColumnIndexOrThrow(MyDB.ClientTab.COGNOME));
5     Log.i(TAG,"" itemId + " " + nome + " " + cognome);
6 }

```

19 Lezione del 08-11

19.1 I Thread

Sono i cosiddetti processi leggeri. Un processo può avere diversi thread.

I thread di uno stesso processo condividono la memoria ed altre risorse (per questo motivo è facile la comunicazione tra loro).

Descriveremo i PThread (*POSIX thread*).

Ad un generico processo sono associati i seguenti dati e le seguenti informazioni:

- ▶ Codice del programma in esecuzione;
- ▶ Un'area di memoria contenente le strutture dati nel programma in esecuzione;
- ▶ File aperti;
- ▶ Stack (per chiamate di procedure e funzioni);
- ▶ Contenuto dei registri della CPU.

I dati possono essere scambiati mediante messaggi (pipe, FIFO) o tenuti in memoria condivisa.

I dati possono essere tenuti in un file che viene acceduto a turno dai 2 processi.

Il context switch tra processi richiede molto lavoro al SO → oltre a cambiare il valore dei vari registri, deve spostarsi dalle aree dati e di codice del processo uscente, a quello del processo entrante.

Se dati e codice di quest'ultimo erano stati swappati in memoria secondaria, occorre prima riportarli in memoria primaria.

Per diminuire il tempo del context switch è nato il concetto dei thread.

19.2 Processo Multi-Thread

Composto da più thread, detti **peer thread**.

Ad ogni thread è associato in modo esclusivo il suo stato della computazione, fatto da:

- ▶ Valore del program counter e degli altri registri della CPU;
- ▶ Uno stack.

Ma un thread condivide con i suoi peer thread:

- ▶ Il codice in esecuzione;
- ▶ Dati;
- ▶ File aperti.

Il context switch avviene anche tra ognuno dei peer thread che formano task, in modo che tutti possano portare avanti la computazione.

Il context switch richiede il salvataggio e il ripristino della CPU e dello stack (diversi per ogni thread).

Codice, dati e file aperti sono gli stessi per tutti e non devono essere cambiati → il context switch risulta essere più veloce.

Questo fa dei thread le unità esecutive indipendenti all'interno di un processo, caratterizzate da:

- ▶ Un **thread ID**;
- ▶ Un insieme di registri (incluso un contatore di programma ed un puntatore di pila);
- ▶ Una pila per le variabili locali e gli indirizzi di ritorno (di chiamate a funzioni);
- ▶ Una maschera di segnali;
- ▶ Una priorità.

Tutti i thread relativi ad uno stesso processo ne condividono:

- Lo spazio di indirizzamento;
- I descrittori dei file aperti;
- La disposizione ed i gestori dei segnali;
- Le credenziali.

19.3 Identificare i Thread

Il processo ha un `pid` di tipo `pid_t`, mentre il thread ha `tid` di tipo `pthread_t` (struttura, funzione per il controllo `int pthread_equal(pthread_t t1, pthread_t t2)` che restituisce 0 se diversi, altrimenti altro valore).

```
pthread_t pthread_self(void)
```

Restituisce il tid del thread corrente.

19.4 Creazione dei thread

Analogamente ai processi, quando un thread viene creato è associato ad un pezzo di codice.

Tuttavia solo se il processo ospite (e quindi il suo corrispondente programma) è **single-thread**, il thread è associato all'intero programma.

Se invece è **multi-thread**, ciascun thread di tale processo è associato ad una funzione da eseguire, detta **funzione di avvio**.

I Thread possono assumere gli stati di:

- Running;
- Sleeping;
- Blocked;
- Terminated.

Quando un thread termina la situazione è analoga al caso dei processi → è il programma che deve preoccuparsi che le risorse occupate dal thread siano rilasciate al sistema, facendo in modo che venga effettuata una wait a livello di thread.

Quando un programma viene mandato in esecuzione tramite una chiamata `exec`, viene creato un singolo thread, detto **thread principale o iniziale**;

Ulteriori thread vanno creati esplicitamente.

Ogni thread ha numerosi attributi, da assegnare alla creazione:

- ◊ Livello di priorità;
- ◊ Dimensione iniziale della pila;

◊ ...;

All'atto della creazione di un thread è necessario specificare la funzione di avvio → il thread inizierà la propria esecuzione richiamando la funzione di avvio.

La terminazione di un thread può avvenire in 3 circostanze:

1. Una chiamata esplicita di terminazione del thread;
2. Una funzione di avvio ritorna;
3. Il processo contenente il thread termina.

19.5 Funzione `pthread_create`

Per creare thread addizionali relativi ad uno stesso processo POSIX prevede la funzione:

```
1 #include <pthread.h>
2 int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void *(*start_func) (void *), void
    arg);
```

Se la chiamata ha successo, tid punta al thread ID;

- ▶ `*attr` permette di specificare gli attributi del thread (se è uguale a `NULL`, gli attributi sono quelli di default);
- ▶ `start_func` è l'indirizzo della funzione di avvio;
- ▶ `*arg` è l'indirizzo dell'argomento accettato dalla funzione di avvio;
- ▶ Restituisce 0 in caso di successo, un intero positivo (secondo le convenzioni di `<sys/errno.h>`) in caso di errore.

19.6 Creare un thread

```
1 typedef void (*thread_start)(void *);
2
3 int pthread_create(pthread_t *tid, // argomento di ritorno, conterrà il tid del nuovo thread
4                     const pthread_attr_t *attributes, // attributi del thread
5                     thread_start start, // indirizzo della funzione da cui partire
6                     void *argument); // l'argomento passato alla funzione start
```

19.7 Esempio

```
1 #include <pthread.h>
2
3 pthread_t ntid;
4 // NB: Non esiste un modo portabile per stampare thread ID (dipende dalla piattaforma)
```

```

5 void printids(const char *s){
6     pid_t pid;
7     pthread_t tid;
8     pid = getpid();
9
10    // pthread_self -> thread creato non usa ntid perchè potrebbe non essere inizializzato
11    tid = pthread_self();
12
13    printf("%s pid %u tid (0x%x)\n", s, (unsigned int)pid, (unsigned int) tid, (unsigned int)tid);
14}
15
16 void *thr_fn(void* arg){
17     printids("New thread");
18     return ((void*) 0);
19}
20
21 int main(void){
22     int err;
23     err = pthread_create(&ntid, NULL, thr_fn, NULL);
24     if (err != 0){
25         printf("Can't create thread %s\n", strerror(err));
26         exit(1);
27     }
28     printids("Main thread");
29     sleep(1); // Processo main può terminare prima
30     exit(0);
31 }

```

19.8 Trattare gli errori

Siccome i thread condividono la memoria, è meglio non usare una variabile globale (come `errno`) per i codici di errore.

Le funzioni pthread restituiscono direttamente un codice d'errore.

19.9 Risorse condivise

I thread di uno stesso processo condividono:

- ▶ Memoria;
- ▶ PID e PPID;
- ▶ File descriptor;
- ▶ Reazioni ai segnali.

I thread di uno stesso processo non condividono lo **stack**

19.10 Terminazione di un thread

Invocare una **exit** provoca la terminazione dell'intero processo.

Per terminare solo il thread corrente, si può:

- ▶ Invocare `return` dalla routine di start, il valore di ritorno è l'exit status;
- ▶ Invocare `pthread_exit`;
- ▶ Invocare `pthread_cancel` da un altro thread.

Un thread può richiedere esplicitamente la propria terminazione, lasciando traccia del proprio stato di terminazione per qui thread che lo attendono attraverso:

```
1 #include <pthread.h>
2 void pthread_exit(void *status);
3
4 /*
5
6 *status punta all'oggetto che definisce lo stato della terminazione del thread.
7 NB: Questo valore non deve essere una variabile locale del thread chiamante -> pena la sua
8 scomparsa alla terminazione del thread stesso
9 */
```

Altri thread possono raccogliere il valore di uscita usando `pthread_join`.

Controllare che i dati puntati da ret sopravvivano alla terminazione dei thread:

- ▶ Lo status non deve puntare allo stack;
- ▶ È ok fare uso di variabili globali o allocate dinamicamente

19.11 Aspettare la terminazione di un thread

Un thread può attendere per la terminazione di un altro thread relativo allo stesso processo.

```
1 #include <pthread.h>
2 int pthread_join(pthread_t *tid, void **status);
3 /*
4 tid è l'ID del thread del quale si vuole attendere la terminazione
5 status punta al valore restituito dal thread per cui si è atteso, indicante il suo stato di
   terminazione
```

```

6   (Se è NULL, tale stato non viene restituito)
7   Restituisce 0 in caso di successo, un intero positivo (secondo le convenzioni di <sys/errno.h>) in
8   caso di errore.
9 */

```

19.12 Esempio - create, join

```

1 #include <pthread.h>
2 #include <stdio.h>
3 #include <errno.h>
4
5 void *start_func(void *arg) { // funzione di avvio
6     printf("%s", (char *)arg);
7     printf(" and my TID is: %d\n", (int) pthread_self());
8 }
9
10 int main(void){
11     int en;
12     pthread_t tid1, tid2;
13     char *msg1 = "Hello world, I am thread #1";
14     char *msg2 = "Hello world, I am thread #2";
15     printf("The launching process has PID: %d\n", (int)getpid());
16     printf("The main thread has TID: %d\n", (int)pthread_self());
17     // NB: In questo modo si va incontro ad una raise condition (è visto a livello globale)
18
19     // Creo i 2 thread
20     if ((en = pthread_create(&tid1, NULL, start_func, msg1) != 0))
21         errno=en, perror("pthread_create"), exit(1);
22
23     if ((en = pthread_create(&tid2, NULL, start_func, msg2) != 0))
24         errno=en, perror("pthread_create"), exit(2);
25
26     // Attendo i 2 thread
27     if ((en = pthread_join(tid1, NULL) != 0))
28         errno=en, perror("pthread_join"), exit(1);
29
30     if ((en = pthread_join(tid2, NULL) != 0))
31         errno=en, perror("pthread_join"), exit(2);
32 }

```

19.13 Condivisione memoria

I 2 thread condividono lo stesso spazio di indirizzamento, e quindi vedono le stesse variabili → se uno dei 2 modifica una variabile, la modifica è vista anche dall'altro thread.

Nel caso dei processi tradizionali, una cosa simile è ottenibile solo usando esplicitamente un segmento di memoria condivisa.

19.14 Variabili globali

I thread di un task possono condividere variabili usando quelle globali:

```
1 #include <pthread.h>
2 #include <stdio.h>
3 int global_var = 5;
4
5 void *tbody(void *arg){
6     printf("Ciao sono un thread, ora modifico una var globale\n");
7     global_var = 27;
8     *(int *) arg = 10;
9     pthread_exit((int *)50); // oppure return ((int *) 50);
10 }
```

19.15 Esempio - variabili globali, locali, allocazione dinamica

```
1 typedef struct foo{
2     int a;
3     int b;
4 } myfoo;
5
6 myfoo test; // VARIABILE GLOBALE
7
8 void stampa(char *st, struct foo *test){
9     printf("%s: tid=%d a=%d b=%d\n", st, pthread_self(), test->a, test->b);
10 }
11
12 void *fun1(void *arg){
13     myfoo test2 = {1,2}; // VARIABILE LOCALE
14     printf("%s %d\n", arg, pthread_self());
15     stampa(arg, &test2);
16     pthread_exit((void *) &test2);
17 }
18
19 void *fun2(void *arg){
20     test.a=3;
21     test.b=4; // VARIABILE GLOBALE
22     printf("%s %d\n", arg, pthread_self());
23     stampa(arg, &test);
24     pthread_exit((void *) &test);
25 }
26
27 void *fun3(void *arg){
```

```

28 myfoo *test3;
29 test3 = malloc(sizeof(struct foo)); // VARIABILE ALLOCATA DINAMICAMENTE
30 test3 -> a = 5;
31 test3 -> b = 6;
32 printf("%s %d\n", arg, pthread_self());
33 stampa(arg, test3);
34 pthread_exit((void *) test); // c
35 }
36 }
37
38 int main(void){
39 char st[100];
40 pthread_t tid1, tid2, tid3;
41 myfoo *b; // Puntatore alla struttura (non allocata)
42
43 pthread_create (&tid1, NULL, fun1, "Thread 1"); // LOCALE
44 pthread_join(tid1, (void *), &b);
45 stampa("Master ", b);
46
47 pthread_create (&tid2, NULL, fun2, "Thread 2"); // GLOBALE
48 pthread_join(tid2, (void *), &b);
49 stampa("Master ", b);
50
51 pthread_create (&tid3, NULL, fun3, "Thread 3"); // DINAMICA
52 pthread_join(tid3, (void *), &b);
53 stampa("Master ", b);
54 }
```

19.16 Cancellare un thread

```
1 int pthread_cancel(pthread_t tid);
```

Chiede che il thread specificato da `tid` venga terminato (non aspetta la sua terminazione).

Restituisce 0 in caso di successo, un codice di errore in caso contrario

19.17 La funzione `pthread_detach`

In alcuni casi è opportuno che lo stato di terminazione di un thread T non venga memorizzato fintanto che un altro processo T_1 relativo allo stesso processo attenda per T , ma sia invece cancellato subito dopo la terminazione di T :

```
1 #include <pthread.h>
2 int pthread_detach (pthread_t *tid);
```

- ▶ `tid` è l'ID del thread che si vuole distaccare;
- ▶ Restituisce 0 in caso di successo, un intero positivo (secondo le convenzioni di `<sys/errno.h>`) in caso di errore.

In questo caso non è possibile invocare `pthread_join`

19.18 Thread e fork

Se un thread chiama fork, nasce un nuovo processo con un solo thread.

Potenziali problemi con i mutex in possesso di altri thread.

19.19 Thread e segnali

Le chiamate a signal influenzano tutti i thread.

Se arriva un segnale a un processo, succede che:

- ▶ Se il processo ha impostato un handler, il segnale arriva ad *uno qualunque* dei thread (che esegue l'handler);
- ▶ Se invece la reazione al segnale consiste nel terminare il processo, *tutti i thread* vengono terminati.

19.20 Inviare un segnale a un thread

```
1 int pthread_kill(pthread_t tid, int sign);
2
3 /*
4  * Manda il segnale sign al thread specificato da tid
5  * Restituisce 0 in caso di successo, un codice di errore in caso contrario.
6 */
```

Se è impostato un handler viene eseguito nel thread `tid`.

Se non è impostato un handler, e il comportamento di default è di terminare il processo, vengono comunque terminati tutti i thread.

19.21 Attributi di un thread

Un thread può essere creato in *detached state* (stato sconnesso).

Un thread può bloccare i tentativi di essere cancellato (cancellabilità).

Altri attributi:

- ▶ Posizione e dimensione dello stack;
- ▶ Attributi *real-time*.

19.22 Gestione Attributi

```
1 #include <pthread.h>
2 int pthread_attr_init(pthread_attr_t *attr);
3 int pthread_attr_destroy(pthread_attr_t *attr);
```

Inizializza e distrugge una struttura per gli attributi di un thread. Uso:

- ▶ Si alloca una struttura `pthread_attr_t` (struttura opaca);
- ▶ Si chiama `pthread_attr_init`;
- ▶ Si modificano gli attributi contenuti nella struttura usando apposite funzioni;
- ▶ Si passa la struttura a `pthread_create`;
- ▶ Si distrugge la struttura con `pthread_attr_destroy`.

Restituiscono 0 in caso di successo, un codice di errore in caso contrario.

19.23 Detached State

Se non ci interessa il valore di ritorno di un thread, conviene crearlo in *detached state*.

19.24 Impostare detached state

```
1 int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

Imposta l'attributo `detach-state` della struttura puntata da `attr`.

L'argomento `detachstate` può essere:

- ▶ `PTHREAD_CREATE_JOINABLE` (default);
- ▶ `PTHREAD_CREATE_DETACHED`.

Restituisce 0 in caso di successo, un codice di errore in caso contrario.

19.25 Esempio

```
1 #include <pthread.h>
2
3 int makethread(void * (*fn)(void *), void arg *){
4     int err;
5     pthread_t tid;
6     pthread_attr_t attr;
7     err = pthread_attr_init (&attr);
```

```

8     iff(err != 0) return (err);
9
10    err = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
11
12    iff(err == 0)
13        err = pthread_create(&tid, &attr, fn, arg);
14
15    pthread_attr_destroy(&attr);
16    return err;
17 }
```

19.26 Cancellabilità

In ogni istante, un thread può essere cancellabile o meno.

Quando partono tutti i thread sono cancellabili. Quando un altro thread chiama `pthread_cancel`:

- ▶ Se il thread è cancellabile, viene cancellato;
- ▶ Se il thread non è cancellabile, la richiesta di cancellazione viene memorizzata, in attesa che il thread diventi cancellabile.

19.27 Impostare la cancellabilità

```
1 int pthread_attr_setcancelstate(int state, int *oldstate);
```

Imposta la cancellabilità a `state` e restituisce la vecchia cancellabilità in `oldstate`.

`state` e `oldstate` possono assumere i valori:

- ▶ `PTHREAD_CANCEL_ENABLE`
- ▶ `PTHREAD_CANCEL_DISABLE`

Restituisce 0 in caso di successo, un codice di errore in caso contrario.

20 Lezione del 10-11

Esercitazione su scripting (BASH).

21 Lezione del 12-11

21.1 Sensori HW

Android supporta 3 tipi di sensori:

- ▶ Sensori di movimento
 - ◊ Misurano movimento e rotazione sui 3 assi (sensore di gravità, giroscopio, sensore di rotazione);
- ▶ Sensori di ambiente
 - ◊ Misurano l'ambiente → pressione, temperatura, . . .;
- ▶ Sensori di posizione
 - ◊ Misurano la posizione (sensore di orientamento e magnetometro).

21.2 Sensori disponibili e simulati

Alcuni sensori sono disponibili:

- ▶ Accelerometro;
- ▶ Giroscopio;
- ▶ ...;

Altri sensori invece sono virtuali:

- ▶ Gravità;
- ▶ Accelerazione lineare.

21.3 Framework

Determinare quali sensori sono disponibili.

1. Determina le capacità di un sensore (range massimo, costruttore, batteria, richiesta, risoluzione);
2. Acquisisce RAW data dal sensore e la frequenza di acquisizione;
3. Può registrare o meno eventi riguardanti il sensore.

21.4 Utilizzare i sensori

Per utilizzare un sensore:

- ▶ Accedere al sensore;
- ▶ Specificare come gestire gli eventi;
- ▶ Registrarsi per ricevere dati

21.5 Sensor Service

```
1 String s_name = Context.SENSOR_SERVICE;
2
3 // sMan risulta utile per gestire l'accesso ai sensori -> getDefaultSensor(...)
4 SensorManager sMan = (SensorManager) getSystemService(s_name);
5
6 /*
7  * gyros:
8  * 1. Proprietà;
9  * 2. Costruttore;
10 * 3. Nome;
11 * 4. Dettagli sull'accuratezza;
12 * 5. Range.
13 */
14
15
16 List<Sensor> gyros = sMan.getSensorList(Sensor.TYPE_<type>)
```

21.6 Tipi di sensori

- ▶ TYPE_GYROSCOPE;
- ▶ TYPE_PRESSURE;
- ▶ TYPE_LIGHT;
- ▶ TYPE_GRAVITY;
- ▶ ...

21.7 Sensor

```
1 Sensor lightSens = sMan.getDefaultSensor(Sensor.TYPE_LIGHT);
2
3 float lightSen.getMaximumRange();
4 float lightSens.getPower();
```

21.8 SensorEventListener

Viene invocato per ogni evento:

- ▶ onSensorChanged(SensorEvent);
- ▶ onAccuracyChanged;

SensorEvent:

- ▶ accuracy;
- ▶ sensor;
- ▶ values;
- ▶ timestamp.

21.9 registerListener

```

1 boolean registerListener(SEL, Sensor, UpdateRate);

2 /*
3  * Dove:
4  * SEL -> SensorEventListener
5  * Sensor -> Observed Sensor
6  * UpdateRate -> SENSOR_DELAY_FASTEST, SENSOR_DELAY_GAME, SENSOR_DELAY_NORMAL, SENSOR_DELAY_UI
7
8  * La classe inizia a ricevere aggiornamenti dal sensore
9 */
10

```

21.10 SensorEvent

SensorEvent:

- ▶ Accuratezza
 - ◊ SensorManager.SENSOR_STATUS_ACCURACY_LOW
 - ◊ SensorManager.SENSOR_STATUS_ACCURACY_MEDIUM
 - ◊ SensorManager.SENSOR_STATUS_ACCURACY_HIGH
 - ◊ SensorManager.SENSOR_STATUS_UNRELIABLE
- ▶ values (float[])
 - ◊ La dimensione dipende dal sensore;
 - ◊ Il valore dipende dall'unità di misura del sensore.

21.11 Registrare un suono

```

1 mRecorder = new MediaRecorder();

2

3 mRecorder.Set AudioSource(MediaRecorder.AudioSource.MIC); // VOICE_CALL, VOICE_RECOGNITION, ...
4 mRecorder.Set OutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
5 mRecorder.setOutputFile(mFileName);
6 mRecorder.setAudioEncoderEncoder(MediaRecorder.AudioEncoder.AMR_NB);

7
8 try {

```

```
9     mRecorder.prepare();
10 } catch (IOException e){}
11
12 mRecorder.start();
13 ...
14 mRecorder.stop();
15 mRecorder.release();
16 mRecorder = null;
```

21.12 GPS

- ▶ Recupero del Location Manager (`Context.LOCATION_SERVICE`);
- ▶ Selezione del LocationProvider
 - ◊ `LocationManager.GPS_PROVIDER`;
 - ◊ `LocationManager.NETWORK_PROVIDER`;
 - ◊ `LocationManager.PASSIVE_PROVIDER`;

Per ricevere aggiornamenti dal sensore:

```
1 Location getLastKnownLocation(provider)
2
3 locMan.requestLocationUpdates(prov, time, distance, LocationListener);
```

21.13 Text to Speech (TTS)

Richiede al sistema di lanciare un'activity. Successivamente processa il risultato.

Può essere considerato come un sensore virtuale.

22 Lezione del 15-11

22.1 Sincronizzazione di thread POSIX

Proprio per il fatto che i thread condividono la memoria abbiamo un maggior rischio di **race condition** (accesso di più thread a stessi dati).

Si rende quindi necessario l'uso di meccanismi di sincronizzazione:

- ▶ **mutex** (semaforo binario);
- ▶ **condition variabile**.

22.2 Sincronizzazione

La sincronizzazione è necessaria quando si accede a variabili condivise:

- **Variabili/Strutture** dati globali (statiche e dinamiche).

NB: Tutte le operazioni possono essere non atomiche → dipende dall'architettura.

22.3 Esempio

```
1 typedef struct foo{int a; int b} myfoo;
2
3 myfoo test; // VARIABILE GLOBALE
4
5 void *inc(void *arg){
6     test.a++;
7     test.b++;
8     printf("tid=%d a=%d b=%d\n", pthread_self(), test.a, test.b);
9     pthread_exit((void*)&test);
10 }
11
12 int main(void){
13     char st[100];
14     pthread_t tid;
15     int i = 0;
16     myfoo *b;
17     while(i++ <10){
18         pthread_create(&tid, NULL, inc, NULL); // Thread concorrenti
19     }
20     sleep(1);
21 }
```

22.4 Mutex POSIX

Un mutex POSIX è caratterizzato dalle seguenti proprietà:

- È una variabile di tipo `pthread_mutex_t` che può essere inizializzata con diversi attributi (tipo, scopo, ...);
- Può assumere solo 2 stati alternativi
 - ◊ `locked`;
 - ◊ `unlocked`;
- Può essere chiuso solo da un processo alla volta, ed il processo che chiude il mutex ne diviene il possessore fino alla successiva chiusura;

- ▶ Può essere riaperto solo dal proprio possessore;
- ▶ Deve essere condiviso tra tutti i processi che intendono sincronizzare l'accesso ad una regione critica (blocco cooperativo).

22.5 I mutex

Un mutex è un semaforo binario (rosso o verde). È mantenuto in una struttura:

```
pthread_mutex_t
```

Tale struttura va allocata e inizializzata. Per farlo:

- ▶ Se la struttura è allocata staticamente → `pthread_mutex_c = PTHREAD_MUTEX_INITIALIZER`.
- ▶ Se la struttura è allocata dinamicamente (es con malloc) va chiamato `pthread_mutex_init`.

22.6 Inizializzare e distruggere un mutex

```

1 #include <pthread.h>
2 int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
3
4 int pthread_mutex_destroy(pthread_mutex_t *mutex);
5
6 /*
7
8   Inizializza e distrugge un mutex
9   Quando viene inizializzato è in stato aperto
10  Restituiscono 0 in caso di successo, un codice di errore altrimenti
11  attr può essere settato a NULL (utilizzo degli attributi di default)
12
13 */

```

22.7 Usare i mutex

```

1 int pthread_mutex_lock(pthread_mutex_t *mutex);
2 int pthread_mutex_trylock(pthread_mutex_t *mutex);
3 int pthread_mutex_unlock(pthread_mutex_t *mutex);
4
5 /*
6
7   Acquisiscono e rilasciano il semaforo
8   Restituiscono 0 in caso di successo, un codice di errore altrimenti
9
10 */

```

Se il semaforo è in stato di locked:

- ▶ Lock blocca il thread finché il semaforo si libera;
- ▶ Trylock invece non blocca, ma restituisce subito l'errore `EBUSY`.

22.8 Esempio

```
1 myfoo test; // VARIABILE GLOBALE
2
3 pthread_mutex_t sem=PTHREAD_MUTEX_INITIALIZER;
4
5 void *inc(void *arg){ // Incrementa a e b
6     pthread_mutex_lock(&sem);
7     test.a++;
8     test.b++;
9     printf("tid=%d a=%d b=%d\n", pthread_self(), test.a, test.b);
10    pthread_mutex_unlock(&sem);
11    pthread_exit((void*)&test);
12 }
13
14
15 int main(void){
16     char st[100];
17     pthread_t tid;
18     int i = 0;
19     myfoo *b;
20     while(i++ <10){
21         pthread_create(&tid, NULL, inc, NULL); // Thread concorrenti
22     }
23 }
```

22.9 Sincronizzazione

Può essere:

- ▶ Per sezione critica
 - ◊ SOLO quando una struttura condivisa viene modificata in un unico punto nel codice;
 - ◊ È sufficiente associare un mutex alla sezione critica;
- ▶ Per struttura
 - ◊ Quando la struttura può essere modificata in più punti nel codice;
 - ◊ È utile se più strutture devono essere condivise contemporaneamente;
 - ◊ È necessario associare un mutex alla struttura.

22.10 Attributi mutex

```
1 #include <pthread.h>
2 // Crea in attr un attributo di mutex come quelli richiesti dalla pthread_mutex_init()
3 int pthread_mutexattr_init(pthread_mutexattr_t *attr);
4
5 // Dealloca l'attributo di mutex in attr
6 int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
7
8 /*
9
10 Entrambe restituiscono sempre 0
11 Attualmente LinuxThreads supporta solo l'attributo relativo a tipo di mutex
12
13 */
```

22.11 Tipologie di mutex

- ▶ **fast** → semantica classica
 - ◊ Pertanto un thread che esegue 2 `mutex_lock()` consecutivi sullo stesso mutex causa uno stall;
- ▶ **recursive** → conta il numero di volte che un thread blocca il mutex e lo rilascia solo se esegue un pari numero di `mutex_unlock();`
- ▶ **error-checking** → controlla che il thread che rilascia il mutex sia lo stesso thread che lo possiede.

L'inizializzazione di un mutex può essere:

- ▶ Statica;
- ▶ Dinamica.

In maniera statica l'inizializzazione si riduce alle macro:

```
1 fastmutex = PTHREAD_MUTEX_INITIALIZER;
2 recmutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
3 errchkmutex = PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
```

In maniera dinamica, usiamo la chiamata di libreria:

```
1 int pthread_mutex_init(pthread_mutex_t *mp, const pthread_mutexattr_t *mattr);
2
3 /*
4
5 Dove:
```

```

6   - mp è un mutex precedentemente allocato;
7   - mattr sono gli attributi del mutex (NULL per il default);
8   - restituisce sempre 0.
9
10 */

```

22.12 Lock per tipologia

Blocca un mutex:

- ▶ Se era sbloccato il thread chiamante ne prende possesso bloccandolo immediatamente e la funzione ritorna subito;
- ▶ Se era bloccato da un altro thread il thread chiamante viene sospeso sino a quando il possessore non lo rilascia;
- ▶ Se era bloccato dallo stesso thread chiamante dipende dal tipo di mutex
 - ◊ fast → stallo, perché il chiamante stesso, che possiede il mutex, viene sospeso in attesa di un rilascio che non avverrà mai;
 - ◊ error checking → la chiamata fallisce;
 - ◊ recursive → la chiamata ha successo, ritorna subito, incrementa il contatore del numero di lock eseguiti dal thread chiamante.

22.13 Unlock per tipologia

Sblocca un mutex che si assume fosse bloccato. Ad ogni modo la semantica esatta dipende dal tipo di mutex:

- ▶ fast → il mutex viene rilasciato sbloccato e la chiamata ha sempre successo;
- ▶ recursive → si decrementa il contatore del numero di lock eseguiti dal thread chiamante sul mutex, e lo si sblocca solamente se tale contatore si azzera;
- ▶ error checking → sblocca il mutex solo se al momento della chiamata era bloccato e posseduto dal thread chiamante, in tutti gli altri casi la chiamata fallisce senza alcun effetto sul mutex

22.14 Attributi mutex

Si può scegliere il tipo usando queste macro:

fast	PTHREAD_MUTEX_FAST_NP
recursive	PTHREAD_MUTEX_FAST_NP
error checking	PTHREAD_MUTEX_FAST_NP

e le funzioni per **fissare/conoscere** il tipo:

```

1 #include <pthread.h>
2 // Restituisce 0 in caso di successo, un intero diverso da 0 in caso contrario
3 int pthread_mutexattr_settype(const pthread_mutexattr_t *attr, int *kind);
4
5 // Restituisce sempre 0
6 int pthread_mutexattr_gettype(const pthread_mutexattr_t *attr, int *kind);

```

22.15 Deadlock

Condizione di attesa ciclica → per evitarlo (soluzione base) → acquisire i mutex sempre nello stesso ordine.

NB → Questo non sempre è possibile; pertanto può essere necessario utilizzare algoritmi specifici e `pthread_mutex_trylock`.

22.16 Limiti dei mutex

Se un thread attende il verificarsi di una condizione su una risorsa condivisa con altri thread. Con i soli mutex sarebbe necessario un ciclo del tipo:

```

1 while(1){
2     lock(mutex);
3     if(<condizione sulla risorsa condivisa>)
4         break;
5
6     // Attesa e verifica ciclica sulla var, finchè la condizione verificata non rompe il ciclo, quindi
7     // sblocca
8     unlock(mutex);
9     ...
10 }
11 <sezione critica>;
12 unlock(mutex);

```

I mutex come strumento di cooperazione risultano:

- ▶ Inefficienti;
- ▶ Ineleganti.

Per risolvere questi problemi elegantemente servono altri strumenti.

22.17 Variabili di condizione

Strumenti di sincronizzazione tra thread che consentono di:

- ▶ Attendere passivamente il verificarsi di una condizione su una risorsa condivisa;

- ▶ Segnalare il verificarsi di tale condizione.

La condizione interessa sempre e comunque una risorsa condivisa. Pertanto le variabili condizioni possono sempre associarsi al mutex della stessa per evitare corse critiche sul loro utilizzo:

```
1 int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Servono per attendere che una condizione si verifichi, escludendo race conditions.

Utilizzata con i mutex → modificata dopo lock mutex e visibile dopo acquisizione del mutex.

Una condition variable è mantenuta in una struttura `pthread_cond_t`. Tale struttura va allocata ed inizializzata.

Se la struttura è allocata staticamente → `pthread_cond_t c = PTHREAD_COND_INITIALIZER`.

Se la struttura è allocata dinamicamente → chiamata di libreria `pthread_cond_init`.

22.18 Inizializzare e distruggere una condition variable

```
1 int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
2 int pthread_cond_destroy(pthread_cond_t *cond);
3
4 /*
5
6     Inizializza e distrugge una condition variable, rispettivamente.
7     Per la destroy non devono esistere thread in attesa.
8     Restituiscono 0 in caso di successo, un codice di errore altrimenti.
9     attr può essere NULL (attributi di default)
10
11 */
```

22.19 Inizializzazione

```
1 #include <pthread.h>
2 extern void do_work;
3
4 int thread_flag;
5 pthread_cond_t thread_flag_cv;
6 pthread_mutex_t thread_flag_mutex;
7
8 void initialize_flag(){
9     // Inizializza mutex associato a variabile condizione
10    pthread_mutex_init(&thread_flag_mutex, NULL);
11    // Inizializza variabile condizione a flag
12    pthread_cond_init(&thread_flag_cv, NULL);
```

```
13 // Inizializza flag  
14 thread_flag = 0;  
15 }
```

22.20 Usare una condition variable

Thread che aspetta una condizione:

```
1 mutex_lock(m)  
2 while(condizione falsa)  
3     cond_wait(c,m)  
4     fa qualcosa  
5 mutex_unlock(m)
```

Thread che rende vera una condizione:

```
1 mutex_lock(m)  
2 rendi la condizione vera  
3 cond_broadcast(c)  
4 mutex_unlock(m)
```

22.21 Attendere una condition variable

```
1 int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);  
2  
3 /*  
4  Attende che cond sia segnalata come vera  
5  Restituisce 0 in caso di successo, un codice di errore altrimenti.  
6  Il mutex protegge la condizione:  
7  + Deve essere acquisito prima di chiamare cond_wait;  
8  + Durante l'attesa, cond_wait rilascia il mutex;  
9  + Finita l'attesa, cond_wait riprende il mutex;  
10 */
```

22.22 Inizializzare e distruggere una condition variable

```
1 int pthread_cond_signal(pthread_cond_t *cond);  
2 int pthread_cond_broadcast(pthread_cond_t *cond);
```

- ▶ **signal** risveglia esattamente un thread in attesa su una condition variable;
- ▶ **broadcast** risveglia tutti i thread in attesa su una condition variable;
- ▶ Restituiscono 0 in caso di successo, un codice di errore altrimenti;
- ▶ **attr** può essere NULL (attributi di default).

22.22.1 Segnalazione

```
1 #include <pthread.h>  
2 int pthread_cond_signal(pthread_cond_t *cond);
```

- ▶ Nel caso in cui ci siano più thread in attesa, ne viene scelto uno ed uno solo effettuando una scelta non deterministica;
- ▶ Se non ci sono thread in attesa non effettua nulla.

22.23 Timed Wait & Broadcast

```

1 #include <pthread.h>
2 int pthread_cond_timedwait(pthread_cond_t *cond,
3                             pthread_mutex_t *mutex,
4                             const struct timespec abstime);

```

- ▶ **cond** è una variabile condizione;
- ▶ **mutex** è un mutex associato alla variabile;
- ▶ **abstime** è la specifica di un tempo assoluto;
- ▶ **pthread_cond_timedwait()** permette di restare in attesa fino all'istante specificato restituendo il codice di errore **ETIMEDOUT** al suo scadere;
- ▶ Restituisce 0 in caso di successo, un codice di errore altrimenti.

```

1 #include <pthread.h>
2 int pthread_cond_broadcast(pthread_cond_t *cond);

```

- ▶ Causa la ripartenza di tutti i thread che sono in attesa sulla variabile condizione **cond**
 - ◊ Se non ci sono thread in attesa, non accade nulla;
- ▶ Restituiscono 0 in caso di successo, un codice di errore altrimenti.

22.24 Attributi

Analoghi agli attributi dei mutex.

Attualmente LinuxThreads non supporta alcun tipo di attributo ed il secondo parametro di **pthread_cond_init()** viene ignorato.

Fanno parte dello standard POSIX:

```

1 #include <pthread.h>
2 int pthread_condattr_init(pthread_condattr_t *attr);
3 int pthread_condattr_destroy(pthread_condattr_t *attr);

```

22.25 Dati privati di un thread

Condividono il segmento dati

Complementarietà rispetto ai processi:

- ▶ Thread
 - ◊ Semplice scambiare dati con altri thread;
 - ◊ Appositi meccanismi avere dati privati (TSD);
- ▶ Processi
 - ◊ Semplice disporre di dati privati del processo;
 - ◊ Appositi meccanismi per dialogare con altri processi (IPC).

22.26 Thread Specific Data e Chiavi

Ogni thread possiede un'area di memoria privata, la TSD area, indicizzata da chiavi.

La TSD area contiene associazioni tra le chiavi ed un valore di tipo **void***:

- ▶ Diversi thread possono usare le stesse chiavi ma i valori associati variano di thread in thread;
- ▶ Inizialmente tutte le chiavi sono associate a NULL.

Thread-specific data → Associare a una stessa chiave, dati diversi per ciascun thread.

22.27 Funzioni per TSD

```
1 int pthread_key_create(...) // crea una chiave TSD
2 int pthread_key_delete(...) // deallocazione una chiave TSD
3
4 int pthread_setspecific(...) // Per associare un certo valore ad una chiave TSD
5
6 void *pthread_getspecific(...) // Per ottenere il valore associato ad una chiave TSD
```

22.28 Creare una chiave

```
1 int pthread_key_create(pthread_key_t *key, void(*destructor)(void*));
```

1. Crea una chiave per i dati privati;
2. **key** è l'indirizzo della chiave da inizializzare;

3. `destructor` è un puntatore alla funzione distruttore che deve essere chiamata alla terminazione di un thread (`pthread_exit()`);
4. Restituisce 0 in caso di successo, un codice di errore in caso contrario.

22.29 Usare una chiave

```
1 void *pthread_getspecific(pthread_key_t *key);
```

Restituisce l'indirizzo associato alla chiave `key` nel thread chiamante.

Restituisce `NULL` se nessun indirizzo è stato associato a `key`.

23 Lezione del 17-11

23.1 I socket

Introdotti con la release 4.2 di BSD nel 1983, i socket rappresentano lo *standard de facto* per la programmazione di rete.

Il loro successo è dovuto al fatto che costituiscono una interfaccia estremamente potente e flessibile:

- ▶ Consentono la comunicazione interprocesso sia localmente (ambito di uno stesso sistema), sia tra sistemi differenti interconnessi tramite rete;
- ▶ Sebbene il loro impiego più diffuso sia con la famiglia di protocolli di Internet (TCP/IP), possono essere utilizzati con molte altre famiglie di protocolli di rete.

23.2 API per IPC

Astrazione alla base di una API per IPC.

Permettono la comunicazione tra 2 processi:

- ▶ Locale;
- ▶ Remoto.

Indipendenti dal Linguaggio e dal Protocollo:

- ▶ Spesso vengono denominati Berkeley (BSD) socket;
- ▶ Rappresentano un estremo della comunicazione.

23.3 Socket e descrittori

- ▶ Nascondono al programmatore tutti i dettagli della comunicazione;
- ▶ Referenziati tramite descrittori (interi non negativi), esattamente come i file;
- ▶ Diversi tipi ottenibili tramite la chiamata di sistema `socket()`.

23.4 Domini e Stili di Comunicazione

La logica alla base della programmazione con i socket è semplice → Per realizzare i vari tipi di comunicazione viene utilizzato sempre lo stesso insieme di API generiche, precisandone la semantica (il funzionamento) attraverso opportuni argomenti.

I socket permettono di specificare il tipo di comunicazione attraverso le nozioni di:

- ▶ Dominio → Scelta di una famiglia di protocolli;
 - ◊ Identificati univocamente da un intero non negativo, cui corrispondono una o più costanti simboliche del tipo `PF_-nomefamiglia`, definite nell'header `<sys/socket.h>`.
- ▶ Stile → Definisce il tipo di canale logico instaurato per la comunicazione, ovvero le caratteristiche della trasmissione.

23.4.1 Dominio

Tra quelli più importanti ricordiamo:

- ▶ `PF_LOCAL` → comunicazione in locale tramite FS (reale o virtuale);
- ▶ `PF_INET` → la famiglia TCP/IP con IPv4;
- ▶ `PF_INET6` → la famiglia TCP/IP con IPv6;
- ▶ `PF_IPX` → famiglia di protocolli per reti Novell;
- ▶ `PF_APPLETALK` → famiglia di protocolli per reti Appletalk;

A ciascun dominio possono corrispondere in teoria uno o più schemi di indirizzamento, ossia tipi di indirizzi, per cui i socket prevedono la nozione di famiglia di indirizzi, ciascuna individuata univocamente attraverso un numero non negativo, cui corrisponde una costante simbolica del tipo `AF_nomefamiglia`.

23.4.2 Stili di comunicazione

Le trasmissioni possono:

- ▶ Avvenire a flusso o a pacchetti;
- ▶ Essere affidabili o non;
- ▶ Richiedere o meno la negoziazione di una connessione;

► ...

Lo stile di comunicazione è individuato da costanti simboliche del tipo `SOCK_nomestile` definite anche esse nell'header `<sys/socket.h>`.

Gli stili principali sono:

- ▶ `SOCK_STREAM` → Canale di trasmissione a flusso, con connessione, sequenziale ed affidabile;
- ▶ `SOCK_DGRAM` → Trasmissione a pacchetti (datagram) di lunghezza massima prefissata, senza connessione e non affidabile;
- ▶ `SOCK_RAW` → Accesso a basso livello ai protocolli di rete ed alle varie interfacce.

Assegnato un dominio, la scelta dello stile di comunicazione corrisponde in pratica ad individuare uno specifico protocollo tra quelli appartenenti al dominio.

Non tutte le combinazioni *dominio-stile di comunicazione* sono valide.

23.5 Indirizzamento

Ogni famiglia di protocolli ha una sua forma di indirizzamento (per individuare le parti in comunicazione è necessario specificarne gli indirizzi) ed in corrispondenza a questa particolare struttura di indirizzi.

Gli indirizzi vengono specificati alle socket API tramite puntatori ad opportune strutture dati, i cui nomi incominciano con `sockaddr_` ed il cui suffisso richiama il nome del relativo dominio.

Le funzioni devono poter gestire molteplici strutture e il problema di come passare i puntatori è stato risolto con l'introduzione di una struttura di indirizzi generica (definita nell'header `<sys/socket.h>`):

```
1 struct sockaddr{  
2     sa_family_t sa_family; // Address family : AF_xxx  
3     char sa_data[14]; // Address (protocol-specific)  
4 }
```

I prototipi delle funzioni che gestiscono indirizzi fanno uso di puntatori al tipo `struct sockaddr` → è necessario effettuare una conversione al tipo di struttura di indirizzi effettivamente utilizzata nella chiamata.

Per il programmatore la struttura `sockaddr` non ha altra rilevanza che quella di imporre la conversione di tipo, ma il kernel la utilizza per recuperare il campo `sa_family` e determinare il tipo di indirizzo.l

23.5.1 Indirizzi

Siamo interessati a `sa_family AF_INET` (dominio internet).

In tal caso servono 2 byte per un numero di porta e 4 byte per indirizzo IP:

```
1 #include <netinet/in.h>
2 struct sockaddr_in{
3     short int sin_family; // Address family
4     unsigned short int sin_port; // Port number
5     struct in_addr sin_addr; // IP Internet Address
6     unsigned char sin_zero[8]; // Stessi byte di struct sockaddr
7 }
```

23.6 Comunicazione Client-Server

1. Crea un socket tramite la funzione omonima ed assegna ad esso un nome (indirizzo) affinché sia condivisibile da altri processi (funzione `bind`);
2. Attende per le connessioni dei client sul socket predisposto tramite la funzione `listen`, che associa al socket una coda di lunghezza opportuna per la gestione di connessioni simultanee da parte dei client;
3. Accetta le connessioni da parte dei client tramite la funzione `accept`, il cui effetto è quello di creare un nuovo socket per ogni nuova connessione con un client *C*. Tale socket è quello che realizza effettivamente il canale di comunicazione con *C*;
4. Se il server è
 - ▶ **iterativo**, un client è posto in attesa sulla coda generata da `listen` fintanto che il server ha terminato di servire il client precedente;
 - ▶ **concorrente**, per ogni client viene generato un processo ad hoc per offrire il servizio, in modo che più client possano essere serviti in modalità concorrente. In tal caso l'attesa sulla coda è limitata al tempo necessario alla gestione della richiesta del client da parte del server ed allo start-up del nuovo processo.

Dal lato `client` la procedura è molto più semplice:

1. Il client crea un socket tramite la funzione `socket`, ma tale chiamata non fa seguire una `bind` (il socket non deve essere indirizzabile);
2. Per stabilire una connessione con il server, richiama la funzione `connect` utilizzando il nome del socket predisposto dal server come indirizzo.

23.7 Server

Operazione	Funzione
Crea il socket	socket
Gli assegna un indirizzo	bind
Si mette in ascolto	listen
Accetta nuove connessioni	accept
Chiude il socket	close
Se il socket è locale	unlink
Cancella il file corrispondente	

23.8 Funzione socket

```
1 #include <syst/socket.h>
2 int socket(int family, int type, int protocol);
3
4 /*
5
6 Apre un socket, allocando una voce nella tabella dei file del kernel e permettendo la specifica
7     del dominio, dello stile e del protocollo di comunicazione.
8
9 + family è la famiglia a cui appartiene il protocollo (dominio);
10 + type definisce il tipo di comunicazione;
11 + protocol specifica il protocollo della famiglia.
12 */
```

Restituisce -1 in caso di insuccesso, un numero positivo (il socket descriptor) altrimenti.

23.8.1 Valori e Tipi

Valori per family	Tipo di Protocolli
(AF_) PF_UNIX	Protocolli per comunicazione locali su sistemi UNIX
(AF_) PF_INET	Protocollo per internet (v4)
(AF_) PF_INET6	Protocollo per internet (v6)
(AF_) PF_APPLETALK	Protocolli per LAN Appletalk
(AF_) PF_X25	Protocolli dello standard ITU X.25 per reti a commutazione di pacchetti

Valori per type	Tipo di trasmissione
SOCK_STREAM	a flusso sequenziale ed affidabile
SOCK_DGRAM	A pacchetti di lunghezza massima fissata, non sequenziale e non affidabile
SOCK_RAW	Accesso a basso livello (costruzione manuale dei pacchetti)
SOCK_SEQPACKET	A pacchetti di lunghezza massima fissata sequenziale ed affidabile.

Valori per protocollo	Protocollo della famiglia INET
IPPROTO_TCP	TCP
IPPROTO_UDP	UDP
IPPROTO_RAW	IP
IPPROTO_ICMP	ICMP

23.8.2 Creare un socket locale

```

1 int fd = socket(PF_LOCAL, SOCK_STREAM, 0);
2 if (fd<0)
3     perror("socket"), exit(1);
4
5 int fd = socket(PF_INET, SOCK_STREAM, 0);
6 if (fd<0)
7     perror("socket"), exit(1);
8
9 int fd = socket(PF_INET, SOCK_DGRAM, 0);
10 if (fd<0)
11     perror("socket"), exit(1);

```

23.8.3 Funzione bind

```

1 #include <sys/socket.h>
2 int bind(int sd, const struct sockaddr *my_addr, int addrlen);
3
4 /*
5
6 Assegna un indirizzo locale al socket individuato dal socket descriptor sd.
7 + sd è un socket descriptor ottenuto da una precedente chiamata a socket;
8 + myaddr è l'indirizzo locale, specificando secondo il formato caratteristico della famiglia di
9     protocolli cui sd è riferito;
+ addrlen è la lunghezza del suddetto indirizzo;

```

```
10 */  
11
```

Restituisce -1 in caso di insuccesso, un numero positivo (il socket descriptor) altrimenti.

NB: Il terzo argomento deve essere pari a `sizeof` del secondo argomento.

23.9 Indirizzi locali

In `sys/un.h`:

```
#define UNIX_PATH_MAX 108  
struct sockaddr_un{  
    sa_family_t sun_family;  
    char sun_path[UNIX_PATH_MAX];  
}  
  
// Per usarlo:  
    struct sockaddr_un mio_indirizzo;  
mio_indirizzo.sun_family = AF_LOCAL;  
strcpy(mio_indirizzo.sun_path, "/tmp/mio_socket");  
bind(fd, (struct sockaddr*) &mio_indirizzo, sizeof(mio_indirizzo));
```

23.10 Funzione listen

```
1 #include <syst/socket.h>  
2 int listen(int sd, int backlog);  
3  
4 /*  
5  
6     Mette il socket in modalità di ascolto (in attesa di nuove connessioni).  
7     Il secondo argomento specifica quante connessioni possono essere in attesa di essere accettate.  
8  
9 */
```

Restituisce 0 in caso di successo, -1 altrimenti.

23.11 Funzione accept

```
1 int accept(int sockfd, struct sockaddr *indirizzo_client, socklen_t *dimensione_indirizzo);  
2  
3 /*  
4  
5     Il secondo e terzo argomento servono ad identificare il client (Possono essere NULL).  
6
```

```

7 Restituisce un nuovo descrittore in caso di successo, -1 altrimenti
8
9 Crea un nuovo socket, dedicato a questa nuova connessione.
10 Il vecchio socket resta in ascolto
11
12 */

```

23.12 Struttura di un server

```

1 int fd1, fd2;
2 struct sockaddr_un mio_indirizzo;
3
4 mio_indirizzo.sun_family = AF_LOCAL;
5 strcpy(mio_indirizzo.sun_path, "/tmp/mio_socket");
6
7 fd1 = socket(PF_LOCAL, SOCK_STREAM, 0);
8 bind(fd1, (struct sockaddr*) &mio_indirizzo, sizeof(mio_indirizzo));
9
10 listen(fd1, 5);
11 fd2 = accept(fd1, NULL, NULL);
12
13 ...
14
15 close(fd2);
16 close(fd1);
17
18 unlink("/tmp/mio_socket");

```

23.13 Connettersi ad un server

```

1 int connect(int sockfd, const struct sockaddr* serv_addr, socklen_t addrlen);
2
3 /*
4
5 Connnette il socket sockfd all'indirizzo serv_addr.
6 Il client deve conoscere l'indirizzo del server.
7 Il terzo argomento deve essere pari a sizeof del secondo argomento.
8 Restituisce 0 oppure -1.
9
10 */

```

23.14 Funzione close

```

1 #include <unistd.h>

```

2 `int close(int sock_fd);`

- ▶ Restituisce 0 in caso di successo, -1 in caso di errore;
- ▶ Serve per dichiarare che non si vuole più usare il socket;
- ▶ Più processi dello stesso host possono condividere la stessa socket
 - ◊ Solo se tutti avranno eseguito una `close()` il SO provvederà a chiudere la connessione (necessariamente di tipo `SOCK_STREAM`) concludendo il protocollo TCP;
- ▶ La chiusura è simmetrica → la connessione sarà effettivamente chiusa quando sarà stata chiusa sia sul server che sul client.

23.15 Struttura di un client

```
1 int fd;
2
3 struct sockaddr_un indirizzo;
4
5 indirizzo.sun_family = AF_LOCAL;
6 strcpy(indirizzo.sun_path, "/tmp/mio_socket");
7
8 fd = socket(PF_LOCAL, SOCK_STREAM, 0);
9 connect(fd, (struct sockaddr*) &indirizzo, sizeof(indirizzo));
10 ...
11
12 close(fd);
```

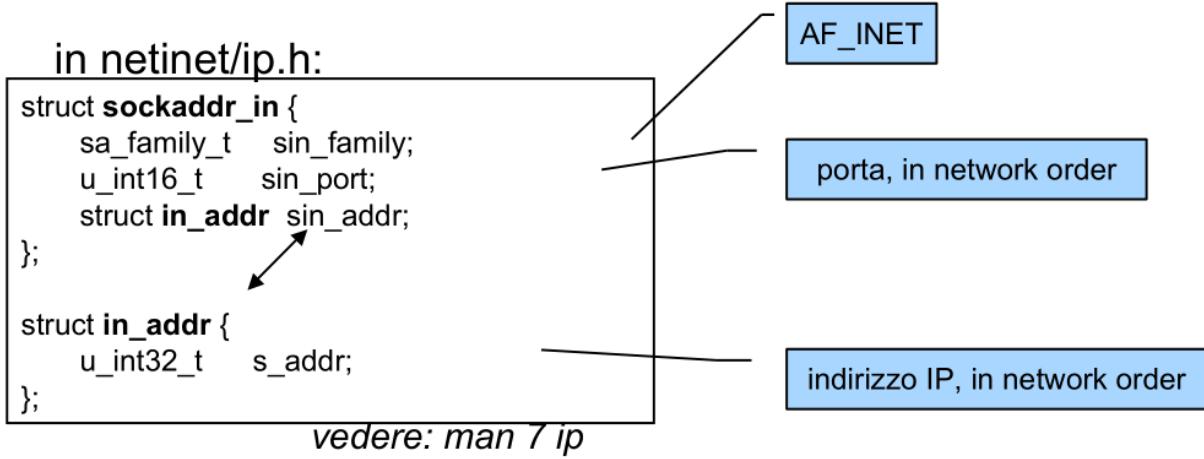
23.16 Leggere da una socket

- ▶ Si può usare `read`;
- ▶ Se non ci sono dati da leggere, `read` **blocca** il processo in attesa di dati (come per una pipe);
- ▶ È normale ottenere meno bytes di quelli richiesti;
- ▶ Ottenere 0 bytes significa che il socket è vuoto ed inoltre è stato chiuso.

23.17 Scrivere su una socket

- ▶ Si può usare `write`;
- ▶ È normale riuscire a scrivere meno bytes di quelli richiesti;
- ▶ Se il socket è stato chiuso, il processo riceve il segnale `SIGPIPE`
 - ◊ Di default, questo segnale termina il processo;
 - ◊ Se si ignora questo segnale (`signal(SIGPIPE, SIG_IGN)`), `write` restituisce -1 e imposta `errno` a `EPIPE`;
 - ◊ Si può catturare il segnale.

23.18 Indirizzi TCP/IP



NB: Prestare attenzione al numero di porta scelto → Potrebbe essere una porta riservata!

Identificati da:

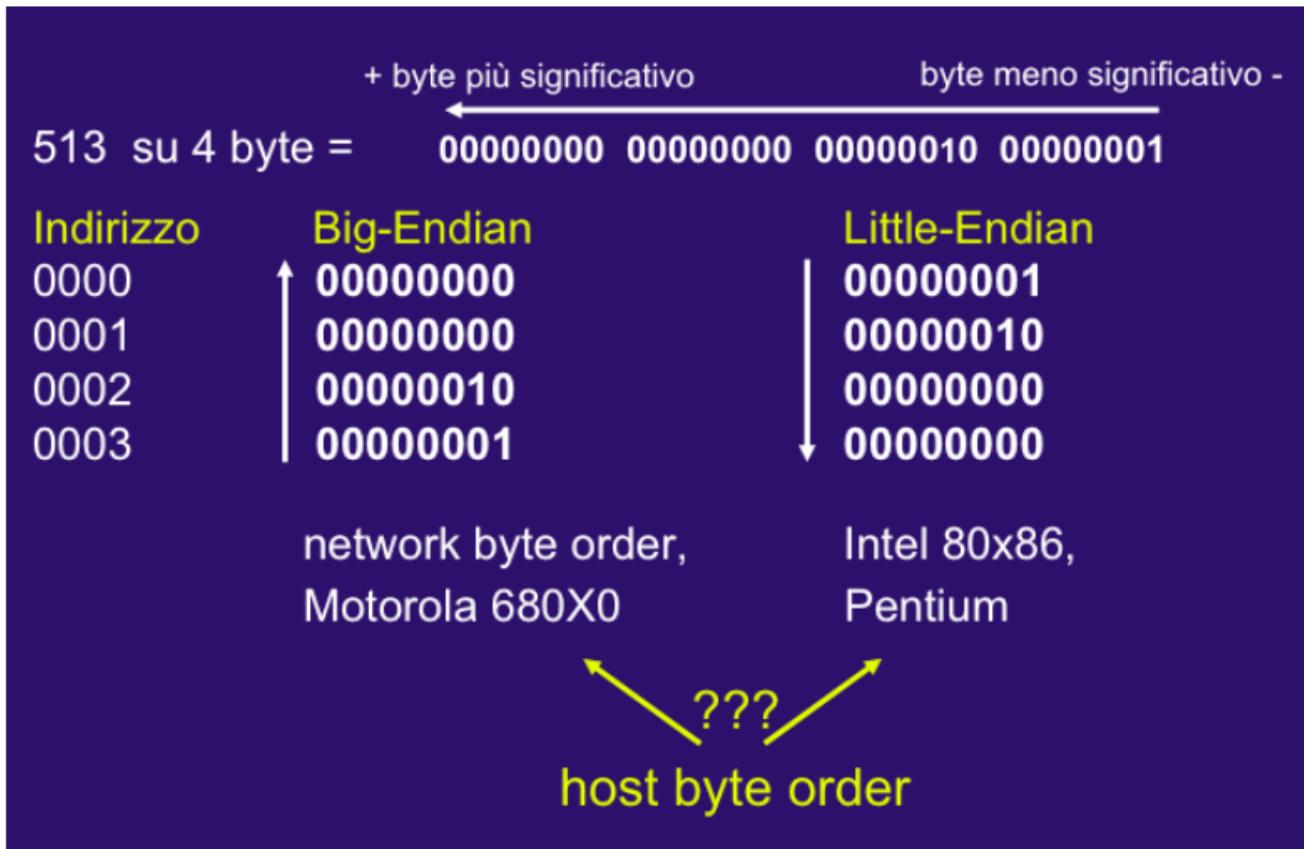
- ▶ Indirizzo IP (intero a 32 bit);
- ▶ Una porta (intero a 16 bit);

Porte

- ▶ Per offrire diversi servizi dallo stesso indirizzo IP;
- ▶ Da 0 a 1023 → Porte riservate (processi di root);
- ▶ Da 5000 a 32768 → Porte utente.
- ▶ Altre → Porte effimere (per i client, ai quali non interessa scegliere una porta specifica);

Per visualizzare la lista ufficiale fare riferimento al sito di [IANA](#).

23.19 Ordine byte



I processori possono essere di entrambi i tipi:

- I protocolli TCP/IP lavorano con `big endian`;
- Sono necessarie funzioni di conversioni.

Conversione tra `unsigned`:

- `#include <netinet/in.h>`
- `uint32_t htonl(uint32_t x)`
- `uint16_t htons(uint16_t x)`
- `uint32_t ntohl(uint32_t x)`
- `uint16_t ntohs(uint16_t x)`

h	= Host
n	= Network (big endian)
l	= Long (4 bytes)
s	= Short (2 bytes)

23.20 Specificare indirizzi IP

Attraverso la notazione *dotted*:

```
1 struct sockaddr_in indirizzo;
2 if/inet_aton("143.255.5.3", &indirizzo.sin_addr) == 0)
3 perror("inet_aton"), exit(1);
4
5 /*
6  inet_aton (ascii to network);
7  Riempe direttamente una struttura in_addr.
8 */
```

Restituisce 0 in caso di errore!

23.21 Impostare indirizzo

```
1 struct sockaddr_in my_addr;
2
3 // Host byte order
4 my_addr.sin_family = AF_INET;
5
6 // Short, network byte order
7 my_addr.sin_port = htons(MYPORT);
8
9 // Long, network byte order
10 inet_aton("10.12.110.57", &(my_addr.sin_addr));
11
12 // A zero tutto il resto
13 memset(&(my_addr.sin_zero), '\0', 8);
```

23.22 Indirizzi TCP/IP per il server

Il server chiama bind per stabilire su quale indirizzo mettersi in ascolto.

Di solito, il server sceglie solo la porta. Come indirizzo IP, sceglie INADDR_ANY, così accetta connessioni dirette a qualunque indirizzo (uno stesso host può avere più indirizzi IP).

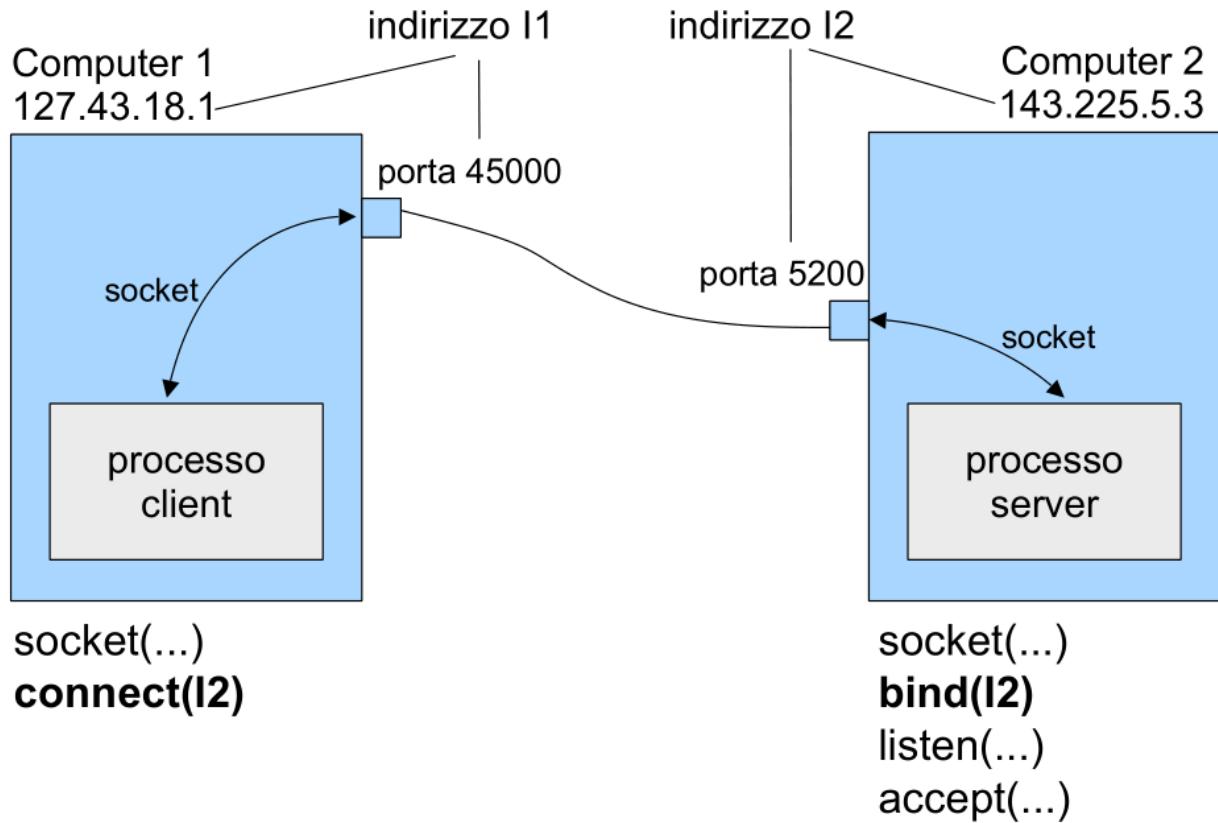
```
1 struct sockaddr_in my_addr;
2
3 my_addr.sin_family = AF_INET;
4 my_addr.sin_port = htons(5200);
5 my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
6
7 bind(fd, (struct sockaddr*) &mio_indirizzo, sizeof(mio_indirizzo));
```

23.23 Indirizzi TCP/IP per il client

Il client deve conoscere l'indirizzo IP e la porta del processo server

```
1 struct sockaddr_in indirizzo;  
2  
3 indirizzo.sin_family = AF_INET;  
4 indirizzo.sin_port = htons(5200);  
5  
6 inet_aton("10.12.110.57", &(my_addr.sin_addr));  
7  
8 connect(fd, (struct sockaddr*) &indirizzo, sizeof(indirizzo));
```

23.24 Schema della connessione



23.25 Trasmissione dati

```
1 #include <unistd.h>  
2 ssize_t write(int fd, const void *buf, size_t count);  
3
```

```

4  /*
5
6      Invia il contenuto del buffer buf al socket specificato.
7      Si usa esclusivamente con SOCK_STREAM.
8      Restituisce il numero di byte inviati oppure -1 in caso di errore
9      È la stessa funzione che consente la scrittura su file
10
11 */

```

23.26 Ricezioni dati

```

1 #include <unistd.h>
2 ssize_t read(int fd, const void *buf, size_t count);
3
4 /*
5
6     Solo per socket connessi.
7     Legge un messaggio di lunghezza massima len dal socket.
8
9     Se non c'è alcun messaggio, il programma rimane sospeso (chiamata bloccante).
10
11    La funzione ritorna il numero di byte letti, -1 in caso di errore.
12    È la stessa funzione che consente la lettura da un file
13
14 */

```

23.27 Leggere e scrivere su socket

Si usano le funzioni:

- ▶ **send** → Come write (richiede connessione stabilita), ma ha flag per specificare modalità di scrittura;
- ▶ **sendto** → Permette la scrittura su connectionless socket;
- ▶ **sendmsg** → Specificare buffer multipli;
- ▶ **recv** → Come read con flag;
- ▶ **recvfrom** → Ottenere indirizzo della fonte;
- ▶ **rcvmsg** → Ricevere da buffer multipli.

24 Lezione del 19-11

24.1 Presentazione del progetto

2 Tracce:

► POTHOLEs

- ◊ Applicazione che sia in grado di tracciare discontinuità del traccio stradale percorso;
- ◊ Utilizzo del GPS;
- ◊ Opzionale la possibilità di visualizzare gli eventi sulla mappa;
- ◊ Il server va realizzato in C su piattaforma **UNIX/Linux** e deve essere ospitato online su Microsoft Azure.
- ◊ Il client va realizzato in linguaggio Java su piattaforma Android e deve fare utilizzo dei sensori di accelerazione.

► RandomChat Room

- ◊ Il sistema deve gestire una random chat in cui i clients si collegano ad una stanza telematica e vengono messi in contatto con altri client in maniera random;
- ◊ I client possono scambiarsi messaggi testuali fino alla chiusura da parte di una delle parti delle chat;

25 Lezione del 22-11

25.1 send() e sendto()

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3
4 int send(int s, const void *msg, int len, unsigned int flags);
5 int sendto(int s, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen
6 );
7 /*
8  send() può essere utilizzata solo se s è connesso
9  sendto() sempre perchè richiede di specificare l'indirizzo di destinazione
10
11 Restituisce -1 in caso di errore oppure il numero di byte effettivamente trasmessi
12 flags si può lasciare a 0
13
14 */
```

25.2 recv() e recvfrom()

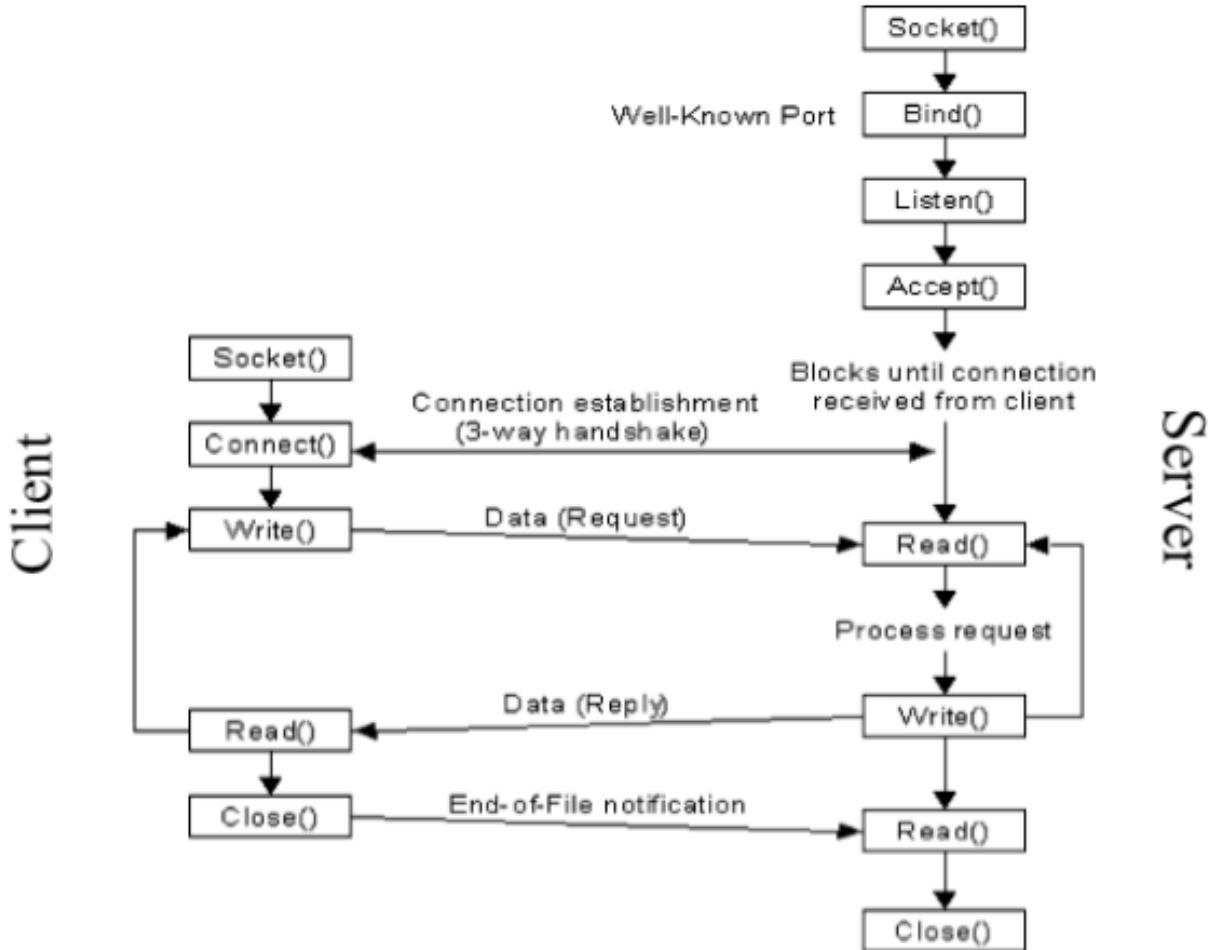
```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3
4 int recv(int s, void *buf, int len, unsigned int flags);
5 int recvfrom(int s, void *buf, int len, unsigned int flags, struct sockaddr *from, int *fromlen);
6 /*
7
8 Ricevono in buf non più di len byte. Se from non è NULL, la struttura sockaddr verrà riempita con
9 l'indirizzo del mittente
```

```

9 Restituisce -1 in caso di errore oppure il numero di byte effettivamente ricevuti
10 flags si può lasciare a 0
11
12 */

```

25.3 Comunicazione TCP/IP



25.3.1 Apertura Comunicazione TCP/IP

- Il Server si predispone ad accettare le richieste di connessione, mediante le chiamate a socket, bind, listen e infine accept che realizza una apertura passiva (passive open), cioè senza trasmissione di dati;
- Il Client effettua le chiamate a socket, bind e infine connect che realizza una apertura attiva (active open) mediante la spedizione di un segmento TCP detto SYN segment (synchronize) in cui è settato a 1 il flag **syn**, a 0 il flag **ack**, e che trasporta un numero di sequenza iniziale (**J**) che è il numero di sequenza iniziale dei dati che il client vuole mandare al server. Il segmento contiene un header TCP con i numeri di porta ed eventuali opzioni su cui accordarsi, e di solito non

- contiene dati. Il segmento viene incapsulato in un datagram IP;
3. Il Server deve rispondere al segmento SYN del client spedendogli un segmento SYN (flag syn settato a 1) con il numero di sequenza iniziale (K) dei dati che il server vuole mandare al client in quella connessione. Il segmento presenta inoltre nel campo Ack number il valore $J+1$ che indica che si aspetta di ricevere $J+1$, e presenta il flag ack settato a 1, per validare il campo Ack number;
 4. Il client, ricevendo il SYN del server con l'Ack number $J+1$ sa che la sua richiesta di connessione è stata accettata, e dal sequence number ricevuto K capisce che i dati del server inizieranno da $K+1$, quindi risponde con un segmento ACK (flag syn settato a 0 e flag ack settato a 1) con Ack number $K+1$, e termina la connect;
 5. Al ricevimento dell'ACK $K+1$ il server termina la accept.

25.3.2 Chiusura comunicazione

1. Una delle applicazioni su un end-system effettua la chiusura attiva (active close) chiamando la funzione close che spedisce un segmento FIN (flag FIN settato a 1), con un numero di sequenza M pari all'ultimo dei byte trasmessi in precedenza più
 1. Con ciò si indica che viene trasmesso un ulteriore dato di 1 byte, che è il FIN stesso;
2. L'end system che riceve il FIN effettua la chiusura passiva (passive close) all'insaputa dell'applicazione
 - ▶ Per prima cosa il modulo TCP del passivo spedisce all'end-system attivo un segmento ACK con Ack number pari a $M+1$, come riscontro per il FIN ricevuto;
 - ▶ Poi il TCP passivo trasmette all'applicazione padrona di quella connessione il segnale FIN, sotto forma di end-of-file che viene accodato ai dati non ancora letti dall'applicazione. Poiché la ricezione del FIN significa che non si riceverà nessun altro dato, con l'end-of-file il TCP comunica all'applicazione che lo stream di input è chiuso.

25.4 Server a Programmazione Concorrente

Un generico server attende le richieste di connessione su una determinata porta.

Possono arrivare richieste concorrenti e da molteplici client.

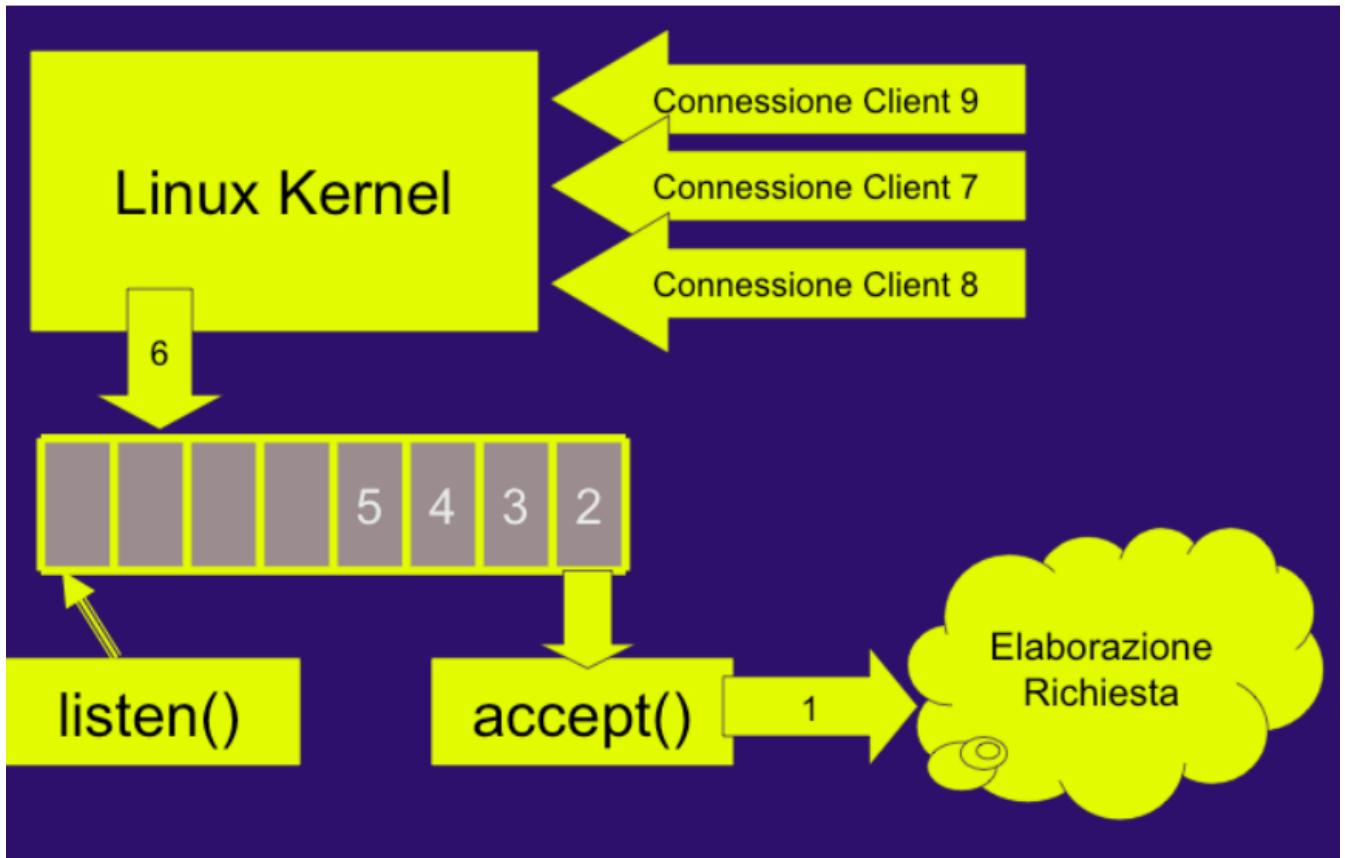
Una soluzione senza programmazione concorrente:

- ▶ Client serviti una alla volta, finché la connessione non è terminata, non vengono serviti altri client interessati al servizio;
 - ❖ **NB** → Comunque è il SO che gestisce le richieste concorrenti ed in effetti le serializza.

Nei server reali le richieste di vari client devono essere gestite concorrentemente.

Se il server è concorrente, per ogni client viene generato un processo ad hoc per offrire il servizio, in modo che più client possano essere serviti in modalità concorrente. In tal caso l'attesa sulla coda è limitata al tempo necessario alla gestione della richiesta del client da parte del server ed allo start-up del nuovo processo.

25.4.1 Coda di Connessione



25.5 Opzioni Socket

```
1 int getsockopt(int fd, level, option, void *val, socklen_t len);  
2 int setsockopt(int fd, level, option, void *val, socklen_t *len);
```

- ▶ Leggono e scrivono le opzioni di un socket;
- ▶ Le opzioni si dividono in *livelli*, identificati da appositi costanti
 - ◊ Livello SOCKET → `SOL_SOCKET`;
 - ◊ Livello TCP → `IPPROTO_TCP`;
 - ◊ Livello IP → `IPPROTO_IP`;
- ▶ Il significato di *val* dipende dall'opzione;
- ▶ *len* è pari alla dimensione dell'oggetto puntato da *val*;
- ▶ Restituiscono 0 in caso di successo, -1 altrimenti (e impostano `errno`).

25.6 Opzione SO_REUSEADDR

Opzione di livello socket.

Permette a bind di assegnare un indirizzo ancora occupato.

`val` deve puntare ad un intero:

- ▶ Se l'intero è diverso da 0, questa opzione è attiva;
- ▶ Se l'intero è uguale da 0, questa opzione è inattiva;
- ▶ `len` è pari a `sizeof(int)`.

25.7 Opzioni SO_SNDTIMEO, SO_RCVTIMEO

Opzioni di livello socket.

Impostano un timeout per le operazioni di lettura e scrittura.

Terminato il timeout, le operazioni di lettura e scrittura vengono interrotte con valore di ritorno negativo (errore) e `errno == EWOULDBLOCK`.

`val` deve puntare ad una struttura `timeval` (struct).

`len` è pari a `sizeof(timeval)`.

I timeout si annullano impostando un nuovo timeout di 0 secondi e 0 microsecondi.

25.8 Nomi di dominio

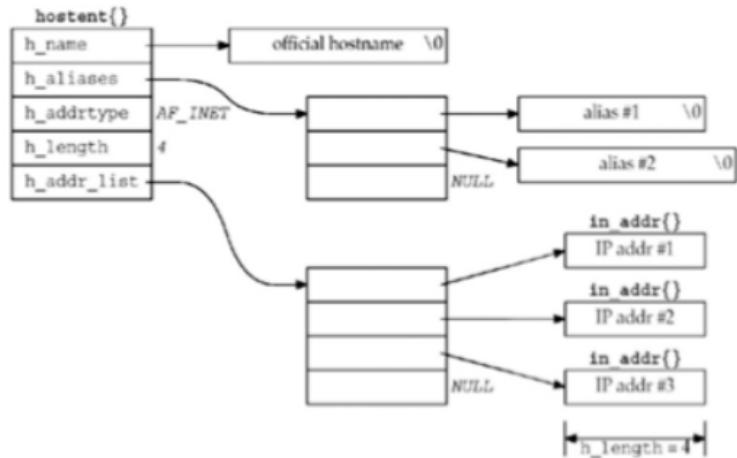
```
1 struct hostent *gethostbyname(const char *name);  
2  
3 struct hostent{  
4     char *h_name; // Nome canonico dell'host  
5     char **h_aliases; // Lista di alias  
6     int h_addrtype; // Famiglia dell'indirizzo  
7     int h_length; // Lunghezza dell'indirizzo  
8     char **h_addr_list; // Lista di indirizzi  
9 }
```

Restituisce l'indirizzo IP (oltre ad altre informazioni) corrispondente al nome di dominio dato.

L'indirizzo si trova in `h_addr_list[0]`, già in network order.

Un nome può corrispondere a più indirizzi.

- Struttura hostent
[Stevens]



```

struct sockaddr_in indirizzo;

struct hostent *p = gethostbyname("www.unina.it");
if (!p) perror("gethostbyname"), exit(1);
indirizzo.sin_addr.s_addr = *(uint32_t *)(p->h_addr_list[0]);
  
```

25.8.1 Stampare nome dominio

Partiamo da un indirizzo IP in network order, come quello ottenuto da `accept`:

```

1 char *inet_ntoa(struct in_addr in); // Non è thread-safe
  
```

Converte l'indirizzo in una stringa in formato dotted.

La stringa viene sovrascritta da ogni nuova chiamata.

25.9 Risoluzione indirizzi

- ▶ `gethostbyname()` → Dato un hostname, restituisce una struttura dati che specifica anche i suoi indirizzi IP;
- ▶ `gethostbyaddr()` → Dato un indirizzo IP, restituisce una struttura dati che specifica anche il suo hostname;
- ▶ `getserverbyname()` → Dato un nome di servizio e protocollo, restituisce una struttura dati che specifica i suoi nomi e l'indirizzo di porta;
- ▶ `gethostname()`, `getdomainname()` → Restituisce l'hostname della macchina;
- ▶ `herror()` → Stampa un messaggio di errore per `gethostname()`.

```

1 #include <netdb.h>
2 struct hostent *gethostbyname(const char *name);
3
  
```

```

4 struct hostent{
5     char *h_name; // Nome canonico dell'host
6     char **h_aliases; // Lista di alias
7     int h_addrtype; // Famiglia dell'indirizzo
8     int h_lenght; // Lunghezza dell'indirizzo
9     char **h_addr_list; // Lista di indirizzi
10 }
11
12 #define h_addr h_addr_list[0] // primo indirizzo di h_addr_list
13
14 /*
15
16     h_addr contiene l'indirizzo IP dell'host
17     h_lenght contiene la lunghezza dell'indirizzo
18
19 */

```

25.10 Lookup.c

```

1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4 extern int h_errno;
5 int main(int argc, char **argv){
6     int x, x2;
7     struct hostent *hp;
8     // Lookup dell'hostname
9     for (x=1; x<argc; ++x){
10         hp = gethostbyname(argv[x]);
11         // In caso di errore
12         if(!hp){
13             fprintf(stderr, "%s: host '%s'\n", hsterror(h_errno), argv[x]);
14
15             continue;
16         }
17         ...
18     }
19 }

```

26 Lezione del 24-11

26.1 Formato dei dati

La comunicazione deve tener conto della rappresentazione dei dati (**Big/Little** Endian).

Per i byte nel messaggio scambiato:

- ▶ Si trasmettono sequenze di caratteri;
- ▶ Un dato numerico o strutturato lo si converte in una stringa che verrà decodificata dall'host (framing & parsing);
- ▶ Si può utilizzare una rappresentazione standard.

26.2 Lettura dei dati

Potrebbe accadere che la `read()` possa restituire meno byte di quelli richieste, anche se lo stream è ancora aperto. Accade se il buffer a disposizione del socket nel kernel è stato esaurito.

È necessario quindi ripetere la `read` fino ad ottenere tutti i dati.

26.3 Scrittura dati

Prestare attenzione alla scrittura su canale chiuso (viene generato infatti un segnale `SIGPIPE` che, se non gestito, porta alla terminazione del programma).

26.4 Buffer output

Le socket TCP hanno un buffer per l'output (send buffer) in cui vengono collocati temporaneamente i dati da trasmettere. La sua dimensione può essere configurata attraverso `SO_SNDBUFFER`. Se si chiama `write()` con n byte sul socket TCP, il kernel cerca di copiare n byte sul buffer del socket. Se il buffer del socket è più piccolo o parzialmente occupato, verranno copiati un numero minore di byte.

Se il socket è bloccante alla fine della `write` saranno inviati i byte indicati dal valore di ritorno della `write`.

26.5 Scrittura completa

```
1 ssize_t writen(int fd, const char *buf, size_t n){  
2     size_t nleft, ssize_t nwritten;  
3     char *ptr;  
4     ptr = buf;  
5     nleft = n;  
6     while(nleft > 0){  
7         if ((nwritten = send(fd, ptr, nleft, MGS_NOSIGNAL)) < 0){  
8             if (errno == EINTR)  
9                 nwritten = 0; // Richiamo la write() di nuovo  
10            else  
11                return(-1); // Errore  
12        }  
13        nleft -= nwritten;  
14        ptr += nwritten;  
15    }
```

```
16     return(n);  
17 }
```

26.6 Lettura completa

```
1 ssize_t readn(int fd, char *buf, size_t n){  
2     size_t nleft, ssize_t nread;  
3     nleft = n;  
4     while(nleft > 0){  
5         if ((nread = read(fd, buf+ n - nleft, nleft))< 0){  
6             if (errno == EINTR)  
7                 return(-1); // Errore  
8             else if(nread == 0){  
9                 // EOF, connessione chiusa, termino, esce e restituisce il numero di byte letti  
10                break;  
11            }  
12            else  
13                nleft -= nread; // Continuo a leggere  
14        }  
15    }  
16    return(n - nleft); // return >= 0  
17 }
```

26.7 Comunicazione senza connessione socket UDP

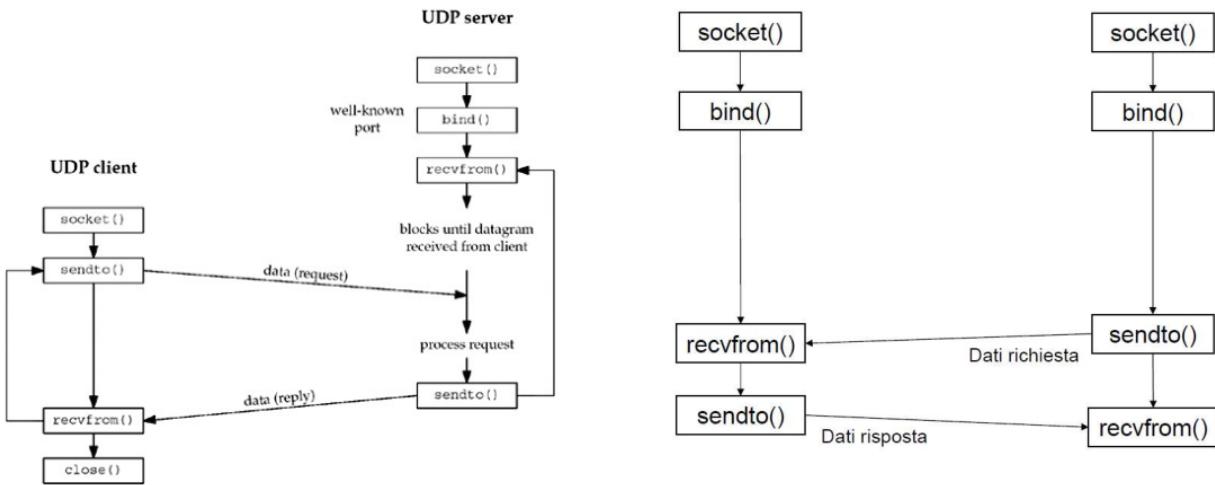
UDP è un protocollo di trasporto semplice. UDP non stabilisce connessione tra client e server e non offre garanzie di affidabilità.

UDP invia pacchetti di taglia fissa (i datagram).

L'assenza di connessione rende UDP più veloce di TCP per trasferimenti di pacchetti di byte.

Gli errori di trasmissione sono sporadici, quindi se i dati non sono critici la comunicazione può essere molto rapida ed efficace.

```
(sockfd = socket(AF_INET, SOCK_DGRAM, 0)
```



26.7.1 Funzione sendto()

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3
4 int sendto(int s, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen
);
```

Trasmissione non affidabile:

- ▶ Non riceviamo un errore se il pacchetto non raggiunge l'host remoto;
- ▶ Solo errori locali (es dimensione pacchetto troppo grande `EMSGSIZE`).

Parametri:

- ▶ `s` → Descrittore socket;
- ▶ `msg` → Puntatore al buffer del messaggio;
- ▶ `len` → Dimensione del pacchetto;
- ▶ `to` → Indirizzo del destinatario;
- ▶ `tolen` → Dimensione della struttura indirizzo.

26.7.2 Funzione recvfrom()

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
```

Restituisce byte ricevuto, -1 in caso di errore.

Parametri:

- ▶ *s* → Descrittore socket;
- ▶ *msg* → Puntatore al buffer del messaggio;
- ▶ *len* → Dimensione del buffer (se dimensione minore del pacchetto si leggono i primi *len* byte);
- ▶ *from* → Indirizzo dell'end point mittente;
- ▶ *fromlen* → Puntatore alla dimensione (byte) della struttura sockaddr;
- ▶ Con flag **MSG_PEEK** non toglie pacchetto dalla coda e verifica solo indirizzo client;
- ▶ Se non arrivano messaggi rimane bloccato, si può settare timeout.

26.7.3 Uso di connect()

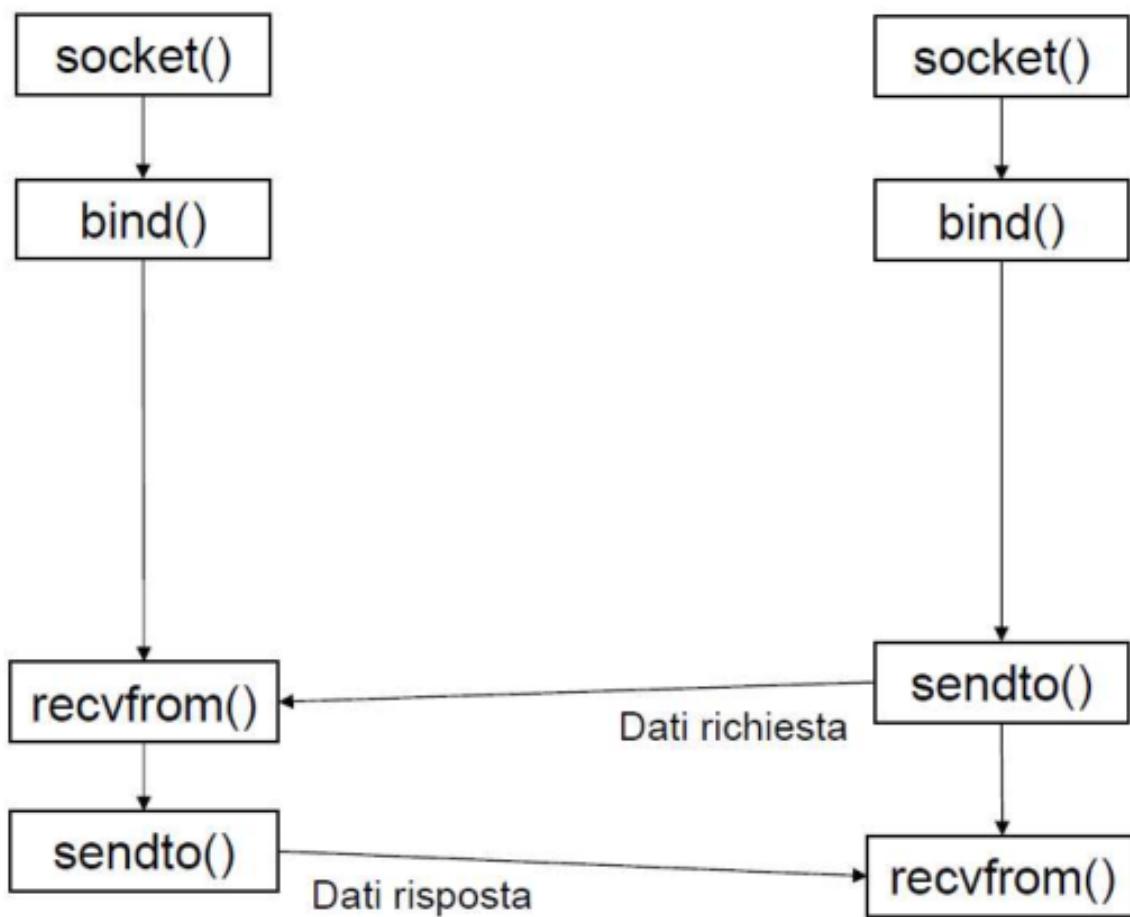
Si può usare anche per comunicazione senza connessione → Si può impostare, ad es, la connessione una volta per tutte e gestire presenza di errori sulle connessioni.

Se è stabilità la **connect()**, si può usare **write**, **send** o **sendto** lasciando **NULL** gli indirizzi.

26.7.4 Comunicazione senza connessioni

Con flusso di dati senza connessione server e client:

- ▶ Faranno **bind()** su indirizzo locale per poi comunicare con **sendto/recvfrom**.



26.7.5 Funzioni bloccanti

La funzione `accept()` e le funzioni per gestire I/O sono bloccanti (`read`, `recv`).

Il server ricorsivo tradizionale:

- ▶ Attende su `accept()` una connessione;
- ▶ Quando accetta una connessione il processo/thread principale crea un processo/thread dedicato per il controllo e si metta in attesa di altro.

Alternative:

- ▶ Usare le opzioni per le socket per impostare un timeout;
- ▶ Usare le socket non bloccanti con funzione `fcntl` (funzione di gestione `fd`) → `fcntl(fd, F_SETFL, O_NONBLOCK)`.

26.8 Select

```
1 #include <sys/time.h>
2 #include <sys/types.h>
3 #include <sys/unistd.h>
4 int select (int numfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *
    timeout);
```

Controllare contemporaneamente lo stato di uno o più descrittori degli insiemi specificati.

Si blocca finché:

- ▶ Non avviene un'attività (**lettura/scrittura**) su un descrittore degli insieme;
- ▶ Non viene generata un'eccezione;
- ▶ No scade un timeout.

Restituisce -1 in caso di errore, 0 se il timeout è scaduto, altrimenti il numero totale di descrittori.

26.8.1 Parametri Select

Insiemi di descrittori da controllare:

- ▶ **readfds** → Pronti per operazioni di lettura
 - ◊ es socket pronto per la lettura se c'è una connessione in attesa di **accept()**;
- ▶ **writefds** → Pronti per operazioni di scrittura;
- ▶ **exceptfds** → Verificare eccezioni

Tutti e 3 sono puntatori a variabili di tipo **fd_set** (bit mask implementata con un array di interi).

numfds è il numero massimo di descrittori controllati da **select()**:

- ▶ Se maxd è il massimo descrittore usato, $numfds = maxd + 1$;
- ▶ Può essere posto uguale alla costante **FD_SETSIZE**.

26.8.2 Timeout

Specifica il valore massimo che la funzione **select()** attende per individuare un descrittore pronto:

```
1 struct timeval {
2     int tv_sec; // Seconds
3     int tv_usec; // Microseconds
4 }
```

- ▶ Se impostato a `NULL` → select si blocca indefinitamente fino a quando è pronto un descrittore.
- ▶ Se impostato a 0 → select non attende (da usare per polling dei descrittori senza bloccare);
- ▶ Se diverso da 0
- ▶ select attende il tempo specificato;
- ▶ select ritorna, se è pronto, uno (o più) dei descrittori specificati (numero positivo) oppure 0 se è scaduto il timeout.

26.8.3 Impostazione degli insiemi

Macro utilizzate per gestire gli insiemi di descrittori

Funzione	Descrizione
<code>FD_SET(int fd, fd_set *set)</code>	Aggiunge fd al set
<code>FD_CLR(int fd, fd_set *set)</code>	Rimuove fd dal set
<code>FD_ISSET(int fd, fd_set *set)</code>	Ritorna vero se fd è all'interno del set
<code>FD_ZERO(fd_set *set)</code>	Pulisce tutte le entry dal set

`int FD_ISSET(int fd, fd_set *set)` → Al ritorno di `select()` controlla se fd appartiene all'insieme dei descrittori set verificando se il bit relativo a fd è pari a 1 (restituisce 0 in caso negativo, un valore diverso da 0 in caso affermativo).

La funzione select rileva i descrittori pronti:

- ▶ Significato diverso per i 3 tipi di descrittori (lettura, scrittura, eccezione).

Un descrittore è pronto in lettura:

- ▶ Nel buffer di ricezione del socket sono arrivati byte in quantità sufficiente (di default pari a 1);
- ▶ Per il lato in lettura è stata chiusa la connessione;
- ▶ Si è verificato un errore sul socket;
- ▶ Se un socket di ascolto e ci sono delle connessioni completate

Un descrittore è pronto in scrittura:

- ▶ Nel buffer di invio del socket c'è spazio sufficiente per inviare;

27 Lezione del 26-11

27.1 Classe contenitore (Contract)

Utile per specificare lo schema del DB. Nel livello più alto della classe si inseriscono dati globali. Saranno presenti classi interne per ogni tabella:

```

1 public final class MyDB{
2     private MyDB(){}
3     public static class ClientTab implement BaseColumns{
4         public static final String TABLE_NAME = "Cliente";
5         public static final String ID = "ID";
6         public static final String NOME = "nome";
7         public static final String COGNOME = "cognome";
8         // ....
9     }
10 }
```

27.2 Room

Livello di astrazione per **SQLLite**. Permette di utilizzarlo nella forma di un **ORM** (Object Relational Mapping).

È stato introdotto nel 2018.

Lavora con DAO ed Entità:

- ▶ Le entità definiscono lo schema del DB;
- ▶ Le DAO offrono metodi per l'accesso al DB.

27.2.1 Creazione di un database Room

1. Creare un classe pubblica astratta che estende **RoomDatabase**;
2. Denotarla con **@Database**;
3. Dichiarare le entità per lo schema del DB e la sua versione.

```

1 @Database(entities = {Word.class}, version = 1) // L'entità definisce lo schema DB
2 public abstract class WordRoomDatabase extends RoomDatabase{
3     public abstract WordDao wordDao(); // DAO per il DB
4     private static WordRoomDatabase INSTANCE; // Creazione del DB come istanza singleton
5     // ... creazione dell'istanza
6 }
```

27.2.2 Room - Entity

```

1 @Entity
2 public class User{
3     @PrimaryKey
4     private int uid;
5 }
```

```

6     @ColumnInfo(name = "first_name")
7     private String firstName;
8
9     @ColumnInfo(name = "last_name")
10    private String lastName;
11
12    // Geter e setter omessi per brevità
13    // Richiesti per il funzionamento di Room (obbligatori quando i campi sono private)
14
15    // Costruttori -> Possono avere un costruttore vuoto solo se il corrispondente DAO può accedere
16    // ai parametri
17    // O un costruttore i cui nomi e tipi corrispondono ai campi dell'Entity
18
19 }

```

27.2.3 Room - DAO

```

1 @Dao
2 public interface UserDAO{
3     @Query("SELECT * FROM user")
4     List<User> getAll();
5
6     @Query("SELECT * FROM user WHERE uid IN (:userIds)")
7     List<User> loadAllByIds(int[] userIds);
8
9     @Query("SELECT * FROM user WHERE first_name LIKE :first AND " + "last_name LIKE : last LIMIT 1")
10    User findByName(String first, String last);
11
12    @Insert
13    void insertAll(User... users); // Abbiamo la possibilità di inserire più utenti (Si aspetta una
14                                // serie di utenti)
15
16    @Delete
17    void delete(User user);
18
19 }

```

27.2.4 Room - Database

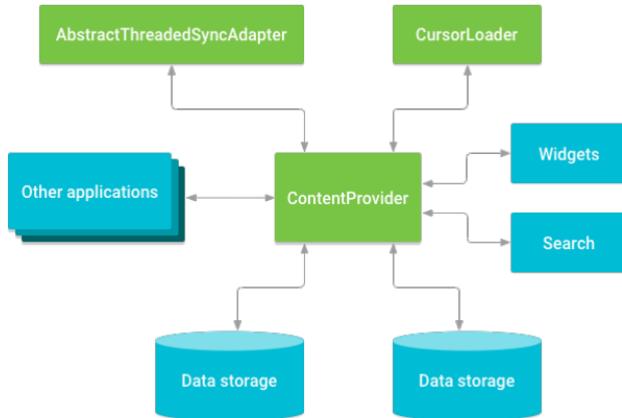
```

1 // Dichiaraione
2 @Database(entities = {User.class}, version = 1)
3 public abstract class AppDatabase extends RoomDatabase{
4     public abstract UserDao userDao();
5 }
6
7 // Utilizzo

```

```
8 AppDatabase db = Room.databaseBuilder(getApplicationContext(), AppDatabase.class, "database-name").  
    build();
```

27.3 Content Provider



Il content provider coordina l'accesso ai dati da parte delle altre applicazioni.

27.3.1 Accedere ai dati

- ▶ Activity o Fragment → Utilizza un **CursorLoader** per rendere la UI utilizzabile;
- ▶ **CursorLoader** → Gestisce la query presente nell'activity;
- ▶ **ContentResolver** → Oggetto nel contesto dell'applicazione
 - ◊ Comunica come client del provider;
 - ◊ Esegue le operazioni CRUD (create, retrieve, update, delete);
- ▶ **ContentProvider** (Collegato a Data Storage) → Riceve le richieste dal client, le esegue e ritorna il risultato.

27.3.2 Content Provider nel Manifest

Come activity e service, i content provider sono aggiunti al manifest:

```
1 <provider  
2     android:name="com.example.android.provider.StubProvider"  
3     android:authorities="com.example.android.provider"  
4     android:exported="false"  
5     android:syncable="true"/>
```

- ▶ **name** → Classe del provider;
- ▶ **authorities** → URI per il provider definisce il DB da usare (per il **ContentResolver**);
- ▶ **exported** → Dati per altre applicazioni (default per API < 17 **true** altrimenti **false**);
- ▶ **syncable** → Dati da sincronizzare.

27.3.3 Content URI

Il content provider deve sempre avere un URI associato:

```
1 public static final Uri CONTENT_URI = Uri.parse("content://com.example.android.prov/elements");
2 // Accedo a tutti gli elementi della tabella del mio provider.
3 // Accedo a singoli elementi se aggiungo /numero_riga
```

27.3.4 Creazione dei Content Provider

Estendere ContentProvider e sovrascrivere:

- ▶ `onCreate()` → Inizializzare il DB della classe Helper;
- ▶ `query()`
 - ◊ `out` → Cursor;
 - ◊ `in` → URI e argomenti della where;
- ▶ `delete()`
 - ◊ `out` → int;
 - ◊ `in` → URI e argomenti della where;
- ▶ `insert()`
 - ◊ `out` → URI;
 - ◊ `in` → URI e ContentValues;
- ▶ `update()`
 - ◊ `out` → int;
 - ◊ `in` → URI, ContentValues e argomenti where
- ▶ `getType()` → ritorna il tipo di dato del cursor tutte le colonne o singolo elementi.

27.3.5 Interrogare Il Content Provider con ContentResolver

```
1 Cursor result = getContentResolver().query(
2                     Uri.parse("content://..."), // Richiesta dati
3                     colonne,
4                     where,
5                     where_args
6                     );
7 Uri rowUri = getContentResolver().insert(
8                     Uri.parse("content://..."), // Inserimento Dati,
9                     ContentValues
10                    );
```

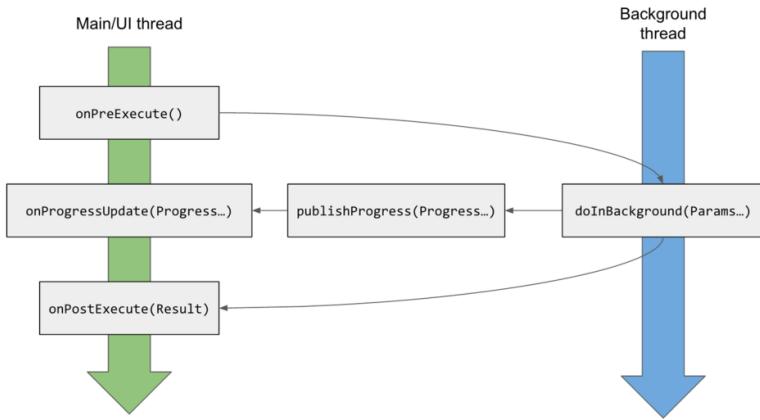
28 Lezione del 29-11

Esercitazione su scripting BASH e programmazione su Socket.

29 Lezione del 01-12

29.1 AsyncTask (deprecati)

Una classe messa a disposizione dall'SDK che offre una serie di task:



- ▶ onPreExecute() (eseguito sul main thread);
 - ◊ onProgressUpdate();
 - ◊ doInBackGround();
 - publishProgress();
- ▶ onPostExecute();

29.1.1 Esempio

```
1 class DownloadFileFromURL extends AsyncTask<Type, Type, Type>{  
2     @Override  
3         protected void onPreExecute(){  
4             super.onPreExecute();  
5         }  
6  
7     @Override  
8         protected void doInBackGround(Params ...){ ... }  
9  
10    @Override  
11        protected void onPostExecute(Params ...){ ... }  
12 }
```

29.2 Modificare la UI - Handler

La UI può essere modificata solo dal Main thread, oppure utilizziamo un modo per comunicare con il main thread (Handler e Message).

29.2.1 Handler

Contenuto in `android.os.Handler`

Aiuta nella lettura dei messaggi dai worker thread all'UI. Viene tipicamente creato nell'UI:

```
1 Handler handler = new Handler();
```

Gestisce l'esecuzione dei Runnable o Messaggi che sono associati alla coda messaggi di un thread.

Per una coda di messaggi del Thread → un Handler

29.2.2 Metodi Handler Runnable

```
1 post(Runnable r);  
2 postDelayed(Runnable r, long delayMillis);
```

L'oggetto `Runnable r` sarà aggiunto alla coda messaggi. Sarà eseguito sul thread dove è attaccato l'handler.

- ▶ `r` → `Runnable` che sarà eseguito;
- ▶ `delayMillis` → Il tempo di attesa prima dell'esecuzione.

Entrambi ritornano un booleano.

29.2.3 Metodi Handler Messaggi

```
1 boolean sendMessage(Message msg); // invio messaggio  
2 Message msg obtainMessage(); // istanzia un nuovo messaggio  
3 void handleMessage(Message msg); // da sovrascrivere per operazioni sul messaggio
```

29.2.4 Esempio

```
1 Thread genNumber = new Thread(new Runnable () {  
2     @Override  
3     public void run(){  
4         Random r = new Random();  
5         int number = r.nextInt(10);  
6         TextView tv = (TextView) findViewById(R.id.textView);  
7         tv.setText(""+ number);  
8     }  
9 };  
10 genNumber.start();
```

29.3 Service

Eseguono task in background. Risultano utili per svolgere operazioni che richiedono molto tempo prima che vengano completate. Non richiedono l'uso di interfaccia utente.

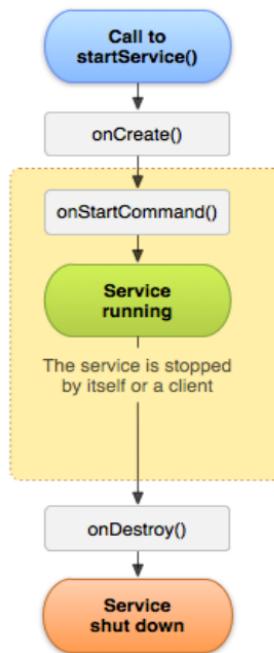
29.3.1 Tipi di servizi

- ▶ **Started** (Lunghe operazioni)
 - ◊ Eseguiti da Activity;
 - ◊ Possono essere eseguiti in background indefinitamente;
 - ◊ Quando uno started service è completo dovrebbe fermarsi tramite `stopSelf()` oppure può essere fermato con `stopService()` da un'Activity;
- ▶ **Bound** (Piccole operazioni);
- ▶ **Scheduled**.

29.3.2 Creazione di un service

```
<manifest ...>
  <application ...>
    <service android:name=".ExampleService" />
    ...
  </application>
</manifest>
```

29.3.3 Ciclo di vita



Service Started o locale:

startService invoca onStartCommand()

29.3.4 Creazione di un Service

- ▶ Service → Classe base per i service, va creato un nuovo thread per il servizio;
- ▶ IntentService → Sottoclasse di service ottimo se il servizio ha una sola richiesta.

30 Lezione del 06-12

30.1 Servizi Started

Sono eseguiti da Activity. Possono essere eseguiti in background indefinitivamente:

```
1 startService(Intent service)
```

Quando uno starter service è completo dovrebbe fermarsi tramite `stopSelf()` oppure può essere fermato con `stopService()` da un'Activity.

30.2 IntentService

- ▶ Crea un worker thread di default;
- ▶ Crea una coda per il thread per `onHandleIntent()`;
- ▶ Ferma il servizio quando ha finito le richieste;
- ▶ Implementa `onBind(null)`.

30.2.1 Esempio

```
1 public class ProvaIntentService extends IntentService{
2     // Il costruttore chiama la super IntentService con il nome del thread
3     public ProvaIntentService(){
4         super("ProvaIntentService");
5     }
6     /*
7      IntentService chiama questo metodo dal worker thread
8      Quando il metodo termina, termina anche il service
9     */
10    @Override
11    protected void onHandleIntent(Intent intent){
12        // Quello che va eseguito
13        try { Thread.sleep(5000);
14        } catch (InterruptedException e){}
15    }
16 }
```

30.2.2 Metodi

- ▶ `onCreate()` → Inizializzazione;
- ▶ `onStartCommand()` → Esecuzione
 - ◊ Intent → passato da `startService()`;
 - ◊ flag → argomenti opzionali per la richiesta;
 - ◊ id della richiesta;
 - ◊ Ritorna un intero per il sistema
 - START_NO_STICKY → In caso di kill vengono svolti solo intent pendenti;
 - START_STICKY → Ricrea e rilancia `onStartCommand()`, ma *null intent*;
 - START_REDELIVER_INTENT → Ricrea e rilancia l'ultimo intent.
- ▶ `onBind()` → Non usato per started service;
- ▶ `onDestroy()` → Fine.

30.3 Bound Service

Un service può essere:

- ▶ `started` → Il service è avviato ed è indipendente dal resto dei componenti delle applicazioni;
- ▶ `bound` → Il service è legato ai componenti delle applicazioni. È attivo finché almeno un componente è legato ad esso.

Lo stesso service può lavorare in entrambi i modi (`onStartCommand()` o `onBind()`).

30.3.1 Avviare un bound service

Si estende Service → `IBinder onBind(Intent intent)`.

Nel componente che si collega

```
boolean bindService(Intent service, ServiceConnection conn, int flags), dove:
```

- ▶ `conn` permette di tener traccia dello stato del service;
- ▶ `service` deve essere un intent esplicito;
- ▶ `flags` può assumere diversi valori quali
 - ◊ 0;
 - ◊ `Content.BIND_AUTO_CREATE`;
 - ◊ `Content.BIND_DEBUG_UNBIND`;
 - ◊ `Content.BIND_NOT_FOREGROUND`;
 - ◊ `Content.BIND_ABOVE_CLIENT`;
 - ◊ `Content.BIND_ALLOW_OOM_MANAGEMENT`;
 - ◊ `Content.BIND_WAIVE_PRIORITY`.

30.3.2 Scollegarsi da un bound service

Nel Componente che si scollega:

```
1 void unbindService(ServiceConnection con)
2 // Richiede il ServiceConnection di bindService
3
4 // Nel service:
5     boolean onUnbind(Intent intent);
6
7 // Se true allora sarà richiamato onRebind al prossimo primo bind
```

30.4 IBinder onBind(Intent intent)

`onBind()` restituisce l'interfaccia di comunicazione. Ogni altro componente ottiene la stessa istanza senza chiamare `onBind()`.

Per comunicare con un Service:

- ▶ Custom Binder Class;
- ▶ Usare Messenger;

30.4.1 Classe Binder

```
1 public class LocalBinder extends Service {
2     private final IBinder mBinder = new LocalBinder();
3     // Si può definire in una classe esterna
4     public class LocalBinder extends Binder{
5         LocalService getService(){
6             return LocalService.this;
7         }
8     }
9
10    @Override
11    public IBinder onBind(Intent intent){
12        return mBinder;
13    }
14 }
```

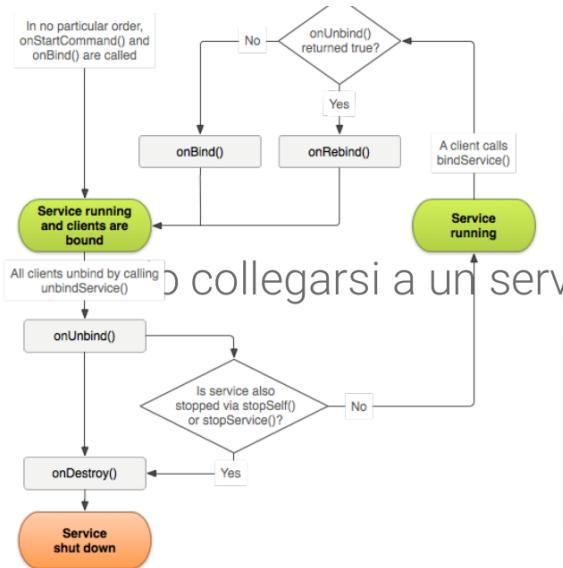
30.5 Usare Messenger

Utile per comunicare con processi remoti (IPC).

```
1 public class MessengerService extends Service {
2     static final int MSG_SAY_HELLO = 1;
3     class incomingHandler extends Handler{
4         @Override
5         public void handleMessage(Message msg){
6             switch (msg.what){
7                 case MSG_SAY_HELLO:
8                     Toast.makeText(getApplicationContext(), "hello!", Toast.LENGTH_SHORT).show();
9                     break
10                default:
11                    super.handleMessage(msg);
12                }
13            }
14        }
15        final Messenger mMessenger = new Messenger(new IncomingHandler());
16        @Override
17        public IBinder onBind(Intent intent){
18            Toast.makeText(getApplicationContext(), "binding", Toast.LENGTH_SHORT).show();
19            return mMessenger.getBinder();
20        }
21    }
22 }
```

30.6 Ciclo di vita di un bound service

Se un service è bound, il ciclo di vita è semplice. Se un service è avviato con `startService()`, allora è considerato `started`
→ Si deve stoppare con `stopSelf()` o `stopService()`



Comunicare quando visibile?
`onStart()` e `onStop()`

Comunicare sempre?
`onCreate()` e `onDestroy()`

30.7 Service management

Un service viene distrutto se servono risorse per un'attività con l'utente.

Importanza (dal meno al più importante):

- ▶ Service in foreground;
- ▶ Service legati a componenti vicini ad utente;
- ▶ Service in background;
- ▶ Long-running service.

Il service verrà riavviato quando ci saranno abbastanza risorse (dipende dal valore ritornato da `onStartCommand()`).

31 Lezione del 10-12

Carrellata di progetti per chi è interessato a tirocini interni (con Cutugno).

Strumenti per la ricerca:

- ▶ Python;

- Android.

Temi di ricerca con una laurea triennale:

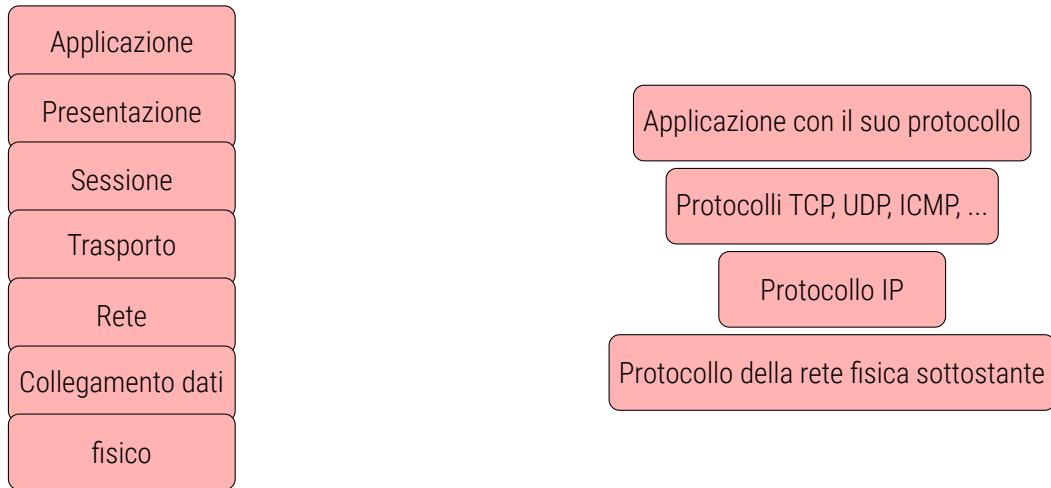
- Testing di strumenti AI a scatola chiusa
 - ❖ Riconoscimento del parlato (wav to string e viceversa);
- Interazione Uomo-Museo (Museo di San Martino).
- L'uso di schermi multitouch per l'introduzione al Museo.

32 Lezione del 13-12

32.1 Architettura Client Server

Architettura base per la comunicazione tra sistemi diversi.

32.2 Protocolli di comunicazione



32.3 Socket

Riprendendo da Java abbiamo

```

1 public class Socket implements Closeable
2 // Eredita da java.lang.Object e java.net.Socket
  
```

32.3.1 Socket client e server

Socket Client

- ▶ Aprire una connessione verso il server;
- ▶ Interagire;
- ▶ Chiudere la connessione.

Socket Server

- ▶ Aprire una ServerSocket;
- ▶ Accettare connessioni;
- ▶ Avviare un thread per ogni client connesso;
- ▶ Interagire;

32.3.2 Leggere da una Socket

```
1  /*
2
3   InputStream
4   InputStreamReader
5   BufferedReader
6
7 */
8  input = new BufferedReader(new InputStreamReader(socket.getInputStream));
```

32.3.3 Scrivere da una Socket

```
1  /*
2
3   OutputStream
4   OutputStreamWriter
5   BufferedWriter
6   PrintWriter
7
8 */
9  PrintWriter out = new PrintWriter(new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()
   )), true); // con il true puliamo il buffer
```

32.3.4 Esempio

Server

```
1  private class ServerThread implements Runnable {
2      private int SERVERPORT = 5415;
3      private ServerSocket serverSocket;
4      public void run(){
5          Socket socket = null;
6          serverSocket = new ServerSocket(SERVERPORT);
7          while(!Thread.currentThread().isInterrupted()){
```

```

8     socket = serverSocket.accept();
9     CommunicationThread commThread = new CommunicationThread(socket);
10    new Thread(commThread).start();
11 }
12 }
13 }
```

Communication Thread

```

1 class CommunicationThread implements Runnable {
2     private Socket clientSocket;
3     private BufferedReader input;
4
5     CommunicationThread(Socket clientSocket){
6         clientSocket = clientSocket;
7         input = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
8     }
9     public void run(){
10
11         while(!Thread.currentThread().isInterrupted()){
12             final String read = input.readLine();
13             if(read==null) break;
14             Log.d("ServerThread", "read: " + read);
15         }
16     }
17 }
18 }
```

Client

```

1 InetAddress serverAddr;
2 serverAddr = InetAddress.getByName(SERVER_IP);
3 sock = new Socket(serverAddr, SERVERPORT);
4
5 PrintWriter out = new PrintWriter(
6                     new BufferedWriter(new OutputStreamWriter(sock.getOutputStream())),
7                     true);
8
9 out.println(<str>);
out.close();
```

32.4 HTTP (HyperText Transfer Protocol)

```
1 public abstract class HttpURLConnection extends URLConnection
```

32.4.1 Flusso di utilizzo

1. Aprire la connessione;
2. Preparare la richiesta (URI, metadata, credenziali, cookies);
3. Inviare il corpo della richiesta (opzionale);
4. Leggere la risposta;
5. Disconnettere.

32.4.2 Esempio

```
1 URL url = new URL("http://www.android.com/");
2 HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
3
4 try{
5     InputStream in = new BufferedInputStream(urlConnection.getInputStream());
6     readStream(in);
7 }finally{
8     urlConnection.disconnect();
9 }
```

32.4.3 Gestione della risposta

In caso di problemi `getInputStream()` lancia `IOExcption`

`InputStream getErrorStream()` risulta utile per gestire le eccezioni

32.4.4 Richieste POST

La richiesta prevede un corpo del messaggio.

```
1 HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
2 try {
3     urlConnection.setDoOutput(true);
4     urlConnection.setChunkedStreamingMode(0); // Utile per le performance
5     OutputStream out = new BufferedOutputStream(urlConnection.getOutputStream());
6     writeStream(out);
7
8     InputStream in = new BufferedInputStream(urlConnection.getInputStream());
```

```
9     readStream(in);  
10 } finally{  
11     urlConnection.disconnect();  
12 }  
13 }
```

32.4.5 Performance

Di default, Input e Output non sono bufferizzati.

32.5 Android Design Support Library

Strumento utile per la UI (dalla 2.1)

32.5.1 Nel build grandle

```
dependencies{  
    compile 'com.android.support:design:<target-version>' // Deprecated  
    implementation 'com.android.support:design:<target-version>' // Recommended  
}
```

32.5.2 Navigation View

```
1 <android.support.v4.widget.DrawerLayout  
2     xmlns:android="http://schemas.android.com/apk/res/android"  
3     xmlns:app="http://schemas.android.com/apk/res-auto"  
4     android:layout_width="match_parent"  
5     android:layout_height="match_parent"  
6     android:fitsSystemWindows="true"  
7     <!-- Your content layout -->  
8     <android.support.design.widget.NavigationView  
9         android:layout_width="match_parent"  
10        android:layout_height="match_parent"  
11        android:layout_gravity="start"  
12        app:headerLayout="@layout/drawer_header"  
13        app:menu="@menu/drawer">  
14     </android.support.v4.widget.DrawerLayout>  
15  
16     <group android:checkableBehaviour="single">  
17         <item  
18             android:id="@+id/navigation_item_1"  
19             android:checked="true"  
20             android:icon="@drawable/ic_android"  
21             android:title="@string/navigation_item_1"/>  
22 
```

```

23     <item
24         android:id="@+id/navigation_item_2"
25         android:checked="true"
26         android:icon="@drawable/ic_android"
27         android:title="@string/navigation_item_2"/>
28     </group>

```

33 Lezione del 15-12

Breve introduzione su come iniziare ad usare Azure.

33.1 ListView (deprecati in favore di RecyclerView)

View che mostra elementi in una lista scorrevole. Usa un `Adapter` che fa da ponte tra un `AdapterView` e i dati.

I dati possono provenire da diverse fonti:

- ▶ Cursor (`CursorAdapter`);
- ▶ `ArrayList` (`ArrayAdapter`);
- ▶ `Map`.

Ognuna delle quali richiede un `ListAdapter` diverso.

33.1.1 Esempio - CursorAdapter

```

1  public class CustomAdapter extends CursorAdapter {
2      public CustomAdapter(Context context, Cursor cursor, int flags) {
3          super(context, cursor, 0);
4      }
5
6      @Override
7      public View newView(Context context, Cursor cursor, ViewGroup parent) {
8          return LayoutInflater.from(context).inflate(R.layout.list_item, parent, false);
9      }
10
11     @Override
12     public void bindView(View view, Context context, Cursor cursor) {
13         TextView textViewName = (TextView) view.findViewById(R.id.lName);
14         TextView textViewAge = (TextView) view.findViewById(R.id.lAge);
15
16         String body = cursor.getString(cursor.getColumnIndexOrThrow(UsersDatabaseHelper.STDT_NAME));
17         int age = cursor.getInt(cursor.getColumnIndexOrThrow(UsersDatabaseHelper.STDT_AGE));
18     }

```

```
19     textViewName.setText(body);
20     textViewAge.setText(String.valueOf(age));
21 }
22 }
```

33.1.2 Visualizzazione custom

Molto spesso capita di gestire dati variabili. È quindi necessario utilizzare un Adapter custom.

33.2 ActionBar

Una barra mostrata nell'Activity:

- ▶ Titolo;
- ▶ Affordance di navigazione;
- ▶ Altri elementi interattivi.

Da una Activity si recupera l'ActionBar con `ActionBar getActionBar()`.

Può essere dinamica:

- ▶ Può scomparire;
- ▶ Cambiano le action;
- ▶ La posizione delle action è gestita da Android.

33.2.1 onOptionItemSelected()

```
1 @Override
2 public boolean onOptionsItemSelected(MenuItem item) {
3     switch (item.getItemId()) {
4         case R.id.action_settings
5             // Lo user clicca sull'oggetto Setings -> mostra la UI dei setting
6             return true;
7         case R.id.action_favorite
8             // Lo user clicca sull'oggetto Favorite -> mostra la UI dei Favorite
9             return true;
10        default
11            // L'azione dell'utente non è stata riconosciuta, invochiamo la superclasse per gestire la
12            // situazione
13            return super.onOptionsItemSelected(item);
14    }
}
```

33.2.2 Affordance di navigazione

L'ActionBar permette di poter tornare all'Activity home (Up Action). È necessario impostare la parent Activity nel manifest.

33.2.3 ToolBar

Un'ActionBar può essere aggiunta a runtime tramite ToolBar.

Si definisce nel Layout.

```
1 Toolbar myToolbar = (Toolbar) findViewById(R.id.my_toolbar);  
2 setSupportActionBar(myToolbar);
```

Richiede che:

- ▶ L'Activity sia AppCompatActivity;
- ▶ Il tema delle activity sia XXX.NoActionBar.