

# La shell di Unix

*Sezione 1 del corso:*

**Laboratorio di Sistemi Operativi**

*A.A. 2009/2010*

Alessandro Valenti

# Introduzione

- Vedremo come utilizzare Unix/Linux come host di sviluppo
  - uso della shell comandi
  - uso dell'editor vim
  - uso del compilatore gcc e del linker
- e come target per le nostre applicazioni
  - interpretate, in linguaggio shell
  - eseguite, dopo la compilazione dei sorgenti in linguaggio C

# Laboratorio di Sistemi Operativi

- corso facoltativo: 3 CFU
  - abbinato al corso di Sistemi Operativi
  - due voti distinti e due registrazioni
  - un unico esame: realizzazione di una applicazione di ricerca shell e di un applicativo concorrente in linguaggio C
- materiale didattico: dispense ed esercizi su [www.coeing.it/didattica/labso](http://www.coeing.it/didattica/labso)
- [alessandro.valenti@unimore.it](mailto:alessandro.valenti@unimore.it)
- ricevimento: dopo le lezioni in Lab. Base o su appuntamento.

# Obiettivi del corso

- Uso della shell di Unix
- Uso dell'editor vim
- Esperienze “ragionate” di programmazione in linguaggio shell (script) ed in linguaggio C (compilati)
- Comprensione dei sistemi di programmazione concorrente orientata ai processi e conoscenza dei sistemi di IPC e sincronizzazione
- Capacità di progettare programmi concorrenti
- Padronanza di Unix come host di sviluppo

# Organizzazione delle lezioni

- Il corso si divide in due sezioni, in modo analogo al corso di Sistemi Operativi, con il quale cercheremo di essere allineati:
  - Prima parte: Shell, 5 settimane circa
  - Seconda parte: C, 7 settimane circa
- Dopo la prima parte, prova in itinere sulla sola shell (50% del voto)
- Lezione in aula il giovedì dalle 12 alle 13 per tutti
- Esercitazioni in Laboratorio Base per turni di max. 60 persone, 2 per PC
- Studio ed esercitazioni individuale a casa su macchina Linux, emulatore Cygwin o macchina virtuale

# Turni di Laboratorio

- Il Laboratorio Base è a nostra disposizione per 6 ore settimanali
  - giovedì dalle 14 alle 17
  - venerdì dalle 14 alle 17
- La distribuzione per turni dipende dal numero totale di iscritti (dalla registrazione che faremo oggi)
  - due turni da 2 ore, se il numero totale non supera 120
    - 1: gio 14-16, 2: ven 14-16
  - tre turni da 1,5 ore, in caso contrario
    - 1: gio 14-15.30, 2: ven 14-15.30, 3: gio o ven 15.30-17

# Laboratorio in prima settimana

- Il Lab. Base verrà utilizzato domani 5 marzo per la registrazione al corso dalle 14 in poi
- Le ore disponibili oggi dalle 14 alle 16 sono utilizzate per una esercitazione sull'editor VIM della durata di circa 1 ora con accesso libero (senza superare il numero di 60 presenti)
  - Al boot, selezionare *Linux*
  - Accedere con nome utente=vostro numero di tessera e password=password della posta elettronica, oppure in caso di problemi
    - Accedere con utente=*guest*
    - password=*prova*
  - Al *prompt*, digitate `vimtutor it`
  - Al termine digitare `exit`

# Macchina virtuale

- Scaricare VMWare Player (free, da <http://www.vmware.com/download/player>)
- Decomprimere `coeing_apl.zip` nella directory in cui VMWare cerca le macchine virtuali
- Eseguire il Player e selezionare la macchina virtuale appena decompressa
  - Utente normale: `dvp` password: `coeing.it`
  - Superuser: `root` password: `coeing.it`
- Tutto il necessario per il nostro corso, compreso `vimtutor`
- Per spegnere, entrare come `root` e digitare `halt`

# Il sistema operativo secondo Unix

- È il gestore delle risorse base della macchina
  - tempo CPU (time slicing)
  - memoria (virtual memory)
  - dispositivi (devices)
    - la gestione dei device è solo quella di basso livello
    - interrupt, memory mapping, file system mapping
    - il S.O. non "sa" cosa fanno i dispositivi ma si limita ad organizzarli in poche categorie di base
- Espleta i suoi servizi attraverso le chiamate di sistema
- Non ha una interfaccia utente

# Un sistema Unix completo

Un dispositivo che adotta Unix o una sua derivazione *standard* deve avere una dotazione software di almeno due componenti

- Il Sistema operativo propriamente detto, o Kernel
- Uno o più programmi applicativi che il kernel metterà in esecuzione come **processi** e che potranno usufruire dei servizi del S.O. attraverso le **syscall**.

Nelle installazioni Unix su PC, server e workstation, troviamo diversi **file eseguibili** posizionati opportunamente nel **filesystem** che, messi in esecuzione dal kernel, concorrono a generare il sistema completo.

In una installazione embedded, un solo eseguibile potrebbe essere sufficiente a completare il sistema.

# Processi di sistema e processi utente

L'esecuzione di un file eseguibile in una macchina Unix equivale alla creazione di un processo

- alcuni processi corrispondono a eseguibili *di base* o *di sistema*
- altri processi possono essere frutto di azioni interattive di un utente
- il kernel tratta tutti i processi in modo equivalente, una distinzione fra processi di sistema e processi utente è quindi solo formale
- esiste un solo processo privilegiato per il kernel: il suo unico privilegio è quello di non dover avere un processo padre poichè messo in esecuzione direttamente dal kernel all'avvio.

# Servizi e demoni

- Si possono quindi considerare processi di sistema quelli che forniscono dei **servizi** e che normalmente vengono messi in esecuzione dal processo di avvio o **init**.
- I processi di sistema che svolgono la loro attività in modo non interattivo con l'utente sono spesso indicati anche come **demoni**.
- L'accesso di un utente "umano" al sistema unix avviene per mezzo di alcuni processi che, con i servizi del kernel, leggono e scrivono su file speciali associati ai dispositivi di I/O
  - console locali o remote, detti anche terminali
  - terminali grafici X via socket (il server è un applicativo, non fa parte del kernel)

# Le shell

Ogni sistema Unix prevede (o può comunque consentire) un accesso da terminale (anche Mac OS X!)

- si può fare tutto da terminale
- attraverso delle interfacce spartane ma complete dette **shell**
- che possono eseguire anche comandi *batch*, mediante **script**
  - la programmazione di shell è talmente consolidata sotto Unix al punto che tutto il processo di avvio è controllato da script
  - per usare Unix bisogna avere confidenza con una shell

# Gli utenti

- L'accesso di ogni utente è verificato dal sistema di login
- un utente ha accesso solo dopo aver indicato una coppia *nome-password* presente nel database degli utenti (uno o due semplici file di testo)
- L'unico utente particolare è *root*, che possiede tutti i diritti di accesso su file e directory
- Ogni *processo* in esecuzione è associato con l'utente ed accede a file e directory secondo i permessi accordati a tale utente o gruppo di utenti
- Ogni *file* presente nel filesystem è associato all'utente che ne ha il controllo e ad un gruppo di utenti

# Command line

- L'utente che ha ottenuto l'accesso al sistema dispone di una interfaccia comandi, la shell (quella che preferisce, tipicamente sh/bash o csh)
- L'interazione con la shell avviene per mezzo di *command line*
- Una command line è una sequenza di caratteri terminata da un ritorno a capo (LF, ascii 0x0A)
- Una sessione di lavoro è una sequenza di command line, che la shell considera alla stregua di un *file*
- Si può terminare una sessione o con il comando esplicito *exit* o semplicemente forzando una *fine di file* con la combinazione CTRL-D
- Gli spazi in una command line hanno il particolare significato di *separatori di argomenti*
- Le particolarità dei caratteri LF e spazio possono essere disabilitate mediante *escaping*

# Esempio di cmdline

```
ls
```

- Un singolo elemento che identifica il nome di un *comando*

```
ls -l
```

- Due elementi, il comando e un *argomento*. Se inizia con il carattere - si dice *opzione*

```
ls file1
```

- Due elementi, il secondo è un argomento non opzione e specifica al comando ls di cercare solo i file che hanno nome uguale a “file1”

# Directory corrente

Per identificare un file nel filesystem possiamo fornire:

- il percorso *assoluto* che si deve compiere attraversando la gerarchia delle directory dalla radice al file stesso
- il percorso *relativo* alla *directory corrente*

Ogni processo ha una associazione con una directory del filesystem rispetto alla quale risolve i nomi relativi.

- Questa è detta directory corrente del processo
- Si modifica con il comando `cd`

```
cd /
```

- Posiziona la dir. corrente (cwd) nella radice

# Prompt

- Nella esercitazione libera della scorsa settimana, abbiamo iniziato ad utilizzare Linux in laboratorio e/o sulla VM
- Il Sistema Operativo, attraverso le richieste di servizio avanzate da alcuni applicativi di sistema, ha messo in esecuzione per noi un *processo* al quale ha assegnato il compito di eseguire un *programma*
- Il programma in questione è una shell, per l'esattezza la shell *bash*
- Quando la shell è *pronta* ad accettare comandi, emette una sequenza di caratteri detta *prompt*

```
valealex@copperbottom: ~ $
```

# Modularità

L'interfaccia utente di Unix è costituita da una serie di applicativi indipendenti e da un ambiente in cui tali applicativi possono essere invocati, combinati fra loro e personalizzati attraverso argomenti ed opzioni.

- L'ambiente di esecuzione è rappresentato da una delle shell disponibili, per noi sarà la bash
- Gli applicativi sono una raccolta di strumenti semplici e, come vedremo, è possibile estenderne la lista con nuovi creati dall'utente, sia binari che script
- Gli applicativi e la sintassi della shell costituiscono un linguaggio modulare con il quale realizzare command line complesse e intere applicazioni

# Il processo Unix

- La modularità dei comandi Unix è fondata sul concetto di *processo*
- Il Sistema Operativo mantiene una lista di processi e fornisce ad ognuno di essi:
  - Una frazione (eventualmente nulla) del tempo CPU
  - Aree di memoria virtuale
  - Informazioni sull'esecuzione come directory corrente, utente associato, tabella di file aperti
- In nessun modo il programma eseguito in un processo può accedere ad aree di memoria di altri processi. Il solo modo per scambiare dati o eventi fra processi è di utilizzare il S.O.

# Utenti

Ad ogni processo corrisponde una associazione con un utente ed un gruppo, attraverso la quale il S.O. applica i permessi di accesso.

In realtà l'associazione avviene con più descrittori di utente e gruppo, al fine di poter gestire i bit *suid* e *sgid*.

- *real*: utente o gruppo che ha messo in esecuzione il processo, indipendente da *suid/sgid*
- *effective*: utente o gruppo per il quale si applicano i permessi di accesso. Uguale a quello del file eseguibile in caso di *suid/sgid*, uguale ai *real* in caso contrario
- *saved*: usato per commutare fra loro *effective* e *real* e vice versa

# CWD

Ad ogni processo corrisponde una associazione con una directory nel filesystem

- rispetto a tale directory, è possibile specificare nomi di file in modo *relativo*:
  - *semplice* se il file si trova esattamente nella directory corrente
  - *non semplice* se il file si trova in qualche sottodirectory
- La directory corrente si cambia con il comando `cd`
  - `cd ~` o `cd` portano la cwd alla *home*
  - `cd /` porta la cwd alla radice
  - `cd -` porta la cwd a quella precedente

# Variabili d'ambiente

Anche il contenuto della memoria fa, ovviamente, parte delle informazioni proprie di un processo. Un'area della memoria di processo è utilizzata per contenere le *variabili d'ambiente*

- Sono stringhe, nella forma `nome=valore`
- Con i comandi della shell possono essere create, modificate o eliminate
- In nessun modo un processo può alterare le variabili di altri processi
  - `a=xyz` crea o modifica la variabile `a`, valore "xyz"
  - `a=` crea o modifica la variabile `a` assegnando la stringa vuota come valore
  - `unset a` elimina la variabile `a`

# Descrittori

- Ultima informazione di nostro interesse associata ai processi Unix è una lista di file aperti, ognuno dei quali è identificato da un numero positivo detto *descrittore* che corrisponde all'indice in tale lista.
- I processi *ereditano* tale lista dal processo che li mette in esecuzione, ma una volta attivi possono modificare la propria lista (solo quella!) aprendo o chiudendo file
- Normalmente un processo viene creato con almeno 3 descrittori attivi:
  - 0 per lo *standard input*
  - 1 per lo *standard output*
  - 2 per lo *standard error*

# Input

Quando un processo esegue delle *letture* da un file aperto, utilizza tale file come input

- La shell utilizza lo standard input per leggere le command line provenienti dalla *console*
- Quando la shell mette in esecuzione un comando, crea un nuovo processo al quale trasmette tale descrittore (0) così che il programma corrispondente possa leggere dal medesimo *canale*
- Normalmente, quando più processi condividono lo stesso descrittore in lettura, solo uno di essi dovrà effettivamente eseguire la read. Gli altri processi (ad esempio la shell che ha messo in esecuzione il comando) vanno tipicamente *in attesa*.

# Output

Quando un processo esegue delle *scritture* su un file aperto, utilizza tale file come output

- La shell utilizza lo standard output per scrivere i risultati dei comandi sulla *console*
- Quando la shell mette in esecuzione un comando, crea un nuovo processo al quale trasmette tale descrittore (1) così che il programma corrispondente possa scrivere sul medesimo *canale*
- Quando più processi condividono lo stesso descrittore in scrittura, le operazioni di scrittura vengono accodate sullo stesso file. Quando la shell esegue un comando, l'output del comando stesso appare di seguito a quello precedentemente generato dalla shell sulla console.

# Semplice ridirezione

La condivisione dei file aperti non è l'unica possibilità. Attraverso una opportuna sintassi delle command line, la shell può operare delle modifiche sulla lista dei file aperti per il nuovo processo

- `>filename` in seguito ad un comando, provoca la chiusura dello standard output corrente (console) e la sua sostituzione con il file di nome `filename`
- `<filename` in seguito ad un comando, provoca la chiusura dello standard input corrente (console) e la sua sostituzione con il file di nome `filename`
- `>>filename` il file non viene sovrascritto come per la ridirezione semplice ma l'output va in append

# Esecuzione di cat

`cat` è un comando presente in ogni versione di Unix che provvede a leggere un file il cui nome (relativo o assoluto) venga dato come argomento ed a riversarne il contenuto su *stdout*

- `cat /etc/passwd` visualizza il contenuto del file `/etc/passwd`
- `cat /etc/passwd > miofile` scrive su `miofile` il contenuto di `/etc/passwd`
- `cat > miofile` riposta su `miofile` tutti i caratteri digitati da console, fino al CTRL-D
- In assenza di argomenti, `cat` opera la lettura dal proprio *stdin*

# File e directory

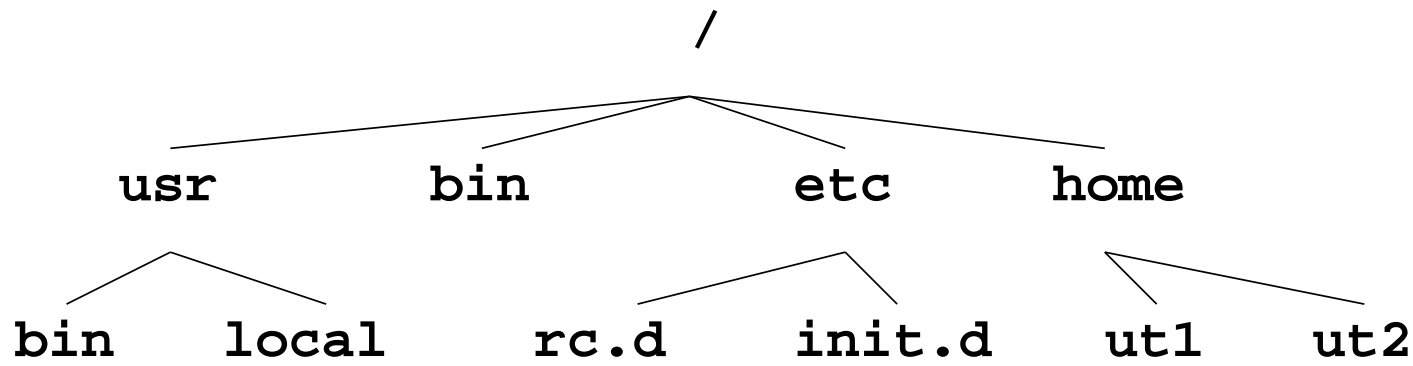
- Ogni dispositivo che adotta Unix deve avere un filesystem
- Rappresenta un contenitore di *file*, memorizzati su uno o più dispositivi fisici
- L'associazione di un dispositivo fisico (es. partizione) al filesystem è indicata come *mount*
- Esiste una unica *radice* del filesystem, indicata con il simbolo “/” (barra) e rappresentata da un file di tipo *directory*
- Una *directory* è una lista di associazioni fra *nomi* e contenitori di informazioni, gli *i-node*

# Filesystem

Università di Modena e Reggio Emilia

Laboratorio di Sistemi Operativi

Il filesystem di unix è una struttura gerarchica avente per radice la root "/" del filesystem



**bin**

Dir.corrente: **/usr**

**/usr/local/bin** assoluto

**local/bin** relativo

# Tipiche directory

`/bin` file eseguibili principali di sistema, normalmente eseguibili dagli utenti

`/dev` file speciali per i dispositivi (o device)

`/etc` file di configurazione del sistema e dei programmi

`/lib` librerie di sistema statiche e dinamiche

`/tmp` file temporanei

`/usr` file di uso comune per tutti gli utenti come programmi, documentazione, esempi. Spesso `/usr` viene *montata* in sola lettura per migliorare la velocità di accesso in lettura e la sicurezza; in tal caso la gerarchia `/var` viene in aiuto.

`/var` file di uso comune, come `/usr`, ma che richiedono l'accesso in scrittura

# Accesso

- Quando un processo richiede al S.O. l'esecuzione di una operazione su un file, Unix verifica le autorizzazioni concesse all'*utente* cui è associato
- Le autorizzazioni sono specificate per la lettura, la scrittura e l'esecuzione di un file
- Ogni file ha tre insiemi di informazioni di autorizzazione.
  - un insieme applicato quando l'utente del processo è lo stesso che *possiede* il file
  - un secondo insieme viene applicato se non è verificata la condizione precedente ma l'utente appartiene ad un particolare *gruppo* di utenti indicato nel file
  - un terzo ed ultimo insieme si applica invece a tutti gli utenti che non rientrano nelle precedenti casistiche

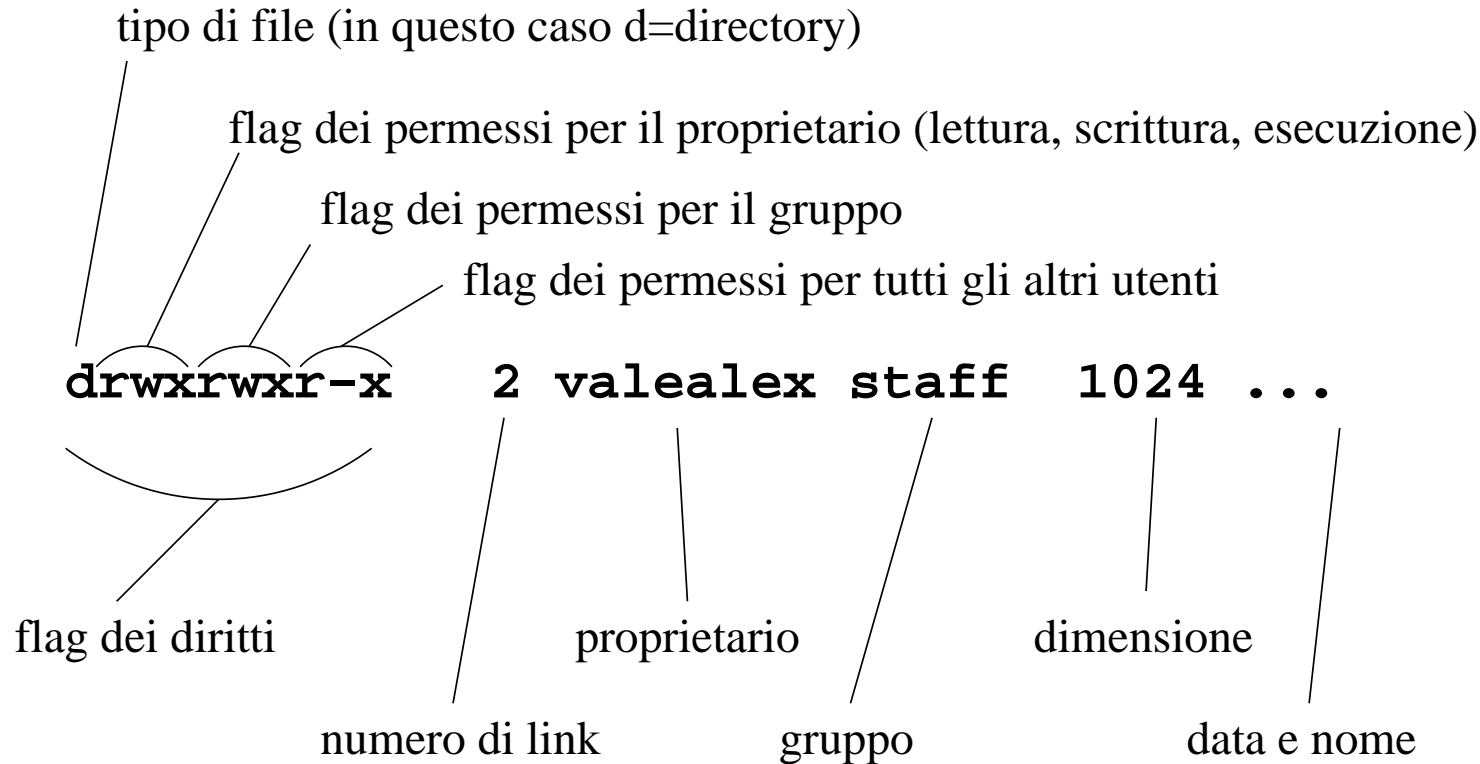
# File e Directory

- La directory è un file *particolare* o meglio, *non regolare*. In Unix, ogni oggetto che trova posto nel filesystem è un file; parleremo di file *regolari* quando questi sono meri contenitori di byte.
- Un file di *tipo directory* contiene un elenco di *nomi* e la relativa associazione con un file del filesystem; questo, a sua volta, può essere di tipo directory dando luogo ad una sotto directory della precedente.
- Ogni file nel filesystem è univocamente associato ad un *i-node*; allo stesso i-node possono corrispondere più entry in una o più directory, dando origine agli *hard link*.

# Proprietà dei file

Università di Modena e Reggio Emilia

Laboratorio di Sistemi Operativi



# Bit dei diritti

A sinistra, il corrispondente valore in *ottale*, ovvero cifre in base 8

bit	carattere	significato
00400	r????????	Se 'r', accesso in lettura consentito al proprietario
00200	?w????????	Se 'w', accesso in scrittura consentito al proprietario
00100	??x???????	Se 'x', accesso in esecuzione consentito al proprietario
00040	???r??????	Se 'r', accesso in lettura consentito a chiunque appartenga al gruppo
00020	????w?????	Se 'w', accesso in scrittura consentito a chiunque appartenga al gruppo
00010	?????x????	Se 'x', accesso in esecuzione consentito a chiunque appartenga al gruppo
00004	???????r??	Se 'r', accesso in lettura consentito a tutti i rimanenti utenti
00002	???????w?	Se 'w', accesso in scrittura consentito a tutti i rimanenti utenti
00001	???????x	Se 'x', accesso in esecuzione consentito a tutti i rimanenti utenti

# Bit speciali

In aggiunta ai 9 bit relativi ai permessi di lettura, scrittura ed esecuzione, esistono tre bit speciali; come per i bit di esecuzione, anche i tre bit speciali cambiano lievemente significato quando applicati a file regolari o a directory.

bit	carattere	significato
04000	??s??????	Se 's', attiva il flag suid o <i>set user id</i>
02000	?????s???	Se 's', attiva il flag sgid o <i>set group id</i>
01000	??i??????t	Se 't', attiva lo sticky bit o <i>restriction deletion flag</i>

# Standard error

- Quando un comando rileva un errore, ad esempio si cerca di accedere ad un file per il quale non si hanno idonei permessi, il comando emette normalmente un messaggio di avvertimento.
- Per evitare il mescolamento di tali messaggi con l'output principale dei comandi, Unix introduce un descrittore per un file da usare come destinazione degli errori, *stderr*
- `ls file_assente > /dev/null`
- `ls file_assente 2> /dev/null`
- `ls file_assente > /dev/null 2>&1`
- `ls file_assente 2>&1 > /dev/null`

# Shell e ridirezione

Le istruzioni di ridirezione sono utilizzate dalla shell per predisporre opportunamente l'esecuzione del processo, secondo questa sequenza:

- Il processo-shell crea un nuovo processo (figlio)
- Il processo-shell predispone i descrittori del figlio
- Il processo-shell attende la conclusione del processo figlio
- Il processo-figlio *esegue* il file eseguibile corrispondente al comando esterno invocato
- Il processo-figlio termina
- Il processo-shell torna *attivo* e presenta il prompt

# Pipeline

La ridirezione su file non è l'unica possibile modifica ai descrittori; Unix mette a disposizione dei file *speciali* che consentono a più processi una esecuzione concorrente:

- La shell può mettere in esecuzione più processi figli in *parallelo*
- instaurando una comunicazione (pipe) fra l'output di un processo e l'input del successivo
- e il sistema operativo provvede a sincronizzare i processi

```
ls -l | cut -c 1-8 | grep '^d'
```

# Command line e variabili

- Le variabili d'ambiente possono essere utilizzate nelle command line in quanto la shell (ogni shell) provvede a sostituirle con il relativo valore

```
filename=miofile  
cat $filename
```

- La sostituzione avviene *prima* dell'esecuzione della cmdline:

```
comando=cat  
$comando miofile
```

- Se una variabile non è definita, viene trasformata nella stringa nulla ""

```
echo a $x b | wc -w
```

# Command substitution

- La shell è anche in grado di sostituire una parte della command line con l'*output* prodotto da un comando.
- Il comando deve essere racchiuso fra apici inversi o fra parentesi precedute dal dollaro

```
echo `pwd`
```

```
echo $(pwd)
```

- Il comando da sostituire viene ovviamente eseguito *prima* della command line in cui compare

# Exit value

- Ogni processo genera un valore numerico in uscita che viene detto *exit value*
- I comandi Unix normalmente utilizzano l'exit value per trasmettere l'esito al chiamante, ad esempio alla shell
- La shell è in grado di recuperare l'exit value dell'ultimo comando eseguito con la variabile `$?`

```
test 1 -gt 0  
echo $?
```

- Solitamente il valore zero corrisponde a “vero” o “successo”

# Qualche comando

cat	riversa il contenuto di un file o di stdin su stdout
cd	cambia directory corrente
chmod	cambia i permessi di un file
cp	copia un file
echo	mette su stdout i caratteri che lo seguono nella cmdline
grep	cerca una stringa su stdin e la riversa su stdout (semplificato)
ls	lista i file su stdout
man	visualizza il manuale di un comando
mkdir	crea una directory
ps	lista i processi su stdout
pwd	riporta su stdout il nome assoluto della dir corrente
rm	cancella un file
rmdir	cancella una directory
test	verifica una espressione
wc	conta caratteri, parole o linee di stdin con output su stdout

# Shell script

- Una command line o una sequenza di command line possono essere memorizzate in un file di testo
- L'attribuzione del permesso di eseguibilità a tali file li trasforma in shell script
- Ogni shell script viene *interpretato* in un *processo* che esegue il programma *bash*, quindi una shell
- Convenzionalmente, gli shell script possono essere memorizzati in file con estensione `.sh`, anche se questo non è necessario.

# Esecuzione con source

- È possibile ottenere l'interpretazione di uno shell script da:
  - La shell attualmente in esecuzione
  - Una nuova shell *figlia* della shell in esecuzione
- Nel primo caso si deve usare il comando:  
`source script.sh`
- oppure l'equivalente comando `./`:  
`. script.sh`

# Esecuzione come file

- Ogni file *eseguibile* può essere messo in esecuzione formulando una opportuna command line
- Il primo elemento di una command line viene utilizzato dalla shell per:
  - identificare un comando
  - identificare un file eseguibile
- Se tale elemento è un nome di file assoluto o relativo, allora la shell mette in esecuzione *quel file*, altrimenti cerca un file eseguibile avente quel *nome* in una lista di directory detta *PATH*

# file e comandi

- Command line in cui il primo elemento è un nome di file:

```
./script.sh a b c
```

- Command line in cui il primo elemento è un nome di comando:

```
script.sh a b c
```

- Se non esiste alcun file eseguibile avente per nome “script.sh” in nessuna delle directory elencate nel PATH, la shell emette un errore:

```
bash: script.sh: command not found
```

# PATH e la ricerca

- La shell utilizza la variabile d'ambiente `PATH` come lista delle directory nelle quali cercare i file eseguibili corrispondenti ai comandi
- Le directory sono separate dal carattere `:`
- La ricerca avviene in maniera sequenziale; in caso di più file omonimi, viene eseguito quello nella directory elencata per prima
- Ogni shell in esecuzione ha il proprio valore della variabile `PATH`, come del resto avviene per ogni variabile; non esiste un concetto di `PATH` valido per tutti i processi ma ogni processo ha la propria

```
echo $PATH
```

```
/usr/local/bin:/usr/bin:/bin
```

# Modifica di PATH

- PATH è una variabile come tutte le altre, quindi è modificabile con una opportuna command line:

- Aggiunta di `miadir` al PATH:

```
PATH=miadir:$PATH
```

- Sovrascrittura di PATH

```
PATH=/dir1:/dir2/dir3:dir4:`pwd`:.
```

# Shell figlia

- Quando uno script viene eseguito *senza* il comando `source`, la shell attuale genera un nuovo processo figlio al quale delega l'esecuzione dello script,
- esattamente allo stesso modo in cui mette in esecuzione ogni altro comando,
- condividendo i descrittori correnti o operando ridirezioni,
- confezionando la lista degli argomenti estratti dalla command line,
- propagando una *copia* delle variabili d'ambiente,
- recuperandone l'exit value al termine

# Esportazione di variabili

- Le variabili definite in una shell vengono trasmesse alla shell figlia con questa modalità:
  - Se una variabile è stata creata e non *esportata*, allora non appare nella shell figlia
  - Se è stata creata e esportata, allora viene trasmessa
  - Se è stata ereditata, allora era sicuramente esportata e viene trasmessa
- L'esportazione si ottiene con il comando `export`, che può essere anche contestuale alla creazione

```
export PATH= ./ : $PATH
```

# Argomenti della cmdline

- Ogni shell può accedere agli elementi della command line che ne ha provocato l'esecuzione
- con \$# ottiene il numero di argomenti, zero se la command line conteneva solo il nome del file o del comando
- con \$0 ottiene il nome del primo elemento della command line, ovvero il nome del file script
- con \$1 ottiene il primo argomento, se presente
- \$2, \$3, . . . \${99} per gli argomenti successivi
- \$\* tutti gli argomenti in un'unica variabile

# true e false

- true e false sono due semplici comandi il cui unico obiettivo è quello di restituire un exit value sempre vero o falso rispettivamente
- sono comodi per il controllo delle parti di uno script che utilizzano l'elaborazione condizionale
- corrispondono comunque a file eseguibili

# if

- È il primo elemento sintattico del linguaggio shell
- Consente di condizionare l'esecuzione di una o più command line nel file script all'exit value di una command line

```
if condizione
then
command line 1
command line 2
fi
```

- Valide condizioni possono essere `true` o `false`

# test

- È un comando che verifica una condizione espressa attraverso argomenti della propria command line
- relativi a stringhe,
- numeri
- oppure elementi del filesystem

# test su stringhe

- Verifica di uguaglianza fra due stringhe:

```
test $s1 = $s2  
test $s1 != pippo  
test pippo = pippo
```

- Attenzione agli spazi! Sono necessari in quanto separatori.
- Attenzione alle stringhe vuote:

```
$s1=" "  
test $s1 != " "  
test "$s1" != " "
```

# test numerici

- Esegue controlli su stringhe che codificano numeri:

```
test $n1 -gt $n2
```

```
test $n1 -eq $n2
```

```
test $n1 -le $n2
```

- Combinazioni booleane

```
test $n1 -gt $n2 -a ! $s1 != $s2
```

- Con parentesi, ma attenti al quoting:

```
test ( ab = ab ) -a ( ab = cd )
```

```
test \( ab = ab \) -a \( ab = cd \)
```

```
test "( ab = ab )" -a "( ab = cd )"
```

# test su file

- File regolare, esistenza

```
test -f $filename
```

- Accessibilità in lettura e scrittura

```
test -r $filename -a -w $filename
```

- Directory

```
test -d $filename
```

- Directory esplorabile, attenzione ai nomi di file “strani”

```
test -d "$filename" -a -x "$filename"
```

# Pathname expansion

Rappresenta il meccanismo attraverso il quale la presenza di alcuni caratteri particolari in una command line ne provoca la sostituzione con uno o più nomi di file

- la lista dei file prodotta dalla *pathname expansion* è ordinata alfabeticamente
- i caratteri speciali sono detti *wild card* e possono essere combinati fra loro e con caratteri *normali* per descrivere dei *pattern*
- nel caso in cui il *pattern* descritto dai wild card non trovi candidati fra i nomi di file, allora la stringa descrittiva del pattern rimane invariata

# Wild card

- La presenza di un carattere \*, provoca il match di qualunque stringa
- un carattere ? sostituisce qualsiasi carattere, singolo
- una sequenza di caratteri racchiusi fra [ e ] sostituisce un singolo carattere preso fra quelli elencati

# Esempi

Supponiamo che nella directory corrente siano presenti i file `abc`, `ade`, `def` e `lmno`. Ecco il risultato di alcuni pathname expansion:

- `*`: `abc ade def lmno`
- `???`: `abc ade def`
- `a??`: `abc ade`
- `[a1]*`: `abc ade lmno`
- `l??`: `l??`

# Quoting

Alcuni caratteri hanno per la shell un significato particolare, come i wild card e il carattere \$. È tuttavia possibile istruire la shell ad astenersi da tale trattamento particolare mediante una operazione nota come *quoting*

- racchiudere una porzione di command line fra virgolette ne provoca il quoting *parziale*, disabilitando il pathname expansion ma mantenendo il significato del carattere \$
- il quoting parziale viene spesso utilizzato per raggruppare una sequenza di caratteri contenente spazi in un singolo argomento o per preservare la presenza di un argomento anche quando questo coincide con la stringa nulla

# Full quoting

- se la porzione di command line viene racchiusa fra apostrofi, allora si ha *full quoting* e ogni carattere speciale viene disabilitato
- facendo precedere un carattere dal *backslash* (\), si ottiene invece il quoting di quel singolo carattere

# exit esplicito

- Ogni processo genera un exit value che può essere recuperato dal processo padre
- nel caso di shell script, tale valore è quello restituito dall'ultima command line interpretata
- è possibile specificare un exit value a piacimento con la keyword `exit`  
`exit $value`

# for

Il `for` rappresenta un costrutto sintattico della shell che consente di:

- elaborare ciclicamente un insieme di command line, quelle comprese fra le keyword `do` e `done`
- il numero di cicli è determinato dal numero di argomenti che seguono la keyword `for`
- ad ogni iterazione, la variabile avente per nome quello indicato nel primo argomento assume uno dei valori presenti nella lista dopo la keyword `in`
- se la keyword `in` non viene specificata, allora come lista dei valori viene utilizzata quella degli argomenti della command line di invocazione

# esempio for

```
for var1 in abc def ghi  
do  
    echo $var1  
done
```

# for e pathname expansion

- Una delle modalità più utilizzate per esprimere la lista di valori per un for è il pathname expansion
- in tal caso la variabile di ciclo assume a turno un valore uguale al *nome* di tutti i file che corrispondono al pattern
- se questo patter è semplicemente espresso con un asterisco, allora la variabile assumerà a turno tutti i nomi di file/direttori presenti nella directory corrente, escludendo quelli che iniziano con il punto
- si può così generare rapidamente uno script che esegua delle command line su tutti i file di un direttorio

# espansione vuota

- attenzione ai casi in cui l'espansione non può avvenire, come in una directory vuota

```
for filename in *; do echo $filename; done  
*
```

# il case

Il comando `test` consente di verificare uguaglianza e disuguaglianza di stringhe. Il costrutto `case` della shell estende questo concetto:

- associando una lista di command line al match di un *pattern*
- permettendo la definizione dei pattern con pochi caratteri speciali

```
case $var in
    pattern)
        cmdline
        ;;
esac
```

# pattern per case

- `abc` solo la stringa “abc”
- `[abc]` solo stringhe di un singolo carattere, sia esso `a`, `b` o `c`
- `?` solo stringhe di un singolo carattere, qualunque esso sia
- `[abc]*` stringhe di almeno un carattere, per le quali il primo sia `a`, `b` o `c`
- `[!xyz]*` stringhe di almeno un carattere, per le quali il primo non sia né `x`, né `y` né `z` (`[^xyz]*` solo in bash)
- `*` qualunque stringa
- `*$2` qualunque stringa la cui parte terminale sia uguale al secondo arg.

# first case match

- Il case si limita ad eseguire le command line elencate di seguito al primo match positivo, fino alla lettura della sequenza ; ;
- Eventuali match successivi sono ignorati, quindi non serve alcun meccanismo di *break*
- Solitamente si mette come ultimo pattern il \* che provoca il match di qualunque stringa

# esplorazione directory

Gli shell script sono comodi per eseguire operazioni di ricerca o modifica in gerarchie di direttori

- attraverso la pathname expansion è possibile eseguire comandi su tutti i file elencati nella directory corrente
  - intendendo per file sia quelli regolari che le directory che eventuali file speciali

```
for filename in *  
do  
    cmdline su $filename  
done
```

# azione su file e directory

- Le command line eseguite per ogni *elemento* della directory possono essere specializzate in funzione del tipo di elemento, se file regolare o directory
- Per esplorare una gerarchia, ogni elemento di tipo directory deve essere a sua volta esplorato in modo ricorsivo
- Se lo script è opportunamente progettato, una sua ulteriore istanza sotto forma di processo figlio raggiunge l'obiettivo

# script ricorsivi

Uno script è invocabile ricorsivamente se

- è invocabile come comando, quindi è un file eseguibile raggiungibile da una directory elencata nel PATH corrente
- il PATH corrente è esportato ad ogni processo figlio (export)
- è in grado di recuperare il proprio *nome*, ad esempio con la variabile di shell \$0, quando invocato a sua volta come comando

```
script.sh -> $0=/assoluto/di/script.sh (OK)
```

```
./script.sh -> $0=./script.sh (KO)
```

# change directory

- Nella progettazione di script ricorsivi, sono possibili due approcci al cambio di directory
  - il comando `cd` viene eseguito dal processo padre prima della invocazione ricorsiva
  - il comando `cd` viene eseguito dal processo figlio all'inizio della propria esecuzione
- Nel primo caso, lo script dovrà contenere un opportuno `cd . .` dopo l'invocazione ricorsiva ma non vi è necessità di passare il nome della directory da esplorare
- Nel secondo caso, non serve il `cd . .` ma il nome della directory da esplorare deve essere trasmesso al processo figlio, ad esempio sulla cmdline

# esempi

```
for filename in *
do
    if [ -d $filename ]
    then
        cd $filename
        $0
        cd ..
    fi
done
```

# esempi

```
cd $1
for filename in *
do
    if [ -d $filename ]
    then
        $0 $filename
    fi
done
```