LabSO 2023

Laboratorio Sistemi Operativi - Unitn A.A. 2022-2023

Michele Grisafi - michele.grisafi@unitn.it

Il corso

Requisiti

- Uso basilare Linux da shell
- Utilizzo editor per codice
- Conoscenza fondamenti file-system (file, cartelle, permessi) e processi
- Teoria di *Sistemi Operativi*

Obiettivo

Dimestichezza nell'ideazione e realizzazione di applicazioni binarie complete o almeno singole componenti impostate con il linguaggio C utilizzabili su shell bash in un sistema Linux debian/ubuntu che implementino metodi di IPC.

Note:

Terminologia e contenuti sono contestualizzati al corso: eventuali approssimazioni o semplificazioni di contenuti e significati sono legati a tale scopo (e quindi i concetti non sono da considerarsi esaustivi in senso assoluto). Alcuni concetti sono solo toccati marginalmente e/o descritti nel momento che si incontrano

Corso

Presentazione contenuti principalmente tramite “slides” commentate direttamente con esposizione concetti.

Slides fornite dopo la lezione.

- Punti fondamentali
- Semplici schematizzazioni / grafici
- Snippet di codice

Possono essere mostrate semplici interazioni tramite screen-sharing.

È richiesto disporre di un sistema che consenta di utilizzare le applicazioni presentate per poter fare prove in modo interattivo.

Esame

- Esercizio di programmazione
- Possibili domande teoriche (relative al lab)
- 3 appelli:
  - 2 in Giugno/Luglio
  - 1 a Gennaio/Febbraio
- **OPPURE** 1 progetto + orale

Lo studente passa il corso se ottiene 18 sia a Teoria che a Laboratorio. La consegna del progetto preclude la partecipazione agli appelli, e viceversa.

Argomenti

- 27/02 - Usare un ambiente linux: **terminal** and **bash**.
- 06/03 - Eseguire e compilare su Linux: **Docker**, **GCC** and **Makefile.**
- 13/03 - Le basi di C.
- 20/03 - Interazione con i file: **streams** and **file descriptors**.
- 27/03 - Comunicazione con il kernel e processi: **system calls e forking.**
- 03/04 - Inter-Process-Communication: **signals**.
- 17/04 - Presentazione progetto.
- 08/05 - IPC: **pipes e fifos**.
- 15/05 - IPC**: message queues.**
- 22/05 - Gestire lavori multipli e la concorrenza: **threads** and **mutexes**.
- 29/05 - Esercitazione

Terminale & bash

Terminale

Il terminale (o *terminal*) è l'ambiente testuale di interazione con il sistema operativo.

Tipicamente è utilizzato come applicazione all'interno dell'ambiente grafico ed è possibile avviarne più istanze, pur essendo anche disponibile direttamente all'avvio (in questo caso normalmente in più istanze accessibili con la combinazione CTRL+ALT+Fx).

Shell

All'interno del terminale, l'interazione avviene utilizzando un'applicazione specifica in esecuzione al suo interno, comunemente detta **SHELL**.

Essa propone un prompt per l'immissione diretta di comandi da tastiera e fornisce un feedback testuale. È anche possibile eseguire sequenze di comandi preorganizzate contenute in file testuali (*script* o *batch*). A seconda della modalità (diretta/script) alcuni comandi possono avere senso o meno o comportarsi in modo particolare.

L'insieme dei comandi e delle regole di composizione costituisce un linguaggio di programmazione orientato allo scripting.

Bash

Esistono numerose shell. Bash è una delle più utilizzate e molte sono comunque simili tra loro, ma hanno sempre qualche differenza (e anche comandi analoghi possono avere opzioni o comportamenti non identici).

Tipicamente - almeno in sessioni non grafiche - al login un utente ha associata una shell particolare.

Un paio di alternative: *zsh*, *ksh*, *sh*, ...

POSIX

Portable **O**perating **S**ystem **I**nterface for Uni**x:** è una famiglia di standard IEEE. Nel caso delle shell in particolare definisce una serie di regole e di comportamenti che possono favorire la portabilità (che però dipende anche da altri fattori del sistema**!**).

La shell *bash* soddisfa molti requisiti POSIX, ma presenta anche alcune differenze ed "estensioni" per agevolare almeno in parte la programmazione. (v. costrutti per confronti logici)

Comandi interattivi

La shell attende un input dall'utente e al completamento (conferma con INVIO) lo elabora.

Per indicare l'attesa mostra all'utente un PROMPT (può essere modificato).

Fondamentalmente si individuano 3 canali:

- Standard input (tipicamente la tastiera), canale 0, detto `stdin`
- Standard output (tipicamente il video), canale 1, detto `stdout`
- Standard error (tipicamente il video), canale 2, detto `stderr`

Struttura generale comandi

Solitamente un comando è identificato da una parola chiave cui possono seguire uno o più “argomenti” opzionali o obbligatori, accompagnati da un valore di riferimento o meno (in questo caso hanno valore di “flag”) e di tipo posizionale o nominale. A volte sono ripetibili.

```
ls -alh /tmp
```

gli argomenti nominali sono indicati con un trattino cui segue una voce (stringa alfanumerica) e talvolta presentano una doppia modalità di riferimento: breve (tipicamente voce di un singolo carattere) e lunga (tipicamente un termine mnemonico)

`app -h`	`app --help`

Termini

- l’esecuzione dei comandi avviene “per riga” (in modo diretto quella che si immette fino a INVIO) (*)
- un “termine” (istruzione, argomento, opzione, etc.) è solitamente una stringa alfanumerica senza spazi
- spaziature multiple sono solitamente valide come singole e non sono significative se non per separare termini
- è possibile solitamente usare gli apici singoli o doppi per forzare una sequenza come termine singolo
- gli spazi iniziali e finali di una riga collassano
- le righe vuote sono ignorate

(*) per i file batch (gli “script”) l’argomento sarà trattato in modo approfondito in un blocco successivo

Commenti

È anche possibile utilizzare dei **commenti** da passare alla shell. L'unico modo formale è l'utilizzo del carattere '**#**' per cui esso e tutto ciò che segue fino al termine della riga è considerato un commento ed è sostanzialmente ignorato.

```
ls -la #-r this part is ignored
```

Alcuni comandi fondamentali

I comandi possono essere “builtins” (funzionalità intrinseche dell’applicazione shell utilizzata) o “esterni” (applicazioni eseguibili che risiedono su disco).

- clear
- pwd
- ls
- cd
- wc
- date
- cat
- echo
- alias/unalias
- test
- read
- file
- chown
- chmod
- cp/mv
- **help**
- type
- grep
- function

Canali in/out

Ogni comando lavora su un insieme di canali, come standard output o un file descriptor*.

```
ls # mostra output della cartella corrente

ls not-existent-item #mostra un messaggio d'errore
```

(*) File Descriptors: l'argomento sarà trattato in modo approfondito in un blocco successivo

Redirezionamento di base

I canali possono essere redirezionati (anche in cascata):

- `ls **1>**/tmp/out.txt **2>**/tmp/err.txt`
- `ls not-existent-item **1>**/tmp/all.txt **2>&1**`

Redirezionamento di base

- **'<'**: `command<file.txt` invia l'input al comando (`file.txt` read-only): `mail -s "Subject" rcpt < content.txt` (anziché interattivo)

- **'<>'**: come sopra ma `file.txt` è aperto in read-write (raramente usato)

- *source*`>`*target*: `command 1>out.txt 2>err.txt`
  redireziona *source* su *target*:
  - *source* può essere sottinteso (vale `1`)
  - *target* può essere un canale (si indica con `&`*n*, ad esempio `&2`)

- **'>|'** : si comporta come > ma forza la sovrascrittura anche se bloccata nelle configurazioni

Redirezionamento di base

- >> : si comporta come > ma opera un *append* se la destinazione esiste
- <<: here-document, consente all'utente di specificare un terminatore testuale, dopodiche accetta l'input fino alla ricezione di tale terminatore.
- <<< : here-string, consente di fornire input in maniera non interattiva.

Esistono molte varianti e possibilità di combinazione dei vari operatori di redirezionamento.

Un caso molto utilizzato è la "soppressione" dell'output (utile per gestire solo side-effects, come ad esempio il codice di ritorno), ad esempio:
`type` *`command`* `1>/dev/null 2>&1` (per sapere se *`command`* esiste)

Ambiente e variabili

- La shell può utilizzare delle variabili per memorizzare e recuperare valori
- I valori sono generalmente trattati come stringhe o interi: sono presenti anche semplici *array* (vettori di elementi)
- Il formato del nome è del tipo `^[_[:alpha:]][_[:alnum:]]*$`
- Per set (impostare) / get (accedere) al valore di una variabile:
  - Il set si effettua con `[export] variabile=valore` (*)
  - Il get si effettua con `$variabile` o `${variabile}` (sostituzione letterale)

(*) lo “scope” è generalmente quello del processo attuale: anteponendo “export” si rende disponibile anche agli eventuali processi figli

Ambiente e variabili

- La shell opera in un ambiente in cui ci sono alcuni riferimenti impostabili e utilizzabili attraverso l'uso delle cosiddette "variabili d'ambiente" con cui si intendono generalmente quelle con un significato particolare per la shell stessa (v. esempi di seguito)

- Tra le variabili d'ambiente più comuni troviamo ad esempio:
  `SHELL`, `PATH`, `TERM`, `PWD`, `PS1`, `HOME`

- Essendo variabili a tutti gli effetti si impostano ed usano come le altre.

Variabili di sistema

Alcune variabili sono impostate e/o utilizzate direttamente dal sistema in casi particolari. Se ne vedranno alcune caratteristiche degli script/batch, ma intanto in modalità diretta se ne usano già diverse:

- `SHELL` : contiene il riferimento alla shell corrente (path completo)
- `PATH` : contiene i percorsi in ordine di priorità in cui sono cercati i comandi, separati da “:”
- `TERM` : contiene il tipo di terminale corrente
- `PWD` : contiene la cartella corrente
- `PS1` : contiene il prompt e si possono usare marcatori speciali
- `HOME` : contiene la cartella principale dell’utente corrente

Esecuzione comandi e `$PATH`

- Quando si immette un comando (o una sequenza di comandi) la shell analizza quanto inserito (*parsing*) e individua le parti che hanno la posizione di comandi da eseguire: se sono interni ne esegue le funzionalità direttamente altrimenti cerca di individuare un corrispondente file eseguibile: questo è normalmente cercato nel file-system solo e soltanto nei percorsi definiti dalla variabile `PATH` a meno che non sia specificato un percorso (relativo o assoluto) nel qual caso viene utilizzato esso direttamente.
  Dall'ultima osservazione discende che per un'azione abbastanza comune come lanciare un file eseguibile (non "installato") nella cartella corrente occorre qualcosa come: `./nomefile`

Array

- Definizione: `lista=("a" 1 "b" 2 "c" 3)` **separati da spazi!**
- Output completo: `${lista[@]}`
- Accesso singolo: `${lista[x]}` (0-based)
- Lista indici: `${!lista[@]}`
- Dimensione: `${#lista[@]}`
- Set elemento: `lista[x]=value`
- Append: `lista+=(value)`
- Sub array: `${lista[@]:s:n}` (from index s, length n)

Variabili $$ e $?

Le variabili `$$` e `$?` non possono essere impostate manualmente (la stessa sintassi lo impedisce dato che i nomi sarebbero `$` e `?` non utilizzabili normalmente):

- `$$` : contiene il PID del processo attuale (*)
- `$?` : contiene il codice di ritorno dell'ultimo comando eseguito

(*) sarà approfondito il concetto di PID in sezioni successive

Esecuzione comandi e parsing

- La riga dei comandi è elaborata con una serie di azioni “in sequenza” e poi rielaborata eventualmente più volte.

- Tra le azioni vi sono:
  - Sostituzioni speciali della shell (es. “!” per accedere alla “history” dei comandi)
  - Sostituzione variabili
  - Elaborazione subshell

- Sono svolte con un ordine di priorità e poi l’intera riga è rielaborata (v. Subshell)

Concatenazione comandi

È possibile concatenare più comandi in un'unica riga in vari modi con effetti differenti:

- `comando1 ; comando2` concatenazione semplice: esecuzione in sequenza
- `comando1 && comando2` concatenazione logica "and": l'esecuzione procede solo se il comando precedente non fallisce (codice ritorno zero)
- `comando1 || comando2` concatenazione logica "or": l'esecuzione procede solo se il comando precedente fallisce (codice ritorno NON zero)
- `comando1 | comando2` concatenazione con piping (v. prox)

Operatori di piping (“pipe”): | e |&

La concatenazione con gli operatori di piping “cattura” l’output di un comando e lo passa in input al successivo:

- `ls | wc -l` : cattura solo stdout
- `ls |& wc -l` : cattura stdout e stderr

NOTA. il comando ls ha un comportamento atipico: il suo output di base è differente a seconda che il comando sia diretto al terminale o a un piping (*)

(*) internamente sfrutta `isatty(STDOUT_FILENO)` (si approfondirà in seguito)

Subshell

- È possibile avviare una subshell, ossia un sotto-ambiente in vari modi, in particolare raggruppando i comandi tra parentesi tonde:
  `( …comandi… )`

- Spesso si usa “catturare” l’output standard (stdout) della sequenza che viene così sostituito letteralmente e rielaborato e si può fare in due modi:

  `$( …comandi… )` oppure `` ` …comandi… ` ``

Nota: attenzione al fatto che le parentesi tonde sono utilizzate anche per definire *array*

Esempio alcuni comandi e sostituzione subshell

```
echo "/tmp" > /tmp/tmp.txt ; ls $(cat /tmp/tmp.txt)
```

i comandi sono eseguiti rispettando la sequenza:

- `echo "/tmp" > /tmp/tmp.txt` crea un file temporaneo con `"/tmp"`
- `ls $(cat /tmp/tmp.txt)` è prima eseguita la subshell:
  - `cat /tmp/tmp.txt` genera in stdout `"/tmp"` e poi con sostituzione:
  - `ls /tmp` mostra il contenuto della cartella `/tmp`

Espansione aritmetica

- La sintassi base per una subshell è da non confondere con l'espansione aritmetica che utilizza le doppie parentesi tonde.

- All'interno delle doppie parentesi tonde si possono rappresentare varie espressioni matematiche inclusi assegnamenti e confronti.

Alcuni esempi:

```
(( a = 7 )) (( a++ )) (( a < 10 )) (( a = 3<10?1:0 ))
```

```
b=$((c+a))
```

Confronti logici - costrutti

I costrutti fondamentali per i confronti logici sono il comando `test` e i raggruppamenti tra parentesi quadre singole e doppie: `test … , [ … ], [[ … ]]`

- `test …` e `[ … ]` sono *built-in* equivalenti
- `[[ … ]]` è una coppia di *shell-keywords*

In tutti i casi il blocco di confronto genera il codice di uscita 0 in caso di successo, un valore differente (tipicamente 1) altrimenti.

NOTA. *built-in* e *shell-keywords*: i *builtins* sono sostanzialmente dei comandi il cui corpo d'esecuzione è incluso nell'applicazione shell direttamente (non sono eseguibili esterni) e quindi seguono sostanzialmente le "regole generali" dei comandi, mentre le *shell-keywords* sono gestite come marcatori speciali così che possono "attivare" regole particolari di parsing. Un caso esemplificativo sono gli operatori "<" e ">" che normalmente valgono come redirezionamento, ma all'interno di `[[ … ]]` valgono come operatori relazionali.

Confronti logici - tipologia operatori

Le parentesi quadre singole sono POSIX-compliant, mentre le doppie sono un'estensione bash. Nel primo caso gli operatori relazionali "tradizionali" (*minore-di*, *maggiore-di*, etc.) non possono usare i termini comuni (<, >, etc.) perché hanno un altro significato (*) e quindi se ne usano di specifici che però hanno un equivalente più "tradizionale" nel secondo caso.

Gli operatori e la sintassi variano a seconda del tipo di informazioni utilizzate: una distinzione sottile c'è per confronti tra stringhe e confronti tra interi.

(*) salvo eventualmente utilizzare il raggruppamento con doppie parentesi tonde per le espansioni aritmetiche

Confronti logici - interi e stringhe

interi		
	`[ … ]`	`[[ … ]]`
uguale-a	`-eq`	`==`
diverso-da	`-ne`	`!=`
minore-di minore-o-uguale-a	`-lt` `-le`	`<` `<=`
maggiore-di maggiore-o-uguale-a	`-gt` `-ge`	`>` `>=`

stringhe		
	`[ … ]`	`[[ … ]]`
uguale-a	= o ==	
diverso-da	!=	
minore-di (ordine alfabetico)	`\<`	`<`
maggiore-di (ordine alfabetico)	`\>`	`>`
nota: occorre lasciare uno spazio prima e dopo i “simboli” (es. non “=” ma “ = “)		

Confronti logici - operatori unari

Esistono alcuni operatori unari ad esempio per verificare se una stringa è vuota o meno oppure per controllare l'esistenza di un file o di una cartella.

Alcuni esempi:

`[[ -f /tmp/prova ]]` : è un file?

`[[ -e /tmp/prova ]]` : file esiste?

`[[ -d /tmp/prova ]]` : è una cartella?

NB: sono essenziali gli spazi dopo **[[** e prima di **]]**. È possibile usare sia **[** che **[[**

Confronti logici - negazione

Il carattere “!” (punto esclamativo) può essere usato per negare il confronto seguente.

Alcuni esempi:

```
[[ ! -f /tmp/prova ]]

[[ ! -e /tmp/prova ]]

[[ ! -d /tmp/prova ]]
```

ESERCIZI - 1

Scrivere delle sequenze di comandi (singola riga da eseguire tutta in blocco) che utilizzano come “input” il valore della variabile DATA per:

1. Stampa “`T`” (per True) o “`F`” (per False) a seconda che il valore rappresenti un file o cartella esistente
2. Stampa “`file`”, “`cartella`” o “`?`” a seconda che il valore rappresenti un file (esistente), una cartella (esistente) o una voce non presente nel file-system
3. Stampa il risultato di una semplice operazione aritmetica (es: ‘$1 < 2$’) contenuta nel file indicato dal valore di DATA, oppure “`?`” se il file non esiste

Soluzione

```
1. [ -e $DATA ] && echo "T" || echo "F"
2. [ -f $DATA ] && echo "file" || ( [ -d $DATA ] && echo "cartella" || echo "?" )
3. [ -f $DATA ] && echo $(( $(cat $DATA) )) || echo "?"
```

NB: se eseguiti all'interno di uno script, la variabile DATA deve essere esportata con `export DATA=valore`

SCRIPT/BATCH

È possibile raccogliere sequenze di comandi in un file di testo che può poi essere eseguito:

- Richiamando il tool “bash” e passando il file come argomento
  - Esempio: `bash ../file.sh`
- Impostando il bit “x” e specificando il percorso completo, o solo il nome se la cartella è in `$PATH`:
  - Esempio: `chmod +x ../file.sh && ../file.sh`

Esempio 1 - subshell e PID

SCRIPT “bashpid.sh”:

```
# bashpid.sh
echo $BASHPID #the id of the current bash process
echo $( echo  $BASHPID)
```

CLI:

```
chmod +x ./bashpid.sh ; echo $BASHPID ; ./bashpid.sh
```

Elementi particolari negli SCRIPT

Le righe vuote e i commenti (#) sono ignorati.

La prima riga può essere un metacommento (detto hash-bang, she-bang e altri nomi simili): `#!application [opts]` che identifica un'applicazione cui passare il file stesso come argomento (tipicamente usato per identificare l'interprete da utilizzare)

Sono disponibili variabili speciali in particolare

- `$@` : lista completa degli argomenti passati allo script
- `$#` : numero di argomenti passati allo script
- `$0, $1, $2, …` : n-th argomento

Alri costrutti

For loop:

```
for i in ${!lista[@]}; do
    echo ${lista[$i]}
done
```

While loop:

```
while [[ $i < 10 ]]; do
    echo $i ; (( i++ ))
done
```

If condition:

```
if [ $1 -lt 10 ]; then
    echo less than 10
elif [ $1 -gt 20 ]; then
    echo greater than 20
else
    echo between 10 and 20
fi
```

Questi costrutti possono essere anche scritti su una sola riga e usati in un terminale.

Esempio 2 - argomenti

SCRIPT “args.sh”:

```
#!/usr/bin/env bash
nargs=$#
while [[ $1 != "" ]]; do
    echo "ARG=$1"
    shift
done
```

CLI:

```
chmod +x ./args.sh
./args.sh uno
./args.sh uno due tre
```

Funzioni

```
function_name () {
    local var1='C' #use a local scoped variable
    echo $1 $2 #normal arguments variables
    return 44 #only return codes
}

function function_name () { commands; }

function_name arg1 arg2  #call the function with 2 args
```

ESERCIZI - 2

- Scrivere uno script che dato un qualunque numero di argomenti li restituisca in output in ordine inverso.
- Scrivere uno script che mostri il contenuto della cartella corrente in ordine inverso rispetto all'output generato da "`ls`" (che si può usare ma senza opzioni). Per semplicità, assumere che tutti i file e le cartelle non abbiano spazi nel nome.

CONCLUSIONI

L’utilizzo di BASH - tramite CLI o con SCRIPT - è basilare per poter interagire attraverso comandi con il file-system, con le risorse del sistema e per poter invocare tools e applicazioni.