LabSO 2022

Laboratorio Sistemi Operativi - A.A. 2021-2022

dr. Andrea Naimoli	Informatica LT andrea.naimoli@unitn.it
dr. Michele Grisafi	Ingegneria informatica, delle comunicazioni ed elettronica (LT) michele.grisafi@unitn.it

Presentazione

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

Elementi

- terminal & bash
- docker
- gcc
- make
- linguaggio C:
  - fondamenti
  - costrutti
  - system-calls
  - IPC

Corso

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

- Punti fondamentali
- Semplici schematizzazioni / grafici
- Esemplificazioni di schermate (screenshot)

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

Possibili quesiti sulla teoria di laboratorio ed esercizi di programmazione in C e bash.

Voto che fa media al 50% con la teoria. Voto minimo 18 per poter passare il corso.

Terminal & bash

Terminal

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).

L'interazione avviene utilizzando un'applicazione specifica in esecuzione al suo interno comunemente detta SHELL.

Bash

SHELL di riferimento per l'interazione con il sistema.

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.

Una shell può essere o meno di login e può essere o meno interattiva

SHELL	login no	login sì
interattiva no	non-login non-interactive	login non-interactive
interattiva sì	non-login interactive	login interactive

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, 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:

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

(il numero del canale è quello standard se non si varia)

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.

Alcuni comandi fondamentali

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

Canali in/out

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

Ogni comando lavora su un insieme di canali (*) e in particolare uno per l’output standard e uno per l’output degli errori (normalmente entrambi vanno a video)

```
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
- `<< <<<` : permettono di utilizzare testi come here-doc

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)`
- 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 ))
```

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?

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

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
- Impostando il bit “x” e:
  - specificando il path completo (relativo o assoluto)
  - Indicando il solo nome se il percorso è presente in $PATH

Esempio 1 - subshell e PID

SCRIPT “bashpid.sh”:

```
# bashpid.sh
echo $BASHPID
echo $( echo  $BASHPID)
```

CLI:

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

Elementi particolari negli SCRIPT

Le righe vuote e i commenti sono ignorati.

I commenti sono porzioni di righe che cominciano con “#” (non in una stringa)

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 $@, $# e $0, $1, $2, ...

Alri construtti

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 10
else
    echo between 10 and 20
fi
```

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
```

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)

CONCLUSIONI

L’utilizzo di BASH - tramite CLI o con SCRIPT - è basilare per poter interagire attraverso comandi per l’uso del file-system e delle altre risorse e per poter invocare tools e applicazioni.

Esistono (numerose) alternative ed è possibile anche sfruttare più strumenti in cooperazione (ad esempio scripts con interpreti differenti).

LabSO 2022

Laboratorio Sistemi Operativi - A.A. 2021-2022

dr. Andrea Naimoli	Informatica LT andrea.naimoli@unitn.it
dr. Michele Grisafi	Ingegneria informatica, delle comunicazioni ed elettronica (LT) michele.grisafi@unitn.it

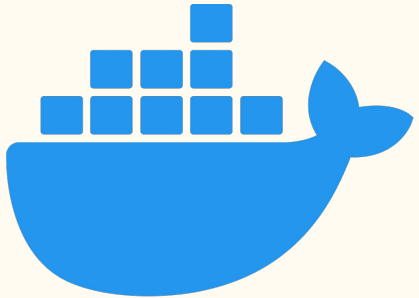
docker

Docker, cos'è?

Tecnologia di virtualizzazione a livello del sistema operativo che consente la creazione, la gestione e l'esecuzione di **applicazioni** attraverso containers.

I **containers** sono ambienti leggeri, dinamici ed isolati che vengono eseguiti sopra il kernel di Linux.

... e molti altri!



docker

Docker Container

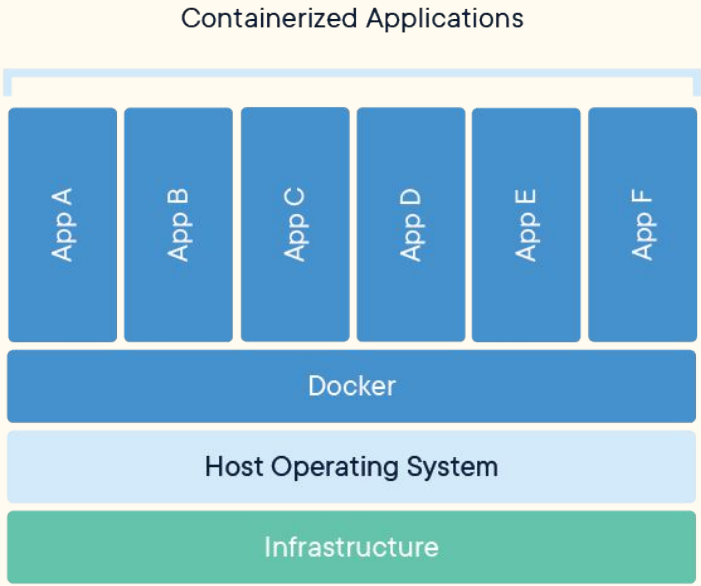
- Virtualizzazione a livello OS
- Containers condividono kernel
- Avvio e creazione in secondi
- Leggere (KB/MB)
- Si distruggono e si rieseguono
- Utilizzo leggero di risorse
- Minore sicurezza

NB: Basati su immagini delle quali se ne trovano tantissime già pronte!

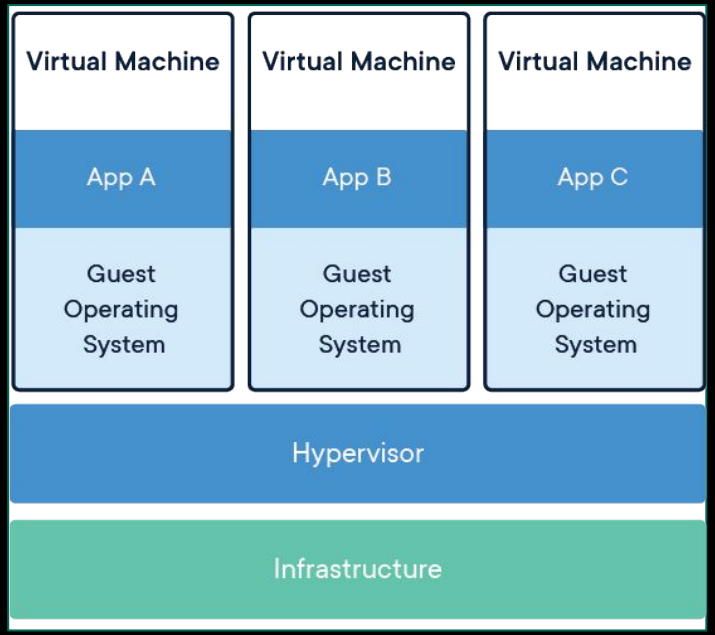
Virtual Machine

- Virtualizzazione a livello HW
- Ogni VM ha il suo OS
- Avvio e creazione in minuti
- Pesanti (GB)
- Si trasferiscono
- Utilizzo intenso di risorse
- Maggiore sicurezza
- Maggiore controllo

Docker Container

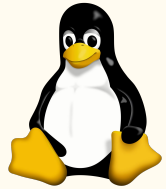


Virtual Machine

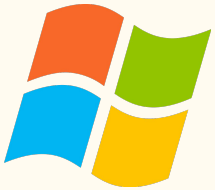


Compatibilità sui vari OS

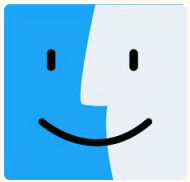
Linux: docker gestisce i containers usando il kernel linux nativo.



Windows: docker gestisce i containers usando il kernel linux tramite WSL2 (originariamente virtualizzato tramite Hyper-V). Gestito da un'applicazione.



Mac: docker gestisce i containers usando il kernel linux virtualizzato tramite xhyve hypervisor. Gestito da un'applicazione.



Containers e immagini

Un’immagine docker è un insieme di “istruzioni” per la creazione di un container. Essa consente di raggruppare varie applicazioni ed eseguirle, con una certa configurazione, in maniera rapida attraverso un container.

I containers sono invece gli ambienti virtualizzati gestiti da docker, che possono essere creati, avviati, fermati ed eliminati. I container devono essere basati su un’immagine!

(esiste un’opzione di creazione da zero - FROM scratch - che però è raramente usata in pratica)

Gestione dei containers

`docker run [options] <image>`: crea un nuovo container da un'immagine

`docker container ls [options]`: mostra i containers attivi ([-a] tutti)

`docker start/stop <container>`: avvia/ferma l'esecuzione del container

`docker exec [options] <container> <command>`: esegue il comando all'interno del container

`docker stats`: mostra le statistiche di utilizzo dei containers

E la panacea di tutti i dubbi... `docker <command> --help`

Parametri ‘run’ opzionali

- `--name <nome>`: assegna un nome specifico al container
- `-d`: detach mode → scollega il container (ed il suo input/output) dalla console*
- `-ti`: esegue container in modalità interattiva*
- `--rm`: elimina container all’uscita
- `--hostname <nome>`: imposta l’hostname nel container
- `--workdir <path>`: imposta la cartella di lavoro nel container
- `--network host`: collega il container alla rete locale **
- `--privileged`: esegue il container con i privilegi dell’host

*Per collegarsi `docker attach <container>`. Per scollegarsi *Ctrl+P, Ctrl+Q*

** la modalità host non funziona su W10 e MacOS a causa della VM sottostante

Esempi

- Esegui `docker run hello-world`
- Esegui `docker run -d -p 80:80 docker/getting-started` e collegati alla pagina “localhost:80” con un qualunque browser

Gestione delle immagini

La community di docker offre migliaia di immagini pronte all'uso ma è possibile crearne di nuove.

`docker images`: mostra le immagini salvate

`docker rmi <imageID>`: elimina un'immagine (se non in uso!)

`docker search <keyword>`: cerca un'immagine nella repository di docker

`docker commit <container> <repository/imageName>`: crea una nuova immagine dai cambiamenti nel container

Altrimenti si possono creare nuove immagini con dei dockerfile...

Dockerfile

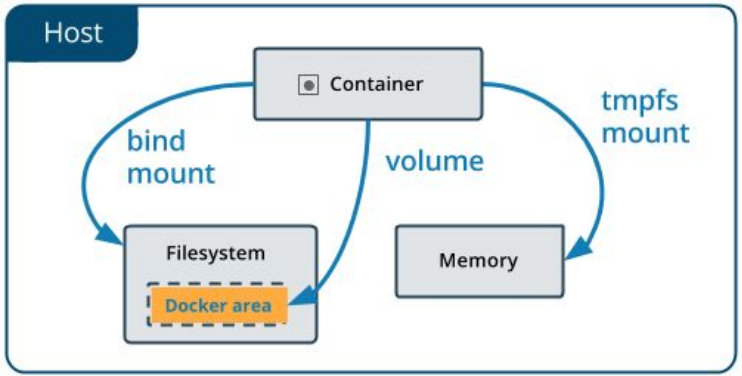
I dockerfile sono dei documenti testuali che raccolgono una serie di comandi necessari alla creazione di una nuova immagine. Ogni nuova immagine sarà generata a partire da un'immagine di base, come Ubuntu o l'immagine minimale 'scratch'. La creazione a partire da un docker file viene gestita attraverso del caching che ne permette la ricompilazione rapida in caso di piccoli cambiamenti.

```
FROM ubuntu:20.04
RUN apt-get update && apt-get install build-essential nano -y
RUN mkdir /home/LabOS
CMD cd /home/labOS && bash
```

```
docker build -t labos/ubuntu - < dockerfile
```

Gestione dei volumi

Docker salva i file persistenti su *bind mount* o su dei *volumi*. Sebbene i **bind mount** siano strettamente collegati con il filesystem dell'host OS, consentendo dunque una facile comunicazione con il containers, i **volumi** sono ormai lo standard in quanto indipendenti, facili da gestire e più in linea con la filosofia di docker.



Sintassi dei comandi

`docker volume create <volumeName>`: crea un nuovo volume

`docker volume ls`: mostra i volumi esistenti

`docker volume inspect <volumeName>`: esamina volume

`docker volume rm <volumeName>`: rimuovi volume

`docker run -v <volume>:</path/in/container> <image>` : crea un nuovo container con il **volume** specificato montato nel percorso specificato

`docker run -v <pathHost>:<path/in/container> <image>` : crea un nuovo container con un **bind mount** specificato montato nel percorso specificato

Il nostro ambiente

```
docker run -ti --rm --name="labOS" --privileged      \
-v /:/host -v "$(pwd):/home/labOS"   \
--hostname "labOS" --workdir /home/labOS    \
ubuntu:20.04 /bin/bash
```

Ed eseguire:

```
apt-get update && apt-get install -y nano build-essential
```

Quando il container è pronto si può fare il commit per salvare le modifiche in una nuova immagine (es.: `docker commit ..... localhost.ext/unitn:labso2021`)

NB: se non aggiungete il flag --rm, ogni volta che uscite il container verrà fermato e potrà essere riavviato con `docker start labOS`

Il nostro ambiente... oppure

Usare il dockerfile in slide #12 per creare un'immagine del laboratorio:

```
docker build -t labOS/ubuntu - < dockerfile
```

E poi è possibile eseguire un container basato sulla nuova immagine 'labOS/ubuntu': usare il comando

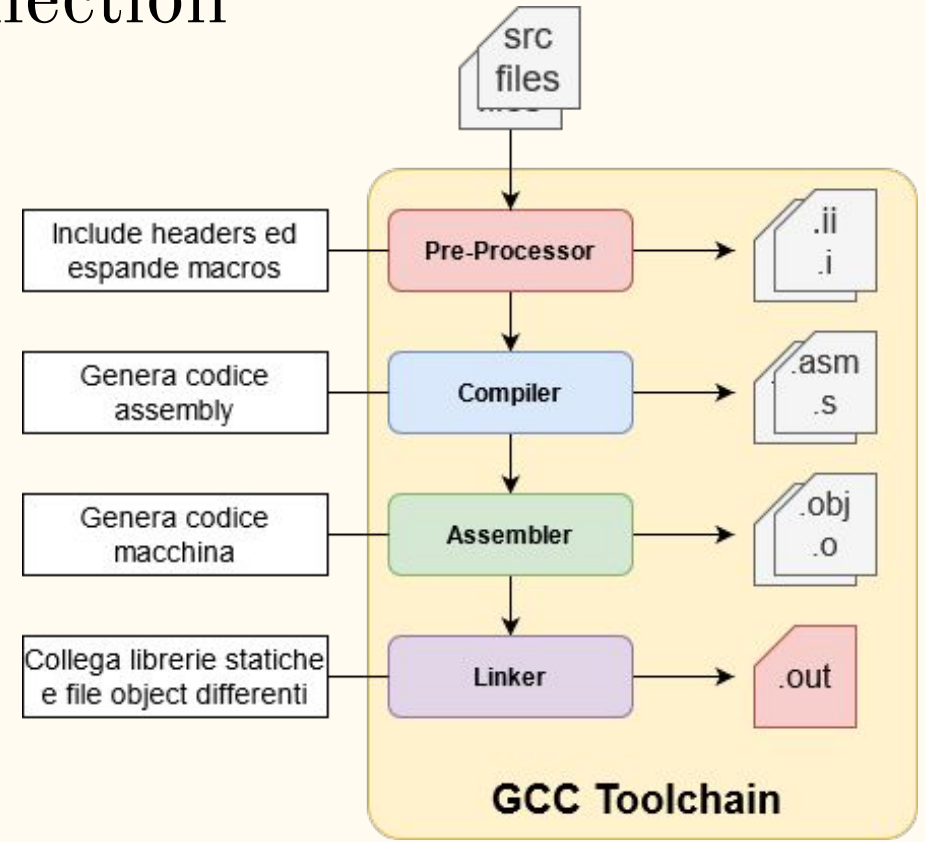
```
docker run -ti --rm --name="labOS" --privileged      \
-v /:/host    -v "$(pwd):/home/labOS"    \
--hostname "labOS" labos/ubuntu
```

gcc

GCC = Gnu Compiler Collection

Insieme di strumenti open-source che costituisce lo standard per la creazione di eseguibili su Linux.

GCC supporta diversi linguaggi, tra cui C, e consente la modifica dei vari passaggi intermedi per una completa personalizzazione dell'eseguibile.



La compilazione

Gli strumenti GCC possono essere chiamati singolarmente:

```
gcc -E <sorgente.c> -o <preProcessed.ii|.i>
```

```
gcc -S <preProcessed.i|.ii> -o <assembly.asm|.s>
```

```
gcc -c <assembly.asm|.s> -o <objectFile.obj|.o>
```

```
gcc <objectFile.obj|.o> -o <executable.out>
```

NB: l'input di ogni comando può essere il file sorgente, e l'ultimo comando è in grado di creare direttamente l'eseguibile.
NB: l'assembly ed il codice macchina generato dipendono dall'architettura di destinazione

Un esempio

1. Provate a compilare una semplicissima applicazione, invocando ogni step singolarmente osservandone l'output.
2. Provate ad aggiungere `#include <stdio.h>` ad inizio file e ripetete il tutto

```c
//main.c
void main(){
    return;
}
```

```c
//main.c
#include <stdio.h>
void main(){
    return;
}
```

make

Make tool

Il Make tool è uno strumento della collezione GNU che può essere usato per gestire la compilazione *automatica* e *selettiva* di grandi e piccoli progetti. *Make* consente di specificare delle dipendenze tra i vari file, per esempio consentendo solo la compilazione di librerie i cui sorgenti sono stati modificati.

Make può anche essere usato per gestire il deployment di un'applicazione, assumendo alcune delle capacità di uno script bash.

Makefile

Make può eseguire dei makefiles i quali contengono tutte le direttive utili alla compilazione di un'applicazione (o allo svolgimento di un altro task).

```
make -f makefile
```

In alternativa, il comando `make` senza argomenti processerà il file `'makefile'` presente nella cartella di lavoro (nell'ordine cerca: `GNUmakefile`, `makefile` e `Makefile`)

Makefiles secondari possono essere inclusi nel makefile principale con delle direttive specifiche, consentendo una gestione più articolata di grandi progetti.

Target, prerequisite and recipes

Una *ricetta* è una lista di comandi bash che vengono eseguiti indipendentemente dal resto del makefile.
I *target* sono generalmente dei files generati da uno specifico insieme di regole.
Ogni target può specificare dei *prerequisiti*, ovvero degli altri file che devono esistere affinchè le regole di un target vengano eseguite. Un prerequisito può essere esso stesso un target!

L'esecuzione di un file make inizia specificando uno o più target `make -f makefile target1 ...` e prosegue a seconda dei vari prerequisiti.

```
target: prerequisite
→   recipe
→   recipe
    ...
```

```
target1: target2 target3
    rule (3)
    rule (4)
    ...

target2: target3
    rule (1)

target3:
    rule (2)
```

Sintassi

Un makefile è un file di testo “plain” in cui righe vuote e parti di testo dal carattere “**#**” fino alla fine della riga non in una ricetta (considerato un commento: sempre che non sia usato l’escaping con “**\#**” o che compaia dentro una stringa con **'** o **"**) sono ignorati.

Le ricette DEVONO iniziare con un carattere di TAB (**non spazi**).

Una ricetta che (a parte il TAB) inizia con @ non viene visualizzata in output, altrimenti i comandi sono visualizzati e poi eseguiti.

Una riga con un singolo TAB è una ricetta vuota.

Esistono costrutti più complessi per necessità particolari (ad esempio costrutti condizionali)

Target speciali

Il target di default eseguito quando non ne viene passato alcuno è il primo disponibile.

`.INTERMEDIATE` e `.SECONDARY:` hanno come prerequisiti i target “intermedi”. Nel primo caso sono poi rimossi, nel secondo sono mantenuti a fine esecuzione

.PHONY: ha come prerequisiti i target che non corrispondono a dei files o comunque da eseguire “sempre” senza verificare l’eventuale file omonimo.

In un target, % sostituisce qualunque stringa. In un prerequisito corrisponde alla stringa sostituita nel target.

```
target: prerequisite
→   rule
→   rule
    ...
```

```
all: ...
  rule

.SECONDARY: target1 ..

.PHONY: target1 …

%.s: %.c
  #prova.s: prova.c
  #src/h.s: src/h.c
```

Variabili utente e automatiche

Le variabili utente si definiscono con la sintassi `nome:=valore` o `nome=valore` e vengono usate con `$(nome)`. Inoltre, possono essere sovrascritte da riga di comando con `make nome=value`.

Le variabili automatiche possono essere usate all'interno delle regole per riferirsi ad elementi specifici relativi al target corrente.

```
target: pre1 pre2 pre3
  echo $@ is 'target'
  echo $^ is 'pre1 pre2 pre3'
  echo $< is 'pre1'
```

```
ONCE:=hello $(LATER)
EVERY=hello $(LATER)
LATER=world

target1:
  echo $(ONCE) # 'hello'
  echo $(EVERY) # 'hello world'
```

Funzioni speciali

`$(eval …)`: consente di creare nuove regole make dinamiche

`$(shell …)`: cattura l'output di un commando shell

`$(wildcard *)`: restituisce un elenco di file che corrispondono alla stringa specificata.

```
LATER=hello
PWD=$(shell pwd)
OBJ_FILES:=$(wildcard *.o)

target1:
  echo $(LATER) #hello
  $(eval LATER+= world)
  echo $(LATER) #hello world
```

Make file - Esempio

```
all: main.out
     @echo “Application compiled”

%.s: %.c
     gcc -S $< -o $@

%.out: %.s
     mkdir -p build
     gcc $< -o build/$@

clean:
     rm -rf build *.out *.s

.PHONY: clean

.SECONDARY: make.s
```

Esercizio per casa

Creare un makefile con una regola help di default che mostri una nota informativa, una regola backup che crei un backup di una cartella appendendo ".bak" al nome e una restore che ripristini il contenuto originale. Per definire la cartella sorgente passarne il nome come variabile, ad esempio:

```
make -f mf-backup FOLDER=...
```

(la variabile FOLDER è disponibile dentro il makefile)

C

Perchè C?

- Struttura minimale
- Poche parole chiave (con i suoi pro e contro!)
- Unix compliant, alla base di Unix, nato per scrivere Unix
- Organizzato a passi, con sorgente, file intermedi ed eseguibile finale
- Disponibilità di librerie conosciute e standard
- Nessuna struttura di alt(issim)o livello, come classi o altro
- Efficiente perché di basso livello
- Pieno controllo del programma e delle sue risorse

Direttive

Il compilatore, nella fase di preprocessing, elabora tutte le direttive presenti nel sorgente. Ogni direttiva viene introdotta con '#' e può essere di vari tipi:

- `#include <lib>`: copia il contenuto del file *lib* (cercando nelle cartelle delle librerie) nel file corrente
- `#include "lib"`: come sopra ma cerca prima anche nella cartella corrente
- `#define VAR VAL`: crea una costante VAR con il contenuto VAL, e sostituisce ogni occorrenza di VAR con VAL.
- `#define MUL(A,B) A*B`: dichiara una funzione con parametri A e B. Queste funzioni hanno una sintassi limitata!
- `#ifdef, #ifndef, #if, #else, #endif`: rende l'inclusione di parte di codice dipendente da una condizione.

Macro possono essere passate a GCC con `-D NAME=VALUE`

Tipi e Casting

C è un linguaggio debolmente tipizzato che utilizza 8 tipi fondamentali. È possibile fare il casting tra tipi differenti:

```
float a = 3.5;
int b = (int)a;
```

La grandezza delle variabili è **dipendente dall'architettura di riferimento** i valori massimi per ogni tipo cambiano a seconda se la variabile è *signed* o *unsigned*.

- void (0 byte)
- char (1 byte)
- short (2 bytes)
- int (4 bytes)
- float (4 bytes)
- long (8 bytes)
- double (8 bytes)
- long double (8 bytes)

NB: non esiste il tipo boolean, ma viene spesso emulato con un char.

Array e stringhe

C supporta l'uso di stringhe che, tuttavia, corrispondono a degli array di caratteri.

```
int nome[DIM];
long nome[] = {1,2,3,4};
char string[] = “ciao”;
char string2[] = {‘c’,’i’,’a’,’o’};
nome[0] = 22;
```

Gli array sono generalmente di dimensione statica e non possono essere ingranditi durante l'esecuzione del programma. Per array dinamici dovranno essere usati costrutti particolari (come malloc).

Le stringhe, quando acquisite in input o dichiarate con la sintassi `"stringa"`, terminano con il carattere '`\0`' e sono dunque di grandezza *str_len+1*

Puntatori di variabili

C si evolve attorno all'uso di puntatori, ovvero degli alias per zone di memorie condivise tra diverse variabili/funzioni. L'uso di puntatori è abilitato da due operatori: '*' ed '&'.

'*' ha significati diversi a seconda se usato in una dichiarazione o in un'assegnazione:

`int *punt;` Crea un puntatore ad intero

`int valore = *(punt);` Ottiene valore puntato

'&' ottiene l'indirizzo di memoria in cui è collocata una certa variabile.

`long whereIsValore = &valore;`

Esempi:

```
float   pie   =   3.4;
float  *pPie  =  &pie;
pie       *=       2;
float pie4 = *pPie * 2;

char  *array  =  "str";
*array     =      's';
array[1]   =      't';
*(array + 2) = 'r';
```

Puntatori di variabili

```
int      i      =      42
int   *   punt   =   &i;
int b = *(punt);
```

Tipo	Nome	Valore	Indirizzo
int	i	42	0xaaaabbbb
int *	punt	0xaaaabbbb	0xccccdddd
int	b	42	0x11112222

```
i = 20;
```

Nome	Valore
i	**20**
punt	0xaaaabbbb
b	**?**

Puntatori di funzioni

```
#include <stdio.h>
float xdiv(float a, float b) {
  return a/b;
}
float xmul(float a, float b) {
  return a*b;
}
void main() {
  float (*punt)(float,float);
  punt = xdiv;
  float res = punt(10,10);
  punt = &xmul;
  res = (*punt)(10,10);
  printf("%f\n", res);
}
```

Main.c

```c
#include <stdlib.h>
#include <stdio.h>
#define DIVIDENDO 3

int division(int var1, int var2, int * result){
    *result = var1/var2;
    return 0;
}

void main(int argc, char * argv[]){
    float var1 = atof(argv[1]);
    float result = 0;
    division((int)var1,DIVIDENDO,(int *)&result);
    printf("%d \n",(int)result);
}
```

CONCLUSIONI

Docker, GCC e make possono essere utilizzati per la gestione delle varie applicazioni in C. Ognuno di questi strumenti non è indispensabile ma permette di creare un flusso di lavoro coerente e strutturato.

LabSO 2022

Laboratorio Sistemi Operativi - A.A. 2021-2022

dr. Andrea Naimoli	Informatica LT andrea.naimoli@unitn.it
dr. Michele Grisafi	Ingegneria informatica, delle comunicazioni ed elettronica (LT) michele.grisafi@unitn.it

C - introduzione

Perchè C?

- Struttura minimale
- Poche parole chiave (con i suoi pro e contro!)
- Unix compliant, alla base di Unix, nato per scrivere Unix
- Organizzato a passi, con sorgente, file intermedi ed eseguibile finale
- Disponibilità di librerie conosciute e standard
- Efficiente perché di basso livello
- Pieno controllo del programma e delle sue risorse
- Ottimo per interagire con il sistema operativo

Direttive e istruzioni fondamentali

- `#include … / #define … /`
- `char / int / … / enum (v. esempio seguente)`
- `for ( initialization ; test; increment ) { … ; }`
- `break / continue`
- `switch (expression){ case val: … [break;] [default: …] }`
- `while (expression) { … } / do { … } while (expression)`
- `if (expression) { … } [else { … }]`
- `struct / union`

(consultare una documentazione standard ed esercitarsi)

Tipi e Casting

C è un linguaggio debolmente tipizzato che utilizza 8 tipi fondamentali. È possibile fare il casting tra tipi differenti:

```
float a = 3.5;
int b = (int)a;
```

La grandezza delle variabili è **dipendente dall' architettura di riferimento** i valori massimi per ogni tipo cambiano a seconda se la variabile è *signed* o *unsigned*.

- void (0 byte)
- char (1 byte)
- short (2 bytes)
- int (4 bytes)
- float (4 bytes)
- long (8 bytes)
- double (8 bytes)
- long double (8 bytes)

NB: non esiste il tipo boolean, ma viene spesso emulato con un char.

sizeof *(operatore)*

`sizeof (type) / sizeof expression`

Si tratta di un operatore che elabora il tipo passato come argomento (tra parentesi) o quello dell'espressione e restituisce il numero di bytes occupati in memoria.

```c
#include <stdio.h>
void main() {
  int x = 10;
  printf("variable x      : %lu\n", sizeof x);
  printf("expression 1/2  : %lu\n", sizeof 1/2);
  printf("int type        : %lu\n", sizeof(int));
  printf("char type       : %lu\n", sizeof(char));
  printf("float type      : %lu\n", sizeof(float));
  printf("double type     : %lu\n", sizeof(double));
}
```

Puntatori di variabili

C si evolve attorno all'uso di puntatori, ovvero degli alias per zone di memorie condivise tra diverse variabili/funzioni. L'uso di puntatori è abilitato da due operatori: '*' ed '&'.

'*' ha significati diversi a seconda se usato in una dichiarazione o in un'assegnazione:

`int *punt;` → crea un puntatore ad intero

`int valore = *(punt);` → ottiene valore puntato

'&' ottiene l'indirizzo di memoria in cui è collocata una certa variabile.

`long whereIsValore = &valore;`

Esempi:

```
float   pie   =   3.4;
float  *pPie  =  &pie;
pie       *=       2;
float pie4 = *pPie * 2;

char  *array  =  "str";
*array     =      's';
array[1]   =      't';
*(array + 2) = 'r';
```

Puntatori di variabili

```
int      i      =      42
int   *   punt   =   &i;
int b = *(punt);
```

Tipo	Nome	Valore	Indirizzo
int	i	42	0xaaaabbbb
int *	punt	0xaaaabbbb	0xccccdddd
int	b	42	0x11112222

```
i = 20;
```

Nome	Valore
i	**20**
punt	0xaaaabbbb
b	**?**

Puntatori di funzioni

```c
#include <stdio.h>
float xdiv(float a, float b) {
  return a/b;
}
float xmul(float a, float b) {
  return a*b;
}
int main() {
  float (*punt)(float,float);
  punt = xdiv;
  float res = punt(10,10);
  punt = &xmul; //& opzionale
  res = (*punt)(10,10);
  printf("%f\n", res);
  return;
}
```

C consente anche di creare dei puntatori a delle funzioni: puntatori che possono contenere l'indirizzo di funzioni differenti.

Sintassi simile ma diversa!

```c
float (*punt)(float,float);
```

```
ret_type (* pntName)(argType,argType,...)
```

main.c

- A parte casi particolari (es. sviluppo moduli per kernel) l'applicazione deve avere una funzione "`main`" che è utilizzata come punto di ingresso.
- Il valore di ritorno è **int**, un intero che rappresenta il codice di uscita dell'applicazione (variabile `$?` in bash) ed è 0 di default se omesso. Può essere usato anche void, ma non è standard.
- Quando la funzione è invocata riceve normalmente in input il numero di argomenti (`int argc`), con incluso il nome dell'eseguibile, e la lista degli argomenti come "vettore di stringhe" (`char * argv[]`) (*)

(*) in C una stringa è in effetti un vettore di caratteri, quindi un vettore di stringhe è un vettore di vettori di caratteri, inoltre i vettori in C sono sostanzialmente puntatori (al primo elemento del vettore) → lista di argomenti spesso indicata con "`char ** argv`"
Utile riferimento generale: `https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html`

main.c

```c
#include <stdlib.h>
#include <stdio.h>
#define DIVIDENDO 3

int division(int var1, int var2, int * result){
    *result = var1/var2;
    return 0;
}

int main(int argc, char * argv[]){
    float var1 = atof(argv[1]);
    float result = 0;
    division((int)var1,DIVIDENDO,(int *)&result);
    printf("%d \n",(int)result);
    return;
}
```

Esecuzione - esempio

Compilazione:

```
gcc main.c -o main
```

Esecuzione:

```
./main arg1 arg2
```

```c
#include <stdio.h>

int main(int argc, char **argv) {
  printf("%d\n", argc);
  printf("%s\n", argv[0]);
  return 0;
}
```

In output si ha “3” (numero argomenti incluso il file eseguito) e “./main” (primo degli argomenti).

In generale quindi `argc` è sempre maggiore di zero.

Direttive

Il compilatore, nella fase di preprocessing, elabora tutte le direttive presenti nel sorgente. Ogni direttiva viene introdotta con '#' e può essere di vari tipi:

`#include <`*`lib`*`>`	copia il contenuto del file *lib* (cercando nelle cartelle delle librerie) nel file corrente
`#include "`*`lib`*`"`	come sopra ma cerca prima nella cartella corrente
`#define `*`VAR VAL`	crea una costante VAR con il contenuto VAL, e sostituisce ogni occorrenza di VAR con VAL
`#define MUL(A,B) A*B`	dichiara una funzione con parametri A e B. Queste funzioni hanno una sintassi limitata!
`#ifdef, #ifndef, #if, #else, #endif`	rende l'inclusione di parte di codice dipendente da una condizione.

Macro possono essere passate a GCC con `-D NAME=VALUE`

Librerie standard

Librerie possono essere usate attraverso la direttiva `#include`.
Tra le più importanti vi sono:

- stdio.h: FILE, EOF, stderr, stdin, stdout, fclose(), etc...
- stdlib.h: atof(), atoi(), malloc(), free(), exit(), system(), rand(), etc...
- string.h: memset(), memcpy(), strncat(), strcmp(), strlen(), etc...
- math.h: sin(), cos(), sqrt(), floor(), etc...
- unistd.h: STDOUT_FILENO, read(), write(), fork(), pipe(), etc...
- fcntl.h: creat(), open(), etc...

...e ce ne sono molte altre.

Direttive - esempi

```c
#include <stdio.h>
#define ITER 5
#define POW(A) A*A

int main(int argc, char **argv) {
#ifdef DEBUG
    printf("%d\n", argc);
    printf("%s\n", argv[0]);
#endif
    int res = 1;
    for (int i = 0; i < ITER; i++){
        res *= POW(argc);
    }
    return res;
}
```

```
gcc main.c -o main.out -D DEBUG=0
gcc main.c -o main.out -D DEBUG=1
```

```
./main.out 1 2 3 4
```

Stesso risultato!

Structs e Unions

Structs permettono di aggregare diverse variabili, mentre le unions permettono di creare dei tipi generici che possono ospitare uno di vari tipi specificati.

```
struct Books{
    char author[50];
    char title[50];
    int bookID;
} book1, book2;

struct Books book3 =
{“Rowling”,”Harry Potter”,2};
strcpy(book1.title,“Moby Dick”);
book2.bookID = 3;
```

```
union Result{
    int intero;
    float decimale;
} result1, result2;

union Result result3;
result3.intero = 22;
result3.decimale = 11.5;
```

Typedef

Typedef consente la definizione di nuovi tipi di variabili o funzioni.

```
typedef unsigned int intero;

typedef struct Books{
    …
} bookType;

intero var = 22; //= unsigned int var = 22;
bookType book1; //= struct Books book1;
```

C - esempio “enum”

```
#include <stdio.h>

enum State {Undef = 9, Working = 1, Failed = 0};
void main() {
    enum State state=Undef;
    printf("%d\n", state); // output è “9”
}
```

C - vettori e stringhe

C - vettori I

I vettori sono sequenze di elementi omogenei (tipicamente liste di dati dello stesso tipo, ad esempio liste di interi o di caratteri).

I vettori si realizzano con un puntatore al primo elemento della lista.

Ad esempio con `int arr[4] = {2, 0, 2, 1}` si dichiara un vettore di 4 interi inizializzandolo: sono riservate 4 aree di memoria consecutive di dimensione pari a quella richiesta per ogni singolo intero (tipicamente 2 bytes, quindi 4*2=8 in tutto)

C - vettori II

`char str[7] = {‘c’, ‘i’, ‘a’, ‘o’, 56,57,0}` : 7*1 = 7 bytes

`str` è dunque un puntatore a `char` (al primo elemento) e si ha che:

`str[n]` corrisponde a `*(str+n)`

e in particolare `str[0]` corrisponde a `*(str+0)`=`*(str)`=`*str`

C - stringhe

Le stringhe in C sono vettori di caratteri, ossia puntatori a sequenze di bytes, la cui terminazione è definita dal valore convenzionale 0 (zero).

Un carattere tra **apici singoli** equivale all'intero del codice corrispondente.

In particolare un vettore di stringhe è un vettore di vettore di caratteri e dunque:

```
char c;        #carattere
char *str;     #vettore di caratteri / stringa
char **strarr; #vettore di vettore di caratteri / vettore di stringhe
```

Si comprende quindi la segnatura della funzione main con `**argv`.

Array e stringhe

C supporta l'uso di stringhe che, tuttavia, corrispondono a degli array di caratteri.

```
int nome[DIM];
long nome[] = {1,2,3,4};
char string[] = “ciao”;
char string2[] = {‘c’,’i’,’a’,’o’};
nome[0] = 22;
```

Gli array sono generalmente di dimensione statica e non possono essere ingranditi durante l'esecuzione del programma. Per array dinamici dovranno essere usati costrutti particolari (come malloc).

Le stringhe, quando acquisite in input o dichiarate con la sintassi `"stringa"`, terminano con il carattere '`\0`' e sono dunque di grandezza *str_len+1*

C - esempio carattere e argc/argv

```c
#include <stdio.h>

int main(int argc, char **argv) {
  int code=0;
  if (argc<2) {
    printf("Usage: %s <carattere>\n", argv[0]);
    code=2;
  } else {
    printf("%c == %d\n", argv[1][0], argv[1][0]);
  };
  return code;
};
```

C - argomenti da CLI

- Per il parsing degli argomenti da CLI la libreria `getopt.h` mette a disposizione `getopt` e `getopt_long`.

  (v. https://www.gnu.org/software/libc/manual/html_node/Getopt.html)

- Si può effettuare un parsing manuale scorrendo gli argomenti.

C - parsing manuale argomenti: esempio

```
#define MAXOPTL 64
#define MAXOPTS 10
#include <stdio.h>
#include <string.h>
// arrays of options and of values
char opt[MAXOPTS][MAXOPTL];
char val[MAXOPTS][MAXOPTL];

int main(int argc, char **argv) {
  int a=0, o=0;
  // loop into arguments:
  while (++a<argc && o<MAXOPTS) {
     if (strcmp("-h", argv[a])==0)
          strcpy(opt[o++], "help");
```

```
…

      if (strcmp("-k", argv[a])==0) {
          strcpy(opt[o++], "key");
         if (a+1<argc)
           strcpy(val[o-1], argv[++a]);
      }
   }
   // dump options (keys/values):
   for (a=0; a<o; a++) {
     printf("opt[%d]: %s,%s\n",a,opt[a],val[a]);
   }
   return 0;
}
```

C - funzioni stringhe `<string.h>`

Dato che le stringhe sono riferite con un puntatore al primo carattere non ha senso fare assegnamenti e confronti diretti, ma si devono usare delle funzioni. La libreria standard string.h ne definisce alcune come ad esempio:

`char * strcat(char *dest, const char *src)` aggiunge src in coda a dest
`char * strchr(const char *str, int c)` cerca la prima occorrenza di c in `str`
`int strcmp(const char *str1, const char *str2)` confronta `str1` con `str2`
`size_t strlen(const char *str)` calcola la lunghezza di `str`
`char * strcpy(char *dest, const char *src)` copia la stringa src in dst
`char * strncpy(char *dest, const char *src, size_t n)` copia n caratteri dalla stringa src in dst

C - esercizi per casa

1. Scrivere un'applicazione che data una stringa come argomento ne stampa a video la lunghezza, ad esempio:
   `./lengthof "Questa frase ha 28 caratteri"`
   deve restituire a video il numero 28.

2. Scrivere un'applicazione che definisce una lista di argomenti validi e legge quelli passati alla chiamata verificandoli e memorizzando le opzioni corrette, restituendo un errore in caso di un'opzione non valida.

3. Realizzare funzioni per stringhe `char *stringrev(*char str)` (inverte ordine caratteri) e `int stringpos(*char str, char chr)` (cerca *`chr`* in *`str`* e restituisce la posizione)

(In tutti i casi si può completare l'esercizio gestendo gli eventuali errori di immissione da parte dell'utente come parametri errati o altro)

C - files

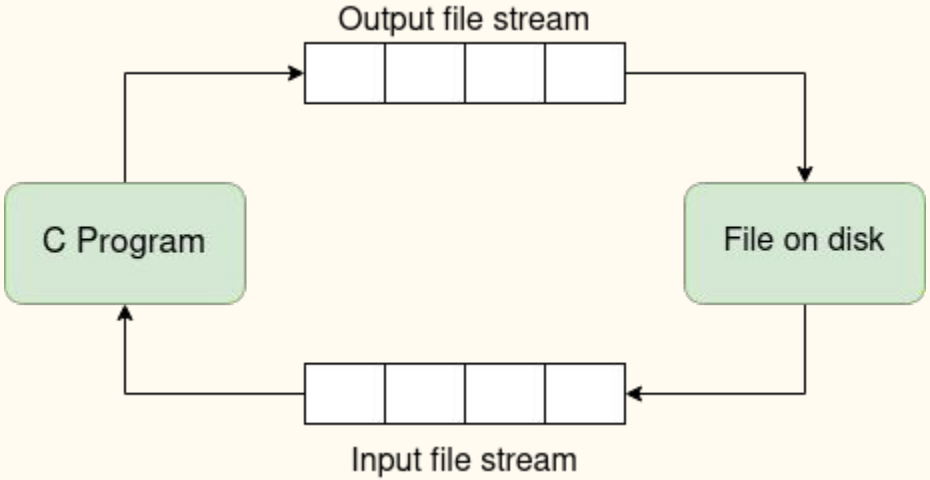
Interazione con i file

In UNIX ci sono due modi per interagire con i file: **streams** e **file descriptors.**

- **Streams**: forniscono strumenti come la formattazione dei dati, bufferizzazione, ecc...
- **File descriptors**: interfaccia di basso livello costituita dalle system call messe a disposizione dal kernel.

Interazione con i file - Streams

Utilizzando gli streams, un file è descritto da un puntatore a una struttura di tipo FILE (definita in stdio.h). I dati possono essere letti e scritti in vari modi (un carattere alla volta, una linea alla volta, ecc.) ed essere interpretati di conseguenza.



Interazione con i file - Stream

```c
#include <stdio.h>

FILE *ptr;  //Declare stream file
ptr = fopen("filename.txt","r+"); //Open

int id;
char str1[10], str2[10];
while (!feof(ptr)){ //Check end of file
    //Read int, word and word
    fscanf(ptr,"%d %s %s", &id, str1, str2);
    printf("%d %s %s\n",id,str1,str2);
}

printf("End of file\n");

fclose(ptr); //Close file
```

filename.txt:

```
1 Nome1 Cognome1
2 Nome2 Cognome2
3 Nome3 Cognome3
```

Modes:

- r: read
- w: write or overwrite (create)
- r+: read and write
- w+: read and write. Create or overwrite
- a: write at end (create)
- a+: read and write at end (create)

Interazione con i file - Stream

```c
#include <stdio.h>
#define N 10

FILE *ptr;
ptr = fopen("fileToWrite.txt","w+");
fprintf(ptr,"Content to write"); //Write content to file
rewind(ptr); // Reset pointer to begin of file
char chAr[N], inC;
fgets(chAr,N,ptr); // store the next N-1 chars from ptr in chAr
printf("'%c' '%s'",chAr[N-1], chAr);
do{
    inC = fgetc(ptr); // return next available char or EOF
    printf("%c",inC);
}while(inC != EOF); printf("\n");
fclose(ptr);
```

File Descriptors

Un file è descritto da un semplice **intero** (file descriptor) che punta alla rispettiva entry nella file table del sistema operativo. I dati possono essere letti e scritti soltanto un buffer alla volta di cui spetta al programmatore stabilire la dimensione.

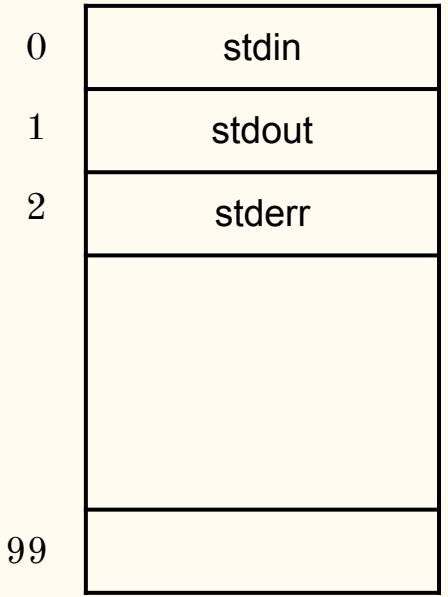
Un insieme di system call permette di effettuare le operazioni di input e output mantenendo un controllo maggiore su quanto sta accadendo a prezzo di un'interfaccia meno amichevole.

File Descriptors

Per accedere al contenuto di un file bisogna creare un canale di comunicazione con il kernel, aprendo il file con la system call open la quale localizza l'i-node del file e aggiorna la *file table* del processo.

A ogni processo è associata una tabella dei file aperti di dimensione limitata (circa 100 elementi), dove ogni elemento della tabella rappresenta un file aperto dal processo ed è individuato da un indice intero (il "file descriptor")

I file descriptor 0, 1 e 2 individuano normalmente standard input, output ed error (aperti automaticamente)

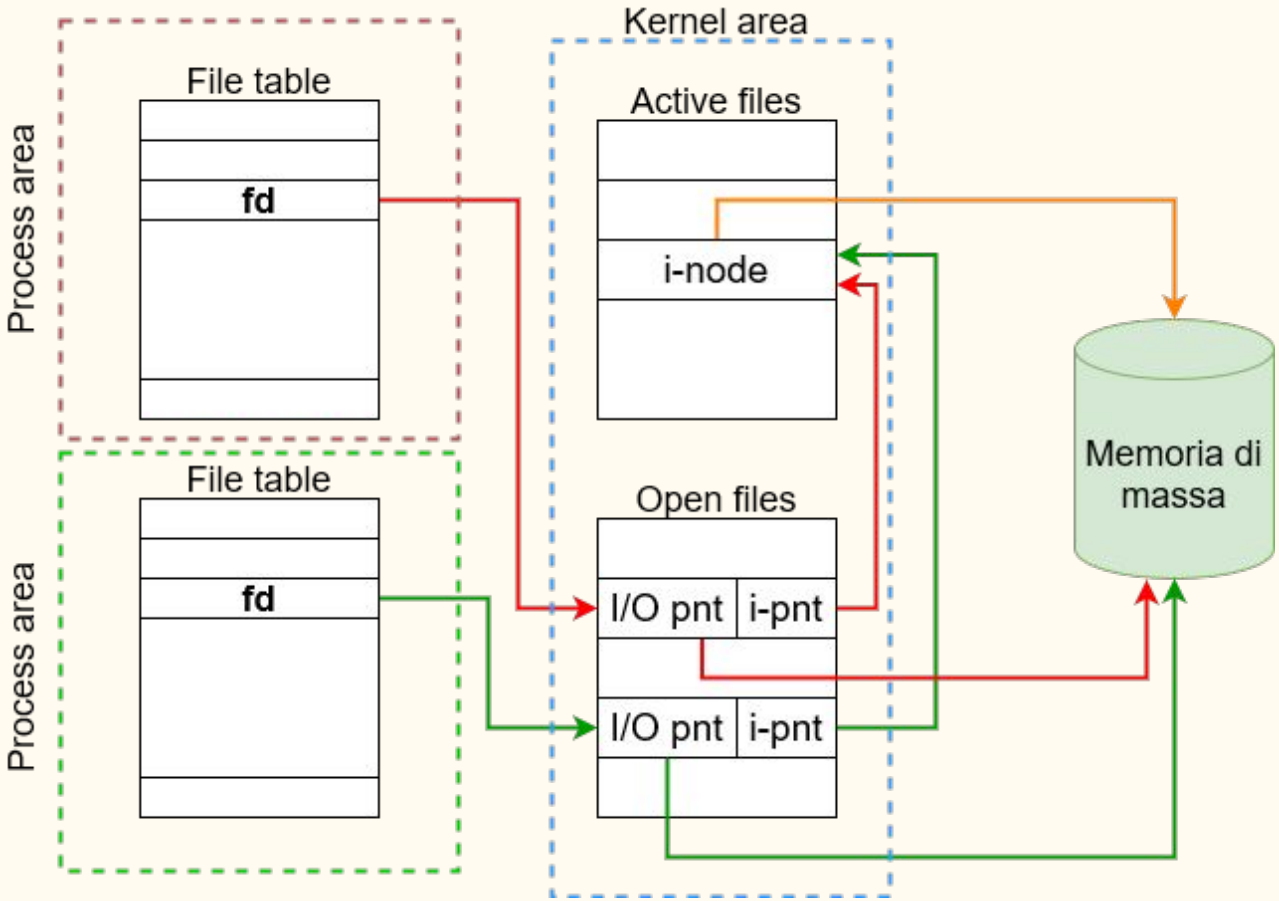


File Descriptors

Il kernel gestisce l'accesso ai files attraverso due strutture dati: **la tabella dei files attivi e la tabella dei files aperti**. La prima contiene una copia dell'inode di ogni file aperto (per efficienza), mentre la seconda contiene un elemento per ogni file aperto e non ancora chiuso. Questo elemento contiene:

- I/O pointer: posizione corrente nel file
- i-node pointer: Puntatore a inode corrispondente

La tabella dei file aperti può avere più elementi corrispondenti allo stesso file!


Kernel area
Process area
File table
fd
Active files
i-node
Memoria di massa
Process area
File table
fd
Open files
I/O pnt
i-pnt
I/O pnt
i-pnt

Interazione files - File Descriptors

L'Input/Output Unix è basato essenzialmente su cinque funzioni: **open**, **read**, **write**, **lseek** e **close.**

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>

//Open new file in Read only
int openedFile = open("filename.txt", O_RDONLY);
char content[10]; int canRead;
do{
     bytesRead = read(openedFile,content,9); //Read 9B from openedFile to buffer content
     content[bytesRead]=0;
     printf("%s",content);
} while(bytesRead > 0);
close(openedFile);
```

Interazione files - File Descriptors

L'Input/Output Unix è basato essenzialmente su cinque funzioni: **open**, **read**, **write**, **lseek** e **close.**

```
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
//Open file (create it with user R and W permissions)
int openFile = open("name.txt", O_CREAT|O_RDWR, S_IRUSR|S_IWUSR);
char toWrite[] = "Professor";

write(openFile, "hello  world\n", strlen("hello world\n")); //Write to file
lseek(openFile, 6, SEEK_SET); // riposiziona l'I/O pointer
write(openFile, toWrite, strlen(toWrite)); //Write to file

close(openFile);
```

Open() flags - File Descriptors

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

Flags: ORed interi che definiscono l'apertura del file

- Deve contenere uno tra O_RDONLY, O_WRONLY, or O_RDWR
- O_CREAT: crea il file se non esistente
- O_APPEND: apri il file in append mode (lseek automatico con ogni write)
- O_TRUNC: cancella il contenuto del file (se aperto con W)
- O_EXCL: se usata con O_CREAT, fallisce se il file esiste già

Mode: definiscono i privilegi da dare al file creato: S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXU, S_IRGRP, ..., S_IROTH

Creat() e lseek()- File Descriptors

```
int creat(const char *pathname, mode_t mode);
```

Alias di open(file,O_CREAT|O_WRONLY|O_TRUNC, mode)

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

Muove la “testina” del file di un certo offset a partire da una certa posizione:

- SEEK_SET = da inizio file,
- SEEK_CUR = dalla posizione corrente
- SEEK_END = dalla fine del file.

C - files: canali standard I

- I canali standard (in/out/err che hanno indici 0/1/2 rispettivamente) sono rappresentati con strutture “stream” (`stdin`, `stdout`, `stderr`) e macro (`STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO`).

- La funzione `fileno` restituisce l’indice di uno “stream”, per cui si ha:
  - `fileno(stdin)=STDIN_FILENO   // = 0`
  - `fileno(stdout)=STDOUT_FILENO // = 1`
  - `fileno(stderr)=STDERR_FILENO // = 2`
- `isatty(stdin) == 1` (se l’esecuzione è interattiva) OPPURE 0 (altrimenti)

`printf("ciao");` e `fprintf(stdout, "ciao");` sono equivalenti!

C - files: canali standard II

```
#include <stdio.h>
#include <unistd.h>

void main() {
  printf("stdin:  stdin ->_flags = %hd, STDIN_FILENO  = %d\n",
    stdin->_flags,  STDIN_FILENO
  );
  printf("stdout: stdout->_flags = %hd, STDOUT_FILENO = %d\n",
    stdout->_flags, STDOUT_FILENO
  );
  printf("stderr: stderr->_flags = %hd, STDOUT_FILENO = %d\n",
    stderr->_flags, STDERR_FILENO
  );
}
```

C - funzioni/operatori

generali e di uso comune

printf / fprintf

```
int printf(const char *format, ...)
int fprintf(FILE *stream, const char *format, ...)
```

Inviano dati sul canale stdout (`printf`) o su quello specificato (`fprintf`) secondo il formato indicato.

Il formato è una stringa contenente contenuti stampabili (testo, a capo, ...) ed eventuali segnaposto identificabili dal formato generale:

```
%[flags][width][.precision][length]specifier
```

Ad esempio: `%d` (intero con segno), `%c` (carattere), `%s` (stringa), ...

Ad ogni segnaposto deve corrispondere un ulteriore argomento del tipo corretto.
(rivedere esempi precedenti)

exit

`void exit(int status)`

Il processo è terminato restituendo il valore `status` come codice di uscita. Si ottiene lo stesso effetto se all'interno della funzione `main` si ha `return status`.

La funzione non ha un valore di ritorno proprio perché non sono eseguite ulteriori istruzioni dopo di essa.

Il processo chiamante è informato della terminazione tramite un "segnale" apposito. I segnali sono trattati più avanti nel corso.

C - piping via bash

C - piping via bash

- In condizioni normali l'applicazione richiamata da bash ha accesso ai canali stdin, stdout e stderr comuni (tastiera/video).
- Se l'applicazione è inserita via bash in un "piping" (come in `ls | wc -l`) allora:
  - Accede all'output del comando a sinistra da stdin
  - Invia il suo output al comando di destra su stdout

```
#define MAXBUF 10
#include <stdio.h>
#include <string.h>

int main() {
    char buf[MAXBUF];
    fgets(buf, sizeof(buf), stdin); // may truncate!
    printf("%s\n", buf);
    return 0;
}
```

```
$ gcc src.c -o pip.out
$ echo "ciao come stai" | ./pip.out
```

C - esempio piping da bash

Esempio di una semplice applicazione che legge da `stdin` e stampa su `stdout` invertendo minuscole [a-z] con maiuscole [A-Z]

```c
#include <stdio.h>

int main() {
    int c, d;
    // loop into stdin until EOF (as CTRL+D)
    while ((c = getchar()) != EOF) { // read from stdin
        d = c;
        if (c >= 'a' && c <= 'z') d -= 32;
        if (c >= 'A' && c <= 'Z') d += 32;
        putchar(d);         // write to stdout
    };
    return (0);
}
```

CONCLUSIONI

Comprendendo il funzionamento dei vari tipi di variabili, in particolare la gestione dei puntatori e dei vettori, e sfruttando poi le funzioni illustrate è possibile realizzare delle applicazioni che manipolano argomenti passati via CLI o anche interagire con processi terzi (in particolare attraverso il file-system o via bash con il piping).

LabSO 2022

Laboratorio Sistemi Operativi - A.A. 2021-2022

dr. Andrea Naimoli	Informatica LT andrea.naimoli@unitn.it
dr. Michele Grisafi	Ingegneria informatica, delle comunicazioni ed elettronica (LT) michele.grisafi@unitn.it

Nota sugli “snippet” di codice

Alcuni esempi di codice possono essere semplificati, ad esempio omettendo il blocco principale con la funzione `main` *(che andrebbe aggiunto) oppure elencando alcune o tutte le librerie da includere tutte su una riga o insieme (per cui invece occorre trascrivere correttamente le direttive* `#include` *secondo la sintassi corretta) o altre semplificazioni analoghe. In questi casi occorre sistemare il codice perché possa essere correttamente compilato e poi eseguito.*

Architettura

Kernel Unix

Il kernel è l'elemento di base di un sistema Unix-like, ovvero il nucleo del sistema operativo. Il kernel è incaricato della gestione delle risorse essenziali: CPU, memoria, periferiche, ecc...

Ad ogni boot il sistema verifica lo stato delle periferiche, monta la prima partizione (root file system) in read-only e carica il kernel in memoria. Il kernel lancia il primo programma (*systemd*, sostituto di *init*) che, a seconda della configurazione voluta (target), inizializza il sistema di conseguenza.

Il resto delle operazioni, tra cui l'interazione con l'utente, vengono gestite con i programmi eseguiti dal kernel.

Kernel e memoria virtuale

I programmi utilizzati dall'utente che vogliono accedere alle periferiche chiedono al kernel di farlo per loro.

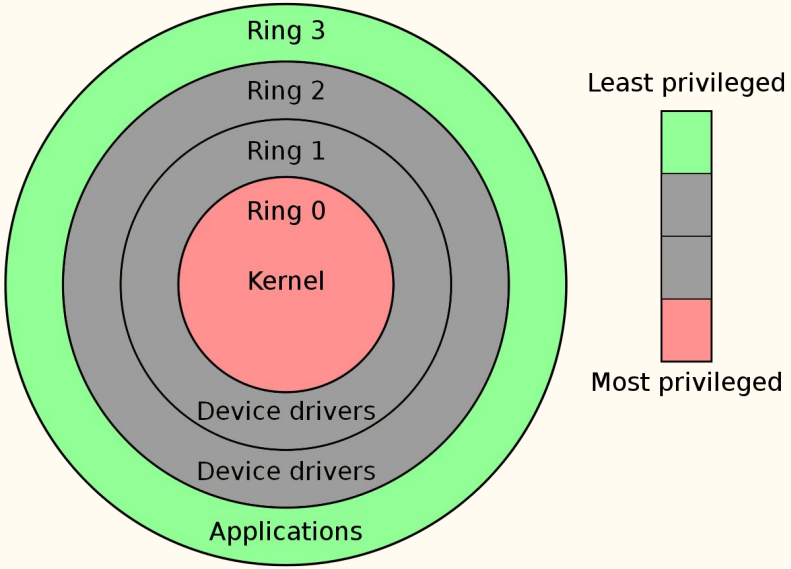
L'interazione tra programmi ed il resto del sistema viene mascherata da alcune caratteristiche intrinseche ai processori, come la gestione hardware della memoria virtuale (attraverso la MMU).

Ogni programma vede se stesso come **unico possessore della CPU** e non gli è dunque possibile disturbare l'azione degli altri programmi → stabilità dei sistemi Unix-like!

Privilegi

Nei sistemi Unix-like ci sono due livelli di privilegi:

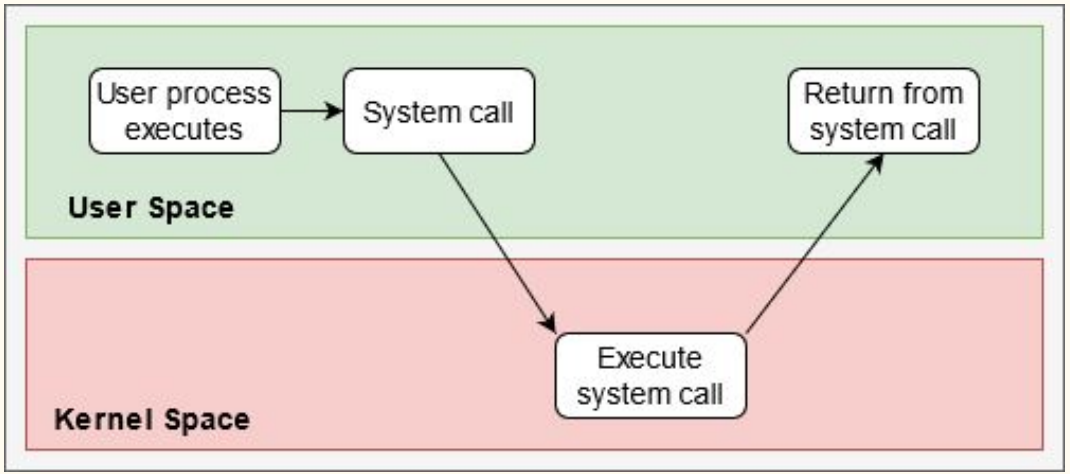
- **User space:** ambiente in cui vengono eseguiti i programmi.
- **Kernel space:** ambiente in cui viene eseguito il kernel.



System calls

System calls

Le interfacce con cui i programmi accedono all'hardware si chiamano **system calls.** Letteralmente "chiamate al sistema" che il kernel esegue nel kernel space, restituendo i risultati al programma chiamante nello user space.



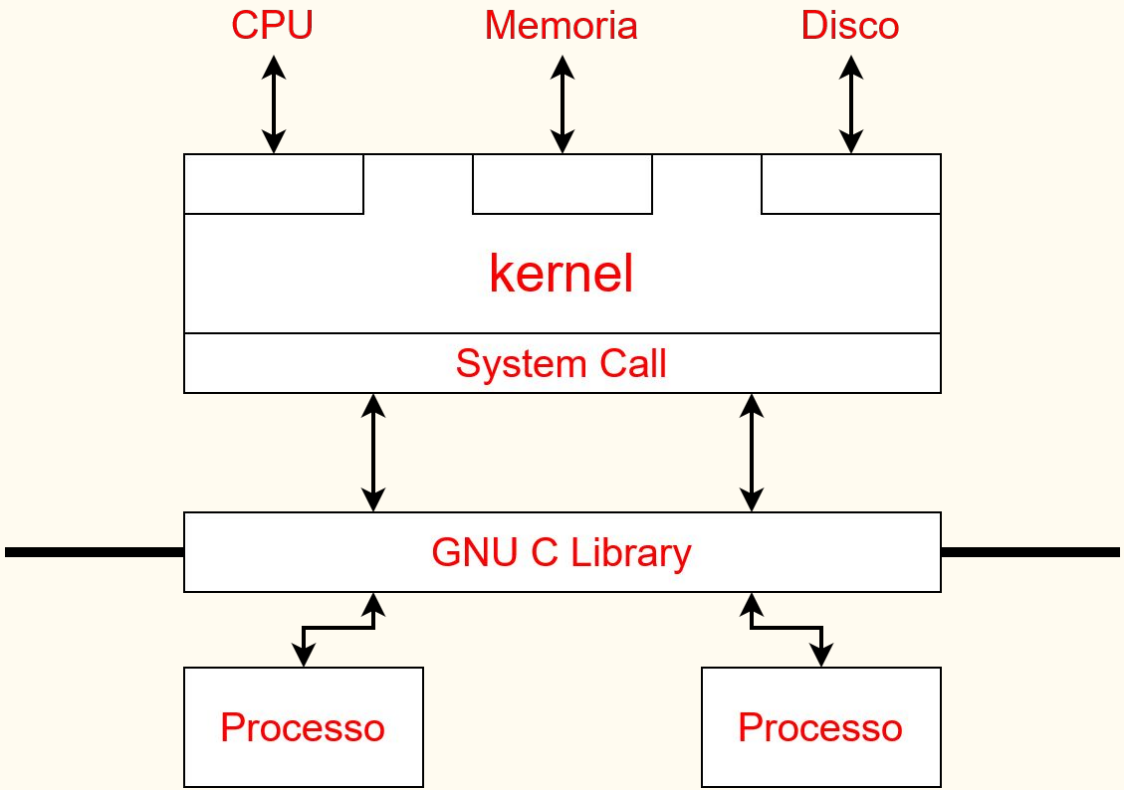
Le chiamate restituiscono "-1" in caso di errore e settano la variabile globale `errno`. Errori validi sono numeri positivi e seguono lo standard POSIX, il quale definisce degli alias.

System calls: librerie di sistema

Utilizzando il comando di shell `ldd` su di un eseguibile si possono visualizzare le librerie condivise caricate e, fra queste, vi sono tipicamente anche *ld-linux.so*, e *libc.so*.

- **ld-linux.so**: quando un programma è caricato in memoria, il sistema operativo passa il controllo a *ld-linux.so* anzichè al normale punto di ingresso dell'applicazione. ld-linux trova e carica le librerie richieste, prepara il programma e poi gli passa il controllo.
- **libc.so**: la libreria GNU C solitamente nota come glibc che contiene le funzioni basilari più comuni.

System Calls: librerie di sistema



Get time: time() e ctime()

```
time_t time( time_t *second )
char * ctime( const time_t *timeSeconds )
```

```c
#include <time.h>                                          //time.c
#include <stdio.h>

void main(){
    time_t theTime;
    time_t whatTime = time(&theTime); //seconds since 1/1/1970
    //Print date in Www Mmm dd hh:mm:ss yyyy
    printf("Current time = %s= %d\n", ctime(&whatTime),theTime);
}
```

Working directory: chdir(), getcwd()

```
int chdir( const char *path );
char * getcwd( char *buf, size_t sizeBuf );
```

```
#include <unistd.h>                                          //chdir.c
#include <stdio.h>

void main(){
    char s[100];
    getcwd(s,100); // copy path in buffer
    printf("%s\n", s); //Print current working dir
    chdir(".."); //Change working dir
    printf("%s\n", getcwd(NULL,100)); // Allocates buffer
}
```

Operazioni con i file

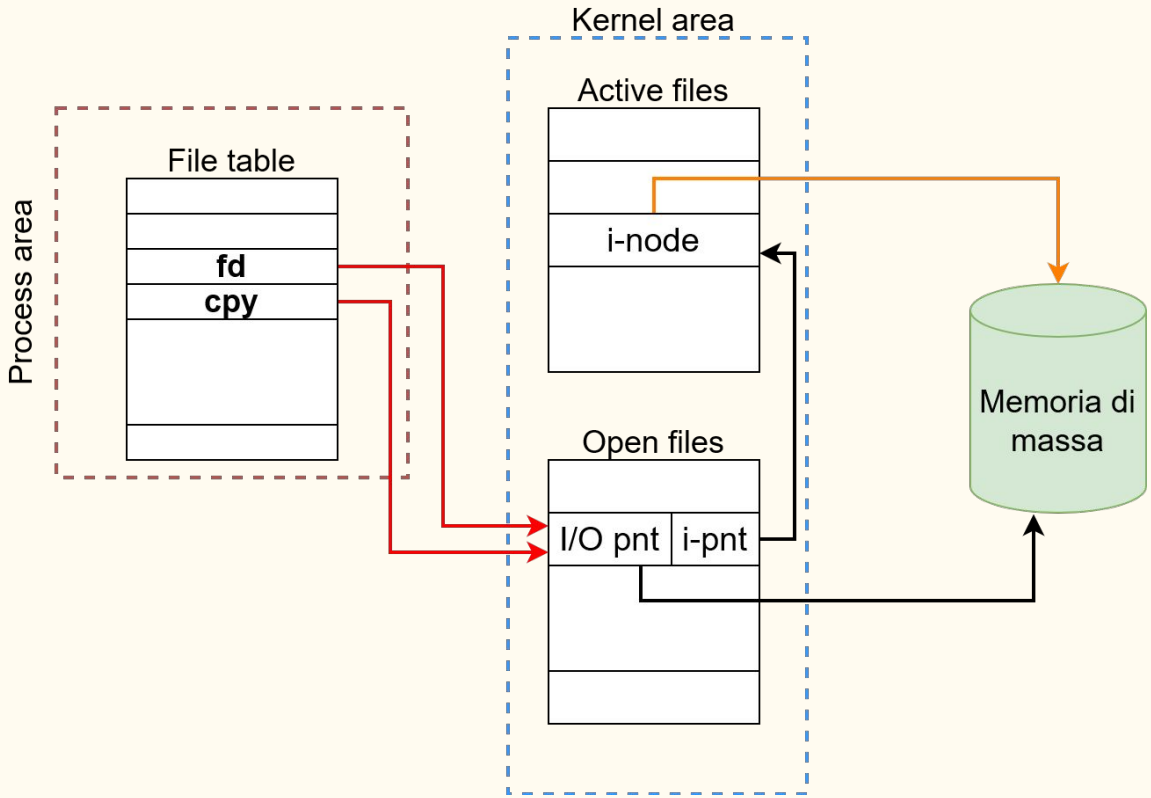
```
int open(const char *pathname, int flags, mode_t mode);
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);


FILE *fopen(const char *filename, const char *mode)
int fclose(FILE *stream)

.....
```

Duplicazione file descriptors: dup(), dup2()

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

```
#include <unistd.h> <stdio.h> <fcntl.h>
int main(void){
    char buf[51];
    int fd = open("file.txt",O_RDWR); //file exists
    int r = read(fd,buf,50); //Read 50 bytes from ‘fd’ in ‘buf’
    buf[r] = 0; printf("Content: %s\n",buf);
    int cpy = dup(fd); // Create copy of file descriptor
    dup2(cpy,22); // Copy cpy to descriptor 22 (close 22 if opened)
    lseek(cpy,0,SEEK_SET); // Move I/O on all 3 file descriptors!
    write(22,"This is a fine\n",16); // Write starting from 0-pos
    close(cpy); //Close ONE file descriptor
}
```

Duplicazione file descriptors: dup(), dup2()



Permessi: chmod(), chown()

```
int chown(const char *pathname, uid_t owner, gid_t group)
int fchown(int fd, uid_t owner, gid_t group)
int chmod(const char *pathname, mode_t mode)
int fchmod(int fd, mode_t mode)
```

//chown.c

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
void main(){
    int fd = open("file",O_RDONLY);
    fchown(fd, 1000, 1000); // Change owner to user:group 1000:1000
    chmod("file",S_IRUSR|S_IRGRP|S_IROTH); // Permission to r/r/r
}
```

```
$ cd .. ; touch file ; ls -la
$ ./executable.o ; ls -la
```

Eseguire programmi: execve() ed alias

```
int execv(const char *path, char *const argv[])
int execvp(const char *file, char *const argv[])
int execvpe(const char *file, char *const argv[],char *const
            envp[])

int execl(const char *path, const char * arg0, …,argn,NULL)
int execlp(const char *file, const char * arg0, …,argn,NULL)
int execle(const char *file, const char * arg0, …,argn,
           NULL, char *const envp[])

int execve(const char *filename, char *const argv[], char
           *const envp[])
```

Esempio: execv()

```c
#include <unistd.h>                                          //execv1.out
#include <stdio.h>
void main(){
    char * argv[] = {"par1","par2",NULL};
    execv("./execv2.out",argv); //Replace current process
    printf("This is execv1\n");
}
```

```c
#include <stdio.h>                                           //execv2.out
void main(int argc, char ** argv){
    printf("This is execv2 with %s and %s\n",argv[0],argv[1]);
}
```

Esempio: execle()

```c
#include <unistd.h>                                          //execle1.out
#include <stdio.h>
void main(){
    char * env[] = {"CIAO=hello world",NULL};
    execle("./execle2.out","par1","par2",NULL,env); //Replace proc.
    printf("This is execle1\n");
}
```

```c
#include <stdio.h>                                           //execle2.out
#include <stdlib.h>
void main(int argc, char ** argv){
    printf("This is execv2 with par: %s and %s. CIAO =
            %s\n",argv[0],argv[1],getenv("CIAO"));
}
```

Esempio: dup2/exec

```c
#include <stdio.h>                                          //execvpDup.c
#include <fcntl.h>
#include <unistd.h>

void main() {
  int outfile = open("/tmp/out.txt",
    O_RDWR | O_CREAT, S_IRUSR | S_IWUSR
  );
  dup2(outfile, 1); // copy outfile to FD 1
  char *argv[]={"./time.out",NULL}; // time.out della slide#10
  execvp(argv[0],argv); // Replace current process
}
```

Chiamare la shell: system()

```
int system(const char * string);
```

```
#include <stdlib.h> <stdio.h>                                    //system.c
#include <sys/wait.h> /* For WEXITSTATUS */

void main(){
    int outcome = system("echo ciao"); // execute command in shell
    printf("Outcome = %d\n",outcome);
    outcome = system("if [[ $PWD < \"ciao\" ]]; then echo min; fi");
    printf("Outcome = %d\n",outcome);
    outcome = system("notExistingCommand");
    printf("Outcome = %d\n",WEXITSTATUS(outcome));
}
```

Altre system calls: segnali e processi

Ci sono molte altre system calls per la gestione dei processi e della comunicazione tra i processi che saranno discusse più avanti.

Forking

System call “fork”

La syscall principale per il forking è “fork”.

Il forking è la “generazione” di nuovi processi (uno alla volta) partendo da uno esistente.

Un processo attivo invoca la syscall e così il kernel lo “clona” modificando però alcune informazioni e in particolare quelle che riguardano la sua collocazione nella gerarchia complessiva dei processi.

Il processo che effettua la chiamata è definito “padre”, quello generato è definito “figlio”.

fork: elementi clonati e elementi nuovi

Sono clonati gli elementi principali come il PC (Program Counter), i registri, la tabella dei file (file descriptors) e i dati di processo (variabili). Le meta-informazioni come il “pid” e il “ppid” sono aggiornate (al contrario di execve()!).

L’esecuzione procede per entrambi (quando saranno schedulati!) da PC+1 (tipicamente l’istruzione seguente il fork o la valutazione dell’espressione in cui essa è utilizzata):

Prossimo step: `printf`	Prossimo step: assegnamento ad `f`
`fork();` `printf(“\n”);`	`f=fork();` `printf(“\n”);`

Identificativi dei processi

Ad ogni processo è associato un identificativo univoco per istante temporale, sono organizzati gerarchicamente (padre-figlio) e suddivisi in insiemi principali (sessioni) e secondari (gruppi). Anche gli utenti hanno un loro identificativo e ad ogni processo ne sono abbinati due: quello reale e quello effettivo (di esecuzione)

- PID - Process ID
- PPID - Parent Process ID
- SID - Session ID
- PGID - Process Group ID
- UID/RUID - (Real) User ID
- EUID - Effective User ID

Approfonditi in un'altra lezione!

getpid(), getppid()

`pid_t getpid()` : restituisce il PID del processo attivo
`pid_t getppid()` : restituisce il PID del processo padre

```c
#include <stdio.h> <unistd.h> <stdlib.h>                    //ppid.c

void main(){
    printf("Subshell $$ = ");
    fflush(stdout); // Forza l'output di printf
    system("echo $$"); // subshell
    printf("PID: %dPPID: %d\n",getpid(),getppid());
}
```

(includendo `<sys/types.h>` e `<sys/wait.h>`: pid_t è un intero che rappresenta un id di processo)

fork: valore di ritorno

La funzione restituisce un valore che solitamente è catturato in una variabile (o usato comunque in un'espressione).

Come per tutte le syscall in generale, il valore è -1 in caso di errore (in questo caso non ci sarà nessun nuovo processo, ma solo quello che ha invocato la chiamata).

Se ha successo entrambi i processi ricevono un valore di ritorno, ma questo è diverso nei due casi:

- Il processo padre riceve come valore il nuovo PID del processo figlio
- Il processo figlio riceve come valore 0

fork: relazione tra i processi

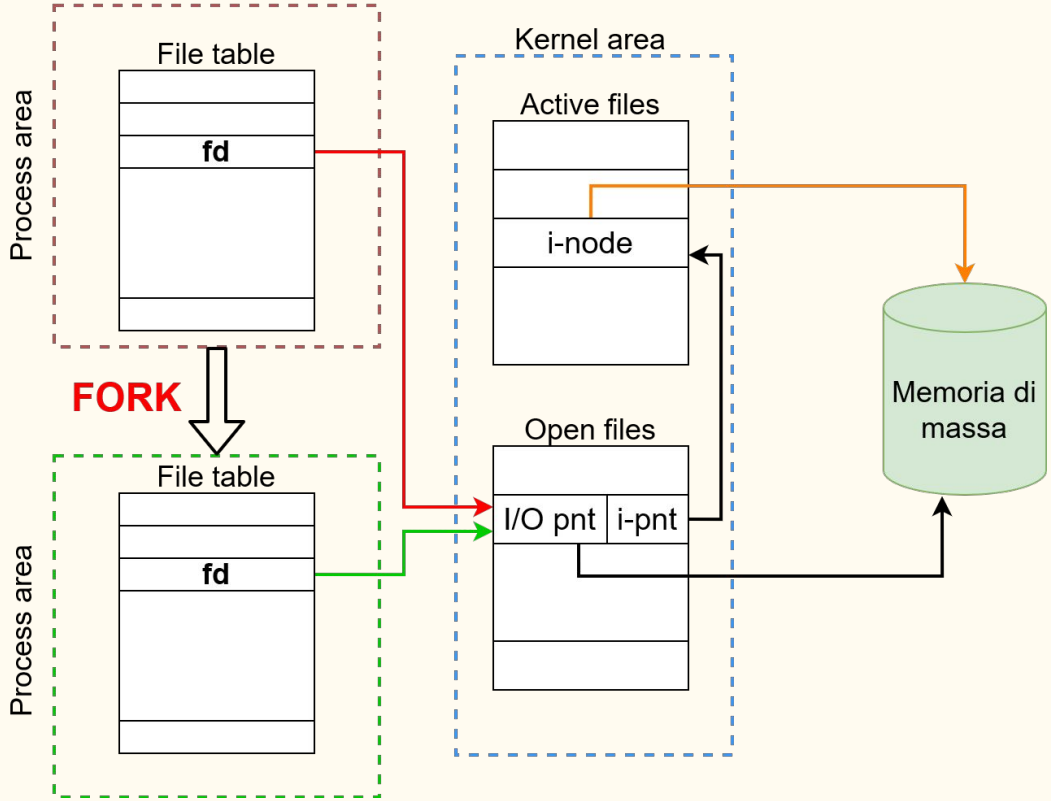
I processi padre-figlio:

- Conoscono reciprocamente il loro PID (ciascuno conosce il proprio tramite `getpid()`, il figlio conosce quello del padre con `getppid()`, il padre conosce quello del figlio come valore di ritorno di `fork()`)

- Si possono usare altre syscall per semplici interazioni come `wait` e `waitpid`

- Eventuali variabili definite prima del fork sono valorizzate allo stesso modo in entrambi: se riferiscono risorse (ad esempio un “file descriptor” per un file su disco) fanno riferimento esattamente alla stessa risorsa.

File Descriptors con fork



fork: wait(), waitpid()

`pid_t wait (int *status)` : attende la **terminazione** di UN figlio (qualunque) e ne restituisce il PID, riportando lo status nel puntatore passato come argomento (se non NULL).

`pid_t waitpid(pid_t pid, int *status, int options)` : analoga a wait ma consente di passare delle opzioni e si può specificare come pid:

- `-n` (<-1: attende un qualunque figlio il cui "gruppo" è |-n|)
- `-1` (attende un figlio qualunque)
- `0` (attende un figlio con lo stesso "gruppo" del padre)
- `n` (>0: attende il figlio il cui pid è esattamente n)

NOTE:
`wait(st)` corrisponde a `waitpid(-1, st, 0)`
`while(wait(NULL)>0);` # attende tutti i figli

Wait: interpretazione stato

Lo stato di ritorno è un numero che comprende più valori “composti” interpretabili con apposite macro, molte utilizzabili a mo’ di funzione (altre come valore) passando lo “stato” ricevuto come risposta come ad esempio:

`WEXITSTATUS(sts)`: restituisce lo stato vero e proprio (ad esempio il valore usato nella “exit”).
`WIFCONTINUED(sts)`: true se il figlio ha ricevuto un segnale SIGCONT.
`WIFEXITED(sts)`: true se il figlio è terminato normalmente.
`WIFSIGNALED(sts)`: true se il figlio è terminato a causa di un segnale non gestito.
`WIFSTOPPED(sts)`: true se il figlio è attualmente in stato di “stop”.
`WSTOPSIG(sts)`: numero del segnale che ha causato lo “stop” del figlio.
`WTERMSIG(sts)`: numero del segnale che ha causato la terminazione del figlio.

Esempio fork multiplo

Ovviamente è possibile siano presenti più “fork” dentro un codice.

Quante righe saranno generate in output dal seguente programma?

```
#include <stdio.h>                    //fork1.c
#include <unistd.h>
int main() {
  fork(); fork(); fork();
  printf("hello\n");
  return 0;
}
```

Esempio fork&wait

```c
#include <stdio.h> <stdlib.h> <unistd.h> <time.h> <sys/wait.h>          //fork2.c
int main() {
  int fid=fork(), wid, st, r; // Generate child
  srand(time(NULL)); // Initialise random
  r=rand()%256; // Get random between 0 and 255
  if (fid==0) {  //If it is child
     printf("Child... (%d)", r); fflush(stdout);
     sleep(3);  // Pause execution for 3 seconds
     printf(" done!\n");
     exit(r); // Terminate with random signal
  } else { // If it is parent
     printf("Parent...\n");
     wid=wait(&st);  // wait for ONE child to terminate
     printf("...child's id: %d==%d (st=%d)\n", fid, wid, WEXITSTATUS(st));
  }
}
```

I processi “zombie” e “orfani”

Normalmente quando un processo termina il suo stato di uscita è “catturato” dal padre: alla terminazione il sistema tiene traccia di questo insieme di informazioni (lo stato) fino a che il padre le utilizza consumandole (con *wait* o *waitpid*). Se il padre non cattura lo stato d’uscita i processi figli sono definiti “zombie” (in realtà non ci sono più, ma esiste un riferimento in sospeso nel sistema).

Se un padre termina prima del figlio, quest’ultimo viene definito “orfano” e viene “adottato” dal processo principale (tipicamente “init” con pid pari a 1).

Un processo zombie che diventa anche orfano è poi gestito dal processo che lo adotta (che effettua periodicamente dei *wait*/*waitpid* appositamente)

Esercizi

Scrivere dei programmi in C che:

1. Avendo come argomenti dei “binari”, si eseguono con *exec* ciascuno in un sottoprocesso (*)
2. idem punto 1 ma in più salvando i flussi di *stdout* e *stderr* in un unico file (*)
3. Dati due eseguibili come argomenti del tipo *ls* e *wc* si eseguono in due processi distinti: il primo deve generare uno *stdout* redirezionato su un file temporaneo, mentre il secondo deve essere lanciato solo quando il primo ha finito leggendo lo stesso file come *stdin*.
   Ad esempio `./main ls wc` deve avere lo stesso effetto di `ls | wc`.

   Suggerimenti:
   - anziché due figli usare padre-figlio
   - usare dup2 per far puntare il file-descriptor del file temporaneo su *stdout* in un processo e *stdin* nell’altro
   - sfruttare *wait* per attendere la conclusione del processo che genera l’output

(*) generando DUE figli

CONCLUSIONI

Tramite l'uso dei *file-descriptors*, di *fork* e della famiglia di istruzioni *exec* è possibile generare più sottoprocessi e "redirezionare" i loro canali di in/out/err.

Sfruttando anche *wait* e *waitpid* è possibile costruire un albero di processi che interagiscono tra loro (non avendo ancora a disposizione strumenti dedicati è possibile sfruttare il file-system - ad esempio con file temporanei - per condividere informazioni/dati).

LabSO 2022

Laboratorio Sistemi Operativi - A.A. 2021-2022

dr. Andrea Naimoli	Informatica LT andrea.naimoli@unitn.it
dr. Michele Grisafi	Ingegneria informatica, delle comunicazioni ed elettronica (LT) michele.grisafi@unitn.it

Nota sugli “snippet” di codice

Alcuni esempi di codice possono essere semplificati, ad esempio omettendo il blocco principale con la funzione `main` *(che andrebbe aggiunto) oppure elencando alcune o tutte le librerie da includere tutte su una riga o insieme (per cui invece occorre trascrivere correttamente le direttive* `#include` *secondo la sintassi corretta) o altre semplificazioni analoghe. In questi casi occorre sistemare il codice perché possa essere correttamente compilato e poi eseguito.*

Segnali

Segnali in Unix

Ci sono vari eventi che possono avvenire in maniera asincrona al normale flusso di un programma, alcuni dei quali in maniera inaspettata e non predicibile. Per esempio, durante l'esecuzione di un programma ci può essere una richiesta di terminazione o di sospensione da parte di un utente, la terminazione di un processo figlio o un errore generico.

Unix prevede la gestione di questi eventi attraverso i **segnali**: quando il sistema operativo si accorge di un certo evento, genera un segnale da mandare al processo interessato il quale potrà *decidere* (nella maggior parte dei casi) come comportarsi.

Segnali in Unix

Il numero dei segnali disponibili cambia a seconda del sistema operativo, con Linux che ne definisce 32. Ad ogni segnale corrisponde sia un valore numerico che un'etichetta mnemonica (definita nella libraria "*signal.h*") nel formato `SIG`XXX. Alcuni esempi:

`SIGALRM` (alarm clock)
`SIGCHLD` (child terminated)
`SIGCONT` (continue, if stopped)
`SIGINT` (terminal interrupt, CTRL + C)
`SIGKILL` (kill process)
`SIGQUIT` (terminal quit)
`SIGSTOP` (stop)
`SIGTERM` (termination)
`SIGUSR1` (user signal)
`SIGUSR2` (user signal)

Segnali in Unix

Per ogni processo, all'interno della process table, vengono mantenute due liste:

- **Pending signals**: segnali emessi che il processo dovrà gestire.
- **Blocked signals**: segnali non comunicati al processo.

Ad ogni schedulazione del processo le due liste vengono controllate per consentire al processo di reagire nella maniera più adeguata.

Gestione dei segnali

I segnali sono anche detti "*software interrupts*" perchè sono, a tutti gli effetti, delle interruzioni del normale flusso del processo generate dal sistema operativo (invece che dall'hardware, come per gli *hardware interrupts*).

Come per gli interrupts, il programma può decidere come gestire l'arrivo di un segnale (presente nella lista *pending*):

- Eseguendo l'azione default.
- Ignorandolo (non sempre possibile) → programma prosegue normalmente.
- Eseguendo un handler personalizzato → programma si interrompe.

Default handler

Ogni segnale ha un suo handler di default che tipicamente può:

- Ignorare il segnale
- Terminare il processo
- Continuare l'esecuzione (se il processo era in stop)
- Stoppare il processo

Ogni processo può sostituire il gestore di default con una funzione "custom" (**a parte per SIGKILL e SIGSTOP**) e comportarsi di conseguenza. La sostituzione avviene tramite la system call `signal()` (definita in "*signal.h*").

signal() system call

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

```
#include <signal.h> <stdio.h> <stdlib.h>
void myHandler(int sigNum){
    printf("CTRL+Z\n");
}
void main(){
    signal(SIGINT,SIG_IGN);    //Ignore signal
    signal(SIGCHLD,SIG_DFL);   //Use default handler
    signal(SIGTSTP,myHandler);//Use myHandler
}
```

Esempio:

```
#include <signal.h>    //sigCST.c
#include <stdio.h>

void myHandler(int sigNum){
    printf("CTRL+Z\n");
    exit(2);
}

int main(void){
    signal(SIGTSTP,myHandler);
    while(1);
}
```

```
$ gcc sig[CST|DFL|IGN].c -o
sig.out
$ ./sig.out
$ <CTRL+Z>
```

```
#include <signal.h>    //sigDFL.c
int main(){
    signal(SIGTSTP,SIG_DFL);
    while(1);
}
```

```
#include <signal.h>    //sigIGN.c
int main(){
    signal(SIGTSTP,SIG_IGN);
    while(1);
}
```

Custom handler

Un handler personalizzato deve essere una funzione di tipo `void` che accetta come argomento un intero, il quale rappresenta il segnale catturato. Questo consente l'utilizzo di un solo handler per segnali differenti.

```
#include <signal.h> <stdio.h>                                    //param.c

void myHandler(int sigNum){
    if(sigNum == SIGINT) printf("CTRL+C\n");
    else if(sigNum == SIGTSTP) printf("CTRL+Z\n");
}

signal(SIGINT,myHandler);
signal(SIGTSTP,myHandler);
```

signal() return

signal() restituisce un riferimento all'handler che era precedentemente assegnato al segnale:

- NULL: handler precedente era l'handler di default
- 1: l'handler precedente era SIG_IGN
- *address*: l'handler precedente era `*(address)`

```
#include <signal.h> <stdio.h>                                    //return.c
void myHandler(int sigNum){}
int main(){
    printf("DFL: %p\n",signal(SIGINT,SIG_IGN));
    printf("IGN: %p\n",signal(SIGINT,myHandler));
    printf("Custom: %p == %p\n",signal(SIGINT,SIG_DFL),myHandler);
}
```

Alcuni segnali

SIGXXX	**description**	**default**
SIGALRM	(alarm clock)	quit
SIGCHLD	(child terminated)	ignore
SIGCONT	(continue, if stopped)	ignore
SIGINT	(terminal interrupt, CTRL + C)	quit
SIGKILL	(kill process)	quit
SIGSYS	(bad argument to syscall)	quit with dump
SIGTERM	(software termination)	quit
SIGUSR1/2	(user signal 1/2)	quit
SIGSTOP	(stopped)	quit
SIGTSTP	(terminal stop, CTRL + Z)	quit

Esempio

```
#include <signal.h> <stdio.h> <unistd.h> <sys/wait.h>   //child.c
void myHandler(int sigNum){
    printf("Child terminated! Received %d\n",sigNum);
}
int main(){
    signal(SIGCHLD,myHandler);
    int child = fork();
    if(!child){
        return 0; //terminate child
    }
    while(wait(NULL)>0);
}
```

Inviare i segnali: kill()

```
int kill(pid_t pid, int sig);
```

Invia un segnale ad uno o più processi a seconda dell'argomento *pid:*

- pid $>$ 0: segnale al processo con PID=pid
- pid $=$ 0: segnale ad ogni processo dello stesso gruppo
- pid $=$ -1: segnale ad ogni processo possibile (stesso UID/RUID)
- pid $<$ -1: segnale ad ogni processo del gruppo |pid|

Restituisce 0 se il segnale viene inviato, -1 in caso di errore.

Ogni tipo di segnale può essere inviato, non deve essere necessariamente un segnale corrispondente ad un evento effettivamente avvenuto!

```c
#include <signal.h> <stdio.h> <stdlib.h> <sys/wait.h> <unistd.h>
//kill.c
void myHandler(int sigNum){printf("[%d]ALARM!\n",getpid());}
int main(void){
    signal(SIGALRM,myHandler);
    int child = fork();
    if (!child) while(1); // block the child
    printf("[%d]sending alarm to %d in 3 s\n",getpid(),child);
    sleep(3);
    kill(child,SIGALRM); // send ALARM, child's handler reacts
    printf("[%d]sending SIGTERM to %d in 3 s\n",getpid(),child);
    sleep(3);
    kill(child,SIGTERM); // send TERM: default is to terminate
    while(wait(NULL)>0);
}
```

Kill da bash

kill è anche un programma in bash che accetta come primo argomento il tipo di segnale (`kill -l` per la lista) e come secondo argomento il PID del processo.

```c
#include <signal.h> <stdio.h> <stdlib.h> <unistd.h>  //bash.c
void myHandler(int sigNum){
    printf("[%d]ALARM!\n",getpid());
    exit(0);
}
int main(){
    signal(SIGALRM,myHandler);
    printf("I am %d\n",getpid());
    while(1);
}
```

```
$ gcc bash.c -o bash.out
$ ./bash.out
# On new window/terminal
$ kill -14 <PID>
```

Programmare un alarm: alarm()

```
unsigned int alarm(unsigned int seconds);
```

```
#include <signal.h> <stdio.h> <stdlib.h> <unistd.h>  //alarm.c
short cnt = 0;
void myHandler(int sigNum){printf("ALARM!\n"); cnt++;}
int main(){
    signal(SIGALRM,myHandler);
    alarm(5); //Set alarm in 5 seconds
    //Set new alarm (cancelling previous one)
    printf("Seconds remaining to previous alarm %d\n",alarm(2));
    while(cnt<1);
}
```

Mettere in pausa: pause()

```
int pause();
```

```
#include <signal.h> <unistd.h> <stdio.h>                    //pause.c

void myHandler(int sigNum){
    printf("Continue!\n");
}
int main(){
    signal(SIGCONT,myHandler);
    signal(SIGUSR1,myHandler);
    pause();
}
```

```
$ gcc pause.c -o pause.out
$ ./pause.out
# On new window/terminal
$ kill -18/-10 <PID>
```

Bloccare i segnali

Oltre alla lista dei “pending signal” esiste la lista dei “blocked signals”, ovvero dei segnali ricevuti dal processo ma volutamente non gestiti. Mentre i segnali ignorati non saranno mai gestiti, i segnali bloccati sono solo *temporaneamente* non gestiti.

Un segnale bloccato rimane nello stato *pending* fino a quando esso non viene gestito oppure il suo handler tramutato in *ignore*.

L’insieme dei segnali che vanno bloccati è detto “signal mask”, una maschera dei segnali che è modificabile attraverso la system call `sigprocmask()`.

Bloccare i segnali: sigset_t

Una signal mask può essere gestita con un `sigset_t`, ovvero una lista di segnali modificabile con alcune funzioni. Queste funzioni modificano il sigset_t, non la maschera dei segnali del processo!

`int sigemptyset(sigset_t *set);` Svuota
`int sigfillset(sigset_t *set);` Riempie
`int sigaddset(sigset_t *set, int signo);` Aggiunge singolo
`int sigdelset(sigset_t *set, int signo);` Rimuove singolo
`int sigismember(const sigset_t *set, int signo);` Interpella

Bloccare i segnali: sigprocmask()

```
int sigprocmask(int how, const sigset_t *restrict set,
                sigset_t *restrict oldset);
```

A seconda del valore di `how` e di `set`, la maschera dei segnali del processo viene cambiata. Nello specifico:

- `how = SIG_BLOCK`: i segnali in `set` sono aggiunti alla maschera;
- `how = SIG_UNBLOCK`: i segnali in `set` sono rimossi dalla maschera;
- `how = SIG_SETMASK`: `set` diventa la maschera.

Se `oldset` non è nullo, in esso verrà salvata la vecchia maschera (anche se `set` è nullo).

Esempio

```
#include <signal.h>

int main(){
    sigset_t mod,old;
    sigfillset(&mod); // Add all signals to the blocked list
    sigemptyset(&mod); // Remove all signals from blocked list
    sigaddset(&mod,SIGALRM); // Add SIGALRM to blocked list
    sigismember(&mod,SIGALRM); // is SIGALRM in blocked list?
    sigdelset(&mod,SIGALRM); // Remove SIGALRM from blocked list

    // Update the current mask with the signals in ‘mod’
    sigprocmask(SIG_BLOCK,&mod,&old);
}
```

Esempio 2

```
$ kill -10 <PID> # ok
$ kill -10 <PID> # blocked
```

```c
#include <signal.h> <unistd.h> <stdio.h>          //sigprocmask.c
sigset_t mod, old;
int i = 0;
void myHandler(int signo){
    printf("signal received\n");
    i++;
}
int main(){
    printf("my id = %d\n",getpid());
    signal(SIGUSR1,myHandler);
    sigemptyset(&mod); //Initialise set
    sigaddset(&mod,SIGUSR1);
    while(1) if(i==1) sigprocmask(SIG_BLOCK,&mod,&old);
}
```

Verificare pending signals: sigpending()

```
int sigpending(sigset_t *set);
```

```
//sigpending.c
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

sigset_t mod,pen;
void handler(int signo){
    printf("SIGUSR1 received\n");
    sigpending(&pen);
    if(!sigismember(&pen,SIGUSR1))
      printf("SIGUSR1 not pending\n");
    exit(0);
}
```

```
int main(){
    signal(SIGUSR1,handler);
    sigemptyset(&mod);
    sigaddset(&mod,SIGUSR1);
    sigprocmask(SIG_BLOCK,&mod,NULL);
    kill(getpid(),SIGUSR1);
    // sent but it's blocked...
    sigpending(&pen);
    if(sigismember(&pen,SIGUSR1))
        printf("SIGUSR1 pending\n");
    sigprocmask(SIG_UNBLOCK,&mod,NULL);
    while(1);
}
```

sigaction() system call

```
int sigaction(int signum, const struct sigaction *restrict
              act, struct sigaction *restrict oldact);

struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask; //Signals blocked during handler
    int sa_flags; //modify behaviour of signal
    void (*sa_restorer)(void); //Deprecated
};
```

https://linux.die.net/man/2/sigaction

Esempio

```c
#include <signal.h> <unistd.h> <stdlib.h> <stdio.h>   //sigaction.c
void handler(int signo){
    printf("signal received\n");
}
int main(){
    struct sigaction sa; //Define sigaction struct
    sa.sa_handler = handler; //Assign handler to struct field
    sigemptyset(&sa.sa_mask); //Define an empty mask
    sigaction(SIGUSR1,&sa,NULL);
    kill(getpid(),SIGUSR1);
}
```

Esempio: blocking signal

```
$ kill -10 <PID> ; sleep 1
  && kill -12 <PID>
```

```c
#include <signal.h> <unistd.h> <stdlib.h> <stdio.h>  //sigaction2.c
void handler(int signo){
    printf("signal %d received\n",getpid());
    sleep(2);
    printf("Signal done\n");
}
int main(){
    printf("Process id: %d\n",getpid());
    struct sigaction sa;
    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask); //Use an empty mask → block no signal
    sigaction(SIGUSR1,&sa,NULL);
    while(1);
}
```

Esempio: blocking signal

```
$ kill -10 <PID> ; sleep 1
  && kill -12 <PID>
```

```c
#include <signal.h> <unistd.h> <stdlib.h> <stdio.h>  //sigaction3.c
void handler(int signo){
    printf("signal %d received\n",getpid());
    sleep(2);
    printf("Signal done\n");
}
int main(){
    printf("Process id: %d\n",getpid());
    struct sigaction sa;
    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask,SIGUSR2); // Block SIGUSR2 in handler
    sigaction(SIGUSR1,&sa,NULL);
    while(1);
}
```

Esempio: sa_sigaction

```
$ echo $$ ; kill -10 <PID> # custom
$ kill -10 <PID> # default
```

```
#include <signal.h> <unistd.h> <stdlib.h> <stdio.h>  //sigaction4.c
void handler(int signo, siginfo_t * info, void * empty){
    //print id of process issuing the signal
    printf("Signal received from %d\n",info->si_pid);
}
int main(){
    struct sigaction sa;
    sa.sa_sigaction = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags |= SA_SIGINFO; // Use sa_sigaction
    sa.sa_flags |= SA_RESETHAND; // Restore def handler afterward
    sigaction(SIGUSR1,&sa,NULL);
    while(1);
}
```

CONCLUSIONI

I segnali sono uno strumento di comunicazione tra processi molto semplice ma efficace: essendo un metodo molto “antico” non è particolarmente flessibile (essendo nato quando le risorse erano più limitate dei tempi attuali), ma nella sua forma base è comodo e multipiattaforma. L’uso di “signal” è standard, ma alcuni comportamenti possono essere indefiniti, mentre “sigaction” è più affidabile ma meno portabile tra sistemi operativi.

LabSO 2022

Laboratorio Sistemi Operativi - A.A. 2021-2022

dr. Andrea Naimoli	Informatica LT andrea.naimoli@unitn.it
dr. Michele Grisafi	Ingegneria informatica, delle comunicazioni ed elettronica (LT) michele.grisafi@unitn.it

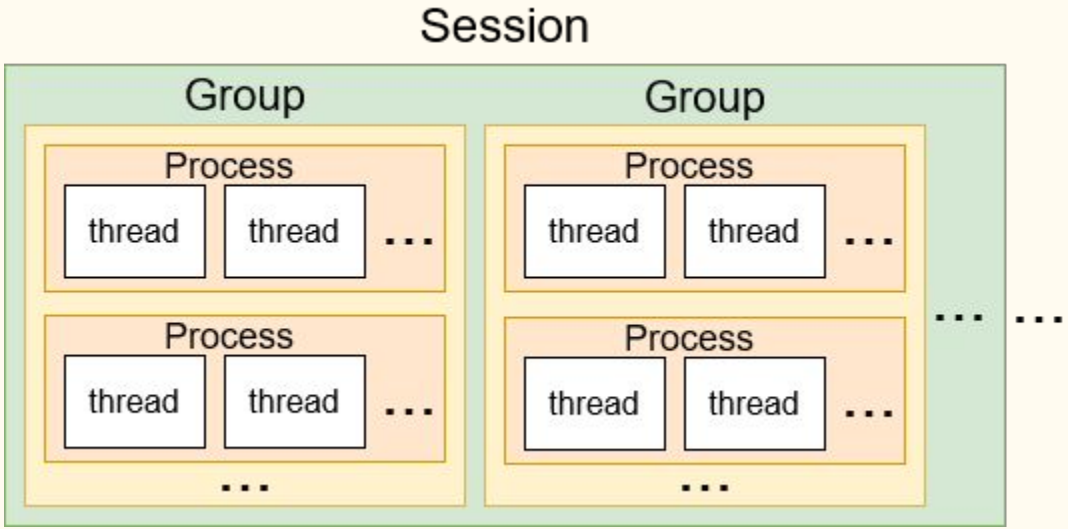
Nota sugli “snippet” di codice

Alcuni esempi di codice possono essere semplificati, ad esempio omettendo il blocco principale con la funzione `main` *(che andrebbe aggiunto) oppure elencando alcune o tutte le librerie da includere tutte su una riga o insieme (per cui invece occorre trascrivere correttamente le direttive* `#include` *secondo la sintassi corretta) o altre semplificazioni analoghe. In questi casi occorre sistemare il codice perché possa essere correttamente compilato e poi eseguito.*

Process groups

Gestione processi in Unix

All’interno di Unix i processi vengono raggruppati secondi vari criteri, dando vita a sessioni, gruppi e threads.



Perchè i gruppi

I process groups consentono una migliore gestione dei segnali e della comunicazione tra i processi.

Un processo, per l'appunto, può:

- Aspettare che tutti i processi figli appartenenti ad un **determinato gruppo** terminino;
- Mandare un segnale a tutti i processi appartenenti ad un **determinato gruppo.**

```
waitpid(-33,NULL,0); // Wait for a children in group 33 (|-33|)
kill(-45,SIGTERM); // Send SIGTERM to all children in group 45
```

Gruppi in Unix

Mentre, generalmente, una sessione è collegata ad un terminale, i processi vengono raggruppati nel seguente modo:

- In bash, processi concatenati tramite pipes appartengono allo stesso gruppo: `cat /tmp/ciao.txt | wc -l | grep ‘2’`
- Alla loro creazione, i figli di un processo ereditano il gruppo del padre
- Inizialmente, tutti i processi appartengono al gruppo di ‘init’, ed ogni processo può cambiare il suo gruppo in qualunque momento.

Il processo il cui PID è uguale al proprio GID è detto *process group leader*.

Group system calls

```
int setpgid(pid_t pid, pid_t pgid); //set GID of proc. (0=self)
pid_t getpgid(pid_t pid); // get GID of process (0=self)
```

```
#include <stdio.h> <unistd.h> <sys/wait.h>                    //setpgid.c
int main(void){
    int isChild = !fork(); //new child
    printf("PID %d PPID: %d GID %d\n",getpid(),getppid(),getpgid(0));
    if(isChild){
        isChild = !fork(); //new child
        if(!isChild) setpgid(0,getpid()); // Become group leader
        sleep(1);
        fork(); //new child
        printf("PID %d PPID: %d GID %d\n",getpid(),getppid(),getpgid(0));
    }; while(wait(NULL)>0);
}
```

Mandare segnali ai gruppi

Nel prossimo esempio:

1. Processo ‘ancestor’ crea un figlio
   a. Il figlio cambia il proprio gruppo e genera 3 figli (Gruppo1)
   b. I 4 processi aspettano fino all’arrivo di un segnale
2. Processo ‘ancestor’ crea un secondo figlio
   a. Il figlio cambia il proprio gruppo e genera 3 figli (Gruppo2)
   b. I 4 processi aspettano fino all’arrivo di un segnale
3. Processo ‘ancestor’ manda due segnali diversi ai due gruppi

Mandare segnali ai gruppi

```
#include <stdio.h><unistd.h><sys/wait.h><signal.h><stdlib.h>//gsignal.c
void handler(int signo){
    printf("[%d,%d] sig%d received\n",getpid(),getpgid(0),signo);
    sleep(1); exit(0);
}
int main(void){
    signal(SIGUSR1,handler);
    signal(SIGUSR2,handler);
    int ancestor = getpid(); int group1 = fork(); int group2;
    if(getpid()!=ancestor ){ // First child
        setpgid(0,getpid()); // Become group leader
        fork();fork(); //Generated 3 children in new group
    }
...
```

```
    else{
        group2 = fork();
        if(getpid()!=ancestor){ // Second child
            setpgid(0,getpid()); // Become group leader
            fork();fork();}} //Generated 3 children in new group
    if(getpid()==ancestor){
        printf("[%d]Ancestor and I'll send signals\n",getpid());
        sleep(1);
        kill(-group2,SIGUSR2); //Send SIGUSR2 to group2
        kill(-group1,SIGUSR1); //Send SIGUSR1 to group1
    }else{
        printf("[%d,%d]chld waiting signal\n", getpid(),getpgid(0));
        while(1);
    }
    while(wait(NULL)>0);
    printf("All children terminated\n");
}
```

Wait figli in un gruppo

Nel prossimo esempio:

1. Processo ‘ancestor’ crea un figlio
   a. Il figlio cambia il proprio gruppo e genera 3 figli (Gruppo1)
   b. I 4 processi aspettano 2 secondi e terminano
2. Processo ‘ancestor’ crea un secondo figlio
   a. Il figlio cambia il proprio gruppo e genera 3 figli (Gruppo2)
   b. I 4 processi aspettano 4 secondi e terminano
3. Processo ‘ancestor’ aspetta la terminazione dei figli del gruppo1
4. Processo ‘ancestor’ aspetta la terminazione dei figli del gruppo2

Wait figli in un gruppo

```
#include <stdio.h><unistd.h><sys/wait.h>                    //waitgroup.c
int main(void){
    int group1 = fork(); int group2;
    if(group1 == 0){ // First child
        setpgid(0,getpid()); // Become group leader
        fork();fork(); //Generated 4 children in new group
        sleep(2); return; //Wait 2 sec and exit
    }else{
        group2 = fork();
        if(group2 == 0){
            setpgid(0,getpid()); // Become group leader
            fork();fork(); //Generated 4 children
            sleep(4); return; //Wait 4 sec and exit
        } ...
```

```
    }
    sleep(1); //make sure the children changed their group
    while(waitpid(-group1,NULL,0)>0);
    printf("Children in %d terminated\n",group1);
    while(waitpid(-group2,NULL,0)>0);
    printf("Children in %d terminated\n",group2);
}
```

CONCLUSIONI

L’organizzazione dei processi in gruppi consente di organizzare meglio le comunicazione e di coordinare le operazioni avendo in particolare la possibilità di inviare dei segnali ai gruppi nel loro complesso.

LabSO 2022

Laboratorio Sistemi Operativi - A.A. 2021-2022

dr. Andrea Naimoli	Informatica LT andrea.naimoli@unitn.it
dr. Michele Grisafi	Ingegneria informatica, delle comunicazioni ed elettronica (LT) michele.grisafi@unitn.it

Nota sugli “snippet” di codice

Alcuni esempi di codice possono essere semplificati, ad esempio omettendo il blocco principale con la funzione `main` *(che andrebbe aggiunto) oppure elencando alcune o tutte le librerie da includere tutte su una riga o insieme (per cui invece occorre trascrivere correttamente le direttive* `#include` *secondo la sintassi corretta) o altre semplificazioni analoghe. In questi casi occorre sistemare il codice perché possa essere correttamente compilato e poi eseguito.*

Errors in C

Gestione errori in C

Durante l'esecuzione di un programma ci possono essere diversi tipi di errori: system calls che falliscono, divisioni per zero, problemi di memoria etc...

Alcuni di questi errori non fatali, come una system call che fallisce, possono essere indagati attraverso la variabile **`errno`**. Questa variabile globale contiene l'ultimo codice di errore generato dal sistema.

Per convertire il codice di errore in una stringa comprensibile si può usare la funzione `char *strerror(int errnum)`.

In alternativa, la funzione `void perror(const char *str)` che stampa su stderr la stringa passatagli come argomento concatenata, tramite ': ', con `strerror(errno)`.

Esempio: errore apertura file

```c
#include <stdio.h> <errno.h>  <string.h>                //errFile.c

extern int errno; // declare external global variable
int main(void){
    FILE * pf;
    pf = fopen ("nonExistingFile.boh", "rb"); //Try to open file
    if (pf == NULL) { //something went wrong!
        fprintf(stderr, "errno = %d\n", errno);
        perror("Error printed by perror");
        fprintf(stderr,"Strerror: %s\n", strerror(errno));
    } else {
        fclose (pf);
    }
}
```

Esempio: errore processo non esistente

```c
#include <stdio.h> <errno.h>  <string.h> <signal.h>     //errSig.c
extern int errno; // declare external global variable
int main(void){
    int sys = kill(3443,SIGUSR1); //Send signal to non existing proc
    if (sys == -1) { //something went wrong!
        fprintf(stderr, "errno = %d\n", errno);
        perror("Error printed by perror");
        fprintf(stderr,"Strerror: %s\n", strerror(errno));
    } else {
        printf("Signal sent\n");
    }
}
```

Pipe anonime

Piping

Il piping connette l'output (stdout e stderr) di un comando all'input (stdin) di un altro comando, consentendo dunque la comunicazione tra i due. Esempio:

```
ls . | sort -R      #stout -> stdin
ls nonExistingDir |& wc       #stdout e stderr -> stdin
cat /etc/passwd | wc | less   #out -> in, out-> in
```

I processi sono eseguiti in **concorrenza** utilizzando un buffer:

- Se pieno lo scrittore (left) si sospende fino ad avere spazio libero
- Se vuoto il lettore si sospende fino ad avere i dati

Esempio

```c
// output.out
#include <stdio.h>
#include <unistd.h>
int main(){
  for (int i = 0; i<3; i++) {
    sleep(2);
    fprintf(stdout,
    "Written in buffer");
    fflush(stdout);
  };
};
```

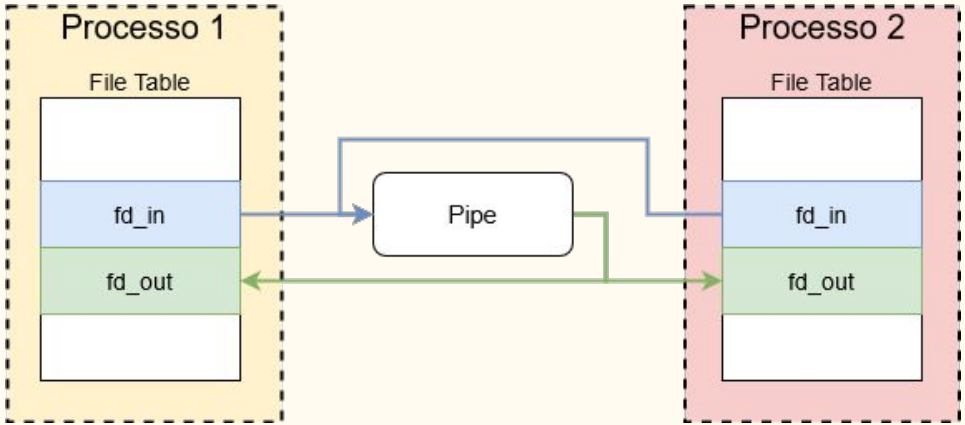
```c
// input.out
#include <stdio.h>
#include <unistd.h>
int main() {
  char msg[50]; int n=3;
  while((n--)>0){
    int c = read(0,msg,50);
    if (c>0) {
        msg[c]=0;
        fprintf(stdout,
        "Read: '%s' (%d)\n",msg,c);
    };
  };
};
```

```
$ ./output.out | ./input.out
```

Pipe anonime

Le pipe anonime, come quelle usate su shell, ‘uniscono’ due processi aventi un antenato comune (oppure tra padre-figlio). Il collegamento è unidirezionale ed avviene utilizzando un buffer di dimensione finita.

Per interagire con il buffer (la pipe) si usano due file descriptors: uno per il lato in scrittura ed uno per il lato in lettura. Visto che i processi figli ereditano i file descriptors, questo consente la comunicazione tra i processi (ma serve l’antenato comune).



Creazione pipe

`int pipe(int pipefd[2]); //fd[0] lettura, fd[1] scrittura`

```
#include <stdio.h>                                                    //pipe.c
#include <unistd.h>
int main(){
    int fd[2], cnt = 0;
    while(pipe(fd) == 0){ //Create unnamed pipe using 2 file descriptors
        cnt++;
        printf("%d,%d,",fd[0],fd[1]);
    }
    printf("\n Opened %d pipes, then error\n",cnt);
    int op = open("/tmp/tmp.txt",O_CREAT|O_RDWR,S_IRUSR|S_IWUSR);
    printf("File opened with fd %d\n",op);
}
```

Lettura pipe: `int read(int fd[0],char * data, int num)`

La lettura della pipe tramite il comando read restituisce valori differenti a seconda della situazione:

- In caso di successo, read() restituisce il numero di bytes effettivamente letti
- Se il lato di scrittura è stato chiuso (da ogni processo), con il buffer vuoto restituisce 0, altrimenti restituisce il numero di bytes letti.
- Se il buffer è vuoto ma il lato di scrittura è ancora aperto (in qualche processo) il processo si sospende fino alla disponibilità dei dati o alla chiusura
- Se si provano a leggere più bytes (`num`) di quelli disponibili, vengono recuperati solo quelli presenti

Esempio lettura pipe

```c
#include <stdio.h>                                          //readPipe.c
#include <unistd.h>
int main(void){
    int fd[2]; char buf[50];
    int esito = pipe(fd); //Create unnamed pipe
    if(esito == 0){
        write(fd[1],"writing",8); // Writes to pipe
        int r = read(fd[0],&buf,50); //Read from pipe
        printf("Last read %d. Received: ‘%s’\n",r,buf);
        // close(fd[1]); // hangs when commented
        r = read(fd[0],&buf,50); //Read from pipe
        printf("Last read %d. Received: ‘%s’\n",r,buf);
    }
}
```

Scrittura pipe: `int write(int fd[1],char * data, int num)`

La scrittura della pipe tramite il comando write restituisce il numero di bytes scritti. Tuttavia, se il lato in lettura è stato chiuso viene inviato un segnale SIGPIPE allo scrittore (default handler quit).

In caso di scrittura, se vengono scritti meno bytes di quelli che ci possono stare (`PIPE_BUF`) la scrittura è “atomica” (tutto assieme), in caso contrario non c’è garanzia di atomicità e la scrittura sarà bloccata (in attesa che il buffer venga svuotato) o fallirà se il flag `O_NONBLOCK` viene usato.

```
int fcntl(int fd, F_SETFL, O_NONBLOCK);
```

Esempio scrittura pipe

```c
#include <unistd.h> <stdio.h> <signal.h> <errno.h> <stdlib.h>
extern int errno;                                    //writePipe.c
void handler(int signo){
    printf("SIGPIPE received\n");   perror("Error");  exit(errno);
}
int main(void){
    signal(SIGPIPE,handler);
    int fd[2]; char buf[50];
    int esito = pipe(fd); //Create unnamed pipe
    close(fd[0]); //Close read side
    printf("Attempting write\n");
    write(fd[1],"writing",8);
    printf("I've written something\n");
}
```

Esempio comunicazione unidirezionale

Un tipico esempio di comunicazione unidirezionale tra un processo scrittore P1 ed un processo lettore P2 è il seguente:

- P1 crea una `pipe()`
- P1 esegue un `fork()` e crea P2
- P1 chiude il lato lettura: `close(fd[0])`
- P2 chiude il lato scrittura: `close(fd[1])`
- P1 e P2 chiudono l'altro fd appena finiscono di comunicare.

Esempio unidirezionale

```
#include <stdio.h> <unistd.h> <sys/wait.h>          //uni.c
int main(){
    int fd[2]; char buf[50];
    pipe(fd); //Create unnamed pipe
    int p2 = !fork();
    if(p2){
        close(fd[1]);
        int r = read(fd[0],&buf,50); //Read from pipe
        close(fd[0]); printf("Buf: ‘%s’\n",buf);
    }else{
        close(fd[0]);
        write(fd[1],"writing",8); // Write to pipe
        close(fd[1]);
    }
    while(wait(NULL)>0);
}
```

Esempio comunicazione bidirezionale

Un tipico esempio di comunicazione bidirezionale tra un processo scrittore P1 ed un processo lettore P2 è il seguente:

- P1 crea due `pipe()`, *pipe1* e *pipe2*
- P1 esegue un `fork()` e crea P2
- P1 chiude il lato lettura di *pipe1* ed il lato scrittura di *pipe2*
- P2 chiude il lato scrittura di *pipe1* ed il lato lettura di *pipe2*
- P1 e P2 chiudono gli altri fd appena finiscono di comunicare.

Esempio bidirezionale

```
#include <stdio.h> <unistd.h> <sys/wait.h>   #define READ 0 #define WRITE 1      //bi.c
int main(){
     int pipe1[2], pipe2[2]; char buf[50];
     pipe(pipe1); pipe(pipe2); //Create two unnamed pipe
     int p2 = !fork();
     if(p2){
          close(pipe1[WRITE]); close(pipe2[READ]);
          int r = read(pipe1[READ],&buf,50); //Read from pipe
          close(pipe1[READ]); printf("P2 received: ‘%s’\n",buf);
          write(pipe2[WRITE],"Msg from p2",12); // Writes to pipe
          close(pipe2[WRITE]);
     }else{
          close(pipe1[READ]); close(pipe2[1]);
          write(pipe1[WRITE],"Msg from p1",12); // Writes to pipe
          close(pipe1[WRITE]);
          int r = read(pipe2[READ],&buf,50); //Read from pipe
          close(pipe2[READ]); printf("P1 received: ‘%s’\n",buf);
     }
     while(wait(NULL)>0);
}
```

Esercizi

- Impostare una comunicazione bidirezionale tra due processi con due livelli di complessità:
  - Alternando almeno due scambi (P1 → P2, P2 → P1, P1 → P2, P2 → P1)
  - Estendendo il caso a mo' di "ping-pong", fino a un messaggio convenzionale di "fine comunicazione"

Gestire la comunicazione

Per gestire comunicazioni complesse c'è bisogno di definire un "protocollo". Esempio:

- Messaggi di lunghezza fissa (magari inviata prima del messaggio)
- Marcatore di fine messaggio (per esempio con carattere NULL o newline)

Più in generale occorre definire la sequenza di messaggi attesi.

Esempio: redirige lo stdout di cmd1 sullo stdin di cmd2

```
#include <stdio.h> <unistd.h> #define READ 0 #define WRITE 1    //redirect.c
int main (int argc, char *argv[]) {
    int fd[2];
    pipe(fd); // Create an unnamed pipe
    if (fork() != 0) { // Parent, writer
        close(fd[READ]); // Close unused end
        dup2(fd[WRITE], 1); // Duplicate used end to stdout
        close(fd[WRITE]); // Close original used end
        execlp(argv[1], argv[1], NULL); // Execute writer program
        perror("connect"); // Should never execute
    } else { // Child, reader
        close(fd[WRITE]); // Close unused end
        dup2(fd[READ], 0); // Duplicate used end to stdin
        close(fd[READ]); // Close original used end
        execlp(argv[2], argv[2], NULL); // Execute reader program
        perror("connect"); // Should never execute
    }
}
```

Pipe con nome/FIFO

FIFO

Le pipe con nome, o FIFO, sono delle pipe che corrispondono a dei file speciali nel filesystem grazie ai quali i processi, senza vincoli di gerarchia, possono comunicare. Un processo può accedere ad una di queste pipe se ha i permessi sul file corrispondente ed è vincolato, ovviamente, dall'esistenza del file stesso.

Le FIFO sono interpretate come dei file, perciò si possono usare le funzioni di scrittura/lettura dei file viste nelle scorse lezioni per interagirvi. Restano però delle pipe, con i loro vincoli e le loro capacità. NB: Non sono dei file: lseek non funziona ed il loro contenuto è sempre vuoto!

Normalmente, aprire una FIFO blocca il processo finchè anche l'altro lato non è stato aperto. Le differenze tra pipe anonime e FIFO sono solo nella loro creazione e gestione.

Creazione FIFO

```
int mkfifo(const char *pathname, mode_t mode);
```

```
#include <sys/stat.h><sys/types.h><unistd.h><fcntl.h><stdio.h>  //fifo.c
int main(void){
    char * fifoName = "/tmp/fifo1";
    mkfifo(fifoName,S_IRUSR|S_IWUSR); //Create pipe if it doesn't exist
    perror("Created?");
    if (fork() == 0){ //Child
        open(fifoName,O_RDONLY); //Open READ side of pipe...stuck!
        printf("Open read\n");
    }else{
        sleep(1);
        open(fifoName,O_WRONLY); //Open WRITE side of pipe
        printf("Open write\n");
    }
}
```

Esempio comunicazione: writer

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

//fifoWriter.c
int main (int argc, char *argv[]) {
    int fd;   char * fifoName = "/tmp/fifo1";
    char str1[80], * str2 = "I'm a writer";
    mkfifo(fifoName,S_IRUSR|S_IWUSR); //Create pipe if it doesn't exist
    fd = open(fifoName, O_WRONLY); // Open FIFO for write only
    write(fd, str2, strlen(str2)+1); // write and close
    close(fd);
    fd = open(fifoName, O_RDONLY); // Open FIFO for Read only
    read(fd, str1, sizeof(str1)); // Read from FIFO
    printf("Reader is writing: %s\n", str1);
    close(fd);
}
```

Esempio comunicazione: reader

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

//fifoReader.c
int main (int argc, char *argv[]) {
    int fd; char * fifoName = "/tmp/fifo1";
    mkfifo(fifoName,S_IRUSR|S_IWUSR); //Create pipe if doesn't exist
    char str1[80], * str2 = "I'm a reader";
    fd = open(fifoName , O_RDONLY); // Open FIFO for read only
    read(fd, str1, 80); // read from FIFO and close it
    close(fd);
    printf("Writer is writing: %s\n", str1);
    fd = open(fifoName , O_WRONLY); // Open FIFO for write only
    write(fd, str2, strlen(str2)+1); // Write and close
    close(fd);
}
```

Pipe anonime vs FIFO

	pipe	**FIFO**
Rappresentazione	Buffer	File
Accesso	2 File descriptors	1 File descriptor
Persistenza	Eliminata alla terminazione di tutti i processi	Esiste finchè esiste il file
Vincoli accesso	Antenato comune	Permessi sul file
Creazione	pipe()	mkfifo()
Max bytes per atomicità	PIPE_BUF = 4096 on Linux, minimo 512 Bytes POSIX	

CONCLUSIONI

La gestione degli errori è fondamentale e occorre coprire tutti i casi “logici” e in particolare verificare che ogni chiamata alle “syscall” non fallisca.

PIPE e FIFO (“named pipes”) sono sistemi di comunicazione tra processi (“parenti”, tipicamente padre-figlio, nel primo caso e in generale nel secondo caso) che consentono scambi di informazioni (messaggi) e sincronizzazione (grazie al fatto di poter essere “bloccanti”).

LabSO 2022

Laboratorio Sistemi Operativi - A.A. 2021-2022

dr. Andrea Naimoli	Informatica LT andrea.naimoli@unitn.it
dr. Michele Grisafi	Ingegneria informatica, delle comunicazioni ed elettronica (LT) michele.grisafi@unitn.it

Nota sugli “snippet” di codice

Alcuni esempi di codice possono essere semplificati, ad esempio omettendo il blocco principale con la funzione `main` *(che andrebbe aggiunto) oppure elencando alcune o tutte le librerie da includere tutte su una riga o insieme (per cui invece occorre trascrivere correttamente le direttive* `#include` *secondo la sintassi corretta) o altre semplificazioni analoghe. In questi casi occorre sistemare il codice perché possa essere correttamente compilato e poi eseguito.*

Queues

Message queues

Una coda di messaggi, message queue, è una lista concatenata memorizzata all'interno del kernel ed identificata con un ID (un intero positivo univoco), chiamato **queue identifier**.

Questo ID viene condiviso tra i processi interessati, e viene generato attraverso una chiave univoca.

Una coda deve essere innanzitutto generata in maniera analoga ad una FIFO, impostando dei permessi. Ad una coda esistente si possono aggiungere o recuperare messaggi tipicamente in modalità "autosincrona": la lettura attende la presenza di un messaggio, la scrittura attende che via sia spazio disponibile. Questi comportamenti possono però essere configurati.

Queue identifier e queue key

Quanto trattiamo di message queue, abbiamo due identificativi:

- **Key**
- **Queue identifier**

Key: intero che identifica un insieme di risorse condivisibili nel kernel, come semafori, memoria condivisa e **code**. Questa chiave univoca deve essere nota a più processi, e viene usata per ottenere il queue identifier.

Queue identifier: id univoco della coda, generato dal kernel ed associato ad una specifica *key*. Questo ID viene usato per interagire con la coda.

Creazione coda

`int msgget(key_t key, int msgflg)`

Restituisce l'identificativo di una coda basandosi sulla chiave "key" e sui flags:

- `IPC_CREAT`: crea una coda se non esiste già, altrimenti restituisce l'identificativo di quella già esistente;
- `IPC_EXCL`: (da usare con il precedente) fallisce se coda già esistente;
- `0xxx`: permessi per accedere alla coda, analogo a quello che si può usare nel file system. In alternativa si possono usare S_IRUSR etc.

```
#include <sys/types.h> <sys/ipc.h> <sys/msg.h>                    //msgget.c
key_t queueKey = 56; //Unique key
int queueId = msgget(queueKey, 0777 | IPC_CREAT | IPC_EXCL);
```

Ottenere chiave univoca

`key_t ftok(const char *path, int id)`

Restituisce una chiave basandosi sul *path* (una cartella o un file), esistente ed accessibile nel file-system, e sull'id numerico. La chiave dovrebbe essere univoca e sempre la stessa per ogni coppia <path,id> in ogni istante sullo stesso sistema.

Un metodo d'uso, per evitare possibili conflitti, potrebbe essere generare un path (es. un file) temporaneo univoco, usarlo, eventualmente rimuoverlo, ed usare l'id per per rappresentare diverse "categorie" di code, a mo' di indice.

```
#include <sys/ipc.h>                                    //ftok.c
key_t queue1Key = ftok("/tmp/unique", 1);
key_t queue2Key = ftok("/tmp/unique", 2); ...
```

Esempio creazione

```
#include <sys/types.h> <sys/ipc.h> <sys/msg.h><stdio.h> //ipcCreation.c
void main(){
    remove("/tmp/unique"); //Remove file
    key_t queue1Key = ftok("/tmp/unique", 1); //Get unique key → fail
    creat("/tmp/unique", 0777); //Create file
    queue1Key = ftok("/tmp/unique", 1); //Get unique key → ok
    int queueId = msgget(queue1Key ,0777 | IPC_CREAT); //Create queue → ok
    queueId = msgget(queue1Key , 0777); //Get queue → ok
    msgctl(queue1Key,IPC_RMID,NULL);  //Remove non existing queue → fail
    msgctl(queueId,IPC_RMID,NULL); //Remove queue → ok
    queueId = msgget(queue1Key , 0777); //Get non existing queue → fail
    queueId = msgget(queue1Key , 0777 | IPC_CREAT); //Create queue → ok
    queueId = msgget(queue1Key , 0777 | IPC_CREAT); //Get queue → ok
    queueId = msgget(queue1Key , 0777 | IPC_CREAT | IPC_EXCL); /* Create
                                    already existing queue -> fail */
}
```

Le queue sono persistenti

```c
#include <sys/ipc.h> <stdio.h> <sys/msg.h>          //persistent.c

void main(){
    key_t queue1Key = ftok("/tmp/unique", 1);
    int queueId = msgget(queue1Key , 0777 | IPC_CREAT | IPC_EXCL);
    perror("Error:");
}
```

Se eseguiamo questo programma dopo aver eseguito il precedente “*ipcCreation.c*” verrà generato un errore dato che la coda esiste già ed abbiamo usato il flag IPC_EXCL!

Comunicazione - 1

```
struct msg_buffer{
    long mtype;
    char mtext[100];
} message;
```

Ogni messaggio inserito nella coda ha:

- Un tipo, categoria, etc... (intero > 0)
- Una grandezza non negativa
- Un payload, un insieme di dati (bytes) di lunghezza corretta

Al contrario delle FIFO, i messaggi in una coda possono essere recuperati anche sulla base del tipo e non solo del loro ordine “assoluto” di arrivo. Così come i files, le code sono delle strutture persistenti che continuano ad esistere, assieme ai messaggi in esse salvati, anche alla terminazione del processo che le ha create. L’eliminazione deve essere esplicita.

Comunicazione - 2

Il payload del messaggio non deve essere necessariamente un campo testuale: può essere una qualsiasi struttura dati. Infatti, un messaggio può anche essere senza payload.

```
typedef struct book{
    char title[10];
    char description[100];
    unsigned short chapters;
} Book;

struct msg_buffer{
    long mtype;
    Book mtext;
} message;

struct msg_empty{
    long mtype;
} message_empty;
```

Inviare messaggi

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

Aggiunge una copia del messaggio puntato da `msgp`, con un payload di dimensione `msgsz`, alla coda identificata da `msquid`. Il messaggio viene inserito immediatamente se c'è abbastanza spazio disponibile, altrimenti la chiamata si blocca fino a che abbastanza spazio diventa disponibile. Se *`msgflg`* è IPC_NOWAIT allora la chiamata fallisce in assenza di spazio.

NB: `msgsz` è la grandezza del payload del messaggio, non del messaggio intero (che contiene anche il tipo)! Per esempio, `sizeof(msgp.mtext)`

Ricevere messaggi - 1

```
ssize_t msgrcv(int msqid,void *msgp,size_t msgsz,long msgtyp,int msgflg)
```

Rimuove un messaggio dalla coda `msqid` e lo salva nel buffer `msgp`. `msgsz` specifica la lunghezza massima del payload del messaggio (per esempio *`mtext`* della struttura `msgp`). Se il payload ha una lunghezza maggiore e `msgflg` è `MSG_NOERROR` allora il payload viene troncato (viene persa la parte in eccesso), se `MSG_NOERROR` non è specificato allora il payload non viene eliminato e la chiamata fallisce.

Se non sono presenti messaggi, la chiamata si blocca in loro attesa. Il flag IPC_NOWAIT fa fallire la syscall se non sono presenti messaggi.

Ricevere messaggi - 2

```
ssize_t msgrcv(int msqid,void *msgp,size_t msgsz,long msgtyp,int msgflg)
```

A seconda di `msgtyp` viene recuperato il messaggio:

- `msgtyp = 0`: primo messaggio della coda (FIFO)
- `msgtyp > 0`: primo messaggio di tipo `msgtyp`, o primo messaggio di tipo diverso da `msgtyp` se `MSG_EXCEPT` è impostato come flag
- `msgtyp < 0`: primo messaggio il cui tipo T è `min(T ≤ |msgtyp|)`

Esempio comunicazione - 1

```c
#include <sys/types.h><sys/ipc.h><sys/msg.h><string.h><stdio.h>  //ipc.c
struct msg_buffer{
    long mtype;
    char mtext[100];
} msgp, msgp2; //Two different message buffers
int main(void){
    msgp.mtype = 20;
    strcpy(msgp.mtext,"This is a message");
    key_t queue1Key = ftok("/tmp/unique", 1);
    int queueId = msgget(queue1Key , 0777 | IPC_CREAT | IPC_EXCL);
    int esito = msgsnd(queueId , &msgp, sizeof(msgp.mtext),0);
    esito = msgrcv(queueId , &msgp2, sizeof(msgp2.mtext),20,0);
    printf("Received %s\n",msgp2.mtext);
}
```

Esempio comunicazione - 2

```
#include <sys/types.h> <sys/ipc.h> <sys/msg.h>  <string.h>  //ipcBook.c

typedef struct book{
    char title[10];
    char description[200];
    short chapters;
} Book;

struct msg_buffer{
    long mtype;
    Book mtext;
} msgp_snd, msgp_rcv; //Two different message buffers
```

Esempio comunicazione - 2

```
    …
int main(void){
    msgp_snd.mtype = 20;
    strcpy(msgp_snd.mtext.title,"Title");
    strcpy(msgp_snd.mtext.description,"This is a description");
    msgp_snd.mtext.chapters = 17;
    key_t queue1Key = ftok("/tmp/unique", 1);
    int queueId = msgget(queue1Key , 0777 | IPC_CREAT);
    int esito = msgsnd(queueId , &msgp_snd, sizeof(msgp.mtext),0);
    esito = msgrcv(queueId , &msgp_rcv, sizeof(msgp_rcv.mtext),20,0);
    printf("Received: %s %s %d\n",msgp_rcv.mtext.title,
        msgp_rcv.mtext.description, msgp_rcv.mtext.chapters);
}
```

Esempio comunicazione - 3

```c
#include <sys/types.h> <sys/ipc.h> <sys/msg.h>  <string.h>  //ipcType.c

struct msg_buffer{
    long mtype;
    char mtext[100];
} mspg_snd,msgp_rcv; //Two different message buffers

int main(int argc, char ** argv){
    int to_fetch = atoi(argv[0]); //Input to decide which msg to get
    key_t queue1Key = ftok("/tmp/unique", 1);
    int queueId = msgget(queue1Key , 0777 | IPC_CREAT);
```

Esempio comunicazione - 3

```
    …
    msgp_snd.mtype = 20;
    strcpy(msgp_snd.mtext,"A message of type 20");
    int esito = msgsnd(queueId , &msgp_snd, sizeof(msgp.mtext),0);
    msgp_snd.mtype = 10; //Re-use the same message
    strcpy(msgp_snd.mtext,"Another message of type 10");
    esito = msgsnd(queueId , &msgp_snd, sizeof(msgp.mtext),0);
    esito = msgrcv(queueId , &msgp_rcv, sizeof(msgp_rcv.mtext),
        to_fetch,0);
    printf("Received: %s %s %d\n",msgp_rcv.mtext.title,
        msgp_rcv.mtext.description, msgp_rcv.mtext.chapters);
}
```

Modificare la coda

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Modifica la coda identificata da `msqid` secondo i comandi `cmd`, riempiendo buf con informazioni sulla coda (ad esempio tempo di ultima scrittura, di ultima lettura, numero messaggi nella coda, etc...). Values for `cmd` are:

- `IPC_STAT`: recupera informazioni da kernel
- `IPC_SET`: imposta alcuni parametri a seconda di `buf`
- `IPC_RMID`: rimuove immediatamente la coda
- `IPC_INFO`: recupera informazioni generali sui limiti delle code nel sistema
- `MSG_INFO`: come IPC_INFO ma con informazioni differenti
- `MSG_STAT`: come IPC_STAT ma con informazioni differenti

https://man7.org/linux/man-pages/man2/msgctl.2.html

msqid_ds structure

```
struct msqid_ds {
    struct ipc_perm msg_perm;   /* Ownership and permissions */
    time_t msg_stime;  /* Time of last msgsnd(2) */
    time_t msg_rtime;  /* Time of last msgrcv(2) */
    time_t msg_ctime; //Time of creation or last modification by msgctl
    unsigned long   msg_cbytes; /* # of bytes in queue */
    msgqnum_t       msg_qnum;   /* # of messages in queue */
    msglen_t        msg_qbytes; /* Maximum # of bytes in queue */
    pid_t           msg_lspid;  /* PID of last msgsnd(2) */
    pid_t           msg_lrpid;  /* PID of last msgrcv(2) */
};
```

ipc_perm structure

```
struct ipc_perm {
    key_t __key;    /* Key supplied to msgget(2) */
    uid_t uid;      /* Effective UID of owner */
    gid_t gid;      /* Effective GID of owner */
    uid_t cuid;     /* Effective UID of creator */
    gid_t cgid;     /* Effective GID of creator */
    unsigned short mode;    /* Permissions */
    unsigned short __seq;   /* Sequence number */
};
```

Esempio modifica coda

```
int main(void){
    struct msqid_ds mod;
    int esito = open("/tmp/unique",O_CREAT,0777);
    key_t queue1Key = ftok("/tmp/unique", 1);
    int queueId = msgget(queue1Key , IPC_CREAT | S_IRWXU  );
    msgctl(queueId,IPC_RMID,NULL);
    queueId = msgget(queue1Key , IPC_CREAT | S_IRWXU  );
    esito = msgctl(queueId,IPC_STAT,&mod); //Get info on queue
    printf("Current permission on queue: %d\n",mod.msg_perm.mode);
    mod.msg_perm.mode= 0000;
    esito = msgctl(queueId,IPC_SET,&mod);  //Modify queue
    printf("Current permission on queue: %d\n\n",mod.msg_perm.mode);
}
```

Esempio 4

```
#include <sys/types.h><sys/ipc.h><sys/msg.h><string.h><stdio.h><unistd.h><wait.h>
//ipc4.c
struct msg_buffer{
    long mtype;
    char mtext[100];
} msgpSND,msgpRCV;
void main(){
    struct msqid_ds mod;
    msgpSND.mtype = 1;
    strcpy(msgpSND.mtext,"This is a message from sender");
    key_t queue1Key = ftok("/tmp/unique", 1);
    int queueId = msgget(queue1Key , 0777 | IPC_CREAT);
    msgctl(queueId,IPC_RMID,NULL); //Remove queue if exists
    queueId = msgget(queue1Key , 0777 | IPC_CREAT); //Create queue
```

```
msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext),0); //Send msg
msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext),0); //Send msg
msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext),0); //Send msg

msgctl(queueId,IPC_STAT,&mod); //Modify queue
printf("Msg in queue: %ld\nCurrent max bytes in queue: %ld\n\n",
        mod.msg_qnum, mod.msg_qbytes);

mod.msg_qbytes = 200; //Change buf to modify queue bytes
msgctl(queueId,IPC_SET,&mod);  //Apply modification

printf("Msg in queue: %ld --> same number\nCurrent max bytes in
        queue: %ld\n\n",mod.msg_qnum, mod.msg_qbytes);

if( fork() != 0 ){ //Parent keep on writing on the queue
    printf("[SND] Sending 4th message with a full queue...\n");
    msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext),0); //Send msg
    printf("[SND] msg sent\n");
```

```
…
        printf("[SND] Sending 5th message with IPC_NOWAIT\n");
        if(msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext),IPC_NOWAIT )
            == -1){ //Send msg
            perror("Queue is full --> Error");
        }
    } else { // Child keeps reading the queue every 3 seconds
        sleep(3); msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext),1,0);
        printf("[Reader] Received msg 1 with msg '%s'\n",msgpRCV.mtext);
        sleep(3); msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext),1,0);
        printf("[Reader] Received msg 2 with msg '%s'\n",msgpRCV.mtext);
        sleep(3); msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext),1,0);
        printf("[Reader] Received msg 3 with msg '%s'\n",msgpRCV.mtext);
        sleep(3); msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext),1,0);
        printf("[Reader] Received msg 4 with msg '%s'\n",msgpRCV.mtext);
```

```
…


        sleep(3);
        if(msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext),1,IPC_NOWAIT)
            == -1){
            perror("Queue is empty --> Error");
        }else{
            printf("[Reader] Received msg 5 with msg '%s'\n",
                    msgpRCV.mtext);
        }
    }
    while(wait(NULL)>0);
}
```

LabSO 2022

Laboratorio Sistemi Operativi - A.A. 2021-2022

dr. Andrea Naimoli	Informatica LT andrea.naimoli@unitn.it
dr. Michele Grisafi	Ingegneria informatica, delle comunicazioni ed elettronica (LT) michele.grisafi@unitn.it

Nota sugli “snippet” di codice

Alcuni esempi di codice possono essere semplificati, ad esempio omettendo il blocco principale con la funzione `main` *(che andrebbe aggiunto) oppure elencando alcune o tutte le librerie da includere tutte su una riga o insieme (per cui invece occorre trascrivere correttamente le direttive* `#include` *secondo la sintassi corretta) o altre semplificazioni analoghe. In questi casi occorre sistemare il codice perché possa essere correttamente compilato e poi eseguito.*

Threads

Threads

I thread sono singole sequenze di esecuzione all'interno di un processo, aventi alcune delle proprietà dei processi. I threads non sono indipendenti tra loro e condividono il codice, i dati e le risorse del sistema assegnate al processo di appartenenza. Come ogni singolo processo, i threads hanno alcuni elementi indipendenti, come lo stack, il PC ed i registri del sistema.

La creazione di threads consente un parallelismo delle operazioni in maniera rapida e semplificata. Context switch tra threads è rapido, così come la loro creazione e terminazione. Inoltre, la comunicazione tra threads è molto veloce.

Per la compilazione è necessario aggiungere il flag `-pthread`, ad esempio:

```
gcc -o program main.c -pthread
```

Creazione

In C i thread corrispondono a delle funzioni eseguite in parallelo al codice principale. Ogni thread è identificato da un ID e può essere gestito come un processo figlio, con funzioni che attendono la sua terminazione.

```
int pthread_create(
    pthread_t *restrict thread, /* Thread ID */
    const pthread_attr_t *restrict attr, /* Attributes */
    void *(*start_routine)(void *), /* Function to be executed */
    void *restrict arg /* Parameters for the above function */
);
```

Esempio creazione

```c
#include <stdio.h> <pthread.h> <unistd.h> //threadCreate.c

void * my_fun(void * param){
    printf("This is a thread that received %d\n", *(int *)param);
    return (void *)3;
}
void main(){
    pthread_t t_id;
    int arg=10;
    pthread_create(&t_id, NULL, my_fun, (void *)&arg);
    printf("Executed thread with id %ld\n",t_id);
    sleep(3);
}
```

Nel kernel...

```
$ ./threadList.out & ps -eLf
```

Light-**W**eight **P**rocess:
LWP: identificativo Thread
NLWP: numero di Threads nel processo

```c
#include <pthread.h> //threadList.c

void * my_fun(void * param){
    while(1);
}
void main(){
    pthread_t t_id;
    pthread_create(&t_id, NULL, my_fun, NULL);
    while(1);
}
```

Terminazione

Un nuovo thread termina in uno dei seguenti modi:

- Chiamando la funzione `noreturn void pthread_exit(void * retval);` specificando un puntatore di ritorno.
- Ritorna dalla funziona associata al thread specificando un valore di ritorno.
- Viene cancellato con `int pthread_cancel(pthread_t thread)`.
- Qualche thread chiama `exit()`, o il thread che esegue `main()` ritorna dallo stesso, terminando così tutti i threads.

Cancellazione di un thread

```
int pthread_cancel(pthread_t thread);
```

Invia una **richiesta** di cancellazione al thread specificato, il quale reagirà (come e quando) a seconda di due suoi attributi: **state** e **type**.

State può essere *enabled* (default) o *disabled*: se *disabled* la richiesta rimarrà in attesa fino a che **state** diventa *enabled*, se *enabled* la cancellazione avverrà a seconda di **type**.

Type può essere *deferred* (default) o *asynchronous*: il primo attende la chiamata di un *cancellation point*, il secondo termina in qualsiasi momento. I *cancellation points* sono funzioni definite nella libreria pthread.h (lista).

Cancellazione di un thread

State e **type** di un thread possono essere modificati solo dal thread stesso con le seguenti funzioni:

```
int pthread_setcancelstate(int state, int *oldstate);
```

con state = `PTHREAD_CANCEL_DISABLE` o `PTHREAD_CANCEL_ENABLE`

```
int pthread_setcanceltype(int type, int *oldtype);
```

Con type = `PTHREAD_CANCEL_DEFERRED` o `PTHREAD_CANCEL_ASYNCHRONOUS`

Esempio cancellazione 1

```c
#include <stdio.h> <pthread.h> <unistd.h>                    //thCancel.c
int i = 1;
void * my_fun(void * param){
    if(i--)
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE,NULL); //Change mode
    printf("Thread %ld started\n",*(pthread_t *)param); sleep(3);
    printf("Thread %ld finished\n",*(pthread_t *)param);
}
int main(void){
    pthread_t t_id1, t_id2;
    pthread_create(&t_id1, NULL, my_fun, (void *)&t_id1); sleep(1); //Create
    pthread_cancel(t_id1); //Cancel
    printf("Sent cancellation request for thread %ld\n",t_id1);
    pthread_create(&t_id2, NULL, my_fun, (void *)&t_id2); sleep(1); //Create
    pthread_cancel(t_id2); //Cancel
    printf("Sent cancellation request for thread %ld\n",t_id2);
    sleep(5); printf("Terminating program\n");
}
```

Esempio cancellazione 2

```c
#include <stdio.h> <pthread.h> <unistd.h>  <string.h>          //thCancel2.c
int tmp = 0;
void * my_fun(void * param){
    pthread_setcanceltype(*(int *)param,NULL); // Change type
    for (long unsigned i = 0; i < (0x9FFF0000); i++); //just wait
    tmp++;
    open("/tmp/tmp",O_RDONLY);  //Cancellation point!
}
int main(int argc, char ** argv){ //call program with 'async' or 'defer'
    pthread_t t_id1; int arg;
    if(!strcmp(argv[1],"async")) arg = PTHREAD_CANCEL_ASYNCHRONOUS;
    else if(!strcmp(argv[1],"defer")) arg = PTHREAD_CANCEL_DEFERRED;
    pthread_create(&t_id1, NULL, my_fun, (void *)&arg); sleep(1); //Create
    pthread_cancel(t_id1); sleep(5); //Cancel
    printf("Tmp %d\n",tmp);
}
```

Aspettare un thread

Un processo (thread) che avvia un nuovo thread può aspettare la sua terminazione mediante la funzione:

```
int pthread_join(pthread_t thread, void ** retval);
```

Questa funzione ritorna quando il thread identificato da `thread` termina, o subito se il thread è già terminato. Se il valore di ritorno del thread non è nullo (parametro di `pthread_exit` o di `return`), esso viene salvato nella variabile puntata da `retval`. Se il thread era stato cancellato, `retval` è riempito con `PTHREAD_CANCELED`.

Solo se il thread è joinable può essere aspettato! Un thread può essere aspettato da al massimo un thread!

Esempio join I

```c
#include <stdio.h> <pthread.h> <unistd.h>                    //thJoin.c
void * my_fun(void * param){
    printf("Thread %ld started\n",*(pthread_t *)param); sleep(3);
    char * str = "Returned string";
    pthread_exit((void *)str); //or 'return (void *) str;'
}
void main(){
    pthread_t t_id;
    void * retFromThread; //This must be a pointer to void!
    pthread_create(&t_id, NULL, my_fun, (void *)&t_id); //Create
    pthread_join(t_id,&retFromThread); // wait thread
    // We must cast the returned value!
    printf("Thread %ld returned '%s'\n",t_id,(char *)retFromThread);
}
```

Esempio join II

```c
#include <stdio.h> <pthread.h> <unistd.h> //threadJoin.c
void * my_fun(void *param){
    printf("This is a thread that received %d\n", *(int *)param);
    return (void *)3;
}
void main(){
    pthread_t t_id;
    int arg=10, retval;
    pthread_create(&t_id, NULL, my_fun, (void *)&arg);
    printf("Executed thread with id %ld\n",t_id);
    sleep(3);
    pthread_join(t_id, (void **)&retval); //A pointer to a void pointer
    printf("retval=%d\n", retval);
}
```

Attributi di un thread

Ogni thread viene creato con degli attributi specificati nella struttura `pthread_attr_t`. Questa struttura è usata solo alla creazione di un thread, ed è poi indipendente dallo stesso (se cambia, gli attributi del thread non cambiano). La struttura va inizializzata e, a fine utilizzo, distrutta. Una volta inizializzati, i vari attributi della struct possono, e devono, essere modificati singolarmente delle funzioni

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_setxxxx(pthread_attr_t *attr, params);
int pthread_attr_getxxxx(const pthread_attr_t *attr, params);
```

Attributi di un thread

- `…detachstate(pthread_attr_t *attr, int detachstate)`
  - `PTHREAD_CREATE_DETACHED` → non può essere aspettato
  - `PTHREAD_CREATE_JOINABLE` → default, può essere aspettato
  - Può essere cambiato durante l'esecuzione con
    `int pthread_detach(pthread_t thread);`
- `...sigmask_np(pthread_attr_t *attr,const sigset_t *sigmask);`
- `...affinity_np(...)`
- `...setguardsize(...)`
- `...inheritsched(...)`
- `...schedparam(...)`
- `...schedpolicy(...)`
- altri

Detached e joinable threads

I threads vengono creati di default nello stato joinable, il che consente ad un altro thread di attendere la loro terminazione attraverso il comando `pthread_join`. I thread joinable rilasciano le proprie risorse non alla terminazione ma quando un thread fa il join con loro (salvando lo stato di uscita) (similmente ai sottoprocessi), oppure alla terminazione del processo. Contrariamente, i thread in stato detached liberano le loro risorse immediatamente una volta terminati, ma non consentono ad altri processi di fare il “join”.

NB: un thread detached non può diventare joinable durante la sua esecuzione, mentre il contrario è possibile.

Esempio attributi

```c
#include <stdio.h> <pthread.h> <unistd.h> //threadAttr.c
void *my_fun(void *param){
    printf("This is a thread that received %d\n", *(int *)param);return (void*)3;
}
void main(){
    pthread_t t_id; pthread_attr_t attr;
    int arg=10, detachSTate;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED); //Set detached
    pthread_attr_getdetachstate(&attr,&detachSTate); //Get detach state
    if(detachSTate == PTHREAD_CREATE_DETACHED) printf("Detached\n");
    pthread_create(&t_id, &attr, my_fun, (void *)&arg);
    printf("Executed thread with id %ld\n",t_id);
    pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE); //Inneffective
    sleep(3); pthread_attr_destroy(&attr);
    int esito = pthread_join(t_id, (void **)&detachSTate);
    printf("Esito '%d' is different 0\n", esito);
}
```

Mutex

Il problema della sincronizzazione

Quando eseguiamo un programma con più thread essi condividono alcune risorse, tra le quali le variabili globali. Se entrambi i thread accedono ad una sezione di codice condivisa ed hanno la necessità di accedervi in maniera esclusiva allora dobbiamo instaurare una sincronizzazione. I risultati, altrimenti, potrebbero essere inaspettati.

Esempio

```
#include <pthread.h> <stdlib.h> <unistd.h><stdio.h>      //syncProblem.c
pthread_t tid[2];
int counter = 0;
void *thr1(void *arg){
    counter = 1;
    printf("Thread 1 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0x00FF0000); i++); //wait some cycles
    counter += 1;
    printf("Thread 1 expects 2 and has: %d\n", counter);
}
void *thr2(void *arg){
    counter = 10;
    printf("Thread 2 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0xFFF0000); i++); //wait some cycles
    counter += 1;
    printf("Thread 2 expects 11 and has: %d\n", counter);
}
...
```

Esempio

```
...
void main(void){
    pthread_create(&(tid[0]), NULL, thr1,NULL);
    pthread_create(&(tid[1]), NULL, thr2,NULL);
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
}
```

Una possibile soluzione: mutex

I mutex sono dei semafori imposti ai thread. Essi possono proteggere una determinata sezione di codice, consentendo ad un thread di accedervi in maniera esclusiva fino allo sblocco del semaforo. Ogni thread the vorrà accedere alla stessa sezione di codice dovrà aspettare che il semaforo sia sbloccato, andando in sleep fino alla sua prossima schedulazione.

I mutex vanno inizializzati e poi assegnati ad una determinata sezione di codice.

Creare e distruggere un mutex

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const
                       pthread_mutexattr_t *restrict attr)

int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

```c
#include <pthread.h>                                    //createMutex.c
pthread_mutex_t lock;
int main(void){
    pthread_mutex_init(&lock, NULL); // Create mutex with default attrs
    pthread_mutex_destroy(&lock); // Destroy mutex (it can be re-init)
}
```

Bloccare e sbloccare un mutex

```
int pthread_mutex_lock(pthread_mutex_t *mutex)

int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

```
#include <pthread.h>
pthread_mutex_t lock;
void * thread(void *){
    pthread_mutex_lock(&lock); /* Waits for mutex to be unlocked, then
                                  locks it becoming the owner */
    pthread_mutex_unlock(&lock); //Unlock mutex allowing others to lock it
}...
```

Esempio

```
#include <pthread.h> <stdlib.h> <unistd.h><stdio.h>      //mutex.c

pthread_mutex_t lock;
pthread_t tid[2];
int counter = 0;

void* thr1(void* arg){
    pthread_mutex_lock(&lock);
    counter = 1;
    printf("Thread 1 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0x00FF0000); i++);
    counter += 1;
    pthread_mutex_unlock(&lock);
    printf("Thread 1 expects 2 and has: %d\n", counter);
}...
```

Esempio

```
...
void* thr2(void* arg){
    pthread_mutex_lock(&lock);
    counter = 10;
    printf("Thread 2 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0xFFF0000); i++);
    counter += 1;
    pthread_mutex_unlock(&lock);
    printf("Thread 2 expects 11 and has: %d\n", counter);
}
int main(void){
    pthread_mutex_init(&lock, NULL);
    pthread_create(&(tid[0]), NULL, thr1,NULL);
    pthread_create(&(tid[1]), NULL, thr2,NULL);
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);
}
```

Tipi di mutex

- `PTHREAD_MUTEX_NORMAL`: no deadlock detection
  - Ribloccare quando bloccato → deadlock
  - Sbloccare quando bloccato da altri → undefined
  - Sbloccare quando sbloccato → undefined
- `PTHREAD_MUTEX_ERRORCHECK`: error checking
  - Ribloccare quando bloccato → error
  - Sbloccare quando bloccato da altri → error
  - Sbloccare quando sbloccato → error
- `PTHREAD_MUTEX_RECURSIVE`: multiple locks
  - Ribloccare quando bloccato → increase lock count → richiede stesso numero di sbloccaggi
  - Sbloccare quando bloccato da altri → error
  - Sbloccare quando sbloccato → error
- `PTHREAD_MUTEX_DEFAULT`:
  - Ribloccare quando bloccato → undefined
  - Sbloccare quando bloccato da altri → undefined
  - Sbloccare quando sbloccato → undefined

Esempio

```
#include <pthread.h> <stdlib.h> <unistd.h><stdio.h>     //recursive.c

pthread_mutex_t lock;
pthread_t tid[2];
int counter = 0;

void* thr1(void* arg){
    pthread_mutex_lock(&lock);
    pthread_mutex_lock(&lock);
    counter = 1;
    printf("Thread 1 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0x00FF0000); i++);
    counter += 1;
    pthread_mutex_unlock(&lock);
    printf("Thread 1 expects 2 and has: %d\n", counter);
    pthread_mutex_unlock(&lock);
}...
```

Esempio

```
...
void* thr2(void* arg){
    pthread_mutex_lock(&lock); pthread_mutex_lock(&lock);
    counter = 10;
    printf("Thread 2 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0xFFF0000); i++);
    counter += 1;
    pthread_mutex_unlock(&lock);  pthread_mutex_unlock(&lock);
    printf("Thread 2 expects 11 and has: %d\n", counter);
}
void main(){
    pthread_mutexattr_t attr;
    pthread_mutexattr_settype(&attr,PTHREAD_MUTEX_RECURSIVE);
    pthread_mutex_init(&lock, &attr);
    pthread_create(&(tid[0]), NULL, thr1,NULL);
    pthread_create(&(tid[1]), NULL, thr2,NULL);
    pthread_join(tid[0], NULL); pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);
}
```

CONCLUSIONI

I “thread” sono una sorta di “processi leggeri” che permettono di eseguire funzioni “in concorrenza” in modo più semplice rispetto alla generazioni di processi veri e propri (forking).

I “MUTEX” sono un metodo semplice ma efficace per eseguire sezioni critiche in processi multithread.

È importante limitare al massimo la sezione critica utilizzando lock/unlock per la porzione di codice più piccola possibile, e solo se assolutamente necessario (per esempio quando possono capitare accessi concorrenti).