**TERMINOLOGIA E CONCETTI BASE**
- KERNEL
- BOOTSTRAP
- SYSTEM CALL
- DUAL-MODE
- LOGIN
- SHELL
- FILESYSTEM
- FILENAME
- PATHNAME
- HOME DIRECTORY
- ROOT DIRECTORY
- WORKING DIRECTORY
- PROGRAMMA
- PROCESSO
- THREAD
- PIPE
- DEADLOCK
- LIVELOCK
- STARVATION

**FILE**
- SERIALIZZAZIONE
- I/O

**DIRETTORI**
- DIRECTORY A UN LIVELLO
- DIRECTORY A DUE LIVELLI
- DIRECTORY AD ALBERO
- DIRECTORY A GRAFO ACICLICO
- DIRECTORY A GRAFO CICLICO

**ALLOCAZIONE**
- ALLOCAZIONE INDICIZZATA - SCHEMA COMBINATO
- FUNZIONI UTILI PER DIRETTORI

**COMANDI UTILI**
- LIST
- COPY, REMOVE, MOVE
- GESTIONE PERMESSI
- VISUALIZZAZIONE TESTO
- DIFFERENZE
- WORD COUNT
- LINK
- GESTIONE ARCHIVI
  - ALTERNATIVE A TAR
- OCCUPAZIONE SPAZIO SU DISCO
- SPELL CHECKER

**STRUMENTI PER LA PROGRAMMAZIONE C**
- COMPILER: GCC
- COMPILER: MAKEFILE
- DEBUGGER: GDB

**ESPRESSIONI REGOLARI**
- LETTERALE
- METACARATTERE
- SEQUENZA DI ESCAPE

**COMANDO FIND**

**FILTRI**
- CUT

- SYSCALL PTHREAD_SELF
- SYSCALL PTHREAD_CREATE
- SYSCALL PTHREAD_EXIT
- SYSCALL PTHREAD_JOIN
- SYSCALL PTHREAD_CANCEL
- SYSCALL PTHREAD_DETACH
- TERMINOLOGIA
  - CONCORRENZA
  - PARALLELISMO

**SHELL**

- PARENTESI (), [], {}
- QUOTING
- HISTORY
- ALIASING

**SCRIPT DI SHELL**

- ESECUZIONE DIRETTA
- ESECUZIONE INDIRETTA
- DEBUG DI UNO SCRIPT
- SINTASSI
  - PARAMETRI
  - VARIABILI
    - VARIABILI LOCALI (O DI SHELL)
    - VARIABILI GLOBALI (O D'AMBIENTE)
  - LETTURA (READ)
  - SCRITTURA (OUTPUT)
  - ESPRESSIONI ARITMETICHE
  - COSTRUTTO CONDIZIONALE IF-THEN-FI
  - COSTRUTTO ITERATIVO FOR-IN
  - COSTRUTTO ITERATIVO WHILE-DO-DONE
  - BREAK, CONTINUE E `:`
  - VETTORI

**SINCRONIZZAZIONE**

- LIFO - STACK
- FIFO - QUEUE - BUFFER CIRCOLARE
- SEZIONI CRITICHE
  - SOLUZIONI SOFTWARE
    - SOLUZIONE 1
    - SOLUZIONE 2
    - SOLUZIONE 3
    - SOLUZIONE 4
  - SOLUZIONI HARDWARE
    - SISTEMI SENZA DIRITTO DI PRELAZIONE
    - SISTEMI CON DIRITTO DI PRELAZIONE
    - MECCANISMI DI LOCK - UNLOCK
  - SOLUZIONI AD-HOC (SEMAFORI)
    - OPERAZIONI STANDARD SU SEMAFORI
      - FUNZIONE INIT
      - FUNZIONE WAIT
      - FUNZIONE SIGNAL
      - FUNZIONE DESTROY
  - IMPLEMENTAZIONE DI UN SEMAFORO
    - SEMAFORI TRAMITE PIPE
      - SEMAPHOREINIT (S)
      - SEMAPHORESIGNAL (S)
      - SEMAPHOREWAIT (S)
    - SEMAFORI POSIX
      - SEM_WAIT
      - SEM_TRYWAIT

# TERMINOLOGIA E CONCETTI BASE

## KERNEL

Si tratta della parte centrale dell'OS. Il compito principale è quello di gestire memoria e processori. Esistono diversi tipi di **kernel**:

- *a livelli o stratificati → costituiti da diversi livelli, il più basso è l'HW*
- *micro-kernel → forniscono solo funzionalità di base*
- *kernel monolitici → utilizzo di driver dispositivi (più comuni)*

## BOOTSTRAP

Programma di inizializzazione. Si occupa di caricare il kernel in memoria centrale all'accensione del computer, e successivamente lo esegue. Il **programma di bootstrap** si trova solitamente in ROM o EEPROM.

## SYSTEM CALL

Forniscono l'interfaccia ai servizi forniti dall'OS (entry-point dell'OS). Offrono solitamente funzionalità "di base" e non possono essere modificate.

## DUAL-MODE

L'OS lavora in **dual-mode**, cioè permette una **user mode** che non possiede tutti i privilegi, complementare alla **kernel mode** che invece possiede privilegi amministrativi. Il **dual-mode** assicura che lo user ***NON*** possa assumere il controllo del computer in *kernel mode*.

# LOGIN

Si tratta della procedura di accesso/autenticazione a un sistema o a una sua applicazione. Solitamente occorre fornire *username* e *password*.

# SHELL

Non fa parte dell'OS. Legge i comandi utente e li esegue. Può eseguire:

- *comandi da terminale*
- *comandi da file eseguibili (script)*

# FILESYSTEM

Struttura gerarchica a grafo aciclico in cui sono organizzati:

- *direttori (directory)*
- *file*

# FILENAME

Nome dei file. In UNIX gli unici caratteri non utilizzabili in un **filename** sono:

- *lo slash "/"*
- *il carattere "null"*

# PATHNAME

Sequenza di nomi separati da slash "/". Possono essere specificati in maniera:

- *assoluta*
- *relativa*

Caratteri particolari:

- *"."* → *indica la directory corrente*
- *".."* → *indica la directory padre*

# HOME DIRECTORY

Directory a cui si accede dopo il login. Contiene il materiale dello user loggato.

# ROOT DIRECTORY

Directory principale. E' la radice dell'albero directory, si tratta del punto di origine per interpretare i path assoluti.

# WORKING DIRECTORY

E' il punto di origine per interpretare i path relativi. Ci si riferisce automaticamente qualora non si specifichi un path.

# PROGRAMMA

File eseguibile che risiede su disco. Specifica una serie di **operazioni** per realizzare un procedimento definito (**algoritmo**). Un'operazione si dice **atomica** se nessun processore può interromperla. I programmi sono divisi in:

- *programma sequenziale → operazioni da eseguire in sequenza (ogni nuova istruzione inizia al termine della precedente)*
- *programma concorrente o parallelo → individua operazioni che possono procedere in parallelo (ogni operazione può essere eseguita senza attendere il completamento della precedente)*

## PROCESSO

Si tratta di un programma in esecuzione. E' un'entità attiva.

## THREAD

Un processo può avere al suo interno uno o più flussi di controllo in esecuzione. Ciascun flusso di esecuzione è un **thread**.

## PIPE

Una pipe è un **flusso dati tra due processi**.

## DEADLOCK

Un insieme di entità attendono il verificarsi di un evento che può essere causato solo da un'altra entità dell'insieme.

## LIVELOCK

Situazione simile al deadlock in cui le entità non sono effettivamente bloccate ma non fanno alcun progresso (quello che solitamente definiamo loop infinito).

## STARVATION

A un'entità viene ripetutamente rifiutato l'accesso a una risorsa necessaria al suo progresso.

N.B.:
- starvation NON IMPLICA deadlock
- deadlock IMPLICA starvation

# FILE

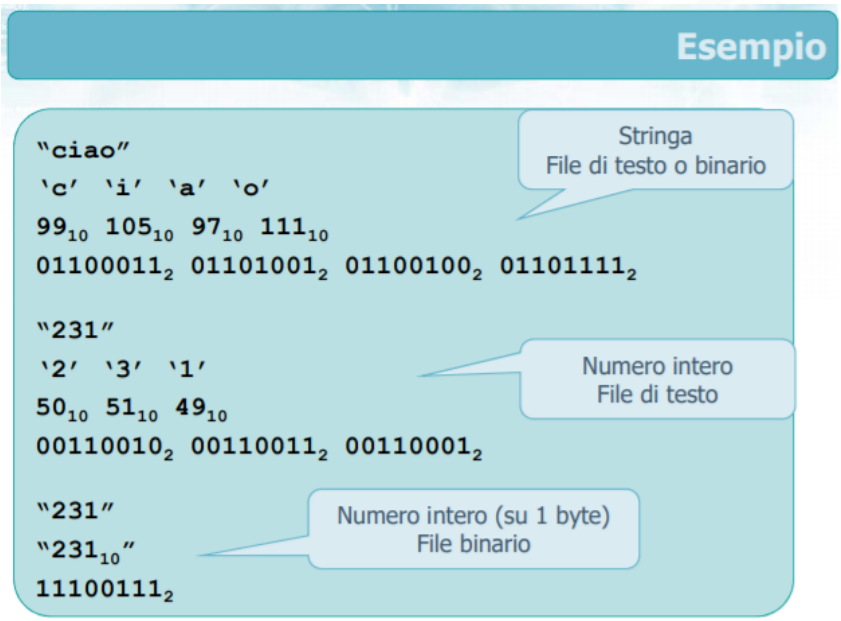
Dal punto di vista logico un **file** può essere visto come:

- *insieme di informazioni correlate*
- *uno spazio di indirizzamento contiguo*

Normalmente i file si distinguono in:

- *file di testo (o ASCII)*
    - Vantaggi:
        - *portabilità*

        - *possibilità di utilizzare editor standard*
    - Svantaggi:
        - *dimensione media relativamente alta*
- *file binari*
    - Vantaggi:
        - *minore dimensione media (compattezza)*
        - *facilità di posizionarsi e modificare il file*
    - Svantaggi:
        - *portabilità limitata*
        - *impossibilità di utilizzare editor standard*




# SERIALIZZAZIONE

Si tratta di un processo di traduzione di una struttura in un formato memorizzabile. Utilizzando la **serializzazione** la struttura può essere memorizzata o trasmessa come un'unica entità. La lettura della sequenza di fa in accordo con la serializzazione effettuata così da poter ricostruire la strutta in maniera identica a quella di partenza.

# I/O

L'**I/O ANSI C** può avvenire in diversi modi:

- *un carattere alla volta* → ***getchar()***, ***putchar()***
- *una riga alla volta* → ***gets()***, ***puts()***
- *I/O formattato* → ***printf()***, ***scanf()***
- *R/W diretto* → ***fread()***, ***fwrite()***

L'**I/O UNIX** si può effettuare interamente mediante solo 5 funzioni:

- ***open()*** → *apre un file dato il path, definendone modalità di accesso e permessi*
- ***read()*** → *legge dal file* ***fd*** *un numero di bytes pari a* ***nbytes***, *memorizzandoli in* ***buf***
- ***write()*** → *scrive* ***nbytes*** *byte contenuti in* ***buf*** *nel file descrittore* ***fd***
- ***lseek()*** → *ogni file ha associata una posizione corrente del file offset. La funzione lseek() assegna un nuovo valore (****offset****) al file offset*
- ***close()*** → *chiude il file descrittore* ***fd***

# DIRETTORI

I file sono organizzati in direttori. Un direttorio è un nodo (o vertice) contenente informazioni sugli elementi in esso contenuti. Su un direttorio si possono effettuare operazioni simili a quelle effettuabili sui file (creazione, cancellazione, ricerca, ecc.).

La struttura di un direttorio dipende da ragioni di:

- *efficiency (efficienza) → es. velocità nel localizzare un file*
- *naming (convenienza) → es. evitare che lo stesso nome attribuito a più file crei problemi*
- *grouping (organizzazione) → es. raggruppare le informazioni in base alle relative caratteristiche*

# DIRECTORY A UN LIVELLO

I file sono contenuti all'interno dello stesso (unico) direttorio.

In termini di prestazioni, con questa organizzazione:

- *Efficiency*
  - *struttura facilmente comprensibile e gestibile*
  - *gestione del file system semplice ed efficiente*
- *Naming*
  - *i file devono avere nomi univoci (problema con grandi quantità di file)*
- *Grouping*
  - *complessa gestione dei file di un singolo utente*
  - *impossibilità di gestire utenti multipli*

# DIRECTORY A DUE LIVELLI

Ogni utente può avere il proprio direttorio (a un livello).

Prestazioni in questo caso:

- *Efficiency*
  - *visione del file system user-oriented*
  - *ricerche efficienti agendo su singoli utenti*
- *Naming*
  - *possibilità di avere file con lo stesso nome purché appartenenti a diversi utenti (specificare path-name per ogni file)*
- *Grouping*
  - *semplificato tra diversi utenti*
  - *complesso per ciascun utente*

# DIRECTORY AD ALBERO

I file sono contenuti in un albero. Ogni utente può gestire più directory e subdirectory.

Prestazioni con questa organizzazione:

- *Efficiency*
  - *ricerche vincolate alla struttura ad albero (quindi profondità e ampiezza)*
- *Naming*

    - *permesso in maniera estesa*
- *Grouping*
    - *permesso in maniera estesa*

## DIRECTORY A GRAFO ACICLICO

Si inizia ad intravedere il concetto di **link**, cioè la possibilità di riferirsi allo stesso file con due nomi diversi o in due directory diverse. In particolare, così è possibile la condivisione di informazioni tra utenti diversi. **La presenza di link aumenta la difficoltà di gestione del file system in quanto occorrerà distinguere gli oggetti nativi dai relativi collegamenti, in fase di *creazione*, *manipolazione* e *cancellazione*.**

## DIRECTORY A GRAFO CICLICO

A differenza delle directory a grafo aciclico, in questo caso viene permessa la creazione di cicli. **E' importante gestire opportunamente i cicli esistenti in tutte le fasi.**

# ALLOCAZIONE

Con **allocazione** si intendono tutte le tecniche di utilizzo dei blocchi dei dischi per la memorizzazione di file. Esistono 3 tecniche principali:

- *contigua (contiguous) → ogni file occupa un insieme contiguo di blocchi*
    - *Vantaggi:*
        - *strategia di allocazione molto semplice*
        - *permette accessi sequenziali immediati*
        - *permette accessi diretti semplici*
    - *Svantaggi:*
        - *occorre decidere una politica di allocazione (first-fit, best-fit, ecc.)*
        - *nessun algoritmo di allocazione risulta privo di difetti, quindi la tecnica sprecherà spazio (**frammentazione esterna**)*
        - *problemi di allocazione dinamica (se il file cresce, potrebbe non entrare più nello spazio allocato)*
- *concatenata (linked) → ogni file può essere allocato gestendo una lista concatenata di blocchi*
    - *Vantaggi:*
        - *permette allocazione dinamica*
        - *elimina frammentazione esterna*
        - *evita l'utilizzo di algoritmi di allocazione complessi*
    - *Svantaggi:*
        - *ogni lettura implica un accesso sequenziale ai blocchi*
        - *un accesso diretto richiederebbe percorrere la catena di puntatori fino a raggiungere l'indirizzo desiderato*
- *indicizzata (indexed) → per permettere un accesso diretto è possibile inglobare tutti i puntatori in una tabella di puntatori (**blocco indice** o **index-node** o **i-node**). Ogni file ha la sua tabella, ovvero un vettore di indirizzi dei blocchi in cui il file è contenuto*

## ALLOCAZIONE INDICIZZATA - SCHEMA COMBINATO

Lo **schema combinato** è utilizzato nei sistemi UNIX/Linux. A ogni file è associato un blocco i-node contenente diverse informazioni, tra cui 15 puntatori ai blocchi dati del file. In particolare:

- *i primi 12 puntatori sono **diretti**, ovvero puntano a blocchi dei file*
- *i puntatori 13, 14, 15 sono **indiretti** con livello di indirizzamento crescente (**puntatori a puntatori nel blocco individuato dal puntatore 13**, o ancora **puntatori a puntatori a puntatori nel blocco individuato dal puntatore 14**, ecc.)*

In questa configurazione, abbiamo:

- ***hard link** → link effettivo o fisico (attenzione: se modifico l'hard link, modifico anche il file nativo)*
- ***soft-link** → link simbolico, si tratta di un file contenente nel suo i-node il riferimento al file nativo*

## FUNZIONI UTILI PER DIRETTORI

- *stat → permette di capire di che tipo di "entry" si tratta (directory, file, link, ecc.)*
- *getcwd, chdir → ottiene/modifica il path della working directory*
- *mkdir, rmdir → crea/cancella una directory*
- *opendir, readdir, closedir → funzioni di visita*

# COMANDI UTILI

## LIST

```
ls [ opzioni ] [ file... ]
```

Opzioni:

| Compatto | Esteso | Effetto |
|---|---|---|
| | --help | *help in linea* |
| -a | --all | *elenca anche i file che iniziano per .* |
| -l | | *output con formato esteso* |
| -g | --group-directories-first | *include l'indicazione del gruppo prima di quella del file* |
| -t | | *elenca i file in ordine temporale (prima il più recente)* |
| -r | --reverse | *ordine inverso (alfabetico o temporale)* |
| -R | --recursive | *elenca anche i file nei sottodirettori* |

## COPY, REMOVE, MOVE

```
cp [ opzioni ] src1 src2 ... dest
rm [ opzioni ] src1 src2 ...
mv [ opzioni ] src1 src2 ... dest
```

Opzioni:

| Compatto | Esteso | Effetto |
|---|---|---|
| | --help | *help in linea* |
| -f | --force | *effettua le operazioni senza chiederne conferma* |
| -i | --interactive | *chiede conferma prima di effettuare qualsiasi operazione* |
| -r, -R | --recursive | *procede ricorsivamente anche nei sottodirettori* |

# GESTIONE PERMESSI

```
chmod [ opzioni ] permessi file
```

Opzioni:

| Compatto | Esteso | Effetto |
|---|---|---|
| -r, -R | --recursive | *procede ricorsivamente anche nei sottodirettori* |

# VISUALIZZAZIONE TESTO

```
cat file1 file2 ...
head [ opzioni ] file ...
tail [ opzioni ] file
```

Opzioni:

| Compatto | Esteso | Effetto |
|---|---|---|
| -l | --lines | *specifica il numero di righe* |
| -f | --follow | *rilegge in loop il file aggiornando l'output se il file viene modificato* |

Altri comandi di visualizzazione:

```
pg [ opzioni ] file
more [ opzioni ] file
less [ opzioni ] file
```

Opzioni:

| Compatto | Esteso | Effetto |
|---|---|---|
| spazio | | *prossima riga* |
| return | | *prossima riga* |
| B | | *pagina precedente* |
| /str | | *ricerca nel testo la prossima occorrenza di str* |
| ?str | | *ricerca nel testo la precedente occorrenza di str* |
| q | | *termina la visualizzazione* |

# DIFFERENZE

```
diff [ opzioni ] entry1 entry2
```

Opzioni:

| Compatto | Esteso | Effetto |
|---|---|---|
| -q | --brief | *indica solo se gli oggetti sono differenti* |
| -b | --ignore-space-change | *ignora gli spazi a fine riga, collassa gli altri* |
| -i | --ignore-case | *ignora la differenza tra maiuscole e minuscole* |
| -w | --ignore-all-space | *ignora completamente ogni tipo di spaziatura* |
| -B | --ignore-blank-lines | *ignora le righe di soli spazi* |

# WORD COUNT

```
wc [ opzioni ] [ file ]
```

Opzioni:

| Compatto | Esteso | Effetto |
|---|---|---|
| -c | --bytes | *valuta il numero di soli byte* |
| -m | --chars | *valuta il numero di soli byte* |
| -w | --words | *valuta il numero di parole* |
| -l | --lines | *valuta il numero di righe* |

# LINK

```
ln [ opzioni ] source [ destination ]
```

Opzioni:

| Compatto | Esteso | Effetto |
|---|---|---|
| | --help | *help in linea* |
| -s | --symbolic | *crea un link simbolico (soft link)* |
| -f | --force | *rimuove eventuali file di destinazione esistenti* |
| -d, -F | --directory | *permette al SU di provare a creare un hard-link con un direttorio* |

# GESTIONE ARCHIVI

Archiviazione e compressione del direttorio **dir**:

```
tar -czvf < file >.tgz < dir >
```

Estrazione del contenuto dell'archivio **<file>.tgz**:

```
tar -xzvf < file >.tgz < dir >
```

Opzioni:

| Compatto | Esteso | Effetto |
|---|---|---|
| -c | | *crea l'archivio* |
| -x | | *estrae l'archivio* |
| -z, -j, -J | | *comprime (gzip, bzip2, 7z)* |
| -f | | *specifica il nome dell'archivio* |
| -v | | *verbose (stampa i messaggi)* |

## ALTERNATIVE A TAR

- *gzip, gunzip*
- *zip, unzip*
- *rar, unrar*
- *compress*

# OCCUPAZIONE SPAZIO SU DISCO

```
df [ opzioni ] disco
```

Si usa per controllare l'occupazione dei dischi.

```
du [ opzioni ] direttorio
```

Si usa per ottenere lo spazio occupato da una directory e tutte le sue sottodirectory.

# SPELL CHECKER

Check sullo spelling dei vocaboli con successiva lista dei suggerimenti (correttore).

```
aspell [ opzioni ] -c file
```

# STRUMENTI PER LA PROGRAMMAZIONE C

## COMPILER: GCC

```
gcc < opzioni > < argomenti >
```

Opzioni:

| Compatto | Esteso | Effetto |
|---|---|---|
| -c file | | *esegue la compilazione non il linker* |
| -o file | | *specifica il nome di output; in genere indica il nome dell'eseguibile finale (linkando)* |
| -g | | *indica a gcc di non ottimizzare il codice e di inserire informazioni extra per poter effettuare il debugging* |
| -Wall | | *stampa warning per tutti i possibili errori nel codice* |
| -Idir (è una i maiuscola) | | *specifica ulteriori direttori in cui cercare gli header file. Esempio:* ***gcc -c -I/home/me/development/ecc sample.c*** |
| -lm (è una L minuscola) | | *specifica utilizzo libreria matematica* |
| -Ldir | | *specifica direttori per ricercare librerie preesistenti* |

## COMPILER: MAKEFILE

Makefile ha 2 scopi principali:

- *effettuare operazioni ripetitive*
- *evitare di (ri)fare operazioni inutili come ricompilare file non modificati*

Si procede in 2 fasi:

- *si scrive un file Makefile*
- *si interpreta il file con l'utility* ***make***

Opzioni:

| Compatto | Esteso | Effetto |
|---|---|---|
| -n | | *non esegue i comandi ma li stampa solo* |
| -i | --ignore-errors | *ignora gli eventuali errori e va avanti* |
| -d | | *stampa informazioni di debug durante l'esecuzione* |
| | --debug=[options] | *Opzioni:*<br>***a = print all info***<br>***b = basic info***<br>***v = verbose = basic + altro***<br>***i = implicit = verbose + altro*** |

Ogni Makefile include:

- ***righe bianche*** → *vengono ignorate*
- ***righe che iniziano per '#'*** → *sono commenti e vengono ignorati*
- ***righe che specificano regole*** → *ogni regola specifica un obiettivo, delle dipendenze e delle azioni e occupa una o più righe. Righe molto lunghe possono essere spezzate inserendo il carattere '\' a fine riga*

## DEBUGGER: GDB

Si tratta di un pacchetto software utilizzato per analizzare il comportamento di un altro programma allo scopo di individuare ed eliminare eventuali errori (bug). Viene spesso utilizzato in maniera integrata con molti editor (emacs, Code::Blocks, ecc.).

# ESPRESSIONI REGOLARI

Una **espressione regolare** (o **pattern**) è una espressione utilizzata per specificare un insieme di stringhe. Vengono utilizzate per effettuare l'accoppiamento (**match**) tra oggetti (nomi di direttori, nomi di file, righe o campi di file, ecc.).

## LETTERALE

Qualsiasi carattere (o sequenza di caratteri) utilizzato nella riceerca del match. Esempio: **ind** in w**ind**ows, **ind**ifferent, ecc.

## METACARATTERE

Uno o più caratteri con significato speciale. Esempio: ***** indica da 0 a ∞ → **b* = {Ø, b, bb, bbb, ...}**

## SEQUENZA DI ESCAPE

Metodo per indicare che un metacarattere deve essere utilizzato come letterale. Esempio: per utilizzare il '**.**' bisogna digitare '**\.**'

# COMANDO FIND

```
find direttorio [ opzioni ] [ azioni ]
```

I passi che vengono effettuati:

1. *visita tutto l'albero a partire dal direttorio **direttorio***
2. *crea l'elenco che soddisfa le **opzioni***
3. *eventualmente effettua per ogni file le **azioni** specificate*

# FILTRI

E' un comando che:

1. riceve il proprio input da standard input
2. lo manipola (lo filtra) secondo determinati parametri e opzioni
3. produce il suo output su standard output

## CUT

Rimuove sezioni specifiche di ogni riga del file indicato.

```
cut [ opzioni ] file
```

Opzioni:

| Compatto | Esteso | Effetto |
|---|---|---|
| -c LIST | --characters=LIST | *seleziona solo i caratteri di posizione indicata* |
| -f LIST | --fields=LIST | *indica la lista dei campi da selezionare (separati da virgola).*<br>*Formato:*<br>***n (=n)***<br>***-n (≤n)***<br>***n- (≥n)***<br>***n1-n2 (≥n1 && ≤n2)*** |
| -d DELIM | --delimiter=DELIM | *usa DELIM per dividere i campi (default: TAB)* |

## TR

Copia lo stdin nello stdout effettuando le sostituzioni oppure le cancellazioni specificate.

```
tr [ opzioni ] set1 [ set2 ]
```

Opzioni:

| Compatto | Esteso | Effetto |
|---|---|---|
| -c, -C | --complement | *utilizza il complement del **set1*** |
| -d | --delete | *cancella i caratteri indicati nel **set1*** |
| -s | --squeeze-repeats | *sostituisce ogni sequenza di un carattere ripetuto incluso nel **set2** con una occorrenza singola dello stesso carattere* |

# UNIQ

Riporta oppure elimina le righe ripetute nel file in ingresso.

```
uniq [ opzioni ] [ inFile ] [ outFile ]
```

Opzioni:

| Compatto | Esteso | Effetto |
|---|---|---|
| -c | --count | *stampa il numero di ripetizioni prima della riga* |
| -d | --repeated | *visualizza solo le righe ripetute* |
| -f N | --skip-fields=N | *ignora i primi N campi per il confronto* |
| -I (è una i maiuscola) | --ignore-case | *case insensitive* |

# BASENAME

Elimina il direttorio (path) e suffisso (estensione) da un nome di file.

```
basename nome [ estensione ]
```

# SORT

Ordina i file in input in ordine alfabetico.

```
sort [ opzioni ] [ file ]
```

Opzioni:

| Compatto | Esteso | Effetto |
|---|---|---|
| -b | --ignore-leading-blanks | *ignora gli spazi iniziali* |
| -d | --dictionary-order | *considera solo spazi e caratteri alfabetici* |
| -f | --ignore-case | *trasforma caratteri minuscoli in maiuscoli* |
| -i | --ignore-nonprinting | *considera solo caratteri stampabili* |
| -n | --numeric-sort | *confronta utilizzando un ordine numerico* |
| -r | --reverse | *ordine inverso* |
| -k c1[,c2] | --key=c1[,c2] | *ordina sulla base dei soli campi selezionati* |
| -m | --merge | *merge più file ordinati (senza riordinare)* |
| -o=f | --output=f | *scrive l'output nel **file f** invece che su **stdout*** |

# GREP

***Global Regular Expression Print***

Cerca nel contenuto dei file di ingresso le righe che hanno un **match** con il pattern fornito e le visualizza su stdout.

```
grep [ opzioni ] pattern [ file ]
```

Opzioni:

| Compatto | Esteso | Effetto |
|---|---|---|
| -e PATTERN | --regexp=PATTERN | *specifica i pattern da ricercare; permette di specificare pattern multipli* |
| -B N | --before-context=N | *prima di ciascun match stampa N righe (oltre alla riga in cui si è verificato il match).*<br>*Inserisce un separatore (--) dopo ogni insieme stampato.* |
| -A N | --after-context=N | *dopo ciascun match stampa N righe (oltre alla riga in cui si è verificato il match).*<br>*Inserisce un separatore (--) dopo ogni insieme stampato.* |
| -H | --with-filename | *stampa il nome del file per ogni match* |
| -i | --ignore-case | *case insensitive* |
| -n | --line-number | *stampa il numero di riga del match* |
| -r, -R | --recursive | *procede in maniera ricorsiva sul sottoalbero* |
| -v | --inverse-match | *stampa solo le righe che non fanno match* |

# PROCESSI

Termini importanti:

- *algoritmo → procedimento logico che, in un numero finito di passi, permette la soluzione di un problema*
- *programma → formalizzazione di un algoritmo attraverso un linguaggio di programmazione; è un'entità passiva, ovvero un file (eseguibile) su disco.*
- *processo → astrazione di un programma in esecuzione; è un'entità attiva.*

## PROCESSI SEQUENZIALI

Le azioni sono eseguite una dopo l'altra, cioè ogni nuova istruzione inizia una volta terminata la precedente. Dato uno stesso input, si genera sempre lo stesso output, indipendentemente da:

- *momento di esecuzione*
- *velocità di esecuzione*
- *quanti altri processi sono in esecuzione sul sistema*

## PROCESSI CONCORRENTI

Più istruzioni possono essere eseguite allo stesso istante. La concorrenza è:

- *fittizia (illusoria) → nei sistemi mono-processore*
- *reale (parallelismo) → nei sistemi multi-processore o multi-core*

# CARATTERISTICHE

Normalmente è possibile:

- *identificare e controllare un processo esistente*
- *creare un nuovo processo → il processo creante assume il ruolo di **processo padre** e quello creato di **processo figlio***
- *attendere, sincronizzare e terminare processi esistenti*

# IDENTIFICAZIONE DI UN PROCESSO

Ogni processo possiede un identificatore univoco **PID (Processor IDentifier)**. Normalmente si tratta di un numero intero non negativo. *(Oltre al PID, ad ogni processo viene associato un GID, cioè l'identificativo del gruppo sotto cui sta girando il processo e un UID, cioè l'identificativo dell'utente sotto cui sta girando il processo)*

Alcuni PID sono riservati:

- *0 → riservato per lo schedulatore dei processi (**swapper**), eseguito a livello kernel*
- *1 → riservato per il processo di **init**. Si tratta di un processo che non muore mai, eseguito con privileggi di super-user, che diventa padre di ogni processo rimasto orfano*

# CREAZIONE DI UN PROCESSO

In base all'OS in uso, si utilizzano procedure differenti:

- Windows API → un nuovo processo viene creato mediante la syscall **CreateProc*ess***
- *UNIX/Linux → un nuovo processo viene create mediante la syscall **fork***

## RISORSE DI UN PROCESSO

Le risorse possono:

- *essere condivise completamente tra padre e figli → stesso spazio di indirizzamento*
- *essere condivise in parte → spazi di indirizzamento parzialmente sovrapposti*
- *non essere condivise → spazi di indirizzamento separati*

In UNIX/Linux padre e figlio **condividono**:

- *il codice sorgente (C)*
- *tutti i descrittori dei file*
- *lo user ID, il group ID, ecc.*
- *la root e la working directory*
- *le risorse del sistema e i limiti di utilizzo*
- *i segnali*

In UNIX/Linux padre e figlio si **differenziano** per:

- *il valore ritornato dalla fork*
- *il PID*
- *Lo spazio dati, lo heap e lo stack (**N.B. il valore delle variabili viene ereditato**)*

# TERMINAZIONE DI UN PROCESSO

Esistono 5 metodi standard per terminare un processo:

- *eseguire una **return** dalla funzione principale*
- *eseguire una **exit***

- *eseguire una **_exit** oppure una **_Exit** (effetti simili alla **exit**, ma non identici)*
- *richiamare **return** dal main dell'ultimo thread del processo*
- *richiamare **pthread_exit** dall'ultimo thread del processo*

Esistono 3 metodi anomali per terminare un processo:

- *richiamare la funzione **abort***
- *ricevere un segnale (**signal**), di terminazione*
- *cancellare l'ultimo thread del processo*

Quando un processo termina, in maniera normale o anomala:

1. *il kernel invia un segnale (**SIGCHLD**) al padre*
2. *la ricezione di un segnale da parte di un processo è un evento asincrono*
3. *il processo padre può decidere di:*
    - *gestire la terminazione del figlio in maniera:*
        - ***asincrona***
        - ***sincrona***
    - *ignorare la terminazione del figlio*

Nel caso in cui un processo decida di gestire la terminazione di un figlio, occorre effettuarne la gestione:

- *asincrona → mediante un gestore del segnale **SIGCHLD***
- *sincrona → mediante una chiamata alle syscall **wait**, **waitpid***

## SYSCALL WAIT

```
pid_t wait (int *statLoc);
```

La chiamata a **wait** da parte di un processo ha effetti diversi a seconda dello stato dei processi figli del processo chiamante:

- *ritorna con un errore se il processo non ha figli (-1)*
- *blocca il processo se tutti i figli del processo sono ancora in esecuzione; alla terminazione di un figlio, la funzione **wait** ritornerà al chiamante lo stato del figlio terminato*
- *resituisce al processo (immediatamente) lo stato di terminazione di un figlio, se **almeno** uno dei figli è terminato ed è in attesa che il suo stato di terminazione sia recuperato*

Parametro:

- ***statLoc** → indica lo stato di terminazione del processo figlio terminato. E' un puntatore ad intero. Le informazioni di stato sono interpretabili con delle macro presenti in <sys/wait.h>. Esempio:*
    - ***WIFEXITED(statLoc)** è vero se la terminazione è stata corretta*
    - ***WEXITSTATUS(statLoc)**, se la terminazione è stata corretta, cattura gli 8 LSBs del parametro passato alla chiamata di terminazione (exit, _exit o _Exit)*

Valore di ritorno:

- *il PID del processo figlio terminato*

## SYSCALL WAITPID

Se si desidera attendere un figlio specifico con una **wait**, occorre:

- *controllare il PID del figlio terminato*
- *eventualmente memorizzarlo nella lista dei processi figlio terminati (per future verifiche/ricerche)*

- *effettuare un'altra **wait** sino a quando termina il figlio desiderato*

Per risolvere questo "problema", ci viene incontro un'altra funzione, la **waitpid**:

```
pid_t waitpid (pid_t pid, int *statLoc, int options);
```

Parametri:

- ***pid*** → *permette di attendere:*
  - *un qualsiasi figlio se **pid = -1** (in questo caso equivale a fare una **wait**)*
  - *il figlio con quel PID se **pid > 0***
  - *un qualsiasi figlio il cui group ID è uguale a quello del chiamante se **pid = 0***
  - *il figlio il cui group ID è uguale a abs(pid) se **pid < -1***
- ***statLoc*** → *stesso significato della **wait***
- ***options*** → *permette controlli aggiuntivi*

# PROCESSI ZOMBIE

Un processo terminato per il quale il padre non ha ancora eseguito una **wait** di dice **zombie**. L'entry viene rimossa solo dopo che il padre ha eseguito una **wait**. Un eccessivo numero di processi zombie, appesantisce e rallenta l'OS.

# PROCESSI ORFANI

Se il padre termina prima di eseguire la wait, il processo figlio diventa **orfano**. I processi orfani vengono ereditati dal processo **init** (*quello con PID=1*) oppure da un processo **init custom utente**.

# TEORIA AGGIUNTIVA

## STATO DI UN PROCESSO

Durante la sua esecuzione, un processo cambia di stato:

- ***New*** → *il processo viene creato e sottomesso all'OS*
- ***Running*** → *in esecuzione*
- ***Ready*** → *logicamente pronto ad essere eseguito, in attesa della risorsa processore*
- ***Waiting*** → *in attesa della disponibilità di risorse da parte del sistema oppure di qualche evento*
- ***Terminated*** → *il processo termina e rilascia le risorse utilizzate*

## PROCESS CONTROL BLOCK (PCB)

L'OS tiene traccia di ogni processo associando ad esso un insieme di dati che includono:

- *stato del processo*
- *program counter*
- *registri della CPU*
- *informazioni utili per lo scheduling della CPU (priorità, ecc.)*
- *informazioni utili per la gestione della memoria*
- *informazioni amministrative varie (limiti, tempi di utilizzo, ecc.)*
- *informazioni sullo stato delle operazioni di I/O (lista file aperti, lista dispositivi I/O, ecc.)*

## CONTEXT SWITCHING

Quando la CPU viene assegnata ad un altro processo, il kernel deve:

- *salvare lo stato del processo **running***
- *caricare un nuovo processo ripristinandone lo stato salvato precedentemente*

Il tempo che un sistema usa per il **context switching** dipende da svariati fattori, quali *HW, OS, numero di processi, politica di scheduling, ecc*..

## SCHEDULING DEI PROCESSI

L'obiettivo della multiprogrammazione è quello di massimizzare l'utilizzo sella CPU da parte dei processi. I processi possono essere classificati in:

- *I/O-bound:*
  - *passano più tempo effettuando I/O che calcoli*
  - *richiedono molti servizi corti da parte della CPU*
- *CPU-bound:*
  - *passano più tempo effettuando calcoli che I/O*
  - *richiedono pochi servizi molto lunghi da parte della CPU*

Per massimizzare l'uso della CPU e soddisfare eventuali vincoli sul tempo di risposta, ogni OS dispone di un **scheduler** mediante il quale gestisce i processi.

Esistono diversi tipi di **scheduler**:

- *scheduler a lungo termine (**long-term scheduler**)*
  - *interviene molto poco frequentemente*
  - *rischedula a tempi dell'ordine di secondi/minuti*
- *scheduler a breve termine (**short-term scheduler**)*
  - *interviene molto frequentemente*
  - *rischedula a tempi dell'ordine dei millisecondi*
  - *deve essere molto veloce*

Lo **scheduler** gestisce i **processi in attesa** di un dispositivo mediante **code** (di **processi**).

# CONTROLLO AVANZATO

La syscall **fork** permette la duplicazione di un processo. Esistono 2 principali applicazioni di tale meccanismo:

- *padre e figlio eseguono **sezioni diverse** di codice*
- *padre e figlio eseguono codici **differenti***

## SOSTITUZIONE DI UN PROCESSO

La syscall **exec** *sostituisce* il processo con un nuovo programma. In particolare la **exec**:

- *non crea un nuovo processo*
- *sostituisce l'immagine del processo corrente (il suo codice, i suoi dati, stack e heap) con quelli di un processo nuovo*
- *il PID del processo non cambia*

### SYSCALL EXEC

Esistono 6 tipi della **syscall exec**:

- *execl, execlp, execle*

- *execv, execvp, execve*

Significato lettere:

| Tipo | Azione |
|---|---|
| l (list) | *la funzione riceve una lista di argomenti* |
| v (vector) | *la funzione riceve un vettore di argomenti* |
| p (path) | *la funzione riceve solo il nome del file (non il path) e lo rintraccia tramite la variabile ambiente PATH* |
| e (environment) | *la funzione riceve un vettore di environment che specifica le variabili di ambiente, invece di utilizzare l'environment corrente* |

Valori di ritorno:

- *nessuno → in caso di successo*
- *il valore -1 → in caso di errore*

Parametri:

- *il path del programma da eseguire*
- *la sua lista di argomenti*
- *le eventuali variabili di ambiente*

**SYSCALL EXECV[P]**

Utilizza come parametro un unico puntatore, che a sua volta individua un vettore di puntatori ai parametri. Il vettore va opportunamente inizializzato. Esempio:

```
char *cmd[] = {"ls", "-laR", ".", (char *) 0};
...
execv ("/bin/ls", cmd);
```

**SYSCALL EXEC[LV]E**

Specifica esplicitamente l'environment mediante un puntatore a vettore di puntatori. Esempio:

```
char *env[] = {"USER=unknown", "PATH=/tmp", NULL};
...
execle (path, arg0, ..., argn, NULL, env);
...
execve (path, argv, env);
```

**CONSIDERAZIONI**

Durante la exec:

- *vengono mantenuti tutti i file descriptor esistenti (compresi stdin, stdout, stderr)*
- *questo comportamento è necessario per ereditare eventuali redirezioni impostate nei comandi di shell ogni volta eseguita la **exec***

In molti OS:

- *solo una versione di **exec** (in genere la **execve**) è implementata come syscall*
- *le altre versioni chiamano tale versione*

# SCHELETRO DI UNA SHELL UNIX

Diversi tipi di comando:

- *eseguito in foreground →* ***<comando>***
- *eseguito in background →* ***<comando> &***

# ESECUZIONE DI UN COMANDO

Può essere conveniente eseguire una stringa di comando dall'interno di un programma in esecuzione (ad esempio, può essere utile inserire data e ora nel nome o nel contenuto di un file). Per risolvere questo "problema" esiste la funzione **system**.

## SYSCALL SYSTEM

La syscall **system**:

1. *passa il comando* ***string*** *all'ambiente host affinché lo esegua*
2. *il controllo viene restituito al processo chiamante una volta terminata l'esecuzione del comando*

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

Parametri:

- *il comando da eseguire*

Valore di ritorno:

- -1 se fallisce la fork o la waitpid usata per realizzarla
- 127, se fallisce la exec usata per realizzarla
- il valore di terminazione della shell che esegue il comando

# SEGNALI

Un segnale è:

- *un* ***interrupt*** *software*
- *una* ***notifica****, inviata dal kernel o da un processo ad un altro processo, per notificargli che si è verificato un determinato evento*

I segnali:

- *permettono di gestire eventi asincroni (es. errori vari, violazioni di accesso in memoria, ecc.)*
- *possono venire utilizzati per la comunicazione tra processi*

Tra i segnali più comuni troviamo:

- ***SIGCHLD*** *→ terminazione di un figlio (viene inviato al padre)*
- ***SIGINT*** *→ equivale alla sequenza Ctrl-C generata da tastiera*
- ***SIGTSTP*** *→ viene inviato in caso di accesso in memoria non valido*
- ***SIGALRM*** *→ viene inviato dopo la chiamata alla syscall sleep(t) (dopo t secondi)*

## GESTIONE DEI SEGNALI

La **gestione di un segnale** implica 3 fasi:

1. *generazione del segnale → quando il kernel o il processo sorgente effettua l'evento necessario*
2. *consegna del segnale → invio del segnale al processo destinatario*

3. *gestione del segnale → per gestire un segnale, un processo deve dire al kernel cosa intende fare nel caso in cui riceva tale segnale*

Per gestire un segnale si possono utilizzare le seguenti syscall:

- ***signal***
- ***kill***
- ***raise***
- ***pause***
- ***alarm***

## SYSCALL SIGNAL

Consente di istanziare un gestore di segnali.

```
void (*signal (int sig, void (*func)(int)))(int);
```

Parametri:

- ***sig*** *→ il segnale da intercettare*
- ***func*** *→ specifica l'indirizzo della funzione da invocare quando si presenta il segnale*

Valore di ritorno:

- *in caso di successo il puntatore al signal handler che gestiva il segnale in precedenza*
- ***SIG_ERR*** *in caso di errore*

La syscall **signal** permette di impostare 3 comportamenti diversi alla ricezione del segnale:

- *lasciare che si verifichi, per ciascun possibile segnale, il comportamento di default (**SIG_DFL**)*
- *ignorare esplicitamente il segnale (**SIG_IGN**)*
- *catturare (catch) il segnale e gestirlo mediante una funzione utente*

## SYSCALL KILL

Invia il segnale **sig** a un processo o ad un gruppo di processi (**per mezzo del parametro pid**).

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

Parametri:

| **Se pid è** | **Si invia il segnale sig...** |
|---|---|
| > 0 | *al processo di **PID** uguale a **pid*** |
| == 0 | *a tutti i processi con **GID** uguale al suo (a cui lo può inviare)* |
| < 0 | *a tutti i processi con **GID** uguale al valore assoluto di **pid** (a cui lo può inviare)* |
| == -1 | *a tutti i processi del sistema (a cui lo può inviare)* |

Valore di ritorno:

- ***il valore 0*** *→ in caso di successo*
- ***il valore -1*** *→ in caso di errore*

## SYSCALL RAISE

La syscall **raise** permette ad un processo di inviare un segnale a se stesso.

```
int raise (int sig);
```

### SYSCALL PAUSE

Sospende il processo chiamante fino all'arrivo di un segnale.

```
int pause (void);
```

### SYSCALL ALARM

Attiva un timer (count-down).

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

Parametri:

- ***seconds*** → *specifica il valore del conteggio (numero di secondi)*

Valore di ritorno:

- *il numero di secondi rimasti prima dell'invio del segnale se ci sono state chiamate precedenti*
- *il valore 0 in caso non ci siano state chiamate precedenti*

Al termine del count-down viene generato il segnale **SIGALRM**.

Se la syscall viene eseguita **prima** che una precedente chiamata abbia originato il segnale, il count-down ricomincia da un nuovo valore (in particolare, se **seconds** è uguale a 0, si disattiva il precedente allarme).

## MEMORIA LIMITATA

La memoria dei segnali "pending" è limitata:

- *si ha al massimo un segnale "pending" (inviato ma non consegnato) per ciascun tipo di segnale*

## FUNZIONI RIENTRANTI

Il comportamento di un segnale prevede:

1. *l'interruzione del flusso di istruzioni corrente*
2. *l'esecuzione del signal handler*
3. *il ritorno al flusso standard alla terminazione del signal handler*

Il **kernel** sa da dove riprendere il flusso di istruzioni precedente, ma il **signal handler** no.

Le funzioni rientranti sono definite dalla "Single UNIX Specification". Si tratta di funzioni che possono essere interrotte senza problemi. Esempi di funzioni rientranti sono: ***read, write, sleep, wait, ecc.*** Esempi di funzioni non rientranti sono: ***printf, scanf, ecc.***

## RACE CONDITIONS

Con questo termine ci si riferisce al comportamento di più processi che lavorano su dati comuni, che dipende dall'ordine di esecuzione.

In particolare, l'**utilizzo di segnali** tra processi può generare **race conditions** che potrebbero interferire con il corretto funzionamento del programma.

# COMANDI DI SHELL PER PROCESSI

Esistono 2 comandi principale per visualizzare lo stato dei processi:

- ***ps*** *(process status of active process) → elenca i processi attivi e i relativi dettagli*

| Compatto | Esteso | Effetto |
|---|---|---|
| -a | | *elenca i processi di tutti gli utenti del sistema* |
| -u | | *visualizza info più dettagliate (resident size, virtual size, ecc.)* |
| -u user | | *visualizza i processi dell'utente <user>* |
| -x | | *aggiunge all'elenco i processi che non hanno un terminale di controllo* |
| -o, -A | | *elenca tutti i processi "running" nel sistema* |
| -f | | *visualizza le info in formato esteso* |
| r (non -r) | | *visualizza sono i processi "running"* |

- ***top*** *→ visualizza informazioni sui processi in esecuzione, aggiornate in run-time*

# COMUNICAZIONE TRA PROCESSI

I processi concorrenti possono essere:

- *indipendenti → quando non viene influenzato e non può influenzare altri processi*
- *cooperanti → quando avviene scambio/condivisione di dati*

## MODELLI DI COMUNICAZIONE

La condivisione di informazioni si denota spesso con il termine **IPC** (*InterProcess Communication*).

I modelli di comunicazione principali sono basati su:

- *memoria condivisa*
- *scambio di messaggi*

Nel modello basato su **memoria condivisa**:

- Condivisione di una zona di memoria
- R/W dei dati in tale area

Questa tecnica è adatta a condividere **quantità elevate** di dati.

Nel modello basato su **scambio di messaggi**:

- occorre instaurare un canale di comunicazione
- richiede l'utilizzo di syscalls

Questa tecnica è adatta allo scambio di dati in **quantità ridotta**.

### CANALI DI COMUNICAZIONE

Sono usualmente caratterizzati da:

- ***denominazione (naming)*** *→ la comunicazione può avvenire specifando esplicitamente il destinatario (diretta) o mediante* ***mailbox*** *(indiretta)*
- ***sincronizzazione*** *→ invio/ricezione in maniera sincrona/asincrona*

- ***capacità*** → *la coda utilizzata per la comunicazione può avere una certa lunghezza (capacità)*

## PIPE

Sono la più vecchia forma di comunicazione in OS UNIX. Forniscono un canale di comunicazione:

- diretto
- asincrono
- a capacità limitata

Le **pipe** risiedono in memoria e sono più efficienti dell'utilizzo di file.

Una **pipe** consiste in un **flusso dati** tra due processi. Essa viene gestita in maniera simile ad un file ed è rappresentata mediante due descrittori (interi), uno per ciasciun estremo. In pratica, un processo **P1** scrive ad una estremità e l'altro processo **P2** legge dall'altra.

La **pipe** è **half-duplex**, cioè permette ai dati di fluire in entrambe le direzioni, ma non contemporaneamente.

## SYSCALL PIPE

La **syscall pipe** crea una **pipe**. Essa ritorna due descrittori di file nel vettore ***fileDescr***:

- *fileDescr[0] → aperto per la lettura della pipe*
- *fileDescr[1] → aperto per la scrittura della pipe*

In pratica l'output effettuato su ***fileDescr[1]*** corrisponde all'input ricevuto su ***fileDescr[0]***.

```
int pipe (int fileDescr[2]);
```

Parametri:

- *il vettore di descrittori di dimensione 2*

Valore di ritorno:

- ***il valore 0*** → *se l'operazione ha avuto successo*
- ***il valore -1*** → *in caso si sia verificato un errore*

Per effettuare una comunicazione tra due processi (padre e figlio):

1. *il processo padre **crea** una pipe*
2. *effettua una **fork***
3. *il processo figlio **eredita** i descrittori dei file*
4. *uno dei due processi (es. padre) scrive nella pipe mentre l'altro (es. figlio) legge dalla pipe*

#### I/O SU PIPE

Le **pipe** si manipolano in maniera identica ai file UNIX. Si utilizzano le primitive **read** e **write**. In particolare:

- *la syscall **read**:*
  - *si blocca se la pipe è vuota*
  - *ritorna solo i caratteri disponibili, se la pipe contiene meno caratteri di quanto richiesto*
  - *ritorna 0 se la pipe è stata chiusa all'altra estremità*
- *la syscall **write**:*
  - *si blocca se la pipe è piena*
  - *ritorna **SIGPIPE** se l'altra estremità è stata chiusa*

# THREADS

Un **thread** è una sezione di un processo che viene schedulata ed eseguita indipendentemente dal processo che l'ha generata. Un **thread** può condividere il proprio spazio di indirizzamento con altri thread.

In particolare:

- *i dati sono condivisi con gli altri thread dello stesso processo*
- *un thread contiene dati privati in quanto indipendente (es. program counter, registri HW, stack, local variables, ecc.)*

Vantaggi derivanti dall'utilizzo di thread:

- *tempi di risposta ridotti → creare un thread è 10-100 volte più veloce che generare un processo*
- *risorse condivise → i thread condividono i dati in maniera automatica*
- *costi minori per la gestione delle risorse → una sezione di codice/dati può servire più clienti*
- *maggiore scalabilità → nei sistemi multi-core, i thread permettono più semplicemente paradigmi di programmazione concorrente basati su **separazione di compiti**, **suddivisione dei dati**, ecc.*

Svantaggi derivanti dall'utilizzo di thread:

- *non esiste protezione tra thread → tutti sono eseguiti nello stesso spazio degli indirizzi e la protezione da parte dell'OS è pressocché impossibile. In particolare, se i thread non sono sincronizzati, l'accesso a dati condivisi potrebbe non essere **thread safe***
- *tra thread non esiste relazione gerarchica padre-figlio → tutti i thread sono uguali*

# MODELLI DI PROGRAMMAZIONE MULTI-THREAD

Esistono 3 modelli di programmazione multi-thread:

- *kernel-level thread*
- *user-level thread*
- *soluzione mista o ibrida*

## KERNEL-LEVEL THREAD

I thread sono gestiti dal kernel. L'OS è a conoscenza dell'esistenza dei thread e fornisce un supporto adeguato per la loro manipolazione (tutte le operazioni sono effettuate tramite syscalls). L'OS mantiene per ciascun thread, informazioni simili a quelle che mantiene per i processi (es. tabella dei thread, Thread Control Block (TCB) per ogni thread attivo, ecc.)

Vantaggi:

- *l'OS è a conoscenza dei thread → può decidere quale processo schedulare, come suddividere il tempo tra i vari processi, ecc.*
- *efficace nelle applicazioni che si bloccano spesso → se un thread si blocca è sempre possibile eseguirne un altro nello stesso processo*
- *permette un effettivo parallelismo → in un sistema multiprocessore si possono eseguire thread multipli*

Svantaggi:

- *a causa del passaggio al modo kernel, la gestione è relativamente lenta e inefficiente*
- *limitazione nel numero massimo di thread*
- *l'OS deve mantenere informazioni costose (es. TCB, ecc.)*

## USER-LEVEL THREAD

Il pacchetto dei thread è inserito completamente nello spazio dell'utente. Il kernel non è quindi a conoscenza dei thread e gestisce solo i processi. I thread vengono gestiti tramite una libreria, che permetterà quindi creazione, sincronizzazione, ecc. Ogni processo ha bisogno di una tabella personale dei thread in esecuzione, si avranno perciò TCB di dimensioni ridotte e una visibilità locale delle informazioni all'interno del processo.

Vantaggi:

- *si possono implementare in tutti i kernel, anche in sistemi che nativamente non supportano i thread*
- *non richiede modifiche all'OS*
- *gestione efficiente → molto più veloci dei thread kernel*
- *al programmatore è consentito generare tutti i thread desiderati*

Svantaggi:

- *l'OS non sa che esistono i thread*
- *possono essere fatte scelte inopportune oppure poco efficienti*
- *occorre comunicare informazioni tra il kernel e il manager dell'utente a run-time*
- *lo scheduler deve mappare i thread utente sull'unico thread kernel (se il thread kernel si blocca, si bloccano tutti i thread utente)*

## IMPLEMENTAZIONE IBRIDA

Questo tipo di implementazione tenta di combinare i vantaggi di entrambi gli approcci.

In particolare:

- *l'utente decide quanti thread utente eseguire e su quanti thread kernel mapparli*
- *il kernel è a conoscenza solo dei thread kernel e gestisce solo tali thread*
- *ogni thread kernel può essere utilizzato a turno da diversi thread utente*

# COESISTENZA DI PROCESSI E THREAD

## UTILIZZO DELLA SYSCALL FORK

In un processo multi-thread, una **fork** duplica solo il thread che richiama la **fork**. I dati privati posseduti dai thread non duplicati dalla **fork**, possono non essere gestibili dall'unico thread duplicato.

## UTILIZZO DELLA SYSCALL EXEC

Una **exec** effettuata da un thread sostituisce con il nuovo programma l'intero processo (non solo il thread che effettua la **exec**).

# LA LIBRERIA PTHREAD

In generale, un libreria di thread, fornisce l'interfaccia per effettuare la gestione dei thread da parte del programmatore.

La gestione può essere effettuata:

- *a livello utente → mediante funzioni*
- *a livello kernel → mediante syscalls*

**POSIX thread** o **Pthread** è una libreria per la gestione di thread. Grazie a questa libreria, il thread è una **funzione** che viene eseguita in maniera indipendente dal resto del programma.

**Pthread** permette di:

- *creare e manipolare thread*
- *sincronizzare thread*
- *proteggere le risorse comuni ai thread*
- *schedulare thread*
- *distruggere thread*

## SYSCALL PTHREAD_EQUAL

```
int pthread_equal (pthread_t tid1, pthread_t tid2);
```

Confronta due identificati di thread.

Parametri:

- *due identificatori di thread*

Valore di ritorno:

- ***diverso da 0*** → *thread uguali*
- ***uguale a 0*** → *thread diversi*

## SYSCALL PTHREAD_SELF

```
pthread_t pthread_self (void);
```

Ritorna l'identificatore di thread del thread chiamante. Viene principalmente utilizzata da un thread per auto-identificarsi e accedere quindi correttamente ai propri dati personali.

## SYSCALL PTHREAD_CREATE

```
int pthread_create (pthread_t *tid, const pthread_attr_t *attr,
             void *(*startRoutine)(void *), void *arg);
```

Crea un nuovo thread.

Parametri:

- *identificatore del thread generato (**tid**)*
- *attributi del thread (**attr**)*
- *routine C eseguita dal thread (**startRoutine**)*
- *argomento passato alla routine di inizio (**arg**)*

Valore di ritorno:

- ***il valore 0*** → *in caso di successo*
- ***un codice di errore*** → *in caso di fallimento*

## SYSCALL PTHREAD_EXIT

```
void pthread_exit (void *valuePtr);
```

Un singolo thread può terminare:

- *effettuando un **return** dalla sua funzione di inizio*

- *eseguendo una **pthread_exit***
- *ricevendo una **pthread_cancel** da un altro thread*

Parametri:

- *il valore valuePtr è mantenuto dall'OS sino a quando un thread fa una **pthread_join***

## SYSCALL PTHREAD_JOIN

Alla sua creazione un thread può essere dichiarato:

- ***joinable** → un altro thread può effettuare una **wait** su di lui e può quindi acquisire il suo stato di uscita*
- ***detached** → non si può attendere esplicitamente la sua terminazione*

```
int pthread_join (pthread_t tid, void **valuePtr);
```

Viene usata da un thread per attendere (wait) un altro specifico thread.

Parametri:

- *identificare del thread atteso (**tid**)*
- *il puntatore (senza tipo) **valuePtr** riferirà il valore ritornato dal thread **tid***

Valore di ritorno:

- ***il valore 0** → in caso di successo*
- ***un codice di errore** → in caso di fallimento*

## SYSCALL PTHREAD_CANCEL

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

Termina il thread indicato. Il thread che fa la richiesta non attende la terminazione del thread (effettua la richiesta e continua).

Parametri:

- *identificatore del thread da terminare (**tid**)*

Valore di ritorno:

- ***il valore 0** → in caso di successo*
- ***un codice di errore** → in caso di fallimento*

## SYSCALL PTHREAD_DETACH

```
int pthread_detach (pthread_t tid);
```

Dichiara il thread **tid** come detached, cioè non sarà più possibile effettuare un join con tale thread.

Parametri:

- identificatore del thread (**tid**)

Valore di ritorno:

- ***il valore 0** → in caso di successo*
- ***un codice di errore** → in caso di fallimento*

# TERMINOLOGIA

## CONCORRENZA

- *atto di eseguire più calcoli nello stesso intervallo di tempo*
- *task multipli (due o più) vengono eseguiti negli stessi intervalli di tempo, senza uno specifico ordine particolare*
- *l'esecuzione dei task sembra contemporanea, ma non lo è*
- *l'effetto è dovuto al time-slicing, ovvero allo scheduler (context switching) che dedica l'unità di calcolo ai vari task per unità di tempo infinitesimali*

## PARALLELISMO

- *atto di eseguire più calcoli simultaneamente*
- *task multipli (o parti diverse dello stesso task) vengono eseguiti(/e) contemporaneamente in sistemi multiprocessori o multi-core*
- *viene permesso da strutture hw apposite (multi-CPU o multi-core)*
- *si ottiene trasformando un flusso di esecuzione sequenziale in uno parallelo*

Lo scopo del parallelismo è aumentare l'efficienza. La valutazione dell'efficienza si effettua mediante parametri quali **tempo** e **memoria**. Esistono diverse metriche per il calcolo dei tempi di esecuzione:

- ***user time*** *→ tempo totale che la CPU dedica ad eseguire un determinato task a livello utente (non tiene conto del tempo impiegato per operazioni di I/O, per esecuzione di routine a livello kernel, ecc.)*
- ***CPU time*** *→ tempo totale dedicato dalla CPU all'esecuzione di un task (tiene conto anche di tempi per I/O, gestione del task da parte del kernel, ecc.)*
- ***wall clock time*** *→ anche detto* ***elapsed time (tempo trascorso)****, è il tempo effettivo richiesto per terminare un determinato compito (tempoFine - tempoInizio)*

# SHELL

Si tratta dell'interfaccia utente dell'OS. In UNIX la shell non è parte del kernel, è un **normale processo utente**.

Ogni shell permette:

- *gestione di comandi in linea → comprende automaticamente quando il costrutto termina e lo esegue immediatamente*
- *scrittura di programmi (script) → memorizzazione dei comandi desiderati in un file e successiva esecuzione degli stessi semplicemente richiamando il file*

La scrittura di uno script evita di:

- *digitare ripetutamente complesse sequenze di comandi*
- *automatizzare operazioni tediose, ripetitive e propense a banali errori*

Una **shell** può essere attivata:

- *automaticamente al login*
- *in modo annidato dentro un'altra shell*

Una **shell** termina digitando:

- *il comando* ***exit***

- *il carattere di **EOF** (tipicamente CTRL-D)*

Alcuni caratteri assumnono un significato particolare all'interno delle shell.

# PARENTESI (), [], {}

Servono per racchiudere variabili, operazioni aritmetiche, ecc..

```
> nome=Gian

> echo $nomemarco      <- variabile inesistente
  (stampa vuota)

> echo {$nome}marco
  {Gian}marco

> echo ${nome}marco
  Gianmarco
```

# QUOTING

Per **quoting** si intende l'utilizzo di:

- *apici* → ' '
- *virgolette* → " "
- *backslash* → \

```
> myVar="Val ore"

> echo $myVar
  Val ore

> echo 'v = $myVar'
  v = $myVar

> echo "v = $myVar"
  v = Val ore

> echo \$myVar
  $myVar

> echo "virgoletta\""
  virgoletta"
```

Lo **stdout** di un comando può essere catturato mediante:

- la sequenza di caratteri **$(...)**
- gli apici inversi (back-quote) ` `

# HISTORY

Ogni **shell** ricorda l'elenco degli ultimi comandi digitati, salvandoli nel file ***.bash_history***

Comandi utili:

| Comando | Significato |
|---|---|
| history | *mostra l'elenco dei comandi eseguiti precedentemente* |
| !n | *eseguo il comando numero **n** nel buffer* |
| !str | *esegue l'ultimo comando che inizia con **str*** |
| ^str1^str2 | *sostituisce nell'ultimo comando **str1** con **str2** e lo esegue nuovamente* |

## ALIASING

Nelle **shell** è possibile definire nuovi nomi per comandi esistenti, mediante il comando ***alias***:

```
alias nome="stringa"
```

Digitando ***alias*** avremo una lista degli alias attivi nella shell utilizzata.

Digitando ***unalias nome*** elimineremo l'alias **nome** dalla shell.

# SCRIPT DI SHELL

Il **linguaggi di shell** sono **interpretati**, cioè non esiste una fase di compilazione esplicita.

Gli script:

- *sono normalmente memorizzati in file di estensione **.sh** (**.bash**)*
- *possono essere eseguiti mediante due tecniche:*
  - *esecuzione diretta*
  - *esecuzione indiretta*

## ESECUZIONE DIRETTA

```
./scriptname args
```

Lo script viene eseguito da riga di comando come un normale file eseguibile.

E' importante notare che:

- *il file abbia il permesso di esecuzione (**chmod +x ./scriptname**)*
- *la prima riga dello script deve specificare il nome dell'interprete dello script (**#!/bin/bash** oppure **#!/bin/sh**, ecc.)*
- *lo script viene eseguito da una sotto-shell*

## ESECUZIONE INDIRETTA

```
source ./scriptname args
```

Si invoca la shell con il nome del programma come parametro.

In questo caso:

- *è la shell corrente ad eseguire lo script*

- *non è necessario che lo script sia eseguibile*
- *le modifiche effettuate dallo script alle variabili di ambiente rimangono valide nella shell corrente*

# DEBUG DI UNO SCRIPT

Non esistono tool specifici per effettuare il debug di script bash. E' possibile però eseguire uno script in "debug" in maniera:

- *completa → intero script*
- *parziale → solo alcune righe dello script*

Opzioni:

| Compatto | Esteso | Effetto |
|---|---|---|
| -o noexec<br>-n | | *esegue uno controllo sintattico ma non esegue lo script* |
| -o verbose<br>-v | | *visualizza i comandi eseguiti (fa l'**echo**)* |
| -o xtrace<br>-x | | *visualizza la traccia di esecuzione dell'intero script* |
| -o nounset<br>-u | | *riporta un errore per variabile non definita* |

# SINTASSI

Il linguaggio **bash** è relativamente di "alto livello", cioè è in grado di unire:

- *comandi standard di shell (es. ls, wc, find, grep, ecc.)*
- *costrutti "standard" del linguaggio di shell (es. I/O, variabili e parametri, operatori aritmetici/logici, ecc.)*

Istruzione/comandi sulla stessa riga devono essere separati dal carattere " **;** " .

Il carattere " **#** " indica la presenza di un commento sulla riga.

La syscall **exit** permette di terminare uno script restituendo un eventuale codice di errore.

```
IL VALORE VERO NELLE SHELL E' 0 (E NON 1 COME IN C)!
```

## PARAMETRI

I parametri dello script possono essere individuati mediante " **$** ":

- *parametri posizionali*
  - ***$0** → nome dello script*
  - ***$1, $2, $3, ecc.** → parametri passati allo script sulla riga di comando*
- *parametri speciali*
  - ***$*** → intera lista (stringa) dei parametri (non include il nome dello script)*
  - ***$#** → contiene il numero di parametri (nome dello script escluso)*
  - ***$$** → contiene il PID del processo*

## VARIABILI

Si suddividono in:

- ***locali (o di shell)*** → *disponibili solo nella shell corrente*
- ***globali (o d'ambiente)*** → *disponibili in tutte le sotto-shell*

Caratteristiche principali:

- *non vanno dichiarate*
- *sono case sensitive*

L'elenco di tutte le variabili definite e il relativo valore viene visualizzato con il comando **set**. Per cancellare il valore di una variabile si utilizza il comando **unset** (*unset nome*).

### VARIABILI LOCALI (O DI SHELL)

Il contenuto ne specifica il tipo (es. stringa, intero , vettoriale o matriciale, ecc.).

N.B. : di default viene memorizzata una stringa anche quando il valore è numerico.

*Assegnazione* → ***nome="valore"*** *(nessuno spazio prima e dopo ' = ') Utilizzo* → ***$nome***

### VARIABILI GLOBALI (O D'AMBIENTE)

Per rendere una variabile **globale**, si utilizza il comando ***export*** (*export nome*).

## LETTURA (READ)

La funzione **read** permette di leggere una riga da stdin.

```
read [ opzioni ] var1 var2 ... varn
```

Opzioni:

| Compatto | Esteso | Effetto |
|---|---|---|
| -n nchars | | *legge n caratteri* |
| -t timeout | | *restituisce 1 se non si introducono i dati entro timeout secondi* |

ecc...

## SCRITTURA (OUTPUT)

Le operazioni di visualizzazoione possono essere effettuate con le funzioni:

- ***echo*** → *visualizza i propri argomenti separati da un carattere di " **a capo** "*
- ***printf*** → *sintassi simile al C, non è necessario separare i campi con la " **,** "*

## ESPRESSIONI ARITMETICHE

Per esprimere espressioni aritmetiche è possibile utilizzare diverse notazioni:

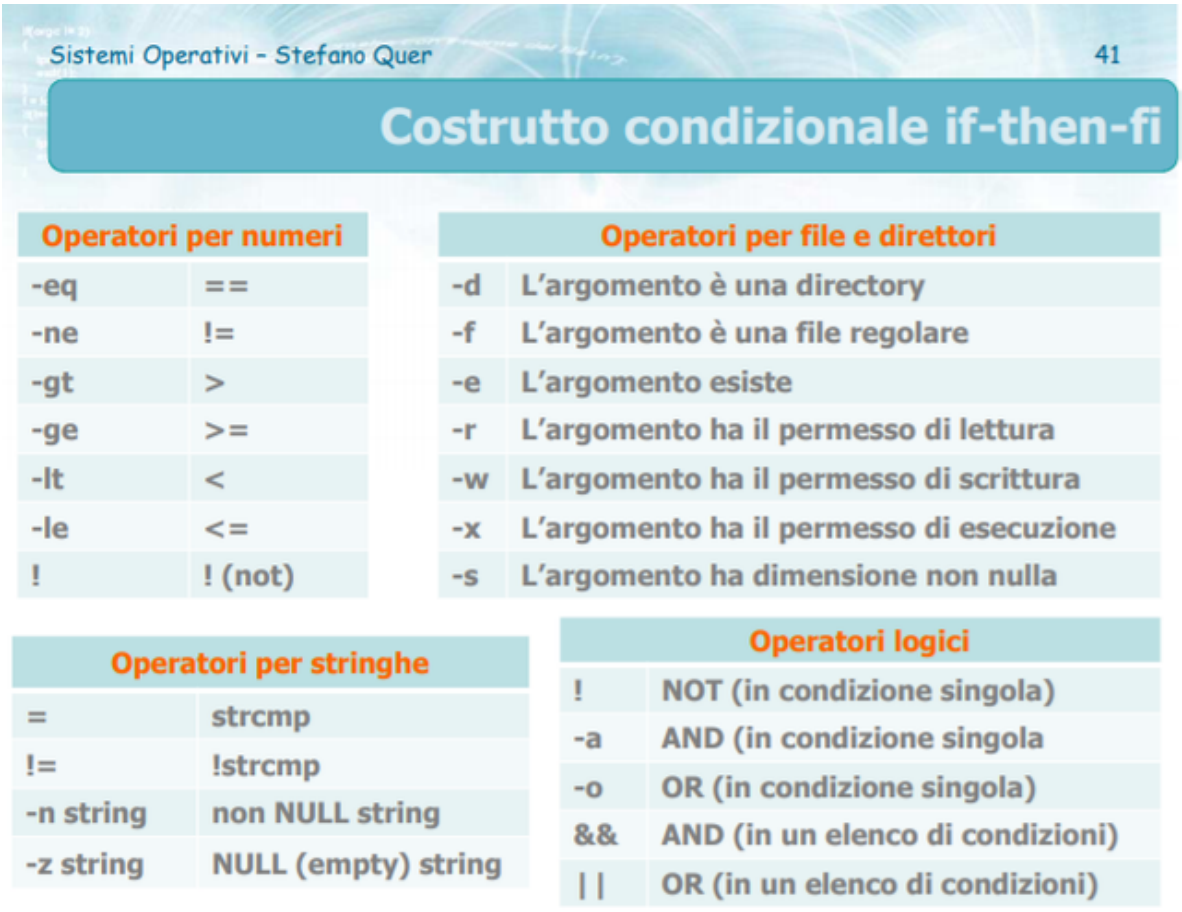
- *il comando **let***
- *le doppie parentesi tonde **(( ... ))***
- *le parentesi quadre **[ ... ]***
- *il costrutto **expr***

## COSTRUTTO CONDIZIONALE IF-THEN-FI

Verifica se lo stato di uscita (exit status) di una sequenza di comandi è uguale a 0; in caso affermativo esegue uno o più comandi.

Il costrutto può essere esteso:

- *per includere la condizione else →* ***if-then-else-fi***
- *per effettuare controlli annidati →* ***if-then-elif-then-else-fi*** *ecc.*




## COSTRUTTO ITERATIVO FOR-IN

Esegue i comandi specificati, una volta per ogni valore assunto dalla variabile **var**. L'elenco dei valori **[list]** può essere indicato:

- *in maniera esplicita (elenco)*
- *in maniera implicita (comandi di shell, wild-cards, ecc.)*

```
# Syntax 1

for var in [list]
do
   statements
done
```

```
# Syntax 2

for var in [list]; do
   statements
done
```

## COSTRUTTO ITERATIVO WHILE-DO-DONE

Si itera fino a quando la condizione è vera.

```
# Syntax 1

while [ cond ]
do
    statements
done
```

```
# Syntax 2

while [ cond ]; do
    statements
done
```

## BREAK, CONTINUE E `:`

I costrutti **break** e **continue** hanno comportamenti standard con i cicli for e while:

- uscita non strutturata dal ciclo
- passaggio all'iterazione successiva

Il carattere `:` può essere utilizzato per creare "istruzioni nulle". Esempio:

```
1 if [ -d "$file" ]; then
2     :           # empty instruction
3 fi
```

## VETTORI

In bash è possibile utilizzare variabili vettoriali mono-dimensionali. Ogni variabile può essere definita come vettoriale (non necessaria ma comunque possibile la dichiarazione esplicita con **declare**).

Non esiste limite alla dimensione di un vettore.

Definizione:

- *elemento per elemento:* ***name[index]="value"***
- *tramite elenco di valori:* ***name=(lista valori separati da spazio)***

Riferimento:

- *al singolo elemento:* ***${name[index]}***
- *a tutti gli elementi:* ***${name[*]}***

Numero di elementi:

- ***${#name[*]}***

Lunghezza dell'elemento index (numero caratteri):

- ***${#name[index]}***

Eliminazione:

- *di un singolo elemento:* ***unset name[index]***
- *di un intero vettore:* ***unset name***

# SINCRONIZZAZIONE

## LIFO - STACK

```
void push (int val) {
  if(top>=SIZE)
    return;
  stack[top] = val;
  top++;
  return;
}
```

```
void pop (int *val) {
  if(top<=0)
    return;
  top--;
  *val=stack[top];
  return;
}
```

- *le funzioni* ***push*** *e* ***pop*** *agiscono sulla stessa estremità dello stack*
- *la variabile* ***top*** *è condivisa*

## FIFO - QUEUE - BUFFER CIRCOLARE

```
void enqueue (int val) {
  if (n>SIZE) return;
  queue[tail] = val;
  tail=(tail+1)%SIZE;
  n++;
  return;
}
```

```
int dequeue (int *val) {
  if (n<=0) return;
  *val=queue[head];
  head=(head+1)%SIZE;
  n--;
  return;
}
```

- *le funzioni* ***enqueue*** *e* ***dequeue*** *agiscono su estremità "diverse" della cosa usando variabili diverse (****head*** *e* ***tail****)*
- *la variabile* ***n*** *è comunque condivisa*

## SEZIONI CRITICHE

Una **sezione critica** (**SC**) o **regione critica** (**RC**) è una sezione di codice, comune a più processi o threads competono per l'uso (in lettura e scrittura) di risorse comuni (es. dati condivisi).

Le corse critiche potrebbero essere evitate se:

- *non si avessero mai più P (o T) nella stessa SC contemporaneamente*
- *quando un P (o T) è in esecuzione nella sua SC nessun altro P (o T) potesse fare altrettanto*
- *il codice nella SC fosse eseguito da un singolo P (o T) alla volta*
- *l'esecuzione del codice nella SC fosse effettuato in mutua esclusione*

Per ovviare a questo problema, occorre stabilire un **protocollo di accesso** per forzare la **mutua esclusione**, cioè:

- *per entrare in una SC, un processo esegue* ***codice di prenotazione***
- *per uscire da una SC, un processo esegue* ***codice di rilascio*** *della regione occupata*

Ogni **SC** è protetta da:

- *una sezione di ingresso (di prenotazione o prologo)*
- *una sezione di uscita (di rilascio)*

Ogni soluzione al problema della **SC**, DEVE soddisfare i seguenti requisiti:

- *mutua esclusione (ME) → un solo P (o T) alla volta deve ottenere l'accesso alla **SC***
- *progresso → se nessun P (o T) si trova nella **SC** e un P (o T) desidera entrarci, deve poterlo fare in un tempo definito (bisogna evitare **deadlock** tra P (o T))*
- *attesa definita → deve esistere un numero definito di volte per cui altri P (o T) riescano ad accedere alla **SC** prima che un P (o T) specifico e che ha fatto una richiesta di accesso possa farlo (bisogna evitare **starvation** di P (o T))*
- *simmetria → la selezione di chi deve accedere alla **SC** non dovrebbe dipendere dalla priorità relativa tra P (o T) o dalla velocità relativa tra P (o T)*

Possiamo trovare diverse soluzioni a questo problema, di diversi tipi:

- *software → logica dell'algoritmo*
- *hardware → soluzioni architetturali*
- *ad-hoc (semafori) → l'OS fornisce strutture e funzioni per ovviare al problema*

# SOLUZIONI SOFTWARE

Si basano sull'utilizzo di **variabili globali**.

## SOLUZIONE 1

*Variabili globali:*

- ***int flag[2] = {FALSE, FALSE};***

$P_i/T_i$

```
while (TRUE) {
  while (flag[j]);
  flag[i] = TRUE;
  SC
  flag[i] = FALSE;
  sezione non critica
}
```

$P_j/T_j$

```
while (TRUE) {
  while (flag[i]);
  flag[j] = TRUE;
  SC
  flag[j] = FALSE;
  sezione non critica
}
```

Con questa soluzione, la mutua esclusione **NON** è assicurata, in quanto $P_i$ e $P_j$ possono accedere alla SC contemporaneamente. La tecnica fallisce in quanto la variabile **lock** (il vettore globale di flag "**SC busy**") viene controllata e modificata mediante 2 istruzioni a sé stanti

## SOLUZIONE 2

*Variabili globali:*

- ***int flag[2] = {FALSE, FALSE};***

```
Pi/Ti
------------------------
while (TRUE) {
  flag[i] = TRUE;
  while (flag[j]);
  SC
  flag[i] = FALSE;
  sezione non critica
}
```

```
Pj/Tj
------------------------
while (TRUE) {
  flag[j] = TRUE;
  while (flag[i]);
  SC
  flag[j] = FALSE;
  sezione non critica
}
```

Con questa soluzione, il **progresso** non è garantito, in quanto i due processi potrebbero rimanere bloccati per sempre (**deadlock** o **livelock**).

## SOLUZIONE 3

*Variabili globali:*

- ***int turn = i;***

```
Pi/Ti
------------------------
while (TRUE) {
  while (turn!=i);
  SC
  turn = j;
  sezione non critica
}
```

```
Pj/Tj
------------------------
while (TRUE) {
  while (turn!=j);
  SC
  turn = i;
  sezione non critica
}
```

In questo caso, si utilizza una sola variabile che indica "di chi è il turno" per eseguire la **SC**.

Con questa soluzione, **l'attesa non è definita**, in quanto se $P_i$ non è interessato, $P_j$ va in **starvation** (o viceversa).

## SOLUZIONE 4

*Variabili globali:*

- ***int turn = i;***
- ***int flag[2] = {FALSE, FALSE};***

```
Pi/Ti
------------------------
while (TRUE) {
  flag[i] = TRUE;
  turn = j;
  while (flag[j] &&
    turn==j);
  SC
  flag[i] = FALSE;
  sezione non critica
}
```

```
Pi/Ti
------------------------
while (TRUE) {
  flag[j] = TRUE;
  turn = i;
  while (flag[i] &&
    turn==i);
  SC
  flag[j] = FALSE;
  sezione non critica
}
```

Questa soluzione è l'unica corretta, cioè l'unica che garantisce:

- *mutua esclusione*
- *progresso (**no deadlock**)*
- *attesa limitata (**no starvation**)*
- *simmetria*

```
In generale, le soluzioni software al problema delle SC risultano
complesse e inefficienti.
```

# SOLUZIONI HARDWARE

Possono essere divise in:

- *soluzioni che non permettono il **diritto di prelazione***
- *soluzioni che permettono il **diritto di prelazione***

## SISTEMI SENZA DIRITTO DI PRELAZIONE

E' un sistema tale per cui:

- *i P (o T) in esecuzione nella CPU non possono essere interrotti*
- *il controllo verrà rilasciato al kernel solo quando il P (o T) lo lascerà volontariamente*

Nei sistemi **mono-processore** e senza diritto di prelazione non esiste il problema delle **SC**.

Nei sistemi **multi-processore** o **multi-core**, il fatto che il kernel sia senza diritto di prelazione implica tempi di risposta eccessivi e non adatti alla programmazione "real-time".

## SISTEMI CON DIRITTO DI PRELAZIONE

In sistemi di questo tipo:

1. un processo in esecuzione in modalità kernel può essere interrotto
2. l'arrivo di un interrupt sposta il controllo del flusso su un altro processo
3. il processo originario verrà terminato in seguito

Nei sistemi **mono-processore** con diritto di prelazione, è possibile risolvere il problema delle **SC** mediante controllo dell'interrupt *(disable interrupt nella sezione d'ingresso, enable interrupt nella sezione d'uscita)*.

```
while (TRUE) {
  disabilita interrupt
  SC
  abilita l'interrupt
  sezione non critica
}
```

In generale, disabilitare gli interrupt ha però diversi svantaggi:

- la procedura è intrinsecamente insicura (in caso di errori, come si segnalano?)
- nei sistemi multi-processore (o multi-core) occorre disabilitare gli interrupt su tutti i processori

## MECCANISMI DI LOCK - UNLOCK

Vengono utilizzati:

- lucchetti di protezione (**lock**)
- istruzioni indivisibili (**atomiche**) per manipolare tali **lock**

Esistono 2 principali istruzioni atomiche di lock:

- ***Test-And-Set*** → *setta e restituisce una variabile di lock globale, agisce in maniera atomica (in un solo ciclo indivisibile)*
- ***Swap*** → *scambia due variabili, di cui una di lock globale, agisce in maniera atomica (in un solo ciclo indivisibile)*

Entrambe le tecniche precedenti:

- *assicurano la mutua esclusione*
- *assicurano il progesso, evitando* ***deadlock***
- *NON assicurano l'attesa definita di un processo, ovvero non garantiscono la non* ***starvation***
- *sono simmetriche*

```
Vantaggi delle soluzioni hardware:
   - utilizzabili in ambienti multi-processore
   - facilmente estendibili a N processi
   - relativamente più semplici da utilizzare per l'utente
   - simmetriche
```

```
Svantaggi delle soluzioni hardware:
   - non facili da implementare a livello hardware
   - starvation
   - busy waiting su spin-lock (spreco di risorse CPU nell'attesa)
```

# SOLUZIONI AD-HOC (SEMAFORI)

Un **semaforo S** è:

- *una variabile intera condivisa*
- *protetta dall'OS*
- *utilizzabile per inviare/ricevere segnali*

Le operazioni su **S** sono sempre eseguite in maniera **atomica** (garantita dall'OS).

## OPERAZIONI STANDARD SU SEMAFORI

- ***init (S, k)*** → *definisce e inizializza il semaforo S al valore k*
- ***wait (S)*** → *permette (nella sezione di ingresso) di ottenere l'accesso alla SC protetta dal semaforo S*
- ***signal (S)*** → *permette (nella sezione d'uscita) di uscire dalla SC protetta dal semaforo S*
- ***destroy (S)*** → *cancella (libera/free) il semaforo S*

### FUNZIONE INIT

Esistono 2 tipi di semafori:

- *binari → il valore di k è 0 oppure 1*
- *con conteggio → il valore di k è un intero nell'intervallo [0,k] con k>1*

```
init (S, k) {
  alloc S (global var);
  S=k;
}
```

**FUNZIONE WAIT**

Se il valore di **S** è negativo o nulla **blocca** il processo chiamante (la risorsa non è disponibile). In ogni caso, decrementa il valore di **S**.

```
wait (S) {
  while (S<=0);
  S--;
}
```

**FUNZIONE SIGNAL**

Incrementa la variabile semaforica.

```
signal (S) {
  S++;
}
```

**FUNZIONE DESTROY**

Rilascia la memoria occupata dal semaforo **S**.

```
destroy (S) {
  free (S);
}
```

## IMPLEMENTAZIONE DI UN SEMAFORO

Definiamo un semaforo come una struttura C munita di:

- *un contatore*
- *una lista (coda) di processi*

```
typedef struct semaphore_s {
  int cnt;               // Numero processi
  process_t *head;       // Lista processi
} semaphore_t;
```

L'implementazione reale permette a un semaforo di avere valori negativi. La coda:

- *può essere implementata con un puntatore nel Process Control Block (PCB) dei processi*
- *può soddisfare le politiche che lo scheduler desidera (es. FIFO, ecc.)*
- *ha un comportamento indipendente dai processi in attesa*

Esistono diverse implementazioni:

- *semafori tramite pipe*
- *semafori POSIX*

- *semafori linux*
- *pthread*

## SEMAFORI TRAMITE PIPE

Data una pipe:

- *il **contatore** di un semaforo è realizzato tramite il concetto di **token***
- *la **signal** è effettuata tramite una **write** di un **token** sulla pipe (non bloccante)*
- *la **wait** è effettuata tramite una **read** di un token dalla pipe (bloccante)*

**SEMAPHOREINIT (S)**

```
#include < unistd.h >

void semaphoreInit (int *S) {
  if (pipe (S) == -1) {
    printf (“Error”);
    exit (-1);
  }
  return;
}
```

Inizializza il semaforo S (la variabile S va definita come variabile globale).

**SEMAPHORESIGNAL (S)**

```
#include < unistd.h >

void semaphoreSignal (int *S) {
  char ctr = ‘X’;
  if (write(S[1], &ctr, sizeof(char)) != 1) {
    printf (“Error”);
    exit (-1);
  }
  return;
}
```

Scrive un carattere (qualsiasi) sulla pipe.

**SEMAPHOREWAIT (S)**

```
#include < unistd.h >

void semaphoreWait (int *S) {
  char ctr;
  if (read (S[0], &ctr, sizeof(char)) != 1) {
    printf (“Error”);
    exit (-1);
  }
  return;
}
```

Legge un carattere dalla pipe (read bloccante).

## SEMAFORI POSIX

Si tratta di un implementazione indipendente dall'OS. Bisogna includere l'**header file** *"semaphore.h"*. Un semaforo è una variabile di tipo **sem_t**. Tutte le funzioni sono denominate *"sem_*"*, e ritornano -1 in caso di errore.

#### SEM_WAIT

```
int sem_wait (sem_t *sem);
```

*Operazione di wait standard:*

- *se il semaforo è uguale a 0 → blocca il chiamante fino a quando può decrementare il valore del semaforo*

#### SEM_TRYWAIT

```
int sem_trywait (sem_t *sem);
```

*Operazione di wait senza blocco:*

- *se il semaforo ha un valore > 0 → lo decrementa e ritorna 0*
- *se il semaforo è uguale a 0 → ritorna -1 (invece di bloccare come la wait)*

#### SEM_POST

```
int sem_post (sem_t *sem);
```

*Classica operazione signal:*

- *incrementa il valore del semaforo*

#### SEM_GETVALUE

```
int sem_getvalue (sem_t *sem, int *valP);
```

*Permette di esaminare il valore di un semaforo:*

- *il valore del semaforo viene assegnato a ***valP***
- *se ci sono processi in attesa, a ***valP** si assegna 0 o un numero negativo il cui valore assoluto è uguale al numero di processi in attesa*

#### SEM_DESTROY

```
int sem_destroy (sem_t *sem);
```

*Distrugge un semaforo creato precedentemente:*

- *può ritornare -1 se si cerca di distruggere un semaforo utilizzato da un altro processo*

## SEMAFORI MUTEX PTHREAD

Il tipo base di un mutex è ***pthread_mutex_t***.

Funzioni base:

- *pthread_mutex_init*
- *pthread_mutex_lock*
- *pthread_mutex_trylock*

- *pthread_mutex_unlock*
- *pthread_mutex_destroy*

**PTHREAD_MUTEX_INIT**

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

Crea un nuovo **lock mutex** (mutex variable):

- *restituisce **mutex** al chiamante*
- *attr specifica gli attributi di **mutex** (default: NULL)*

Valori di ritorno:

- *0 → in caso di successo*
- *il codice di errore → in caso di fallimento*

**PTHREAD_MUTEX_LOCK**

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
```

Controlla il valore del **lock mutex** e:

- *blocca il chiamante se è già locked*
- *acquisisce il lock se non è locked*

Valori di ritorno:

- *0 → in caso di successo*
- *il codice di errore → in caso di fallimento*

**PTHREAD_MUTEX_TRYLOCK**

```
int pthread_mutex_trylock (pthread_mutex_t *mutex);
```

Simile a *phtread_mutex_lock* ma nel caso il lock sia già stato acquisito ritorna senza bloccare il chiamante.

Valori di ritorno:

- *0 → se il lock è stato già acquisito*
- *codice di errore EBUSY → se il mutex era posseduto da un altro thread*

**PTHREAD_MUTEX_UNLOCK**

```
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

Rilascia il **lock mutex** al termine della **SC**.

Valori di ritorno:

- *0 → in caso di successo*
- *il codice di errore → in caso di fallimento*

**PTHREAD_MUTEX_DESTROY**

```
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

Rilascia la memoria occupata dal **lock mutex**. Tale lock non sarà più utilizzabile.

Valori di ritorno:

- *0 → in caso di successo*
- *il codice di errore → in caso di fallimento*

# SCHEDULING CPU

Uno dei principali obiettivi della multiprogrammazione è quello di massimizzare l'utilizzo delle risorse e in particolare della CPU. Per raggiungere questo obiettivo, ogni CPU viene assegnata a più processi o thread:

- *lo **scheduler** deve decidere quale algoritmo utilizzare per assegnare la CPU ad un task (processo o thread)*
- *le prestazioni dello scheduler sono valutate tramite **funzioni di costo***
- *applicazioni diverse richiedono algoritmi e funzioni di costo diverse*

# ALGORITMI

Procedimento generale di scheduling:

1. *la CPU viene assegnata ad un task*
2. *qualora il processo entri in uno stato di attesa, termini, venga ricevuto un interrupt, ecc., è necessario effettuare un **context switching***
3. *per ogni **context switching**:*
    - *il task in running viene spostato nella coda di ready*
    - *un task nella coda di ready viene spostato allo stato di running*

## ALGORITMI SENZA PRELAZIONE

*La CPU non può essere sottratta a un altro task (il task deve rilasciare la CPU volontariamente).*

### FCFS (FIRST-COME FIRST-SERVED)

La CPU viene assegnata ai tast seguendo l'ordine con cui la richiedono (i task vengono quindi gestiti attraverso una coda FIFO).

Vantaggi:

- *facile da comprendere e implementare*

Svantaggi:

- *tempi di attesa relativamente lunghi, variabili e non ottimi*
- *inadatto per sistemi real-time*
- *effetto convoglio, ovvero potrebbe verificarsi che task brevi siano in coda a task lunghi e attendono tempo inutilmente*

### SJF (SHORTEST-JOB FIRST)

A ogni task viene associata la durata della sua prossima richiesta (**next CPU burst**). I task vengono quindi schedulati in ordine di durata della loro prossima richiesta.

Vantaggi:

- *si può dimostrare che SJF è un algoritmo ottimo, utilizzzando il tempo di attesa come criterio*

Svantaggi:

- *possibile l'attesa indefinita (**starvation**)*
- *difficoltà di applicazione derivata dall'impossibilità di conoscere a priori il comportamento futuro (il tempo del burst successivo è ignoto, ma si possono comunque effettuare delle stime)*

### PS (PRIORITY SCHEDULING)

A ogni processo viene associata la sua priorità:

- *la priorità è generalmente rappresentata mediante valori interi*
- *a priorità maggiore viene associata un intero minore*

La CPU viene quindi allocata al processo con la priorità maggiore.

Svantaggi:

- *possibile l'attesa indefinita (**starvation**)*

### RR (ROUND ROBIN)

Anche definito **scheduling circolare**.

E' una versione dell'algoritmo FCFS che permette la prelazione. L'utilizzo della CPU viene suddiviso in **time quantum** (porzioni temporali) e ogni task riceve la CPU per un tempo massimo pari al **quantum**, dopodiché deve rilasciarla e viene interito nuovamente nella ready queue (coda FIFO).

Svantaggi:

- *il tempo di attesa media è relativamente lungo*
- *notevole dipendenza delle prestazioni dalla durata del quanto di tempo (**quantum**)*

### SRTF (SHORTEST-REMAINING-TIME FIRST)

E' una versione di SJF che permette la prelazione.

In particolare, se viene sottomesso un task con burst più breve di quello in esecuzione, la CPU viene "prelazionata".

### MQS (MULTILEVEL QUEUE SCHEDULING)

Viene applicato a situazioni in cui i task possono essere classificati in gruppi diversi (foreground, background, di sistema, ecc.).

La **ready queue** viene suddivisa in code diverse:

- *ogni coda può essere gestita con il proprio algoritmo di scheduling*

## CONSIDERAZIONI AGGIUNTIVE

Lo **scheduler** è un task che deve essere schedulato in maniera simile agli altri task:

- nello scheduling senza prelazione:
  - lo scheduler è invocato ogni volta che un programma termina o lascia il controllo
- nello scheduling con prelazione:
  - lo scheduler è invocato periodicamente da un interrupt periodico della CPU
  - gli altri task non possono prevenire questo procedimento

Lo scheduling può essere effettuato a livello processi oppure di thread.

## SCHEDULING DEI THREAD

- *l'OS gestisce i T a livello kernel e ignora i T a livello utente (gestiti da una libreria)*
- *quindi lo scheduling può essere effettuato solo per i T a livello kernel (se esistono)*

## SCHEDULING PER SISTEMI MULTIPROCESSORE

- *il carico viene distribuito (**bilanciamento del carico**). In particolare per gli OS con code di attesa comuni a tutti i processori il carico è distribuito automaticamente*
- *Esistono diversi schemi:*
  - ***multi-elaborazione asimmetrica** → un processore master distribuisce il carico tra i processori server*
  - ***multi-elaborazione simmetrica** → ciascun processore provvede al proprio schedulling*

## SCHEDULING PER SISTEMI REAL-TIME

Tentano di rispondere in tempo reale al **verificarsi di eventi**, i quali definiscono il comportamento dello scheduling.

Esistono 2 tipi di sistemi real-time:

- ***soft real-time** → danno priorità ai processi critici ma non garantiscono le tempistiche di risposta*
- ***hard real-time** → si garantisce l'esecuzione dei task entro un tempo massimo limite*

# FUNZIONI DI COSTO

Esempi di funzioni di costo:

**Funzioni di costo**

| Funzione di costo | Descrizione | Ottimo |
|---|---|---|
| Utilizzo della CPU (CPU utilization) | Percentuale di utilizzo della CPU | [0-100%] Massimo |
| Produttività (Throughput) | Numero di processi completati nell'unità di tempo | Massimo |
| Tempo di completamento (Turnaround time) | Tempo che trascorre dalla sottomissione al completamento dell'esecuzione | Minimo |
| **Tempo di attesa (Waiting time)** | **Tempo totale passato nella coda ready (somma dei tempi trascorsi in coda)** | **Minimo** |
| Tempo di risposta (Response time) | Tempo intercorso tra la sottomissione e la prima risposta prodotta | Minimo |