

## **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

- TR
- UNIQ
- BASENAME
- SORT
- GREP

## PROCESSI

- PROCESSI SEQUENZIALI
- PROCESSI CONCORRENTI
- CARATTERISTICHE
  - IDENTIFICAZIONE DI UN PROCESSO
  - CREAZIONE DI UN PROCESSO
    - RISORSE DI UN PROCESSO
  - TERMINAZIONE DI UN PROCESSO
    - SYSCALL WAIT
    - SYSCALL WAITPID
- PROCESSI ZOMBIE
- PROCESSI ORFANI
- TEORIA AGGIUNTIVA
  - STATO DI UN PROCESSO
  - PROCESS CONTROL BLOCK (PCB)
  - CONTEXT SWITCHING
  - SCHEDULING DEI PROCESSI
- CONTROLLO AVANZATO
  - SOSTITUZIONE DI UN PROCESSO
    - SYSCALL EXEC
      - SYSCALL EXECV[P]
      - SYSCALL EXEC[LV]E
    - CONSIDERAZIONI
- SCHELETRO DI UNA SHELL UNIX
- ESECUZIONE DI UN COMANDO
  - SYSCALL SYSTEM
- SEGNALI
  - GESTIONE DEI SEGNALI
    - SYSCALL SIGNAL
    - SYSCALL KILL
    - SYSCALL RAISE
    - SYSCALL PAUSE
    - SYSCALL ALARM
  - MEMORIA LIMITATA
  - FUNZIONI RIENTRANTI
  - RACE CONDITIONS
- COMANDI DI SHELL PER PROCESSI
- COMUNICAZIONE TRA PROCESSI
  - MODELLI DI COMUNICAZIONE
  - CANALI DI COMUNICAZIONE

# 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*

```

"ciao"
'c' 'i' 'a' 'o'
9910 10510 9710 11110
011000112 011010012 011001002 011011112

"231"
'2' '3' '1'
5010 5110 4910
001100102 001100112 001100012

"231"
"231"10
111001112

```

Stringa  
File di testo o binario

Numero intero  
File di testo

Numero intero (su 1 byte)  
File binario

## 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 struttura 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	<i>help in linea</i>
-f	--force	<i>effettua le operazioni senza chiederne conferma</i>
-i	--interactive	<i>chiede conferma prima di effettuare qualsiasi operazione</i>
-r, -R	--recursive	<i>procede ricorsivamente anche nei sottodirettori</i>

## GESTIONE PERMESSI

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

Opzioni:

Compatto	Esteso	Effetto
-r, -R	--recursive	<i>procede ricorsivamente anche nei sottodirettori</i>

## VISUALIZZAZIONE TESTO

```
cat file1 file2 ...

head [ opzioni ] file ...

tail [ opzioni ] file
```

Opzioni:

Compatto	Esteso	Effetto
-l	--lines	<i>specifica il numero di righe</i>
-f	--follow	<i>rilegge in loop il file aggiornando l'output se il file viene modificato</i>

Altri comandi di visualizzazione:

```
pg [ opzioni ] file

more [ opzioni ] file

less [ opzioni ] file
```

Opzioni:

Compatto	Esteso	Effetto
spazio		<i>prossima riga</i>
return		<i>prossima riga</i>
B		<i>pagina precedente</i>
/str		<i>ricerca nel testo la prossima occorrenza di str</i>
?str		<i>ricerca nel testo la precedente occorrenza di str</i>
q		<i>termina la visualizzazione</i>

## DIFFERENZE

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

Opzioni:

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

## WORD COUNT

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

Opzioni:

Compatto	Esteso	Effetto
-c	--bytes	<i>valuta il numero di soli byte</i>
-m	--chars	<i>valuta il numero di soli byte</i>
-w	--words	<i>valuta il numero di parole</i>
-l	--lines	<i>valuta il numero di righe</i>

## LINK

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

Opzioni:

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

## 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		<i>crea l'archivio</i>
-x		<i>estrae l'archivio</i>
-z, -j, -J		<i>comprime (gzip, bzip2, 7z)</i>
-f		<i>specifica il nome dell'archivio</i>
-v		<i>verbose (stampa i messaggi)</i>

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

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

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 ricerca del match.

Esempio: **ind** in windows, **ind**ifferent, ecc.

### METACARATTERE

Uno o più caratteri con significato speciale.

Esempio: \* indica da 0 a ∞ → **b\*** = {∅, 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	<i>seleziona solo i caratteri di posizione indicata</i>
-f LIST	--fields=LIST	<i>indica la lista dei campi da selezionare (separati da virgola).</i> <i>Formato:</i> <b><i>n (=n)</i></b> <b><i>-n (≤n)</i></b> <b><i>n- (≥n)</i></b> <b><i>n1-n2 (≥n1 &amp;&amp; ≤n2)</i></b>
-d DELIM	-- delimiter=DELIM	<i>usa DELIM per dividere i campi (default: TAB)</i>

## 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 <b>set1</b>
-d	--delete	cancella i caratteri indicati nel <b>set1</b>
-s	--squeeze-repeats	sostituisce ogni sequenza di un carattere ripetuto incluso nel <b>set2</b> 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	<i>ignora gli spazi iniziali</i>
-d	--dictionary-order	<i>considera solo spazi e caratteri alfabetici</i>
-f	--ignore-case	<i>trasforma caratteri minuscoli in maiuscoli</i>
-i	--ignore-nonprinting	<i>considera solo caratteri stampabili</i>
-n	--numeric-sort	<i>confronta utilizzando un ordine numerico</i>
-r	--reverse	<i>ordine inverso</i>
-k c1[,c2]	--key=c1[,c2]	<i>ordina sulla base dei soli campi selezionati</i>
-m	--merge	<i>merge più file ordinati (senza riordinare)</i>
-o=f	--output=f	<i>scrive l'output nel <b>file f</b> invece che su <b>stdout</b></i>

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

## 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 privilegi 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 **CreateProcess**
- UNIX/Linux → un nuovo processo viene creato 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 effettuare la gestione:

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

## SYS CALL 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** si 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 della 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, execl*
- *execv, execvp, execve*

Significato lettere:

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

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 <b>pid</b> è	Si invia il segnale <b>sig</b> ...
> 0	al processo di <b>PID</b> uguale a <b>pid</b>
== 0	a tutti i processi con <b>GID</b> uguale al suo (a cui lo può inviare)
< 0	a tutti i processi con <b>GID</b> uguale al valore assoluto di <b>pid</b> (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		<i>elenca i processi di tutti gli utenti del sistema</i>
-u		<i>visualizza info più dettagliate (resident size, virtual size, ecc.)</i>
-u user		<i>visualizza i processi dell'utente &lt;user&gt;</i>
-x		<i>aggiunge all'elenco i processi che non hanno un terminale di controllo</i>
-o, -A		<i>elenca tutti i processi "running" nel sistema</i>
-f		<i>visualizza le info in formato esteso</i>
r (non -r)		<i>visualizza solo i processi "running"</i>

- **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 specificando 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*)