Sistemi Operativi

definizioni, formule ed esempi

Pietro Barbiero

Quest'opera contiene informazioni tratte da wikipedia (http://www.wikipedia.en) e dalle dispense relative al corso di Sistemi Operativi tenuto dal professor Quer Stefano del Dipartimento di Automatica e Informatica del Politecnico di Torino (IT).

Indice

Codice

Figure

Parte I

Introduzione ai sistemi operativi (SO)

Capitolo 1

Funzioni principali dei SO

1.1 Componenti di un sistema di elaborazione (bottom-up)

I componenti di un sistema di elaborazione sono:

- hardware: fornisce le risorse di elaborazione (CPU, memoria, periferiche)
- sistema operativo: controlla e coordina l'uso dell'hardware (linux, windows, mac)
- programmi applicativi: forniscono servizi agli utenti (programmi, giochi)
- utenti: fruiscono del sistema (persone, macchine)

1.2 Sistema operativo

Un sistema operativo è un software di interfaccia tra un utente o un programma applicativo e l'hardware; può essere visto come: estensione dell'hardware; gestore delle risorse

1.2.1 Servizi di un SO

I servizi forniti da un SO sono:

- interpretazione dei comandi: l'utente comunica con l'elaboratore attraverso un'interfaccia gestita dal SO
- gestione dei processi (o programmi in esecuzione): il SO gestisce tutti i processi: li crea, li sospende, li cancella, li sincronizza
- gestione della memoria (principale e secondaria): il SO organizza e ottimizza l'accesso alla memoria
- gestione dei dispositivi I/O: il SO nasconde i dettagli dei dispositivi I/O e fornisce un'interfaccia generica all'utente
- gestione di file e file-system: il SO crea, legge, scrive, cancella file e instaura meccanismi di protezione di accesso
- implementazione di meccanismi di protezione: il SO controlla e tiene traccia degli accessi da parte di utenti e processi alle risorse del sistema
- gestione di reti e sistemi distribuiti

1.3 Concetti base di un SO

1.3.1 Kernel

Il kernel è la parte centrale di un SO; il kernel gestisce memorie e processori; è l'unico programma in esecuzione per tutto il tempo in cui l'elaboratore è acceso

1.3.2 Bootstrap (o booting program)

Il bootstrap è il programma che, all'accensione del SO, carica in memoria principale il kernel del SO e lo esegue (generalmente è un programma memorizzato in una memoria ROM e caricato al power-up o al reboot)

1.3.3 System call

Una system call (chiamata di sistema) è il meccanismo usato a livello utente (processo applicativo o persona) per richiedere al SO un servizio a livello kernel; spesso sono implementate in assembler; spesso vi si accede tramite Application Program Interface (API) di alto livello (POSIX API, JAVA API, Win32/64 API)

1.3.3.1 Differenze tra system call e funzioni

Le differenze tra system call e funzioni sono:

- per ogni system call esistono più funzioni di alto livello con lo stesso nome
- le funzioni sono modificabili; le system call no
- le system call forniscono un servizio a livello kernel (o super user)

1.3.3.2 System call UNIX/Linux comuni

Le più comuni system call UNIX/Linux sono:

- per la gestione dei processi: fork, wait, exec, exit, kill
- per la gestione dei file: open, close, read, write, lseek, stat
- per la gestione dei direttori: mkdir, rmdir, unlink, mount, umount, chdir, chmod

1.3.4 Login

Il login è la procedura di accesso ad un sistema informatico; per effettuare un login è necessario fornire: username e password (memorizzata nel file /etc/passwd)

1.3.5 Shell

La shell (guscio) è l'interfaccia utente del SO; la shell legge i comandi (da terminale o da file script) dell'utente e li esegue

1.3.6 File system

Il file system (sistema di file) è l'insieme dei tipi di dati astratti utilizzati per l'organizzazione, la manipolazione e la memorizzazione dei dati (cartelle e file) di un elaboratore

1.3.7 File name

Un file name (nome del file) è il nome utilizzato per identificare in modo univoco un file memorizzato in un file system; in UNIX i caratteri che non possono essere inseriti in un file name sono: "/" (slash) e "null" (carattere nullo)

1.3.8 Path name

Il path name (nome del percorso) è una stringa composta da nomi di direttori separati da slash che indica la posizione univoca di un direttorio o di un file all'interno di un file system

1.3.8.1 Absolute Path

Un absolute path è un path name che specifica la posizione di un direttorio o di un file a partire dalla radice del file system

1.3.8.2 Relative path

Un relative path è un path name che specifica la posizione di un file a partire da una posizione diversa rispetto alla radice del file system

1.3.8.3 Caratteri speciali

Nei path name: il "." indica il direttorio corrente; i ".." indicano il direttorio padre

1.3.9 Home directory

La home directory è la cartella destinata a contenere i file personali di uno specifico utente; viene assegnata o dal SO o dall'amministratore di sistema; nei sistemi LINUX è individuata dal carattere "~" (tilde); è il direttorio a cui si accede subito dopo aver effettuato il login

1.3.10 Working directory di un processo

La working directory (direttorio di lavoro) di un processo è un direttorio del file system associato dinamicamente (cioè può essere modificato) al processo; le working directories sono i nodi del file system utilizzati come origine per interpretare i relative paths

1.3.11 Programma

Un programma è un file eseguibile dal SO memorizzato su un disco; il programma è un'entità passiva; esistono due tipi di programma:

- programma sequenziale: ogni istruzione del programma viene eseguita al termine dell'istruzione precedente
- programma concorrente (o parallelo): le istruzioni del programma possono essere eseguite contemporaneamente

1.3.12 Processo

Un processo è un programma in esecuzione allocato in memoria principale; il processo è un'entità attiva; nei SO Linux ogni processo è identificato da un numero intero non negativo

1.3.13 Thread di esecuzione

Un thread (filo) di esecuzione o sottoprocesso è una parte di un processo che: può essere eseguita contemporaneamente ad altri thread del processo; condivide risorse con gli altri thread del processo

1.3.14 Pipe

Una pipe è un flusso dati unidirezionale tra due processi

1.3.15 Deadlock

Un deadlock (stallo) è una situazione in cui due o più processi si bloccano a vicenda aspettando che uno esegua una certa azione che serve all'altro e viceversa

1.3.16 Livelock

Un livelock (stallo attivo) è una situazione in cui due o più processi non fanno alcun progresso (ma non sono bloccati)

1.3.17 Starvation

Una starvation (inedia) è una situazione in cui ad un processo viene negata in continuazione la possibilità di ottenere le risorse di cui necessita per continuare la propria esecuzione; deadlock implica starvation (di tutti i processi); starvation non implica deadlock (gli altri processi possono progredire)

Capitolo 2

Classificazione dei SO

2.1 Classificazione per dominio applicativo

I SO possono essere classificati per dominio applicativo nelle seguenti categorie:

- server
- device - embedded
- desktop

2.2 Windows

Windows è una famiglia di SO con interfaccia grafica a finestre commercializzato da Microsoft; sono state commercializzate le seguenti versioni di Windows

- 16 bit
- 16/32 bit
- 32/64 bit

2.3 MAC OS e MAC OS X

MAC OS e MAC OS X sono SO con interfaccia grafica commercializzati da Apple; le loro caratteristiche sono:

- architettura proprietaria e molto chiusa
- micro-kernel facilmente estendibile, adattabile e affidabile (grazie al fatto che ha compiti molto limitati)
- sicurezza elevata (dovuta al fatto che è poco diffuso)
- architettura e software costosi (vale più il marchio del prodotto)

2.4 UNIX/linux

UNIX/linux è una famiglia di SO; standard di UNIX/linux sono:

- ISOC
- POSIX (Portable Operating System Interface): standard che definisce il livello di portabilità di un sistema UNIX
- SUS (Single UNIX Specification): standard che definisce le caratteristiche di un generico sistema UNIX

2.4.1 Linux

Linux (che significa kernel) è una famiglia di SO coperto da licenza GNU di software libero

2.4.1.1 Distribuzioni di Linux

Le distribuzioni di Linux sono:

- CentOS: orientata al mercato aziendale
- Debian: contiene solo software libero
- Fedora: realizzata da GNU/Linux
- Mandriva: realizzata per utenti meno esperti
- Red Hat
- SuSE
- Slackware: orientata agli utenti esperti
- Ubuntu: basata su Debian; completa e semplice

2.4.1.2 Diffusione

SO Linux si trovano:

- nell'1.6% dei desktop
- nel 60% dei server

Il 95% degli effetti speciali di Hollywood sono sviluppati su SO Linux
Debian è composto da 283 milioni di righe di codice il cui sviluppo proprietario (tipico Microsoft e Apple) richiederebbe 73000 anni e 8.16 miliardi di dollari
Linux è il punto di riferimento per lo sviluppo kernel in quanto viene considerato il SO più evoluto

2.5 Confronto

2.5.1 Diffusione

Diffusione dei maggiori SO:

- Windows: 80% dei desktop (preistallato su quasi tutti i desktop)
- MAC OS X: 6.5% dei desktop (senza un computer MAC non si può utilizzare)
- Linux: 2% dei desktop; 60% dei server

2.5.2 Costi e licenze

Costi e licenze dei maggiori SO:

- Windows: 100 euro circa; software proprietario
- MAC OS X: generalmente più caro di Windows; software proprietario
- Linux: gratuito; software libero

2.5.3 Installazione

Installazione dei maggiori SO:

- Windows: fino a 60 minuti; software aggiuntivo a pagamento
- MAC OS X: preinstallato; permette l'utilizzo solo di software specifico
- Linux: da 5 a 60 minuti; software libero (per la maggior parte)

2.5.4 Stabilità

Stabilità dei maggiori SO:

- Windows: richiede riavvii frequenti
- MAC OS X: stabile
- Linux: struttura modulare estremamente stabile (va riavviato solo dopo l'aggiornamento del kernel)

2.5.5 Sicurezza

Sicurezza dei maggiori SO:

- Windows: 11.000 malware scoperti (2005); difficile liberarsi dei problemi causati
- MAC OS X: pochi virus progettati per attaccarlo
- Linux: essendo open source è estremamente difficile che venga infettato da virus

2.5.6 Aspetto

Aspetto dei maggiori SO:

- Windows: parzialmente modificabile; fornisce prompt dei comandi $MS-DOS$
- MAC OS X: più gradevole di Windows
- Linux: esistono moltissime alternative di aspetto; le shell sono integrate nella console

2.5.7 Prestazioni

Prestazioni dei maggiori SO:

- Windows: richiede moltissime risorse; lento
- MAC OS X: ha più o meno le stesse prestazioni di Windows; eccellente per applicazioni grafiche; fatica a gestire in modo efficiente il sovraccarico della CPU
- Linux: massima efficienza nella gestione delle risorse hardware; prestazioni paragonabili alle workstation; su applicazioni in cui la CPU è sovraccarica risulta $2-3$ volte più veloce di MAC OS X

Capitolo 3

File-system Linux

3.1 File (o archivio)

Un file è un contenitore di dati in formato elettronico

3.1.1 Filename

Il nome di un file può essere una sequenza di caratteri qualunque (eccetto: / \ ” ’ * ; ? [ ] ( ) ∼ ! $ { } < > # @ & |)
Se il nome di un file inizia con il carattere “.” il file è nascosto
Non esiste (formalmente) l’estensione di un file; esistono estensioni utilizzate per scopi specifici

3.1.2 File path

Il path (assoluto o relativo) di un file è una stringa di nomi di directory separate da “/” che indica in modo univoco la posizione di un file all’interno di un file-system

3.2 Codifica dei caratteri

I codici standard per la codifica dei caratteri sono:

- Extended ASCII (American Standard Code for Information Interchange): composto da 255 caratteri
- Unicode (implementato come UCS o UTF): composto da 110000 caratteri

3.2.1 File di testo

Un file di testo è un file i cui bit sono organizzati a gruppi (8, 16, ...) ognuno dei quali rappresenta caratteri (lettere, numeri, ...) di un codice (ASCII o Unicode)

3.2.2 File binario

Un file binario è un file i cui bit (singolarmente o a gruppi) possono rappresentare qualunque tipo di dati (anche non caratteri); i file binari necessitano di applicazioni in grado di interpretare il loro contenuto; i file binari sono più compatti dei file di testo (non sono codificati)

3.3 Serializzazione

La serializzazione è un processo di traduzione di una struttura (e.g.: "struct" in C) in un formato tale per cui la memorizzazione e la trasmissione della struttura avvenga come un'unica entità (e non come oggetto composto da entità diverse)

3.4 Parametri di un file-system

I parametri di un file-system sono:

- efficienza: velocità nel localizzare un file
- convenienza: semplicità per un utente di identificare i propri file
- organizzazione: raggruppamento delle informazioni in base alle caratteristiche

3.5 Directory (direttorio)

Una directory è un nodo (di un albero) o un vertice (di un grafo) contenente file e informazioni riguardanti tali file

3.5.1 File-system a un livello

Un file-system a un livello è un file-system in cui tutti i file sono contenuti in un'unica directory

- vantaggi: efficienza (semplice)
- svantaggi: convenienza (filename univoci), organizzazione (gestione multi utente complessa)

3.5.2 File-system a due livelli

Un file-system a un livello è un file-system in cui ogni utente ha una propria directory

- vantaggi: efficienza (semplice)
- svantaggi: convenienza parziale (filename univoci per ogni utente), organizzazione parziale (ogni utente ha solo la sua home)

3.5.3 File-system ad albero

Un file-system ad albero è un file-system in cui ogni directory può contenere come entry altre directory

- vantaggi: convenienza (filename diversi solo nella stessa directory), organizzazione (a discrezione dell'utente)
- svantaggi: efficienza parziale (ricerche più lunghe)

3.5.4 File-system a grafo

Un file-system a grafo è un file-system in cui ogni directory può contenere come entry: altre directory o link ad altri file (Linux non consente link a directory per evitare cicli)

- vantaggi: convenienza (filename diversi solo nella stessa directory), organizzazione (a discrezione dell'utente), condivisione di file e directory (tramite i link)
- svantaggi: efficienza (ricerche più lunghe e gestione di eventuali cicli complessa)

3.5.4.1 Link (collegamento)

Un link è un riferimento (puntatore) ad un'altra entry preesistente; un file viene eliminato quando viene eliminato il suo ultimo link (occorre memorizzare un contatore del numero di link)

3.6 Allocazione

L'allocazione è il processo attraverso il quale il SO riserva una parte della memoria per la memorizzazione di un file

3.6.1 Allocazione contigua

L'allocazione contigua è una tecnica di allocazione in cui ogni file occupa blocchi di memoria contigui; per memorizzare un file è necessario specificare l'indirizzo del primo blocco e la dimensione del file

3.6.1.1 Prestazioni

Le prestazioni dell'allocazione contigua sono:

- vantaggi: tecnica semplice (per memorizzare un file sono necessari solo 2 parametri), permette accessi sequenziali immediati, permette accessi diretti semplici (tramite offset)
- svantaggi: politica di allocazione complessa e mai efficiente (frammentazione esterna, si creano "buchi"), problemi di allocazione dinamica (la dimensione del file non può aumentare liberamente)

3.6.2 Allocazione concatenata

L'allocazione concatenata è una tecnica di allocazione in cui ogni file occupa blocchi di memoria organizzati in una lista concatenata (ogni blocco contiene un puntatore al blocco successivo); per memorizzare un file è necessario specificare l'indirizzo del primo e dell'ultimo blocco

3.6.2.1 Prestazioni

Le prestazioni dell'allocazione concatenata sono:

- vantaggi: permette allocazione dinamica di file, elimina la frammentazione esterna
- svantaggi: efficiente solo per accessi sequenziali (non per accessi diretti), spazio per memorizzazione puntatori, poco affidabile (se si perde un puntatore si butta tutto)

3.6.2.2 File Allocation Table (FAT)

La FAT è una tabella di puntatori a blocchi di memoria; la FAT è molto lenta perché per la lettura di un blocco di memoria sono necessari 2 accessi alla memoria

3.6.3 Allocazione indicizzata

L'allocazione indicizzata è una tecnica di allocazione in cui per ogni file esiste un blocco di memoria che contiene l'elenco dei puntatori ai blocchi che compongono il file; per memorizzare un file è necessario specificare i suoi puntatori; per leggere un file è necessario specificare il puntatore al blocco indice

3.6.3.1 Prestazioni

Le prestazioni dell'allocazione indicizzata sono:

- vantaggi: permette allocazione dinamica di file, elimina la frammentazione esterna, permette accesso diretto efficiente, affidabile
- svantaggi: gestione dei blocchi indice (e.g.: inode)

3.7 Inode

L'inode è un blocco di memoria (nei sistemi Unix/linux) associato ad ogni file che contiene tutte le informazioni relative a quel file; l'inode contiene 12 puntatori diretti a blocchi dati del file; 3 puntatori indiretti a blocchi dati del file; con questa tecnica è possibile memorizzare file di dimensione pari a 2^{60} *byte*

3.7.1 Puntatore indiretto

Un puntatore indiretto è un puntatore che punta ad un blocco di memoria che contiene puntatori a blocchi dati

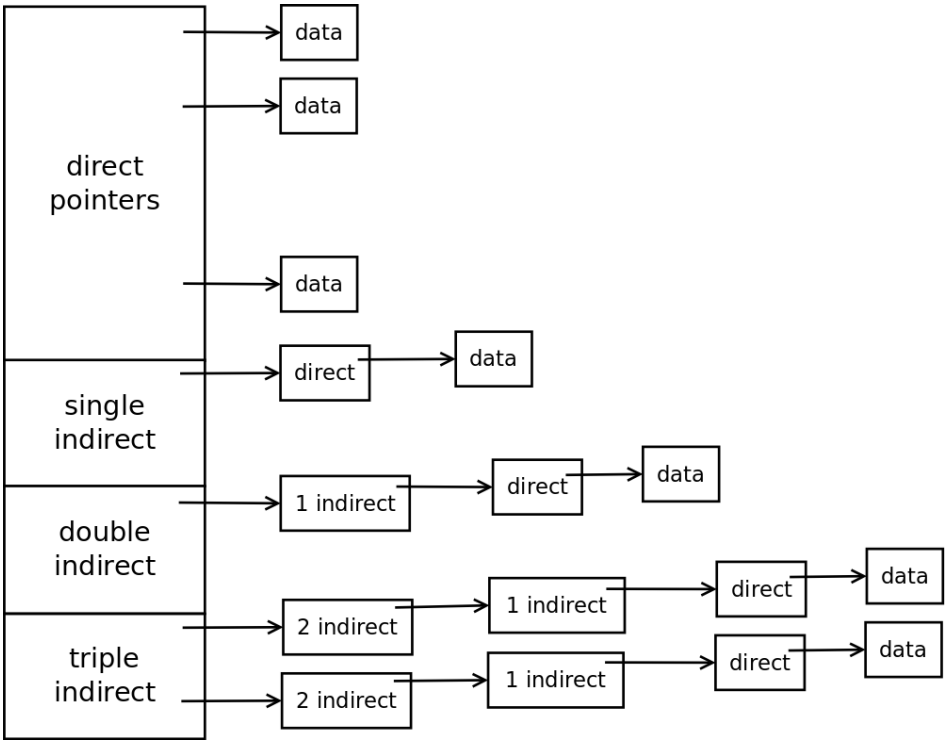


Figura 3.1: Puntatori diretti e indiretti dell'inode

3.7.2 Hard link

Un hard link è una directory entry che punta all'inode di un file; un file è fisicamente rimosso quando tutti i suoi hard link sono stati rimossi
Non è possibile creare hard link: verso directory e verso file memorizzati su altri file-system

3.7.3 Soft link

Un soft link è una directory entry che contiene il path name di un file (cioè il percorso della entry che punta all'inode del file)

Parte II

Comandi Linux

Capitolo 4

Comandi shell

4.1 Shell

La shell è un'interprete di comandi che permette all'utente di comunicare con il SO attraverso una serie di funzioni predefinite

4.1.1 Bourne again shell (bash)

Bash è una shell testuale del progetto GNU

4.2 Sintassi dei comandi UNIX-like

La sintassi dei comandi UNIX-like è: <`comando`> `[opzioni][argomenti]`
I comandi troppo lunghi possono essere continuati nella riga successiva tramite il carattere "\" al fondo della prima riga
Si possono fornire più comandi sulla stessa riga separandoli con il carattere ";"

4.2.1 Comandi in foreground e in background

Un comando può essere eseguito in:

- foreground: <`comando`>: la shell esegue una `fork()`: il figlio esegue il comando e la shell aspetta il figlio con una `wait()`
- background: <`comando`> `&`: la shell esegue una `fork()`: il figlio esegue il comando e la shell torna libera

4.3 Comandi bash

Per iniziare una sessione di lavoro è necessario fornire al SO due parametri: username e password oppure username@hostname (se ci si connette da un terminale remoto)

4.3.1 Superuser: `sudo`

Il comando `sudo` permette di eseguire comandi come superuser

4.3.2 Uscita: `exit`, `logout`, `<ctrl+d>`

I comandi per terminare una sessione di lavoro sono:

- `username@pcname~$ exit`
- `username@pcname~$ logout`
- `username@pcname~$ <ctrl+d>`

4.3.3 Aiuto: `man`, `apropos`, `whatis`, `whereis`

I comandi utili per visualizzare la documentazione sui comandi sono:

- pagina del manuale del comando:

```
man <comando>
```

- comandi legati al comando specificato:

```
apropos <comando>
```

- breve spiegazione della funzione del comando:

```
whatis <comando>
```

- path in cui si trova l'eseguibile del comando:

```
whereis <comando>
```

4.3.4 Completamento automatico: `<tab>`

Il comando (tasto) `<tab>` fornisce il completamento automatico di un comando

4.3.5 Storico dei comandi recenti: `<frecce>`

I comandi (tasti) ↑ e ↓ permettono la navigazione tra i comandi utilizzati di recente

4.4 Navigazione nel file system

4.4.1 Path della working directory: `pwd`

Il comando `pwd` (print working directory) mostra il path e il nome della directory corrente

```
pwd [OPTION]...
```

4.4.2 Contenuto della working directory: `ls`

Il comando `ls` (list segment) visualizza l'elenco delle informazioni sui file e il contenuto delle directory:

```
ls [OPTION]...  [FILE]...
```

4.4.2.1 Opzioni

Le opzioni di `ls` sono:

- `ls -a`: elenca i file nascosti
- `ls -l`: output in formato esteso
- `ls -g`: include l'indicazione del gruppo
- `ls -t`: elenca i file in ordine temporale
- `ls -r`: elenca i file in ordine (temporale o alfabetico) inverso
- `ls -R`: elenca anche i file che si trovano nelle subdirectory

4.4.2.2 Formato esteso di `ls`

Il formato esteso del comando `ls` fornisce le seguenti informazioni:

- numero di blocchi del sotto albero (e.g.: `totale 16`); generalmente un blocco contiene 1 kByte (e.g.: 16 kB)
- tipo e diritti di accesso per tre gruppi di utenti (e.g.: `drwxrwxr-x`)
  - user (owner) (e.g.: u = `rwx`)
  - group (e.g.: g = `rwx`)
  - others (e.g.: o = `r-x`)
  - tipo
    - `-`: file normale
    - `d`: directory
    - `s`: socket file
    - `l`: link file
  - diritti di accesso
    - `r`: read (file: lettura)(directory: elenco file)
    - `w`: write (file: scrittura)(directory: creazione/cancellazione file)
    - `x`: execute (file: esecuzione)(directory: attraversamento)
- numero di link
- username del proprietario (e.g.: `owner`)
- gruppo del proprietario (e.g.: `ownergroup`)
- spazio occupato in byte
- data dell'ultima modifica (mese giorno ora)
- nome del file o della cartella

4.4.3 Cambiare working directory: `cd`

Il comando `cd` (change directory) cambia la directory corrente nella directory `DEST`

```
cd [OPTION]...  DEST
```

```
username@pcname~/Scrivania/prova/c1$ ls -la
totale 16
drwxrwxr-x 3 owner ownergroup 4096 ott 2 11:23 .
drwxrwxr-x 3 owner ownergroup 4096 ott 2 11:17 ..
drwxrwxr-x 2 owner ownergroup 4096 ott 2 11:17 c2
-rw-rw-r-- 1 owner ownergroup 0 ott 2 11:16 f
-rw-rw-r-- 1 owner ownergroup 36 ott 2 11:23 .fn
-rw-rw-r-- 1 owner ownergroup 0 ott 2 11:20 .fn~
```

Codice 4.1: comando ls

4.4.4 Confronto: `diff`

Il comando `diff` confronta due file o due directory

```
diff [OPTION]...  FILE1 FILE2
```

Il comando specifica:

- `a` (added): le righe aggiunte
- `d` (deleted): le righe cancellate
- `c` (changed): le righe cancellate

4.4.4.1 Opzioni

Le opzioni di `diff` sono:

- `-b`: ignora gli spazi a fine riga e collassa gli altri
- `-i`: ignora la differenza maiuscolo/minuscolo
- `-w`: ignora completamente la spaziatura

e.g.:

```
username@pcname~/Scrivania/prova/$ diff -b files filesc
1c1
< io sono un fungo
---
> io sono un fungo...
3d2
< ci sono tanti tanti funghi in giro??
4a4
> ciao a tutti!!!
```

Codice 4.2: comando diff

4.5 Manipolazione directory

4.5.1 Creazione cartella: `mkdir`

Il comando `mkdir` (make directory) crea una nuova directory avente path `DEST`

```
mkdir [OPTION]...  DEST...
```

4.5.2 Rimozione cartella (vuota): `rmdir`

Il comando `rmdir` (remove directory) elimina la directory non vuota avente path `DEST`

```
rmdir [OPTION]...  DEST...
```

4.5.3 Spostamento (o ridenominazione): `mv`

Il comando `mv` (move) effettua lo spostamento o la ridenominazione di file e/o directory `FILE` nella directory `DEST` (non ha l'opzione `--recursive`)

```
mv [OPTION]...  SOURCE...  DEST
```

e.g.:

```
username@pcname~/Scrivania/prova/c1$ mv -f .fn ./c2
```

4.5.4 Rimozione: `rm`

Il comando `rm` (remove) effettua la cancellazione di file e/o directory `FILE`

```
rm [OPTION]...  FILE...
```

e.g.:

```
username@pcname~/Scrivania/prova/c1$ rm -i ./c2/f ./c2/.fn
rm:  rimuovere file regolare vuoto ./c2/f?  s
rm:  rimuovere file regolare ./c2/.fn?  n
```

Codice 4.3: comando rm

4.5.5 Copia: `cp`

Il comando `cp` (copy) effettua la copia di file e/o directory `SOURCE` nella directory `DEST`

```
cp [OPTION]...  SOURCE...  DEST
```

e.g.:

```
username@pcname~/Scrivania/prova/c1$ cp f c2
```

Codice 4.4: comando cp

4.5.5.1 Opzioni (tranne `ln`)

Le opzioni dei comandi di manipolazione del file system sono:

- `-f = --force`: non chiede conferma dell'operazione
- `-i = --iterative`: chiede conferma per ciascun file
- `-r = -R = --recursive`: opera ricorsivamente su tutti i file

4.5.6 Collegamento: `ln`

Il comando `ln` crea un collegamento ad un file o ad una directory

```
ln [OPTION]...  SOURCE [DEST]
```

Di default il comando crea un hard-link (collegamento fisico) e se non è specificato il path di destinazione crea un collegamento con lo stesso nome della sorgente nella directory corrente

4.5.6.1 Opzioni

Le opzioni di `ln` sono:

- `-s`: crea un soft-link (collegamento simbolico)
- `-f`: rimuove eventuali file di destinazione esistenti

4.6 Operazioni sui file

4.6.1 Contenuto: `less`

Il comando `less` mostra il contenuto del file avente path e nome specificati da `DEST`

```
less DEST...
```

4.6.1.1 Opzioni

Le opzioni di `less` sono:

- `<spazio>`: mostra la pagina successiva
- `<return>`: mostra la riga successiva
- `<b>`: mostra la pagina precedente
- `/<stringa>`: mostra l'occorrenza successiva della stringa specificata
- `?<stringa>`: mostra l'occorrenza precedente della stringa specificata
- `<q>`: termina la visualizzazione

4.6.2 Contenuto di più file: `cat`

Il comando `cat` visualizza il contenuto dei file specificati

```
cat [OPTION]...  [FILE]...
```

4.6.3 Contenuto iniziale: `head`

Il comando `head` visualizza il contenuto delle prime righe di un file

```
head [OPTION]...  [FILE]...
```

4.6.3.1 Opzioni

L'opzione di `head` è `-<n>` che specifica il numero delle prime n righe da visualizzare

4.6.4 Contenuto finale: `tail`

Il comando `tail` visualizza il contenuto delle ultime righe di un file

```
tail [OPTION]...  [FILE]...
```

4.6.4.1 Opzioni

Le opzioni di `tail` sono:

- `-<n>`: specifica il numero delle ultime n righe da visualizzare
- `+<n>`: visualizza tutto il file tranne le prime n righe
- `-r`: visualizza le righe in ordine inverso
- `-f`: rilegge continuamente il file

4.6.5 Righe, parole, byte: `wc`

Il comando `wc` conta il numero di righe, il numero di parole e il numero di byte di un file

```
wc [OPTION]...  [FILE]...
```

4.6.5.1 Opzioni

Le opzioni di `wc` sono:

- `-l`: conta solo il numero di righe
- `-w`: conta solo il numero di parole
- `-c`: conta solo il numero di byte

4.7 Manipolazione dei permessi

4.7.1 Cambiare i permessi: `chmod`

Il comando `chmod` consente di cambiare i permessi di un file

```
chmod [OPTION]...  [MODE][FILE]...
```

e.g.:

```
username@pcname~/Scrivania/prova/c1$ chmod go-rwx prova.c
```

Codice 4.5: comando chmod

4.7.1.1 Codifica dei diritti di accesso

I diritti di accesso sono codificati in due modi:

- codifica ottale (e.g.: `rwx--x---` $\implies$ 710: u = `111` = 7; g = `001` = 1; o = `000` = 0)
- codifica simbiolica (e.g.: `u+rw`)
  - lettere: `u`(ser), `g`(roup), `o`(ther), `a`(ll)
  - simboli: `+`(add), `-`(subtract), `=`(untouched)
  - caratteri dei diritti: `r`, `w`, `x`

4.8 Gestione dei processi

4.8.1 Lista dei processi: `ps`

Il comando `ps` visualizza i processi attivi

```
ps [OPTION]...
```

4.8.1.1 Opzioni

Le opzioni di `ps` sono:

- `-e`: visualizza tutti i processi
- `-l`: visualizza i dettagli di ogni processo

4.8.2 Lista dei processi runtime: `top`

Il comando `top` visualizza un pannello runtime dei processi attivi

```
top
```

Capitolo 5

Strumenti di programmazione

5.1 Editor

Un editor è un programma di composizione di testi

5.1.1 Editor Unix/linux

I più comuni editor per Unix/linux sono: VI (VIM), e Emacs

5.2 Compiler

Un compiler è un programma che traduce istruzioni scritte in un codice sorgente (linguaggio di programmazione) in istruzioni scritte in codice oggetto (linguaggio macchina, comprensibile all'elaboratore)

5.2.1 GNU Compiler Collection (GCC)

GCC è un compilatore (e linker) multi target del progetto GNU

5.2.2 Compilazione: `gcc`

Il comando `gcc` compila (e linka) i file sorgente `FILEIN` nel file oggetto `FILEOUT`

```
gcc [OPTION]...  [FILEOUT] [FILEIN]
```

e.g.: compilazione, link ed esecuzione:

```
username@pcname~/Scrivania/prova/c1$ ls
prova.c
username@pcname~/Scrivania/prova/c1$ gcc -c prova.c; ls
prova.c prova.o
username@pcname~/Scrivania/prova/c1$ gcc -o exefile prova.o; ls
exefile prova.c prova.o
username@pcname~/Scrivania/prova/c1$ ./exefile
Hello world!!!
```

Codice 5.1: gcc (compilazione e link)

5.2.2.1 Opzioni

Le opzioni di `gcc` sono:

- `-c [FILEIN]...`: esegue la compilazione dei file
- `-o [FILEOUT] [FILEIN]...`: esegue il link dei file nell'eseguibile
- `-g`: indica a GCC di non ottimizzare il codice e di inserire informazioni extra per poter effettuare il debug
- `-Wall`: stampa warning per tutti i possibili errori nel codice
- `-I [DEST]`: specifica il path in cui cercare gli header file
- `-Im`: specifica l'utilizzo della libreria matematica
- `-L [DEST]`: specifica il path in cui cercare librerie preesistenti

5.3 Debugger

Il debugger è un software utilizzato per l'analisi del comportamento di un altro software allo scopo di individuare eventuali errori (bug)

5.3.1 GNU Debugger (GDB)

GDB è un debugger del progetto GNU; può essere usato come tool "stand-alone" nella shell o su emacs

5.3.2 Debugging: `gdb`

Il comando `gdb` attiva il debugger GDB

5.3.2.1 Comandi per `gdb`

I comandi per GDB sono:

- esecuzione step-by-step
  - `run` o `r`
  - `next` o `n`
  - `next <numeroStep>`
  - `step` o `s`
  - `step <numeroStep>`
  - `stepi` o `si`
  - `finish` o `f`
- comandi per breakpoint
  - `info break`
  - `break` o `b` o `<ctrl-x-blank>`
  - `break <numeroLinea>`

  - `break <nomeFunzione>`
  - `<fileName>:<numeroLinea>`
  - `disable <numeroBreak>`
  - `enable <numeroBreak>`

- comandi di visualizzazione
  - `print` o `p`
  - `print <espressione>`
  - `display <espressione>`

- operazioni sullo stack
  - `down` o `d`
  - `up` o `u`
  - `info args`
  - `info locals`

- comandi di listing del codice
  - `list` o `l`
  - `list <n>`
  - `list <f>`

- comandi vari
  - `file <fileName>`
  - `exec <fileName>`
  - `kill`

5.4 Makefile

Un makefile è un file di testo che contiene: il grafo delle dipendenze degli input e gli script necessari per eseguire in modo automatico delle operazioni

5.4.1 Istruzioni di un makefile

Un makefile contiene cinque tipi di istruzioni:

- regole esplicite: specificano come aggiornare un file specifico
- regole implicite: specificano come aggiornare una classe di file
- definizioni di variabili
- direttive: indicano quando leggere altri makefile e/o quando ignorare porzioni di makefile
- commenti (cominciano con il carattere "#")

5.4.2 Struttura di una regola

Una regola è composta da:

- target: il nome della regola
- dipendenze: i file da cui i target dipendono
- comandi: comandi che costruiscono i file del target a partire dalle dipendenze

```
[TARGET]: [DEPENDENCIES]...
<tab>[COMMAND]...
```

e.g.:

```
#variabili
CC = gcc
#CC = g++
FLAGS = -Wall -g
FILES = *bak* *~ core

#compila il programma
target1:  prova.c
    $(CC) $(FLAGS) -c prova.c funz.h funz.c
    $(CC) $(FLAGS) -o exefile prova.o funz.o

#esegue il programma e pulisce la cartella
exeandclean:  target1
    ./exefile
    rm -rf $(FILES)
```

Codice 5.2: makefile

5.4.3 Utility per i makefile: `make`

Il comando `make` esegue le istruzioni contenute in un makefile

```
make [-f][MAKEFILE] [OPTION]...  [TARGETS]...
```

L'opzione `-f` permette di eseguire makefile diversi dallo standard (`Makefile`)
e.g.:

```
username@pcname~/Scrivania/prova/$ make -f mymakefile exeandclean
gcc -c prova.c funz.h funz.c
gcc -o exefile prova.o funz.o
./exefile
Hello world!!!
rm -rf *bak* *~ core
```

Codice 5.3: make

5.4.3.1 Opzioni

Le opzioni di `make` sono:

- `-n`: non esegue i comandi ma li stampa solo
- `-i` o `--ignore-errors`: ignora eventuali errori
- `-d` o `--debug=[OPTION]` stampa informazioni di debug durante la compilazione
  - `a` (all): stampa tutte le informazioni
  - `b` (basic): stampa le informazioni di base
  - `v` (verbose): stampa le informazioni basic più altro
  - `i` (implicit): stampa le informazioni verbose più altro

Capitolo 6

Comandi POSIX

6.1 Manipolazione file: I/O UNIX

La manipolazione di file in ambiente UNIX/linux si può effettuare attraverso cinque funzioni di I/O della libreria POSIX (corrispondenti ad altrettante system call): open, read, write, close, lseek

6.1.1 Apertura: `open()`

La funzione `open()` apre il file specificato in `path` in una modalità descritta da `flags` e ne definisce i permessi `mode` (opzionale); restituisce: il descrittore del file in caso di successo (intero) e il valore -1 in caso di errore

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int flags, node_t mode);
```

Il `flags` si ottiene mediante le costanti contenute nel file "fcntl.h" (eventualmente in OR):

- costanti obbligatorie
  - `O_RDONLY`: accesso solo in lettura
  - `O_WRONLY`: accesso solo in scrittura
  - `O_RDWR`: accesso in lettura e scrittura
- costanti opzionali (`O_CREATE`, `O_EXCL`, `O_TRUNC`, `O_APPEND`, `O_SYNC`)

Il `mode` si ottiene tramite i comandi (eventualmente in OR):

- `S_I[RWX]USR`
- `S_I[RWX]GRP`
- `S_I[RWX]OTH`

6.1.2 Lettura: `read()`

La funzione `read()` legge dal file `fd` un numero di byte pari a `nbytes` e li memorizza in `buff`; restituisce: il numero di byte letti in caso di successo, il valore -1 in caso di errore e il valore 0 in caso di `EOF`

```
#include <sys/types.h>
#include <unistd.h>
int read(int fd, void *buf, size_t nbytes);
```

6.1.3 Scrittura: `write()`

La funzione `write()` scrive sul file `fd` un numero di byte pari a `nbytes` di `buff`; restituisce: il numero di byte scritti in caso di successo e il valore -1 in caso di errore

```
#include <sys/types.h>
#include <unistd.h>
int write(int fd, void *buf, size_t nbytes);
```

e.g. (copia di un file in un altro):

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#define BUFFSIZE 4096

int main(int argc, char *argv[]){
    int nR, nW, fdR, fdW;
    char buf[BUFFSIZE];

    if(argc!=3){
        fprintf(stderr, ``Error in number of parameters'');
    }

    fdR = open(argv[1], O_RDONLY);
    fdW = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
    if(fdR==(-1) || fdW==(-1)){
        fprintf(stdout, ``Error Opening a File.\n'');
        exit(1);
    }

    while((nR = read(fdR, buf, BUFFSIZE)) > 0){
        nW = write (fdW, buf, nR);
        if (nR != nW)
            fprintf(stderr,``Error:  Read %d, Write %d).\n'', nR, nW);
    }
    if (nR < 0)
        fprintf(stderr, ``Write Error.\n'');

    close(fdR);
    close(fdW);
    exit(0);
}
```

Codice 6.1: funzioni I/O Linux (main.c)

```
OPT = -Wall -g
FILES = *~

target:
    gcc $(OPT) -c main.c
    gcc $(OPT) -o exefile main.o
    ./exefile in out
    rm -rf $(FILES)
    less out
```

Codice 6.2: funzioni I/O Linux (makefile)

6.1.4 Chiusura: `close()`

La funzione `close()` chiude il file `fd` (tutti i file vengono chiusi automaticamente al termine del processo); restituisce: il valore 0 in caso di successo e il valore -1 in caso di errore

```
#include <unistd.h>
int close(int fd);
```

6.2 Manipolazione directory

La manipolazione delle directory in ambiente UNIX/linux si può effettuare attraverso otto funzioni della libreria POSIX (corrispondenti ad altrettante system call): stat, getchw, chdir, mkdir, rmdir, opendir, dirent, closedir

6.2.1 Entry: `stat()`

La funzione `stat()` restituisce un puntatore alla struttura `sb` (`struct stat`) contenente tutte le informazioni del file indicato da `path`; restituisce: il valore 0 in caso di successo; il valore -1 in caso di errore

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *path, struct stat *sb);
```

6.2.1.1 Struttura `struct stat`

La struttura `struct stat` contiene tutte le informazioni riguardanti un file

```
struct stat {
    node_t st_mode;   /* tipo */
    ino_t st_ino;   /* numero di inode */
    dev_t st_dev;
    ...
};
```

Il tipo del del file `st_node` si può ricavare attraverso le funzioni macro:

- `S_ISREG(st_mode)`: file normale
- `S_ISDIR(st_mode)`: directory
- `S_ISBLK(st_mode)`: block file

- `S_ISCHR(st_mode)`: character file
- `S_ISFIFO(st_mode)`: coda FIFO
- `S_ISSOCK(st_mode)`: socket file
- `S_ISLINK(st_mode)`: link simbolico

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

int main(int argc, char *argv[]){
    struct stat buff;
    char *s;
    s = malloc(25*sizeof(char));

    if(stat(argv[1],&buff) < 0)
        fprintf(stdout, ``stat error\n'');

    if(S_ISREG(buff.st_mode))
        strcpy(s, ``un file'');
    else if(S_ISDIR(buff.st_mode))
        strcpy(s, ``una directory'');

    printf(``\n\n**********\n L'oggetto \``%s\'' e':
            %s\n**********\n\n'', argv[1], s);

    return 0;
}
```

Codice 6.3: funzione stat() (main.c)

```
target:
    gcc -c main.c
    gcc -o exefile main.o
    ./exefile main.c
    ./exefile .
```

Codice 6.4: funzione stat() (makefile)

6.2.2 Path della working directory: `getcwd()`

La funzione `getcwd()` restituisce nella stringa `buf` di dimensione `size` il path della working directory; restituisce: il path in caso di successo; `NULL` in caso di errore

```
#include <unistd.h>
char* getcwd(char *buf, int size);
```

6.2.3 Cambiare working directory: `chdir()`

La funzione `chdir()` cambia la directory corrente nella directory `*path`; restituisce: il valore 0 in caso di successo; il valore -1 in caso di errore

```
#include <unistd.h>
int chdir(char *path);
```

6.2.4 Creazione cartella: `mkdir()`

La funzione `mkdir()` crea una nuova directory nel `path` indicato; restituisce: il valore 0 in caso di successo; il valore -1 in caso di errore

```
#include <unistd.h>
#include <sys/stat.h>
int mkdir(const char *path, mode_t mode);
```

6.2.5 Rimozione cartella: `rmdir()`

La funzione `rmdir()` elimina la directory avente `path` indicato; restituisce: il valore 0 in caso di successo; il valore -1 in caso di errore

```
#include <unistd.h>
#include <sys/stat.h>
int rmdir(const char *path);
```

6.2.6 Apertura cartella: `opendir()`

La funzione `opendir()` apre la cartella `filename` in lettura; restituisce: il puntatore alla cartella in caso di successo; `NULL` in caso di errore

```
#include <dirent.h>
DIR* opendir(const char *filename);
```

6.2.7 Lettura cartella: `readdir()`

La funzione `readdir()` legge il contenuto (le entry) della cartella puntata da `dp`; restituisce: il puntatore alla cartella in caso di successo; `NULL` in caso di errore o al termine della lettura

```
#include <dirent.h>
struct dirent readdir(DIR *dp);
```

6.2.7.1 Struttura `struct dirent`

La struttura `struct dirent` contiene tutte le informazioni riguardanti una cartella

```
struct dirent {
    ino_t d_ino;   /* numero di inode */
    char d_name[NUM_MAX+1];   /* nome entry */
    ...
};
```

6.2.8 Chiusura cartella: `closedir()`

La funzione `closedir()` chiude la cartella puntata da `dp`; restituisce: il valore 0 in caso di successo; il valore -1 in caso di errore

```
#include <dirent.h>
int readdir(DIR *dp);
```

Parte III

Processi

Capitolo 7

Introduzione

7.1 Algoritmo

Un algoritmo è un procedimento logico che permette la risoluzione di un problema in un numero finito di passi

7.2 Programma

Un programma è un'entità passiva (file in memoria) che formalizza un algoritmo attraverso un linguaggio di programmazione

7.3 Processo

Un processo è un'entità attiva (operazioni compiute dal processore) che corrisponde ad un'astrazione di un programma in esecuzione; un processo è composto da:

- codice sorgente
- area dati (variabili statiche globali)
- stack (variabili statiche locali)
- heap (variabili dinamiche)

7.3.1 Processi sequenziali

I processi sequenziali sono processi in cui ogni operazione viene eseguita dopo il termine di quella precedente (comportamento deterministico)

7.3.2 Processi concorrenti

I processi concorrenti sono processi in cui le operazioni possono essere eseguite contemporaneamente (comportamento non deterministico)
La concorrenza può essere: reale (sistemi multiprocessore e multi-core) o fittizia

7.3.3 Processi automatici

I processi automatici sono processi che vengono eseguiti al bootstrap e terminati allo shut-down (attesa messaggi posta elettronica, controllo virus)

7.3.4 Processi su richiesta dell'utente

I processi su richiesta dell'utente sono processi la cui esecuzione viene avviata in modo esplicito da parte dell'utente (stampante, browser)

7.4 Stati di un processo

Gli stati di un processo durante la sua esecuzione sono:

- new: il processo viene creato e sottomesso al SO
- running: il processo è in esecuzione
- ready: il processo è pronto per l'esecuzione e in attesa di risorse dal processore
- waiting: il processo è in attesa di risorse da parte del sistema
- terminated: il processo termina e rilascia le risorse utilizzate

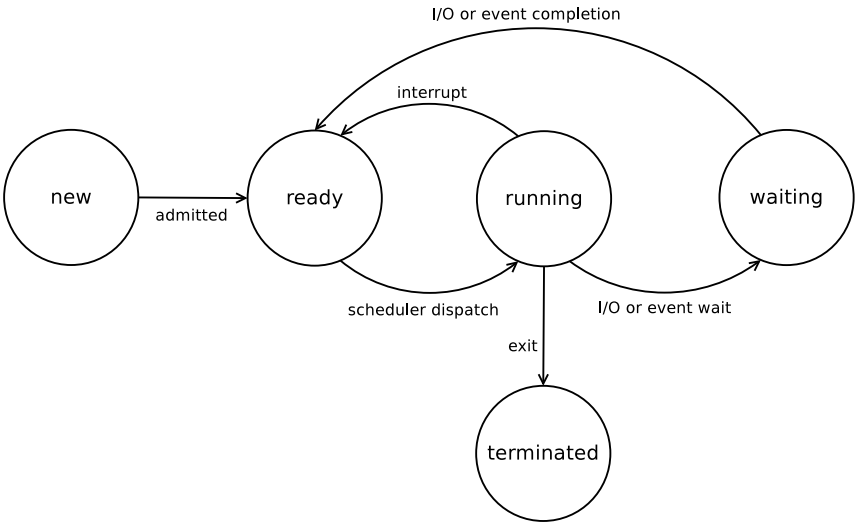


Figura 7.1: Stati di un processo

7.4.1 Context switch

Un context switch (cambiamento di contesto) è uno stato del SO in cui avviene un cambio del processo correntemente in esecuzione sulla CPU; il context switch genera un ritardo temporale tra l'esecuzione di un processo e l'altro

7.5 Process Control Block (PCB)

Il PCB è un blocco di dati associato ad ogni processo; tali dati riguardano:

- stato del processo
- program counter (indirizzo dell'istruzione successiva)
- registri della CPU
- informazioni per lo scheduling della CPU

- informazioni per la gestione della memoria
- informazioni sulle operazioni di I/O

7.6 Scheduler

Lo scheduler è un programma che stabilisce un ordinamento temporale per l'esecuzione dei processi attraverso algoritmi di scheduling; lo scheduler si occupa di:

- inserisce i PCB dei processi che richiedono una risorsa in una coda (ogni coda si riferisce ad un possibile stato dei processi: coda di ready, coda di running,...)
- preleva dalle code i processi che andranno eseguiti

7.7 Architetture parallele

Esistono diverse architetture parallele:

- Single Instruction Single Data (SISD): in un certo istante una singola istruzione viene eseguita da un singolo processo su un singolo dato (no parallelismo; bit)
- Single Instruction Multpile Data (SIMD): in un certo istante una singola istruzione viene eseguita da più processi su più dati (dati parallelizzati; numeri)
- Multiple Instruction Single Data (MISD): in un certo istante più istruzioni vengono eseguite da più processi su un singolo dato
- Multiple Instruction Multiple Data (MIMD): in un certo istante più istruzioni vengono eseguite da più processi su più dati (parallelismo su dati e su istruzioni)

7.7.1 Speed-up

I vantaggi in termini di tempo ottenibili da architetture concorrenti sono lineari (solo per un numero ridotto di processori)

7.8 Grafo di precedenza

Un grafo di precedenza è un grafo aciclico diretto in cui:

- i nodi corrispondono a istruzioni o a processi
- gli archi corrispondono a condizioni di precedenza: ogni nodo può essere eseguito solo dopo che il padre è terminato

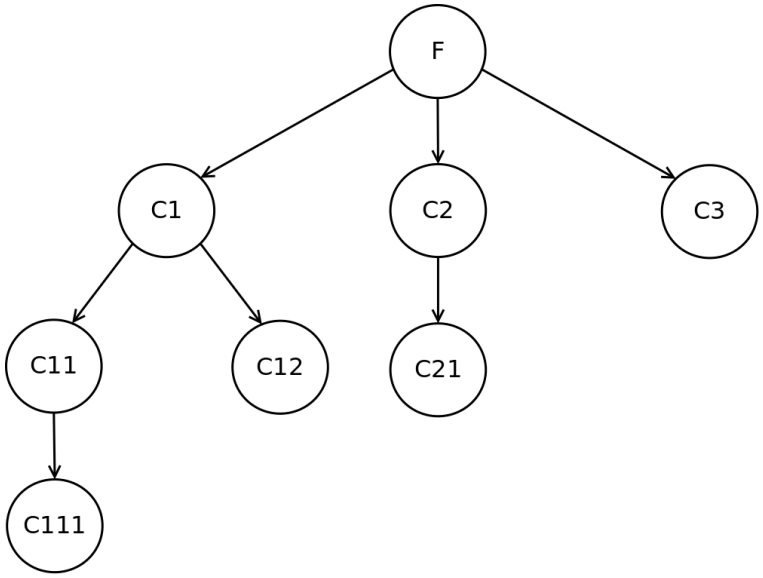


Figura 7.2: Grafo di precedenza

7.9 Condizioni di Bernstein

Le condizioni di Bernstein sono le condizioni necessarie affinché un algoritmo possa essere scritto in modo concorrente; dati due processi `Pi` e `Pj`, essi sono parallelizzabili se valgono le seguenti condizioni (`R(P)`: input del processo `P`; `W(P)`: output del processo `P`):

- `R(Pi)` $\cap$ `W(Pj)` = $\emptyset$
- `W(Pi)` $\cap$ `R(Pj)` = $\emptyset$
- `W(Pi)` $\cap$ `W(Pj)` = $\emptyset$

Capitolo 8

Operazioni sui processi

8.1 Identificazione

8.1.1 Process Identifier (PID)

Il PID è un intero non negativo (UNIX/linux), generato automaticamente dal SO, che identifica in modo univoco ciascun processo

8.1.1.1 PID riservati

I PID riservati dal SO UNIX/linux sono:

- PID = 0: per lo scheduler dei processi
- PID = 1: per il processo `init` (invocato al termine del bootstrap) che è l'antenato di tutti i processi

8.1.2 System call: `getpid() getppid()`

Le system call `getpid() getppid()` restituiscono rispettivamente il PID (intero non negativo) del processo corrente e del processo padre

```
#include <unistd.h>
pid_t getpid();
pid_t getppid();
```

8.2 Creazione

8.2.1 System call: `fork()`

La system call `fork()` genera un processo figlio identico al padre eccetto che per il PID; restituisce: in caso di successo: al processo padre il PID del figlio; al figlio il valore 0; in caso di errore restituisce -1 al processo padre

```
#include <unistd.h>
pid_t fork(void);
```

8.2.2 Risorse condivise

I processi padre e figlio condividono (in UNIX/linux):

- il codice sorgente
- tutti i descrittori dei file
- lo user ID e il group ID
- la root e la working directory
- le risorse del sistema

8.2.3 Risorse non condivise

I processi padre e figlio non condividono (in UNIX/linux):

- il valore ritornato dalla `fork`
- il PID
- lo spazio dati, la heap e lo stack

La `fork` implica la duplicazione delle risorse non condivise

8.3 Terminazione

Un processo può essere terminato in 8 modi:

- standard
  - `return` dalla funzione principale
  - `exit`
  - `_exit` o `_Exit`
  - `return` dal `main` dell'ultimo thread del processo
  - `pthread_exit` dall'ultimo thread del processo
- non standard
  - `abort`
  - ricevere un segnale di terminazione
  - cancellare l'ultimo thread del processo

Al termine di ogni processo il kernel invia al padre un segnale `SIGCHLD` (Signal Child)

8.4 Sincronizzazione

Il padre di un processo terminato può decidere di:

- ignorare l'evento (default)
- gestire la terminazione del figlio mediante: un gestore del segnale `SIGCHLD` o una system call

8.4.1 System call: `wait()`

Le system call `wait()` ricongiunge il padre a uno dei suoi figli operando nel seguente modo:

- restituisce lo stato di terminazione del figlio (immediatamente) se almeno uno dei figli è terminato (attraverso il parametro `statLoc`)
- blocca il processo padre se tutti i figli sono ancora in esecuzione (fino a che un figlio non termina)
- invia un messaggio di errore se il padre non ha figli

restituisce: il PID del figlio terminato

```
#include <sys/wait.h>
pid_t wait(int *statLoc);
```

8.4.1.1 Parametro `statLoc`

Il parametro `statLoc` è un puntatore ad un intero che specifica lo stato di uscita del processo figlio Lo stato di uscita del processo figlio può essere "catturato" dal padre attraverso le macro:

- `WIFEXITED(statLoc)`: ritorna un valore diverso da `0` se il figlio è terminato correttamente
- `WEXITSTATUS(statLoc)`: ritorna gli 8 bit meno significativi del valore di ritorno del figlio

8.4.2 System call: `waitpid()`

La system call `waitpid()` ricongiunge il padre a uno dei suoi figli (come la `wait()`); inoltre permette di:

- non fermare il padre se i figli sono tutti ancora in esecuzione
- attendere la terminazione di un figlio specifico tramite la variabile `pid`

```
#include <sys/wait.h>
pid_t waitpid(pid_t, int *statLoc, int options);
```

Il parametro `pid` può assumere i seguenti valori:

- `pid=-1`: il padre attende un figlio qualunque
- `pid>0`: il padre attende il figlio identificato dal `pid`
- `pid=0`: il padre attende qualunque figlio il cui group ID sia uguale al chiamante
- `pid<-1`: il padre attende qualunque figlio il cui group ID sia uguale a `|pid|`

8.4.3 Processo zombie

Un processo zombie è un processo terminato per il quale il padre non ha ancora eseguito una `wait()`; il processo viene rimosso solo dopo la `wait()` del padre; se il padre termina prima di eseguire la `wait()` il processo viene ereditato dal processo `init`

8.5 Controllo avanzato

8.5.1 System call: `exec`

La system call `exec` uccide il processo in corso e lo sostituisce con un nuovo programma avente lo stesso PID del precedente; ciò che rimaneva da eseguire nel processo padre non viene fatto
Esistono tipi diversi di `exec`:

- `execl[pe]` (lista): la funzione riceve come input una lista di parametri
- `execv[pe]` (vettore): la funzione riceve come input un vettore di parametri
- `exec[lv]p` (path): la funzione rintraccia i file del nuovo processo a partire dalla variabile d'ambiente PATH
- `exec[lv]e` (environment): la funzione riceve un vettore con le variabili di ambiente del processo da uccidere

Le system call `exec` ritornano: il valore -1 in caso di errore; non ritornano in caso di successo

```
#include <unistd.h>
int execl(char *path, char *arg0, ..., (char*)0);
int execlp(char *name, char *arg0, ..., (char*)0);
int execle(char *path, const char *arg0, ..., char *envp[]);
int execv(char *path, char *argv[]);
int execvp(char *name, char *arg[]);
int execve(char *path, char *arg[], char *envp[]);
```

8.5.1.1 Parametri

I parametri delle funzioni `exec` sono:

- `char *path`: percorso del file-system relativo al file eseguibile
- `char *name`: filename del file eseguibile (corrisponde ad `argv[0]` in C)
- `char *arg`: argomenti (file, stringhe, numeri) da passare all'eseguibile (corrispondono ad `argv[i]` in C)

8.5.2 System call: `system`

La system call `system` invoca il comando `string` all'interno di una shell; restituisce: il codice di terminazione del comando in caso di successo; -1 o 127 in caso di errore

```
#include <stdlib.h>
int system(const char *string);
```

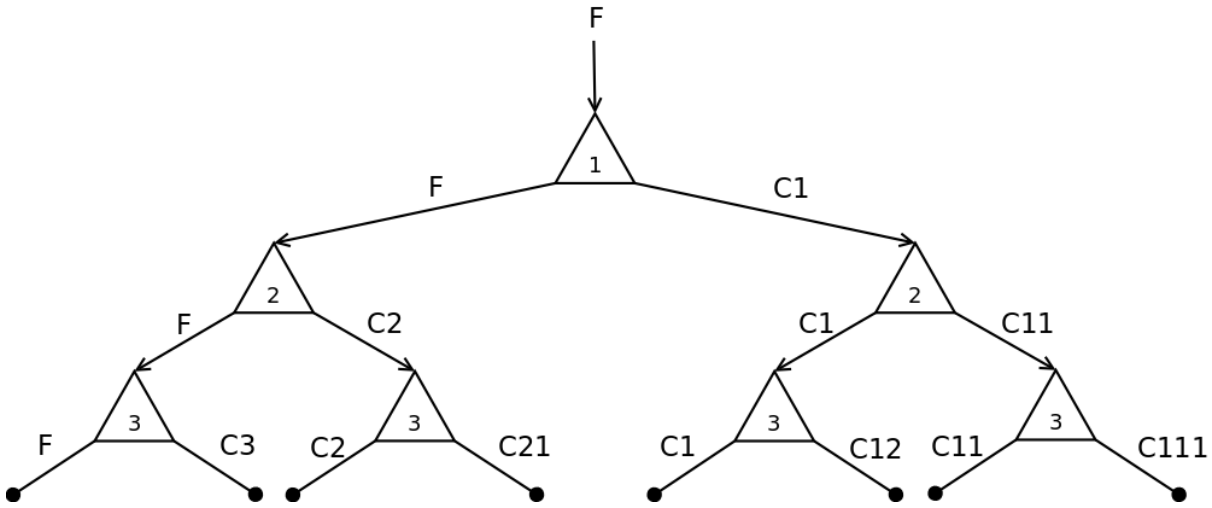
8.6 *Esercizi*

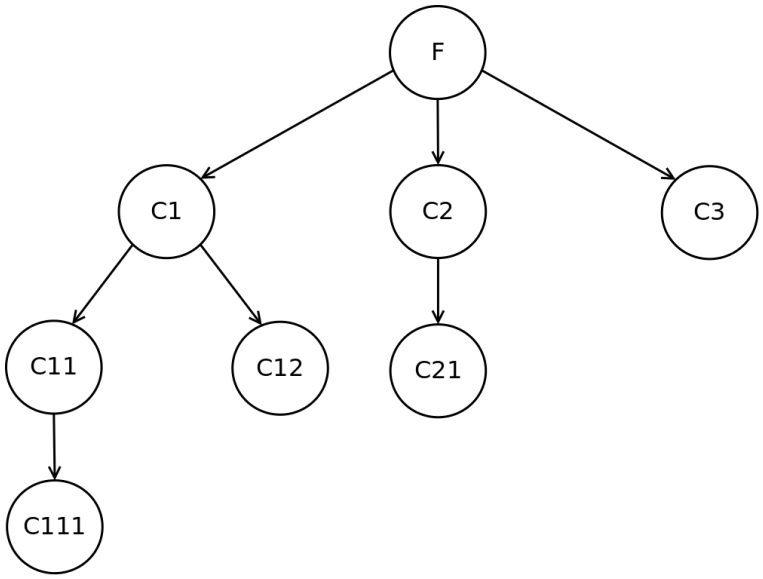
8.6.1 *Fork*

Esercizio 1. *Disegnare il control flow graph e l'albero di generazione dei processi relativo al seguente schema di programma:*

```
int main(){
    fork(); //fork 1
    fork(); //fork 2
    fork(); //fork 3
}
```

Le fork vengono effettuate da tutti i processi attivi del programma





8.6.2 *Programmi concorrenti*

Esercizio 2. *Scrivere un programma concorrente che dato un intero n: generi n processi figlio; visualizzi i PID dei figli*

1) fork.c

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

int main(int argc, char *argv[]){
    int i, n;
    n = atoi(argv[1]);

    fprintf(stdout, ‘‘Father PID=%d\n’’, getpid());
    for(i=0; i<n; i++)
        if(fork() == 0){
            fprintf(stdout, ‘‘Child %d:  PID=%d\n’’, i+1, getpid());
            break;
        }

    return 0;
}
```

2) bash

```
username@pcname~/Scrivania/prova$./exefile 3
Father PID=7841
Child 1:  PID=7842
Child 2:  PID=7843
Child 3:  PID=7844
```

Esercizio 3. *Scrivere un programma concorrente in grado di: generare un processo figlio; far terminare i due processi (padre e figlio) separatamente; visualizzare il PID del processo che termina e il PID del suo padre*

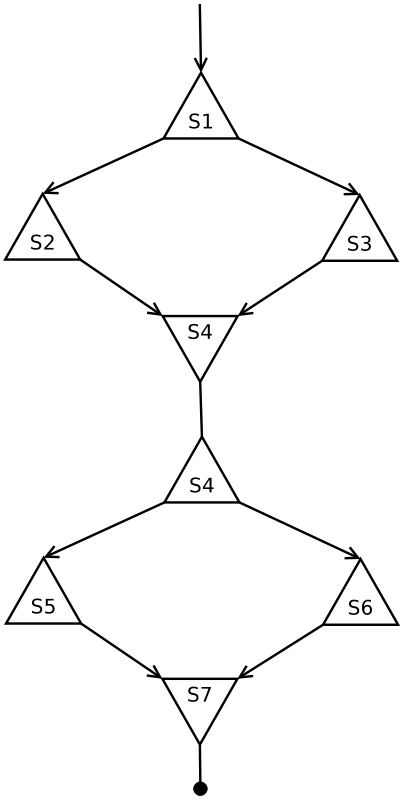
1) fork.c

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

int main(int argc, char *argv[]){
    int tC, tF;
    pid_t pid;
    tC = atoi(argv[1]);     // wait time child
    tF = atoi(argv[2]);     // wait time father

    fprintf(stdout, ``Main\n'');
    fprintf(stdout, ``PID=%d; Parent PID=%d\n'', getpid(), getppid());

    pid = fork();
    if(pid == 0){
        sleep(tC);
        fprintf(stdout, ``Child\n'');
        fprintf(stdout, ``PID=%d; Parent PID=%d; '', getpid(), getppid());
        fprintf(stdout, ``Returned Value=%d\n'', pid);
    }
    else{
        sleep(tF);
        fprintf(stdout, ``Father\n'');
        fprintf(stdout, ``PID=%d; Parent PID=%d; '', getpid(), getppid());
        fprintf(stdout, ``Returned Value=%d\n'', pid);
    }
}
```

2) bash

```
username@pcname~/Scrivania/prova$./exefile 5 2
Main
PID=5820; Parent PID=5153
Father
PID=5820; Parent PID=5153; Returned Value=5821
username@pcname~/Scrivania/prova$
Child
PID=5821; Parent PID=1930; Returned Value=0

username@pcname~/Scrivania/prova$./exefile 2 5
Main
PID=5950; Parent PID=5153
Child
PID=5951; Parent PID=5950; Returned Value=0
Father
PID=5950; Parent PID=5153; Returned Value=5951
```

Esercizio 4. *Scrivere un programma C concorrente in grado di costruire l'albero dei processi mostrato in figura*



1) conc.c

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

int main(){
        pid_t pid, cpid;
        int status;

        printf(``S1\n'');
        pid = fork();
        if(pid == 0){
                sleep(1);
                printf(``S2-child\n'');
                exit(0);
        }
        else{
                sleep(3);
                printf(``S2-father\n'');
                wait((int*) 0);
        }
```

```
                printf(``S4\n'');
        pid = fork();
        if(pid == 0){
                int status1 = 5;
                sleep(4);
                printf(``S5-child\n'');
                printf(``CHILD: fpid:  %d; mypid:  %d; mystatus:  %d\n'',
                        getppid(), pid, status1);
                status = -34;
                exit(status1);
        }
        else{
                sleep(1);
                printf(``S5-father\n'');
                cpid = wait(&status);
                printf(``FATHER: mypid=%d; cpid:  %d; status:  %x; '',
                        getpid(), cpid, status);
                printf(``WIFEXITED: %d; WEXITSTATUS: %d\n'',
                        WIFEXITED(status), WEXITSTATUS(status));
        }
        sleep(4);
        printf(``S7\n'');
        return 0;
}
```

2) bash

```
username@pcname~/Scrivania/prova$./exe
S1
S2-child
S2-father
S4
S5-father
S5-child
CHILD: fpid:  6085; mypid:  0; mystatus:  5
FATHER: mypid=6085; cpid:  6089; status:  500; WIFEXITED: 1; WEXITSTATUS: 5
S7
```

Capitolo 9

Segnali

9.1 Interrupt

Un interrupt è l'interruzione di un processo corrente dovuto al verificarsi di un evento straordinario

9.2 Segnale

Un segnale è un interrupt software, cioè un evento di sistema inviato in modo asincrono da processo ad un altro processo

9.2.1 Segnali principali

I segnali principali dei SO UNIX/linux sono:

- `SIGALRM`: genera un orologio con un allarme
- `SIGKILL`: uccide un processo
- `SIGCHLD`: segnale restituito al padre da un processo figlio quando si ferma o termina
- `SIGUSR1` e `SIGUSR2`: segnali general purpose riservati all'utente

9.2.2 Ciclo di vita dei segnali

Il ciclo di vita di un segnale è composto da 3 fasi:

- generazione del segnale: un segnale viene generato da un evento scatenato da un processo
- consegna del segnale: il SO consegna il segnale al processo destinatario; un segnale non ancora consegnato si dice pendente
- gestione del segnale: il processo destinatario può chiedere al kernel di:
  - utilizzare il comportamento di default del segnale
  - ignorare il segnale
  - utilizzare il comportamento specificato in una funzione del processo destinatario chiamata signal handler

9.3 Sistem call per la gestione dei segnali

9.3.1 Istanziazione di un gestore di segnali: `signal()`

La system call `signal()` consente di instanziare un gestore di segnali attraverso la funzione avente indirizzo `func` (signal handler)(il parametro `int` della `func` indica il codice del segnale da gestire) per gestire il segnale avente codice `sig`; restituisce: il puntatore al signal handler precedente in caso di successo; il segnale `SIG_ERR` in caso di errore

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

Il signal handler `func` può assumere i seguenti valori:

- `SIG_DFL` (default): termina il processo (in genere)
- `SIG_IGN` (ignore): ignora il segnale (implica avere un comportamento indefinito); i segnali `SIGKILL` e `SIGSTOP` non possono essere ignorati
- `signalHandlerFunctionName`: esegue la funzione utente di gestione del segnale

9.3.2 Invio di un segnale: `kill()`

La system call `kill()` invia un segnale di codice `sig` ad un processo (o gruppo di processi) avente PID `pid`; per inviare un segnale ad un processo occorrono opportuni privilegi; restituisce: 0 in caso di successo; -1 in caso di errore

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

Il `pid` può essere:

- `>0`: invia il segnale al processo di PID uguale a `pid`
- `==0`: invia il segnale a tutti i processi del gruppo a cui appartiene la `kill()`
- `<0`: invia il segnale a tutti i processi avente PID uguale a `|pid|`
- `==1`: invia un segnale a tutti i processi del sistema

9.3.3 Autoinvio di un segnale: `raise()`

La system call `raise()` invia un segnale di codice `sig` al processo chiamante (equivale a `kill(getpid(), sig)`); restituisce: 0 in caso di successo; -1 in caso di errore

```
#include <signal.h>
int raise(int sig);
```

9.3.4 Sospensione di un segnale: `pause()`

La system call `pause()` sospende il processo chiamante sino all'arrivo di un segnale; restituisce: il valore -1 quando viene eseguito e terminato un gestore di segnali

```
#include <signal.h>
int pause(void);
```

9.3.5 Invio di un allarme: `alarm()`

La system call `alarm()` attiva un count-down di `seconds` secondi al cui termine genera un segnale `SIGALRM`; ogni attivazione di `alarm()` resetta il timer (se `seconds==0` si disattivano gli allarmi); restituisce: il numero di secondi rimasti prima dell'invio del segnale da parte chiamate precedenti (se presenti); 0 se non ci sono state chiamate precedenti

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

9.4 Limiti dei segnali

I limiti dei segnali sono:

- memoria per i segnali pendenti è limitata
- richiedono funzioni rientranti
- producono race conditions