



POLITECNICO DI TORINO

# Sistemi Operativi

Appunti del corso da 6 CFU del Prof. Stefano Quer

A.A. 2019/20

Autore: Marco Smorti

Data: Gennaio 2020

Gli argomenti capitati nelle domande di teoria dell'esame in a.a. precedenti sono contrassegnati con il simbolo ☺

## INDICE

1. Introduzione ai sistemi operativi (parte A) .....	3
2. Introduzione ai sistemi operativi (parte B) .....	5
3. Comandi UNIX & Linux (parte A) .....	5
4. Strumenti per la programmazione C .....	7
5. I file in ambiente Linux .....	8
6. I direttori in ambiente Linux .....	10
7. Introduzione ai processi .....	17
8. Comandi di Shell per la gestione dei processi .....	23
9. Processi: Aspetti teorici .....	24
10. Controllo avanzato .....	26
11. Segnali .....	31
12. Comunicazione tra processi .....	39
13. Pipe e ridirezione in UNIX/Linux .....	43
14. Espressioni regolari e comando find .....	44
15. I filtri .....	48
16. I thread .....	51
17. La libreria Pthread .....	54
18. Concorrenza: aspetti teorici .....	62
19. Comandi UNIX & Linux (Parte B) .....	66
20. Le shell .....	67
21. Gli script di shell .....	70
22. Le sezioni critiche .....	82
23. Soluzioni software .....	84
24. Soluzioni hardware .....	87
25. I semafori .....	91
26. Problemi di sincronizzazione tipici .....	102
27. Lo scheduling .....	109
28. Definizione del problema e modellizzazione .....	115
29. Tecniche di prevenzione .....	119
30. Tecniche per evitare uno stallo .....	121

# 1. Introduzione ai Sistemi Operativi (Parte A)

## Moduli e servizi tipici di un SO

1. **Interprete dei comandi**
2. **Gestione dei processi**
3. Gestione della memoria principale
4. Gestione della memoria secondaria
5. Gestione dei dispositivi di I/O
6. **Gestione file e file system**
7. Implementazione dei meccanismi di protezione
8. Gestione delle reti e sistemi distribuiti

In grassetto i moduli analizzati nel corso.

### Interprete dei comandi

L'utente e il SO comunicano attraverso una interfaccia che può essere testuale oppure grafica. Tale interfaccia viene detta **shell**. Il SO deve permettere ad un utente di gestire i processi, la memoria (principale e secondaria), instaurare politiche di protezione e gestire la rete e le connessioni esterne.

### Gestione dei processi

Un processo (entità attiva) è un programma (entità passiva) **in esecuzione**. Per essere eseguito richiede delle risorse e il SO deve essere in grado di creare, sospendere e cancellare un processo, oltre a fornire meccanismi di comunicazione e sincronizzazione tra processi.

### Gestione file e file-system

Le informazioni su memoria di massa sono organizzate in file-system, a sua volta suddivisi in direttori e contenenti dei file. Il SO deve essere in grado di creare, leggere, scrivere, cancellare direttori e file ed instaurare opportuni meccanismi di protezione di accesso ed ottimizzare le operazioni R/W.

## Terminologia

- **Kernel:** parte centrale di un SO. Gestisce le risorse, in particolare memoria e processi. E' un unico programma in esecuzione per tutto il tempo. Vi sono diversi tipi di kernel: Kernel a livelli, Micro-kernel e Kernel monolitici.
- **Bootstrap:** programma di inizializzazione. Carica il kernel in memoria centrale all'accensione e lo esegue permettendo l'inizializzazione di tutto l'intero SO.
- **Login:** seriously man?
- **Shell:** Interfaccia UNIX che **NON** fa parte del SO. Legge i comandi utente e li esegue.
- **File-system:** struttura gerarchica di direttori e file.
- **File-name:** nome del file. Gli unici caratteri che non possono essere inseriti per un nome sono lo slash "/" e il carattere "null"
- **Path-name:** sequenza di nomi separati da "/". Un solo punto ".." indica il directory corrente, due punti "../" il directory padrone. I path-name possono essere **assoluti o relativi**.
- **Home directory:** directory a cui si accede dopo il login. Contiene tutto il materiale dell'utente che ha fatto il login (si individua con la tilde).
- **Root directory:** directory principale e radice di tutti i direttori ed è il punto di origine per interpretare i path assoluto.
- **Working directory:** punto di origine per interpretare i path relativi ed è inizialmente pari alla home directory. Ci si riferisce automaticamente nel caso in cui non si specifichi un path.
- **Programma:** file eseguibile che risiede nel disco, deta **entità passiva**.

- **Programma sequenziale:** operazioni eseguite in sequenza, una nuova azione inizia al termine della precedente (fetch-decode-execute).
- **Programma concorrente o parallelo:** diverse valutazioni possono avvenire in parallelo e una operazione può essere eseguita senza attendere la fine della precedente.
- **Processo:** programma in esecuzione e **entità attiva**. Possiede un **identificatore univoco (PID)**.
- **Thread:** un processo **raggruppa** le risorse e quest'ultimo può avere all'interno uno o più flussi di controllo in esecuzione. Ognuno dei flussi è un thread ed ogni thread ha un identificatore locale al processo.
- **Pipe:** flusso dati tra due processi. Nel caso più semplice è un canale di comunicazione **half-duplex**.
- **Deadlock:** un insieme di entità attendono il verificarsi di un evento che può essere causato solo da un'altra entità dell'insieme (stallo potenziale e stallo conclamato).
- **Livelock:** simile al deadlock ma le entità non sono effettivamente bloccate. Non fanno alcun progresso. Due unità effettuano il **polling** per verificare lo stato dell'altro e non fanno progressi (visto che stanno effettuando polling non sono in deadlock).
- **Starvation:** a una entità viene ripetutamente negato l'accesso a una risorsa necessaria al suo progresso.  
Starvation **non** implica deadlock (se una unità è in starvation, le altre possono proseguire)  
Deadlock **implica** starvation (nessuna entità procede quindi sono tutte in starvation)

## System call ☺

Le system call forniscono l'interfaccia ai servizi forniti dal SO. Sono spesso implementate in **assembler** e vi si accede tramite le **API** (Application Program Interface). Le System call sono limitate (ne esistono poche centinaia a seconda del SO).

## Differenza System-call e funzione libreria ☺

Entrambe forniscono servizi all'utente, ma:

- Per ogni system call esistono una o più funzioni di alto livello con lo stesso nome.
- Le funzioni possono essere sostituite o modificate, mentre le system call no.
- Le system call forniscono funzionalità di base, mentre le funzioni di libreria sono più elaborate.
- Le system call richiedono il passaggio dalla modalità utente alla modalità superuser. Questo perché il SO si protegge lavorando in **dual-mode**. Gli utenti lavorano in modalità utente (bit di modo=1). Ogni system call sposta il bit a 0. Le chiamate vengono effettuate tramite una interruzione software o **trap**.

Esempio: la funzione libreria **printf** utilizza la system call **write**.

## 2. Introduzione ai Sistemi Operativi (Parte B)

Questo pacchetto di slide non contiene alcuna informazione utile ai fini dell'esame ma soltanto dati storici di commercio dei SO.

## 3. Comandi UNIX & Linux (parte A)

### Path

Per indicare un file all'interno del file system si deve specificare il relativo percorso (path), che può essere:

- **Assoluto** se si riferisce alla radice del sistema (Es. /dir1/dir2/file)
- **Relativo** se si riferisce alla working directory corrente (Es. subdir1/subdir2/file)

### Comandi Linux utili

**man** – manuale della funzione

**ls** – elenca contenuto di un direttorio -> -a (tutto, anche padre) -l (long list con dettagli per file) -R (mostra ricorsivamente il contenuto dell'albero)

**tab** – completa la parola

**cp** – copia file e direttori

**cd** – sposta la posizione della directory

**rm** – cancella file

**rmdir** – rimuove direttorio ma solo se la directory è vuoto

**rm -r** – Rimuove tutto ricorsivamente

**mv** – Rinomina i file

**cd ..** – Sale al Padre

**chmod** – Modifica i permessi dei file/directory (con -R posso dare diritti ricorsivamente a una intera directory)

### Permessi ☺

I permessi di base sono 3:

- **r** read
- **w** write
- **x** execute

Questi permessi possono essere forniti a 3 diversi tipi di utenti:

- **u** user (proprietario)
- **g** group
- **o** others (altri utenti)

I vari permessi per i tre tipi di utente sono definiti mediante tre cifre in base otto:

- **rwx rwx rwx** – 777
- **rw- rw- rw-** 666
- **rwx -x ---** 710

oppure tramite una lettera riferita al tipo di utente seguita da un simbolo (+, -, =) e dal carattere che si vuole attribuire (r,w,x). Si ricorda che la funzione **rm** funziona soltanto se il direttorio che si vuole rimuovere è vuoto, altrimenti occorre utilizzare il comando **rm -rf**.

I permessi assumono significato diverso se si tratta di file o directory. Nel caso di un file i permessi hanno un significato prevedibile. Nel caso dei direttori, invece:

- **r** permette di **elencare i file**
- **w** permette la creazione e/o cancellazione di file in direttori
- **x** consente l'attraversamento della directory

Ad esempio, il comando **cd dir** fallisce se dir non ha diritti di esecuzione.

## Visualizzazione di un testo

Tramite il comando

**cat file1 file2**

è possibile visualizzare e concatenare due o più file.

I comandi:

**head [opzioni] file**  
**tail [opzioni] file**

consentono di visualizzare le prime/ultime **n** righe di un file (di default n=10).

Esistono ulteriori comandi:

**more [opzioni] file**  
**less [opzioni] file**

Il primo permette di visualizzare il file, mentre il secondo funziona come il precedente ma permette l'utilizzo delle frecce per muoversi nel testo già visualizzato.

Inoltre,

**diff [opzioni] entry1 entry2**

permette di controllare le differenze tra due entry (file o directory) ed elenca il numero di righe aggiunte, cancellate o modificate.

Infine,

**wc [opzioni] [file]**

consente di conteggiare il numero di linee, parole e byte in un file.

## Hard e Soft Link ☺

In UNIX esistono due tipi di link:

- **Soft link:** particolare tipo di file che contiene un **path** a un altro oggetto (file o directory) e permette riferimenti tra file-system diversi. Se si rimuove il file, il soft link rimane pendente (come un puntatore a NULL).
- **Hard link:** associazione tra il nome di un oggetto e il suo contenuto. Non è possibile creare hard link su file system diversi o verso un directory. Il file viene rimosso soltanto quando viene rimosso l'ultimo dei suoi hard link.

Il comando

**ln [opzioni] source [destination]**

permette di creare un **link** che di default è un **hard link**. L'indicazione della destinazione è opzionale, poiché se non viene specificata, il link viene creato con lo stesso nome del source e nel directory corrente. Per creare un soft link è necessario specificarlo con l'opezione **-s**.

Si ricorda che il comando **rm** rimuove un file solo se il numero degli hard link è pari a 0, mentre il comando **mv** equivale a eseguire prima il comando **ln** e poi il comando **rm**.

## 4. Strumenti per la programmazione C

Il comando per compilare è

**gcc <opzioni> <argomenti>**

Le varianti più utilizzate sono:

**gcc -c file.c**                        compilazione di singoli file

**gcc -o myexe fil1.o**                e poi link dei file oggetto in un unico eseguibile

Oppure è possibile unire i comandi scrivendo:

**gcc -o myexe file1.c file2.c main.c**

Le opzioni più comuni sono:

**-c file** //esegue compilazione e non linker

**-o file** //specificia il nome di output; in genere indica il nome dell'eseguibile finale (linkando)

**-g** //indica di non ottimizzare il codice

**-Wall** //stampa warning per tutti i possibili errori nel codice

**-lm** //specificia l'utilizzo della libreria matematica

con \ passo alla riga successiva

### Makefile

Automatizza le operazioni di compilazione. Si procede in due fasi:

- Si scrive un file Makefile
- Si interpreta tale file con l'utility make

L'opzione **make** esegue automaticamente il Makefile. Tale comando può essere eseguito con diverse opzioni (tra cui **-d** che stampa informazioni di debug).

Si utilizza come segue:

**make -f nomefile** //specifico il nome del Makefile.

Il file makefile contiene una o più sezioni che hanno uno specifico formato:

**target: dependency**  
**<tab>command**

Se ci sono più regole in genere viene eseguita la prima. Si può eseguire una in particolare indicando il nome del target. 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.

Per eseguire una regola specifica si utilizza:

```
make -f <myMakefile> <nomeTarget>
```

#### ESEMPI

**target:**

```
<tab>gcc -Wall -o myExe main.c -lm
```

Unica regola all'interno del makefile di nome **target** e il target non ha dipendenze. Corrisponde a eseguire tale comando su riga di comando.

**Project1:**

```
<tab>gcc -Wall -o project1 myFile1.c
```

**Project2:**

```
<tab>gcc -Wall -o project2 myFile2.c
```

In questo caso ci sono più target e bisogna scegliere quale eseguire. Il default consiste nell'eseguire il primo target.

```
make <nomeFile> //esegue Project1  
make -f project2 //esegue project2
```

## 5. I file in ambiente Linux ☺

I file memorizzano informazioni a lungo termine. Dal punto di vista logico, un file può essere visto come **insieme di informazioni correlate**, memorizzate su un dispositivo utilizzando un **sistema di codifica**.

### Codifica ASCII

- **ASCII, American Standard Code for Information Interchange**

Tale codifica era basata inizialmente sull'alfabeto inglese e codifica 128 caratteri in 7-bit (binario) di cui 32 non stampabili (e 96 stampabili).

- **Extended ASCII (o high ASCII)**

Estensione dell'ASCII a 8-bit e 255 caratteri.

### Codifica Unicode

Standard industriale che include le codifiche per ogni sistema di scrittura esistente. Contiene più di 110.000 caratteri. Si può interpretare in diversi modi:

- UCS (Universal Character Set)
- **UTF** (Unicode Transformation Format) di cui ne esistono più versioni:
  - UTF-8 con codifica a gruppi di **8-bit** (che corrisponde all'ASCII a 8 bit)
  - UTF-16 con codifica a gruppi di **16 bit**
  - UTF-32 con codifica a **32 bit** (di lunghezza fissa)

I file si distinguono in: **file di testo** e **file binari**. Il kernel UNIX non fa distinzione tra file di testo e file binari.

- **File di testo(o ASCII)**: consiste in dati codificati in ASCII in sequenza di bit. I file di testo sono generalmente **line-oriented**, ovvero tramite la **newline** ci si sposta sulla riga successiva. La newline viene codificata con 1 carattere su Linux e con 2 caratteri su Windows.
- **File binari**: sequenza di bit non "byte-oriented". La più piccola unità di R/W è il **bit**. Include ogni possibile sequenza di 8 bit e non necessariamente questi sono caratteri stampabili. Si utilizzano i file binari per **compattezza** (un file di testo codifica ad esempio 123 in 1+2+3 e poi in binario quindi 8+8+8 bit, mentre un file binario codifica 123 come un unico numero a 8 bit).

**Serializzazione:** processo di traduzione di una struttura in un formato memorizzabile. Tramite la serializzazione è possibile memorizzare o trasmettere una struttura come un'unica entità. Ovviamente bisogna mettersi d'accordo tra chi scrive e chi legge.

Si possono fare le stesse operazioni senza utilizzare la libreria C, ma con la POSIX. Quando si fa una read e una write si chiamano delle system call al kernel che vengono definite "**unbuffered I/O**". Le funzioni sono: **open, read, write, close**. Nel kernel UNIX il file descriptor è un intero non negativo. Con una open, ad esempio, riceviamo un intero e non più un file pointer.

**Lo standard I/O è "fully buffered"** cioè l'operazione di I/O avviene solo quando il buffer è pieno.  
L'operazione di "flush" indica la scrittura del buffer su I/O.

```
#include <stdio.h>
void setbuf (FILE *fp, char *buf);
int fflush (FILE *fp);
```

Nei processi concorrenti si utilizza

```
setbuf (stdout, 0);
fflush (stdout);
```

## POSIX Standard Library

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

**open, read, write, lseek, close**

Tale tipologia di accesso fa parte di POSIX e si indica con il termine "**unbuffered I/O**" poiché ciascuna operazione di read o write corrisponde a una system call al kernel.

### System call **open()** ☺

Nel kernel UNIX un "file descriptor" è un intero non negativo (e non un FILE\*) dove per convenzione:

- Lo standard input corrisponde a 0
- Lo standard output corrisponde a 1
- Lo standard error corrisponde a 2

La funzione open apre un file definendone i permessi. Si ha un valore di ritorno che è il descrittore del file in caso di successo, oppure -1.

```
int open (const char *path, int flags);
```

Il parametro **flags** ha molteplici opzioni, alcune obbligatorie come le seguenti:

- **O\_RDONLY**
- **O\_WRONLY**
- **O\_RDWR**

Altre invece sono opzionali:

- **O\_CREAT** crea il file se non esiste
- **O\_TRUNC** rimuove il contenuto del file
- **O\_APPEND** appende al file

#### System call **read()** ☺

```
int read (int fd, void *buf, size_t nbytes);
```

La funzione legge dal file **fd** un numero di byte uguale a **nbytes** memorizzandoli in **buf**. Il valore di ritorno è **il numero di byte letti in caso di successo**. Il valore ritornato è inferiore a nbytes nel caso in cui la fine viene raggiunta prima di leggere gli nbytes oppure se **la pipe da cui si sta leggendo non contiene nbytes bytes**.

#### System call **write()** ☺

```
int write (int fd, void *buf, size_t nbytes);
```

La funzione scrive **nbytes** byte di **buf** nel file descrittore di **fd**. Restituisce il numero di byte scritti in caso di successo (di norma **nbytes**) oppure **-1** in caso di errore. Si ricorda che la write scrive sui buffer di sistema e non sul disco.

#### System call **close()** ☺

```
int close (int fd);
```

Chiude il file aperto (i file si chiudono automaticamente quando un processo termina). Restituisce 0 in caso di successo oppure -1 altrimenti.

## 6. I direttori in ambiente Linux ☺

Nessun sistema di memorizzazione contiene un unico file, ma essi vengono organizzati in **direttori**. Ogni directory viene visto come un **nodo** (di un albero) che contiene informazioni sugli elementi in esso contenuti.

Direttori e file risiedono nella memoria di massa.

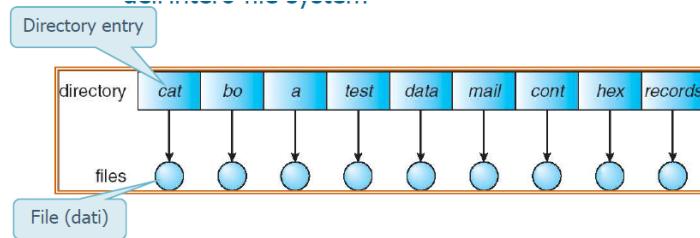
Su un directory è possibile effettuare operazioni di: *creazione, cancellazione, elenco del contenuto, ridenominazione, visita e ricerca*.

La struttura di un directory dipende da:

- **Efficiency (efficienza)** cioè la velocità nel manipolare il file system (es. localizzare un file)
- **Naming (convenienza)** cioè la semplicità per un utente di identificare i propri file ed evitare che lo stesso nome attribuito a più file crei problemi
- **Grouping (organizzazione)** raggruppare le informazioni in base alle relative caratteristiche (programmi di editing, compilatori, giochi, etc..)

## Direttorio a livello singolo ☺

La struttura più semplice è a **livello singolo**, ovvero tutti i file del file system sono contenuti all'interno dello stesso directory. Ciò che li distingue è il nome ed ogni nome deve essere **univoco**.



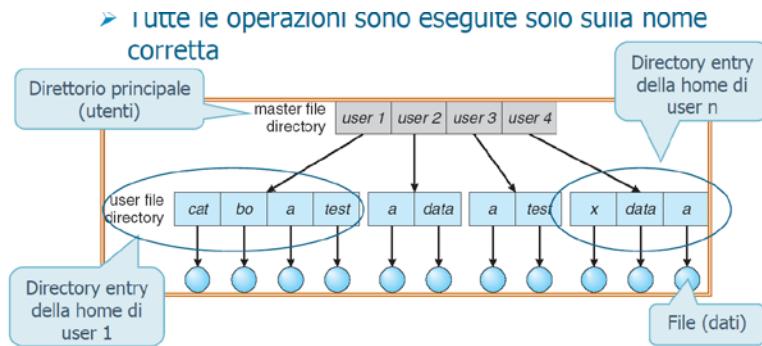
Per ciascun file si evidenziano la **directory entry**, ovvero il nome ed eventualmente altre informazioni del file ed il **dato** che viene identificato tramite puntatore ed è separato dunque dalla directory entry (come in foto).

Prestazioni del directory a un livello:

- *Efficiency*: struttura facilmente comprensibile e gestibile, gestione del file system semplice ed efficiente
- *Naming*: i file devono avere nomi univoci e presenta limiti evidenti all'aumentare del numero di file memorizzati
- *Grouping*: la gestione dei file di un utente singolo è complessa, mentre la gestione di utenti multipli è praticamente impossibile

## Direttori a due livelli ☺

Nei direttori a due livelli i file sono contenuti in un albero a due livelli, dove ogni **utente** può avere il **proprio directory** ed ogni user ha la sua **home directory**. Tutte le operazioni vengono eseguite solo sulla **home corrente**.

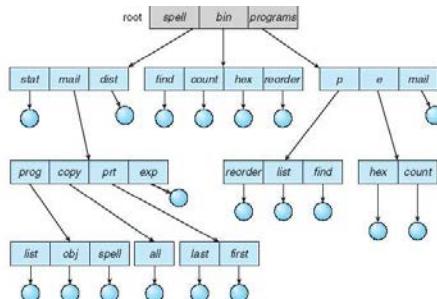


Prestazioni del directory a due livelli:

- *Efficiency*: visione del file-system "user oriented", ricerche semplificate e efficienti agendo su utenti singoli
- *Naming*: È possibile avere file con lo stesso nome purchè appartenenti a utenti diversi. Occorre specificare un path-name per ogni file
- *Grouping*: semplificato tra utenti diversi ma complesso per ciascun utente singolo

## Direttori ad albero ☺

Il direttorio ad albero è una **generalizzazione** dei precedenti. Tutti i file sono contenuti in un albero, ed ogni nodo/vertice può contenere come entry un altro nodo/vertice dell'albero.

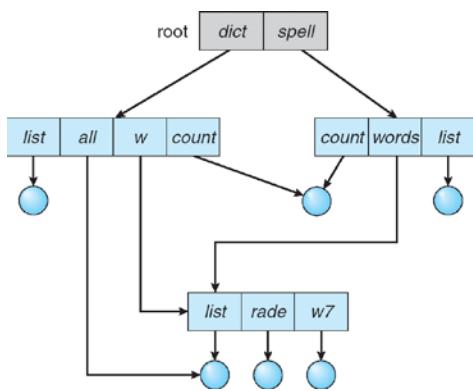


Ogni utente può gestire tanto file quanto direttori e sotto-direttori, perciò nascono da qui i concetti di **working directory, cambio directory, path assoluto e relativo**.

Prestazioni del direttorio ad albero:

- *Efficiency*: ricerche vincolate alla struttura ad albero e quindi alla sua profondità e ampiezza
- *Naming*: permesso in maniera estesa
- *Grouping*: permesso in maniera estesa

## Direttori a grafo aciclico ☺



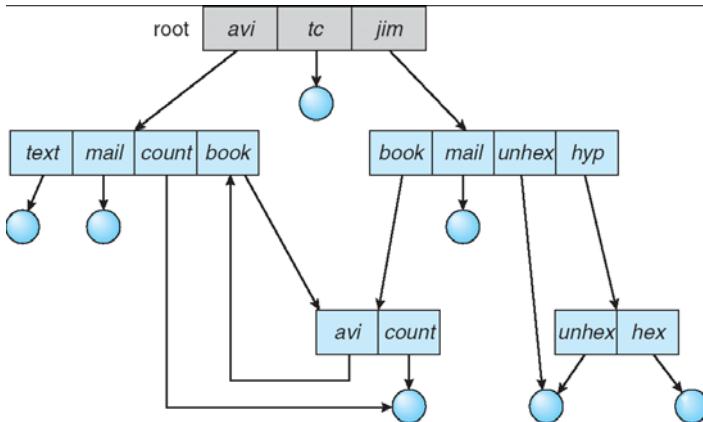
I file system ad albero **non permettono la condivisione di informazioni**. Può essere utile individuare lo stesso oggetto con nomi diversi sia da parte di utenti diversi che dallo stesso utente ma con path diversi. La copia delle informazioni non risolve tale problema a causa dell'aumento dello spazio occupato dal file system e la coerenza delle informazioni presenti nelle varie copie. I grafi aciclici hanno questa funzione.

La presenza di link aumenta la difficoltà di gestione dei file system, perciò occorre distinguere gli oggetti nativi dai relativi collegamenti in fase di creazione, manipolazione e cancellazione.

- **Creazione**: nei sistemi UNIX-like la strategia standard è quella di creare **collegamenti o link** (riferimenti a un'altra entry pre-esistente).
- **Visita e ricerca**: se la entry è un link occorre accorgersene ed effettuare un indirizzamento indirizzo (cioè risolvere il collegamento) ed utilizzarlo per raggiungere l'entry originaria. Tramite link ogni entry del file system può essere raggiungibile con più path assoluti (e con nomi diversi)
- **Cancellazione**: occorre stabilire come gestire il link e come gestire l'oggetto riferito. La cancellazione del link viene in genere effettuata in maniera immediata e non influisce sull'oggetto originale. Per la cancellazione dell'oggetto bisogna distinguere vari casi.
  - **Soft-link** porta alla cancellazione immediata dei dati. I link rimangono pendenti e appena si cercherà di utilizzare il link ci si accorgerà che il file riferito è scomparso.
  - **Hard-link** la cancellazione dei dati avviene soltanto alla cancellazione dell'**ultimo link**. Per evitare link pendenti occorre tenerne traccia (e gestire la presenza di oggetti e link multipli), ma ciò potrebbe risultare inefficiente perciò conviene memorizzare soltanto il contatore (cioè il numero di hard-link). Tali informazioni si possono vedere col comando **ls -i**

La creazione di un link a un directory potrebbe causare la nascita di un ciclo nel file system. La gestione di un grafo ciclico richiede maggiore complessità (una ricerca dovrebbe assicurarsi di non finire in un loop) e tra le possibili strategie la più semplice è quella di **non permettere** la creazione di link a direttori.

### Direttori a grafo ciclico ☺



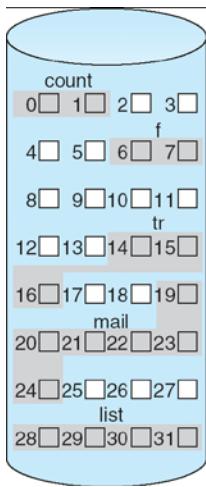
Nel caso in cui si permetta la creazione di cicli bisogna gestire opportunamente i cicli esistenti in tutte le fasi. La gestione può essere effettuata con diversi approcci che dovranno tenere conto di diverse problematiche: un elemento potrebbe **auto-referenziarsi** e **non essere mai cancellato** e/o rilevato, oppure la gestione più semplice consiste nel **non visitare mai i link**.

### Allocazione ☺

Per allocazione si intendono le tecniche di utilizzo dei blocchi dei dischi per la memorizzazione di file. Le metodologie principali sono:

- **Contigua** (contiguous): ogni file occupa un insieme contiguo di blocchi.
- **Concatenata** (linked): ogni file può essere allocato gestendo una lista concatenata di blocchi
- **Indicizzata** (indexed): ogni file ha la sua tabella, ovvero un vettore di indirizzi dei blocchi in cui il file è contenuto

### Allocazione contigua ☺



Per ciascun file il directory specifica l'indirizzo del primo blocco e la lunghezza del file.

Il file occupa i blocchi da **b** fino a **b+n-1**

Ogni file presenta frammentazione interna (l'ultimo blocco è parzialmente occupato).

I vantaggi risiedono nella strategia di allocazione molto semplice che permette di memorizzare poche informazioni per ogni file.

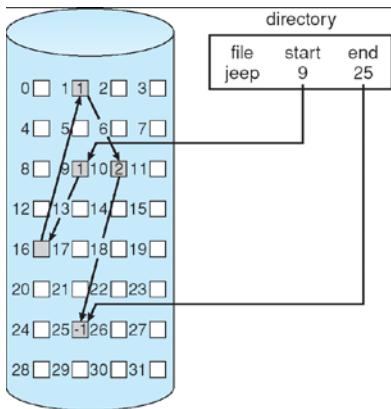
Gli accessi sequenziali sono immediati e ogni blocco si trova dopo il precedente e prima del successivo.

Permette inoltre accessi diretti semplici.

Gli **svantaggi** sono molteplici: bisogna decidere la politica di allocazione (cioè ricercare uno spazio libero di dimensione

sufficiente), nessun algoritmo di allocazione è privo di difetti e quindi le tecniche sprecano spazio, ovvero si ha la **frammentazione esterna** (insieme di blocchi non utilizzati) e sono presenti problemi di allocazione dinamica, poiché i file non possono crescere liberamente in quanto lo spazio disponibile è limitato dal file successivo.

## Allocazione concatenata ☺



Il direttorio contiene un puntatore al primo e uno all'ultimo blocco del file. Ogni blocco contiene un puntatore al blocco successivo. I blocchi di ciascun file sono sparsi per l'intero disco.

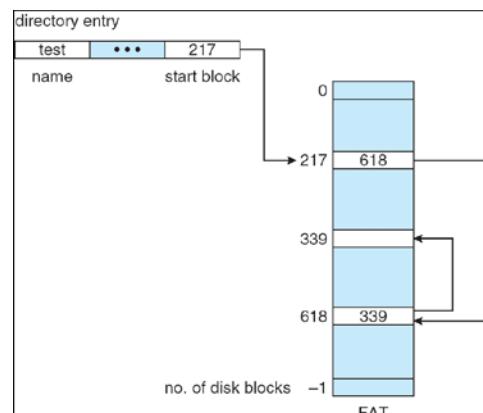
Questo tipo di allocazione risolve i problemi dell'allocazione contigua poiché permette l'allocazione dinamica dei file, elimina la frammentazione esterna ed evita l'utilizzo di algoritmi di allocazione complessi.

**Svantaggi:** ogni lettura implica un accesso sequenziale ai blocchi, perciò risulta efficiente solo per accessi sequenziali. Un accesso diretto richiede la lettura di una catena di puntatori sino a raggiungere l'indirizzo desiderato ed ogni accesso a un puntatore implica una operazione di lettura dell'intero blocco. La memorizzazione dei puntatori richiede inoltre spazio ed è critica dal punto di vista dell'affidabilità e rende lo spazio utile minore.

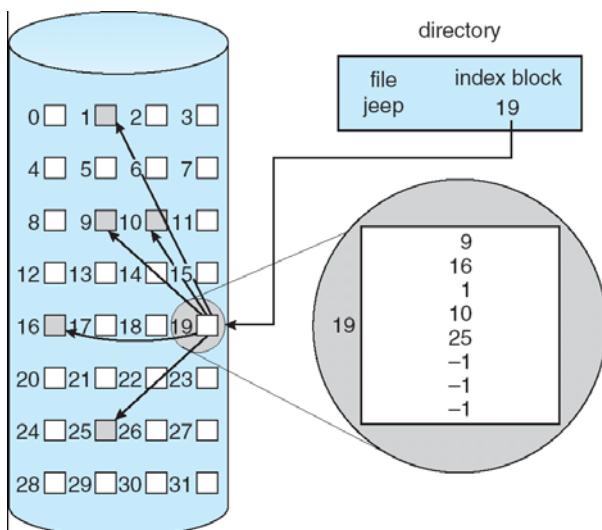
L'allocazione utilizzata da MS-DOS è basata su **FAT (File Allocation Table)** ed è una variante del metodo di allocazione concatenato.

È presente una tabella con un elemento per ciascun blocco presente sul disco. La sequenza dei blocchi appartenenti a un file è individuata a partire dalla directory mediante l'elemento di partenza del file nella FAT e dalla sequenza di puntatori presenti direttamente nella FAT (e non più nei blocchi).

I riferimenti non sono più memorizzati nei blocchi su disco ma direttamente negli elementi della FAT. La lettura di ogni blocco richiede due accessi a disco: il primo accesso viene effettuato alla FAT ed il secondo al blocco di dati. Ne consegue un **accesso lento, affidabilità critica** (persa la FAT si perde tutto) e la dimensione della FAT è critica.



## Allocazione indicizzata ☺



Per garantire un accesso diretto efficiente è possibile inglobare tutti i puntatori in una tabella di puntatori detta **blocco indice** o **i-node** (index node). Ogni file ha la sua tabella, ovvero un vettore di indirizzi dei blocchi in cui il file è contenuto. L'i-esimo elemento del vettore individua l'i-esimo blocco del file.

Il direttorio contiene solo il puntatore al blocco indice (e non è una FAT perché i puntatori sono tutti in sequenza, cioè **non ho una lista di puntatori**).

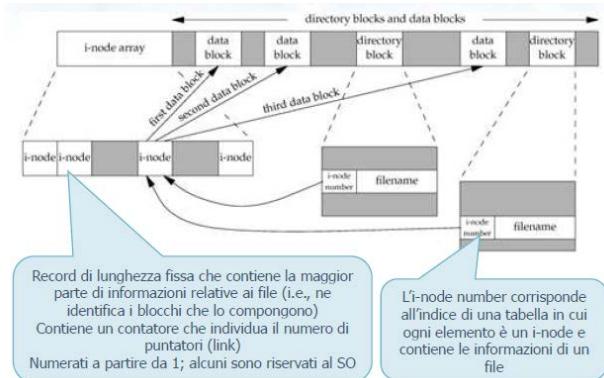
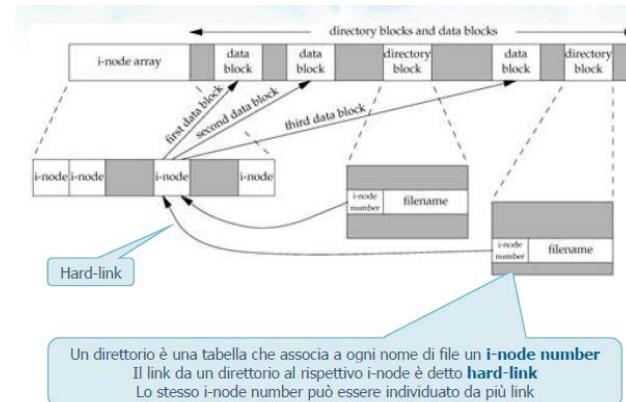
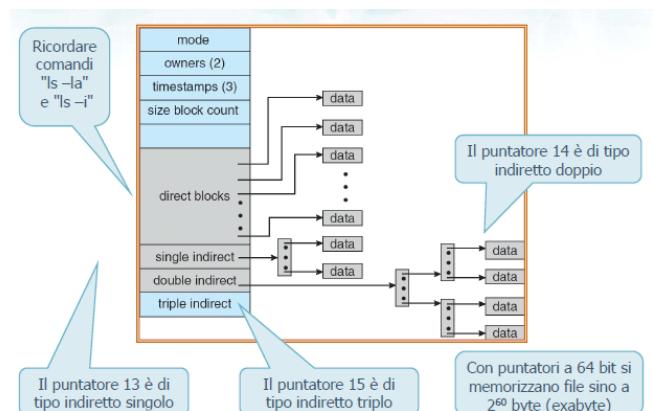
Rispetto all'allocazione concatenata, occorre sempre allocare un blocco indice.

I blocchi indice di dimensione ridotta permettono di non sprecare troppo spazio, mentre quelli di

dimensione elevata aumentano in numero di riferimenti inseribili nel blocco indice: in ogni caso occorre gestire situazioni in cui il blocco indice **non** è sufficiente a contenere tutti i puntatori ai blocchi del file. Esistono vari schemi, e quello UNIX/Linux è quello **combinato**.

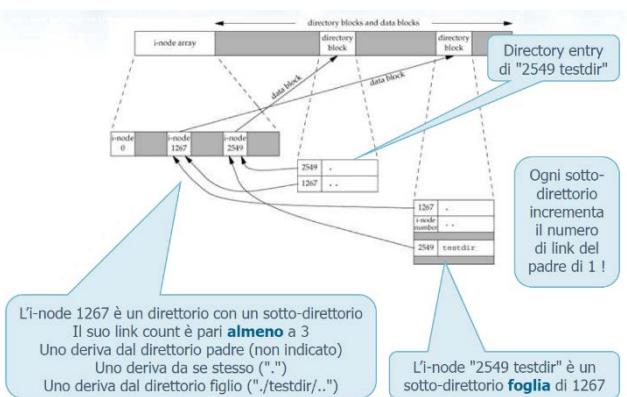
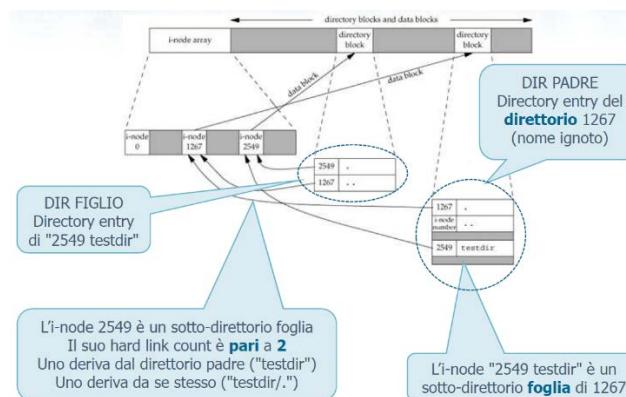
## Allocazione indicizzata: schema combinato ☺

A ogni file è associato un blocco detto **i-node** ed ognuno contiene diverse informazioni tra cui **15 puntatori ai blocchi dati del file**. I primi 12 sono puntatori diretti, ovvero puntano a blocchi del file, mentre i puntatori **13, 14 e 15** sono puntatori indiretti, con **livello di indirizzamento crescente**. Il blocco individuato **non contiene dati**, ma i puntatori (i puntatori ai puntatori) a blocchi di dati del file.



**Hard link (link effettivo o fisico):** Directory entry che punta a un i-node. Non esistono hard link verso directory e verso file su altri file system. Un file è fisicamente rimosso solo quando tutti i suoi hard link sono stati rimossi.

**Soft link (link simbolico):** Il blocco dati individuato dall'i-node punta a un blocco che contiene il path name del file. File che come unico blocco dati ha il nome di un altro file.



## Manipolazione del file system

Lo standard POSIX mette a disposizione un insieme di funzioni per effettuare la manipolazione dei direttori. La funzione **stat** permette di capire di che tipo di entry si tratta (file, directory, link) e tale operazione è permessa da una struttura C di tipo **struct stat**.

Altre funzioni di manipolazione sono: **getcwd**, **chdir**, **mkdir**, **rmdir**, **opendir**, **readdir**, **closedir**.

Utilizzando le seguenti librerie:

```
#include <sys/types.h>
#include <sys/stat.h>
```

si includono le funzioni **stat**, **Istat** (info su link simbolico) e **fstat** (informazioni su file aperto).

Il campo **st\_mode** della struct stat codifica il tipo di file. Alcune macro aiutano a capire il tipo di file:  
**S\_ISREG** (regular file), **S\_ISDIR** (directory).

Con la libreria:

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

si include la funzione **getcwd** che ottiene il path del directory di lavoro. Include anche **chdir** per modificare il path del directory di lavoro.

Definendo invece:

```
#include <unistd.h>
#include <sys/stat.h>
```

È possibile creare o rimuovere un nuovo directory tramite **mkdir** e **rmdir**.

Infine, la definizione di:

```
#include <dirent.h>
```

permette di utilizzare le funzioni **opendir**, **dirent** e **closedir** per aprire, visitare e chiudere un directory. La **opendir** restituisce il directory, mentre **dirent** richiede il directory già aperto e restituisce la **struct dirent** contenente le informazioni come nome file e il numero di i-node.

## 7. Introduzione ai processi ☺

**Esecuzione sequenziale:** sequenza di azioni eseguite una **dopo** l'altra. Da uno stesso input si genera sempre lo stesso output, indipendentemente dal momento e dalla velocità di esecuzione (e da quanti altri processi sono in esecuzione in quel momento).

**Esecuzione concorrente:** le azioni possono essere seguite allo **stesso** istante, con comportamento **non deterministico** e non esiste relazione d'ordine. La concorrenza può essere **reale** (nei sistemi multi-core) oppure **fittizia** (nei sistemi mono-processore).

Al **bootstrap** vengono eseguiti numerosi processi sia **automatici** (che vengono eseguiti all'avvio e terminano allo shut-down) come i *Daemon Process, attesa messaggi di posta, controllo e scan virus* oppure a **richiesta dell'utente** come i *servizi di gestione stampante, WEB server, etc..*

Di norma è possibile:

- **identificare e controllare** un processo tramite le system call quali *pid, getpid, getppid*
- **creare** un processo con le system call *fork, exec, system*. Quando viene creato un processo, il processo creante si dice processo **padre** e quello creato viene detto **figlio**. E' possibile dunque creare un **albero di processi**.
- **attendere, sincronizzare e terminare** i processi tramite le system call *exit, wait, waitpid*.

Ogni processo possiede un identificatore univoco detto **PID** (Process Identifier). In genere tale valore è un **intero non negativo** che, anche essendo univoco, in UNIX tali valori vengono riutilizzati.

Essendo univoci possono essere inclusi dal processo per generare oggetti unici (come ad esempio nel caso di processi diversi in esecuzione concorrente che devono scrivere su file, il PID può essere utilizzato per creare file univoci). Alcuni di questi identificatori sono riservati:

- **0** riservato per lo **schedulatore di processi** noto col nome di **swapper** ed eseguito a livello kernel.
- **1** riservato per il processo **init** che viene invocato alla fine del bootstrap ed eseguito a livello utente (con privilegi di super-user). La sua particolarità è che **non** termina mai e **diventa padre di ogni processo rimasto orfano**.

```
#include <unistd.h>

pid_t getpid(); // Process ID
pid_t getppid(); // Parent Process ID
uid_t getuid(); // User ID
gid_t getgid(); // Group ID
```

Al PID di un processo sono associati altri identificatori. Le funzioni come la *getpid* restituiscono gli identificatori del processo chiamante ma **non esiste** una system call per ottenere il PID di un figlio.

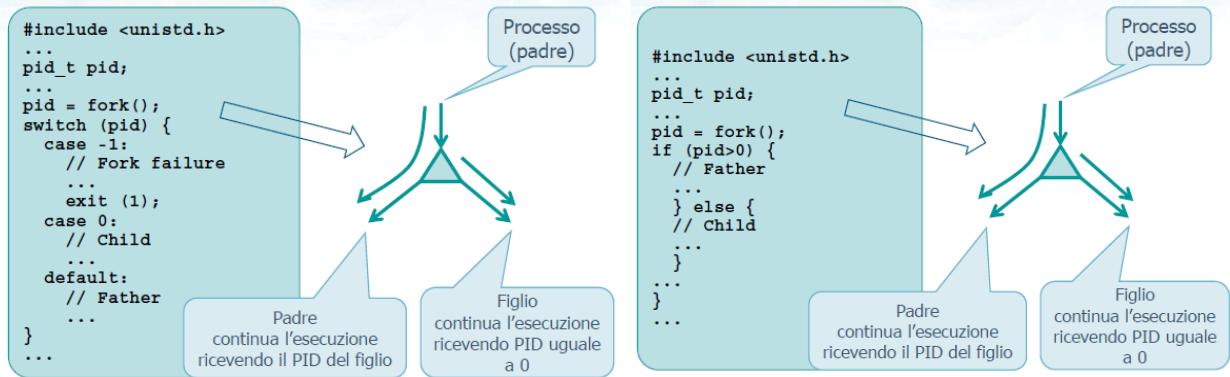
### Creazione di un processo ☺

Per creare un nuovo processo si utilizza la system call **fork()** che genera un processo detto **processo figlio**. Il figlio è una **copia identica** del padre tranne che per il **Process ID** ritornato dalla fork.

Il padre riceve l'ID del figlio (ogni processo può avere più figli e li distingue con il PID) mentre il figlio riceve valore **0** e può identificare il padre tramite la system call **getppid()**.

In pratica la fork viene chiamata una volta ma ritorna due volte (una nel padre e una nel figlio).

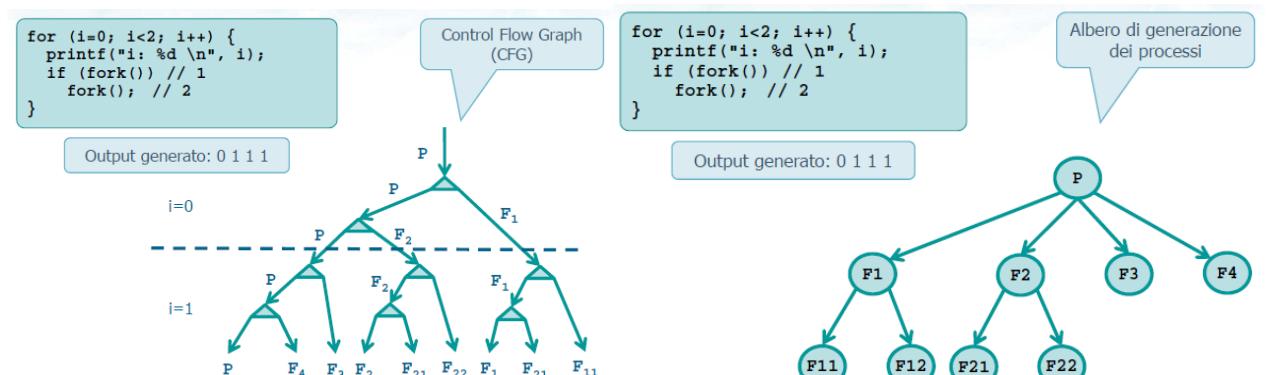
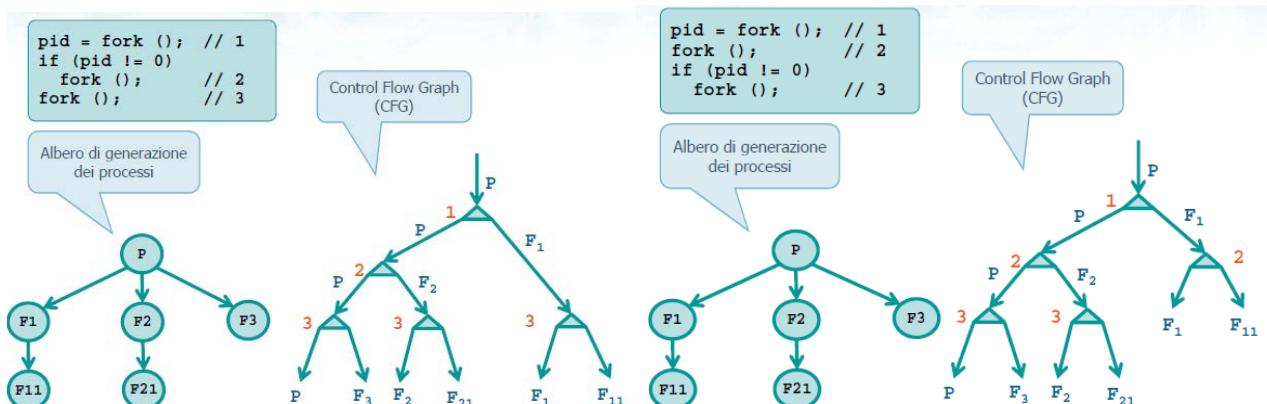
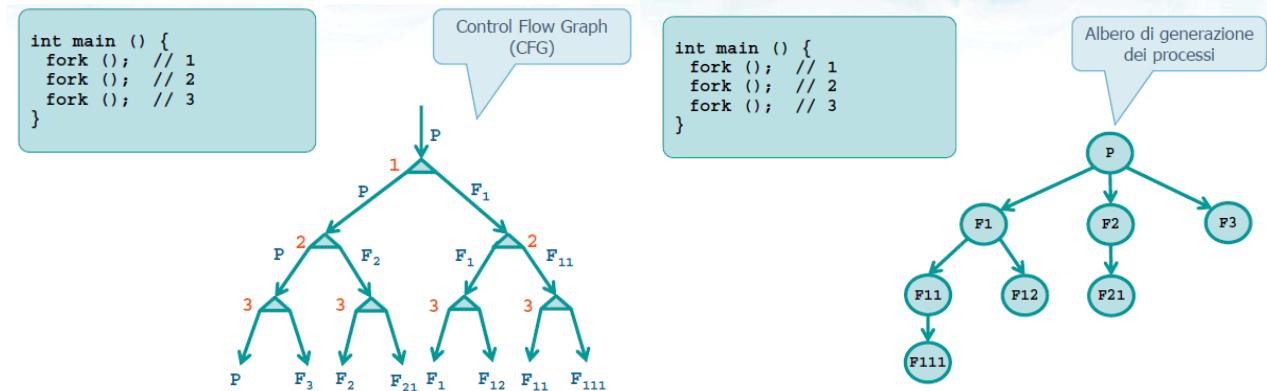
```
#include <unistd.h>           Se l'operazione non si conclude (non si può allocare un nuovo processo)
pid_t fork (void);          la fork restituisce valore -1.
```



La system call

### unsigned int sleep (unsigned int sec)

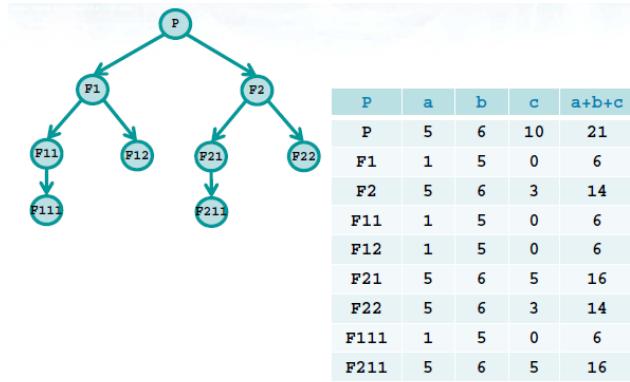
mette il processo in wait per (almeno) sec secondi.



- ❖ Dato il seguente programma riportarne l'output e il grafo di generazione dei processi

```
int main() {
    int a, b=5, c;
    a = fork(); /* #1 */
    if (a) {
        a = b; c = split(a, b++);
    } else {
        fork(); /* #2 */
        c = a++; b += c;
    }
    if (b > c) {
        fork(); /* #3 */
    }
    printf("%d", a+b+c);
    return 0;
}
```

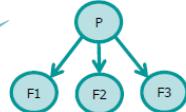
```
int split(int a, int b) {
    a++;
    a = fork(); /* #4 */
    if (a) {
        a = b;
    } else {
        if (fork()) /* #5 */ {
            a--;
            b += a;
        }
    }
    return a+b;
}
```



- ❖ Scrivere un programma concorrente che
- Dato un valore intero n
  - Sia in grado di generare n processi figlio
- ❖ Ciascun processo figlio visualizzi il proprio PID e termini

```
int i, n;
...
scanf ("%d", &n);
printf ("Start PID=%d\n",
       getpid());
for(i=0; i<n; i++) {
    if (fork() == 0) {
        printf ("Proc %d (PID=%d)\n",
               i, getpid());
        break;
    }
}
printf ("End PID=%d (PPID=%d)\n",
       getpid(), getppid());
exit(0);
```

```
> ps
PID TTY      TIME CMD
088 pts/10    00:00:00 bash
> ./04a01e06-fork
Star PID=3225
End PID=3225 (PPID=2088)
Proc 2 (PID=3228)
End PID=3228 (PPID=1314)
Proc 1 (PID=3227)
End PID=3227 (PPID=1314)
Proc 0 (PID=3226)
End PID=3226 (PPID=1314)
```



Il processo figlio è una nuova entry nella tabella dei processi. Le **risorse** del processo possono:

- **Essere condivise completamente tra padre e figli** perciò avranno stesso spazio di indirizzamento
- **Essere condivise in parte** tramite spazi di indirizzamento parzialmente sovrapposti
- **Non essere condivise affatto** tramite spazi di indirizzamento separati

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

- Il codice sorgente (C)
- Tutti i descrittori dei file (in particolare stdin e stdout)
- User ID, Group ID
- Root e Working Directory
- Risorse del sistema e limiti di utilizzo

Mentre si **differenziano** per:

- Valore ritornato dalla fork
- PID (Il padre conserva il PID, il figlio ne ottiene uno nuovo)
- Lo spazio dati
- Lo Heap
- Lo stack

Gli ultimi 3 punti in realtà non sono corretti perché moderni SO utilizzano la tecnica del **copy-on-write** e la memoria è duplicata solo se strettamente necessario, ovvero quando uno dei due processi effettua una scrittura.

```
char c, str[10];
c = 'X';
if (fork()) {
    // parent (!=0)
    c = 'F';
    strcpy (str, "father");
    sleep (5);
} else {
    // child (==0)
    strcpy (str, "child");
}

fprintf(stdout, "PID=%d; PPID=%d; c=%c; str=%s\n",
        getpid(), getppid(), c, str);
```

```
PID=2777; PPID=2776; c=X; str=father
PID=2776; PPID=2446; c=F; str=father
```

Output

## Terminazione di un processo ☺

Esistono 5 metodi standard per terminare un processo:

- Eseguire una **return** dalla funzione principale
- Eseguire una **exit**
- Eseguire una **\_exit** o **\_Exit** (effetti simili alla exit)
- Richiamare **return** dal main dell'ultimo thread del processo
- Richiamare **pthread\_exit** dall'ultimo thread del processo

Ed esistono 3 metodi anomali per terminare un processo:

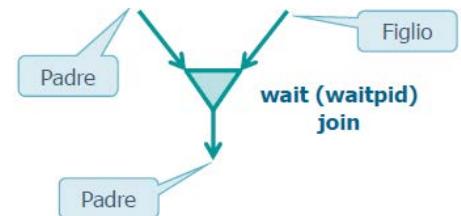
- Richiamare la funzione **abort** (sottocaso del successivo, poiché questa funzione genera il segnale SIGABORT)
- Ricevere un segnale (**signal**) di terminazione
- Cancellare l'ultimo thread del processo

Quando un processo termina tanto in maniera normale quanto anomala accade che:

- Il kernel invia un segnale **SIGCHLD** al padre
- La ricezione di un segnale da parte di un processo è un evento asincrono
- Perciò il padre può decidere di: **gestire** la terminazione del figlio in maniera **sincrona o asincrona**, oppure **ignorare** la terminazione del figlio (che è ciò che avviene di **default**).

Se un processo decide di gestire la terminazione di un figlio occorre effettuarne la gestione:

- Asincrona mediante un gestore del segnale **SIGCHLD**
- Sincrona tramite una chiamata alla system call **wait** o **waitpid**



```
#include <sys/wait.h>
```

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

Una chiamata alla system call **wait** da parte di un processo restituisce **errore** se il processo non ha figli, **blocca** il processo se **tutti** i figli del processo sono ancora in esecuzione e ritornerà non appena **uno dei figli termina**. Perciò, la wait restituisce 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. Se un processo termina e il padre non fa una wait, il suo stato di terminazione rimane **pendente**.

Il parametro **statLoc** è un puntatore a un intero (se non è NULL indica lo stato di uscita del processo figlio) e le informazioni sono interpretabili con le macro presenti in <sys/wait.h>: **WIFEXITED(statLoc)** è vero se la terminazione è stata corretta e in tal caso **WEXITSTATUS(statLoc)** cattura gli 8 LSBs del parametro passato a exit.

La funzione restituisce il **PID** del processo figlio terminato.

```
...  
pid_t pid, childPid;  
int statVal;  
...  
pid = fork();  
if (pid==0) {  
    // Child  
    sleep (5);  
    exit (6);  
} else {
```

```
// Father  
childPid = wait (&statVal);  
printf("Figlio terminato: PID = %d\n", childPid);  
if (WIFEXITED(statVal))  
    printf ("Valore restituito: %d\n",  
           WEXITSTATUS (statVal));  
else  
    printf ("Terminazione anomala\n");  
}  
exit(25);  
...  
echo $? (da shell) visualizza 25
```

## Processi Zombie 😊

Un processo terminato per il quale il padre non ha ancora eseguito una **wait** si dice **zombie**.

- Il segmento dati del processo **non** viene rimosso dalla process table per tenere traccia dello stato di uscita
- L'entry viene rimossa solo dopo che il padre ha eseguito una **wait** (se i padri non eseguono wait rimangono innumerevoli processi zombie)
- Se il padre termina prima di eseguire la **wait** il processo figlio viene ereditato dal processo **init** (quello con PID=1) e il figlio non diviene più zombie alla terminazione

## System call **waitpid()**

Se si desidera attendere un figlio specifico con una **wait** occorre **controllare** il PID del figlio terminato ed eventualmente morizzare il PID del figlio terminato nella lista dei processi figlio terminati (per eventuali ricerche) ed effettuare un'altra wait sino a quando termina il figlio desiderato.

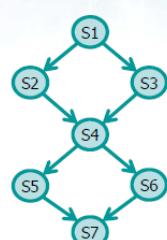
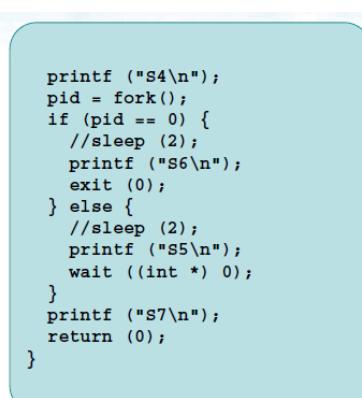
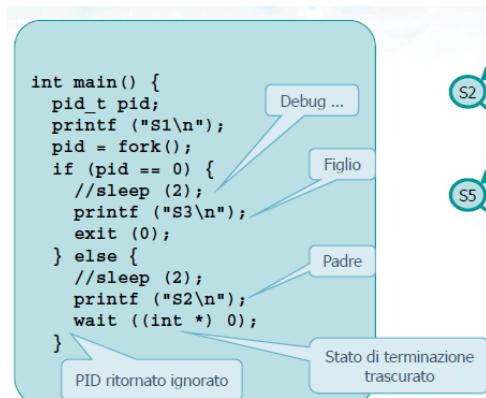
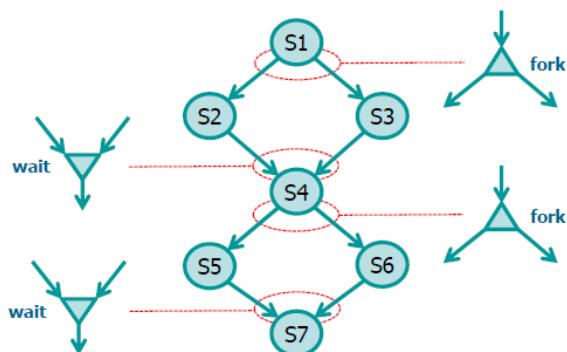
La **waitpid** si differenzia dalla **wait** in quanto **permette di non fermarsi** se tutti i figli sono in esecuzione e può attendere la **terminazione di un figlio specifico**.

```
#include <sys/wait.h>
```

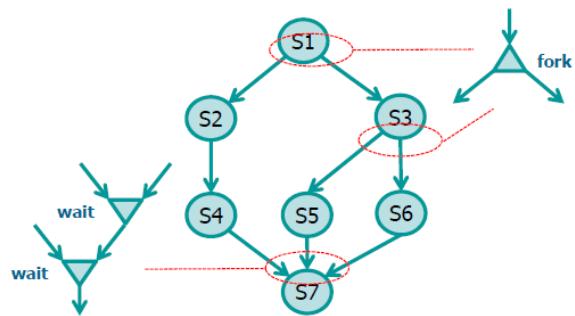
```
pid_t waitpid (pid_t pid, int *statLoc, int options);
```

Il parametro **pid** permette di attendere un **qualsiasi figlio** se è **-1** (e corrisponde alla wait) oppure un figlio specifico se il **PID=pid** è **> 0**. E' possibile anche attendere un qualsiasi figlio il cui group ID è uguale a quello del chiamante se **pid=0** oppure un figlio il cui group ID è uguale a **abs(pid)** se **< -1**. Il parametro **options** permette controlli aggiuntivi.

- ❖ Realizzare il seguente Control Flow Graph (CFG) tramite le system call **fork** e **wait**



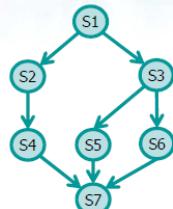
- ❖ Realizzare il seguente Control Flow Graph (CFG) tramite le system call **fork** e **wait**



```

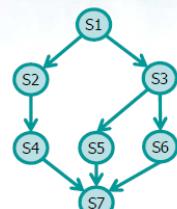
int main () {
    pid_t pid;
    printf ("S1\n");
    if ( (pid = fork()) == -1 )
        err_sys( "can't fork" );
    if ( pid == 0 ){
        P356();
    } else {
        printf ("S2\n");
        printf ("S4\n");
        while (wait((int *)0) != pid);
        printf ("S7\n");
        exit (0);
    }
    return (1);
}
  
```

Controllo su diverse terminazioni (inutile in questo caso e sostituibile con waitpid)



```

P356() {
    pid_t pid;
    printf ("S3\n");
    if ( (pid = fork()) == -1 )
        err_sys( "can't fork" );
    if (pid > 0){
        printf ("S5\n");
        while (wait((int *)0) != pid );
    } else {
        printf ("S6\n");
        exit (0);
    }
    exit (0);
}
  
```



- ❖ Scrivere un programma in grado di

- Ricevere sulla riga di comando un valore intero **n**
- Allocare dinamicamente un vettore di interi di dimensione **n** e leggerlo da tastiera
- Visualizzare (a video) gli elementi del vettore in ordine inverso (dall'elemento **n** all'elemento 0) utilizzando **n-1** processi ciascuno dei quali visualizza un singolo elemento del vettore
- Suggerimento
  - Sincronizzare i processi mediante system call **wait** in modo da stabilire l'ordine di visualizzazione degli elementi del vettore

```

int main(int argc, char *argv[])
{
    int i, n, *vet;
    int retValue;
    pid_t pid;
    n = atoi (argv[1]);
    vet = (int *) malloc (n * sizeof (int));
    if (vet==NULL) {
        fprintf (stderr, "Allocation Error.\n");
        exit (1);
    }
    fprintf (stdout, "Input:\n");
    for (i=0; i<n; i++) {
        fprintf (stdout, "vet[%d]:", i);
        scanf ("%d", &vet[i]);
    }
}
  
```

```

fprintf (stdout, "Output:\n");
for (i=0; i<n-1; i++) {
    pid = fork();
    if (pid>0) {
        pid = wait (&retValue);
        break;
    }
    fprintf (stdout, "Run PID=%d\n", getpid());
}

fprintf (stdout, "vet[%d]:%d - ", i, vet[i]);
fprintf (stdout, "End PID=%d\n", getpid());

exit (0);
}
  
```

## 8. Comandi di shell per la gestione dei processi

I comandi di shell standard permettono di eseguire processi in modo **sequenziale** e tali processi sono eseguiti in **foreground**. L'utilizzo del carattere **&** permette di eseguire processi in **background**. In questo modo il processo viene eseguito in maniera indipendente dalla shell e lascia il terminale libero per altri lavori e dunque è possibile eseguire processi in parallelo.

Es. **comando1 &**

Per controllare lo stato dei processi esistono due comandi principali:

- Il comando **ps** (**process status of active process**) elenca i processi attivi e i relativi dettagli. Senza opzioni (default) stampa (in formato compatto) lo stato dei processi con lo stesso user ID dell'utente da cui si effettua il comando.
  - **-a** elenca i processi di tutti gli utenti del sistema
  - **-u** elenca informazioni più dettagliate
  - **-u user** elenca processi dell'utente **<user>**
  - **-f** visualizza le informazioni in format esteso
- Il comando **top** visualizza informazioni sui processi in esecuzione, aggiornata in **run-time**.

### Il comando Kill

Il comando

**Kill [-sig] pid**

Permette di inviare un segnale a un processo dalla linea di comando di una shell. Il comando **invia** il segnale **sig** al processo di PID **pid**. L'opzione **sig** indica il **codice** del segnale, mentre il parametro **pid** è il **PID** del processo a cui inviare il segnale.

Ogni segnale **sig** può essere inviato mediante un nome o il numero corrispondente (la lista si può ottenere con l'opzione **-l**):

- **SIGKILL = KILL = 9**
- **SIGUSR1 = USR1 = 10**
- **SIGALRM = ALRM = 14**

Il segnale di default del comando kill è **SIGTERM (o TERM)** che è il comando di terminazione standard. Dunque per terminare (incondizionatamente) un processo è possibile usare questi 3 comandi equivalenti (supponendo PID=10234):

```
kill -9 10234  
kill -SIGKILL 10234  
kill -KILL 10234
```

### Comando killall

Il comando di shell

**killall [-sig] name**

termina tutti i processi di dato nome. Può essere utile per terminare tutti i processi generate dallo stesso programma senza specificarne esplicitamente tutti i PID.

```
killall -9 myProgram
```

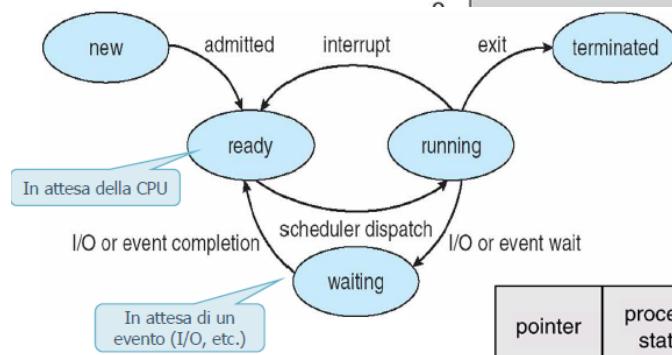
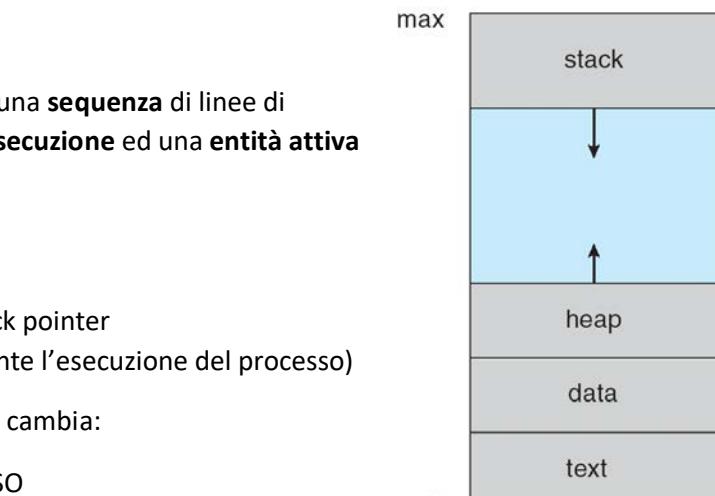
## 9. Processi: aspetti teorici ☺

Il programma è un'entità passiva composta da una sequenza di linee di codice. Il processo invece è un **programma in esecuzione** ed una entità attiva composta da:

- Codice sorgente e program counter
- Area dati (variabili globali)
- Stack (parametri e variabili locali) e stack pointer
- Heap (variabili dinamiche allocate durante l'esecuzione del processo)

Durante l'esecuzione di un processo il suo stato cambia:

- **New**: processo creato e sottomesso al SO
- **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 un qualche evento
- **Terminated**: il processo termina e rilascia le risorse utilizzate.



pointer	process state
process number	
program counter	
registers	
memory limits	
list of open files	
⋮	

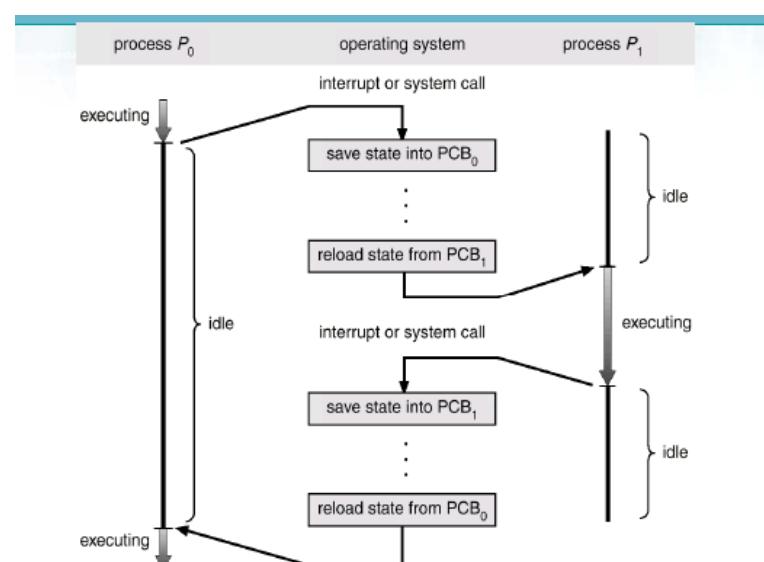
### Process Control Block (PCB) ☺

Il SO tiene traccia di ogni processo associando a esso un insieme di dati. Tali dati includono: lo **stato del processo** e il **program counter** (indirizzo della successiva istruzione da eseguire); i **registri della CPU** (in numero e tipo dipendente dall'hardware); le **informazioni utili per lo scheduling della CPU** (priorità, puntatori alle code di scheduling); le **informazioni utili per la gestione della memoria** come il registro base, il registro limite, tabelle pagine o segmenti; **informazioni amministrative varie** come il tempo di utilizzo CPU, limiti; **informazioni sullo stato delle operazioni di I/O** come la lista dei dispositivi I/O e la lista dei file aperti.

### Context switching ☺

Quando la CPU viene assegnata ad un altro processo, il kernel deve **salvare** lo stato del processo running e **caricare** un nuovo processo ripristinandone lo stato salvato precedentemente. Il tempo dedicato al context switching è **overhead**, cioè **lavoro non utile** direttamente ad alcun processo.

Il tempo impiegato dipende da svariati fattori: hardware, SO, numero di processi, politica di scheduling, etc..



## Scheduling dei processi ☺

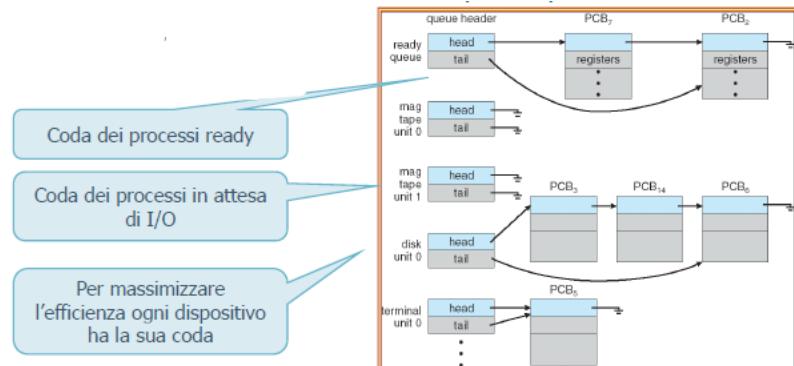
L'obiettivo della multiprogrammazione è quello di massimizzare l'utilizzo della CPU da parte dei processi. Tali processi sono classificati in:

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

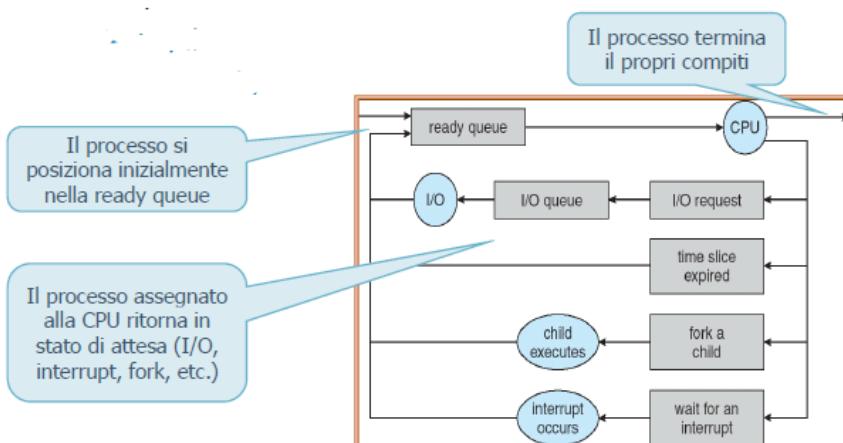
Per massimizzare l'utilizzo della CPU e soddisfare eventuali vincoli sul tempo di risposta, ogni SO gestisce i processi mediante uno **scheduler**, che ha il compito di selezionare tra i processi disponibili il successivo da eseguire. Esistono diversi tipi di scheduler:

- **Scheduler a lungo termine**: si occupa di schedulare i processi **su disco** ed interviene meno frequentemente e rischedula a tempi dell'ordine dei secondi/minuti. Seleziona il processo da inserire nella **ready list** e quali sono pronti all'esecuzione in memoria centrale. In sostanza controlla il grado di multiprogrammazione.
- **Scheduler a breve termine**: schedula i processi in **RAM** e seleziona prevalentemente i processi per la CPU. Interviene molto frequentemente (rischedulando all'ordine dei millisecondi) e deve essere molto veloce.

Lo scheduler gestisce i processi in attesa di un dispositivo tramite **code (di processi)** ed esistono diverse code (una per ogni dispositivo). Le code non sono altro che **liste concatenate**.



Il **diagramma di accodamento** specifica la gestione dei processi nelle varie code. Ogni rettangolo rappresenta una coda.



## 10. Controllo avanzato ☺

La system call **fork** permette di duplicare un processo, ma esistono due principali applicazioni di tale meccanismo:

- Padre e figlio eseguono **sezioni diverse** di codice
- Padre e figlio eseguono **codici differenti** (comune a tutte le shell e richiede l'utilizzo della famiglia di comandi **exec**).

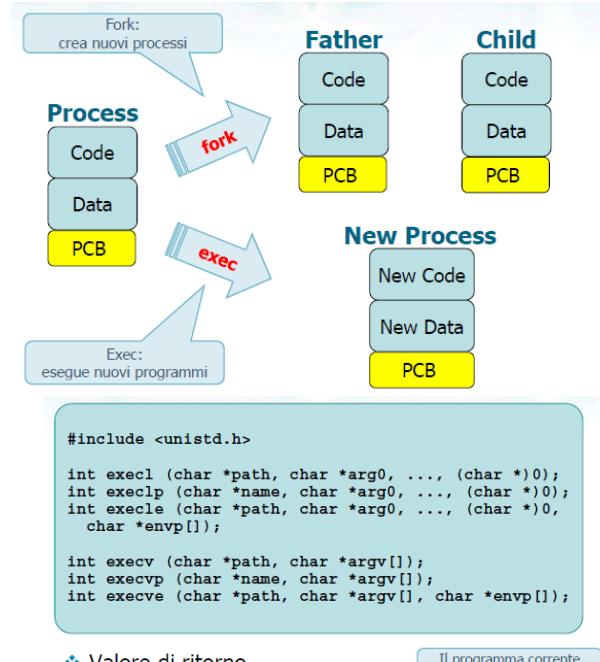
La system call **exec sostituisce** il processo con un nuovo programma che inizia l'esecuzione in maniera standard (cioè dal main). La exec **non crea** un nuovo processo ma **sostituisce** l'immagine del processo corrente con quelli di un processo nuovo e il **PID non cambia**. Ricapitolando:

- La fork **duplica** un processo **esistente**
- La exec esegue un **nuovo** programma

Esistono sei versioni della system call exec:

- execl, execl, execle
- execv, execvp, execve

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



❖ Valore di ritorno

- **Nessuno**, in caso di successo
- Il valore -1, in caso di errore

Il programma corrente non esiste più

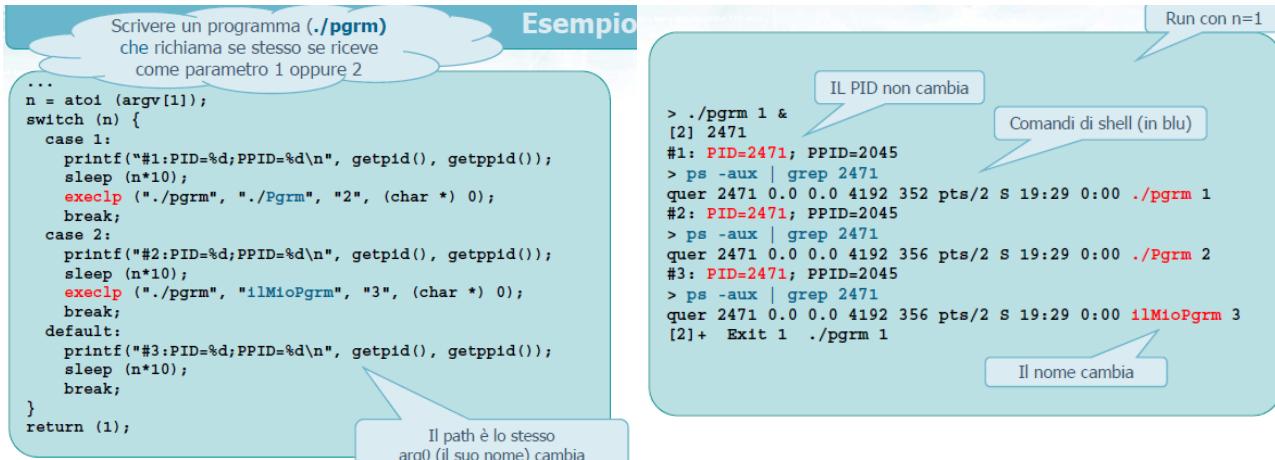
La funzione exec riceve il **path (nome)** del programma da eseguire, che può essere il nome di un file oppure il nome del file seguito dal relativo path. Inoltre, nelle versioni "p" è sufficiente specificare solo il nome del file. Nelle versioni non-"p" il nome dovrebbe includere il path.

Nelle versioni "l" la exec riceve un elenco di parametri (come il main C) che contiene nella prima posizione il nome dell'eseguibile e in tutte le altre posizioni contiene i parametri dell'eseguibile.

Nelle versioni "v" l'argomento è un vettore di puntatori agli argomenti stessi ed è simile alla matrice dinamica \*\*argv (non identica perché il valore argv[i]==NULL indica la fine degli argomenti).

Nelle versioni non-"e" le **variabili di ambiente** sono ereditate dal processo chiamante, mentre nelle versioni "e" le variabili di ambiente sono specificate esplicitamente. Si indica una seconda matrice dinamica NULL-terminated, ovvero un vettore di puntatori a stringhe di caratteri. Tali stringhe specificano i valori delle variabili di ambiente desiderate.





La `execv[p]` utilizza come parametro un unico puntatore che individua un vettore di puntatori ai parametri (il vettore va opportunamente inizializzato).

```

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

```

L'ultimo argomento è indicato con un puntatore NULL

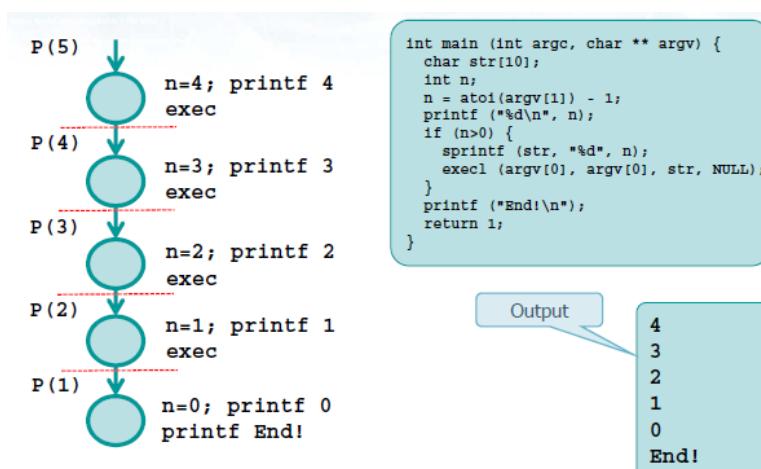
La `exec[lv]e` specifica esplicitamente l'**environment** ed è un puntatore a un vettore di puntatori. Per le altre funzioni l'environment è ereditato dal processo chiamante.

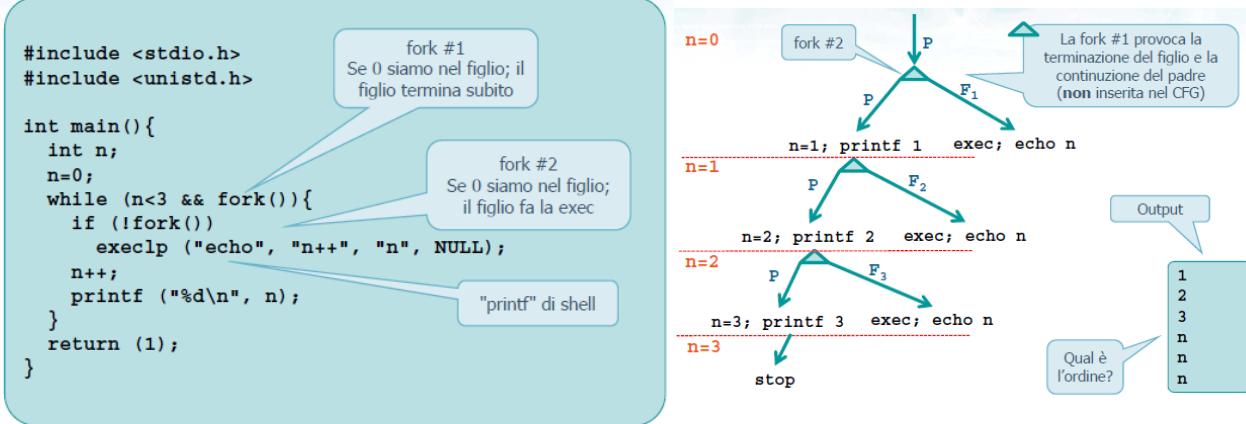
```

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

```

Si osservi che durante la exec vengono mantenuti tutti i file descriptor esistenti (compresi stdin, stdout, stderr) e questo comportamento è necessario per **ereditare** eventuali redirezioni impostate nei comandi di shell una volta eseguite la exec. In molti sistemi operativi esiste solo una versione della exec (in generale la `execve`) che viene implementata come system call. Tutte le altre versioni chiamano tale versione.





## Scheletro di una shell ☺

Quando un comando viene eseguito in foreground, il comportamento è il seguente:

- ❖ Comando eseguito in foreground
  - <comando>

```

while (TRUE) {
    write_prompt ();
    read_command (command, parameters);
    if (fork() == 0)
        /* Child: Execute Command */
        execve (command, parameters);
    else
        /* Father: Wait Child */
        wait (&status);
}

```

Il programma cambia il processo no. Il padre rimane tale e può fare la wait.

Quando invece il comando viene eseguito in background (con &) il padre non attende e il comportamento è il seguente:

- ❖ Comando eseguito in background
  - <comando> &

```

while (TRUE) {
    write_prompt ();
    read_command (command, parameters);
    if (fork() == 0)
        /* Child: Execute Command */
        execve (command, parameters);
#if 0
    else
        /* Padre: NON Attende */
        wait (&status);
#endif
}

```

Può essere conveniente eseguire una **stringa di comando** dall'interno di un programma in esecuzione, ad esempio può essere utile inserire data o ora nel nome o nel contenuto di un file. A tale scopo è nata la funzione **system**

**#include <stdlib.h>**

**Int system (const char \*string);**

Questa system call passa il comando **string** all'ambiente host affinché questo lo esegua. In pratica invoca il comando string all'interno di una shell. Il controllo viene restituito al processo chiamante una volta che l'esecuzione del comando è terminata.

Essendo implementata con fork, exec e wait la funzione ha diverse condizioni di terminazione:

- -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 (con formato specificato dalla waitpid)

**Esempi**

```

...
system ("date");
...
system ("date > file");
...

...
system ("ls -laR");
...

char str[L];
...
strcpy (str, "ls -la");
system (str);
...

```

Le versioni iniziali di UNIX implementavano la system call system tramite fork, exec e wait ed erano inefficienti a causa del polling (della wait che doveva controllare la terminazione di tutti i processi)

```
while ( (lastpid=wait(&status)) != pid
    && lastpid!=-1 );
```

Per tali motivi adesso si utilizza la **waitpid** al posto della wait.

```

int system (const char *cmd) {
    pid_t pid;
    int status;
    if (cmd == NULL)
        return(1);
    if ( (pid = fork()) < 0) {
        status = -1;
    } else if (pid == 0) {
        execl("/bin/sh", "sh", "-c", cmd, (char *) 0);
        _exit(127);
    } else {
        while (waitpid (pid, &status, 0) < 0)
            if (errno != EINTR) {
                status = -1;
                break;
            }
    }
    return(status);
}

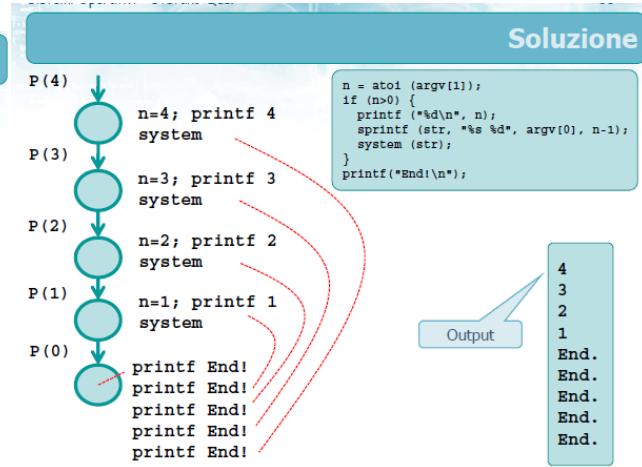
```

### Esercizio

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

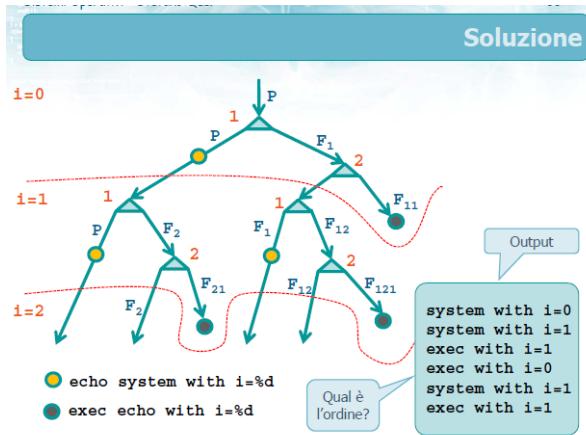
int main(int argc, char ** argv){
    int n;
    char str[10];
    n = atoi (argv[1]);
    if (n>0) {
        printf ("%d\n", n);
        sprintf (str, "%s %d", argv[0], n-1);
        system (str);
    }
    printf("End!\n");
    return (1);
}
```

Run con n=4



### Esercizio

```
#include ...
int main () {
    char str[100];
    int i;
    for (i=0; i<2; i++){
        if (fork()!=0) {
            sprintf (str, "echo system with i=%d", i);
            system (str);
        } else {
            if (fork()==0) {
                sprintf (str, "exec echo with i=%d", i);
                execvp ("echo", "myPgrm", str, NULL);
            }
        }
    }
    return (0);
}
```



## 11. I segnali ☺

L'**interrupt** è un'interruzione del procedimento corrente dovuto al verificarsi di un evento straordinario. Esso viene generato da un dispositivo hardware che invia una richiesta di servizio alla CPU, ma anche dai **processi software** che richiede l'esecuzione di una particolare **operazione tramite system call**.

Un **segnale** è un interrupt software, cioè un evento di sistema inviato a un **processo** da un **altro processo**. Il meccanismo è tendenzialmente insicuro e impreciso, alcuni dei quali ancora permangono e sono inaffidabili per effettuare alcune cose (infatti si utilizzano i semafori). Il vantaggio dei segnali è che **permettono di gestire eventi asincroni**, notificando il verificarsi di eventi particolari in maniera asincrona.

I segnali possono essere utilizzati anche per la **comunicazione** tra processi (che equivale al codice del segnale, poche informazioni).

Alcuni esempi di segnali comuni sono:

- **SIGCHLD** che corrisponde alla **terminazione di un figlio** e viene inviato al padre (l'azione default è ignorare il segnale).
- **SIGINT** corrisponde a premere sul terminale Ctrl+C e viene inviato al processo in esecuzione e ha come azione di default quella di terminare il processo.
- **SIGTSTP** corrisponde ad un accesso in memoria non valido inviato dal kernel al processo. L'azione di default è quella di sospendere l'esecuzione.
- **SIGALARM** corrisponde alla system call sleep (t) e viene inviato dopo t secondi e la sua azione di default è quella di far ripartire il processo.

Alcuni segnali non possono essere modificati dal loro comportamento di default.

Segnali inviati dall'exception handler		
Eccezione	Exception handler	Segnale
Divide error	divide_error()	SIGFPE
Debug	debug()	SIGTRAP
Breakpoint	int3()	SIGTRAP
Overflow	overflow()	SIGSEGV
Bounds check	bounds()	SIGSEGV
Invalid opcode	invalid_op()	SIGILL
Segment not present	segment_not_present()	SIGBUS
Stack segment fault	stack_segment()	SIGBUS
General protection	general_protection()	SIGSEGV
Page fault	page_fault()	SIGSEGV
Interval reserved	none	None
Floating point error	coprocessor_error()	SIGFPE

I segnali sono disponibili dalle prime versioni di UNIX ed originariamente erano gestiti in maniera poco affidabile perché **potevano andare perduti** (e ancora in parte esiste il problema perché i segnali **non** hanno delle code e in caso di invio multipli alcuni potrebbero essere perduti). Un altro difetto delle vecchie versioni è che il gestore dei segnali (signal handler) doveva essere ricaricato tutte le volte. Se non si vuole adottare il comportamento standard dei segnali ma lo si vuole modificare, è necessario stanziare un gestore dei segnali (uno per ogni segnale che resta in vita sempre). Il processo, infine, non poteva ignorare la ricezione di un segnale.

I segnali hanno un nome standard che inizia per **SIG** (standardizzato dallo standard POSIX) e sono definiti nel **signal.h** e nei sistemi operativi ne esistono circa 31.

Alcuni segnali principali sono:

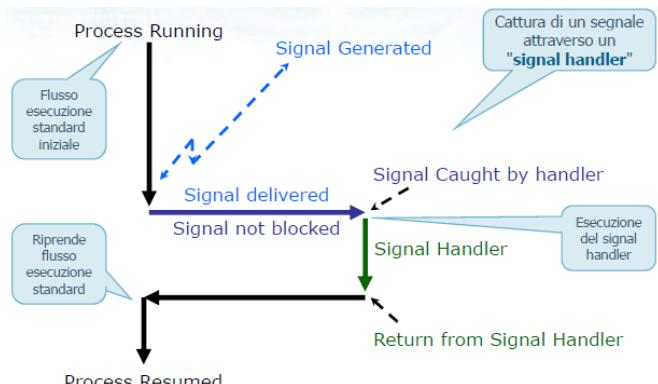
Segnali principali	
Nome	Descrizione
SIGABRT	Process abort, generato chiamando la funzione abort
SIGALRM	Alarm clock, generato dalla funzione alarm
SIGFPE	Floating-Point exception
SIGILL	Illegal instruction
SIGKILL	Kill (non mascherabile)
SIGPIPE	Write on a pipe with no reader
SIGSEGV	Invalid memory segment access
SIGCHLD	Child process stopped or exited
<b>SIGUSR1</b>	User-defined signal 1/2
<b>SIGUSR2</b>	Comportamento di default: terminazione Disponibile per utilizzo in applicazioni utente

I segnali **SIGUSR1** e **SIGUSR2** sono disponibili per l'utilizzo in applicazioni utente che non hanno significato particolare.

### Gestione dei segnali ☺

La gestione dei segnali implica tre fasi:

- **Generazione del segnale:** in genere molto semplice e viene utilizzata la system call `kill` (che invia a qualcuno un segnale).
- **Consegna del segnale:** a carico del sistema operativo che si occupa di gestirlo. Prima di essere consegnato un segnale risulta pendente, una volta consegnato il destinatario assume le azioni richieste dal segnale. Il segnale ha un tempo di vita che va dalla generazione alla consegna.
- **Gestione del segnale:** per gestirlo è possibile utilizzare il comportamento di default (implicitamente o esplicitamente), ignorare il segnale oppure **gestire (catch)** il segnale. Il processo dovrà comunicare al kernel cosa fare nel caso in cui lo si voglia gestire.



Nella foto a destra vi è il processo standard, finché un altro processo non genera un segnale e lo invia al processo. Tale segnale verrà consegnato (se il processo non lo blocca) e nel caso in cui process lo voglia gestire (in modo non default) dovrà aver caricato un gestore di segnali e vi sarà un interrupt software e il processo viene interrotto e si salta all'interno della routine di gestione del segnale. Una volta terminata la routine di gestione del segnale, l'esecuzione riprende. L'interruzione viene gestita dal sistema operativo poiché il signal handler non conosce il punto di interruzione del programma (perciò deve essere arbitrario). Tutto il meccanismo viene gestito attraverso 4/5 system call:

- **signal:** istanzia un gestore di segnali (non invia segnali)
- **kill:** invia un segnale (non uccide nessuno)
- **pause:** sospende un processo sino all'arrivo di un segnale
- **alarm:** invia un segnale **SIGALARM** dopo un tempo prestabilito

## System call signal() 😊

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

Parametro del signal handler ricevuto  
Parametro del signal handler ritornato

La system call signal consente di instanziare un gestore di segnali che si occuperà di gestire un determinato segnale invece di utilizzare il comportamento di default.

Il primo parametro indica il **codice del segnale** (int sig).

Il secondo parametro indica il **puntatore** alla funzione che gestisce il segnale sig. La funzione è particolare perché deve ricevere un intero (codice del segnale) e restituisce un void\* (puntatore al gestore attivo sino a quel momento).

Tutte le volte che si effettua una signal si specifica il nome di una nuova funzione ma si riceve il nome della funzione precedente che gestiva lo stesso segnale. Quelli interni (void e int iniziali e finali) sono quelli di ritorno.

La funzione, in caso di errore, restituisce **SIG\_ERR** che “simula” una funzione che ritorna **((void (\*)()) -1**.

Esiste anche il codice **SIG\_DFL** che simula **((void (\*)()) 0**, ovvero il **comportamento di default** per un determinato segnale. Per impartire il comportamento di default si effettuerà una chiamata del tipo:

**signal(SIG..., SIG\_DFL)**

e va effettuata una chiamata per ogni segnale che si vuole esplicitare.

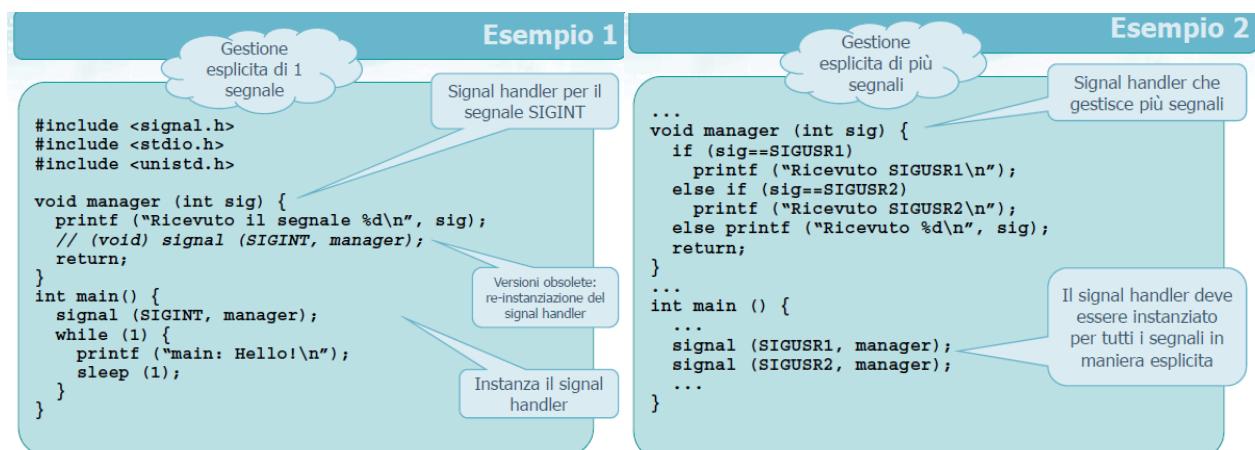
Se invece si vuole **ignorare esplicitamente** un segnale, si effettuerà la chiamata al segnale **SIG\_IGN** **((void (\*)()) 1**.

**signal (SIG..., SIG\_IGN)**

Alcuni segnali non possono essere ignorati (come il SIGKILL e SIGSTOP).

Se il processo vuole catturare il segnale, bisogna stanziare un gestore di segnali, cioè verrà chiamato il puntatore alla funzione che gestirà il segnale.

**signal(SIG..., signalHandlerFunction)**



Si nota come nelle versioni obsolete bisognava istanziare nuovamente il gestore. Oggi non è più necessario. Si nota inoltre che lo stesso manager gestisce entrambi i segnali, ma viene istanziato due volte il segnale signal (uno per segnale).

### Esempio 3-A

```
Gestione sincrona di SIGCHLD (con wait)

if (fork() == 0) {
    // child
    sleep (1);
    printf ("i=%d PID=%d\n", i, getpid());
    exit (i);
} else {
    // father
    sleep (5);
    pid = wait (&code);
    printf ("Wait: ret=%d code=%x\n", pid, code);
}
```

Quando un figlio muore al padre viene inviato un SIGCHLD

Wait: ret = 3057 code = 200

### Esempio 3-B

```
Comportamento mascherato di una wait

signal (SIGCHLD, SIG_IGN);

if (fork() == 0) {
    // child
    sleep (1);
    printf ("i=%d PID=%d\n", i, getpid());
    exit (i);
} else {
    // father
    sleep (5);
    pid = wait (&code);
    printf ("Wait: ret=%d code=%x\n", pid, code);
}

L'esecuzione di una signal(SIGCHLD,SIG_IGN) evita che i figli diventino degli zombie mentre una signal(SIGCHLD,SIG_DFL) non è sufficiente a tale scopo (anche se SIGCHLD viene ignorato)
```

Disabilita (SIG\_IGN) la gestione del segnale SIGCHLD, ricevuto alla terminazione di un figlio

PID=3057

Nessuna attesa:  
Wait: ret = -1 code = 7FFF

### Esempio 3-C

```
Gestione asincrona del segnale SIGCHLD

static void sigChld (int signo) {
    if (signo == SIGCHLD)
        printf("Received SIGCHLD\n");
    return;
}
...
signal(SIGCHLD, sigChld);
if (fork() == 0) {
    // child
    ...
    exit (i);
} else {
    // father
    ...
}
```

Gestione **sincrona** della terminazione di un figlio. Quando il figlio termina genera un segnale SIGCHLD che viene consegnato al padre e la cattura viene fatta attraverso la wait.

Il figlio dorme un secondo, stampa e termina col codice i (che può essere catturato con le macro).

Il padre fa una sleep di 5 (dovrebbe svegliarsi dopo) e fa la wait che ritorna il pid e il codice di terminazione (in code c'è i).

E' possibile ignorare SIGCHLD impartendo tramite una signal il comando di SIG\_IGN. A questo punto la wait dovrebbe catturare il segnale SIGCHLD ma essendo ignorato, restituisce -1 cioè come se il figlio non esistesse. La wait, dunque, non ha significato.

L'ultimo caso è la cattura **asincrona** del segnale. Il gestore del segnale si chiama sigChld e se riceve SIGCHLD stampa semplicemente un messaggio.

Viene quindi fatta una signal dove viene avviato il gestore del segnale e in qualsiasi punto del codice, quando verrà terminato un figlio, si stamperà un messaggio.

## System call kill() ☺

Per inviare un segnale si utilizza:

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

dove si specifica il PID a cui inviare il segnale e il codice del segnale. Si può inviare segnali **solo a processi** di cui si ha **diritto**. Il superuser può inviare segnali a tutti i processi.

La kill può essere utilizzata per conoscere se un processo è ancora in vita. In particolare, se il campo **sig** è pari a 0 (sig=0) si invia un segnale NULL (non si invia alcun segnale) e se si riceve -1 tale processo non esiste (0 in caso di successo).

### System call raise()

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

La raise invia un segnale al processo stesso (è come fare una kill col PID del processo stesso).

### System call pause() ☺

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

Sospende il processo chiamante sino all'arrivo di un segnale. Ritorna solo quando viene eseguito un gestore di segnali e questo termina la sua esecuzione (in questo caso la funzione restituisce -1).

### System call alarm() ☺

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

Attiva un timer (countdown) al termine del quale viene inviato un **SIGALRM**. Si specificano il numero di secondi e al termine si invia il segnale al processo stesso. Se il timer viene risettato, la funzione restituisce il tempo

mancante. Anche alarm utilizza lo stesso timer della sleep, perciò non è detto che sono n secondi effettivi, ma dipende dallo scheduler.

Si può implementare la sleep tramite **alarm** e **pause**.

Si chiama il gestore di segnali per evitare che il segnale sia inaspettato, successivamente il main impone un alarm e si mette in pausa finché l'alarm non scade e lo sveglia. La system call **alarm** può essere implementata utilizzando **fork**, **signal**, **kill**, **pause**, **sleep**.

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

void myAlarm (int sig) {
    if (sig==SIGALRM)
        printf ("Alarm on ... \n");
    return;
}
```

Instanzio il gestore di segnali, faccio una fork e il figlio dorme e poi manda un segnale al padre. Il padre andrà in pausa finché non riceverà il segnale dal figlio. Il gestore è necessario al padre, visto che il figlio non deve gestire il segnale, perciò si potrebbe spostare la funzione signal al padre, ma il figlio non è certo che il padre abbia stanziatato il gestore dei segnali, perciò è più corretto inserire la signal al di fuori della fork, in modo che si sia sicuri che il segnale venga gestito. Un altro problema è che il padre non abbia ancora raggiunto pause quando riceve il segnale, andando quindi a dormire senza più essere risvegliato.

**Inserire la signal prima delle fork SEMPRE.**

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

static void sig_alarm(int signo) {return;}
unsigned int sleep1(unsigned int nsecs)
{
    if (signal(SIGALRM, sig_alarm) == SIG_ERR)
        return (nsecs);
    alarm (nsecs);
    pause ();
    return (alarm(0));
}
```

Il gestore va instanziato **prima** di settare l'allarme  
Si imposta un allarme e si va in pausa

```
int main (void) {
    pid_t pid;
    (void) signal (SIGALRM, myAlarm);
    pid = fork();
    switch (pid) {
        case -1: /* error */
            printf ("fork failed");
            exit (1);
        case 0: /* child */
            sleep(5);
            kill (getppid(), SIGALRM);
            exit(0);
    }
    /* father */
    ...
    return (0);
}
```

Il figlio attende e invia il segnale SIGALRM  
Il padre procede come deve e riceverà il segnale SIGALRM dal figlio

## Limiti dei segnali ☺

Il primo limite dei segnali è che l'informazione è ridotta: non è possibile inviare dati.

Un altro limite è la **memoria** dei segnali “pending” **limitata**. Si ha al massimo un segnale “pending” (invia ma non consegnato) per ciascun tipo di segnale. I segnali successivi (dello stesso tipo) vengono perduti.

Inoltre, i segnali **possono essere bloccati** e quindi **non essere ricevuti**.

```
...
static void sigUsr1 (int);
static void sigUsr2 (int);

static void
sigUsr1 (
    int signo
) {
    if (signo == SIGUSR1)
        printf("Received SIGUSR1\n");
    else
        printf("Received wrong SIGNAL\n");

    fprintf (stdout, "sigUsr1 sleeping ...\\n");
    sleep (5);
    fprintf (stdout, "... sigUsr1 end sleeping.\\n");
    return;
}
```

Programma con 2 gestori di segnale: **sigUsr1** e **sigUsr2**

Vengono gestiti i due segnali **sigusr1** e **sigusr2**. Il primo gestore (**sigusr1**) stampa un segnale alla ricezione di **SIGUSR1**, aspetta 5 secondi e riparte.

```
static void
sigUsr2 (
    int signo
) {
    if (signo == SIGUSR2)
        printf("Received SIGUSR2\\n");
    else
        printf("Received wrong SIGNAL\\n");

    fprintf (stdout, "sigUsr2 sleeping ...\\n");
    sleep (5);
    fprintf (stdout, "... sigUsr2 end sleeping.\\n");

    return;
}
```

Programma con 2 gestori di segnale: **sigUsr1** e **sigUsr2**

**SigUsr2** effettua la stessa identica cosa di **SigUsr1**.

```
int
main (void) {
    if (signal(SIGUSR1, sigUsr1) == SIG_ERR) {
        fprintf (stderr, "Signal Handler Error.\\n");
        return (1);
    }
    if (signal(SIGUSR2, sigUsr2) == SIG_ERR) {
        fprintf (stderr, "Signal Handler Error.\\n");
        return (1);
    }
    while (1) {
        fprintf (stdout, "Before pause.\\n");
        pause ();
        fprintf (stdout, "After pause.\\n");
    }
    return (0);
}
```

Il main instanzia i gestori di segnali e itera in attesa di segnali (da shell)

Infine, il main istanzia i due gestori di segnali e va in un ciclo infinito in cui fa delle pause.

Il programma si sveglierà dalla pausa tutte le volte che riceverà un **SIGUSR1** o **SIGUSR2**

Se si fa girare questo programma in background è possibile inviarigli dei segnali tramite il comando

**kill -USR1 PID** il comando fa svegliare il processo e lo rimette a dormire.

Se si invia più volte lo stesso comando su shell come segue:

**kill -USR1 PID ; kill -USR1 PID ; kill -USR1 PID**

L'effetto è che due di questi segnali vengono persi. Se invece si invia il comando seguente i segnali vengono ricevuti entrambi.

**kill -USR1 PID ; kill -USR2 PID**

## Funzioni rientranti (reentrant) ☺

Il comportamento in presenza di un segnale prevede:

- Interruzione del flusso di istruzioni corrente
- Esecuzione signal handler
- Ritorno al flusso standard alla terminazione del signal handler

Perciò in questo caso il kernel **sa** da dove riprendere il flusso di istruzioni precedente, mentre il signal handler **non sa** dove esso è stato interrotto dal segnale.

Se il signal handler effettua una operazione **non compatibile** con il flusso di esecuzione originale possono verificarsi problemi.

Se ad esempio si **interrompe** una malloc e il signal handler effettua un'altra malloc la funzione malloc mantiene una lista delle aree di memoria rilasciate e tale lista può essere corrotta.

Se invece l'esecuzione di una funzione che utilizza una **variabile statica** viene interrotta e richiamata dal signal handler la variabile può essere utilizzata per memorizzare un valore e quindi può essere sovrascritta.

La "Single UNIX specification" definisce le **funzioni rientranti** che possono essere **interrotte senza problemi**:

**read, write, sleep, wait, etc..**

Mentre la maggior parte delle funzioni I/O standard C non lo sono (printf, scanf).

## Race conditions

Si ha una **race condition** quando il comportamento di più processi che lavorano su dati comuni dipende dall'ordine di esecuzione. In particolare, l'utilizzo di segnali tra processi, essendo un evento asincrono, può generare race conditions che portano il programma a non funzionare nel modo desiderato.

### Race conditions: Esempio A

- ❖ Si supponga un processo voglia risvegliarsi dopo un certo numero di secondi

Vedere implementazione di **sleep** mediante **alarm** e **pause**

```
static void  
myHandler (int signo) {  
    ...  
}  
...  
signal (SIGALARM, myHandler)  
alarm (nSec);  
pause ();
```

Vedere implementazione di **alarm** mediante **fork**, **signal**, **kill** e **pause**

- ❖ Purtroppo non si possono fare previsioni specifiche sull'arrivo di un segnale

➤ Per esempio, il segnale può arrivare prima che il processo entri in pausa se il sistema è molto carico

```
static void  
myHandler (int signo) {  
    ...  
}  
...  
signal (SIGALARM, myHandler)  
alarm (nSec);  
pause ();
```

Il segnale **SIGALRM** può arrivare prima della pausa

La **pause** blocca il processo per sempre visto che il segnale di allarme è stato perduto

### Race conditions: Esempio B

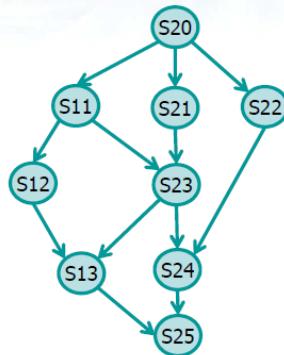
- ❖ Si supponga due processi  $P_1$  e  $P_2$  vogliano sincronizzarsi mediante l'utilizzo di segnali
- ❖ Purtroppo
  - Se il segnale di  $P_1$  ( $P_2$ ) arriva prima che  $P_2$  ( $P_1$ ) sia entrato in pause
  - Il processo  $P_2$  ( $P_1$ ) si blocca indefinitamente in attesa di un segnale

```
P1  
while (1) {  
    ...  
    kill (pidP2, SIG...);  
    pause ();  
}  
  
P2  
while (1) {  
    pause ();  
    ...  
    kill (pidP1, SIG...);  
}
```

## Esercizio

- Nonostante i loro difetti i segnali possono fornire un rozzo meccanismo di sincronizzazione

- Ipotizzando di trascurare le corse critiche** (e utilizzando fork, wait, signal, kill, e pause) realizzare il seguente grafo di precedenza



## Soluzione

```
static void
sigUsr (
    int signo
) {
    if (signo== SIGUSR1)
        printf ("SIGUSR1\n");
    else if (signo==SIGUSR2)
        printf ("SIGUSR2\n");
    else
        printf ("Signal %d\n", signo);
    return;
}
```

Definizione del gestore di segnali

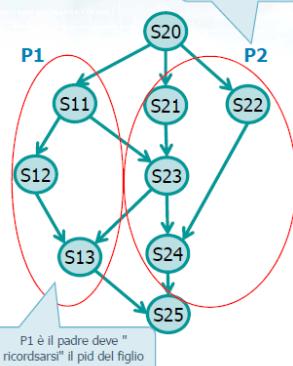
## Soluzione

```
int main (void) {
    pid_t pid;

    if (signal(SIGUSR1, sigUsr) == SIG_ERR) {
        printf ("Signal Handler Error.\n");
        return (1);
    }
    if (signal(SIGUSR2, sigUsr) == SIG_ERR) {
        printf ("Signal Handler Error.\n");
        return (1);
    }
}
```

Instanziamento del gestore di segnali per i segnali SIGUSR1 e SIGUSR2

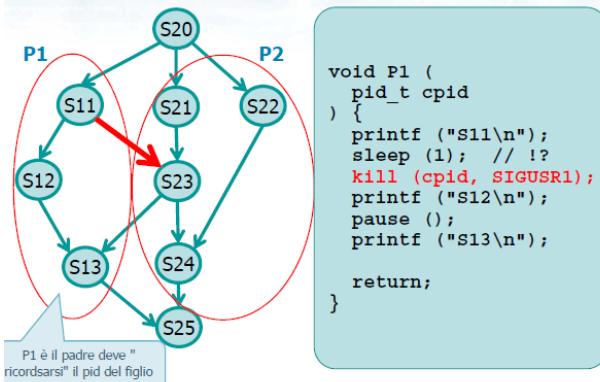
## Soluzione



P2 è il figlio può ottenere il pid del padre con getppid

```
printf ("s20\n");
pid = fork ();
if (pid > (pid_t) 0) {
    P1 (pid);
    wait ((int *) 0);
} else {
    P2 ();
    exit (0);
}
printf ("s25\n");
return (0);
```

## Soluzione

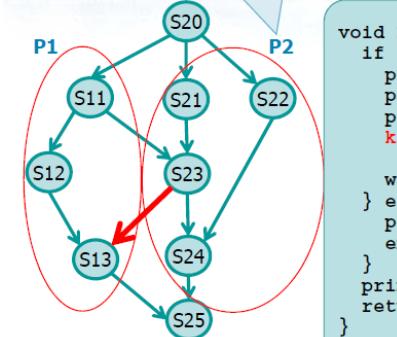


## Soluzione

```
void P1 (
    pid_t cpid
) {
    printf ("s11\n");
    sleep (1); // !?
    kill (cpid, SIGUSR1);
    printf ("s12\n");
    pause ();
    printf ("s13\n");

    return;
}
```

## Soluzione



## Soluzione

```
void P2 ( ){
    if (fork () > 0) {
        printf ("s21\n");
        pause ();
        printf ("s23\n");
        kill (getppid (), SIGUSR2);
        wait ((int *) 0);
    } else {
        printf ("s22\n");
        exit (0);
    }
    printf ("s24\n");
    return;
}
```

P1 riceve il pid del figlio (**cpid**) di S20 e invia SIGUSR1. Successivamente stampa S12 e dorme in attesa della risposta di P2 e poi stampa S13.

P2 farà un'ulteriore fork (per S21 e 22) e il padre farà S21 e andrà in pausa in attesa del SIGUSR1 di P1. Una volta ricevuto stamperà S23 e farà una kill per “rispondere” a S13 (con SIGUSR2). Se si inserisce la sleep in P2, P1 potrebbe inviare la kill prima che P2 vada in pause, non facendo funzionare il tutto. Il padre va infine in wait aspettando che il figlio termini (S22) per poi ritornare.

La return di P2 sveglierà il padre che è in wait.

## 12. Comunicazione tra processi

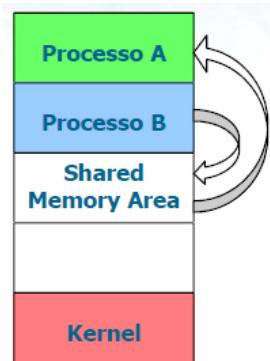
I processi concorrenti possono essere: **indipendenti** o **cooperanti**.

Un processo è indipendente se:

- Non può essere influenzato dagli altri processi
- Non può influenzare l'esecuzione di altri processi

Un processo è cooperante in caso contrario:

- La cooperazione può avvenire solo tramite lo scambio oppure la condivisione di dati
- Scambio e condivisione di dati richiedono l'implementazione di meccanismi opportuni



La condivisione di informazioni si denota spesso con il termine **IPC** (InterProcess Communication). I modelli di comunicazione principali sono basati su: **memoria condivisa** e **scambio di messaggi**.

### Memoria condivisa

Per memoria condivisa si intende un'area di memoria in cui più processi riescono a condividere dei dati.

Normalmente il SO **impedisce** a un processo di accedere alla memoria di un altro processo, perciò è molto complicata tale operazione.

L'operazione è possibile se ci si accorda sui diritti e sulle strategie di accesso e di gestione. E' possibile condividere l'area di memoria su disco, quindi due processi possono condividere la memoria su uno o più **file** scambiandosi il nome del file o il puntatore prima di effettuare una fork.

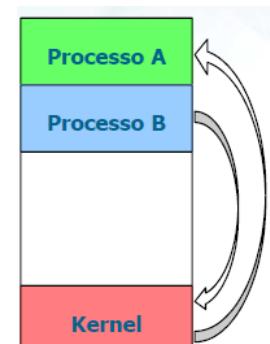
La seconda strategia è quella dei **file mappati** in memoria, cioè si riserva una quantità di memoria ed è possibile associarla ad un file pointer ed è gestita tramite le operazioni di lettura/scrittura da file e tali operazioni avvengono in memoria centrale (e non su disco).

La tecnica di memoria condivisa viene utilizzata quando si **condividono elevate quantità di dati**.

### Scambio di messaggi

Con questa tecnica la comunicazione avviene mediante lo scambio di messaggi (invio di segnali tra processi) e tale scambio di messaggi fa intervenire il kernel del SO.

- Occorre instaurare un canale di comunicazione
- Adatta allo scambio di dati in quantità ridotta
- Richiede l'utilizzo di system calls la cui chiamata necessita l'intervento del kernel
- L'intervento del kernel causa un rallentamento nei tempi di esecuzione



Un canale di comunicazione può essere caratterizzato da più parametri.

1. Il **naming**: come ci si riferisce al ricevente o al trasmittente in un certo messaggio. Normalmente la comunicazione può essere **diretta** o **indiretta** e può essere **simmetrica** o **asimmetrica**. La comunicazione è diretta se viene effettuata specificando **esplicitamente** il destinatario e il ricevente:  
**send (dest, messaggio)** e **receive (src, &messaggio)**

La comunicazione è indiretta se avviene tramite **mailbox**:

**send (mailboxAddress, messaggio)** e **receive (mailboxAddress, &messaggio)**

In entrambi i casi la comunicazione è **simmetrica** poiché viene specificato sia chi invia il messaggio che chi lo riceve mentre è **asimmetrica** quando non vengono specificati entrambi i parametri (ma soltanto uno, ad es. la posta elettronica io non devo specificare da chi ricevo le e-mail).

**2. Sincronizzazione:** tanto l'invio quanto la ricezione di messaggi può essere:

- Sincrona, i.e., bloccante (se chi fa il send si blocca finché il messaggio non è ricevuto)
- Asincrona, i.e., non bloccante (quando si può scrivere il messaggio e continuare a fare altro anche se il messaggio non è stato ancora consegnato)

**3. Capacità:** La coda utilizzata per la comunicazione può avere lunghezza di capacità:

- Pari a zero: il canale non può avere messaggi in attesa, non c'è buffering; il sender si blocca in attesa del receiver
- Limitata: il sender si blocca nel caso la coda sia piena
- Infinita: il sender non si blocca mai

## Canali di comunicazione



UNIX prevede moltissime tipologie, ma si analizzeranno soltanto: le **pipe** e i **semafori**. Servono a scopi completamente diversi.

## Le pipe 😊

Le pipe sono la più vecchia forma di comunicazione nei SO di tipo UNIX. Consistono in un **flusso di dati** tra due processi che devono avere una qualche relazione gerarchica:

- Una pipe si gestisce in maniera simile ad un file
- Ogni pipe è rappresentata tramite due descrittori interi (uno per estremo)
- Un processo **P1** scrive a una estremità e l'altro processo **P2** legge dall'altra estremità.



Storicamente una pipe è di tipo **half-duplex**, cioè i dati possono fluire in entrambe le direzioni (da P1 a P2 oppure da P2 a P1) ma **non** contemporaneamente (ma di fatto per problemi di sincronizzazione assumono modalità **simplex**, monodirezionale). Meccanismi più potenti (e.g., full-duplex) sono nati più recentemente e hanno portabilità più limitata.

La pipe può essere utilizzata per la comunicazione tra processi con un parente comune poiché i file descriptor devono essere comuni ai due processi comunicanti e quindi tali processi devono avere un antenato comune.

## System call pipe() ☺

```
#include <unistd.h>
int pipe (int fileDescr[2]);
```

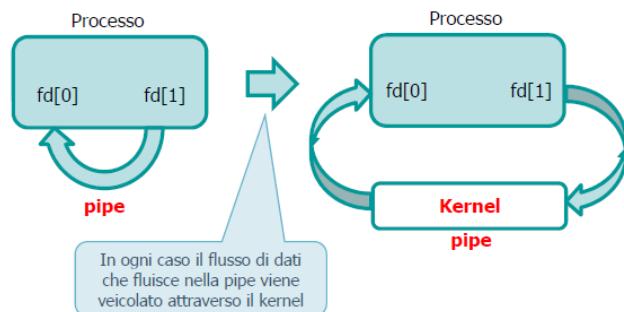
La system call pipe crea una pipe. La funzione ritorna **due** descrittori di file nel vettore **fileDescr**. In **fileDescr[0]** aperto per la **lettura** della pipe, in **fileDescr[1]** aperto per la **scrittura** sulla pipe.

L'output effettuato su **fileDescr[1]** corrisponde all'input ricevuto su **fileDescr[0]**.

Facile da ricordare mnemonicamente perché si parla di R/W quindi: read 0, write 1.

Il valore di ritorno della funzione è **0** se l'operazione ha avuto successo, **-1** altrimenti.

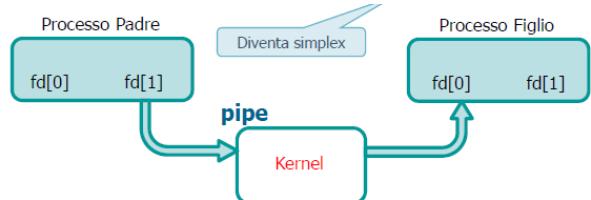
Se si effettua il comando pipe all'interno di un processo, tale operazione è **inutile**:



Perché entrambi gli estremi sono posseduti dallo stesso processo ed inoltre l'operazione non è neanche veloce perché transita dal kernel del SO.

Occorre quindi creare una pipe che metta in comunicazione due processi (padre e figlio) e questo viene ottenuto mediante **clonazione** del padre dopo aver creato la pipe.

1. Il padre crea la pipe
2. Il padre effettua una fork
3. Il figlio **eredita** i descrittori dei file
4. Uno dei due processi (es. padre) scrive nella pipe mentre l'altro legge dalla pipe.
5. Il descrittore non utilizzato viene chiuso.



### N.B. PRIMA PIPE E POI FORK SEMPRE

Il descrittore della pipe è un intero e il R/W su pipe sono simili a R/W su file. Si utilizzano le primitive **read** e **write**. Tecnicamente possono esistere più scrittori e più lettori contemporanei ma il caso standard è quello di avere un unico scrittore e un unico lettore poiché con più scrittori i dati scritti possono risultare interallacciati. Con più lettori invece risulta indeterminato chi è il lettore di un dato specifico.

La system call **read** è **bloccante** perché se un processo cerca di leggere da una pipe **vuota** si blocca in attesa che qualcuno ci scriva sopra. Se viene scritta o sono presenti una parte dei byte che si vuole leggere si va avanti e la funzione ritorna 0 se la pipe è stata chiusa all'altra estremità.

La system call **write** è **bloccante** perché si blocca se la pipe è piena e va in attesa che la pipe si liberi e qualcuno legga all'altra estremità. La dimensione della pipe dipende dalle architetture e dall'implementazione e non è certamente infinita. Generalmente il valore della costante PIPE\_BUF è **1024**. La write restituisce **SIGPIPE** se l'altra estremità è stata chiusa.

## Esempio

- ❖ Si instanzi una comunicazione di un dato tra un processo padre e un processo figlio, ovvero
  - Si crei una pipe mettendola in comune tra un processo padre e un processo figlio
  - Si trasferisca un singolo carattere dal processo padre al processo figlio
- ❖ Flusso logico
  - Creazione della pipe
  - Clonazione del processo
  - Chiusura del terminale inutilizzato della pipe
  - Operazioni di read e write ai due estremi

Sistemi Operativi - Stefano Quer

```

Esempio

if (pid == 0) {
    // Child reads
    close (file[1]);
    n = read (file[0], &cR, 1);
    printf("Read %d bytes: %c\n", n, cR);
    exit(EXIT_SUCCESS);
} else {
    // Parent writes
    close (file[0]);
    n = write (file[1], &cW, 1);
    printf ("Written %d bytes: %c\n", n, cW);
}
exit(EXIT_SUCCESS);

```

L'estremo non utilizzato viene chiuso (per prudenza)

Il figlio legge

Il padre scrive

La trasmissione di informazioni complesse richiede la gestione di un qualche tipo di protocollo di comunicazione

Lettura e scrittura sono bloccanti. Quindi i processi si sincronizzano.

## Altro esempio

Sistemi Operativi - Stefano Quer

24

## Esempio

- ❖ Le pipe hanno dimensione infinita?
  - Ovvero, qual è la dimensione di una pipe?
- ❖ Dato che la **write** è una system call bloccante è possibile comprendere la dimensione di una pipe effettuando operazioni di **write** sino a quando l'operazione si blocca

Poi si clona il processo

```

Esempio

if (fork()) {
    fprintf (stdout, "\nFather PID=%d\n", getpid());
    sleep (1);
    for (i=0; i<SIZE; i++) {
        nW = write (fd[1], &c, 1);   Il padre scrive quanto basta
        n = n + nW;
        fprintf (stdout, "W %d\r", n);
    }
} else {
    fprintf (stdout, "Child  PID=%d\n", getpid());
    sleep (10);
    for (i=0; i<SIZE; i++) {
        nR = read (fd[0], &c, 1);   Il figlio legge altrettanto
        n = n + nR;
        fprintf (stdout, "\t\t\t\tR %d\r", n);
    }
}

V = CR = Carriage Return (non a capo)

```

## Esempio

```

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

int main () {
    int n;
    int file[2];
    char cR;
    char cW = 'X';
    pid_t pid;
    if (pipe(file) == 0) {
        pid = fork ();
        if (pid == -1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }
    }
}

```

Prima si crea la pipe

Poi si clona il processo

Si dichiara il carattere da inviare con cW mentre in cR andrà il carattere letto. Si crea la pipe (che restituisce il vettore di descrittori) e si effettua la fork controllandone il corretto funzionamento. Si distingue il comportamento del padre e figlio. Visto che il figlio legge chiude l'estremità 1 di scrittura e il padre chiuderà l'estremità 0. Essendo lettura e scrittura bloccanti i processi si **sincronizzano**.

## Esempio

```

...
#define SIZE 524288
int fd[2];

int main () {
...
int i, n, nR, nW;
char c = '1';
setbuf (stdout, 0);

...
pipe(fd);
n = 0;

```

Prima si crea la pipe

## Esempio

```

> ./pgrm
Father PID=2272
Child  PID=2273
W 0
...
W 65536
...
W 65536 R 0
...
W 524288 R 524288

```

Il numero di caratteri scritti aumenta sino alla dimensione della pipe

Quando la pipe è piena la write si blocca

Dopo 10 secondi si incomincia a leggere e si incomincia a svuotare la pipe

R & W avvengono in parallelo sino a raggiungere SIZE caratteri

## Esempio

- ❖ Che cosa succede se non si rispetta la gestione half-duplex di una pipe?
- È possibile invertire le operazioni di lettura e di scrittura?
- È possibile avere lettori e/o scrittori multipli?
- ❖ In generale in risultato è indefinito ... però è possibile ottenere un risultato corretto nel primo caso ...

Il programma riceve una stringa nel parametro argv[1]

```
int fd[2];
setbuf (stdout, 0);
pipe (fd);
if (fork() != 0) {
    while (1) {
        if (strcmp(argv[1], "F") == 0 || strcmp(argv[1], "FC") == 0) {
            c = 'F';
            fprintf (stdout, "Father Write %c\n", c);
            write (fd[1], &c, 1);
        }
        sleep (2);
        if (strcmp(argv[1], "C") == 0 || strcmp(argv[1], "FC") == 0) {
            read (fd[0], &c, 1);
            fprintf (stdout, "Father Read %c\n", c);
        }
        sleep (2);
    }
    wait ((int *) 0);
}
```

Se argv[1] è "F"  
il padre scrive solo e il figlio legge solo

Se argv[1] è "C"  
il figlio scrive solo e il padre legge solo

## Esempio

```
} else {
    while (1) {
        if (strcmp(argv[1], "F") == 0 || strcmp(argv[1], "FC") == 0) {
            read (fd[0], &c, 1);
            fprintf (stdout, "Child Read %c\n", c);
        }
        sleep (2);
        if (strcmp(argv[1], "C") == 0 || strcmp(argv[1], "FC") == 0) {
            c = 'C';
            fprintf (stdout, "Child Write %c\n", c);
            write (fd[1], &c, 1);
        }
        sleep (2);
    }
    exit (0);
}
```

Se argv[1] è "FC"  
padre e figlio scrivono alternandosi

Only the father writes ...OK

Only the child writes ...OK

Father and child alternate writing every 2 seconds...OK

But how do they alternate in real cases?

```
> ./pgrm F
Father Write F
Child Read F
...
^C
> ./pgrm C
Child Write C
Father Read C
...
^C
> ./pgrm FC
Father Write F
Child Read F
Child Write C
Father Read C
...
^C
```

Se entrambi devono leggere e scrivere (caso FC in cui si entra in entrambi gli IF) tutto funziona se i processi sono sincronizzati perché il figlio legge, aspetta 2 sec, scrive, aspetta 2 sec e il padre fa la stessa cosa ma alternata: scrive, aspetta 2 sec, legge, aspetta 2 sec.

## 13. Pipe e ridirezione in UNIX/Linux ☺



La comunicazione tra processi può essere realizzata anche quando i processi sono eseguiti mediante comandi di shell. Il collegamento tra standard output e standard input si chiama **pipe** e crea in memoria un canale diretto tra i due processi.

Comando ; : permette di eseguire più istruzioni in sequenza. Ad esempio, ls ; ls permette la stampa a video due volte del comando ls.

Comando | : comando di **pipe**. Esempio: ls | more crea una pipe. Ls scrive sulla pipe e viene letto dal comando more. Esempio: ls -R | cat | more vengono eseguiti in pipe.

Esempio: ls -R | sort ordina ciò che viene inviato in pipe.

Comando > : effettua la **ridirezione** di un output. Esempio: ls -la > a salva il contenuto di ls in un file chiamato a.

Comando >> : effettua l'**append**, cioè non cancella il contenuto di quello che c'era prima nella destinazione. Esempio: ls 1> a 2> b. Lo stdout va in a mentre lo stderr va in b. (b sarà vuoto perché non genera errori il comando).

Comando < : effettua la redirezione su input. Molto inutile. Esempio: cat < file cat prende l'input da file.

Comando << : esempio: cat << EOF riceve un input immediato fino al tappo (EOF o qualsiasi stringa).

## 14. Espressioni regolari e comando find

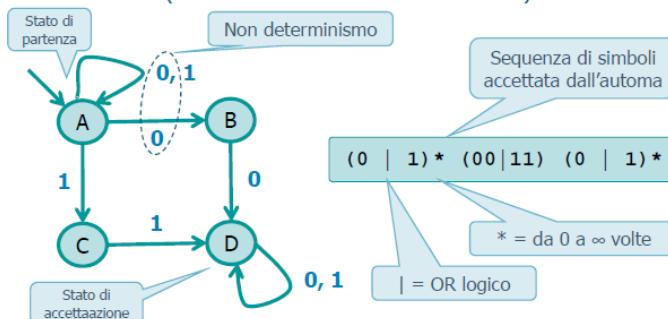
Le espressioni regolari sono state standardizzate da POSIX nel 1992. Esistono varie versioni molti simili ma non identici:

- **BRE, Basic Regular Expression**
- **ERE, Extended Regular Expression**
- **PCRE, Perl Compatible Regular Expression**

Un'espressione regolare, anche detta **pattern**, è un'espressione utilizzata per specificare un insieme di stringhe, in altre parole sono operatori compatti utilizzati per rappresentare sequenze complesse. Ad esempio:  $a \mid b^*$  indica l'insieme delle stringhe che hanno una "a" oppure una " $b^*$ " dove \* indica da 0 a più presenze di b {a, vuoto, b, bb, bbb, bbbb,...}.

Le espressioni sono utilizzate per effettuare l'accoppiamento (**match**) tra oggetti. Viene spesso utilizzato quando bisogna analizzare file, righe, programmi e riconoscere i vari tipi (ad esempio l'indentazione automatica di un file .c).

Una espressione regolare corrisponde a un **Automa Non Deterministico (NFA)**



A sinistra l'automa non deterministico inizia il suo comportamento nello stato A e ci si evolve se alla fine della stringa ci si ritrova in D nello stato di accettazione.

Nelle espressioni regolari si utilizza una terminologia complessa fatta da:

- **Letterale**: qualsiasi carattere utilizzato nella ricerca del match. Ad esempio: **ind** in windows, **indifferent**, etc.. (cerco qualsiasi occorrenza di ind in qualsiasi punto).
- **Metacarattere**: da uno a più caratteri con significato speciale. Ad esempio: \* indica da 0 a infiniti simboli precedenti.
- **Sequenza di escape**: metodo per indicare che un metacarattere deve essere utilizzato come letterale. Ad esempio il carattere '.' Si indica con '\.'. Il '\' indica che ciò che segue non è più un metacarattere.

Metacaratteri	
Operatore	Significato
[...]	Specifica un elenco o un intervallo di simboli
(...)	Gestisce la precedenza tra operatori Raggruppa insiemi di simboli in sottoespressioni Permette riferimenti a espressioni precedenti (backward reference)
	Effettua l'OR tra espressioni regolari Basic RE: [...] e (...)
Ancore	Significato
\<	Inizio parola
\>	Fine parola
^	Inizio riga
\$	Fine riga
Caratteri speciali	Significato
\+ \? \.	Caratteri '+', '?', '/'
\n	New line
\t	Tabulazione

Metacaratteri	
Caratteri	Significato
c	Un qualsiasi simbolo c (Tranne quelli utilizzati a scopi speciali)
.	Un carattere qualsiasi (non '\n')
\s	Uno spazio o una tabulazione
\d	Una cifra 0-9
\D	Non una cifra
\w	qualsiasi carattere tra 0-9, A-Z, a-z
\W	qualsiasi carattere non in 0-9, A-Z, a-z

Sovra-insieme  
(non tutti sono disponibili per tutti i comandi)

Quantificatori e Intervalli	Significato
*	Elemento presente [0, ∞] volte
+	Elemento presente [1, ∞]
?	Elemento presente [0, 1] volte
[c <sub>1</sub> c <sub>2</sub> c <sub>3</sub> ]	Uno qualsiasi dei caratteri in parentesi
[c <sub>1</sub> -c <sub>2</sub> ]	Uno qualsiasi dei caratteri nel range
[^c <sub>1</sub> -c <sub>2</sub> ]	Uno qualsiasi dei caratteri non nel range
{n}	Elemento presente esattamente n volte
{n <sub>1</sub> ,n <sub>2</sub> }	Elemento presente da n <sub>1</sub> a n <sub>2</sub> volte

Sovra-insieme

Comando grep:  
ammette anche le versioni {n<sub>1</sub>} ovvero {,n<sub>2</sub>}

## Esempi

### Esempi

RE	Significato
ABCDEF	La stringa "ABCDEF"
a*b	Un qualunque numero di a seguite da una b
ab?	Solo a oppure ab
a{5,15}	Da 5 a 15 ripetizioni della lettera "a"
(fred){3,9}	Da 3 a 9 ripetizioni della stringa "fred"
.+	qualsiasi sequenza (non vuota)
myfunc.*(.*)	Una funzione il cui nome inizia per "myfunc"
^ABC.*	Una riga che inizia con "ABC"
.*h\$	Una riga che finisce con "h"
hello\>	Una parola che termina con "hello"
a+b+	Una o più a seguite da una o più b

### Esempi

RE	Significato
[a-zA-Z0-9]	Una lettera o una cifra
A b	A oppure b
\w{8}	Sequenza di 8 caratteri alfabetici o numerici (minuscoli o maiuscoli)
((4\.[0-2]) (2\.[1-3]))	Numeri 4.0, 4.1, 4.2 oppure 2.1, 2.2, 2.3
(.)\1	Due caratteri identici
(.).\1	qualsiasi stringa palindroma di 5 caratteri (e.g., radar, civic, 12321, etc.)

Basic RE: \(.)\1 e \(.)\1\2\\  
Extended RE: \(.)\1 e \(.)\1\2\1

### Esercizio

- ❖ Scrivere una espressione regolare per rintracciare
  - Una qualsiasi data con formato gg/mm/aaaa
    - Giorno e mese possono essere espressi su 1 o 2 cifre

\d{1,2}\\\d{1,2}\\\d{4}

- Tutte le righe che contengono un solo numero intero incluso tra 1 e 50 (inclusi)

(^[1-9]{1}\$|^1[0-4]{1}[0-9]{1}\$|^50\$)

ABCDEF: sequenza letterale standard.

a\*b: indica da 0 a infinite presente di a ed 1 sola b.

?" indica 0 oppure 1.

a{5,15} deve comparire tra 5 e 15 volte.

(fred){3,9} la "fred" deve comparire tra 3 e 9 volte.

il "+" è una variante di \* da 1 a infinite volte del punto. Il punto indica un qualsiasi carattere.

myfunc.\*(.\*) primo tentativo rozzo di trovare i prototipi di funzioni il cui nome inizia con qualcosa, seguito da myfunc e seguito da qualcosa.

^ indica l'inizio della riga, voglio ABC a inizio riga seguito da qualsiasi cosa.

.\*h\$ voglio una riga che termini con h.

hello\&gt; voglio una parola che termini con hello.

a+b+ una o più a seguite da una o più b.

[a-zA-Z0-9] indica tutti i caratteri maiuscoli/minuscoli e le cifre.

A|b indica A or b.

\w{8} sequenza di 8 caratteri alfabetici o numerici (insensitive case).

4 seguito da sequenza di escape (ovvero il punto).

Il "\" può essere usato per ripetere il match. Il punto è un qualsiasi carattere, ma con \1 vuol dire ripeterlo un'altra volta quindi due caratteri.

Nell'ultimo esempio sui numeri da 1 a 50:

deve essere presente a inizio riga [1-9] che compare una volta sola, oppure sempre a inizio riga una cifra da 1 a 4 seguita da una cifra da 0 a 9 oppure 50.

## Comando find

Il comando **find** permette di ricercare file, direttori o link che soddisfano (match) un particolare criterio, creandone un elenco. Se necessario, esegue anche sugli oggetti dell'elenco dei comandi di shell.

Il comando ritorna il path (relativo) degli oggetti rintracciati e non (solamente) il loro nome.

Questo è importante per la scrittura delle espressioni regolari di ricerca e delle azioni da effettuare.

Il comando **find** è composto da:

**find directory options actions**

Sostanzialmente visita tutto l'albero a partire dal directory **directory** creando l'elenco che soddisfa le **options** ed eventualmente effettua per ogni file l'azione contenuta in **actions**.

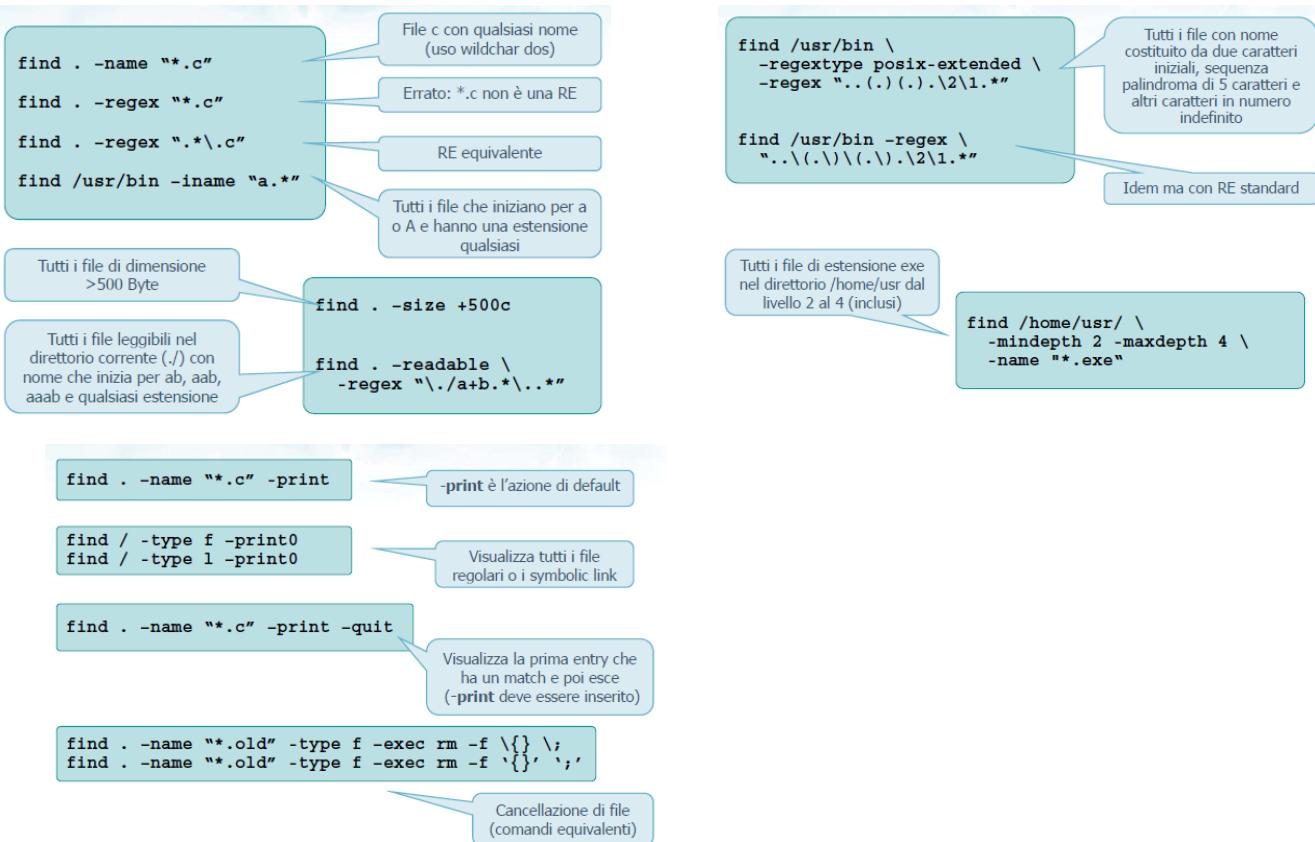
Il primo parametro è molto semplice poiché viene semplicemente indicata la directory dove operare, ad esempio:

- .
- /usr/bin
- ./subDirA/subDirB

Il parametro **options** è invece molto ampio, poiché è possibile agire su numerose cose. Quelle utili sono le seguenti:

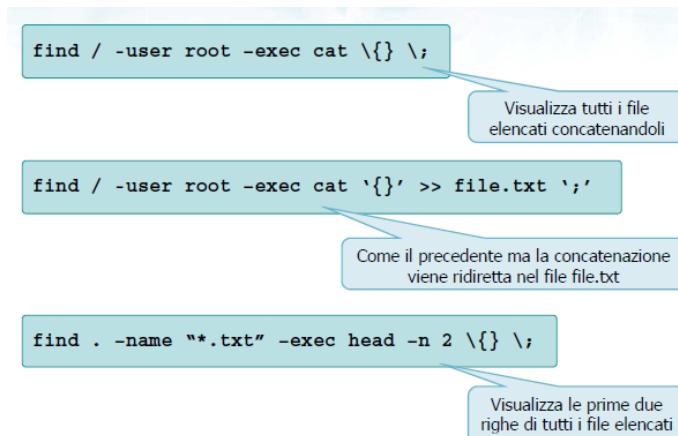
- 
- **find [<directory>] [<options>] [<actions>]**
    - Search for files in a directory hierarchy (with a specified root, i.e., <directory>).
    - Options:
      - -name <pattern>: search files whose name matches the pattern.
      - -regex <pattern>: search files whose path matches a regular expression.
      - -regextype posix-extended: specify posix-extended format for regular expressions
      - -type <f|l|d>: search files of a specific type.
      - -mindepth <depth>: search files starting from the specified directory tree depth.
      - -maxdepth <depth>: search files up to the specified directory tree depth.
      - -size <[+,-]n[cwkMG]>: search files whose size starts from (+) or goes up to (-) the specified size. (c=bytes, w=words, k=kilobytes, M=megabytes, G=gigabytes).
      - -user uname: File is owned by user uname (numeric user ID allowed).
      - -group gname: File belongs to group gname (numeric group ID allowed).
    - Actions:
      - -exec <cmd>: execute command on each matched file.
        - \{} (or '{}') can be used as a placeholder for the file path.
        - The command must end with \; (or ';' ).

-name differisce da -regex perché il primo non utilizza le regexp. Ad esempio in -name si può utilizzare .\* per indicare qualsiasi estensione file, mentre nel secondo caso il .\* ha un significato diverso.



L'azione di default del comando `find` è la stampa che equivale al comando `print`:

#### `find directory options -print`



È però possibile eseguire un qualsiasi comando di shell su ciascuno degli oggetti appartenenti all'elenco restituito.

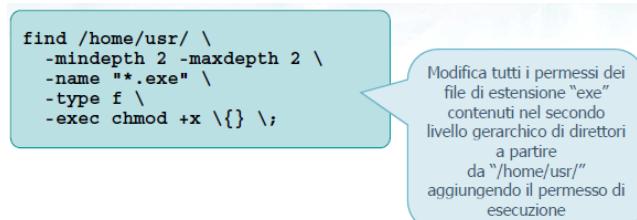
In generale si specifica l'esecuzione di un comando tramite l'opzione `exec` che è la versione sicura di `execdir`. Tale comando deve agire su qualcosa che va specificato tramite delle espressioni specifiche:

`find directory options -exec comando '{}';`

`find directory options -exec comando '\{}';`

Dove il comando viene eseguito nel directory

in cui si esegue la `find` con `exec` e la `find` sostituisce la stringa '`{}`' (`\{}) con il file corrente dell'elenco e la stringa ';' (\;) termina il comando eseguito dalla find.`



## Soluzione

- ❖ Si riportino i comandi UNIX per effettuare quanto indicato, utilizzando eventuali ridirezioni e pipe
  - Visualizzare quanti file di estensione ".txt" sono presenti nell'albero con radice la working directory
  - Visualizzare quante righe sono presenti in tutti i file di estensione ".txt"

```
find . -name "*.txt" | wc  
find . -name "*.txt" -exec wc \{\} \;
```

Pipe: applica wc (word count) all'elenco trovato dalla find (elenco di tutti i file con estensione .txt)

-exec: applica wc (word count) al contenuto di tutti i file (\{\}) rintracciati con la find

## 15. I filtri

In UNIX/Linux un filtro è un comando che riceve il proprio input da standard input e lo manipola (lo filtra) secondo determinati parametri e opzioni. Produce il suo output su standard output.

Sostanzialmente sono comandi che permettono un qualche tipo di manipolazione di testi. I filtri più comuni sono: awk, cat, cut, compress, grep, head, perl, sed, sort, tail, tr, uniq, wc. Alcuni di questi comandi sono discretamente complessi come **grep**, **sort** mentre altri sono linguaggi di scripting completi come **sed**, **awk**. I filtri spesso utilizzano espressioni regolari come comandi in pipe (ovvero utilizzati con l'operatore |) con altri.

### cut

Cut permette di effettuare la rimozione di specifiche sezioni dal file. Il formato è **cut [options] file**

È possibile effettuare la rimozione di determinati campi di un file con l'opzione **-f LIST** (--fields=LIST) oppure di caratteri con **-c LIST**. In generale si specifica anche il delimitatore dei campi con **-d DELIM** (--delimiter=DELIM).

**cut -f 1,3 file.txt** (seleziona i campi 1 e 3 di tutte le righe del file)

**cut -f 1-3, 5-6 -d " " foo.txt** (seleziona i campi da 1 a 3 e da 5 a 6 del file, specificando che i campi sono delimitati da spazio)

### tr

tr copia lo standard input nello standard output effettuando le sostituzioni oppure le cancellazioni specificate. Va utilizzato ridirigendo il suo input con l'output di altri comandi. Il formato è il seguente:

**tr [options] set<sub>1</sub> [set<sub>2</sub>]**

tra le opzioni principali c'è il **--complement (-c)** che utilizza il complemento del set<sub>1</sub> oppure il **--delete, -d** che cancella i caratteri indicati nel set<sub>1</sub>.

Sistemi Operativi - Stefano Squarci

## Esempi

- All'interno di set<sub>1</sub> e set<sub>2</sub>:
  - \num = carattere di codice ASCII num
  - \n = newline
  - \\ = backslash

```
tr -d abcd < file.txt  
cat file.txt | tr ab BA  
echo ciao | tr ia IA
```

Visualizza su standard output le righe di file.txt in cui sono stati eliminati i caratteri a, b, c, d

Visualizza su standard output le righe di file.txt in cui 'a' è stato sostituito con 'B' e 'b' con 'A'

Visualizza ciao su standard output

## Uniq

Riporta oppure elimina le righe ripetute nel file in ingresso. Il formato è **uniq [options] [inFile] [outFile]**.

Questo comando richiede che il file sia **ordinato**. Senza opzioni elimina le righe ripetute.

Alcune opzioni sono:

- **--count, -c** (Stampa il numero di ripetizioni prima della riga)
- **-repeated, -d** (Visualizza solo le righe ripetute)
- **--skip-fields=N, -f N** (Ignora i primi N campi per il confronto)
- **--ignore-case, -l** (Case insensitive)

## Basename

Elimina il directory (path) e suffisso (estensione) da un nome di file. Il formato è **basename nome [estensione]**.

Serve per sbarazzarsi del path oppure sostituire estensioni o cambiare path.

## Sort

```
> basename /home/quer/current/file.txt  
file.txt  
  
> basename /home/quer/current/file.txt ".txt"  
file  
  
> basename /home/quer/current/file.txt .txt  
file  
  
> basename /home/quer/current/file.txt txt  
file.
```

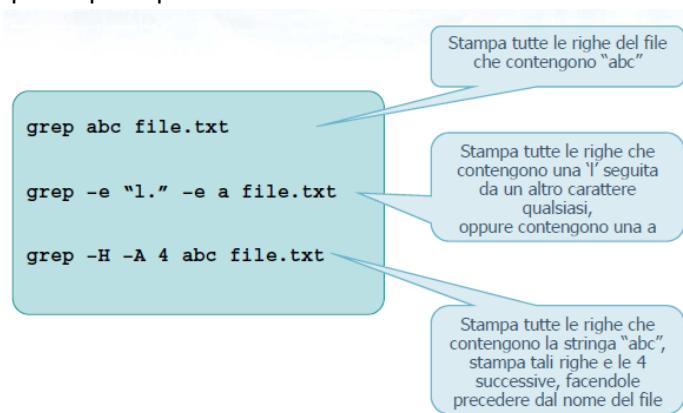
Effettua l'ordinamento di un file (può anche eseguire la fusione tramite il merge). Il formato è **sort [options] [file]**. Le opzioni principali sono:

- **--ignore-leading-blanks, -b** (Ignora gli spazi iniziali)
- **--dictionary-order, -d** (Considera solo spazi e caratteri alfabetici)
- **--ignore-case, -f** (Trasforma caratteri minuscoli in maiuscoli, case insensitive)
- **--numeric-sort, -n** (Confronta utilizzando un ordine numerico)
- **--reverse, -r** (Ordine inverso)
- **--key=c1[,c2], -k c1[,c2]** (Ordina sulla base dei soli campi selezionati)
- **--merge, -m** (Merge file già ordinati (senza riordinare))
- **--output=f, -o=f** (Scrive l'output nel file f invece che su standard output)

## 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 standard output. Il comando esiste in diverse versioni: **grep** (versione standard) oppure **egrep**, **fgrep**, **rgrep** dove egrep equivale a "grep -E" e usano Extended RE nel pattern. Il formato è **grep [options] pattern [file]** e tra le opzioni principali troviamo:

- **--regexp= PATTERN, -e PATTERN**  
(Specifica i pattern da ricercare, permette di specificare pattern multipli)
- **--after-context=N, -A N** (Dopo ciascun match stampa ancora N righe (oltre alla riga in cui si è trovato il match))
- **--with-filename, -H** (Stampa il nome del file per ogni match)
- **--ignore-case, -l** (Case insensitive)



- --line-number, -n (stampa il numero di riga del match)
- --recursive, -r, -R (procede in maniera ricorsiva sul sottoalbero)
- --inverse-match, -v (Stampa solo le righe che non fanno match)

### Soluzione

- ❖ Si riporti il comando UNIX per effettuare quanto indicato, utilizzando eventuali ridirezioni e pipe
- Visualizzare tutti i file del directory corrente ordinando le righe per ora di creazione decrescente

```
total 28
drwxr-xr-x 1 quer quer 512 Nov 12 10:17 .
drwxr-xr-x 1 root root 512 Sep 26 16:08 ..
-rw-r--r-- 1 quer quer 1669 Oct  8 22:23 .bash_history
-rw-r--r-- 1 quer quer 220 Sep 26 16:08 .bash_logout
...
ls -la | \
grep -v -e "total" -e "\$" -e "\$\$" | \
sort -n -r -k 8
```

Output di ls -la  
Già incluso nel precedente  
Elimina i direttori ".", ".." e la riga "total"

### Soluzione

- ❖ Si riporti il comando UNIX per effettuare quanto indicato, utilizzando eventuali ridirezioni e pipe

- Visualizzare tutte le righe dei file di estensione ".txt" che contengono una stringa palindroma
- Di 3 caratteri (e.g. "aba")
  - Di 5 caratteri (e.g. "abcba")

```
grep -e "(.).\1" *.txt
grep -e "(.)\(.).\2\1" *.txt
```

Basic Reg Exp  
Extended Reg Exp (per 5 caratteri)

```
grep -extended-regexp -e "(.).\1" *.txt
grep -E -e "(.)\(.).\2\1" *.txt
```

Esame del 29.06.2015

### Soluzione

- ❖ Si riportino i comandi UNIX per effettuare quanto indicato, utilizzando eventuali ridirezioni e pipe
- Nel directory "/home/foo" cercare i file con il nome che inizia con il carattere "L" e estensione "txt". In questi file ricercare la presenza della stringa "laib". Visualizzare il nome del file e l'intera riga in cui tale stringa viene rintracciata.

```
find /home/foo -name "L*.txt" -exec \
grep -H "laib" '{}' \;
```

### Soluzione

- ❖ Si riportino i comandi UNIX per effettuare quanto indicato, utilizzando eventuali ridirezioni e pipe

- Trovare tutti i file di estensione "txt" nel directory "/home" memorizzati tra il livello di profondità 3 (incluso) e il livello di profondità 5 (incluso) dell'albero dei direttori e che siano leggibili. Di questi modificare il proprietario in "ugo".

```
find /home -mindepth 3 -maxdepth 5 \
-name "*.txt" -readable \
-exec chown "ugo" '{}' \;
```

Esame del 29.06.2015

### Soluzione

- ❖ Si riportino i comandi UNIX per effettuare quanto indicato, utilizzando eventuali ridirezioni e pipe
- Per ogni file di estensione "txt" presente nel directory corrente ricavare il nome e il numero di caratteri presenti nel file. L'elenco venga ordinato in base al numero di caratteri in ordine numerico inverso e memorizzato nel file "stat.txt".

```
find . -name "*.txt" \
-exec wc -c '{}' \; | sort -rn -k 1,1 > stat.txt
```

Oppure -k 1

### Soluzione

- ❖ Si riportino i comandi UNIX per effettuare quanto indicato, utilizzando eventuali ridirezioni e pipe

- Un'applicazione C è formata da main.c, f1.c, f2.c e main.h. Scrivere un Makefile con due target.

- Il primo sia in grado di compilare l'applicazione denominando l'eseguibile myapp.
- Il secondo rimuova eventuali file temporanei e sposti l'eseguibile nel directory "/user/bin"

```
<tab>
compile: main.c f1.c f2.c
          gcc -o myapp main.c f1.c f2.c
install:
          rm *.tmp
          cp myapp /user/bin
```

## 16. I thread ☺

I thread nascono per lo stesso motivo per cui sono nati i processi: era utile avere in parallelo l'esecuzione di più tipi di codice, ma anche per altri problemi. Il primo è relativo ai costi di creazione di un processo che sono relativamente elevati ed il secondo è per avere **memoria condivisa** (i processi non condividono quasi nulla). Un processo può eseguire altri processi attraverso: la clonazione (**fork**), sostituzione del processo attuale con un altro programma (**exec**) oppure chiamata esplicita (windows **CreateProcess**).

Ogni processo ha un **proprio** spazio di indirizzamento ed ha una **singola traccia** di esecuzione.

I problemi di trasferimento dati sono quindi elevati in caso di processi cooperanti (poiché non condividono informazioni). La clonazione implica inoltre un aumento sensibile della memoria utilizzata e tempi di creazione elevati.

Infine, la gestione dei processi multipli richiede uno scheduling complesso e delle operazioni di context-switching costose. I processi vengono detti **heavyweight process** (task con un solo thread, ovvero singola traccia di esecuzione).

Può essere utile gestire questi thread proprio per avere costi di gestione più bassi, un unico spazio di indirizzamento e tracce di esecuzione multiple.

I **thread** sono stati introdotti dallo standard POSIX e definiti come:

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

Processo: unità che raggruppa risorse

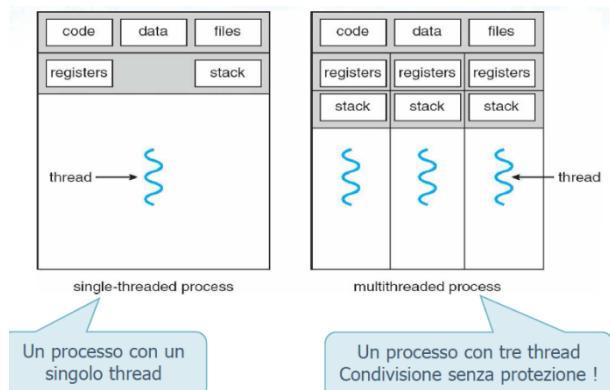
Thread: unità di schedulazione della CPU (processo leggero, **lightweight process**)

I dati che vengono condivisi nei thread sono:

- Sezione di codice
- Sezione di dati (variabili, descrittori, etc..)
- Risorse del sistema operativo (es. segnali)

I dati che rimangono privati sono invece:

- Program counter e registri hardware (visto che più thread eseguiranno linee diverse di codice)
- Stack, ovvero, variabili locali e storia dell'esecuzione (visto che un thread implica un flusso di esecuzione a se stante all'interno del processo stesso).



### Vantaggi ☺

L'utilizzo dei thread consente **tempi di risposta ridotti e risorse condivise**. Creare un thread è 10-100 volte più veloce rispetto a generare un processo e nei thread la condivisione delle risorse è automatica (rispetto ai processi in cui non lo è).

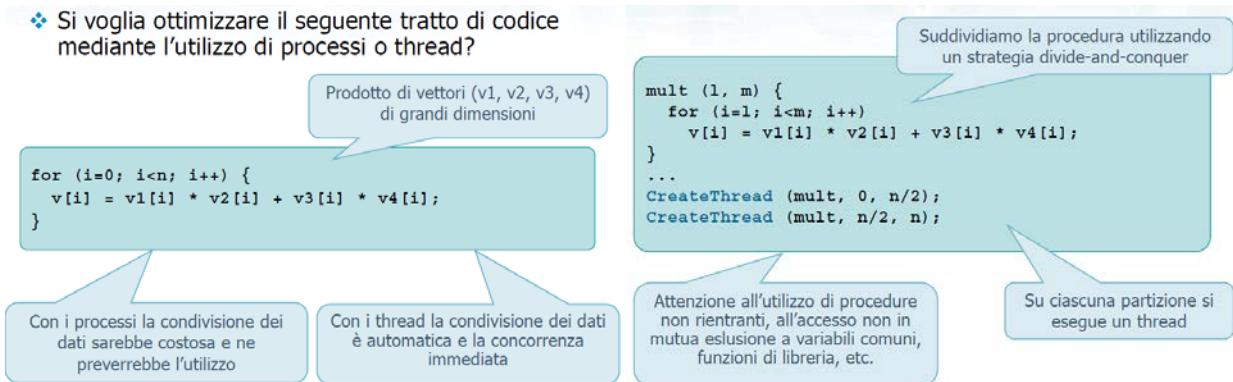
Si hanno anche **costi minori per la gestione delle risorse** visto che un'unica sezione di codice e/o dati può servire più clienti, ed una **maggior scalabilità** poiché i vantaggi della programmazione multi-thread aumentano nei sistemi multi-processore.

### Svantaggi ☹

Non esiste protezione tra thread perché tutti sono eseguiti nello stesso spazio degli indirizzi e se i thread **non** sono sincronizzati l'accesso a dati comuni non è **thread safe**. Nei thread **non esiste relazione gerarchica** padre-figlia anche se viene restituito l'identificatore del thread.

## Esempio

- ❖ Si voglia ottimizzare il seguente tratto di codice mediante l'utilizzo di processi o thread?

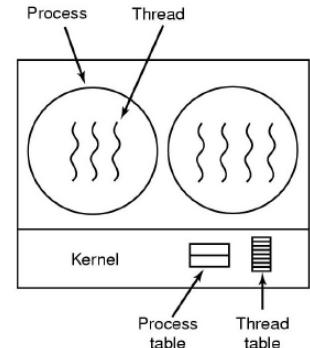


## Modelli di programmazione multi-thread ☺

- Kernel-level thread: implementazione dei thread a livello kernel
- User-level thread: implementazione dei thread a livello utente (nello spazio utente)
- Soluzione mista o ibrida: implementazione dei thread a livello sia user sia kernel

### Kernel-level thread ☺

I thread con questo modello vengono gestiti a livello kernel. Il sistema operativo manipola tanto processi quanto thread ed è a conoscenza dell'esistenza dei thread. Fornisce un supporto adeguato per la loro manipolazione e tutte le operazioni sui thread (creazione, sincronizzazione, etc.) sono effettuate mediante **system call**. Il sistema operativo mantiene per ciascun thread delle informazioni simili a quelle che mantiene per ogni processo: tabella dei thread, Thread Control Block (TCB) per ogni thread attivo. Tali informazioni gestite sono "globali" all'interno dell'intero sistema operativo.



I vantaggi risiedono nel fatto che il sistema operativo è a **conoscenza** di tutte le informazioni sui thread ed è possibile scegliere quale processo schedulare ed eventualmente allocare più tempo di CPU a processi con molti thread rispetto a quelli con pochi thread.

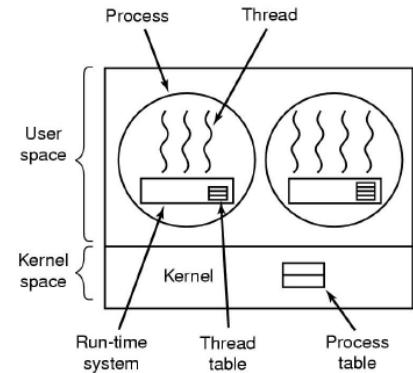
Un altro vantaggio è anche quello di avere efficacia nelle applicazioni che si bloccano spesso (e.g., read bloccante): i thread ready possono essere schedulati anche se appartengono allo stesso task di un thread che ha chiamato una system call bloccante, cioè se un thread si blocca è sempre possibile eseguirne un altro nello stesso processo o in un altro perché il sistema operativo controlla tutti i thread di tutti i processi. E' evidente che questa opzione permette un effettivo parallelismo ed in un sistema multiprocessore si possono eseguire thread multipli.

Gli svantaggi risiedono nel **passaggio al modo kernel** la cui gestione è relativamente lenta e inefficiente. Il context switch è dunque costoso e vi è una limitazione sul numero massimo di thread (bloccandone la nascita di nuovi). Infine, le informazioni sono costosi (a causa della tabella dei thread e del TCB).

## User-level thread ☺

Il pacchetto dei thread è inserito completamente nello spazio dell'utente perciò il kernel non è a conoscenza dei thread e gestisce solo i processi. I thread sono gestiti run-time tramite una libreria e ciò permette il supporto mediante un insieme di chiamate a funzioni a livello utente. Creare un thread, sincronizzare thread, schedularli, etc., non richiede l'intervento del kernel poiché si utilizzano funzioni non system call.

Ogni processo ha bisogno di una tabella personale dei thread in esecuzione e le informazioni necessarie sono minori che nel caso di gestione kernel-level: TCB di dimensioni ridotte, visibilità locale delle informazioni (all'interno del processo).



**Vantaggi:** si possono implementare in tutti i kernel, anche nei sistemi che non supportano thread in maniera nativa e non richiedono modifiche al sistema operativo: gestione efficiente, context switch veloce, manipolazione efficiente. Sono anche centinaia di volte più veloci dei thread kernel ed al programmatore è consentito generare tutti i thread desiderati. Al limite potrebbe essere possibile pensare a scheduling/gestioni personalizzate dei thread all'interno di ciascun processo (da parametri diversi a ciascun thread).

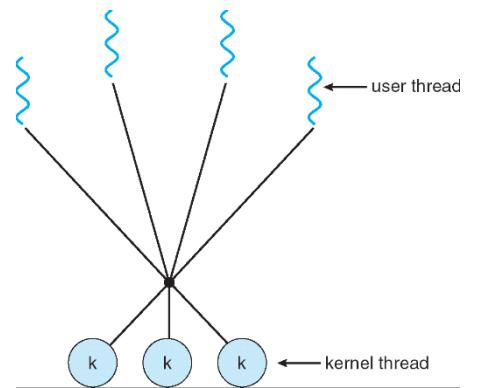
**Svantaggi:** il sistema operativo non sa che esistono i thread perciò possono essere fatte scelte inopportune oppure poco efficienti: Il sistema operativo potrebbe schedulare un processo il cui thread in esecuzione potrebbe fare una operazione bloccante. In questo caso l'intero processo potrebbe rimanere bloccato anche se al suo interno diversi altri thread potrebbero essere eseguiti. Occorrerebbe comunicare informazioni tra kernel e manager utente run-time e senza questo meccanismo di comunicazione esisterebbe solo un thread in run per task anche in un sistema multiprocessore. Non esiste scheduling all'interno di un processo singolo, ovvero non esistono interrupt all'interno di un singolo processo. Se un thread in esecuzione non rilascia la CPU non è bloccabile.

## Implementazione ibrida n:m ☺

Uno dei problemi della programmazione multithread è quella di definire il rapporto tra thread utente e thread kernel. Praticamente tutti i sistemi operativi moderni dispongono di kernel thread.

L'idea base è di avere **m** thread utente e di accoppiarli a **n** thread kernel (in generale  $n < m$ ).

L'implementazione mista tenta di combinare i vantaggi di entrambi gli approcci: l'utente decide quanti thread utente eseguire e su quanti thread kernel mapparli (fatto dalla libreria). Il kernel è a conoscenza solo dei thread kernel e gestisce solo tali thread. Ogni thread kernel può essere utilizzato a turno da diversi thread utente (ovvero possono essere rimappati).



## Coesistenza di processi e thread

Data la coesistenza di processi e thread nascono problematiche di vario genere (molte delle quali non sono risolte in maniera univoca). Ad esempio: una **fork** duplica solo il thread che effettua la chiamata oppure tutti i thread del processo? (Indefinito: dipende dalla libreria). Una **exec** sostituisce solo il thread chiamante con il nuovo processo o tutti i thread? Se un processo riceve un segnale quale thread se ne fa carico?

## 17. La libreria Pthread

Le librerie di thread forniscono l'interfaccia per effettuare la gestione dei thread da parte del programmatore. Le più utilizzate sono le librerie **thread POSIX** (livello utente e kernel), **Windows 32/64** (realizzata a livello kernel).

**Java** ad esempio è implementata tramite una libreria thread del sistema operativo ospitante.

La libreria POSIX threads viene denominata **Pthreads**. È definita in C ma è disponibile in altri linguaggi. Attraverso questa libreria il thread è una **funzione** che viene eseguita in maniera indipendente dal resto del programma. Questo forma un processo concorrente costituito da più thread (insieme di funzioni) in esecuzione indipendente che condividono le risorse del processo.

Pthreads permette di: creare e manipolare thread, sincronizzare thread, proteggere le risorse comuni ai thread, schedulare thread, distruggere thread. Definisce più di 60 funzioni di gestione.

Tutte le funzioni hanno nome `pthread_*`:

- **pthread\_equal** che permette di determinare se due thread sono lo stesso thread
- **pthread\_self** che permette ad un thread di capire chi è (e capire qual è il proprio TID)
- **pthread\_create** equivalente alla fork
- **pthread\_exit** permette di terminare un thread
- **pthread\_join** equivalente di wait/waitpid
- **pthread\_cancel** che non ha equivalente a livello di processo
- **pthread\_detach** permette di staccare un thread affinché non sia più joinable (nessuno può più aspettarli)

La libreria Pthread si trova in **pthreads.h** e va inserito nel file .c come `#include <pthread.h>`. Inoltre, durante la compilazione va inclusa tale libreria:

```
gcc -Wall -g -o <exeName> -pthread <file.c>
gcc -Wall -g -o <exeName> <file.c> .pthread
```

### Thread Identifier

Ciascun thread possiede un identificatore. Nei processi era detto PID (pid\_t) mentre nei thread è definito come **pthread\_t**. La differenza risiede nel fatto che il tipo pthread\_t è **opaco** (struttura di cui non conosciamo i campi) perciò è necessario utilizzare delle funzioni per manipolarlo.

Inoltre il significato di un TID ha significato solo all'interno di quel processo (mentre il PID è globale).

#### System call pthread\_equal ()

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

La funzione riceve due TID e restituisce un valore pari a 0 se i due sono uguali oppure diverso da 0 se sono diversi.

#### System call pthread\_self ()

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

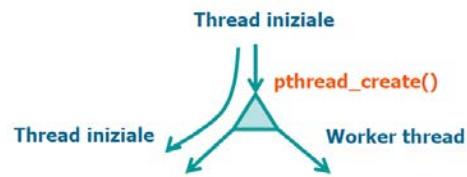
dati personali).

Un thread che vuole conoscere il proprio ThreadID chiama questa funzione e può essere utilizzata insieme a **pthread\_equal** per **autoidentificarsi** (importante per accedere correttamente ai propri

## System call pthread\_create()

E' l'equivalente della **fork** con l'unica differenza che colui che procede a sinistra è detto **thread iniziale**, mentre quello a destra è detto **worker thread** (anche se sono entrambi definibili così perché sono uguali).

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



Il valore di ritorno è un intero per identificare se la funzione ha avuto successo (0) o meno. I parametri:

- Il parametro **pthread\_t \*tid** è il puntatore al thread\_id generato
- Il parametro **const pthread\_attr\_t \*attr** sarà quasi sempre NULL
- Il parametro **void \*(\*startRoutine)(void \*)** è la **funzione da eseguire**
- Il parametro **void \*arg** è il puntatore all'oggetto che la funzione deve ricevere

## System call pthread\_exit()

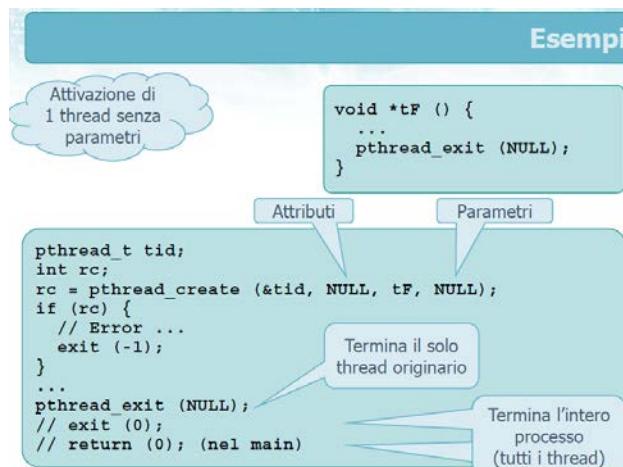
Un intero processo (con tutti i suoi thread) termina se un suo thread effettua una **exit (\_exit o \_Exit)**, se il main effettua una **return** oppure se un suo thread riceve un segnale la cui azione è terminare.

Un singolo thread può terminare effettuando un return dalla sua funzione di inizio, oppure eseguendo una **pthread\_exit** o ricevendo un **pthread\_cancel** da un altro thread.

In sostanza, un thread **non effettua** una return ma sempre **pthread\_exit** altrimenti con la return terminano tutti i thread. Tale funzione restituisce come la exit un puntatore che può essere “raccolto”.

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

### Esempi



tF è la funzione di thread e non riceve nulla perciò nella **pthread\_create** ci sarà il parametro **NULL**. Si nota che la funzione tF termina tramite **pthread\_exit** e non ritorna nulla (parametro **NULL**). Dall'altra parte il chiamante deve definire il **pthread\_t (tid)** e deve chiamare la **pthread\_create** che restituisce un valore (per capire se vi è stato errore) e successivamente passerà la variabile tid da riempire e il nome alla funzione da eseguire. Nel momento in cui si esegue tale funzione vi sarà una biforcazione: il thread originale continuerà ad eseguire il codice (partendo da if) mentre il thread eseguirà la funzione tF.

Non è definito quale dei due thread parte prima.

**NON SI PUO' DEFINIRE IL PTHREAD\_T TID COME VARIABILE GLOBALE** (proprio perché la velocità relativa può essere diversa e il tid potrebbe essere vuoto se il T2 parte prima del T1).

## Esempio

Attivazione di N thread con 1 parametro

```

void *tF (void *par) {
    int *tidP, tid;
    ...
    tidP = (int *) par;
    tid = *tidP;
    ...
    pthread_exit (NULL);
}

pthread_t t [NUM_THREADS];
int rc, i;

for (i=0; i<NUM_THREADS; i++) {
    rc = pthread_create (&t[i], NULL, tF,
        (void *) &i);
    if (rc) { ... }
}
pthread_exit(NULL);

```

Collezione i tid

Parametro puntatore a intero

alla funzione. La funzione tF dovrà fare un cast a ritrovo della variabile intera che però è la variabile di ciclo (che quindi varia) e la funzione cattura il valore presente in quel momento ma tutto dipende dallo scheduling (quindi si possono verificare errori). Per risolvere il problema vi sono due opzioni:

Attivazione di N thread con 1 parametro

```

void *tF (void *par) {
    long int tid;
    ...
    tid = (long int) par;
    ...
    pthread_exit(NULL);
}

pthread_t t [NUM_THREADS];
int rc; long int i;

for (i=0; i<NUM_THREADS; i++) {
    rc = pthread_create (&t[i], NULL, tF,
        (void *) i);
    if (rc) { ... }
}
pthread_exit (NULL);

```

Cast di dato singolo void \* ↔ long int

Parametro long int con cast esplicito a un puntatore

Attivazione di N thread con 1 parametro

```

void *tF (void *par) {
    int *tid, taskid;
    ...
    tid = (int *) par;
    taskid = *tid;
    ...
    pthread_exit(NULL);
}

int tA[NUM_THREADS];
for (i=0; i<NUM_THREADS; i++) {
    tA[i] = i;
    rc = pthread_create (&t[i], NULL, tF,
        (void *) &tA[i]);
    if (rc) { ... }
}
pthread_exit (NULL);

```

Cast di vettore di puntatori void \* ↔ int

Se il parametro è in un vettore posso passare il puntatore

## ESEMPIO ERRATO (di proposito)

Si vogliono generare NUM\_THREADS e si definirà quindi un vettore di tid e di volta in volta si raccoglie nella posizione corretta il suo relativo TID. Si chama la funzione tF e in questo caso si passa un **parametro** (si analizza dopo come).

La funzione tF riceve quindi un parametro definendolo come void \* ma normalmente si vuole passare al thread il valore di i (per identificare tramite intero un thread, cosa comune e comoda). Si prende quindi il puntatore di questa variabile (&i) ma lo si trasforma in void \* prima di mandarlo

long int è in qualche modo assimilabile a un puntatore e quindi lo si può passare come parametro by value. **SOLUZIONE POCO PULITA**

Definisco un vettore di interi (che non variano) e ad ogni thread assegno il proprio valore. Questo chiarisce l'idea che ogni thread lavora su una porzione di dati messi in comune. Si può pensare di avere un vettore di strutture (e non di interi) come nell'esempio successivo.

## ESEMPIO AVANZATO

**Esempio**

```

    Attivazione di N thread con 1 struct
    struct tS {
        int tid;
        char str[N];
    };

    void *tF (void *par) {
        struct tS *tD;
        int tid; char str[L];
        tD = (struct tS *) par;
        tid = tD->tid; strcpy (str, td->str);
        ...
    }

    pthread_t t[NUM_THREADS];
    struct tS v[NUM_THREADS];
    ...
    for (i=0; i<NUM_THREADS; i++) {
        v[i].tid = i;
        strcpy (v[i].str, str);
        rc = pthread_create (&t[i], NULL, tF, (void *) &v[i]);
        ...
    }
    ...
  
```

Cast di vettore di strutture  
Puntatore a struct convertito in void \*

In questo caso si è definita una struttura tS (thread Structure) che contiene un intero (il tid) e ad esempio una stringa.

Si definisce un vettore di queste strutture lungo tanto quanto il numero di thread da eseguire e nel ciclo di esecuzione dei thread, prima di eseguire i thread stessi, si va a caricare la stringa e il TID (si può fare anche in un ciclo a parte). Alla fine si fa la stessa cosa di prima passando come prima il puntatore alla cella del parametro della funzione.

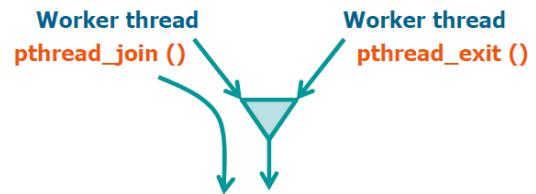
Il tF dovrà trasformare opportunamente la struttura ricevuta e poi catturare i vari campi.

## System call pthread\_join ()

Alla sua creazione un thread può essere dichiarato come **joinable** se un altro thread può effettuare una "wait" (pthread\_join) su di lui oppure **detached** se non si può attendere esplicitamente la sua terminazione (non è joinable).

Se il thread è joinable il suo stato di terminazione viene mantenuto sino a quando un altro thread esegue una **pthread\_join** per quel thread, mentre se è detached il suo stato di terminazione viene subito rilasciato. In ogni caso il thread che chiama la **pthread\_join** rimane bloccato sino a quando il thread richiesto non effettua una **pthread\_exit**.

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



I parametri:

- **pthread\_t tid**: Identificatore del thread atteso (tid)
- **void \*\*valuePtr**: puntatore (senza tipo) che riferisce al valore ritornato dal thread tid che può essere ritornato dalla pthread\_exit, da una return oppure essere PTHREAD\_CANCELED se il thread è stato cancellato

La funzione restituisce 0 in caso di successo e restituisce un codice di errore in caso di fallimento.

**Esempio**

```

    Ritorno lo stato di exit (il pid in questo caso)
    void *tF (void *par) {
        long int tid;
        ...
        tid = (long int) par;
        ...
        pthread_exit ((void *) tid);
    }

    void *status;
    long int pid;
    ...
    /* Wait for threads */
    for (t=0; t<NUM_THREADS; t++) {
        rc = pthread_join (t[t], &status);
        pid = (long int) status;
        if (rc) { ... }
    }
    ...
  
```

t[t] collezionava i tid  
Attesa dei thread e cattura del loro stato in status

La funzione main decide di attendere i thread generati iterando NUM\_THREADS volte facendo una join (attendendo quindi i vari thread) ricevendo lo status (cioè la variabile di uscita di tF). Tale status deve essere riportato al suo tipo originario (definito all'interno di tF come long int) e successivamente può essere letto.

## Esempio

- Utilizzo di una variabile globale comune a più thread

```
int myglobal;  
  
void *threadF (void *arg) {  
    int *argc = (int *) arg;  
    int i, j;  
    for (i=0; i<20; i++) {  
        j = myglobal;  
        j = j + 1;  
        printf ("t");  
        if (*argc > 1) sleep (1);  
        myglobal = j;  
    }  
    printf ("(T:myglobal=%d)", myglobal);  
    return NULL;  
}
```

La variabile globale viene incrementata tramite copia in j

Il thread può attendere oppure no

Il main crea un thread (Esempio 2) raccogliendo nella variabile mythread il TID e chiama la funzione threadF con parametro argc e fa un ciclo in cui la variabile myglobal viene modificata e tutte le volte che si modifica viene stampato un carattere m che sta per main e infine fa la join col thread creato prima.

Anche il worker thread generato dal main fa la stessa cosa, effettuando un ciclo incrementando la variabile myglobal salvandola in j e incrementandola, e infine copiando j in myglobal. Il thread può attendere o meno in base al valore di argc.

## Esempio 2

```
int main (int argc, char *argv[]) {  
    pthread_t mythread;  
    int i;  
    pthread_create (&mythread, NULL, threadF, &argc);  
    for (i=0; i<20; i++) {  
        myglobal = myglobal + 1;  
        printf ("m");  
        sleep (1);  
    }  
    pthread_join (mythread, NULL);  
    printf ("(M:myglobal=%d)", myglobal);  
    exit (0);  
}
```

Siccome entrambi i thread lavorano sulla stessa variabile alcuni elementi possono essere persi e di fatti, analizzando l'esecuzione del programma si hanno diversi risultati.

Il thread parte subito  
Nessun incremento viene perduto

## Esempio 2

```
> ./pgrm  
tttttttttttttttttt (T:myglobal=21)mmmmmmmmmmmmmmmm  
m(M:myglobal=40)  
  
Thread e main si alternano ogni secondo  
Gli incrementi del thread vengono perduti
```

```
> ./pgrm 1  
mtmtmtmtmtmtmtmtmtmtmtmtmtmtmtmtmtmtmtmtmtmtmtmtmt  
(T:myglobal=21)  
M:myglobal=21)  
  
2sec di attesa per il main  
Alcuni incrementi si perdono altri no
```

```
> ./pgrm 1  
mtmtmtmtmtmtmtmtmtmtmttm (T:myglobal=21)mmmmmmmm  
m(M:myglobal=30)
```

Nel primo caso non si passa alcun valore (e l'argomento è 1 e non è maggiore di 1 perciò non si aspetta) e viene stampata m e poi il thread esegue tutte le operazioni (perché il main aspetta 1 secondo) e stampa il valore di myglobal e successivamente parte il main che itera nuovamente e myglobal ha il valore corretto.

Se invece si inserisce il parametro anche il thread aspetta 1 secondo come il main e quindi si perde il 50% degli incrementi.

Se invece il main attende 2 secondi si perdono alcuni incrementi mentre altri no. Per evitare questi problemi

occorrerà **sincronizzare** i thread.

## System call pthread\_cancel ()

```
int pthread_cancel (  
    pthread_t tid  
,
```

Questa funzione termina il thread indicato. È come se tale thread eseguisse una pthread\_exit con parametro PTHREAD\_CANCELED. Il thread che fa la richiesta non attende la terminazione del thread (effettua la richiesta e continua). Il parametro è il

TID da cancellare e restituisce 0 in caso di successo.

## System call pthread\_detach()

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

Questa funzione dichiara il thread **tid** come detached ed eventuali chiamate alla **pthread\_join** dovrebbero fallire poiché non sono più joinable.

### Esempio

- Creare un thread e poi renderlo detached

```
pthread_t tid;
int rc;
void *status;

rc = pthread_create (&tid, NULL, PrintHello, NULL);
if (rc) { ... }

pthread_detach (tid); // Detach a thread

rc = pthread_join (tid, &status);
if (rc) {
    // Error
    exit (-1);
}

pthread_exit (NULL); // Errore se si attende
```

### Esempio

- Creare un thread utilizzando l'attributo della **pthread\_create**

```
pthread_attr_t attr;
void *status;

pthread_attr_init (&attr);
pthread_attr_setdetachstate (&attr,
    PTHREAD_CREATE_DETACHED);
// PTHREAD_CREATE_JOINABLE;

rc = pthread_create (&t[t], &attr, tP, NULL);
if (rc) { ... }

pthread_attr_destroy (&attr);

rc = pthread_join (thread[t], &status);
if (rc) {
    // Error
    exit (-1);
}
```

Crea un thread come detached

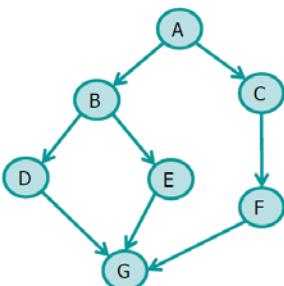
Distrugge l'attributo

Errore se si attende

Nell'esempio a destra viene creato un thread che è già detached e in questo caso vengono utilizzati gli **attributi**. Il default (senza parametri) è che tale thread sia joinable.

### Esercizio proposto

- Realizzare, tramite l'utilizzo di thread, il seguente grafo di precedenza

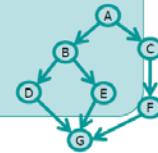


### Soluzione

```
void waitRandomTime (int max){
    sleep ((int)(rand() % max) + 1);
}

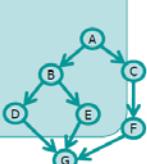
int main (void) {
    pthread_t th_cf, th_e;
    void *retval;

    srand (getpid());
    waitRandomTime (10);
    printf ("A\n");
    
```



### Soluzione

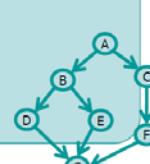
```
waitRandomTime (10);
pthread_create (&th_cf,NULL,CF,NULL);
waitRandomTime (10);
printf ("B\n");
waitRandomTime (10);
pthread_create (&th_e,NULL,E,NULL);
waitRandomTime (10);
printf ("D\n");
pthread_join (th_e, &retval);
pthread_join (th_cf, &retval);
waitRandomTime (10);
printf ("G\n");
return 0;
}
```



### Soluzione

```
static void *CF () {
    waitRandomTime (10);
    printf ("C\n");
    waitRandomTime (10);
    printf ("F\n");
    return ((void *) 1); // Return code
}

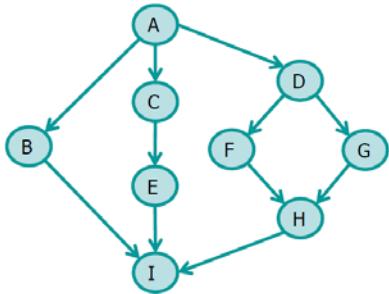
static void *E () {
    waitRandomTime (10);
    printf ("E\n");
    return ((void *) 2); // Return code
}
```



## ESERCIZIO DA SVOLGERE

### Esercizio

- Realizzare, tramite l'utilizzo di thread, il seguente grafo di precedenza



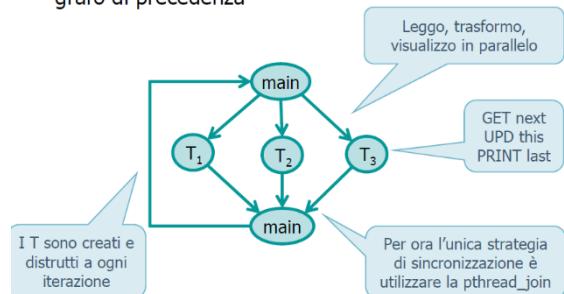
## Esempio svolto

### Esercizio

- Un file contiene un numero di caratteri indefinito
- Realizzare un programma con thread concorrenti in cui tre thread ( $T_1, T_2, T_3$ ) lavorino in pipeline per gestire il file
  - $T_1$ : Legge dal file un carattere alla volta
  - $T_2$ : Trasforma il carattere letto da  $T_1$  in maiuscolo
  - $T_3$ : Visualizza il carattere prodotto da  $T_2$  su standard output

### Soluzione

- Realizzare, tramite l'utilizzo di thread, il seguente grafo di precedenza



Non conoscendo ulteriori meccanismi di sincronizzazione bisogna generare, fare la join e ricreare i thread ad ogni iterazione. Segue soluzione.

### Soluzione

```

static void *GET (void *arg) {
    char *c = (char *) arg;
    *c = fgetc (fg);
    return NULL;
}

static void *UPD (void *arg) {
    char *c = (char *) arg;
    *c = toupper (*c);
    return NULL;
}

static void *PRINT (void *arg) {
    char *c = (char *) arg;
  
```

### Soluzione

```

FILE *fg;

int main (int argc, char ** argv) {
    char next, this, last;
    int retC;
    pthread_t tGet, tUpd, tPrint;
    void *retV;

    if ((fg = fopen(argv[1], "r")) == NULL){
        perror ("Errore fopen\n");
        exit (0);
    }
    this = ' ';
    last = ' ';
    next = ' ';
  
```

Il primo thread fa la GET da file, il secondo fa l'update del carattere in maiuscolo e il terzo fa la stampa.

Vecchio codice del prof Laface mantenuto uguale (tramite funzioni brutte come fgetc, putchar).

Il main apre il file e inizia tre variabili globali (next, last, next) che possono essere anche inserite come variabili locali e passate come parametro alle funzioni, che sono inizializzate al carattere spazio.

## Soluzione

È possibile gestire  
separatamente i  
primo due caratteri

```
while (next != EOF) {  
    retC = pthread_create (&tGet,NULL,GET,&next);  
    if (retC != 0) fprintf (stderr, ...);  
    retC = pthread_create (&tUpd,NULL,UPD,&this);  
    if (retC != 0) fprintf (stderr, ...);  
    retC = pthread_create (&tPrint,NULL,PRINT,&last);  
    if (retcode != 0) fprintf (stderr, ...);  
    retC = pthread_join (tGet, &retV);  
    if (retC != 0) fprintf (stderr, ...);  
    retC = pthread_join (tUpd, &retV);  
    if (retC != 0) fprintf (stderr, ...);  
    retC = pthread_join (tPrint, &retV);  
    if (retC != 0) fprintf (stderr, ...);  
    last = this;  
    this = next;  
}
```

## Soluzione

Gestione degli ultimi  
due caratteri (coda)

```
// Last two chars processing  
  
retC = pthread_create(&tUpd,NULL,UPD,&this);  
if (retC!=0) fprintf (stderr, ...);  
retC = pthread_create(&tPrint,NULL,PRINT,&last);  
if (retC != 0) fprintf (stderr, ...);  
retC = pthread_join (tUpd, &retV);  
if (retC != 0) fprintf (stderr, ...);  
retC = pthread_join (tPrint, &retV);  
if (retC != 0) fprintf (stderr, ...);  
retC = pthread_create(&tPrint,NULL,PRINT,&this);  
if (retC != 0) fprintf (stderr, ...);  
return 0;  
}
```

Successivamente si ha un lungo ciclo in cui sino a quando non si ha EOF si fanno 3 pthread\_create (ognuno lavora sul proprio carattere).

Dopo aver fatto le create il main effettua le tre join (in qualsiasi ordine) e alla fine si slittano i tre caratteri.

Si legge sempre un solo carattere.

Si creano e si distruggono perché i thread potrebbero andare a velocità differenti.

Bisogna gestire separatamente gli ultimi due caratteri in coda.

Si aggiorna il corrente, si stampa l'ultimo, si fa la join e si stampa ciò che è stato appena aggiornato.

Il passo successivo sarà evitare la ricreazione dei thread.

## 18. Concorrenza: Aspetti teorici

Definizione di **concorrenza**:

*"Task multipli (due o più) che vengono eseguiti in intervalli di tempo sovrapposti, senza ordine specifico particolare."*

L'esecuzione di tali task sembra contemporanea, ma in realtà non lo è e ciò è dovuto al time-slicing della CPU, cioè allo scheduler (context switching) che dedica l'unità di calcolo ai vari task per unità di tempo infinitesimali.

Definizione di **parallelismo**:

*"Task multipli (o parti diverse dello stesso task) che vengono eseguiti (eseguite) negli stessi intervalli di tempo in sistemi multi-processori o multi-core."*

Viene permesso da strutture hardware apposite (multi-CPU o multi-core) e si ottiene trasformando un flusso di esecuzione sequenziale in uno parallelo.

Concorrenza e parallelismo non identificano la stessa cosa:

- Concorrenza = manipolare molte cose nello stesso istante
- Parallelismo = fare molte cose nello stesso istante

I termini spesso si usano in maniera intercambiabile. I due concetti esistono da molti anni poiché lo sviluppo è stato sorprendente dopo il 2004, anno in cui Intel cancella i processori Tejas e Jayhawk poiché l'aumento delle frequenze (clock) dei processori ha raggiunto i suoi limiti a causa dell'eccessivo consumo di potenza.

$$P = C \cdot V^2 \cdot F$$

A causa della fine del "frequency scaling" il parallelismo è diventato uno dei principali paradigmi della programmazione. Essa porta con se però numerose sfide e insidie (bug). Questi cambiamenti hanno portato a cambiamenti analoghi nei paradigmi utilizzati.

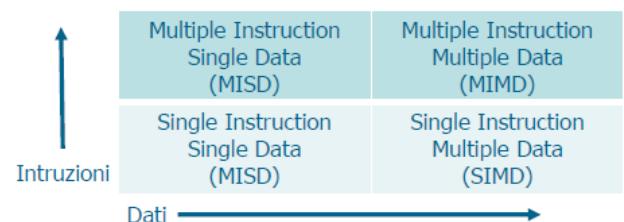
La **Prima legge di Moore del 1965** diceva che: *"Il numero di transistor nei processi raddoppierà ogni 12 mesi"*.

Nel **2010** però la **Legge di Bill Dally** afferma che *"Seguire la legge di Moore non ha più senso. Possiamo aumentare il numero di trasnsistor e dei core di 4 volte ogni 3 anni. Facendo lavorare ogni core leggermente più lentamente, perciò in maniera più efficiente, possiamo più che triplicare le prestazioni mantenendo lo stesso consumo totale."*

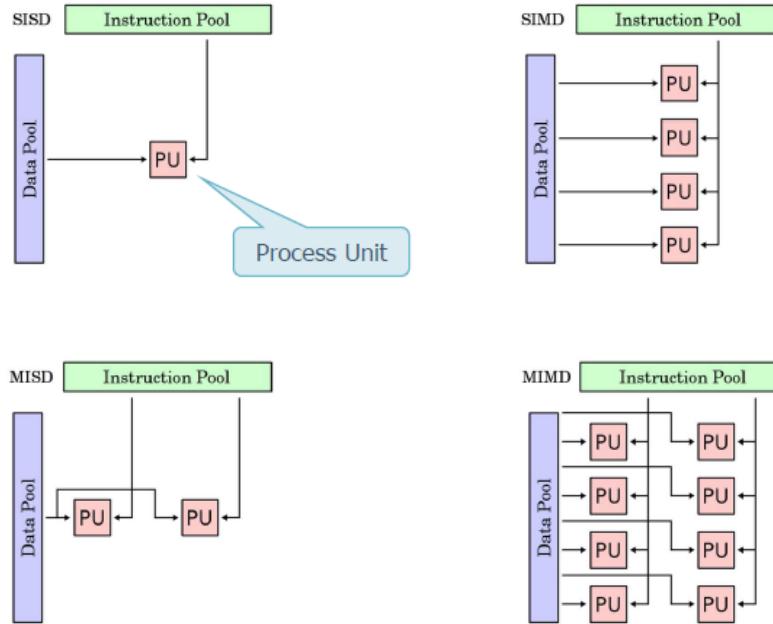
Esistono diverse tipologie di parallelismo:

- **Bit-level**: la lunghezza di una "word" determina l' "efficacia" di una istruzione (e.g., adder a 8 vs 16 bit)
- **Instruction-level**: utilizzo di "multi-stage" pipelines per l'esecuzione del flusso di istruzioni (e.g., fetch, decode, execute)
- **Task-level**: computazioni distinte sono eseguite in parallelo (e.g., un ordinamento e un prodotto matriciale sono eseguiti contestualmente)

La prima classificazione per architetture parallele venne introdotta da Flynn ed è stata **parzialmente superata** poiché molte architetture sono miste o non classificabili direttamente, ma ancora **ampiamente utilizzata** poiché semplice e comprensibile.



- **SISD (Single Instruction Single Data)** schema classico senza parallelismo
- **SIMD (Single Instruction Multiple Data)** una istruzione singola opera su diversi flussi di dati
- **MISD (Multiple Instruction Single Data)** istruzioni multiple operano su un singolo flusso di dati
- **MIMD (Multiple Instruction Multiple Data)** istruzioni multiple operano su diversi flussi di dati



## Speed-up

Lo scopo principale del parallelismo è l'aumento dell'efficienza. Per valutarla, è possibile analizzare i parametri quali **tempo** e **memoria**. Esistono diverse metriche per il calcolo dei tempi di esecuzione, tra cui:

- **User time:** tempo totale che la CPU dedica a eseguire un determinato task. Questo metodo non tiene conto del tempo perso nella gestione (ad esempio operazioni di I/O, esecuzioni di routine a livello kernel, etc..)
- **CPU time:** tempo totale dedicato dalla CPU all'esecuzione di un task. Questo a differenza del precedente tiene conto dei tempi di gestione. Su una architettura parallela occorrerà valutare il tempo dedicato al task da parte di tutte le unità di calcolo. Nella maggior parte dei casi tale tempo sarà maggiore del tempo del processo sequenziale in esecuzione su un processore unico. La valutazione confronto in base allo user o al cpu time è tendenzialmente peggiorativa.
- **Wall clock time:** detto anche **elapsed time** (tempo trascorso) è il tempo effettivamente richiesto per terminare un determinato compito. In altre parole:

$$\text{Wall clock time} = \text{tempo di fine} - \text{tempo di inizio task}$$

equivale a cronometrare il tempo di esecuzione indifferentemente dal fatto che tale esecuzione sia portata avanti su sistemi mono o multi processore. Prendendo questa metrica come riferimento si ha:

$$\text{speedup} = \frac{\text{Wall clock time sequential algorithm}}{\text{Wall clock time parallel algorithm}}$$

Raddoppiando il numero di elementi di processamento dovrebbe dimezzarsi il wall-clock time perciò i vantaggi dovrebbero essere lineari, ma tale comportamento si ottiene **raramente** perché se un processo non è parallelizzabile aumentare il numero di processori/core **non modifica** il run-time. Inoltre, tale comportamento si ha solo per **un numero ridotto di processi/core** perché dopo una dipendenza lineare iniziale la curva dello speed-up si appiattisce (asintoto orizzontale)

## Legge di Amdhal

I vantaggi teorici ottenibili mediante concorrenza sono stati analizzati per la prima volta da Amdhal. La legge di Amdhal [1967] specifica il miglioramento teorico ottenibile tramite parallelismo:

$$speedup = \frac{1}{S + \frac{1-S}{n}}$$

Dove:

- $n$  = numero di processori o core
- $S$  = percentuale dell'elapsed time trascorso ad eseguire la parte sequenziale di un programma (non parallelizzabile)

Alla legge di Ahmdal occorrerebbe però aggiungere l'**overhead** relativo alla gestione degli  $n$  thread (o processi)

$$speedup = \frac{1}{S + \frac{1-S}{n} + H(n)}$$

Dove  $H(n)$  è l'overhead di gestione del sistema operativo e di sincronizzazione inter-thread.

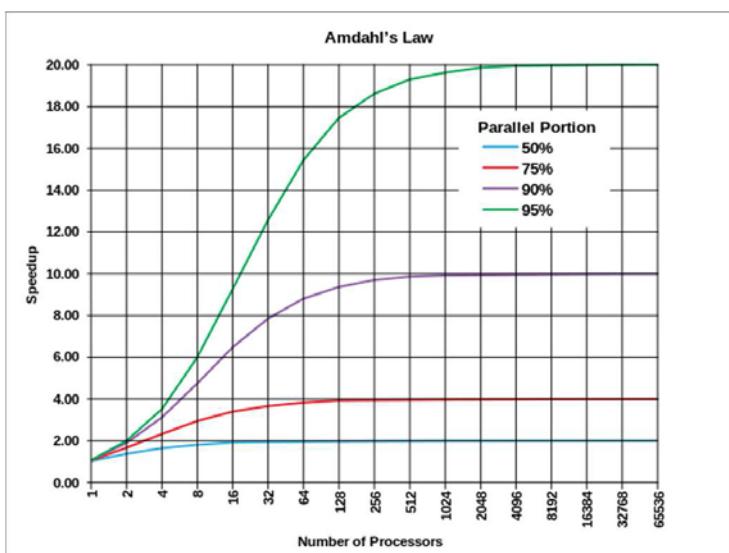
Nell'ipotesi più ottimistica  $H(n)=0$  e all'aumentare del numero di processori si ha:

$$speedup = \lim_{n \rightarrow \infty} \frac{1}{S + \frac{1-S}{n}} = \frac{1}{S}$$

Esempi:

- $S = 10\% \rightarrow speedup_{massimo} = 10$
- $S = 20\% \rightarrow speedup_{massimo} = 5$
- $S = 50\% \rightarrow speedup_{massimo} = 2$

In altre parole, piccole porzioni di programma non parallelizzabili limitano lo speed-up complessivo ottenibile



Corollario alla legge di Amdhal: “Diminuire la parte serializzabile e aumentare la parte parallelizzabile è più importante che aumentare il numero di processori.”

## Limiti legge di Amdhal

I limiti non dipendono solo dalla disponibilità di cicli di CPU, ma anche da altri fattori:

- I sistemi multi-core possono avere molteplici cache abbassando la latenza della memoria e aumentando l'efficienza del sistema
- Alcuni algoritmi hanno formulazioni parallele migliori ovvero con un numero minore di passi computazionali
- Amhdal assume che la dimensione dei problemi rimanga costante, mentre in genere aumenta con l'aumentare delle risorse disponibili e ciò che rimane costante è il tempo di esecuzione

## Legge di Gustafson

Negli anni '80 presso i Sandia National Lab sono stati ottenuti speed-up lineari con 1024 processori su applicazioni sulle quali Amhdal avrebbe previsto un comportamento non lineare. Perciò la **legge di Gustafson** (o equazione di Barsis) prevede uno speed-up lineare

$$speedup = N + (1 - N) \cdot S$$

Dove:

- N = numero di processori
- S = frazione tempo spesa nella parte seriale

## Parallelizzazione

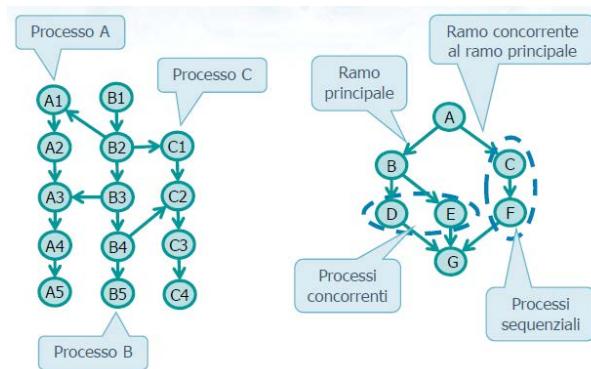
La parallelizzazione di un algoritmo può essere effettuata solo mediante decomposizione

- **Task decomposition:** decomposizione del programma in funzioni e analisi di quali funzioni possono agire in parallelo
- **Data decomposition:** decomposizione del problema in funzione del parallelismo applicabile sui dati piuttosto che della sua natura funzionale / logica
- **Data flow decomposition:** decomposizione del problema in base al flusso di dati tra le varie funzioni o dei task da completare

La decomposizione di un problema può essere effettuata solo conoscendone le dipendenze. I vincoli di precedenza possono essere rappresentati mediante **grafo di precedenza** in relazione con il Control Flow Graph (CFG) e con gli Alberi di generazione dei processi.

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

- I vertici corrispondono a istruzioni singole, blocchi di istruzioni, processi
- Gli archi corrispondono a condizioni di precedenza (Un arco dal vertice A al vertice B significa che B può essere eseguito solo una volta terminato A)
- La precedenza può essere imposta mediante tecniche di sincronizzazione (Sincronizzazione = meccanismo utilizzato per imporre dei vincoli all'ordine di esecuzione delle unità di processamento (processi o thread))



## **19. Comandi UNIX & Linux (Parte B)**

Questo pacchetto di slide non contiene informazioni utili ai fini dell'esame.

## 20. Le Shell

Le shell sono lo **strato più esterno** del sistema operativo che permette di avere un'interfaccia utente verso il sistema operativo, interpretando i comandi degli utenti e passandoli al kernel. Le shell erano l'unica interfaccia prima dell'introduzione dei server grafici.

In UNIX le shell **non sono** parte del kernel, ma un normale **processo utente** simile a DOS (ma più potente) ed è anche un ambiente di programmazione **nativo** del SO.



Le shell permettono la **gestione di comandi** su linea di comando ma anche la **scrittura di programmi (script)** tramite la memorizzazione dei comandi desiderati in un file. L'esecuzione dei comandi richiama il file stesso.

Scrivere uno script evita di digitare ripetutamente complesse sequenze di comandi e quindi **automatizzare** operazioni ripetitive. Tra le shell principali l'originale è la **Bourne shell (sh)** ed altre varianti, fino al **Bourne Again shell (bash)** che è la più recente e utilizzata (ed anche pseudo compatibile con le altre).

Qui sono riportati esempi di costrutti fatti in shell **tcsh** e in shell **bash**

tcsh	bash
<code>set myVar = "ciao"</code>	<code>myVar="ciao"</code>
<code>setenv MY_DIR /home/usr/</code>	<code>export MY_VAR=/home/usr/</code>
<code>if (\$str1===\$str2) then ... else ... endif</code>	<code>if test \$str1==\$str2 then ... else ... fi</code> <code>if [ \$str1==\$str2 ]; then ... else ... fi</code>

Ogni sistema operativo ha più shell disponibili e ciò si può vedere andando a guardare l'elenco in **/etc/shells**. E' possibile cambiare shell tramite comando **chsh**.

Una shell può essere attivata **automaticamente al login** oppure è possibile **annidarla** dentro un'altra shell. Nel momento in cui si esegue una shell vengono effettuate alcune operazioni presenti nei **file di avvio** che ne consentono l'inizializzazione. Le categorie di file di avvio sono due: **file di login** (accedendo con password) oppure **file non di login** in cui la shell viene eseguita mediante icona o menù di sistema.

Una delle difficoltà dell'utilizzo delle shell è che alcuni caratteri assumo un **significato particolare** all'interno delle shell. Gli script bash mettono a disposizione complessi meccanismi di sostituzione o di espansione e si tratta di graffe, tilde, variabili, etc.. e ciò avviene seguendo un ordine preciso.

```
> nome=Gian
> echo $nomemarco
echo: comando di stampa
> echo ${nome}marco
{Gian}marco
> echo ${nome}marco
Gianmarco
```

Variabile non esistente

In questo esempio si assegna alla variabile "nome" la parola Gian. Se si facesse una **echo \$nome** si stamperebbe "Gian" (poiché \$ va a prendere il contenuto della variabile). Nel caso in cui si stampasse **echo \$nomemarco** non esistendo tale variabile non si avrebbe nulla a video.

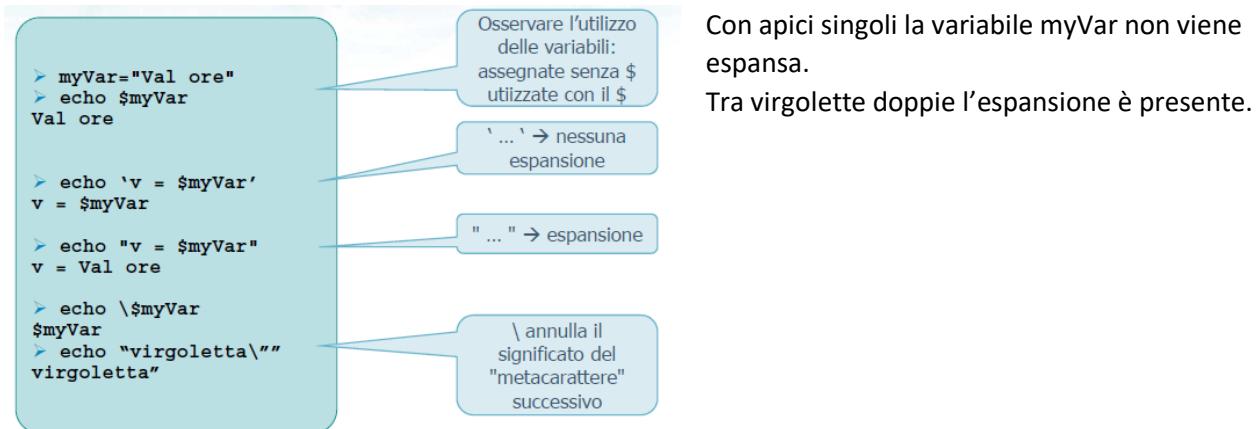
Se invece si vuole stampare prima nome e poi marco, si

utilizzano le parentesi graffe e all'interno la variabile viene espansa in Gian. Se invece il dollaro viene inserito all'esterno delle graffe, le parentesi vengono omesse nella stampa e la variabile viene espansa.

## Quoting

Per quoting si intende l'utilizzo delle virgolette:

- **Apici ''**: Identificano una stringa al cui interno **non sono espanso** le variabili e non possono essere annidati
- **Le virgolette " "**: Identificano una stringa al cui interno le variabili **sono espanso** e possono essere annidate
- **Il backslash \**: Identifica il carattere di escape, ovvero elimina il significato speciale del carattere che lo segue



**Cattura dello stdout di un comando:** lo standard output di un comando può essere **catturato** mediante:

- Sequenza di caratteri \$(...)
- Apici inversi (back-quote) ``

Alt-96: `      Alt-239: '      Alt-123 {      alt-125 }

In particolare, l'output di un comando può essere memorizzato in una variabile

```
>d=$(date)
>echo $d
>Fri Nov 22 10:00:01
      CET 2013
>d=`date`
...

```

obsoleta

```
>out=`cat file.txt`
>echo $out
>... listato file ...
...
>out=< file.txt
>echo $out
>... listato file ...

```

Ogni shell ricorda l'elenco degli ultimi comandi digitati ed in bash tale elenco è contenuto nel file **.bash\_history** ed è memorizzato nella home dell'utente. Le shell permettono di fare riferimento a tale elenco.

Comando	Significato
history	Mostra l'elenco dei comandi eseguiti precedentemente
!n	Segue il comando numero n nel buffer
!str	Esegue l'ultimo comando che inizia con str
^str1^str2	Sostituisce nell'ultimo comando str1 con la str2

## Aliasing

Nelle shell è possibile definire nomi nuovi a comandi esistenti ed il comando **alias** permette di definire tali nomi:

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

Questo comando definisce un nuovo alias per "stringa". La shell conserva un elenco di alias ed il semplice comando **alias** (seguito da niente) fornisce l'elenco degli alias attivi nella shell utilizzati. I vecchi alias possono venire eliminati tramite il comando

```
unalias nome
```

che elimina l'alias nome dalla shell

### Esempi

```
> alias
alias egrep='egrep --color=auto'
alias emacs='emacs -r -geometry 100x36 -fn 9x15 &
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias ls='ls --color=auto'
alias mx='xdvi -mfmode lfour:1200'

> alias ll="ls -la"
          Alias esistenti

          Definizione di un
          nuovo alias

> unalias emacs
> unalias ll
          Cancellazione di un alias
          pre-esistente
          (l'eventuale comando ritorna
          ad essere quello che era)
```

## 21. Gli script di shell

I linguaggi di shell sono linguaggi **interpretati**, cioè non esiste una fase di compilazione esplicita, ma viene interpretato a run time (cioè è presente un interprete che capisce i comandi e li trasforma in linguaggio macchina).

Vantaggi e svantaggi:

- Le shell sono disponibili in ogni ambiente UNIX/Linux
- Il ciclo di produzione più veloce
- Minore efficienza in fase di esecuzione
- Minori possibilità e ausili di debug

Gli script sono utilizzati per la scrittura di software “Quick and dirty” e prototipale, i.e., tradotto successivamente in un linguaggio di alto livello.

### BASH vs Python (e altri)

**Scelta:** Il principale punto di forza di BASH nei confronti di altri linguaggi (python, ruby, lua, etc.) è la sua ubiquità. Se il numero di righe di codice è inferiore a 100, conviene scegliere BASH, altrimenti PYTHON

**Prestazioni:** per avere alte prestazioni normalmente si scrive un programma non uno script. L'interprete BASH è molto veloce nel partire (start time) e se occorre manipolare file ASCII, oppure utilizzare pesantemente comandi o filtri tipo sort, uniq, etc., BASH è più adatto e veloce (“will smoke Python performance wise”). Se occorre manipolare numeri floating point Python è conveniente (“will win hands down”).

Gli script sono normalmente memorizzati in un file di esensione **.sh (.bash)** pur ricordando che in Linux le estensioni non vengono utilizzate per determinare il tipo di file. Possono essere eseguiti mediante due tecniche: **esecuzione diretta o indiretta**.

#### Esecuzione diretta

```
./scriptname args
```

Questa tecnica implica rendere l'oggetto (scriptname) eseguibile:

- Bisogna ricordarsi di fornire ai file i permessi di esecuzione **chmod +x ./scriptname**
- La prima riga dello script deve specificare il nome dell'interprete dello script **#!/bin/bash** o **#!/bin/sh**
- È possibile eseguire lo script utilizzando una shell particolare **/bin/bash ./scriptname args**

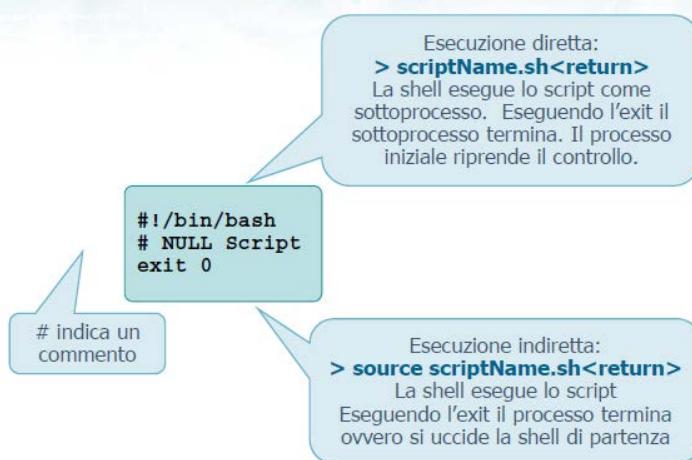
In questo caso lo script viene eseguito da una sotto-shell. Ovvero, eseguire uno script in maniera diretta implica eseguire un nuovo processo e dunque ambienti (variabili) del processo originario e di quello eseguito non coincidono. Le modifiche alle variabili di ambiente effettuate dallo script sono perdute.

## Esecuzione indiretta

```
source ./scriptname args
```

Si invoca la shell con il nome del programma come parametro. È la shell corrente a eseguire lo script e non è necessario che lo script sia eseguibile e le modifiche effettuate dallo script alle variabili di ambiente rimangono valide nella shell corrente.

### Esempio: esecuzione diretta e indiretta



Il comando lanciato con esecuzione diretta uccide la sotto-shell, mentre quello con esecuzione indiretta uccide la shell corrente (e chiude la finestra).

## Debug di uno script

Non esistono tool specifici per effettuare il debug di script bash, ma è ovviamente sempre possibile aggiungere delle "echo" esplicite oppure eseguire uno script in "debug" in maniera:

- Completa (l'intero script): si ottiene indicando una opzione di "debug" a livello di intero script
- Parziale (ovvero solo alcune righe dello script): si ottiene indicando una opzione di "debug" a livello di alcune righe dello script mediante il comando **set**

Opzioni disponibili

- **-o noexec, -n** esegue un controllo sintattico ma non esegue lo script
- **-o verbose, -v** visualizza i (fa l'eco dei) comandi eseguiti
- **-o xtrace, -x** visualizza la traccia di esecuzione dell'intero script
- **-o nounset, -u** riporta un errore per variabile non definita

Debug dell'intero script:

- Dal comando **/bin/bash -n ./scriptname args**
- All'interno dello script **#!/bin/bash -v**, **#!/bin/bash -x**, etc..

Debug parziale (inserendo al posto dei puntini il codice)

- **set -o verbose ... set +o verbose**
- **set -v ... set +v**
- **set -x ... set +x**

## Sintassi: regole generali

Il linguaggio bash è relativamente di "alto livello", ovvero è in grado di mischiare comandi standard di shell (ls, wc, find, grep, ...) e costrutti "standard" del linguaggio di shell come input e output, variabili e parametri, operatori (aritmetici, logici, etc.), costrutti di controllo (condizionali, iterativi), array, funzioni, etc. Le istruzioni/comandi sulla stessa riga devono essere separate dal carattere ';' ma spesso si scrivono istruzioni su righe successive.

Il carattere # indica la presenza di un **commento** sulla riga ed il commento si espande dal carattere # in poi su tutta la riga.

La system call exit permette di terminare uno script restituendo un eventuale codice di errore

- exit
- exit 0/1 (0 è il valore vero nelle shell, qualsiasi altro numero invece è falso)

```
#!/bin/bash
# This line is a comment
rm -rf ../../newDir/
mkdir ../../newDir/
cp * ../../newDir/
ls ../../newDir/ ;
# 0 is TRUE in shell programming
exit 0
```

Dalla shell chiamante:  
echo \$?  
fornisce 0

Inizia con una stringa dell'interprete che deve leggere il comando (da trovare col comando whereis bash). Seguono una serie di comandi shell classici e termina con exit 0 (true). Il ; presente è superfluo. Per stampare il valore "0" bisogna effettuare echo \$? una volta lanciato lo script.

I parametri dello script (riga di comando) possono essere individuati mediante \$ e si distinguono in:

- **Parametri posizionali:** \$0 indica il nome dello script, mentre \$1, \$2, \$3, .. tutti gli altri.
- **Parametri speciali:** \$\* indica l'intera riga di comando (da \$0 incluso in poi); \$# memorizza il numero di parametri (escludendo \$0 cioè il nome dello script); \$\$ memorizza il PID del processo.

```
#!/bin/bash
# Using command line parameters
echo "Il programma $0 e' in run"
echo "Parametri: $1 $2 $3 ... "
echo "Numero dei parametri $#"
echo "Lista dei parametri $*"
shift
echo "Parametri: $1 $2 $3 ... "
shift
echo "Parametri: $1 $2 $3 ... "
exit 0
```

Le "..." effettuano l'espansione delle variabili  
\$0, \$1, etc. possono anche comparire fuori dalle "  
\$0 rimane immutato; quindi \$1=\$2, \$2=\$3, etc.  
\$0 rimane immutato; quindi \$1=\$2, \$2=\$3, etc.

sui parametri è possibile applicare il comando **shift** che sposta a sinistra di una posizione tutti i parametri (facendo shiftare ad esempio il parametro 2 in 1), escludendo \$0 che resta sempre tale.

## Variabili

Si suddividono in

- **Locali** (o di shell): disponibili solo nella shell corrente
- **Globali** (o d'ambiente): disponibili in tutte le sotto-shell, ovvero sono esportate dalla shell corrente a tutti i processi da essa eseguiti

Queste variabili hanno delle caratteristiche:

- Non vanno dichiarate poiché si creano assegnandone un valore
- Sono case sensitive, cioè var != VAR != Var != ...

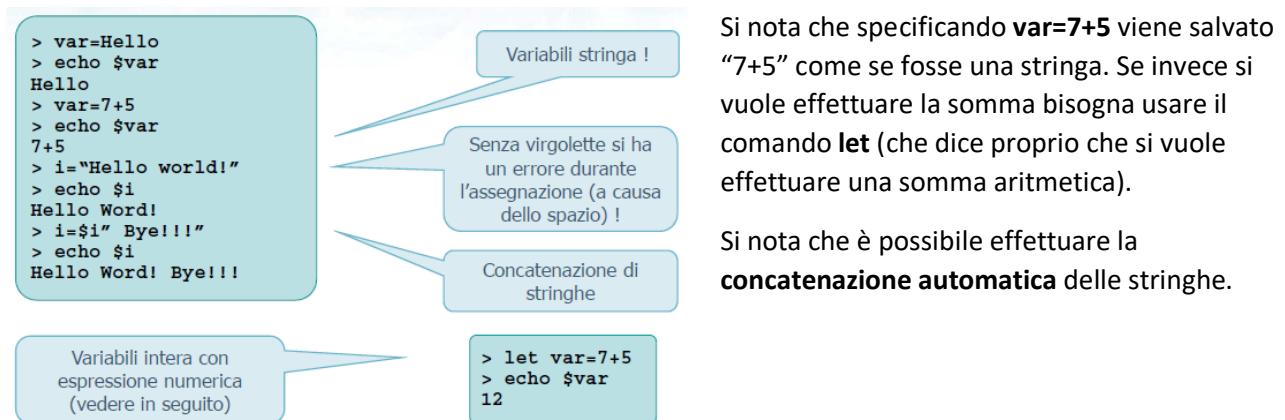
Alcune di queste variabili sono riservate a scopi particolari e l'elenco di tutte le variabili definite e il relativo valore viene visualizzato con il comando **set**. Per cancellare il valore di una variabile si utilizza il comando **unset**, ad esempio **unset nome**.

## Variabili locali

Le variabili locali sono caratterizzate da nome e contenuto: il **contenuto** ne specifica il tipo (costante, stringa, intere, vettoriali o matriciali) e di **default** viene memorizzata una **stringa** anche quando il valore è numerico.

L'assegnazione avviene in questo modo: **nome="valore"** (non devono esserci spazi prima e dopo '=' e le virgolette sono necessarie se il valore assegnato contiene spazi)

Mentre l'utilizzo richiede il \$: **\$nome**



## Variabili globali

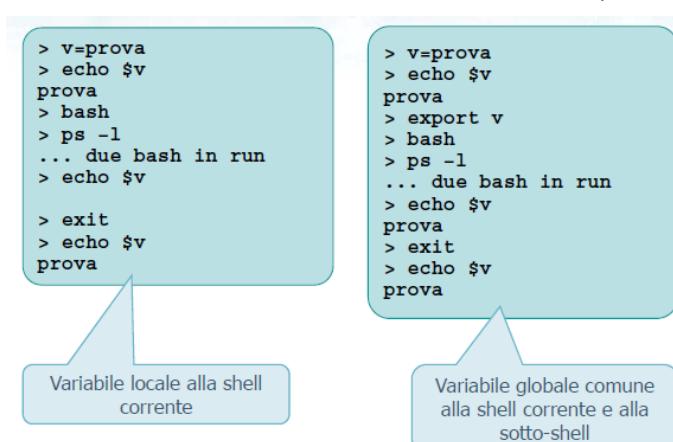
Per rendere una variabile “globale”, ovvero permettere la sua visibilità anche da altri processi si utilizza il comando **export**

**export nome**

Si osservi che diverse variabili di ambiente sono **riservate** e **pre-definite** e quando una shell viene eseguita tali variabili sono inizializzate automaticamente a partire da valori dell’”environment”. In genere tali

variabili sono definite con **lettere maiuscole** per distinguere da quelle utente e si possono visualizzare con il comando **printenvb** (o **env**).

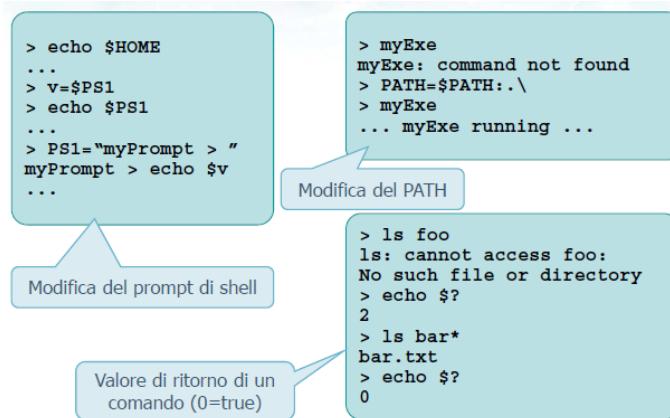
Export non verrà mai usato all'esame.





Variabili predefinite	
Variabile	Significato
\$?	Memorizza il valore di ritorno dell'ultimo processo: 0 in caso di successo, valore diverso da 0 (compreso tra 1 e 255) in caso di errore. Nelle shell il valore 0 corrisponde al valore vero (al contrario del linguaggio C).
\$SHELL	Indica la shell in uso corrente
\$LOGNAME	Indica lo username utilizzato per il login
\$HOME	Indica la home directory dell'utente corrente
\$PATH	Memorizza l'elenco dei direttori separati da ':' utilizzati per la ricerca dei comandi (eseguibili)
\$PS1	Specificano il prompt principale e quello ausiliario (di solito '\$' e '>' rispettivamente)
\$IFS	Elenca i caratteri utilizzati per separare le stringhe lette da input (vedere comando read della shell)

Altri esempi più utili dei precedenti



Se si modifica la variabile globale PATH si potrebbe non riuscire più a trovare molti comandi di shell poiché tale variabile è utilizzata per la ricerca dei comandi eseguibili.

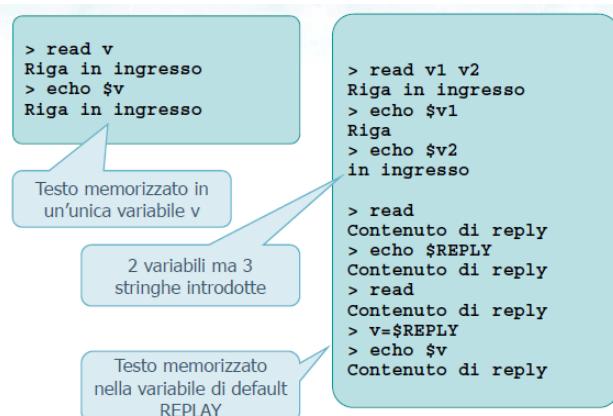
## Lettura (input)

La funzione **read** permette di eseguire dell'input interattivo, in altre parole permette di leggere una riga da stdin

### Read [opzioni] var1 var2 ... var<sub>n</sub>

Ogni read può essere seguita o meno da una lista di variabili ed ogni variabile specificata conterrà una delle stringhe introdotte in ingresso. Eventuali stringhe in eccesso saranno memorizzate tutte nell'ultima variabile e nel caso non siano specificate variabili, tutto l'input viene memorizzato nella variabile REPLY. La read supporta alcune opzioni, tra cui:

- **-n nchars** che ritorna dalla lettura dopo nchars caratteri senza attendere il new line
- **-t timeout** che imposta un timeout sulla fase di lettura e restituisce 1 se non si introducono i dati entro timeout secondi



## Esercizi

**Esercizio**

❖ Si scriva uno script di shell che a seguito di due messaggi legga da tastiera due valori interi e ne visualizzi la somma e il prodotto

```
#!/bin/bash
# Sum and product
echo -n "Reading n1: "
read n1
echo -n "Reading n2: "
read n2
let s=n1+n2
let p=n1*n2
echo "Sum: $s"
echo "Product: $p"
exit 0
```

-n non a capo

Lettura da tastiera

Espressioni aritmetiche (vedere in seguito)

No spazi prima e dopo =, +, \*

**Esercizio**

❖ Si scriva uno script che legga da tastiera il nome di un utente e visualizzi quanti login ha effettuato

➤ La lista degli utenti connessi è fornita dal comando who oppure w

```
#!/bin/bash
# Number of login(s) of a specific user
echo -n "User name: "
read user

# who is logged | the user | word count #lines
times=$(who | grep $user | wc -l)

echo "User $user has $times login(s)" --lines = -l = #line
exit 0
```

Uso di comandi di shell, variabili, etc.

--lines = -l = #line

**Esercizio**

❖ Si scriva uno script che legga da tastiera una stringa e ne visualizzi la lunghezza

```
#!/bin/bash
# String length
echo "Type a word: "
read word

# echoing without newline | word count chars
l=$(echo -n $word | wc -c)

echo "Word $word is $l characters long"
exit 0
```

echo -n = no new line

--chars = -m = #char  
-bytes = -c = #bytes

Commenti es. 1: echo -n non va a capo

Commenti es.2: who stampa tutti gli user online, in pipe con grep vengono filtrati quelli corrispondenti a user e infine wc -l ne conta le righe.

Commenti es.3: della word si fa un eco per poterla stampare senza inserire il carattere newline facendo -n (ma non viene stampata) e inviare in pipe a wc -c che ne conta i caratteri.

## Scrittura (output)

Le operazioni di visualizzazione possono essere effettuate con le funzioni

- echo
- printf

La funzione **printf** ha una sintassi simile a quella del linguaggio C poiché utilizza sequenze di escape e non è necessario separare i vari campi con la ",". Serve per formattare un output.

La funzione **echo** visualizza i propri argomenti, separati da spazi e terminati da un carattere di "a capo". Accetta diverse opzioni tra cui **-e** che interpreta i caratteri di escape (\b backspace, \n newline, \t tabulazione, \\ barra inversa) e anche l'opzione **-n** che sopprime il carattere di "a capo" finale.

**Esempi: I/O**

```
echo "Printing with a newline"
echo -n "Printing without newline"
echo -e "\n escape \t\t characters"
printf "Printing without newline"
printf "%s \t%s\n" "Ciao. It's me:" "$HOME"
```

Diverse operazioni di output

I & O insieme in uno script intero

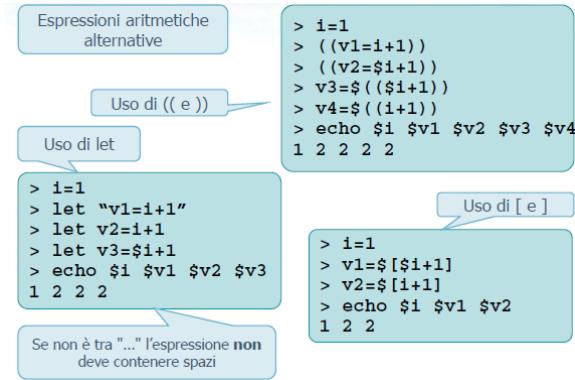
```
#!/bin/bash
# Interactive input/output
echo -n "Insert a sentence: "
read w1 w2 others
echo "Word 1 is: $w1"
echo "Word 2 is: $w2"
echo "The rest of the line is: $others"
exit 0
```

## Espressioni aritmetiche

Per esprimere espressioni aritmetiche è possibile utilizzare diverse notazioni, tra cui:

- Il comando **let “...”**
- Le doppie parentesi tonde **(( ... ))**
- Le parentesi quadre **[ ... ]**
- Il costrutto **expr**, poco efficiente che valuta il valore di una espressione richiamando una nuova shell.

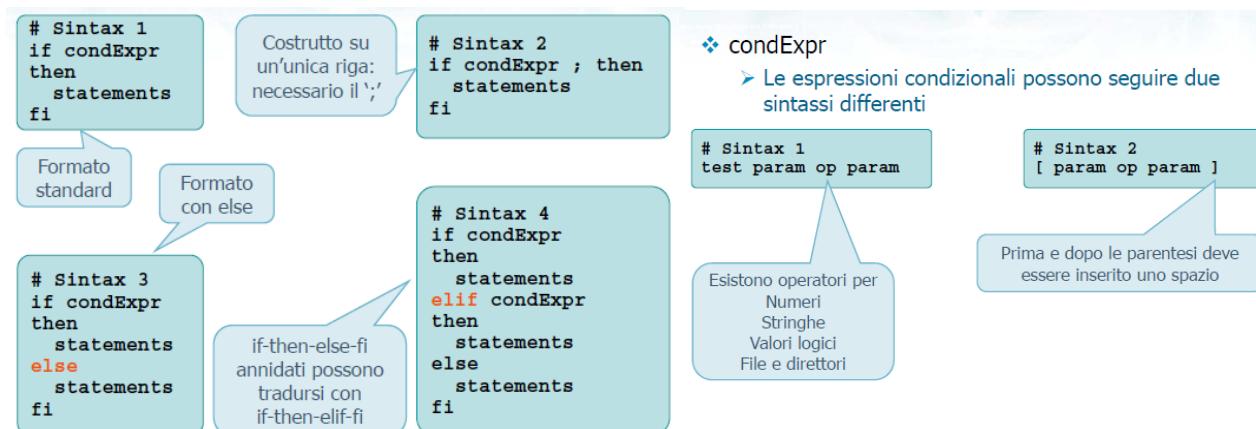
Viene usato praticamente sempre let, ma possono trovarsi script con le altre notazioni.



## Costrutto condizionale if-then-fi

Il costrutto condizionale **if-then-fi** verifica se lo stato di uscita (exit status) di una sequenza di comandi è uguale a 0 (cioè **vero**). In caso affermativo esegue uno o più comandi.

Il costrutto può essere esteso per includere la condizione **else if-then-else-fi** oppure per effettuare controlli annidati **if-then-...-if-then-...-fi-fi** e **if-then-elif-...-fi**

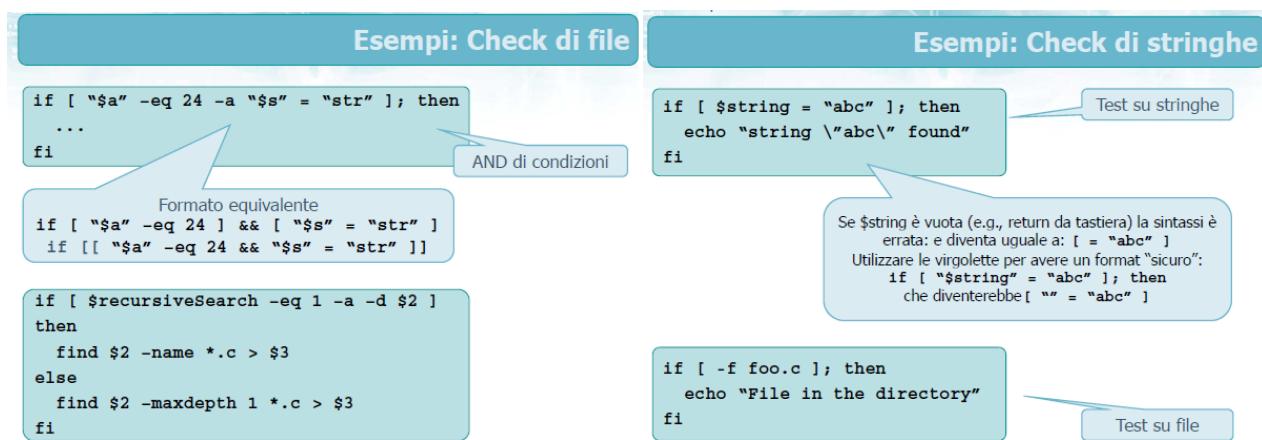
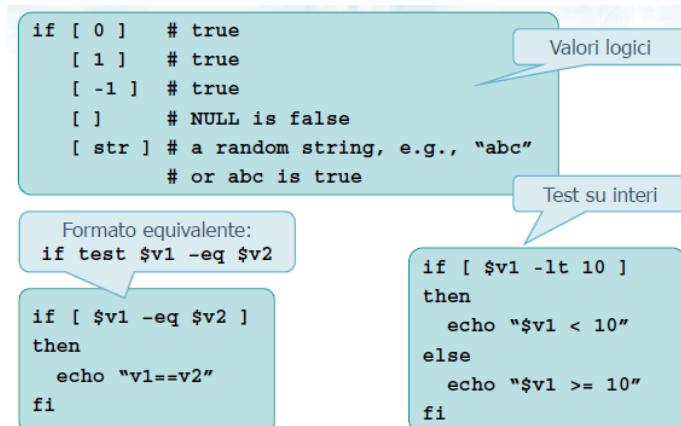


Operatori per numeri		Operatori per file e direttori	
-eq	==	-d	L'argomento è una directory
-ne	!=	-f	L'argomento è una file regolare
-gt	>	-e	L'argomento esiste
-ge	>=	-r	L'argomento ha il permesso di lettura
-lt	<	-w	L'argomento ha il permesso di scrittura
-le	<=	-x	L'argomento ha il permesso di esecuzione
!	! (not)	-s	L'argomento ha dimensione non nulla

Operatori per stringhe		Operatori logici	
=	strcmp	!	NOT (in condizione singola)
!=	!strcmp	-a	AND (in condizione singola)
-n string	non NULL string	-o	OR (in condizione singola)
-z string	NULL (empty) string	&&	AND (in un elenco di condizioni)
			OR (in un elenco di condizioni)

## Esempi



Si nota che conviene aggiungere le "" a \$string perché nel caso in cui essa è nulla, si ottiene una condizione del tipo \_ == "yes" e da errore. Perciò scrivendo "\$string"="abc" si ha che il caso peggiore è "" = "abc" che restituisce falso (e quindi un risultato corretto).

## Costrutto iterativo for-in

Il costrutto **for-in** esegue i comandi specificati, una volta per ogni valore assunto dalla variabile var. L'elenco dei valori [list] può essere indicato in maniera esplicita (tramite elenco) oppure in maniera implicita (comandi di shell, wild-cards, etc.)

```
# Syntax 1  
for var in [list]  
do  
    statements  
done
```

```
# Syntax 2  
for var in [list]; do  
    statements  
done
```

Osservazione: costrutto definito, i.e., itera un numero **predefinito** di volte

### Esempio

#### Esempi: for con elenco esplicito

```
for foo in 1 2 3 4 5 6 7 8 9 10  
do  
    echo $foo  
done
```

Stampa un elenco di numeri

```
for str in foo bar echo charlie tango  
do  
    echo $str  
done
```

Stampa un elenco di stringhe

```
num="2 4 6 9 2.3 5.9"  
for file in $num  
do  
    echo $file  
done
```

Stampa un elenco di numeri utilizzando una variabile (vettoriale, vedere in seguito)

#### Esempi: for e wild-chars

```
n=1  
for i in $* ; do  
    echo "par # $n = $i"  
    let n=n+1  
done
```

Iterazione sui parametri dello script

Visualizza tutti i parametri ricevuti sulla riga di comando

```
for f in $(ls | grep txt); do  
    chmod g+x $f  
done
```

Cambia i privilegi a file specifici

```
for i in $(echo {1..50})  
do  
    echo -n "$i " >> number.txt  
done
```

Genera un file con i numeri da 1 a 50 sulla stessa riga, separati da uno spazio e li scrive in number.txt

Osservazione: la ridirezione è a livello di echo; '>' genererebbe un nuovo file a ogni iterazione

## Costrutto iterativo while-do-done

Questo costrutto effettua iterazione indefinita (il numero di iterazioni è ignoto): si itera sino a quando la condizione è vera e si termina il ciclo quando la condizione è falsa.

```
# Syntax 1  
  
while [ cond ]  
do  
    statements  
done
```

```
# Syntax 2  
  
while [ cond ] ; do  
    statements  
done
```

## Esempio

### Esempio: Script completo

```
#!/bin/bash

limit=10
var=0
while [ "$var" -lt "$limit" ]
do
    echo "Here we go again $var"
    let var=var+1
done

exit 0
```

Visualizza 10 volte il messaggio indicato

### Esempio: Script completo

```
#!/bin/bash

echo "Enter password: "
read myPass

while [ "$myPass" != "secret" ]; do
    echo "Sorry. Try again."
    read myPass
done

exit 0
```

Visualizza il messaggio indicato sino all'introduzione della stringa corretta

### Esempio: Script completo

```
#!/bin/bash

n=1
while read row
do
    echo "Row $n: $row"
    let n=n+1
done < in.txt > out.txt

exit 0
```

Lettura di righe intere in 1 variabile (sino al new-line)

Dato che il costrutto while-do-done è considerato come unico, la redirezione (di I/O) deve essere fatta al termine del costrutto

Scrivere  
echo ... > out.txt  
implicherebbe sovrascrivere il file tutte le volte. Al limite usare  
echo ... >> file.txt

Scrivere  
while read row < in.txt  
implicherebbe rileggere sempre la stessa riga del file

Nomi dei file costanti.  
Possibile l'uso di parametric o variabili  
:\$1 > \$2

L'esempio a sinistra mostra che il costrutto vuole la ridirezione al termine. Esso leggerà da in.txt e stamperà in out.txt.

## Altri esempi

### Esercizi

- ❖ Scrivere uno script bash in grado di
  - Ricevere due interi n1 e n2 sulla riga di comando oppure di leggerli da tastiera se non sono presenti sulla riga di comando
  - Visualizzare una matrice di n1 righe e n2 colonne di valori interi crescenti a partire dal valore 0
  - Esempio
 

```
> ./myScript 3 4
0 1 2 3
4 5 6 7
8 9 10 11
```

```
#!/bin/bash
if [ $# -lt 2 ] ; then
    echo -n "Values: "
    read n1 n2
else
    n1=$1
    n2=$2
fi
n=0
```

Lettura dati in ingresso

```
r=0
while [ $r -lt $n1 ] ; do
    c=0
    while [ $c -lt $n2 ] ; do
        echo -n "$n "
        let n=n+1
        let c=c+1
    done
    let r=r+1
    echo
done
exit 0
```

Doppio ciclo di visualizzazione

L'echo finale serve per andare a capo.

## Break, continue e ‘:’

I costrutti **break** e **continue** hanno comportamenti standard con i cicli for e while, cioè forniscono un’uscita non struttura dal ciclo (break) oppure il passaggio all’iterazione successiva (continue). Il carattere ‘:’ invece può essere utilizzato per creare “istruzioni nulle”:

```
if [ -d "$file" ]; then
:
#Empty instruction
fi
```

Il carattere : viene anche utilizzato come condizione vera. Ad esempio **while :** corrisponde a **while [ 0 ]**

## Vettori

In bash è possibile utilizzare variabili vettoriali mono-dimensionali. Ogni variabile può essere definita come vettoriale e la dichiarazione esplicita non è necessaria (ma possibile con il costrutto **declare**). Non esiste limite alla dimensione di un vettore e neanche alcun vincolo sull’utilizzo di indici contigui. Gli indici partono usualmente da 0 esattamente come in C e **non** sono associative (no hashing). Si supponga che **name** sia il nome di un vettore.

### Definizione

- Elemento per elemento: **name[index]=“value”**
- Tramite elenco di valori: **name=(lista di valori separate da spazi)**

Un nuovo elemento può essere aggiunto in qualsiasi momento.

### Riferimento

- Al singolo elemento:  **\${name[index]}**
  - A tutti gli elementi:  **\${name[\*]}** (oppure @).
  - Numero di elementi:  **\${#name[\*]}**
  - Lunghezza dell’elemento index (numero di caratteri):  **\${#name[index]}**
- L’utilizzo delle graffe è obbligatorio.

### Distribuzione

Il costrutto **unset** può distruggere vettori o elementi di vettori.

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

**Esempi: Uso di vettori**

Init come lista e stampa

```
> vet=(1 2 5 ciao)
> echo ${vet[0]}
1
> echo ${vet[*]}
1 2 5 ciao
> echo ${vet[1-2]}
2 5
> vet[4]=bye
> echo ${vet[*]}
1 2 5 ciao bye
```

Eliminazione

```
> unset vet[0]
> echo ${vet[*]}
2 5 ciao bye
> unset vet
> echo ${vet[*]}
```

Indici non contigui

```
> vet[5]=50
> vet[10]=100
> echo ${vet[*]}
50 100
```

## Esercizio

- ❖ Realizzare uno script in grado di
  - Leggere un insieme indefinito di numeri
  - Terminare la fase di lettura quando viene introdotto il valore 0
  - Visualizzare i valori in ordine inverso
- Esempio
  - Input n1: 10
  - ...
  - Input n10: 100
  - Input n11: 0
  - Output: 100 ... 10

## Soluzione

```
#!/bin/bash
i=0
while [ 0 ] ; do
    echo -n "Input $i: "
    read v
    if [ "$v" -eq "0" ] ; then
        break;
    fi
    vet[$i]=$v
    let i=i+1
done
```

Anche :

```
echo
let i=i-1
while [ "$i" -ge "0" ]
do
    echo "Output $i: ${vet[$i]}"
    let i=i-1
done
exit 0
```

Input

echo \${vet[\*]} visualizzerebbe gli elementi nello stesso ordine e separati da uno spazio

Output  
in ordine inverso

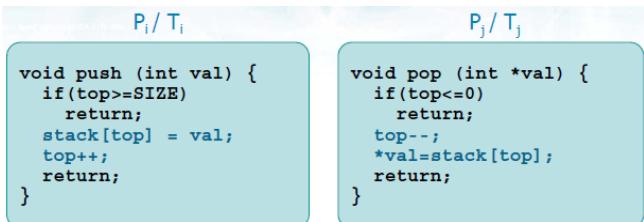
## 22. Le sezioni critiche ☺

L'ambiente di sviluppo prevede programmazione **concorrente** (tramite Processi o Thread) e molto spesso tali entità devono essere **cooperanti**. Le problematiche sono molteplici: necessità di manipolare dati condivisi, possibilità di **corse critiche** (cioè che il risultato dipende dall'ordine di esecuzione) e possono esistere tratti di codice **non rientrati** (non interrompibile). La strategia è quella di **sincronizzare** opportunamente processi e thread, rendendo i programmi indipendenti dalla loro velocità relativa.

Orario	Persona A	Persona B
10.00	Frigo? Finito latte.	
10.05	Va al negozio.	
10.10	Arriva al negozio.	Frigo? Finito latte.
10.15	Acquista il latte.	Va al negozio.
10.20	Arriva a casa.	Arriva al negozio.
10.25	Ritira il latte in frigo.	Acquista il latte.
10.30		Arriva a casa.
10.35		Ritira il latte in frigo.

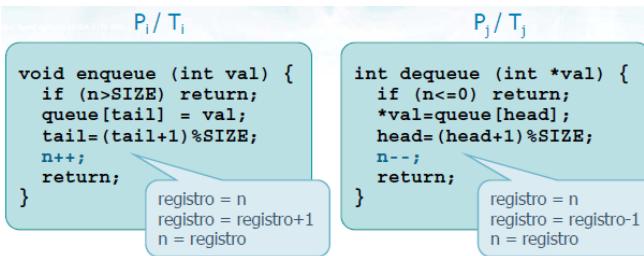
Il problema "Too much milk problem" è molto semplice, poiché due persone notano in tempi diversi il frigo vuoto e dunque comprano entrambi il latte.

Tale problema dal punto di vista informatico può essere visto come un sistema **LIFO – Stack**.



Le funzioni **push** e **pop** agiscono sulla stessa estremità dello stack e la variabile **top** è condivisa. Se si effettua prima **top++** e poi **top--** possono esserci dei problemi ed è anche possibile perdere o sovrascrivere una **push** oppure fare una **pop** di un valore inesistente.

Un'altra alternativa è **FIFO – queue – Buffer circolare**



Le funzioni **enqueue** e **dequeue** agiscono su estremità "diverse" della coda usando variabili diverse **tail** e **head** ma la variabile **n** è comunque condivisa ed è possibile perdere un incremento o decremento.

Una sezione critica (**SC**) o regione critica (**RC**) è dunque una sezione di codice, comune a più processi (o thread), nella quale i processi (o thread) possono accedere (in lettura e scrittura) a oggetti comuni, ovvero una SC o RC è: una sezione di codice nella quale più processi (o thread) competono per l'uso (in lettura e **scrittura**) di risorse comuni (e.g., dati condivisi).

Le corse critiche potrebbero essere evitate se:

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

Occorre quindi stabilire un **protocollo** di accesso per forzare la **mutua esclusione** per ciascuna SC, ovvero per entrare nella propria SC un processo esegue un codice di **prenotazione** che deve essere bloccante se la SC è utilizzata da un altro processo. All'uscita della SC un processo esegue il codice di **rilascio** della regione occupata, sbloccando eventuali P o T in attesa.

$P_i / T_i$	$P_j / T_j$
<pre>while (TRUE) {     ...     sezione d'ingresso <b>SC</b>     sezione d'uscita     ...     sezione non critica }</pre>	<pre>while (TRUE) {     ...     sezione d'ingresso <b>SC</b>     sezione d'uscita     ...     sezione non critica }</pre>

Ogni SC è protetto da una sezione di ingresso (di prenotazione o di prologo) e da una sezione di uscita (di rilascio). Le sezioni non critiche non devono essere protette.

## Condizioni 😊

Ogni soluzione al problema delle SC deve soddisfare i seguenti requisiti:

- **Mutua esclusione:** un solo P (o T) alla volta deve ottenere l'accesso alla regione critica
- **Progresso:** se nessun P (o T) si trova nella SC e un P (o T) desidera entrarci, deve poterlo fare in un tempo definito permettendo solo ai P (o T) in fase di prenotazione di partecipare alla selezione. Nessun P (o T) fuori dalla SC può bloccare altri P (o T). In altre parole, occorre evitare **deadlock** tra P (o T)
- **Attesa definita:** deve esistere un numero definito di volte per cui altri P (o T) riescano ad accedere alla SC prima che un P (o T) specifico e che ha fatto una richiesta di accesso possa farlo. Ovvero, occorre evitare **starvation** di P (o T)
- **Ogni soluzione dovrebbe essere simmetrica:** la selezione di chi deve accedere alla SC non dovrebbe dipendere dalla priorità/velocità relativa tra P (o T)

Le SC ammettono soluzioni:

- **Software:** la correttezza risiede nella logica dell'algoritmo così come formulato dal programmatore. Sono impegnative perché non sono generalizzabili a n utenti e molto spesso inducono ad errori
- **Hardware:** la soluzione si basa su soluzioni architettonali particolari (e.g., istruzioni macchina atomiche). Sono più facilmente estendibili delle precedenti ma si suppone che l'utente le utilizzi e dunque hanno dei vincoli.
- **Ad-Hoc:** il sistema operativo fornisce funzioni e strutture dati e il programmatore le utilizza in maniera opportuna (**Semaforo**). Questa è una soluzione intermedia delle precedenti.

## 23. Soluzioni software ☺

Le soluzioni software mettono tutto a carico dell'utente che deve forzare la mutua esclusione. Si basa sull'utilizzo di **variabili globali** che vengono condivisi e sono facili da implementare. Verrà analizzato il caso con due soli P (o T) denominati  $P_i$  ( $T_i$ ) e  $P_j$  ( $T_j$ ) e di fatto dato  $i$  allora  $j=1-i$  e viceversa (dato uno si trova l'altro) ovvero se  $i=0$  allora  $j=1-i=1$  e viceversa. Casi con più di due P (o T) sono complessi in quanto le procedure analizzate non sono facilmente generalizzabili.

Inoltre, supporremo esistano i valori logici TRUE (1) e FALSE (0).

### Mutua esclusione: Soluzione 1 ☺

La variabile globale utilizzata è:

```
int flag[2] = {FALSE, FALSE};
```

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

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

Si nota in maniera molto semplice che i due P (o T) sono **simmetrici** (stesso codice, solo indici scambiati). Si ha un protocollo di accesso e di rilascio della sezione critica. La variabile globale è un vettore di due valori eventualmente impostati entrambi su false. Il primo  $P_i$ ( $T_i$ ) nota che la variabile è false, dunque esce dal ciclo e imposta il flag a 1 andando ad eseguire la sezione critica e infine rilasciandola.

La soluzione **non** funziona perché la mutua esclusione non è assicurata perché se i processi effettuano contemporaneamente il ciclo while, usciranno entrambi dal ciclo.

Inoltre, la variabile di flag è detta di **lock** (che serve a proteggere la SC) sulla quale vi è un'**attesa attiva** (**busy waiting**) o **spin lock** perché entrambi effettuano un ciclo while di "attesa" che utilizza la CPU (per questo è detta attiva).

### Mutua esclusione: Soluzione 2 ☺

La variabile globale utilizzata è:

```
int flag[2] = {FALSE, FALSE};
```

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

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

La soluzione 2 tenta di risolvere il problema della soluzione 1 con un approccio simmetrico: invece di testare e settare il flag prima effettua il set e poi il test. Neanche questa soluzione funziona perché può non esserci progresso, ovvero si può presentare **deadlock**. Come la soluzione 1 presenta busy waiting con spin lock.

## Mutua esclusione: Soluzione 3 ☺

La variabile globale utilizzata è:

```
int turn = i;
```

**P<sub>i</sub> / T<sub>i</sub>**

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

**P<sub>j</sub> / T<sub>j</sub>**

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

In questo caso si utilizza un'unica variabile di flag detta variabile di **turno**. All'inizio il turno è di **i**, perciò se il turno non è di **i** si aspetta (e stessa cosa fa **j**). Chi ha il turno entra, svolge la SC e passa il turno all'altro processo.

In questo caso l'attesa **non è** definita perché il processo potrebbe non voler utilizzare la SC e non settterà il flag e dunque l'altro processo resterebbe in attesa e provocherebbe **starvation**.

## Mutua esclusione: Soluzione 4 ☺

Le variabili globali utilizzate sono:

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

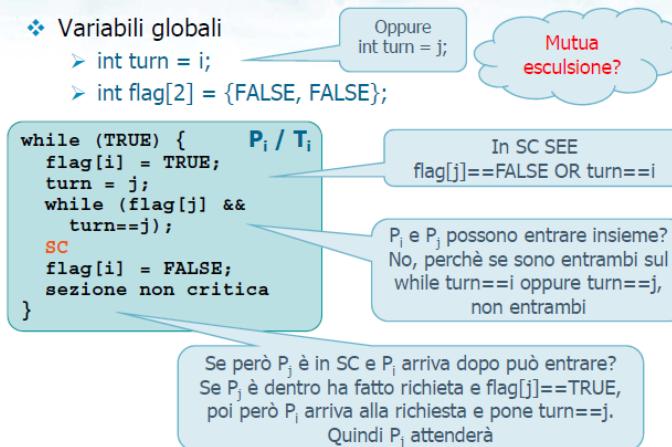
**P<sub>i</sub> / T<sub>i</sub>**

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

**P<sub>j</sub> / T<sub>j</sub>**

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

Quest'ultima soluzione è **valida**. Utilizza una combinazione delle soluzioni precedenti. Si effettua un ciclo infinito in cui ci si prenota e si da il turno all'altro processo e si controlla lo stato dell'altro. Se anche l'altro processo voleva entrare ed il turno è suo (se l'altro non ha ripassato il turno) si aspetta. Se invece una delle due variabili è falsa si esce dall'attesa e si svolge la SC rilasciando poi il codice. Si analizzano ora tutte le condizioni.



Si entra in sezione critica quando il while termina, che termina soltanto se una delle due condizioni è falsa. P<sub>i</sub> e P<sub>j</sub> anche se arrivano sulla riga del while **non** possono entrare entrambi perché anche se entrambi avessero il flag a TRUE, il turno sarà solo di uno dei due. Se uno è già dentro e l'altro cerca di entrare non riesce, perché impostando il valore a TRUE passerà il turno all'altro processo.



L'unico punto in cui un processo può rimanere bloccato nel while. Supponendo che P<sub>i</sub> sia sul ciclo while e P<sub>j</sub> non voglia entrare, ma se non vuole entrare il suo flag è FALSE e dunque entra.

Viceversa se entrambi sono sul ciclo while il turno è di uno dei due e quindi uno dei due entrerà. Se P<sub>i</sub> è sul ciclo e P<sub>j</sub> esce, P<sub>j</sub> mette il flag a FALSE e P<sub>i</sub> entra.

Supponiamo che P<sub>j</sub> sia nel SC e che sia velocissimo ed esce impostando il flag a false ma prima che l'altro processo inizi, P<sub>i</sub> riparte e imposta nuovamente il flag a TRUE. P<sub>j</sub> non riesce a entrare più volte perché mette il turno a i e dunque il turno è dell'altro processo ed entra.

Banale, il codice è simmetrico.

Il P (o T) in attesa è comunque in busy waiting su spin lock perciò permane il problema del consumo della risorsa "CPU time". In generale le soluzioni software al problema delle SC risultano complesse e inefficienti perché l'assegnazione o il controllo di una variabile da parte di un P/T è una operazione "invisibile" agli altri P/T e le operazioni di controllo e modifica non sono "atomiche", quindi si possono avere reazioni al valore presunto di una variabile invece che a quello reale.

## 24. Soluzioni hardware ☺

Si basano sulla creazione di operazioni di tipo **atomico**.

Le soluzioni hardware al problema della SC possono essere classificate come segue:

- Soluzioni per sistemi che **non** permettono il diritto di **prelazione** (portare via una risorsa a qualcun altro) ed è una soluzione relativamente semplice ma poco probabile.
- Soluzioni per sistemi che permettono il diritto di **prelazione** tramite soluzioni basate sulla gestione delle interruzioni o soluzioni basate su una “estensione” delle soluzioni software, ovvero basate su un qualche tipo di lock o istruzione atomica. Tale aspetto è però complicato in presenza di sistemi multi-processore o multi-core.

Un sistema senza diritto di prelazione è tale per cui i P (o T) in esecuzione nel kernel **non** possono essere interrotti ed il controllo del kernel verrà rilasciato solo quando il P (o T) lo lascerà volontariamente. Se questo fosse il caso nei sistemi **mono** processori non esisterebbe il problema della SC in quanto solo un P (o T) impegna l'unica CPU in un certo momento ma di fatto è una soluzione **non implementabile** sia perché esistono i sistemi multi-processore, sia perché i sistemi senza diritto di prelazione sono pericolosi.

In un sistema con diritto di prelazione un processo in esecuzione in modalità di sistema può essere interrotto, tramite ad esempio l'arrivo di un **interrupt** che sposta il controllo del flusso su un altro processo ed il processo originario verrà terminato in seguito.

Nei sistemi mono processore con diritto di prelazione è possibile risolvere il problema della SC **disabilitando l'interrupt** (e riabilitandola all'uscita) come segue

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

Si osservi che l'interrupt deve poter essere abilitato/disabilitato dal processo stesso

Disabilitare l'interrupt può però essere pericoloso perché un processo potrebbe eseguire tutto il codice (e non solo la SC) prima di riattivare l'interrupt ed inoltre nei sistemi multiprocessore non è chiaro cosa vuol dire disabilitare l'interrupt. Una strategia alternativa è quella di mimare le soluzioni software, utilizzando

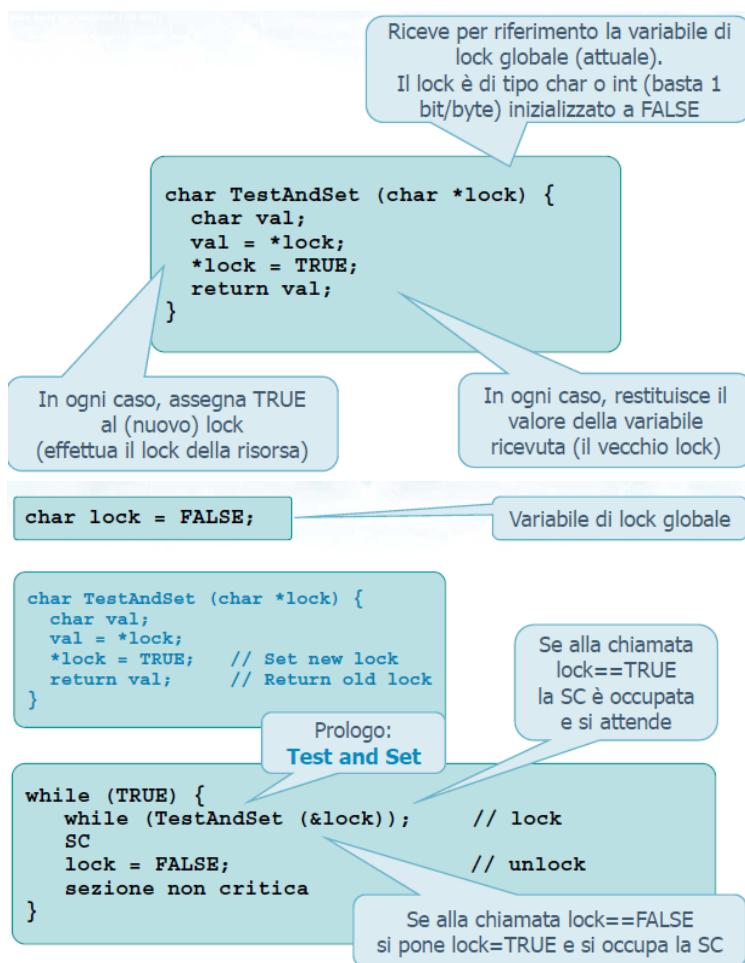
- Lucchetti di protezione, i.e., “**lock**”: si potrebbe utilizzare un lock per ciascuna SC ed il valore del lock permette l'accesso oppure lo vieta, proteggendo così la risorsa da accessi multipli
- Istruzioni indivisibili, i.e., “**atomiche**”: Un'istruzione atomica viene eseguita in un unico “memory cycle” e dunque non può essere interrotta e permette verifica e modifica contestuale di una variabile globale

In altre parole, l'operazione di blocco/sblocco deve essere atomica, cioè **senza verifiche** (non controllo se è sbloccato/bloccato).

Esistono due principali istruzioni atomiche di lock:

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

## Test and set: implementazione e utilizzo ☺



Non fa alcun test, ma agisce come segue: riceve una variabile globale **char \*lock** (ma è di tipo booleana) ricevuta come riferimento perché la modificherà e la funzione utilizza una variabile locale a cui assegna il valore globale di lock. Successivamente setta Lock a true e restituisce val. **Ritorna ciò che riceve ma setta lock a true**. Se riceve falso la mette vera, se riceve vero setta ancora vero.

Il protocollo di accesso è un ciclo while che richiama TestAndSet e se all'inizio lock è false, si passa tale valore, viene restituito false ma la variabile lock diventa true. Visto che si riceve false si può andare in sezione critica e allo stesso tempo la si ha occupata. Quando si termina l'utilizzo la variabile viene impostata false.

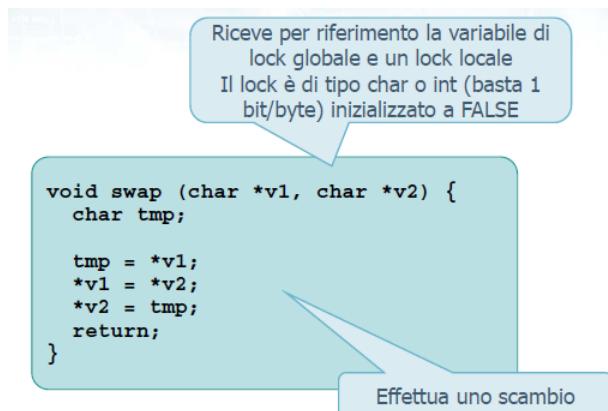
Se invece lock è TRUE si riceve nuovamente true e si resta nel ciclo.

In generale si avranno n variabili di lock per n sezioni critiche (è una sola per ogni

sezione critica e se più thread vogliono accedere guarderanno la stessa variabile di lock).

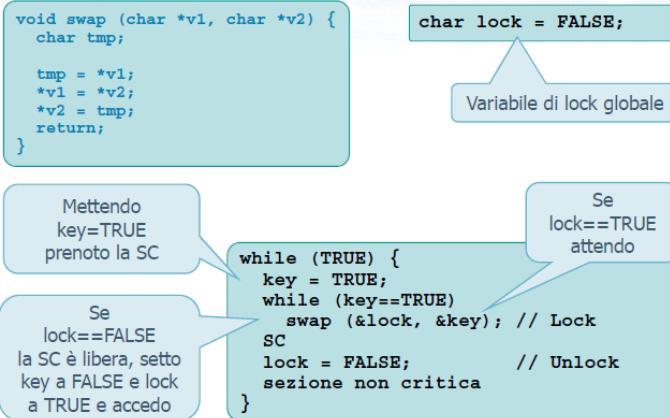
Anche questa soluzione è busy-waiting su spin-lock: consuma cicli mentre attende.

## Swap: implementazione e utilizzo ☺



Standard funzione di swap che riceve due variabili per riferimento e le scambia.

Ciò che cambia è il protocollo di utilizzo.

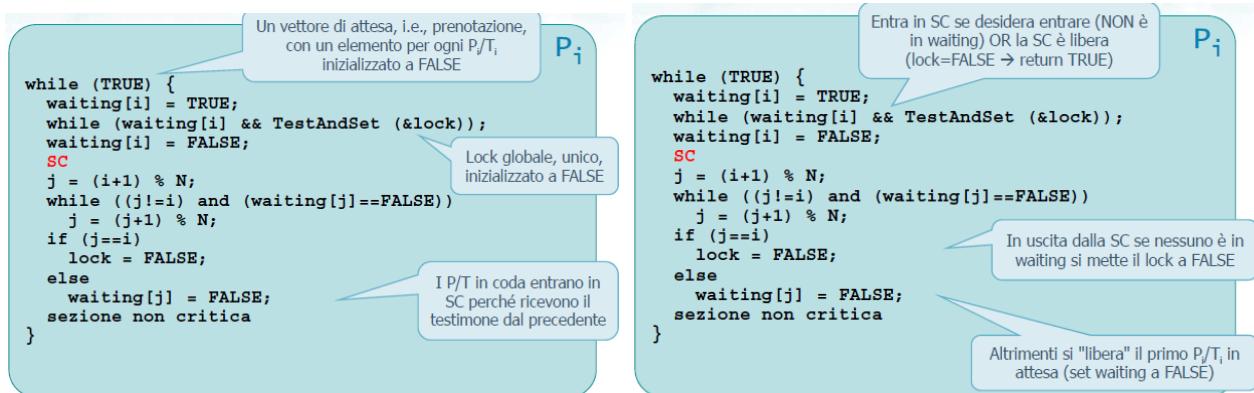


Si hanno tre righe per il protocollo di accesso e una per quella di rilascio. All'inizio il lock è FALSE. Quando si vuole entrare si imposta la propria chiave a TRUE (variabile locale di prenotazione) e finché la chiave è TRUE si fa lo swap tra lock e key. Se si effettua lo swap tra un valore vero e uno falso la key diventa FALSE ma il lock diventa TRUE perciò blocca la SC e visto che Key è falsa esce dal ciclo ed esegue la sezione critica. Alla fine, si rilascia il lock impostandolo a FALSE. Se invece lock è TRUE e key anche, si fa lo swap tra due variabili identiche e dunque si continua a ciclare.

Anche questa soluzione è Busy-waiting su spinlock: consuma cicli mentre attende.

### Mutua esclusione senza starvation ☺

Le tecniche precedenti assicurano la mutua esclusione, il progresso (evitando il deadlock) ma non assicurano l'attesa definita di un processo, ovvero non garantiscono la non starvation e sono simmetriche. Per soddisfare tutti e quattro i criteri occorre estendere le soluzioni precedenti.



Si nota la presenza della variabile **waiting** che è lunga tanto quanto il numero di P (o T) che competono per tale risorsa. Il primo ciclo while è identico al caso di TestAndSet poiché anche se è presente **waiting[i]**, essa è TRUE poiché viene impostata alla riga precedente. Finché entrambe le variabili sono TRUE, il P (o T) rimane in attesa. La presenza della variabile di **waiting** eviterà successivamente la **starvation**, poiché serve nella condizione di uscita in modo da scandire chi si è prenotato e passare il testimone al processo successivo.

Infatti, il P (o T) una volta uscito dal ciclo utilizzerà la SC e imposterà il proprio **waiting** a FALSE (poiché non sta più attendendo). Finita la SC, si partirà da un valore **j** pari a **i+1** ma creato in maniera circolare (% N) in modo da scandire tutti i P (o T) fino a visitarli tutti e tornare a se stessi. Se tutti i P (o T) incontrati sono FALSE si arriva al caso **if(j==i)** poiché si torna a se stessi e si mette **lock = FALSE** come nel caso originario del TestAndSet.

Se invece almeno un valore del vettore **waiting** è TRUE, il while si interromperà perciò **j!=i** e a quel punto si metterà **waiting[j]=FALSE** in modo da far partire tale processo che era in attesa nel primo ciclo while (impostandolo a FALSE la condizione del primo while risulta falsa e quindi esce) e si prenderà il possesso della SC.

Il lock non viene modificato poiché qualcuno lo occupava prima e successivamente lo occupa di nuovo.

## Conclusioni ☺

Vantaggi delle soluzioni hardware:

- Utilizzabili in ambienti multi-processore
- Facilmente estendibili a N processi
- Relativamente semplici da utilizzare dal punto di vista software, cioè dal punto di vista utente
- Simmetriche

Svantaggi delle soluzioni software:

- **Non facili da implementare a livello hardware:** le operazioni atomiche avvengono su variabili globali (qualsiasi)
- **Starvation:** la selezione dei processi in busy-waiting per la SC è arbitraria e gestita dai processi stessi non dal SO
- **Busy waiting su spin-lock:** spreco di risorse ovvero di cicli di CPU nell'attesa. In pratica il busy-waiting è utilizzabile solo quando le attese sono molto brevi ed inoltre si ha **inversione di priorità (priority inversion).**

## Inversione di priorità ☺

Questo è un problema generale dello scheduling.

Supponendo che un processo con bassa priorità occupi una risorsa ed un processo con alta priorità cerchi di occupare la medesima risorsa, ovviamente va in attesa sul lock di protezione e quindi in busy waiting. Dato che la CPU viene dedicata completamente al processo a priorità elevata (che lo spreca in busy waiting) il processo a priorità bassa non viene eseguito e quindi non rilascia la risorsa. In altre parole, la risorsa viene occupata dal processo a minore priorità molto più a lungo del necessario.

Una possibile cura a questo problema consiste nell'utilizzare il **protocollo di ereditarietà della priorità:** un processo in possesso di un lock eredita automaticamente la priorità del processo con priorità maggiore in attesa dello stesso lock.

## 25. I semafori ☺

Le soluzioni software sono complesse da utilizzare dal punto di vista del programmatore mentre quelle hardware sono difficili da realizzare dal punto di vista del progettista hardware. I sistemi operativi forniscono quindi primitive più adatte, denominate semafori (Introdotte da Dijkstra nel 1965) che non si basano su implementazioni con busy waiting e quindi non sprecano risorse.

Un semaforo **S** è una variabile intera condivisa protetta dal sistema operativo (le funzioni che la manipolano sono system call) ed è anche utilizzabile per inviare e ricevere segnali. Le operazioni su S sono sempre eseguite in maniera atomica (garantita dal sistema operativo) ed è impossibile per due processi eseguire operazioni contemporanee sullo stesso semaforo. Una volta che si entra in tali funzioni, la funzione non può essere interrotta.

Tutte le implementazioni dei semafori mettono a disposizione una libreria in cui esistono almeno 4 system call:

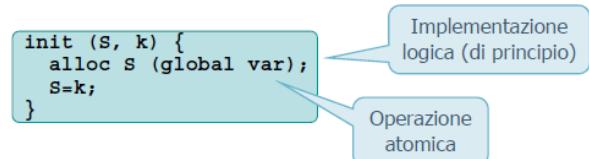
- **init (S, k)**: definisce e inizializza il semaforo S al valore k
- **wait (S)**: permette nella sezione di ingresso di ottenere l'accesso della SC protetta dal semaforo S
- **signal (S)**: permette nella sezione di uscita di uscire dalla SC protetta dal semaforo S
- **destroy (S)**: cancella (libera)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

Queste funzioni non hanno NULLA a che vedere con gli argomenti precedenti.

### init (S, k)

Questa funzione definisce e inizializza il semaforo **S** al valore **k** dove k è di tipo intero. Esistono due tipi di semafori:

- **Semafori binari**: il valore di k è quello del semaforo in un istante qualsiasi è un intero uguale a 0 oppure a 1
- **Semafori con conteggio**: il valore di k è quello del semaforo in un istante qualsiasi è un intero nell'intervallo [0, k]



I codici proposti sono da intendere come **pseudocodice**.

### wait (S)

Equivale alla `sleep()` o `down()`. Se il valore di S è negativo o nullo blocca il processo chiamante (la risorsa non è disponibile) poiché entra in un ciclo while.

Se S è negativo il suo valore assoluto S indica il numero di P (o T) in attesa (nel caso logico S non può mai essere negativo perché se S è nullo ci si ferma sul while e non si prosegue a decrementare).

In ogni caso decrementa il valore di S



Originariamente era denominata P() dall'olandese "probeer te verlagen", i.e., "try to decrease"

## signal (S)

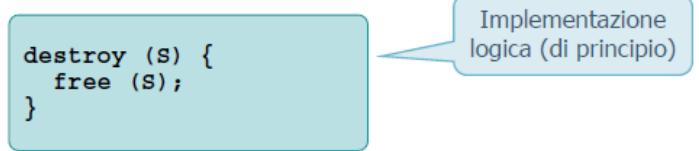
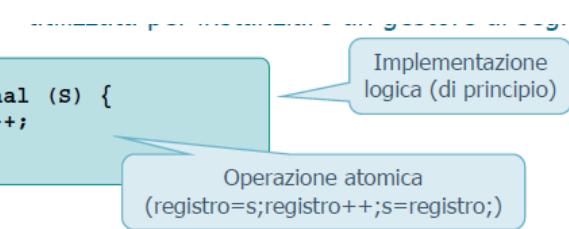
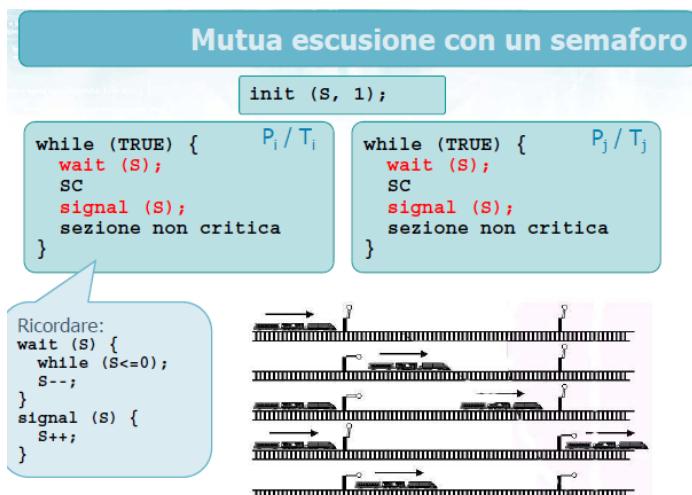
Incrementa la variabile semaforica e se S era negativo o nullo un qualche P (o T) risultava essere bloccato e ora potrà accedere. E' necessario un bilanciamento tra wait e signal, poiché ogni signal sveglia un processo dal ciclo while e gli permette di riportare la variabile a 0.

Originariamente denominata V(), dall'olandese "verhogen", i.e., "to increment". Da non confondere con la system call signal utilizzata per instanziare un gestore di segnali. Nell'implementazione reale bisognerà fare operazioni in più del semplice incremento.

## destroy (S)

Rilascia la memoria occupata dal semaforo S: le implementazioni reali di un semaforo richiedono molto di più di una semplice variabile globale per definire un semaforo. E' presente nelle implementazioni reali, ma spesso non utilizzata negli esempi.

## Mutua esclusione con un semaforo



Si supponga di voler forzare un meccanismo di mutua esclusione con un unico semaforo inizializzato a 1. Il primo a fare la wait potrà passare mentre il successivo aspetterà finché l'altro processo è nella SC e non arriva alla signal.

Si supponga ora che il semaforo binario sia in comune tra più processi.

	P <sub>1</sub> /T <sub>1</sub>	P <sub>2</sub> /T <sub>2</sub>	P <sub>3</sub> /T <sub>3</sub>	S	queue
init (s, 1); ...				1	
wait				0	
SC	wait			0	P <sub>2</sub> /T <sub>2</sub>
	blocked	wait		0	P <sub>2</sub> /T <sub>2</sub> , P <sub>3</sub> /T <sub>3</sub>
			blocked	0	
signal				1	P <sub>2</sub> /T <sub>2</sub> , P <sub>3</sub> /T <sub>3</sub>
	SC			0	P <sub>3</sub> /T <sub>3</sub>
	signal			1	
		SC		0	
		signal		1	

Al più 1 P/T alla volta nella SC

Il semaforo viene inizializzato una sola volta e tutti i processi fanno la wait e la signal (il codice è identico a prima). La wait cambia il valore del semaforo da 1 a 0, permettendo al primo processo di entrare nella SC e P2 e P3 rimarranno nella wait ad attendere. Non appena P1 effettua la signal riporta il semaforo a 1 ed uno degli altri due entrerà in SC e riporterà il semaforo a 0. Alla fine avverrà nuovamente una signal e l'ultimo processo potrà partire.

Si supponga ora di avere un semaforo con conteggio (supponendo di avere 2 binari per 3 treni)

	P <sub>1</sub> /T <sub>1</sub>	P <sub>2</sub> /T <sub>2</sub>	P <sub>3</sub> /T <sub>3</sub>	S	queue
init (s, 2);				2	
...				1	
wait (s);				0	
sc di P <sub>i</sub>	wait			0	
signal (s);		SC	wait	0	P <sub>3</sub> /T <sub>3</sub>
			blocked	0	
				1	
			SC	0	
		signal		1	
Al più 2 P/T alla volta nella SC			signal	2	

Il semaforo è inizializzato a 2. La prima wait lo decremente a 1, ma essendoci ancora spazio anche il secondo thread entrerà in sezione critica mentre il terzo rimarrà in attesa. Non appena arriverà la signal del primo processo il terzo potrà partire. Al termine le signal degli altri due thread riporteranno il contatore a 2.

## ESEMPI

### Uso dei semafori: Esempio 1

- Ottenere uno specifico ordine di esecuzione
  - P<sub>i</sub> esegue la sua SC prima di P<sub>j</sub>
  - SC di P<sub>i</sub> va eseguita prima della SC di P<sub>j</sub>



init (s, 0);

SC di P <sub>i</sub> , signal (s);	P <sub>i</sub> / T <sub>i</sub>	P <sub>j</sub> / T <sub>j</sub>
---------------------------------------	---------------------------------	---------------------------------

### Uso dei semafori: Esempio 2

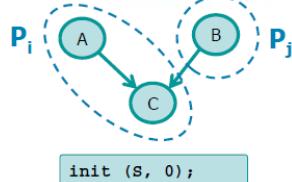
- Sincronizzare due processi P<sub>i</sub> e P<sub>j</sub> in modo che
  - P<sub>i</sub> attenda P<sub>j</sub> in un preciso punto
  - P<sub>j</sub> attenda P<sub>i</sub> in un preciso punto

init (s<sub>1</sub>, 0);  
init (s<sub>2</sub>, 0);

while (TRUE) { ... signal (s <sub>1</sub> ); ... wait (s <sub>2</sub> ); } ...	P <sub>i</sub> / T <sub>i</sub>	while (TRUE) { ... wait (s <sub>1</sub> ); ... signal (s <sub>2</sub> ); } ...	P <sub>j</sub> / T <sub>j</sub>
--	---------------------------------	--	---------------------------------

### Uso dei semafori: Esempio 3

- Ottenere il seguente grafo di precedenza



init (s, 0);

A wait (s); C	P <sub>i</sub> / T <sub>i</sub>	B signal (s);	P <sub>j</sub> / T <sub>j</sub>
---------------------	---------------------------------	------------------	---------------------------------

Si vuole eseguire P<sub>i</sub> prima di P<sub>j</sub>. Si inizializza il semaforo a 0, così se P<sub>j</sub> tenterà di effettuare una wait rimarrà bloccato fin quando P<sub>i</sub> effettuerà la SC e farà una signal (mettendo il semaforo a 1).

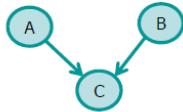
Se si vuole sincronizzare due processi si inizializzano due processi servono due semafori S1 ed S2 entrambi inizializzati a 0. I due processi hanno un ciclo infinito.

Pi fa una signal su S1 mentre l'altro fa una wait su S1 e viceversa. Vuol dire che entrambi i processi partono ma se Pi arriva alla signal prima e finisce prima di Pj si ferma sulla wait di S2. Se Pj è più lento arriverà dopo e perciò verrà subito svegliato dalla wait(S1) e una volta arrivato alla signal(S2) sveglierà Pi.

Nell'esempio 3 Pi è l'insieme di due nodi. Di fatto Pi deve effettuare sia A che C ma prima di fare C deve aspettare e sincronizzarsi con Pj, perciò farà una wait e Pj effettuerà B e chiamerà una signal.

### Uso dei semafori: Esempio 3

- Ottenere il seguente grafo di precedenza



`init (S, 0);`

A  
`signal (S);`

C  
`wait (S);  
wait (S);  
C`

B  
`signal (S);`

Nel caso in cui ognuno è un nodo a se stante. A farà ciò che deve perché non deve aspettare nessuno e successivamente farà una signal, B farà la stessa cosa. C invece farà due wait e successivamente effettuerà C.

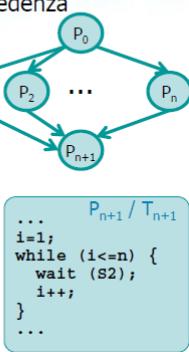
### Uso dei semafori: Esempio 4

- Ottenere il seguente grafo di precedenza

Costrutto cobegin-coend  
(begin-end concorrente)

`init (S1, 0);  
init (S2, 0);`

Osservazione:  
nessuno dei processi  
concorrenti è ciclico



`...  
P0 / T0  
i=1  
while (i<=n) {  
 signal (S1);  
 i++;  
}  
...`

`...  
P1 / T1  
wait (S1);  
...  
signal (S2);  
...`

`...  
Pn1 / Tn1  
i=1;  
while (i<=n) {  
 wait (S2);  
 i++;  
}  
...`

Questo è il caso più generale perché P0 deve partire prima, sveglierà tutti gli altri che andranno in parallelo e una volta terminati tutti sveglieranno P(n+1).

Tutti i processi intermedi sono rappresentati nel codice centrale.

P0 farà ciò che deve e con un ciclo while effettuerà n signal che sveglieranno gli n processi. Ognuno di questi processi farà una signal e dualmente l'ultimo processo farà un ciclo while in cui aspetterà tutti gli n processi.

## Errori nell'uso dei semafori

### Errori nell'uso dei semafori: Esempio 1

Solo un P (tra N)  
in SC

`init (S, 1);`

`P1 / T1`

```
while (TRUE) {  
    ...  
    signal (S);  
    SC1  
    wait (S);  
    ...  
}
```

Entra nella SC e fa  
entrare anche altri 2  
processi

`P1 / T1`

```
while (TRUE) {  
    ...  
    wait (S);  
    SC2  
    wait (S);  
    ...  
}
```

La wait in "eccesso"  
blocca tutti i processi

`P1 / T1`

```
while (TRUE) {  
    ...  
    signal (S);  
    SC3  
    signal (S);  
    ...  
}
```

La signal in "eccesso"  
fa entrare tutti

### Errori nell'uso dei semafori: Esempio 2

Acquisizione di  
due risorse

`init (S, 1);  
init (Q, 1);`

`P1 / T1`

```
while (TRUE) {  
    ...  
    wait (S);  
    ... Use S  
    wait (Q);  
    ... Use S and Q  
    signal (Q);  
    signal (S);  
    ...  
}
```

Accede all'HD e poi al DVD

`P2 / T2`

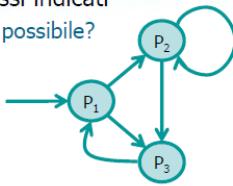
```
while (TRUE) {  
    ...  
    wait (Q);  
    ... Use Q  
    wait (S);  
    ... Use Q and S  
    signal (S);  
    signal (Q);  
    ...  
}
```

Accede al DVD e poi all'HD

## ESERCIZIO

### Soluzione

- ❖ Siano dati i semafori e i processi indicati  
➤ Quale ordine di esecuzione è possibile?



```
init (S1, 1);
init (S2, 0);
```

```
...  
while (1) {  
    wait (S1);  
    SC di P1;  
    signal (S2);  
}  
...
```

```
...  
while (1) {  
    wait (S2);  
    SC di P2;  
    signal (S2);  
}  
...
```

```
...  
while (1) {  
    wait (S2);  
    SC di P3;  
    signal (S1);  
}  
...
```

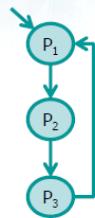
Se P1 parte ed è più veloce passerà subito dalla wait, mentre P2 e P3 dovranno entrambi aspettare perché S2 è inizializzato a 0.

Quando P1 terminerà farà una signal su S2 che potrà svegliare P2 o P3.

Supponendo venga svegliato P2 alla fine farà una signal di S2 perciò o si autosveglia oppure sveglierà P3. P3 una volta che è partito farà una sezione critica e sveglierà P1.

### Soluzione

- ❖ Si realizzi mediante semafori il seguente grafo di precedenza  
➤ Tutti i processi devono essere ciclici



```
init (S1, 1);
init (S2, 0);
init (S3, 0);
```

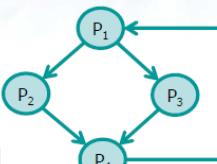
```
...  
while (1) {  
    wait (S1);  
    SC di P1;  
    signal (S2);  
}  
...
```

```
...  
while (1) {  
    wait (S2);  
    SC di P2;  
    signal (S3);  
}  
...
```

```
...  
while (1) {  
    wait (S3);  
    SC di P3;  
    signal (S1);  
}  
...
```

Processi ciclici: se i processi sono ciclici significa che in qualche modo si autosincronizzano e chi li ha creati li **crea un'unica volta** e poi si sincronizzano per tutta la loro durata.

Si utilizzano 3 semafori, ciascun processo ha un semaforo. P1 aspetta su S1 (passerà perché S1 è inizializzato a 1) mentre gli altri aspetteranno.



- ❖ Si realizzi mediante semafori il seguente grafo di precedenza  
➤ Tutti i processi devono essere ciclici

```
init (S1, 1);
init (S2, 0);
init (S3, 0);
init (S4, 0);
```

```
while (1) {  
    P1  
    wait (S1);  
    SC di P1;  
    signal (S2);  
    signal (S3);  
}
```

```
P2  
while (1) {  
    wait (S2);  
    SC di P2;  
    signal (S4);  
}
```

```
P3  
while (1) {  
    wait (S3);  
    SC di P3;  
    signal (S4);  
}
```

```
P4  
while (1) {  
    wait (S4);  
    wait (S4);  
    SC di P4;  
    signal (S1);  
}
```

La differenza adesso è che P2 e P3 sono in parallelo. La soluzione con 3 semafori è errata perché P2 e P3 aspettano sullo stesso semaforo e questo è un problema, perché se P2 è molto veloce potrebbe svegliarsi due volte di fila (visto che P1 fa la signal sullo stesso semaforo due volte) ed è un problema perché i processi sono CICLICI, e di conseguenza P3 potrebbe non svegliarsi mai, perciò ciascuno deve aspettare sul proprio semaforo.

### Soluzione errata

- ❖ Si realizzi mediante semafori il seguente grafo di precedenza  
➤ Tutti i processi devono essere ciclici

```
init (S1, 1);
init (S2, 0);
init (S3, 0);
```

```
P1  
while (1) {  
    wait (S1);  
    SC di P1;  
    signal (S2);  
}
```

```
P2  
while (1) {  
    wait (S2);  
    SC di P2;  
    signal (S3);  
}
```

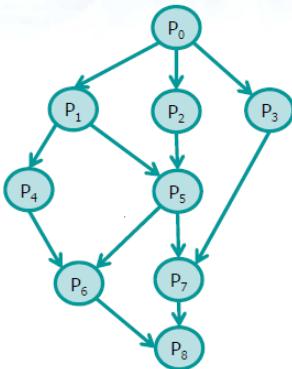
```
P3  
while (1) {  
    wait (S2);  
    SC di P3;  
    signal (S2);  
}
```

```
P4  
while (1) {  
    wait (S3);  
    wait (S3);  
    SC di P4;  
    signal (S1);  
}
```

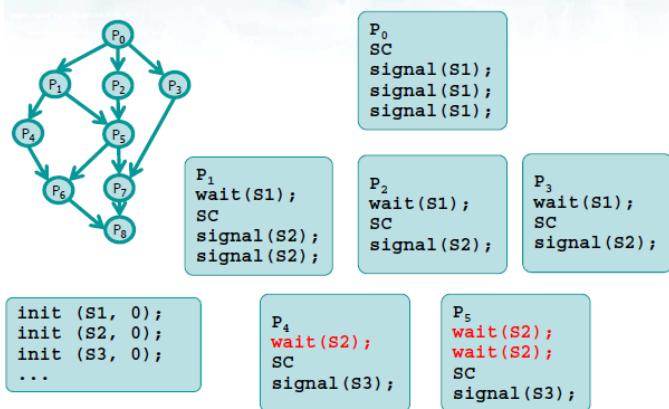
## Esercizio

- ❖ Si realizzi mediante semafori il seguente grafo di precedenza

➤ In processi **non** sono ciclici



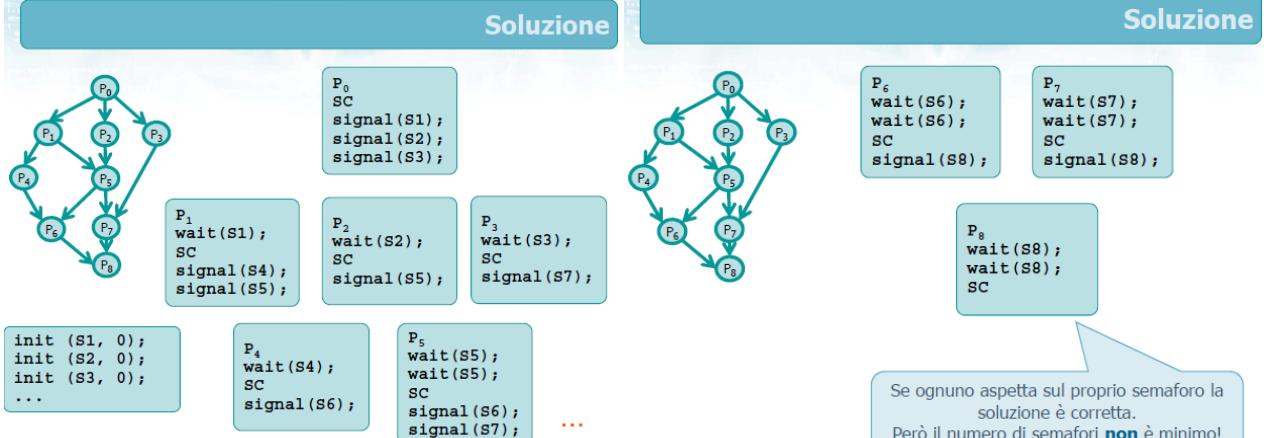
## Soluzione errata



**Soluzione errata:** qui si può usare lo stesso semaforo per svegliare P1, P2, P3 perché NON SONO CICLICI (non hanno il while(1)).

Entrambi fanno la propria sezione critica, poi P1 fa due signal su S2 perché anche P4 e P5 utilizzano lo stesso segnale. Anche P2 sveglia P5 ma qui c'è l'errore perché quello di prima non è vero, visto che le due signal di P1 potrebbero svegliare entrambe P5 ed inoltre c'è una precedenza P2 P4 (che non esiste), cioè P2 potrebbe svegliare P4.

## Soluzione

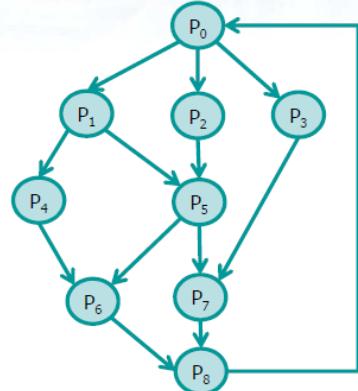


Se ognuno aspetta sul proprio semaforo la soluzione è corretta.  
Però il numero di semafori **non** è minimo!

## ESERCIZIO DA SVOLGERE

### Esercizio

- ❖ Si realizzi mediante semafori il seguente grafo di precedenza
  - Versione A: I processi **non** sono ciclici **ma** il numero di semafori utilizzato sia minimimo
  - Versione B: I processi sono ciclici



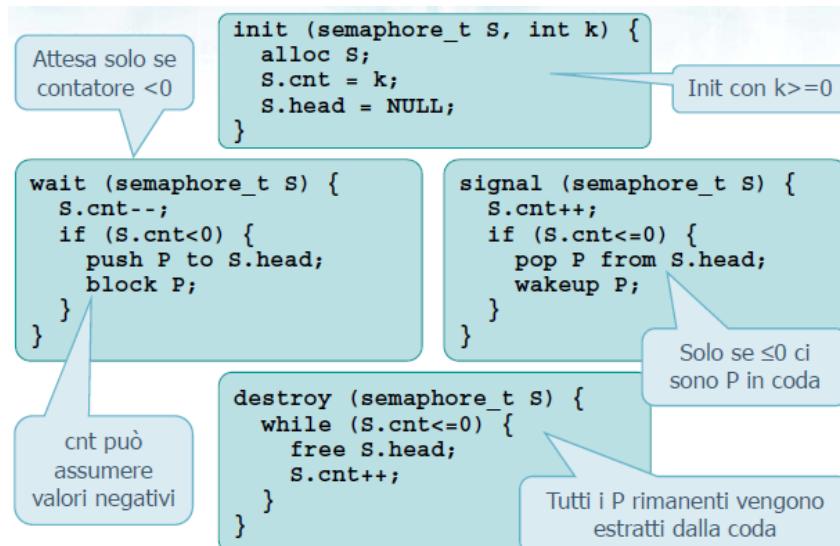
### Implementazione di un semaforo

I semafori vanno implementati senza ricorrere all'attesa attiva **busy waiting (spin-lock)**. Si definisce un semaforo come una struttura C munita di un contatore e di una lista (coda) di processi.

```

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

Dunque le funzioni precedentemente introdotte diventano:



La init è molto simile alla precedente perché alloca il semaforo e inizializza i parametri.

La controparte è la destroy nella quale si deve fare una free esterna del semaforo (non riportata in foto) ma prima di farla ci si occupa della coda di processi nel semaforo. Se il semaforo ha conteggio  $\leq 0$  effettua la free di un processo della lista e incrementa il contatore.

La wait effettua un decremento del contatore (dunque può diventare negativo) e il processo viene inserito nella coda del semaforo e il processo viene bloccato nel senso che va in coda di wait.

La signal effettua un incremento del contatore e se il contatore è  $\leq 0$  si effettua la pop del primo in lista, risvegliando tale processo.

L'implementazione reale permette a un semaforo di avere **valori negativi** ed il suo valore assoluto indica il numero di processi in coda sul semaforo. La coda può essere implementata con un puntatore nel Process Control Block (PCB) dei processi, può soddisfare le politiche che lo scheduler desidera (e.g., FIFO) ed ha un comportamento indipendente dai processi in attesa.

Esistono diverse implementazioni

- **Semafori tramite pipe**
- **Semafori POSIX**
- Semafori Linux
- Pthread: Mutex (Mutua esclusione) e Condition Variable (Variabili condizionali)

System call in Pthread:

```
pthread_cond_init
pthread_cond_wait
pthread_cond_signal
pthread_cond_broadcast
pthread_cond_destroy
```

Si osservi che i semafori sono **oggetti globali** (si veda la `sem_init`) e non appartengono a un processo particolare.

**Verranno introdotti solo quelli POSIX e tramite pipe.**

### Semafori tramite pipe ☺

Data una pipe

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



Le funzioni che seguono sono wrapper delle funzioni read/write/init già viste per le pipe.

### semaphoreInit (S) ☺

```
#include <unistd.h>
void semaphoreInit (int *s) {
    if (pipe (*s) == -1) {
        printf ("Error");
        exit (-1);
    }
    return;
}
```

Inizializza il semaforo S e la variabile S va definita come variabile globale. Ad es. `int S[2];` o `int *S=malloc(2*sizeof(int));`

## semaphoreSignal (s) ☺

```
#include <unistd.h>

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

Riceve il puntatore al vettore S di dimensione 2 e si scrive (ad esempio il carattere X) un solo carattere e visto che si tratta di una signal si effettuerà una **write** (in posizione 1, read in posizione 0) nell'estremo 1 della variabile puntatore ctr di un solo carattere. Si suppone di non eccedere la capacità massima della pipe.

## semaphoreWait (s) ☺

```
#include <unistd.h>

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

La wait non è altro che una read effettuata dall'estremo 0 e se il carattere non è stato ancora scritto **si aspetta** perché la read è bloccante.

## ESEMPIO

### Esempio

```
int main() {
    int S[2];
    pid_t pid;
    semaphoreInit (S);
    pid = fork();
    // Check for correctness
    if (pid == 0) { // child
        semaphoreWait (S);
        printf("Wait done.\n");
    } else { // parent
        printf("Sleep 3s.\n");
        sleep (3);
        semaphoreSignal (S);
        printf("Signal done.\n");
    }
    return 0;
}
```

Si utilizza una variabile comune a tutti e due i processi: S ed anche il semaforo risulta globale perché la pipe deve essere condivisa. Se il pid==0 siamo nel figlio e si fa una wait, mentre il padre dorme 3 secondi e sveglierà il figlio.

## Semafori POSIX

L'implementazione è indipendente dal SO (POSIX) e l'header file è **semaphore.h** da inserire nei file .c

```
#include <semaphore.h>
```

Un semaforo è una variabile di tipo **sem\_t**

```
sem_t *sem1, *sem2, ...;
```

e tutte le funzioni sono denominate **sem\_\*** ed in caso di errore ritornano il valore -1.

**System call:**

```
sem_init  
sem_wait  
sem_try solo per completezza  
sem_post  
sem_getvalue solo per completezza  
sem_destroy
```

**sem\_init()**

```
int sem_init (sem_t *sem, int pshared, unsigned int value);
```

Inizializza il semaforo e riceve il **puntatore** al semaforo, **value** è il valore iniziale e il valore **pshared** dovrebbe essere:

- 0 se il semaforo è **locale** al processo corrente
- 1 se il semaforo può essere **condiviso** tra diversi processi

Linux non supporta i semafori condivisi, perciò:

- **Se lavoro con processi** -> Pipe
- **Se lavoro con thread** -> POSIX

**sem\_wait()**

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

Riceve lo stesso puntatore e se il semaforo è = 0 blocca il chiamante sino a quando può decrementare il valore del semaforo. Comportamento identico a quello già visto.

**sem\_try() – NON SERVE**

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

Equivale ad una wait **non bloccante**. Se il semaforo ha un valore maggiore di 0, lo decremente e ritorna 0. Se il semaforo è uguale a 0, ritorna -1 (invece di bloccare il chiamante come la wait). Serve se si vuole fare un controllo sul valore del semaforo.

**sem\_post()**

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

Classica operazione di signal. Incrementa il valore del semaforo.

### **sem\_getvalue() – NON SERVE**

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

E' un'evoluzione della try poiché ci permette di esaminare il valore di un semaforo. Il valore del semaforo viene assegnato a \*valP (i.e., valP è il puntatore all'intero che indica il valore del semaforo dopo la chiamata). Se ci sono processi in attesa, a \*valP si assegna 0 o un numero negativo il cui valore assoluto è uguale al numero di processi in attesa.

### **sem\_destroy()**

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

Distrugge un semaforo creato precedentemente e può ritornare -1 se si cerca di distruggere un semaforo utilizzato da un altro processo.

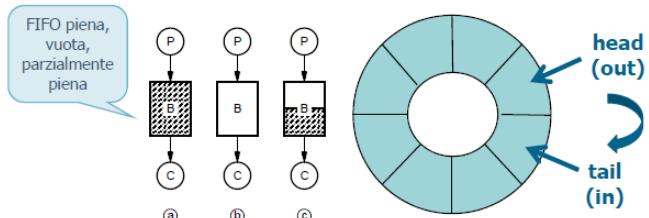
### **ESEMPIO**

```
...
#include "semaphore.h"
...
sem_t *sem;
...
sem = (sem_t *) malloc(sizeof(sem_t));
sem_init (sem, 0, 0);
...
... create sub processes or threads ...
...
sem_wait (sem);
... SC ...
sem_post (sem);
```

## 27. Problemi di sincronizzazione tipici

### Produttore – Consumatore ☺

Dal punto di vista teorico tale problema è già noto. Si traduce in informatica nel problema di una coda FIFO. I prodotti vengono inseriti da un lato ed estratti dall'altro. Utilizza un buffer circolare di dimensione SIZE per memorizzare gli elementi prodotti e da consumare.



La soluzione già proposta è quella dell'accesso sequenziale tramite funzioni di inserimento/estrazione in coda. Il problema che si può verificare è quello della variabile n che è comune ad entrambe le funzioni e dunque si può perdere qualche incremento/decremento.

```
#define SIZE ...
...
int queue[SIZE];
int tail, head;
...
void init () {
    tail = 0;
    head = 0;
    n = 0;
}
```

FIFO standard (non ADT)

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

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

Nell'accesso sequenziale **enqueue** e **dequeue** non sono mai contemporanee perciò non vi sono problemi.  
Nell'accesso concorrente/parallelo si possono avere due casi:

- **1 solo produttore e 1 solo consumatore:** le operazioni di enqueue e dequeue agiscono su estremità diverse della coda ma la variabile **n** è comunque condivisa
- **P produttori e C consumatori:** come il caso precedente con in più operazioni di accesso sullo stesso estremo della coda

### 1° soluzione (accesso concorrente)

Per un accesso parallelo/concorrente con 1 produttore e 1 consumatore è possibile rimuovere la variabile contatore **n** ed inserire al suo posto un semaforo "full" che conta il numero di elementi pieni ed un semaforo "empty" che conta il numero di elementi vuoti.

```
#define SIZE ...
...
int queue[SIZE];
int tail, head;
...
void init () {
    tail = 0;
    head = 0;
}
```

FIFO standard (non ADT)  
senza la variabile n

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

```
void dequeue (int *val) {
    *val=queue[head];
    head=(head+1)%SIZE;
    return;
}
```

1 Produttore  
1 Consumatore

Invece di n utilizza  
# Elementi pieni  
# Elementi vuoti

```
init (full, 0);
init (empty, SIZE);
```

```
Producer () {
    int val;
    while (TRUE) {
        produce (&val);
        wait (empty);
        enqueue (val);
        signal (full);
    }
}
```

```
Consumer () {
    int val;
    while (TRUE) {
        wait (full);
        dequeue (&val);
        signal (empty);
        consume (val);
    }
}
```

Si nota come nella nuova versione di codice la variabile contatore n è stata rimossa.

Inizialmente si suppone che la coda sia vuota perciò full viene inizializzato a 0 ed empty viene inizializzato alla dimensione massima SIZE.

Il produttore ottiene il valore del prodotto da inserire ed effettua una wait andando a decrementare il valore per poi successivamente andare ad accodare il valore al consumatore.

Se il produttore continua a girare da solo, senza che il consumatore intervenga, arriverà a un punto in cui empty sarà 0 e si bloccherà

all'istruzione wait(empty).

Dall'altro lato il consumatore effettuerà una wait su full e nel caso in cui ci sia qualcosa uscirà dall'attesa e preleverà il valore del prodotto dalla coda e successivamente effettuerà una signal per eventualmente svegliare il produttore.

## 2° soluzione (accesso concorrente) ☺

La soluzione 1 è simmetrica: il produttore produce posizioni piene ed il consumatore produce posizioni vuote. Tale soluzione può essere facilmente estesa al caso in cui coesistano più produttori e più consumatori.

- I produttori e consumatori operano su estremità opposte del buffer: possono farlo **contemporaneamente** purchè la coda non sia piena oppure vuota
- Due produttori oppure due consumatori devono invece agire in mutua esclusione

P Produttori  
C Consumatori

Occorre forzare ME tra P e tra C

```
init (full, 0);
init (empty, SIZE);
init (MEp, 1);
init (MEc, 1);
```

```
Producer () {
    int val;
    while (TRUE) {
        produce (&val);
        wait (empty);
        wait (MEp);
        enqueue (val);
        signal (MEp);
        signal (full);
    }
}
```

```
Consumer () {
    int val;
    while (TRUE) {
        wait (full);
        wait (MEc);
        dequeue (&val);
        signal (MEc);
        signal (empty);
        consume (val);
    }
}
```

Sono stati aggiunti due semafori: MEp (Mutua esclusione produttore), MEc (Mutua esclusione consumatore). Al codice precedente sono state aggiunte due wait e due signal. Un consumatore accede solo se la coda non è vuota e non c'è un altro consumatore. Il produttore accede solo se la coda non è piena e non ci sono altri produttori.

## Readers & Writers ☺

Problema classico in cui si cerca di condividere una base dati tra due insiemi di processi concorrenti:

- Una classe di processi detta **Reader** a cui è consentito accedere a un data-base in **concorrenza** (poiché devono solo leggere)
- Una classe di processi detta **Writer** a cui è consentito accedere al data-base in **mutua esclusione** sia con altri processi Writers sia con i processi Readers (poiché modificano la base dati)

Le due entità dunque **non** sono simmetriche.

Esistono almeno due tipi di problematiche:

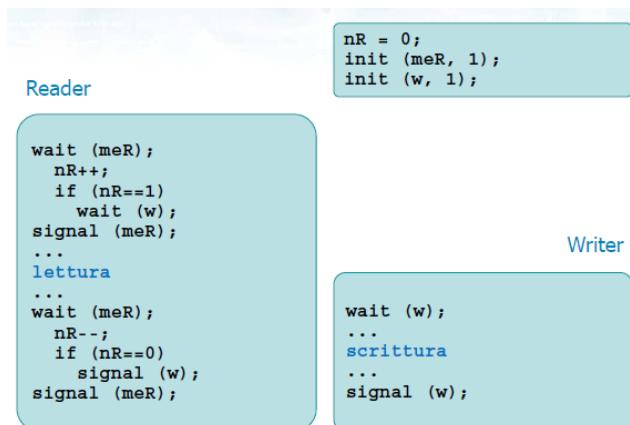
- Primo problema o problema con precedenza ai reader
- Secondo problema o problema con precedenza ai writer

### Precedenza ai reader ☺

Dare precedenza ai reader significa privilegiare l'accesso dei reader rispetto a quello dei writer ovvero: i reader non devono attendere a meno che un writer sia nella SC.

Protocollo di accesso

- I reader possono accedere in concorrenza al database
- Sino a quando arrivano reader i writer attendono
- Quando anche l'ultimo reader termina allora si può svegliare un writer (o un reader ... dipende dallo scheduler)



Si suppone di avere R reader e W writer (con  $R < W$ ). Si utilizzano nR reader attivi, meR mutua esclusione tra i reader e w la mutua esclusione tra i writer.

Si suppone che all'inizio vi siano solo writer: il primo passerà perché il semaforo è inizializzato a 1 mentre gli altri aspetteranno finché il primo writer non avrà terminato che sveglierà uno degli altri.

Se invece si suppone arrivino solo reader: il primo che arriva passa mentre tutti gli altri rimarranno in attesa e successivamente incrementa nR e se è il primo (in questo caso lo è perché  $nR=0$  e dunque  $nR++=1$ ) fa una wait(w) perché in questo modo va a **bloccare** i writer. Successivamente fa una signal(meR) per sbloccare un altro reader e permettergli di eseguire lo stesso pezzo di codice (cioè solo  $nR++$ ). Gli n reader si metteranno tutti nella sezione critica e lavoreranno in parallelo. Prima o poi uno uscirà e si ha un protocollo di uscita simmetrico a quello in ingresso: si decrementa nR e se  $nR==0$  si fa la signal del writer. Questo pezzo di codice viene eseguito da un reader



per volta poiché ancora una volta si fa wait(meR) e alla fine signal(meR). L'ultimo reader aprirà la porta ai writer (a causa della presenza dell'if nR==0).

Se il writer arriva prima del reader sarà il writer a bloccare il reader. Perciò il writer continuerà a scrivere, il primo reader sarà bloccato nella wait(w) dell'if mentre tutti gli altri reader saranno bloccati nella wait (meR). La signal del writer potrà svegliare o un reader oppure un eventuale writer.

Con l'aggiunta di un ulteriore semaforo è possibile rafforzare la precedenza dei reader. Poiché gli altri writer aspettano su meW e non w.

La soluzione utilizza

- Una variabile globale (nR) che conta il numero di reader nella SC
- Un semaforo di mutua esclusione per la manipolazione della variabile nR (meR)
- Un semaforo di mutua esclusione per più writer o per un reader e i writer (w)
- Un semaforo di mutua esclusione per writer (meW)

I writer sono soggetti a starvation, in quanto possono attendere per sempre ma sono possibili soluzioni più complesse senza starvation da parte dei W.

### Precedenza ai writer

Dare precedenza ai writer significa che un writer pronto deve attendere il meno possibile

Protocollo di accesso

- Ogni writer deve attendere che finiscano i reader
- Ogni writer ha priorità su tutti i reader

```
nR = nW = 0;
init (w, 1); init (r, 1);
init (meR, 1); init (meW, 1);
```

```
Reader
wait (r);
wait (meR);
nR++;
if (nR == 1)
    wait (w);
signal (meR);
signal (r);
...
lettura
...
wait (meR);
nR--;
if (nR == 0)
    signal (w);
signal (meR);
```

```
Writer
wait (meW);
nW++;
if (nW == 1)
    wait (r);
signal (meW);
wait (w);
...
scrittura
...
signal (w)
wait (meW);
nW--;
if (nW == 0)
    signal (r);
signal (meW);
```

Si hanno un contatore per i reader e uno per i writer e come in precedenza il semaforo meR serve per proteggere nR e simmetricamente meW protegge nW. In più i reader hanno il semaforo r e il writer il semaforo w.

Se un reader arriva prima fa la wait(r), passa, fa la wait(meR), passa, entra nell'if e fa wait(w) e blocca i writer nel punto precedente alla scrittura.

Viceversa se arriva prima un writer, quest'ultimo farà wait (meW) passerà, farà

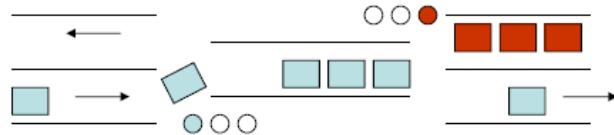
nW++, entrerà nell'if e farà una wait(r) e bloccherà i reader alla wait iniziale in prima riga.

Si nota che i writer sono bloccati dopo il prologo mentre i reader sono bloccati prima del prologo. Questo dimostra che i Writers hanno precedenza, poiché la signal libera i writer che si accumulano nella wait antecedente la scrittura, mentre il reader liberato dall'if finale del writer riparte dall'inizio del codice. Solo l'ultimo writer sveglia i readers.

I reader sono soggetti a starvation, in quanto possono attendere per sempre ma sono possibili soluzioni più complesse senza starvation.

## Tunnel a senso alternato 😊

In un tunnel a senso alternato si deve permettere a qualsiasi numero di auto (processi) di procedere nella stessa direzione. Se c'è traffico in una direzione bisogna bloccare il traffico nella direzione opposta.



Questo problema si può ricondurre a due insiemi di Readers, uno da sinistra a destra e l'altro da destra a sinistra.

```
n1 = n2 = 0;  
init (s1, 1); init (s2, 1);  
init (busy, 1);
```

```
left2right  
wait (s1);  
n1++;  
if (n1 == 1)  
    wait (busy);  
signal (s1);  
...  
Run (left to right)  
...  
wait (s1);  
n1--;  
if (n1 == 0)  
    signal (busy);  
signal (s1);
```

```
right2left  
wait (s2);  
n2++;  
if (n2 == 1)  
    wait (busy);  
signal (s2);  
...  
Run (left to right)  
...  
wait (s2);  
n2--;  
if (n2 == 0)  
    signal (busy);  
signal (s2);
```

Schema simile a quello dei reader precedente ma duplicato due volte.

Si ha un contatore n1 che conta i reader in un senso ed n2 che li conta nell'altro.

Il primo che passa in un senso blocca gli altri ed il primo che passa nell'altro senso blocca gli altri. Il primo che fa la wait(busy) blocca l'altro. Dopodiché tutti quelli che sono passati continuano a passare incrementando n1 o n2 (solo il primo fa wait busy). Il semaforo s1 serve per proteggere la variabile n1. L'ultimo che esce libera gli altri in attesa dall'altro

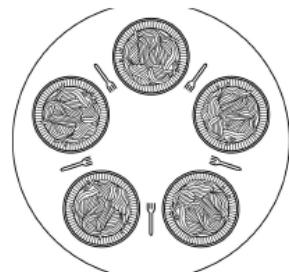
lato.

## I 5 filosofi 😊

Modello del caso in cui diverse risorse sono comuni a diversi processi concorrenti.

Un tavolo è imbandito con 5 piatti di riso e 5 bastoncini (cinesi) ciascuno tra due piatti. Intorno al tavolo siedono 5 filosofi e questi pensano oppure mangiano.

Per mangiare, ogni filosofo ha bisogno di due bastoncini e possono essere ottenuti uno alla volta.



## Soluzioni filosofiche

- Insegnare ai filosofi a mangiare con 1 solo bastoncino
- Fornire più di 5 bastoncini
- Permettere solo al più a 4 filosofi di sedersi al tavolo
- Forzare asimmetria: i filosofi di posizione pari prendono la forchetta sinistra per prima, i filosofi di posizione dispari prendono la forchetta destra per prima

## Modello 1 – ERRATO

Utilizzare un unico semaforo binario (mutex) per proteggere l'unica risorsa "cibo"

- Annulla la concorrenza
- Un solo filosofo mangia (potrebbero farlo in due)
- Funziona per il deadlock ma un filosofo blocca l'azione di mangiare agli altri

```
init (mutex, 1);
```

```
while (true) {  
    Pensa ();  
    wait (mutex);  
    Mangia ();  
    signal (mutex);  
}
```

## Modello 2 – ERRATO

Avere un semaforo per bastoncino può causare deadlock

```
init (chopstick[0], 1);  
...  
init (chopstick[4], 1);  
  
while (true) {  
    Pensa ();  
    wait (chopstick[i]);  
    wait (chopstick[(i+1)mod5]);  
    Mangia ();  
    signal (chopstick[i]);  
    signal (chopstick[(i+1)mod5]);  
}
```

i ∈ [0, 4]

Penso, occupo bastoncino sinistra, occupo bastoncino di destra però può causare deadlock perché le operazioni non sono atomiche ( $i+1 \bmod 5$ ). Quindi prima di occupare quello di destra tutti occupano quello di sinistra e allora tutti sono in deadlock.

## Soluzione ☺

Si utilizza l'idea generale di usare takeForks come protocollo di ingresso e putForks come protocollo di uscita. Ogni filosofo può essere nello stato: THINKING, HUNGRY, EATING. C'è quindi un ulteriore stato di prenotazione.

```
int state[N];  
init (mutex, 1);  
init (sem[0], 0); ...; init (sem[4], 0);  
  
takeForks (int i) {  
    wait (mutex);  
    state[i] = HUNGRY;  
    test (i);  
    signal (mutex);  
    wait (sem[i]);  
}  
  
putForks (int i) {  
    wait (mutex);  
    state[i] = THINKING;  
    test (LEFT);  
    test (RIGHT);  
    signal (mutex);  
}  
  
test (int i) {  
    if (state[i]==HUNGRY && state[LEFT]!=EATING &&  
        state[RIGHT]!=EATING) {  
        state[i] = EATING;  
        signal (sem[i]);  
    }  
}
```

```
while (TRUE) {  
    think ();  
    takeForks (i);  
    eat ();  
    putForks (i);  
}
```

Effettuare takeForks (supponendo 1 solo filosofo che vuole mangiare): fa la wait mutex (inizializzato a 1) e può farlo, si imposta su HUNGRY e lancia la funzione test passando il proprio indice. La funzione test controlla se si è affamati e verifica lo stato del filosofo a sinistra e destra assicurandosi che non stiano mangiando. Se così è ci si imposta sullo stato EATING e si fa la signal sul proprio semaforo (ogni filosofo ne ha uno impostato a 0). Uscendo dalla funzione test si fa la signal(mutex) e successivamente si fa una wait(sem[i]) e ci si sveglia grazie alla signal della funzione test. Questo perché se a sinistra o a destra stanno mangiando la signal non si fa e allora il filosofo si ferma sulla wait(sem[i]).

Supponendo ora che il filosofo stia mangiando, se ne arriva un altro dal lato opposto farà nuovamente la wait(mutex) e tutto il resto senza problemi.

Appena ne arriva un terzo si fermerà sul semaforo `wait(sem[i])` poiché la condizione di test non sarà verificata.

Quando si termina di mangiare si fa una `wait(mutex)` e si imposta il proprio stato su `THINKING` e si fa un test a sinistra e a destra cercando di svegliare i filosofi (passa gli indici `LEFT` e `RIGHT` e non `i`) che sono in attesa sulla `wait`. Questo lo fa per evitare starvation. Uno che si ferma può sbloccarne due.

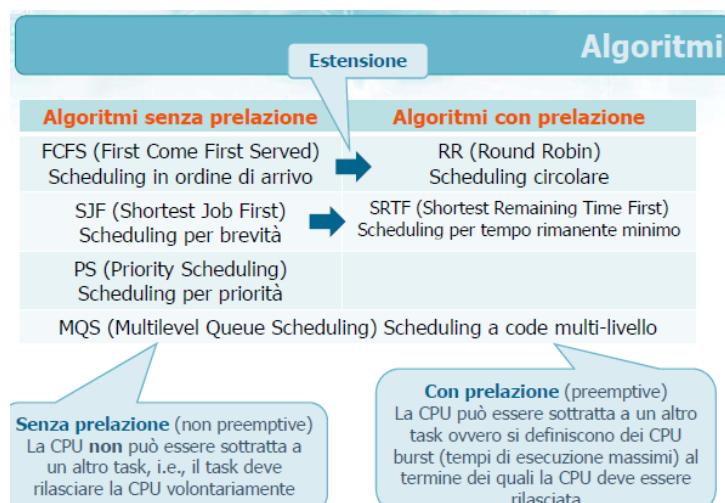
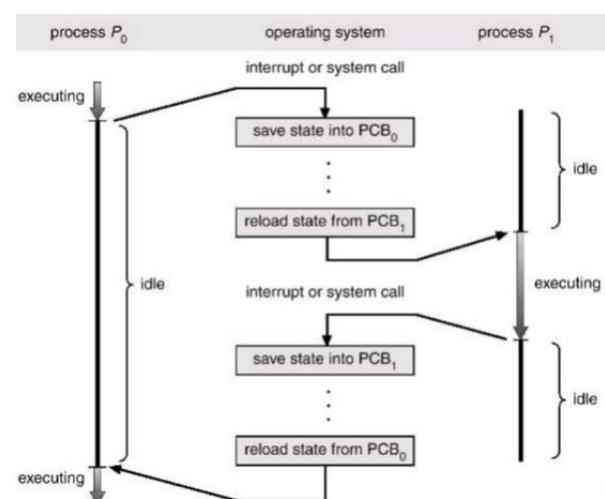
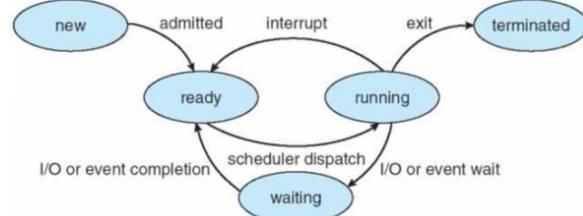
## 27. Lo scheduling ☺

Uno degli obiettivi della multiprogrammazione è quello di massimizzare l'utilizzo delle risorse e in particolare della CPU. Per raggiungere tale obiettivo ogni CPU viene assegnata a più task (i.e., processo o thread). Lo scheduler deve decidere quale **algoritmo** utilizzare per assegnare la CPU a un task. Le prestazioni dello scheduler sono valutate tramite **funzioni di costo**. Applicazioni diverse richiedono algoritmi e funzioni di costo diverse.

### Algoritmi di scheduling

Il procedimento generale per lo scheduling è il seguente:

- La CPU viene assegnata a un task
- Qualora il processo entri in uno stato di attesa, termini, venga ricevuto un interrupt, etc., è necessario effettuare un context switching
- Per ogni context switching: il task in running viene spostato nella coda di ready; un task nella coda di ready viene spostato allo stato di running



Gli algoritmi sono molteplici e possono essere divisi in due categorie: con o senza prelazione (cioè se può o essere interrotta la CPU).

Nella seconda categoria esiste un **burst**, cioè un limite massimo per l'esecuzione superato il quale la CPU viene affidata ad altre operazioni.

## Funzioni di costo

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

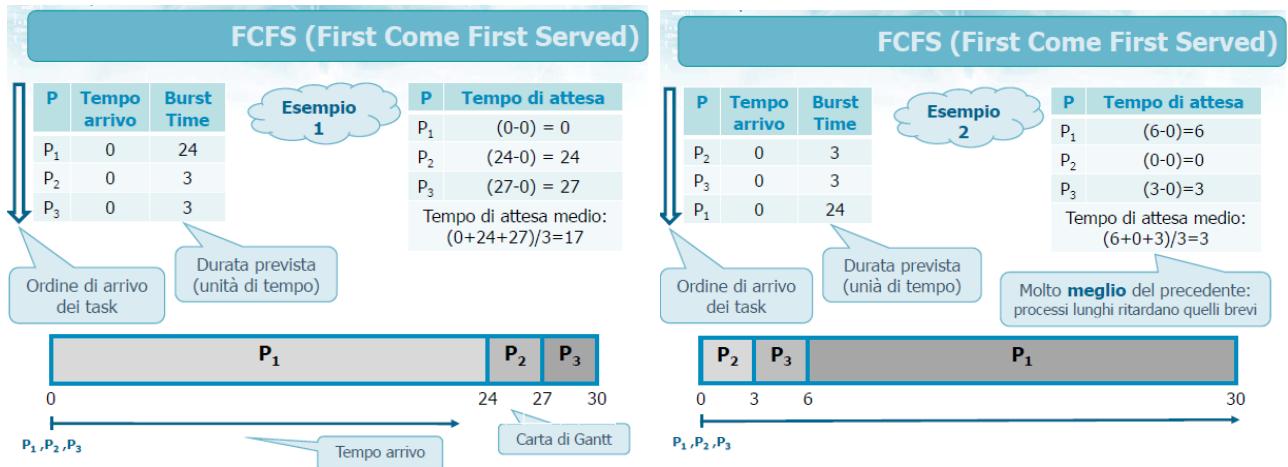
Ogni algoritmo verrà valutato tramite apposite funzioni di costo. Verrà utilizzata la funzione di costo relativa al **tempo di attesa**. Si sommeranno i tempi trascorsi in coda e si farà la media.

## FCFS (First Come First Served) ☺

Con questo algoritmo la CPU è assegnata ai task seguendo l'ordine con cui la **richiedono**. I task vengono gestiti attraverso una coda **FIFO**:

- Un task in arrivo viene inserito in coda
- Un task da servire viene estratto dalla testa

Lo scheduling può essere illustrato mediante un **diagramma (o carta) di Gantt** (1917), cioè tramite un diagramma a barre che illustra la pianificazione (tempi di inizio e fine) delle attività ricordando che nessun task viene interrotto, ovvero la CPU può solo essere rilasciata volontariamente.



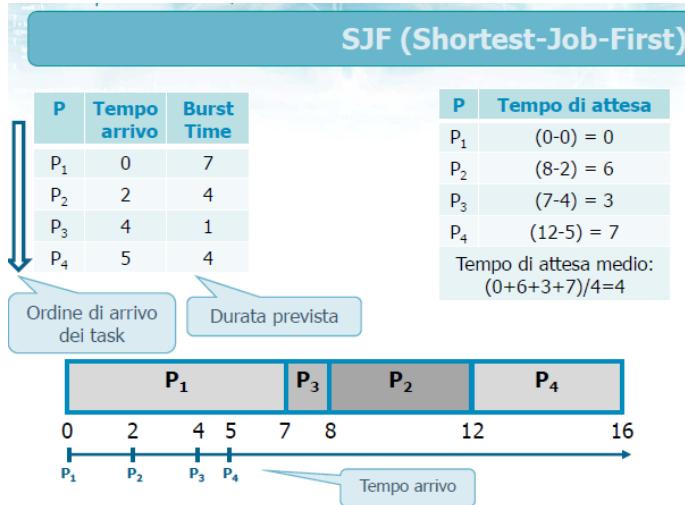
La configurazione vede i 3 processi arrivare nello stesso istante (a sinistra) e ognuno richiede tempi diversi. Quando il tempo di arrivo è lo stesso si suppone di prelevarli dall'alto al basso. A destra si calcola il tempo di attesa come **tempo in cui si viene serviti – tempo di arrivo**.

**Pregi:** Facile da comprendere e da implementare

**Difetti:** tempi di attesa relativamente lunghi e variabili (non ottimi). Inadatto per sistemi real-time (no prelazione). Si ha inoltre l'**effetto convoglio**: task brevi in coda a task lunghi attendono molto tempo inutilmente.

## SJF (Shortest Job First) ☺

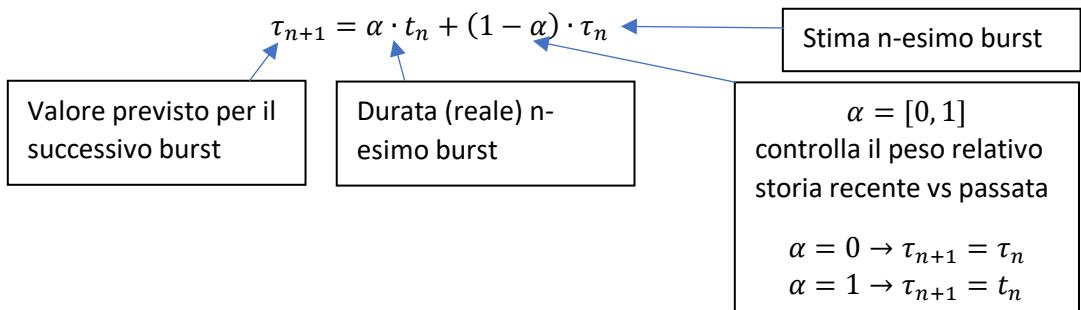
A ogni task viene associata la durata della sua prossima richiesta (next CPU burst): i task vengono schedulati in ordine di durata della loro prossima richiesta. Lo scheduling avviene dunque per brevità ed in caso di ex-aequo si applica lo scheduling FCFS.



Si nota come P<sub>3</sub> non può essere servito subito perché **ARRIVA DOPO**. Attenzione durante esame. Scrivere il tempo di arrivo dei processi.

**Pregi:** si può dimostrare che SJF è un algoritmo ottimo utilizzando il tempo di attesa come criterio. Spostando i processi brevi prima di quelli lunghi il tempo di attesa dei primi diminuisce più di quanto aumenta il tempo di attesa dei secondi.

**Difetti:** è possibile l'attesa indefinita, ovvero la **starvation**. Si ha inoltre difficoltà di applicazione derivata dall'impossibilità di conoscere a priori il comportamento futuro. Il tempo del burst successivo è ignoto ma è possibile effettuare delle stime utilizzando diversi criteri, tra i quali la **media esponenziale**.



Procedendo per sostituzione (invece di  $\tau_n$  inserisco di nuovo la formula)

$$\tau_{n+1} = \alpha \cdot t_n + (1 - \alpha) \cdot \alpha \cdot t_{n-1} + \cdots + (1 - \alpha)^j \cdot \alpha \cdot t_{n-j} + \cdots + (1 - \alpha)^{n+1} \cdot \tau_0$$

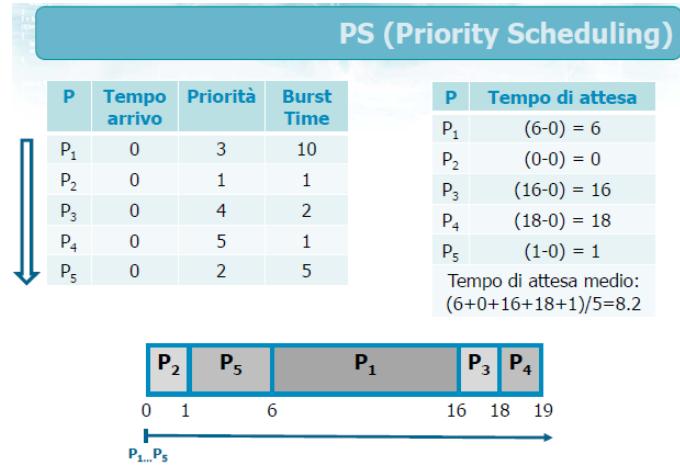
Dato che sia  $\alpha$  che  $1 - \alpha$  sono minori di 1, ogni termine successivo ha un peso minore.

## PS (Priority Scheduling) ☺

A ogni processo viene associata la sua priorità: la priorità è generalmente rappresentata mediante valori interi. A priorità maggiore corrisponde intero minore e le priorità possono essere determinate in base a criteri:

- Interni, memoria utilizzata, numero file utilizzati, etc.
- Esterini, proprietario del task, etc.

La CPU viene allocata al processo con la priorità maggiore



In sostanza l'algoritmo PS è uguale a quello SJF con la durata del CPU burst sostituita dalla priorità.

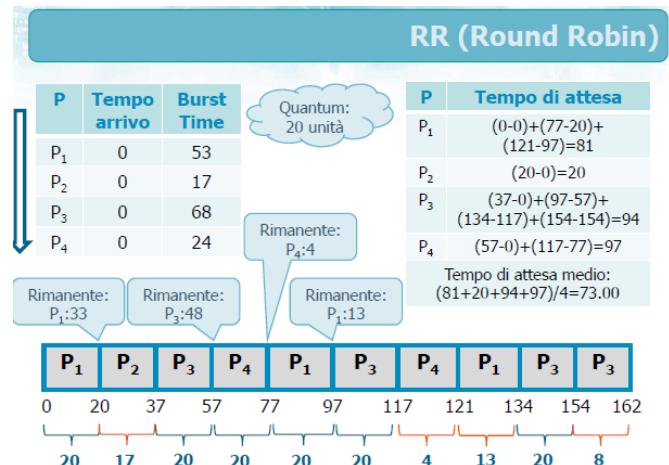
**Difetti:** è possibile avere attesa indefinita, ovvero la starvation. Nei sistemi molto carichi, i task con bassa priorità possono attendere per sempre. Una possibile soluzione alla starvation è l'invecchiamento (aging) dei task, cioè si fa in modo che la priorità venga incrementata gradualmente con il passare del tempo.

## RR (Round Robin) ☺

Il Round Robin viene anche detto scheduling circolare ed è la versione di FCFS che permette la **prelazione**. L'utilizzo della CPU viene suddiviso in "time quantum" (porzioni temporali). Ogni task riceve la CPU per un **tempo massimo** pari al quantum e poi viene inserito nuovamente nella ready queue che viene gestita con modalità FIFO. Eventuali nuovi processi sono aggiunti alla coda. Questo algoritmo è progettato appositamente per sistemi time sharing e real time

Difetti: il tempo di attesa medio è relativamente lungo. Si ha notevole dipendenza delle prestazioni dalla durata del quanto di tempo.

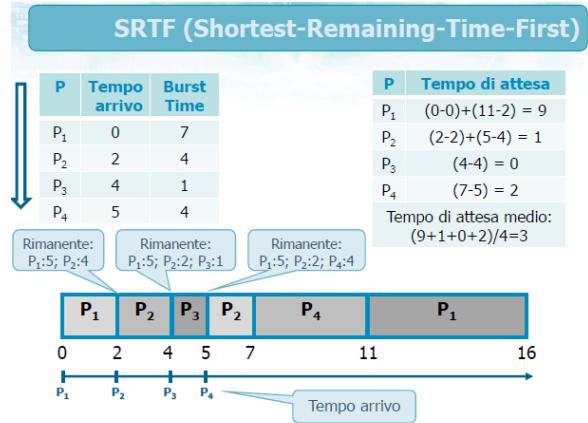
- Quantum lungo: RR degenera in FCFS
- Quantum corto: vengono effettuati troppi context switching e i tempi di commutazione/gestione risultano molto elevati



## SRTF (Shortest-Remaining-Time-First) ☺

Versione di SJF che permette la prelazione. Si procede a uno scheduling SJF ma se viene sottomesso un processo con burst più breve di quello in esecuzione la CPU viene prelazionata.

Le caratteristiche sono simili allo scheduling SJF.

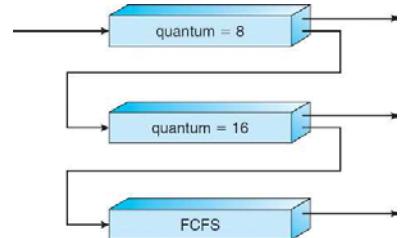


## MQS (Multilevel Queue Scheduling) ☺

Questo algoritmo è applicato a situazioni in cui i task possono essere classificati in gruppi diversi: foreground, background, di sistema, etc.

La ready queue viene suddivisa in code diverse: ogni coda può essere gestita con il proprio algoritmo di scheduling.

L'algoritmo può essere modificato per permettere il trasferimento dei task tra le varie code tramite MQS con retroazione.



## Considerazioni aggiuntive

Lo scheduling può essere effettuato a livello di processi oppure di thread: se il SO prevede thread normalmente lo scheduling viene effettuato a livello di thread, trascurando i processi.

### Scheduling dei thread

Il SO gestisce i T a livello kernel e ignora i T a livello utente (gestiti da una libreria) perciò lo scheduling può essere effettuato solo per i T a livello kernel (se esistono).

### Scheduling per sistemi multiprocessori

Tutte le esemplificazioni precedenti sono state fatte supponendo l'esistenza di una sola CPU. Nel caso siano disponibili più unità elaborate il carico può essere distribuito: il **bilanciamento del carico** è automatico per SO con code di attesa comuni a tutti i processori.

Esistono diversi schemi:

- Multi-elaborazione **asimmetrica**: un processore master distribuisce il carico tra i processori server
- Multi-elaborazione **simmetrica**: ciascun processore provvede al proprio scheduling

### Scheduling per sistemi real-time

Tentano di rispondere in tempo reale ai verificarsi di eventi. Gli eventi guidano il comportamento dello scheduling e si definisce **latenza** il tempo che intercorre tra il verificarsi dell'evento e la sua gestione. Esistono due tipi di sistemi real-time:

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

## ESERCIZIO ESAME



Si consideri il seguente insieme di processi

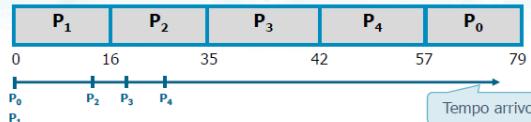
P	Tempo arrivo	Burst Time	Priorità
P <sub>0</sub>	0	22	5
P <sub>1</sub>	0	16	2
P <sub>2</sub>	15	19	4
P <sub>3</sub>	17	7	1
P <sub>4</sub>	25	15	1

Ordine di arrivo dei task

Priorità massima = valore inferiore  
Quantum temporale = 10

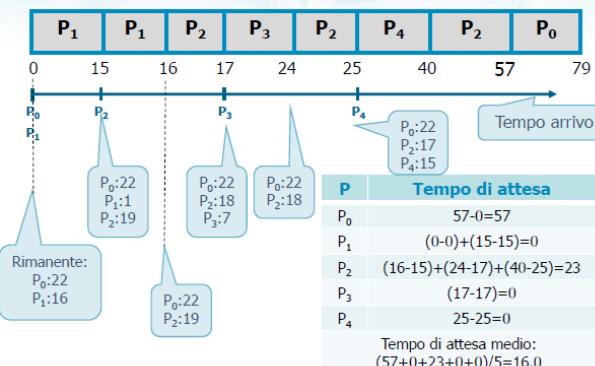
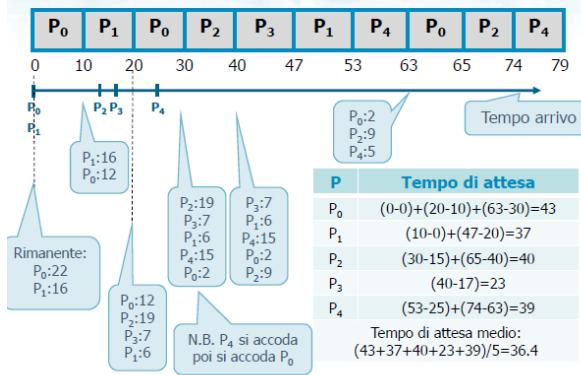
Rappresentare il diagramma di Gantt per gli algoritmi PS (Priority Scheduling), RR (Round Robin) e SRTF (Shortest Remaining Time First)

Calcolare il tempo di attesa medio



P	Tempo di attesa
P <sub>0</sub>	57-0=57
P <sub>1</sub>	0-0=0
P <sub>2</sub>	16-15=1
P <sub>3</sub>	35-17=18
P <sub>4</sub>	42-25=17

Tempo di attesa medio:  
(57+0+1+18+17)/5=18.6



## 28. Definizione del problema e modellizzazione

### Stallo (deadlock) ☺

Condizione di **stallo (deadlock)**: un P/T richiede una risorsa non disponibile, entra in uno stato di attesa e l'attesa non termina più.

Lo stallo consiste quindi in un insieme di P/T attendono tutti il verificarsi di un evento che può essere causato solo da un altro processo dello stesso insieme.

Deadlock **implica** starvation **non** il contrario: la starvation di un P/T implica che tale P/T attende indefinitamente ma gli altri P/T possono procedere in maniera usuale e non essere in deadlock. Tutti i P/T in deadlock sono in starvation.

Condizioni per il verificarsi di un deadlock	
Condizioni	Descrizione
Mutua esclusione (mutual exclusion)	Deve esserci almeno una risorsa <b>non condivisibile</b> . La prima richiesta è soddisfatta, le successive no.
Possesso e attesa (hold and wait)	Un processo <b>mantiene</b> almeno una risorsa e <b>attende</b> per aquisire almeno un'altra risorsa.
Impossibilità di prelazione (no preemption)	<b>Non</b> esiste il diritto di <b>prelazione</b> per almeno una risorsa. Almeno una risorsa non può essere sottratta ma solo rilasciata da chi la utilizza.
Attesa circolare (circular wait)	Esiste un insieme $\{P_1, \dots, P_n\}$ di P tale che $P_1$ <b>attende</b> una risorsa tenuta da $P_2$ , $P_2$ <b>attende</b> una risorsa tenuta da $P_3$ , ..., $P_n$ <b>attende</b> una risorsa tenuta da $P_1$ .

Devono verificarsi tutte contemporaneamente per avere un deadlock

Condizioni necessarie ma non sufficienti  
Sono distinte ma non indipendenti (e.g., 4→2)

### Modellizzazione del deadlock ☺

Molti sistemi operativi utilizzano la tecnica **dello “struzzo”** cioè si ignora il problema supponendo la probabilità di un deadlock nel sistema sia bassissima. Questo metodo è tanto meno appropriato quanto più aumentano la concorrenza e la complessità dei sistemi.

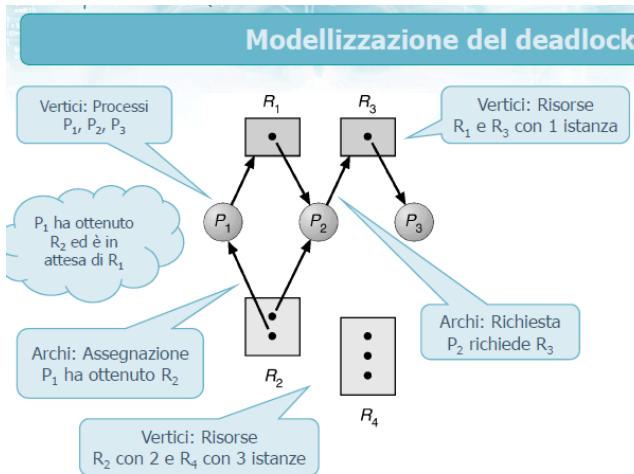
Normalmente una situazione di deadlock può essere modellizzata mediante un **grafo di allocazione o assegnazione** delle risorse:  $G = (V, E)$

L'insieme di vertici V è costituito da:

- $P = \{P_1, P_2, \dots, P_n\}$   
Insieme dei processi del sistema (n in tutto) in cui i processi sono indistinguibili e in numero indefinito ed ogni processo utilizza una risorsa facendone accesso mediante protocollo standard costituito da: *Richiesta, Utilizzo, Rilascio*.
- $R = \{R_1, R_2, \dots, R_m\}$   
Insieme delle risorse del Sistema (m in tutto). Le risorse sono suddivise in classi (tipi) ed ogni risorsa di tipo  $R_i$  ha  $W_i$  istanze (ci possono essere risorse con istanze unitaria, ad esempio un'unica stampante). Tutte le istanze di una classe sono **identiche**: una qualsiasi istanza soddisfa una richiesta per quel tipo di risorsa (se così non fosse occorrebbe riformulare la suddivisione in classi)

L'insieme degli archi E è suddiviso in

- Archi di richiesta:  $P_i \rightarrow R_j$ , i.e., da processo a risorsa
- Archi di assegnazione:  $R_j \rightarrow P_i$ , i.e., da risorsa a processo

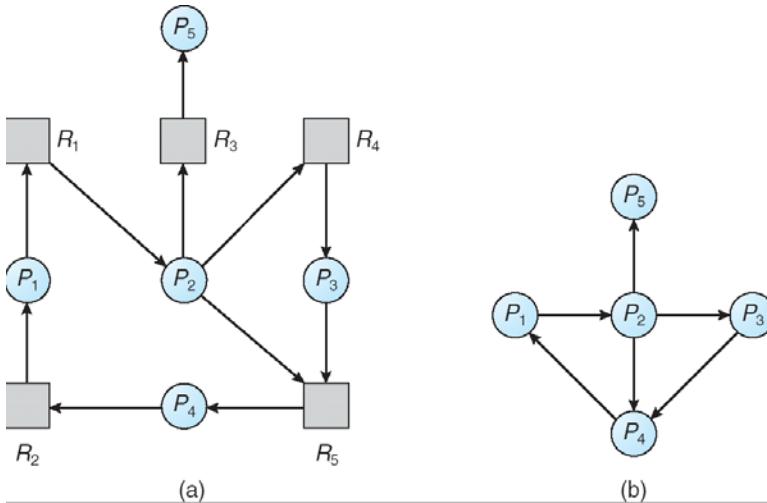


I cerchi sono i processi: ve ne sono 3. Vi sono inoltre 4 risorse (i quadrati). I puntini all'interno delle risorse servono graficamente per rappresentare le **istanze**. Qualsiasi istanza di  $R_2$  può soddisfare la richiesta (idem  $R_4$ ).

Nessuno ha fatto richiesta a  $R_4$ . La prima istanza di  $R_2$  è stata assegnata a  $P_1$  e la seconda a  $P_2$ .

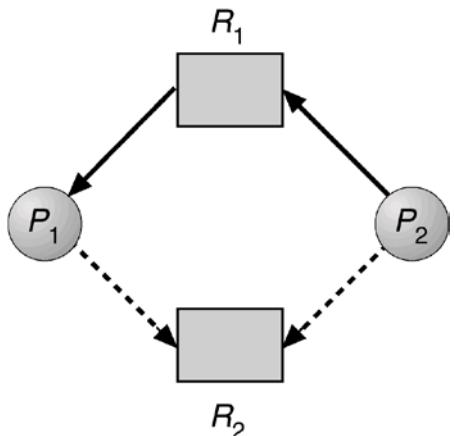
$P_2$  attende la risorsa di  $R_3$  e  $P_1$  attende la risorsa di  $R_1$ .

Dal grafo di allocazione delle risorse è possibile ottenere il grafo **di attesa** in cui le risorse spariscono e si rappresenta l'attesa di un processo nei confronti dell'altro.



Ad esempio, si cancellano tutte le risorse e si utilizza la proprietà transitiva.  $P_2$  richiede  $R_3$  che è stato assegnato a  $P_5$ , dunque nel grafo di attesa l'arco andrà da  $P_2$  a  $P_5$ .

In alcuni casi è utile estendere il grafo di allocazione in un **grafo di rivendicazione**: si aggiungono al grafo di allocazione delle risorse degli archi di reclamo, di rivendicazione o di intenzione di richiesta (claim edge). L'arco  $P_i \rightarrow R_j$  indica che il processo  $P_j$  richiederà (in futuro) la risorsa  $R_j$  ed è rappresentato mediante linea tratteggiata.



## Tecniche di gestione a posteriori

Si permette al sistema di entrare in uno stato di deadlock per poi intervenire.

L'algoritmo richiede quindi due passi:

- **Rilevazione** (della condizione di stallo) dove il sistema esegue un algoritmo di rilevazione dello stallo all'interno del sistema (detection)
- **Ripristino** (del sistema): se esiste uno stallo si applica una strategia di ripristino (recovery)

### Rilevazione

Dato un grafo di assegnazione è possibile verificare la presenza di uno **stallo** verificando la presenza di **cicli**.

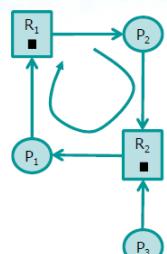
**Se il grafo non contiene cicli allora non c'è deadlock.** Se il grafo contiene uno o più cicli allora:

- Vi è sicuramente deadlock se esiste solo un'istanza per ciascun tipo di risorsa
- C'è la **possibilità** di deadlock se esiste più di un'istanza per tipo di risorsa

La presenza di cicli è condizione necessaria ma non sufficiente nel caso di istanze multiple (algoritmo del Banchiere).

#### Esempio

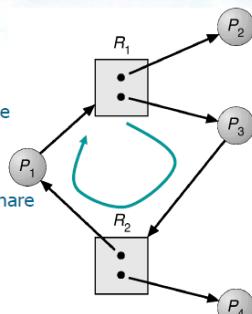
- ❖ Processi
  - >  $P_1, P_2, P_3$
- ❖ Risorse
  - >  $R_1$  e  $R_2$  con una istanza
- ❖ Esistenza di un ciclo
- ❖ Condizione di stallo
  - >  $P_1$  attende  $P_2$
  - >  $P_2$  attende  $P_1$



Si nota un ciclo che denota il fatto che si ha un deadlock.  $P_1$  attende  $P_2$  e  $P_2$  attende  $P_1$ .

#### Esempio

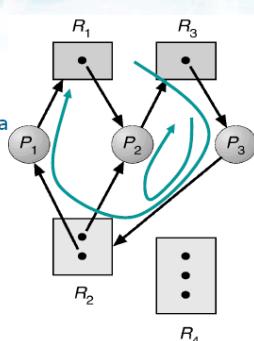
- ❖ Processi
  - >  $P_1, P_2, P_3, P_4$
- ❖ Risorse
  - >  $R_1$  e  $R_2$  con due istanze
- ❖ Esistenza di un ciclo
- ❖ Non esiste stallo
  - >  $P_2$  e  $P_4$  possono terminare
  - >  $P_1$  può acquisire  $R_1$  e terminare
  - >  $P_3$  può acquisire  $R_2$  e terminare



In questo caso vi sono 4 processi e 2 risorse che non hanno istanze unitarie. Per riassumere:  $R_1$  è stata affidata  $P_2$  e  $P_3$ , ma  $P_3$  attende  $R_2$  perché quest'ultima è stata assegnata a  $P_1$  e  $P_4$ .  $P_1$  però attende  $R_1$ . In questo caso non esiste stallo perché  $P_2$  e  $P_4$  possono terminare e se ad esempio  $P_2$  termina,  $P_1$  può sfruttare l'istanza liberata da  $P_2$  e la stessa cosa con  $P_4$ .

#### Esempio

- ❖ Processi
  - >  $P_1, P_2, P_3$
- ❖ Risorse
  - >  $R_1$  e  $R_3$  con una istanza
  - >  $R_2$  con due istanze
  - >  $R_4$  con tre istanze
- ❖ Esistenza di due cicli
- ❖ Condizione di stallo
  - >  $P_1$  attende  $R_1$
  - >  $P_2$  attende  $R_3$
  - >  $P_3$  attende  $R_2$



In questo caso vi sono due cicli e vi è una condizione di stallo, poiché vi sono istanze unitarie.  $P_1$  attende  $R_1$ ,  $P_2$  attende  $R_3$  e  $P_3$  attende  $R_2$  perciò questi si attendono a vicenda.

## Complessità

Ogni fase di rilevazione ha un costo poiché è necessario determinare i cicli nel grafo. La presenza di cicli può essere verificata mediante **visita in profondità**. Un grafo è aciclico se una visita in profondità **non** incontra archi etichettati “back” (archi verso vertici grigi). Se si raggiunge un vertice grigio, ovvero si attraversa un arco backward, si ha un ciclo.

Il costo temporale di tale operazione è pari a

- $O(|V|+|E|)$  per rappresentazioni con lista di adiacenza
- $O(|V|^2)$  per rappresentazioni con matrice di adiacenza

Quando spesso si effettuano le rilevazioni?

- Ogni volta che un processo fa una richiesta che non viene soddisfatta immediatamente
- A intervalli di tempo fissi, e.g., ogni 30 minuti
- A intervalli di tempo variabili, e.g., quando l'utilizzo della CPU scende sotto una certa soglia

## Ripristino

Una volta che un ciclo è stato rilevato, bisogna ripristinarlo. Per ripristinare un corretto funzionamento sono possibili diverse strategie

- Terminare tutti i processi in stallo
- Terminare un processo alla volta tra quelli in stallo selezionando un processo, terminandolo, ricontrollando la condizione di stallo ed eventualmente iterando
- Prelazionare le risorse a un processo alla volta tra quelli in stallo: si seleziona un processo, si prelazionano le sue risorse, se ne fa il roll-back, si ricontrolla la condizione di stallo, eventualmente iterando

Ripristino	
Strategia	Descrizione
Terminare tutti i processi in stallo	<ul style="list-style-type: none"><li>• Complessità: semplice causare inconsistenze sulle basi dati</li><li>• Costo: molto più alto di quanto potrebbe essere strettamente necessario</li></ul>
Terminare un processo alla volta tra quelli in stallo	<ul style="list-style-type: none"><li>• Complessità: alta in quanto occorre selezionare l'ordine delle vittime con criteri oggettivi (priorità, tempo di esecuzione effettuato e da effettuare, numero risorse possedute, etc.)</li><li>• Costo: elevato, i.e., dopo ogni terminazione occorre ricontrolare la condizione di stallo</li></ul>
Prelazionare le risorse a un processo alla volta	<ul style="list-style-type: none"><li>• Complessità: occorre effettuare il rollback, i.e., fare ritornare il processo vittima a uno stato sicuro</li><li>• Costo: la selezione di una vittima deve minimizzare il costo della prelazione</li></ul>

## Conclusioni

Rilevazione e ripristino sono operazioni **complesse** logicamente ed **onerose** temporalmente. In ogni caso se un processo richiede molte risorse è possibile causare starvation. Lo stesso processo viene ripetutamente scelto come vittima e incorre in rollback ripetuti. È quindi opportuno inserire il numero di terminazioni tra i parametri di scelta della vittima, facendone aumentare la priorità o simili.

## 29. Tecniche di prevenzione ☺

Le tecniche di prevenzione cercano di **controllare** le modalità di richiesta delle risorse per **prevenire** il verificarsi di **almeno una** delle condizioni necessarie. In altre parole cercano di prevenire il verificarsi di almeno una delle condizioni seguenti:

- Mutua esclusione (mutual exclusion)
- Possesso e attesa (hold and wait)
- Impossibilità di prelazione (no preemption)
- Attesa circolare (circular wait)

Mutua esclusione	
Uno stallo si verifica a causa della "mutua esclusione" quando un processo rimane indefinitivamente in attesa su una risorsa non condivisibile	
Quindi uno stallo potrebbe essere evitato se 1. Non esistessero risorse non condivisibili 2. Non si potesse rimanere in attesa di una risorsa non condivisibile	
Strategia 1	Proibire risorse non condivisibili Tale strategia è considerata generalmente molto restrittiva
Strategia 2	Proibire l'attesa su risorse non condivisibili Tale strategia è considerata complessa da realizzare

Possesso e attesa	
Un stallo si verifica a causa di un "possesso e attesa" quando un P possiede una o più risorse e ne chiede di ulteriori	
Quindi una condizione di possesso e attesa potrebbe essere evitata imponendo che un P chieda una risorsa solo quando non ne possiede altre	
Request All First (RAF)	I P devono acquisire tutte le risorse necessarie prima di iniziare l'attività di elaborazione <ul style="list-style-type: none"><li>• Scarso utilizzo delle risorse</li><li>• Risorse eventualmente assegnate molto prima di essere utilizzate</li></ul>
Release Before Request (RBR)	Ai P è consentito richiedere risorse solamente se non ne hanno già acquisite in precedenza Prima di ogni nuova richiesta ogni P deve rilasciare le risorse già possedute <ul style="list-style-type: none"><li>• Possibilità di starvation</li><li>• P che richiedono molte risorse molto utilizzate possono dover "ricominciare" molto spesso</li></ul>

## Impossibilità di prelazione

Un stallo si verifica a causa dell' "impossibilità di prelazione" quando una risorsa non può essere sottratta a un P

In generale è complesso sottrarre risorse a altri processi in esecuzione  
Però si potrebbe ottenere un effetto simile

Permettere la prelazione delle risorse possedute dal processo stesso	Se un processo, che mantiene alcune risorse, ne chiede un'altra che non può essere allocata immediatamente, è costretto a rilasciare tutte le risorse mantenute (preemption) Le risorse liberate sono aggiunte alla lista delle risorse che il processo attende di acquisire Il processo sarà svegliato solo quando potrà riacquisire tutte le sue vecchie risorse e quelle nuove che richiede
--	--

## Impossibilità di prelazione

Un stallo si verifica a causa dell' "impossibilità di prelazione" quando una risorsa non può essere sottratta a un P

Permettere la prelazione di risorse possedute da un altro processo purchè esso sia in fase di attesa	Se la risorsa richiesta non è disponibile si verifica quale processo la possiede Se il processo che la possiede è a sua volta in attesa si sottrae a tale processo la risorsa richiesta assegnandola al processo che ne ha fatto richiesta In caso contrario il processo viene messo in attesa e in futuro potranno essere prelazionate le sue risorse
--	--

Entrambe le strategie

- Sono applicabili a risorse su cui è semplice salvare e ripristinare lo stato (registri CPU, memoria centrale, etc.)
- Non sono applicabili quando lo stato della risorsa non è ricostruibile (file, stampanti, etc.)

## Attesa circolare

Un stallo si verifica a causa di un' "attesa circolare" quando un insieme di P è tale per cui ogni P dell'insieme attende una risorsa posseduta da un altro P dell'insieme

Per evitare tale condizione si potrebbe imporre un ordinamento totale tra tutte le classi di risorse

### Hierarchical Resource Usage (HRU)

- Impone una relazione di ordinamento totale tra i vari tipi di risorse, associando a ciascuno di essi un numero intero. Esempio: HD = 1, DVD = 5, stampanti = 12
  - Forza ogni processo a richiedere le risorse con un ordine crescente di enumerazione
- In generale la verifica HRU sia applicato può essere effettuata dal
- Programmatore
  - Sistema operativo. Il tool "witness", disponibile in UNIX versione FreeBSD, controlla l'ordine dei lock effettuati dai processi

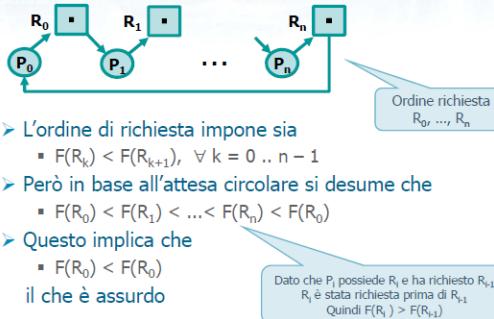
## Attesa circolare

- ❖ Sia  $F$  la funzione che impone un ordine univoco tra tutte le classi di risorse  $R_i$  del sistema
- ❖ Un processo  $P$ 
  - Abbia precedentemente richiesto una istanza della risorsa  $R_{old}$  e faccia richiesta di una istanza di  $R_{new}$
  - Se  $F(R_{new}) > F(R_{old})$ 
    - La risorsa viene concessa
  - Se  $F(R_{new}) \leq F(R_{old})$ 
    - Il processo deve rilasciare tutte le risorse di tipo  $R_i$  per cui  $F(R_{new}) \leq F(R_i)$  prima di ottenere  $R_{new}$

## Attesa circolare

- ❖ È possibile dimostrare che tale condizione è sufficiente per evitare l'attesa circolare
- Ovvero, se le risorse si richiedono in un certo ordine è vero che non è possibile avere attesa circolare?
- Procediamo per assurdo, supponendo ci sia attesa circolare, ovvero supponiamo che esista un insieme di processi che
  - Sono stati richiesti con ordine specificato, e.g., in ordine numerico crescente
  - Risultino in attesa circolare

## Attesa circolare



## 30. Tecniche per evitare uno stallo

### Evitare il deadlock ☺

Le tecniche per **evitare** condizioni di stallo forzano i  $P$  a fornire (a priori) informazioni aggiuntive sulle richieste che effettueranno nel corso della loro esistenza. Ogni  $P$  deve indicare quali e quante risorse di ogni tipo saranno necessarie per terminare il suo compito. Tali informazioni permetteranno di schedulare i processi in modo che non si verifichino deadlock. Se l'esecuzione di un processo può causare deadlock essa viene ritardata opportunamente. I principali algoritmi si differenziano per la quantità e il tipo di informazioni richieste. Il modello più semplice si basa sul forzare tutti i processi a dichiarare il massimo numero di risorse di ciascun tipo di cui il processo avrà bisogno. In genere provocano una riduzione nell'utilizzo delle risorse e una minore efficienza del sistema. Si basano sul concetto di **stato sicuro** e di **sequenza sicura**. Il sistema operativo è in uno stato sicuro se esiste una sequenza sicura, cioè se esiste un ordine ben preciso di tali processi tale per cui tutti possono essere portati a termine senza incorrere in una situazione di stallo.

## Stato sicuro

<b>Stato sicuro</b>	Il sistema è in grado di <ul style="list-style-type: none"> <li>• Allocare le risorse richieste a tutti i processi in esecuzione</li> <li>• Impedire il verificarsi di uno stallo</li> <li>• Trovare una sequenza sicura</li> </ul>
<b>Sequenza sicura</b>	Una sequenza di schedulazione dei processi $\{P_1, P_2, \dots, P_n\}$ tale che per ogni $P_j$ le richieste che esso può ancora effettuare possono essere soddisfatte impiegando le risorse attualmente disponibili più le risorse liberate dai processi $P_j$ con $j < i$

**Stati non sicuri**

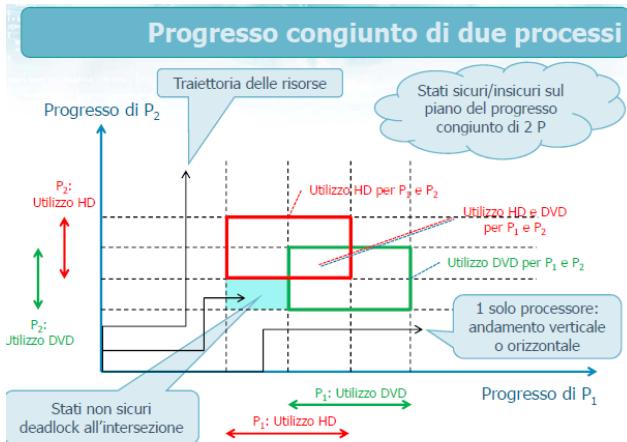
**Stallo**

**Stati sicuri**

Uno stato si dice **non sicuro** in caso contrario.

Non necessariamente uno stato non sicuro è uno stato di stallo: condurrà in uno stato di stallo in caso di comportamento standard.

## Traiettoria delle risorse ☺



Si analizza il progresso congiunto di due processi (in modo da restare sul piano). Nel diagramma si rappresenta: un piano cartesiano con due assi in cui sulla X è rappresentato il progresso di P1 e sulla Y il progresso di P2. Si suppone che il sistema operativo abbia una sola CPU e due soli processi. In questo modo la CPU può essere assegnata solo ad uno dei due processi.

Con la spezzata iniziale (fumetto traiettoria delle risorse) si nota che P1 all'inizio non progredisce, poi tocca a P1 e successivamente di nuovo a P2.

Si suppone inoltre che entrambi P1 e P2 vogliano utilizzare due unità. Gli intervalli di tempo sono rappresentati accanto agli assi.

L'area verde è l'area in cui tanto P1 tanto P2 utilizzano il DVD e l'area rossa è l'area in cui P1 e P2 utilizzano l'HD. Entrambe queste aree sono aree in cui **non** si deve finire e viene definita **area proibita**.

Il sistema all'inizio è in uno stato sicuro, successivamente viene eseguito P2 e lo stato è ancora sicuro perché i sistemi non vanno in deadlock perché la traiettoria non incontra l'area proibita.

Invece, la seconda traiettoria va a finire nell'area celeste e **non** è un'area sicura perché prima o poi una volta entrata in quell'area non si avrà altra scelta che quella di andare in deadlock.

## Strategie

Per evitare uno stallo ci si assicura che il sistema rimanga sempre in uno stato sicuro. All'inizio il sistema è in uno stato sicuro ed ogni richiesta di nuova risorsa sarà soddisfatta se lascerà il sistema in uno stato sicuro, altrimenti sarà ritardata ed il processo che ha effettuato la richiesta sarà posto in attesa. Esistono due classi di strategie:

- Con risorse aventi istanze unitarie
- Con risorse aventi istanze multiple

## Algoritmo per istanze unitarie

Sono basate sulla determinazione di cicli, gestendo il **grafo di rivendicazione**. Tutte le richieste che saranno effettuate **devono** essere dichiarate a priori, ovvero all'inizio dell'esecuzione. Esse sono indicate come archi di reclamo sul grafo di rivendicazione.

Nel momento in cui una richiesta viene effettuata il corrispondente arco di reclamo **dovrebbe** essere trasformato in un arco di assegnazione ma prima di effettuare tale trasformazione si verifica però se soddisfacendola si generano dei cicli. Se il grafo non presenterà comunque alcun ciclo, la conversione verrà effettuata e la risorsa assegnata. In caso negativo, l'assegnazione della risorsa richiesta porterebbe il sistema in uno stato non sicuro e quindi viene rimandata. Quando una risorsa viene rilasciata l'arco di assegnazione ritorna a essere un arco di reclamo (per gestire eventuali richieste successive).



## Algoritmo per istanze multiple ☺

Valutano lo stato del sistema per comprendere se le risorse disponibili sono sufficienti per portare a termine tutti i P. Si basano sul numero di risorse disponibili, assegnate e massimo richiesto. Ogni processo:

- Deve dichiarare a priori il massimo uso di risorse
- Quando richiede una risorsa può essere bloccato
- Quando ottiene le risorse che gli servono deve garantire che le restituirà in un tempo finito

L'algoritmo più utilizzato è quello del **banchiere**. È costituito da due sezioni: la prima verifica che uno stato sia sicuro e la seconda verifica che una nuova richiesta possa essere soddisfatta permettendo al sistema di rimanere in uno stato sicuro. L'algoritmo utilizza le strutture dati elencate di seguito:

Siano dati:  
 • Un insieme di processi  $P_i$  con cardinalità n  
 • Un insieme di risorse  $R_j$  con cardinalità m

Nome	Dim.	Contenuto e significato
<b>Fine</b>	[n]	$\text{Fine}[r]=\text{false}$ indica che $P_r$ non ha terminato
<b>Assegnate</b>	[n][m]	$\text{Assegnate}[r][c]=k$ $P_r$ possiede k istanze di $R_c$
<b>Massimo</b>	[n][m]	$\text{Massimo}[r][c]=k$ $P_r$ può richiedere al massimo k istanze di $R_c$
<b>Necessità</b>	[n][m]	$\text{Necessità}[r][c]=k$ $P_r$ ha bisogno di altre k istanze di $R_c$ $\forall i \forall j \text{ Necessità}[i][j] = \text{Massimo}[i][j] - \text{Assegnate}[i][j]$
<b>Disponibili</b>	[m]	$\text{Disponibili}[c]=k$ disponibilità pari a k per $R_c$

### Esempio

- ❖ Applicando l'algoritmo del banchiere il sistema sottostante si trova in uno stato sicuro?  
 ➤ Sequenza sicura:  $P_1, P_3, P_0, P_2, P_4$

P	Fine	Assegnate	Massimo	Necessità	Disponibili
		$R_0 \ R_1 \ R_2$			
$P_0$	F	0 1 0	7 5 3		3 3 2
$P_1$	F	2 0 0	3 2 2		
$P_2$	F	3 0 2	9 0 2		
$P_3$	F	2 1 1	2 2 2		
$P_4$	F	0 0 2	4 3 3		

Si ha una tabella in cui sono presenti: un vettore di fine, una matrice assegnate, una matrice massimo, una matrice necessità e un vettore disponibili.

Si hanno 5 processi (n=5) e 3 risorse (m=3). Le dimensioni della foto precedente possono essere sostituite con questi parametri.

All'inizio tutti i flag sono False (alla fine tutti true) e indica se un processo ha terminato o meno. Per evitare stalli tutti dovranno avere True al termine. L'ordine di terminazione sarà la **sequenza sicura**. La matrice di assegnate indica le risorse assegnate a ciascun processo per ciascuna tipologia di risorse

in **quel momento**. Ad esempio  $P_0$  ha 0 risorse di tipo  $R_0$ , 1 risorsa di  $R_1$  e 0 risorse di  $R_2$ . Il massimo è ciò che i processi hanno dovuto dichiarare all'inizio. La necessità è ciò che è ancora necessario, mentre le disponibili sono le risorse disponibili una volta che il sistema operativo ne ha già assegnate alcune.

Si parte mettendo tutti i flag a False e si calcolano le Necessità effettuando la differenza tra il massimo e le assegnate:

P	Fine	Assegnate	Massimo	Necessità	Disponibili
		R <sub>0</sub> R <sub>1</sub> R <sub>2</sub>			
P <sub>0</sub>	F	0 1 0	7 5 3	7 4 3	3 3 2
P <sub>1</sub>	F	2 0 0	3 2 2	1 2 2	
P <sub>2</sub>	F	3 0 2	9 0 2	6 0 0	
P <sub>3</sub>	F	2 1 1	2 2 2	0 1 1	
P <sub>4</sub>	F	0 0 2	4 3 3	4 3 1	

Successivamente ci si chiede se tramite le risorse disponibili è possibile far terminare qualcuno dei processi. (esiste qualche processo che ha numero di necessità per risorsa minore delle risorse disponibili?). In questo caso se ne possono far terminare 2 (122 e 011, quindi sia P1 che P3 potrebbero terminare).

Scelgo di terminare P1 e nel momento in cui termina le risorse assegnate verranno liberate e **sommate** a quelle disponibili (e il flag diventa True). Perciò le risorse disponibili diventano **5 3 2**. Successivamente si fa terminare P3 e appena terminerà le disponibili diventeranno **7 4 3**. Successivamente si può terminare P0 che terminerà e passerà a **7 5 3** (cedendo quelle assegnate). Con 7 5 3 è possibile terminare P2 che sarà il 4° a terminare e si arriverà a **10 5 5**. L'ultimo a terminare sarà P4 che porterà le risorse a **10 5 7**. La sequenza: P1 → P3 → P6 → P2 → P4 è una sequenza sicura. Ciò vuol dire che è un modo in cui il sistema può far terminare i processi senza andare in stallo.

Fino ad ora si è vista la prima parte dell'algoritmo del banchiere, cioè verificare se uno stato è sicuro. La seconda verifica che una nuova richiesta possa essere soddisfatta permettendo al sistema di rimanere in uno stato sicuro.

Ci si chiede ad esempio se  $P_1(1,0,2)$  può essere soddisfatta. Per esserlo deve rispettare due condizioni:

- Deve essere minore o uguale alle necessità
- Deve essere minore o uguale alle risorse disponibili

Se entrambe le condizioni sono opportune, la richiesta è soddisfatta.

❖ La richiesta di  $P_1(1, 0, 2)$  può essere soddisfatta?

- Si ...
- Nuovo stato del sistema ...

P	Fine	Assegnate	Massimo	Necessità	Disponibili
		R <sub>0</sub> R <sub>1</sub> R <sub>2</sub>			
P <sub>0</sub>	F	0 1 0	7 5 3	7 4 3	3 3 2
P <sub>1</sub>	F	2 0 0	3 2 2	1 2 2	
P <sub>2</sub>	F	3 0 2	9 0 2	6 0 0	
P <sub>3</sub>	F	2 1 1	2 2 2	0 1 1	
P <sub>4</sub>	F	0 0 2	4 3 3	4 3 1	

Richiesta soddisfatta (aggiorno assegnate e decremento disponibili e necessità):

❖ Il nuovo stato è sicuro?

➤ Sequenza sicura:  $P_1, P_3, P_0, P_4, P_2$

P	Fine	Assegnate	Massimo	Necessità	Disponibili
		R <sub>0</sub> R <sub>1</sub> R <sub>2</sub>			
P <sub>0</sub>	F	0 1 0	7 5 3	7 4 3	2 3 0
P <sub>1</sub>	F	3 0 2	3 2 2	0 2 0	
P <sub>2</sub>	F	3 0 2	9 0 2	6 0 0	
P <sub>3</sub>	F	2 1 1	2 2 2	0 1 1	
P <sub>4</sub>	F	0 0 2	4 3 3	4 3 1	

Adesso ci si chiede nuovamente se lo stato è sicuro, andando a verificare se esiste una sequenza sicura che fa sì che questo stato sia sicuro. Perciò si ripete lo stesso algoritmo precedente con questa nuova configurazione.

P	Fine	Assegnate	Massimo	Necessità	Disponibili
		R <sub>0</sub> R <sub>1</sub> R <sub>2</sub>			
P <sub>0</sub>	X T	0 1 0	7 5 3	7 4 3	2 3 0
P <sub>1</sub>	X T	3 0 2	3 2 2	0 2 0	5 3 2
P <sub>2</sub>	X T	3 0 2	9 0 2	6 0 0	7 4 3
P <sub>3</sub>	X T	2 1 1	2 2 2	0 1 1	7 5 3
P <sub>4</sub>	X T	0 0 2	4 3 3	4 3 1	1 0 5 5

Handwritten annotations: circled numbers 3, 1, 4, 2, 7; circled P1, P3, P0, P2, P4; arrows pointing from P1 to P3, P3 to P0, P0 to P2, P2 to P4.

Altri esempi:

Esercizio  
Stesso stato iniziale

❖ La richiesta  $P_4(3, 3, 0)$  può essere soddisfatta?

➤ No ... non c'è disponibilità

❖ La richiesta  $P_0(0, 3, 0)$  può essere soddisfatta?

➤ Si ... ma lo stato risultante non è sicuro

P	Fine	Assegnate	Massimo	Necessità	Disponibili
		R <sub>0</sub> R <sub>1</sub> R <sub>2</sub>			
P <sub>0</sub>	F	0 1 0	7 5 3	7 4 3	2 3 0
P <sub>1</sub>	F	3 0 2	3 2 2	0 2 0	
P <sub>2</sub>	F	3 0 2	9 0 2	6 0 0	
P <sub>3</sub>	F	2 1 1	2 2 2	0 1 1	
P <sub>4</sub>	F	0 0 2	4 3 3	4 3 1	

## Verifica di una richiesta da parte di $P_i$ in pseudocodice

```

se
   $\forall_j \text{Richieste}[i][j] \leq \text{Necessità}[i][j]$ 
  AND
   $\forall_j \text{Richieste}[i][j] \leq \text{Disponibili}[j]$ 
  ALLORA
     $\forall_j \text{Disponibili}[j] = \text{Disponibili}[j] - \text{Richieste}[i][j]$ 
     $\forall_j \text{Assegnate}[i][j] = \text{Assegnate}[i][j] + \text{Richieste}[i][j]$ 
     $\forall_j \text{Necessità}[i][j] = \text{Necessità}[i][j] - \text{Richieste}[i][j]$ 

se lo stato risultante è sicuro
  si conferma tale assegnazione
  altrimenti si ripristina lo stato precedente
  
```

## Verifica di uno stato in pseudocodice

```

1.
 $\forall i \forall j \text{ Necessità}[i][j] = \text{Massimo}[i][j] - \text{Assegnate}[i][j]$ 
 $\forall i \text{ Fine}[i] = \text{false}$ 

2.
Trova  $i$  per cui
 $\text{Fine}[i] = \text{falso}$  AND  $\forall j \text{ Necessità}[i][j] \leq \text{Disponibili}[j]$ 
Se tale  $i$  non esiste goto step 4

3.
 $\forall j \text{ Disponibili}[j] = \text{Disponibili}[j] + \text{Assegnate}[i][j]$ 
 $\text{Fine}[i] = \text{true}$ 
goto step 2

4.
Se  $\forall i \text{ Fine}[i] = \text{true}$  il sistema è in uno stato sicuro
  
```

## ESERCIZIO

Esercizio							Esercizio						
<b>P</b>	<b>Fine</b>	<b>Assegnate</b>	<b>Massimo</b>	<b>Necessità</b>	<b>Disponibili</b>	<b>P</b>	<b>F</b>	<b>A</b>	<b>M</b>	<b>N</b>	<b>D</b>		
		R <sub>0</sub> R <sub>1</sub> R <sub>2</sub>	P <sub>0</sub>	F	3	9		3					
P <sub>0</sub>	F	1 0 0	3 2 2		1 1 2	P <sub>1</sub>	F	2	4				
P <sub>1</sub>	F	5 1 1	6 1 3			P <sub>2</sub>	F	2	7				
P <sub>2</sub>	F	2 1 1	3 1 4			P <sub>3</sub>	F	0 0 2	4 2 2				

❖ La richiesta P<sub>1</sub> (1, 0, 1) può essere soddisfatta?  
 ➤ Si ...

❖ La richiesta P<sub>2</sub> (1, 0, 1) può essere soddisfatta?  
 ➤ Si ... ma lo stato risultante non è sicuro

❖ I seguenti due stati sono sicuri o non sicuri?  
 (problemi risorsa unica)

... stato sicuro

... stato non sicuro

## Conclusioni

La complessità dell'algoritmo del banchiere è

$$O(m \cdot n^2) = O(|R| \cdot |P|^2)$$

Si basa inoltre su ipotesi poco realistiche:

- I processi devono indicare le richieste in anticipo: le risorse necessarie non sempre sono note ed inoltre non è noto quando saranno necessarie
- Suppone le risorse siano in numero costante: le risorse possono aumentare o ridursi a causa di guasti temporanei o duraturi
- Richiede una popolazione fissa di processi: i processi attivi nel sistema aumentano e si riducono in maniera dinamica