

SISTEMI OPERATIVI

1) INTRODUZIONE e TERMINOLOGIA

Il **SISTEMA OPERATIVO (OS)** è un software eseguito in modalità protetta (**kernel mode**) che controlla e coordina l'uso dell'hardware. Da un punto di vista top-down, possiamo vederlo come una macchina estesa (astrae i vari dispositivi del sistema); da un punto di vista bottom-up, possiamo considerarlo come il gestore delle risorse, ovvero un insieme di moduli, ognuno dei quali fornisce determinati **servizi**:

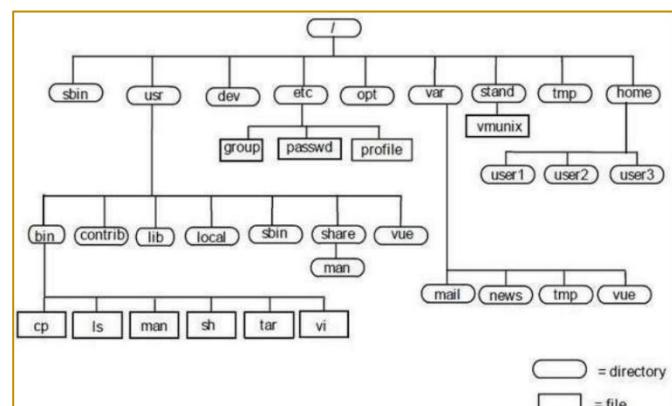
- interprete dei comandi (utente effettua il proprio lavoro mediante un interprete di comandi);
- gestione dei processi (processo [unità attiva] = programma [unità passiva] in esecuzione);
- gestione della memoria principale (e secondaria);
- gestione dei dispositivi I/O;
- gestione dei file e filesystem;
- meccanismi di protezione;
- gestione delle reti.

Vediamo i termini legati ai sistemi operativi:

- **KERNEL** → nucleo del sistema operativo. Tramite livelli di astrazione, fornisce ai processi in esecuzione un accesso sicuro all'hardware; gestisce i processori, il multitasking e la memoria. È l'unico programma in esecuzione per tutto il tempo. Ce ne sono vari tipi:
 - **Kernel monolitico** = realizza i servizi con moduli separati (ma ben integrati), mediante le system call (o syscall) [Unix, Linux];
 - **Microkernel** = definisce delle macchine virtuali sopra l'hardware (lento, ma stabile);
 - **Kernel ibrido** = approccio intermedio tra i due precedenti [Windows NT];
 - **Esokernel** = sistemi operativi verticali (separano la protezione dalla gestione).
- **BOOTSTRAP** → carica il kernel in memoria centrale all'accensione del computer [firmware nella ROM];
- **SYSTEM CALL (syscall)** → sono gli entry-point dell'OS (spesso vi si accede con un API di alto livello). Si differenziano dalle funzioni di libreria perché:
 - non possono essere modificate o sostituite;
 - forniscono funzionalità di base;
 - vengono eseguite in "**KERNEL MODE**" [bit di modo = 0], mentre le funzioni di libreria in "**USER MODE**" [bit di modo = 1] (infatti l'OS si protegge lavorando in dual-mode);

Le syscall sono effettuate mediante un'interruzione software, detta "**trap**", la quale passa in kernel-mode cambiando il bit di modo, prende il codice della syscall richiamata (che identifica usando una tabella apposita nel kernel space) e ritorna in user-mode [11 passi di esecuzione].

- **LOGIN (autenticazione)** → procedura di accesso ad un sistema o ad una sua applicazione;
- **SHELL** → interprete "command-line" dei comandi dell'utente (che possono essere digitati sul terminale oppure letti da un file di "script"). Non fa parte dell'OS;
- **FILESYSTEM** → struttura gerarchica a grafo aciclico in cui sono organizzati **file** e **directories (direttori)**. La sua radice (root) è indicata con "/" [infatti nei nomi dei file non possiamo usare né "/" né il carattere "null"]. Dentro "home" troviamo gli utenti del computer, mentre dentro "bin" troviamo i comandi di shell; dopo il login, l'utente accederà alla sua **HOME DIRECTORY** (per l'utente user1 sarà /home/user1 e corrisponde al simbolo ~).



I nomi dei **percorsi** (**path**) di sistema seguono delle regole:

- “.” = directory **corrente**, mentre “..” = directory **padre** (livello sopra quello corrente);
- possono essere specificati in 2 modalità:
 - **ASSOLUTA** = parto dalla “/” (root);
 - **RELATIVA** = parto dalla home directory (directory corrente) e mi sposto con “.” O “..”;

Un esempio può essere fatto con l'accesso tramite **comandi** (supponendo che posizione corrente sia in user1):

- **ls** ././user2 [relativo] oppure **ls** /home/user2 [assoluto] **elenca** i file e le directories in user2;
- **cp** ./././usr/bin/tar ././user2 mi **copia** il contenuto di tar (partenza) dentro user2 (arrivo);
- **cd** ./././usr/bin [relativo] oppure **cd** /usr/bin [assoluto] **mi sposta** di cartella da corrente a bin.
- **PROGRAMMA** → entità **passiva**; file **eseguibile** (su disco) contenente operazioni per realizzare un algoritmo. Può essere **sequenziale** (ogni istruzione inizia dopo la precedente [ciclo di fetch, decode e execute]) o **parallelo** [concorrente] (più istruzioni eseguite contemporaneamente, la cui relazione temporale è specificata da un **grafo di precedenza delle tasks**). Inoltre definiamo **atomica** un'operazione che non può essere interrotta.
- **PROCESSO** → entità **attiva**; **programma in esecuzione**. Nei sistemi UNIX, ogni processo è rappresentato da un **identificatore** intero univoco (non negativo). La relazione tra processi è rappresentata con l'**albero dei processi** (al bootstrap esiste un unico processo).
- **THREAD** (o **processo leggero**) → **flusso di esecuzione** (legato al program counter); ogni processo può contenere **più threads** (quindi più program counters). Ogni thread ha un **identificatore locale al processo**.
- **PIPE** → **flusso di dati tra 2 processi** [è uno **pseudo-file**]; è un **canale di comunicazione** di tipo **half-duplex** (ovvero da P_1 a P_2 oppure da P_2 a P_1) [simplex solo un verso, full-duplex entrambi i versi in contemporanea].
- **DEADLOCK** (**stallo**, impasse) → avviene quando 2 entità si **bloccano** a vicenda. Caso particolare è il **LIVELOCK** (stallo **attivo**), dove le 2 entità si fermano, ma non sono effettivamente bloccate (es. quando 2 persone si incontrano in corridoio e fanno passi nella stessa direzione, nonostante lo spazio per passare entrambi ci sia).
- **STARVATION** (**fame**, inedia) → ad un'entità viene rifiutato **l'accesso ad una risorsa necessaria** al suo progresso (**starvation** non implica deadlock, ma deadlock implica starvation).

2) COMANDI LINUX

Partiamo dal presupposto di usare **Ubuntu su VirtualBox** (hypervisor che funziona da host di Linux). Per quanto riguarda le **SESSIONI DI LAVORO** posso iniziare una con l'autenticazione (**login** e **password**) [anche da terminale remoto con **ssh -X <username@hostname>**] e terminarla con **exit** (o **logout** o **ctrl-d**). Per quanto riguarda i **COMANDI**, questi sono tutti **documentati** (**man** comando oppure comando **--help**) e hanno struttura **comando [opzioni] [argomenti]**, dove:

- **OPZIONI** → specificano azioni/effetti particolari (opzionali) e hanno 2 formati:
 - specificate da un solo carattere con **-carattere** [es. **-d**];
 - specificate da una stringa con **--stringa** [es. **--ld**].
- **ARGOMENTI** → specificano i target su cui il comando deve agire (opzionali).

⚠ I comandi troppo lunghi possono essere continuati sulla riga successiva mettendo come ultimo carattere della riga corrente un “\”. Inoltre si possono fornire più comandi sulla stessa riga separandoli con “;”.

Ora vediamo una serie di **COMANDI UTILI** (e le **tabelle delle loro opzioni**):

- **ls [opzioni] [file...]** → **elenca il contenuto di una cartella**. Se il nome del file inizia con il carattere “.” significa che il file è nascosto; invece file obsoleti sono spesso rinominati automaticamente appendendo al nome il carattere “~”. Quello che vediamo come risultato della **ls** è:

Opzioni		
Formato	Significato	Effetto
Compatto	Esteso	
	--help	Help in linea
-a	--all	Elenca anche i file che iniziano per .
-l		Long list Format
-g	--group-directories-first	Include l-indicazione de gruppo prima di quella del file
-t		Newest first
-r	--reverse	Ordine inverso (alfabetico o temporale)
-R	--recursive	Elenca anche i file nei sottodirectory

Per quanto riguarda il tipo e diritti, stiamo parlando dei **PERMESSI**; i permessi sono composti da **1 carattere iniziale che ci indica il tipo di oggetto** (- = normal file, **d** = directory, **s** = socket file, **l** = link file) e da **3 terne** che in ordine si riferiscono a **u** = user (owner), **g** = group, **o** = others. **Ognuna** di queste terne è composta da **3 caratteri che ci dicono i permessi dati agli utenti sopra citati** (**r** = read, **w** = write, **x** = execute). Esistono comandi per **cambiare le proprie generalità** e gestire i permessi: **su username** (per diventare un utente diverso), **sudo comando** (per eseguire comandi come superuser), **sudo -u user comando** (per eseguire comando nelle vesti di un altro utente) e **whoami** (per conoscere il proprio username). Per **gestire i permessi** abbiamo i seguenti comandi (con opzione possibile **-r**, **-R**, ovvero recursive [cambiare ricorsivamente anche nei sottodirettori]):

- **chmod [opzioni] permessi file** → **cambiare i permessi di file/directory**. I nuovi permessi possono essere scritti con 3 cifre ottali (es. chmod 775 file) oppure in modo simbolico con una stringa di caratteri indicando verso chi è rivolto il permesso (u = user, g = group, o = other, a = all), in cosa consiste (r, w, x) e eventuali combinazioni tra questi 2 campi (+ = aggiungo, - = tolgo, =);
 - **chown [opzioni] utente entry** → **cambiare il proprietario di una entry**;
 - **chgrp [opzioni] gruppo entry** → **cambiare il gruppo di una entry**.
- **cp [opzioni] src1 src2 ... dest** → **copiare** 1 o più file dalla posizione corrente (source) alla destinazione;
 - **rm [opzioni] src1 src2 ...** → **rimuovere** 1 o più file dalla posizione corrente;
 - **mv [opzioni] src1 src2 ... dest** → **muovere** 1 o più file dalla posizione corrente alla destinazione.
- **cd dest** → **cambiare directory corrente**;
 - **mkdir dir** → **creare directory**;
 - **rmdir dir** → **rimuovere directory** (rimosso solo se è vuoto, altrimenti usare rm -rf)
 - **pwd** → **mostrare il nome della directory corrente**.
- **cat file1 file2** → **concatenare file**;
 - **head [opzioni] file** → **visualizzare le n righe iniziali** di un file (default n = 10);
 - **tail [opzioni] file** → **visualizzare le n righe finali** di un file (default n = 10).
- **pg [opzioni] file** → **mostra contenuto file una pagina alla volta**;
 - **more [opzioni] file** → **visualizzare il contenuto di 1 o più file**;
 - **less [opzioni] file** → **come more ma con le frecce direzionali** per muoversi nel testo già visualizzato.
- **diff [opzioni] entry1 entry2** → **controlla differenze tra 2 file/direttori**;
 - **wc [opzioni] [file]** → **mostra contenuto file una pagina alla volta**;

Total Number of Blocks (default size 1024 bytes)	User (owner) name	Owner group	Entry name
quer@fi:group:~/www'	quer	fmgroup	ls -la
total 72			
drwxr-xr-x 8 quer fmgroup 4096 Oct 7 2013 .			
drwxr-xr-x 34 quer fmgroup 4096 Oct 3 12:37 ..			
drwxr-xr-x 2 quer fmgroup 4096 Oct 15 2009 file			
-rw-r--r-- 1 quer fmgroup 17715 Oct 7 2013 index.htm			
drwxr-xr-x 2 quer fmgroup 4096 Mar 22 2013 misc			
drwxr-xr-x 2 quer fmgroup 4096 Jun 25 2009 paper			
drwxr-xr-x 3 quer fmgroup 4096 May 30 2012 research			
-rw-r--r-- 1 quer fmgroup 18074 Apr 28 2005 stq.jpg			
drwxr-xr-x 10 quer fmgroup 4096 Jun 5 14:56 teaching			
drwxr-xr-x 2 quer fmgroup 4096 Jun 2 20:49 tmp			

Opzioni		
Formato	Significato	Effetto
Compatto	Esteso	
	--help	Help in linea
-f	--force	Effettua le operazioni senza chiederne conferma
-i	--interactive	Chiede conferma prima di effettuare qualsiasi operazione
-r, -R	--recursive	Ricorsivo Procede ricorsivamente anche nei sottodirettori

Opzioni		
Formato	Significato	Effetto
Compatto	Esteso	
	--lines	Specifica il numero di righe. Se n<0 si considerano tutte le righe del file tranne n all'altro estremo. Esempio: • head -n 2 file → visualizza le prime due righe di file. • head -n -2 file → visualizza tutte le righe tranne le ultime due
-f	--follow	Rileggendo continuamente il file aggiornando l'output se il file viene modificato

Opzioni		
Formato	Significato	Effetto
Compatto	Esteso	
spazio		Prossima riga
return		Prossima riga
B		Pagina precedente
/str		Ricerca nel testo la prossima occorrenza di str
?str		Ricerca nel testo la precedente occorrenza di str
q		Termina la visualizzazione

Opzioni		
Formato	Significato	Effetto
Compatto	Esteso	
-q	--brief	Indica solo se gli oggetti sono differenti
-b	--ignore-space-change	Ignora gli spazi a fine riga, collappa gli altri
-i	--ignore-case	Ignora la differenza tra maiuscole e minuscole
-w	--ignore-all-space	Ignora completamente ogni tipo di spaziatura
-B	--ignore-blank-lines	Ignore le righe di soli spazi

Opzioni		
Formato	Significato	Effetto
Compatto	Esteso	
-c	--bytes	Valuta il numero di soli byte
-m	--chars	Valuta il numero di soli byte
-w	--words	Valuta il numero di parole
-l	--lines	Valuta il numero di righe

In UNIX esistono **2 tipi di link**:

- **SOFT-LINK** (simbolico) = file che contiene il percorso di un altro file/direttorio;
- **HARD-LINK** (fisico) = associazione tra il nome di un file/direttorio e il suo contenuto.

Con **ln [opzioni] source [destination]** → **crea un link** (di default si crea un hard link).

Opzioni		
Formato		Significato
Compatto	Esteso	Effetto
	--help	Visualizza un help in linea
-s	--symbolic	Crea un link simbolico (soft link)
-f	--force	Rimuove eventuali file di destinazione esistenti
-d, -F	--directory	Permette al super-user di provare a generare un hard-link con un directory; probabilmente fallirà a causa delle restrizioni del sistema

⚠ Infatti il comando **rm** rimuove un file solo se il numero degli hard link è zero, mentre il comando **mv** equivale ai comandi **ln + rm!**

Per quanto riguarda la **COMPRESIONE** abbiamo:

- **tar -czvf <file>.tgz <dir>** → creazione verbosa di zip su file;
- **tar -xzvf <file>.tgz <dir>** → estrazione verbosa di zip su file.

Opzioni		
Formato		Significato
Compatto	Esteso	Effetto
-c		Crea l'archivio
-x		Estra l'archivio
-z, -J, -j		Comprime (gzip, bzip2, 7z)
-f		Specifica il nome dell'archivio
-v		Verbose (stampa i messaggi)

Per quanto riguarda l'**OCCUPAZIONE DI SPAZIO SU DISCO**:

- **df [opzioni] disco** → controlla occupazione del disco;
- **du [opzioni] directory** → spazio occupato da una directory su disco.

3) PROGRAMMARE C SU LINUX

Su LINUX potrò comunque usare **VSCode** (comodo!). Il comando **gcc <opzioni> <argomenti>** è il comando di compilazione e linker generico per il C; vediamo un esempio:

```
gcc -Wall -g -I. -I/myDir/subDir -o myexe \
  myMain.c \
  fileLib1.c fileLib2.c file1.c \
  file2.c file3.c -lm
```

Il debug è indicato da **-g**; con **-I** si indica la posizione dei file.h (library) mentre con **-o** si indica dov'è l'eseguibile. I file.c scritti sotto vengono compilati e linkati insieme.

In C abbiamo sempre usato il **MAKEFILE** (CMake) ma non lo abbiamo mai visto nel dettaglio: mediante la verifica delle dipendenze e l'istante dell'ultima modifica, il makefile evita di gestire ripetutamente operazioni ripetitive e inutili. Si procede in 2 fasi: **si scrive un file Makefile** (file di testo simile ad uno script) e **si interpreta questo file con l'utility make** (effettuando compilazione e link). Con il comando **make --file <nomeFile>** possiamo eseguire il Makefile di nome specificato (non quello di default). Per **scrivere un Makefile** devo rispettare delle regole: le righe bianche e quelle che iniziano con **#** vengono ignorate (**commenti**) [inoltre anche qui se riga troppo lunga posso andare a capo con il carattere ****]; le righe contengono delle regole e ogni **REGOLA** è composta dal nome del target (file o azione), dalla **lista delle dipendenze da verificare** prima di eseguire i comandi relativi alla regola e dal **comando** (o elenco di comandi) [**ogni comando deve essere preceduto dal carattere tab**]. Vediamo esempi:

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

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

⚠ Qui infatti eseguendo **make** viene eseguita la 1^ regola (project1), mentre se voglio eseguire la 2^ devo specificarla con **make project2!**

```
target: file1.o file2.o
  <tab>gcc -Wall -o myExe file1.o file2.o

file1.o: file1.c myLib1.h
  <tab>gcc -Wall -g -I./dir1 -c file1.c

file2.o: file2.c myLib1.h myLib2.h
  <tab>gcc -Wall -g -I./dir1 -c file2.c
```

⚠ Qui vediamo come funzionano le **DIPENDENZE** (se le **dipendenze del target sono più recenti del target stesso, si eseguono le dipendenze prima dei comandi** procedendo in maniera ricorsiva): qui il target ha come dipendenze file1.o e file2.o; dunque si verifica la regola file1.o, dove se file1.c è più recente di file1.o si esegue la regola (ovvero il comando **gcc**), altrimenti non si esegue (analogo per la regola file2.o).

Se il target non è il nome di un file è un “**phony**” target (ovvero che dovrebbe essere sempre eseguito) [es. il target di nome “target” che abbiamo visto finora]. In questi comandi posso anche memorizzare stringhe (ovvero comandi) come delle **MACRO** che posso richiamare con **\$ [MACRO]** [es. **C = gcc -g** → **\$ (C)** sarà equivalente a scrivere **gcc -g**].

⚠ Per il DEBUG possiamo usare il debugger **gdb** che può ricevere i seguenti comandi:

Azione	Comandi
Comandi esecuzione step-by-step	run (r) next (n) next <numeroStep> step (s) step <numeroStep> stepi (si) finish (f)
Comandi per breakpoint	continue (c) info break break (b), ctrl-x-blank break numeroLinea break nomeFunzione fileName:numeroLinea disable numeroBreak enable numeroBreak
Comandi di stampa	print (p) print espressione display espressione
Operazioni sullo stack	down (d) up (u) info args info locals
Comandi di listing del codice	list (l) list numeroLinea list primaLinea, ultimaLinea
Comandi per operazioni varie	file fileName exec fileName kill

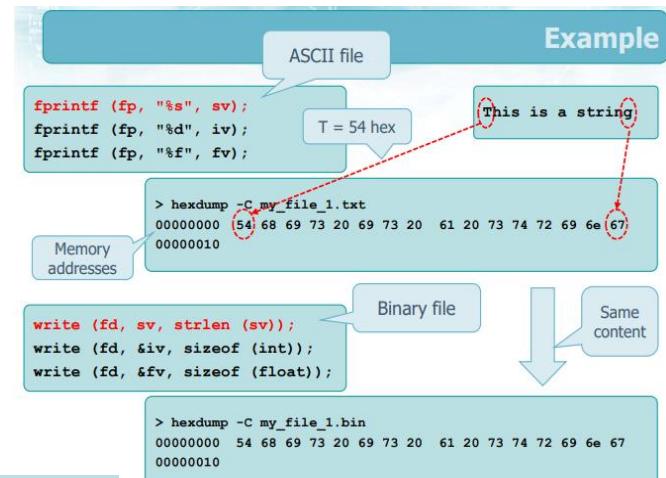
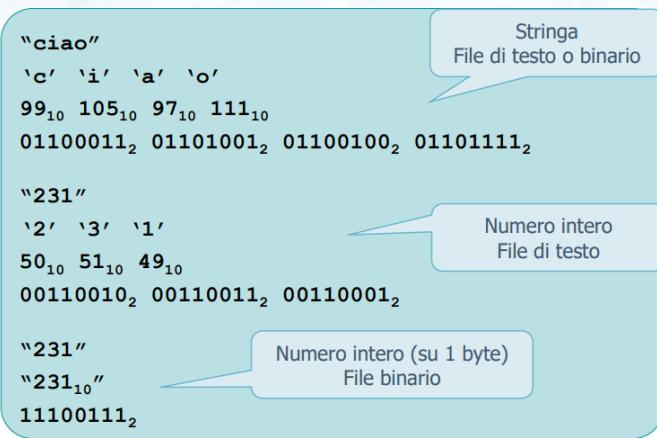
4) FILE SU LINUX

Abbiamo già accennato qualcosa sul **FILESYSTEM** (gestione e memorizzazione dei file, direttori e partizioni di dischi). I **file** memorizzano informazioni a lungo termine (anche quando viene spenta la macchina); a livello di questo corso sono importanti 2 aspetti, ovvero qual è il sistema di codifica (**CODIFICA**) usato nel file e quali sono i metodi di utilizzo dello spazio (**INDIRIZZAMENTO**):

1. **CODIFICA** → lo standard era l'**ASCII** (ma a seconda della regione ne esistono versioni diverse) o meglio dire l'**EXTENDED ASCII** (8-bit e 255 caratteri). Ma dato che questi caratteri sono pochi, oggi si preferisce l'**UNICODE** (3 tipi: UTF-8 [8 bit, come ASCII], UTF-16, UTF-32). Un file è una serie di byte scritti uno dopo l'altro, ma ci sono 2 tipi di file:

- **TESTO** (o ASCII) [C, Java, ...] → sequenze di 0 e 1 che codificano dei caratteri ASCII. Solitamente sono "line-oriented" (qui la più piccola unità è il byte [carattere ASCII]);
- **BINARI** [eseguibili, word, excel etc...] → sequenze di 0 e 1 non "byte-oriented", ovvero non necessariamente gli 8 bit corrispondono a caratteri stampabili (qui la più piccola unità è il bit). Rispetto ai file di testo sono più compatti, più facili da modificare e più facile posizionarsi sul file, ma hanno anche portabilità limitata e non sono editabili in editor standard;

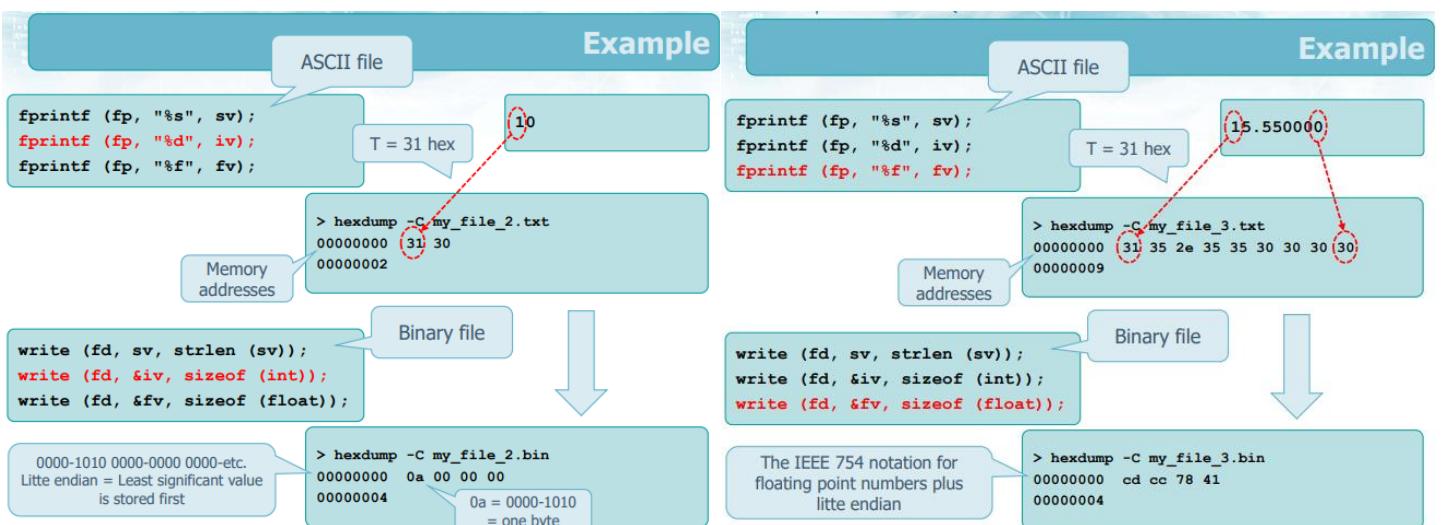
⚠ Il kernel **Linux** non distingue tra file di testo e binari; inoltre il **guadagno effettivo dei file binari è sui numeri** (le stringhe vengono trattate allo stesso modo); sotto vediamo anche la **differenza di apertura in C di un file di testo e binario e dei metodi di scrittura di stringhe, int e float**:



```

fp = fopen ("my_file_1.txt", "w");
fprintf (fp, ...);
fclose (fp);

fd = open ("my_file_1.bin", O_WRONLY|O_CREAT|O_TRUNC,
           S_IRUSR|S_IWUSR);
write (fd, ...);
close (fd);
    
```



Per quanto riguarda il processo di traduzione di una **struttura** (es. in C la struct), quando io la scrivo in memoria senza aggiungere informazioni (ovvero solo la dichiarazione) devo avere un modo di tradurla in un **formato memorizzabile**, per poi riempirla associandole dei valori; alcuni linguaggi di programmazione (Java, Python, C++, ...) supportano questo processo, chiamato **SERIALIZZAZIONE** (dove ho dei caratteri aggiuntivi apposta che mi permettono la scrittura della struttura dati su file) [quindi noi scriveremo delle strutture in modo compatto con la write e le leggeremo in modo compatto con la read, facendo a mano delle operazioni simili alla serializzazione, scrivendo in binario].

Lo standard I/O è “**fully buffered**”. Devo quindi iniziare i programmi C con:

Per quanto riguarda le **funzioni C di I/O** abbiamo:

<code>FILE *fopen (char *path, char *type);</code>	<code>int getc (FILE *fp);</code>	<code>char gets (char *buf);</code>
<code>FILE *fclose (FILE *fp);</code>	<code>int fgetc (FILE *fp);</code>	<code>char *fgets (char *buf, int n, FILE *fp);</code>
<code>int scanf (char format, ...);</code>	<code>int putc (int c, FILE *fp);</code>	<code>int puts (char *buf);</code>
<code>int fscanf (FILE *fp, char format, ...);</code>	<code>int fputc (int c, FILE *fp);</code>	<code>int *fputs (char *buf, FILE *fp);</code>
<code>int printf (char format, ...);</code>	<code>size_t fread (void *ptr, size_t size,</code>	
<code>int fprintf (FILE *fp, char format, ...);</code>	<code>size_t nObj, FILE *fp);</code>	
	<code>size_t fwrite (void *ptr, size_t size,</code>	
	<code>size_t nObj, FILE *fp);</code>	

Per i FILE BINARI

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

Per i file binari, usiamo appunto le **read/write di intere strutture** mediante una singola operazione (potenziali problemi di compatibilità nel gestire architetture diverse) (il return è **size_t** ovvero il **numero di oggetti letti/scritti**).

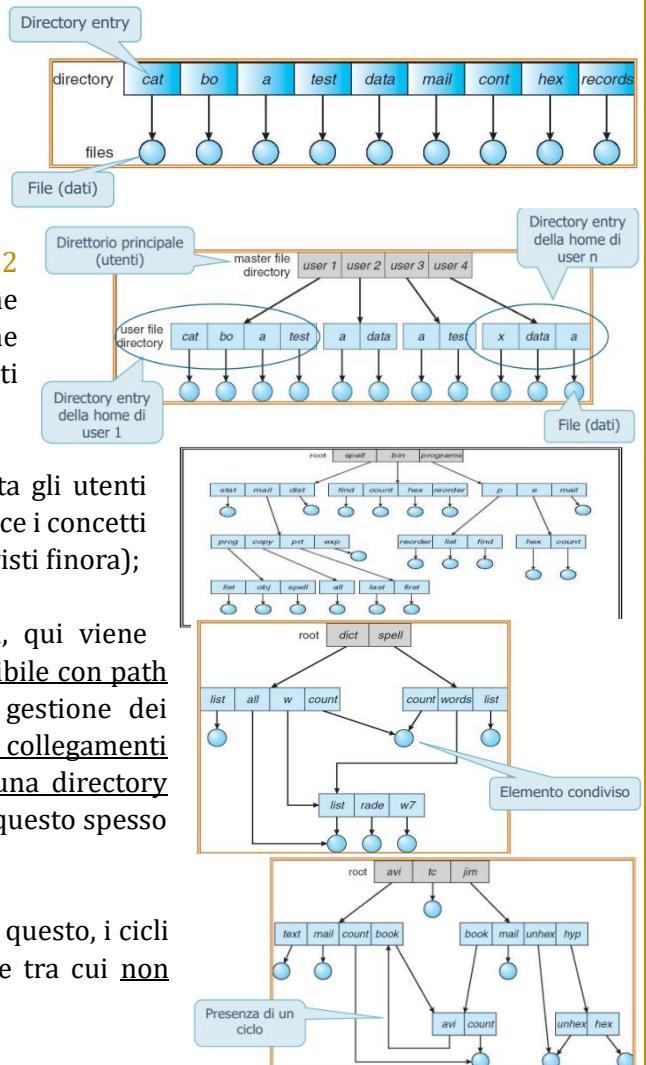
Visto che noi parleremo del **POSIX** (più che del C), qui l'**I/O UNIX** si può effettuare interamente con **5 funzioni di libreria** [**costruite sulle omonime syscall**] (ricordando che nel kernel UNIX un “**file descriptor**” è un intero non negativo (0 = **STDIN**, 1 = **STDOUT**, 2 = **STDERR**):

- `int open (const char *path, int flags, mode_t mode)` → apre il file del path, ritornando il file descriptor, -1 in caso di errore. Il parametro flags si ottiene con l'OR bit-a-bit di costanti nell'header `fcntl.h`; la mode specifica i diritti di accesso (i permessi);
- `int read (int fd, void *buf, size_t nbytes)` → legge dal file fd un n° di byte uguale a nbytes, memorizzandoli in un buffer (buf). Ritorna il numero di byte letti in caso di successo, -1 in caso di errore e 0 per l'EOF;
- `int write (int fd, void *buf, size_t nbytes)` →scrive nbytes byte contenuti in buf nel file descriptor fd. Ritorna il numero di byte scritti in caso di successo, -1 in caso di errore;

- **off_t lseek (int fd, off_t offset, int whence)** → ogni file ha associata una posizione corrente del file offset (indica la posizione di partenza della prossima read/write). La lseek assegna un nuovo valore al file offset. Ritorna il nuovo offset in caso di successo, -1 in caso di errore;
- **int close (int fd)** → chiude il file descriptor fd (tutti i file sono chiusi automaticamente quando il processo termina). Ritorna 0 in caso di successo, -1 in caso di errore.

2) **INDIRIZZAMENTO** → i file sono organizzati in **directories** (direzioni), ovvero dei **nodi** (di un albero) o dei **vertici** (di un grafo) contenente le informazioni sugli elementi al suo interno. La struttura di un directory dipende da ragioni da **EFFICIENCY**, **NAMING** e **GROUPING**; abbiamo:

- **Direttori a 1 livello** = tutti i file sono contenuti nello stesso **direttorio** (file evidenziati dalla directory entry [il nome univoco del file] e dai dati [tramite puntatore]). Struttura facile, i file devono avere nomi univoci e la gestione dei file degli utenti è difficile;
- **Direttori a 2 livelli** = i file sono contenuti in un **albero a 2 livelli**, dove ogni user può avere dunque la sua home directory (tutte le operazioni sono eseguite solo sulla home corretta). Sistema "user-oriented", nomi ripetibili se di utenti diversi;
- **Direttori ad albero** = generalizza i precedenti (in aggiunta gli utenti possono gestire anche le directory oltre che i file) e introduce i concetti di working directory, path assoluto e relativo etc... (quelli visti finora);
- **Direttori a grafo aciclico** = a differenza dei precedenti, qui viene permessa la **condivisione di informazioni** (rendendola visibile con path diversi). La presenza di **link** aumenta la difficoltà di gestione dei filesystem (bisogna distinguere gli oggetti nativi dai loro collegamenti per la loro gestione) [inoltre la creazione di un link a una directory potrebbe causare la nascita di un **ciclo** nel filesystem, per questo spesso sono vietati];
- **Direttori a grafo ciclico** = permette di creare dei **cicli** e per questo, i cicli vanno gestiti in tutte le fasi (diversi approcci di gestione tra cui non visitare mai i link).



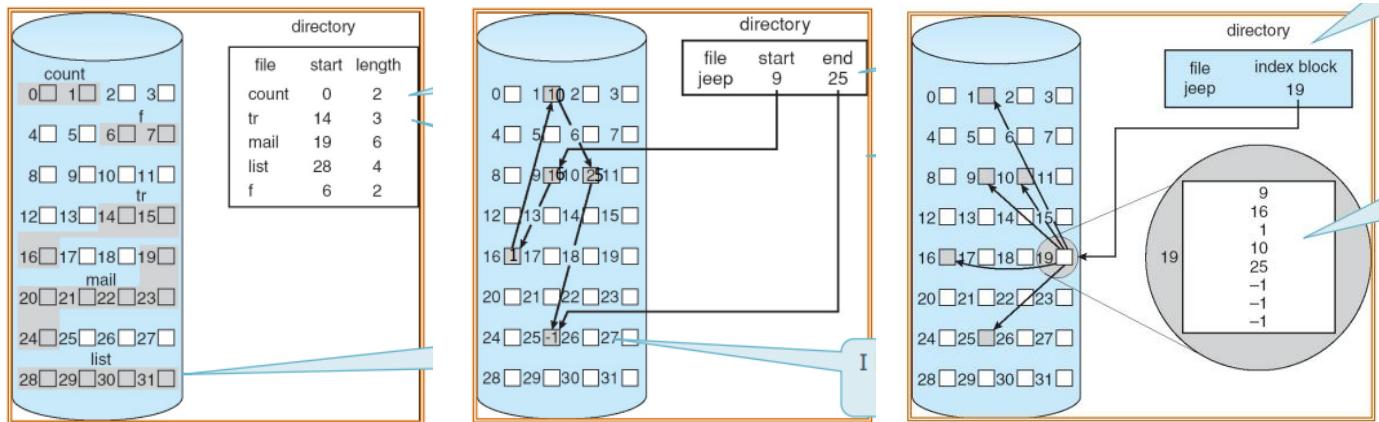
Parlando di come vengono memorizzati i file su disco (**ALLOCAZIONE**) abbiamo 3 tecniche principali:

- **CONTIGUA (contiguous)** → ogni file occupa un insieme contiguo di blocchi (direttorio specifica l'indirizzo del 1° blocco e la lunghezza del file).
 - **PRO:** allocazione semplice, permette accessi sequenziali immediati e accessi diretti semplici
 - **CONTRO:** occorre decidere l'allocazione, viene sprecato spazio (frammentazione esterna), problemi di allocazione dinamica
- **CONCATENATA (linked)** → ogni file può essere allocato gestendo una lista concatenata di blocchi (direttorio contiene un puntatore al 1° e all'ultimo blocco del file, mentre ogni blocco contiene un puntatore al blocco successivo [i blocchi di ciascun file sono sparsi per l'intero disco]).
 - **PRO:** risolve i problemi dell'allocazione contigua (permette l'allocazione dinamica ed elimina la frammentazione esterna)
 - **CONTRO:** ogni lettura implica un accesso sequenziale ai blocchi (efficiente solo per accessi sequenziali) e c'è memorizzazione dei puntatori (meno spazio utile disponibile)

⚠ Un tipo di allocazione concatenata è la **FAT** (File Allocation Table = tabella con un elemento per ciascun blocco presente sul disco); qui i riferimenti non sono memorizzati nei blocchi su disco, ma direttamente negli **elementi della FAT**. Il problema è l'affidabilità e la dimensione della FAT.

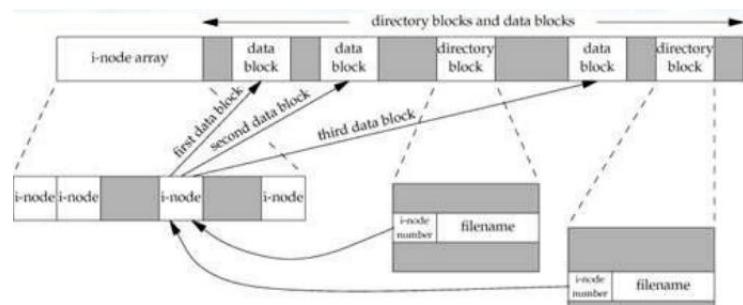
- **INDICIZZATA (linked)** → ingloba tutti i puntatori in una tabella (tabella di puntatori denominata **BLOCCO INDICE (i-node)**). Ogni file ha la sua tabella, ovvero un vettore di indirizzi dei blocchi in cui il file è contenuto (direttorio contiene il solo puntatore al blocco indice). Usare blocchi indice piccoli permette di usare meno spazio, mentre usare blocchi indice grandi permette di inserire più riferimenti nel blocco-codice.

Un particolare schema di allocazione indicizzata è lo schema **COMBINATO** (usato in sistemi **UNIX/Linux**)



dove ad ogni file è associato un i-node e ogni i-node contiene diverse informazioni tra cui 15 puntatori ai blocchi dati del file (12 diretti e gli ultimi 3 indiretti). Qui il **direttorio** è una tabella che **associa ad ogni nome di file un i-node number**; il **link da un directory al rispettivo i-node** è detto **HARD-LINK** (lo stesso i-node number può essere individuato da più link). Infatti distinguiamo tra:

- **HARD-LINK (link fisico)** = **directory entry** che punta ad un **i-node** (non si possono fare hard-link verso directory [no cicli] e verso file su altri filesystem). Un file viene rimosso solo quando tutti i suoi hard link sono stati rimossi;
- **SOFT-LINK (link simbolico)** = i-node punta ad un blocco che contiene il path-name del file (dunque file che come unico blocco dati ha il nome di un altro file).



ESERCIZIO - ALLOCAZIONE

ES Filesystem da 24 blocchi di 1 MByte (N° da 0 a 23)

Blocco { LIBERO = 0
OCCUPATO = 1 }

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
0 1 0 0 0 0 1 0 0 1 1 0 1 0 0 1 0 1 0 0 1 0

BLOCCATO ALLOCARE 3 FILE { FILE 1 = 2.4 MByte
FILE 2 = 1.6 MByte
FILE 3 = 3.8 MByte }

4 TUTORI

2) ALLOCAZIONE CONCATENATA (ogni FUE gestisce tutt'uno contenuto)

BLOCCI	DIR. START	ENTRATA	END
FUE 1	0 → 2 → 3 → -1	0	3
FUE 2	4 → 5 → -1	4	5
FUE 3	7 → 8 → 12 → 14 → -1	7	14

3) ALLOCAZIONE FAT (come la concatenata ma i riferimenti nella FAT sono assorbiti dalla FAT sia nel blocco 23)

FAT	DIR. ENTRY
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23	0
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23	4
7 8 12 14 15 16 17 18 19 20 21 22 23	7

1) ALLOCAZIONE CONTIGUA (ogni FUE occupa un insieme continuo di blocchi)

BLOCCI	DIR. START	ENTRATA LENGTH	(UNUSATO)
FILE 1	2,3,4	2	3
FILE 2	7,8	7	2
FILE 3	NON ALLOCABILE	1	/

4) ALLOCAZIONE INDIRETTA (per ogni FUE c'è un index block con l'elenco)

DIR. ENTRY	BLOCCI INCE
FILE 1	0 2,3,4,-1,...
FILE 2	5 7,8,-1,...
FILE 3	12 14,15,17,18,-1...

I filesystem più usati sono:

- **FAT** → abbiamo la **FAT32** (cluster da 32 bit, aumenta il supporto per file e dischi grossi) e la **exFAT** (aumenta ulteriormente il supporto);
- **NFTS** → aumenta dimensioni supportate dalla FAT. Come Ext supporta il journaling e crittografia del disco;
- **Ext** → simile a NFTS e ad altre prestazioni.

Attributo \ FileSystem	FAT32	exFAT	NTFS	Ext4
Dimensione Disco Massima	2 Tb	64 Zb	2 Tb (estendibile a 256tb aumentando il cluster)	1 Eb
Dimensione File Massima	4 Gb	16 Zb	Quanto il disco	16 Tb
Principale Utilizzo	Chiavetta USB	Chiavetta USB	Disco Interno Windows	Disco Interno Linux

5) PROCESSI

Ricordiamo che il **PROCESSO** è l'entità attiva, ovvero l'esecuzione di un programma (entità passiva); a seconda della loro esecuzione, abbiamo:

- **PROCESSI SEQUENZIALI** = azioni eseguite una dopo l'altra (ogni istruzione inizia terminata la precedente);
- **PROCESSI CONCORRENTI** = più istruzioni eseguite contemporaneamente (non deterministico). La concorrenza può essere fittizia (sistemi mono-processore) e reale (parallelismo, sistemi multi-core).

I processi possono essere creati, gestiti, sincronizzati, terminati etc... Ogni processo possiede un **PID** (**Process ID**, ovvero un identificatore univoco intero non negativo); data la sua univocità, se processi concorrenti generano file simili, si può usare il PID per creare nomi di file univoci [esempio `sprintf(filename, "file-%d", getpid());`]. Alcuni di questi PID sono riservati:

- **0** riservato per lo **schedulatore dei processi** (ovvero lo swapper, che viene eseguito a livello kernel);
- **1** riservato per il **processo di init** (invocato alla fine del bootstrap, non termina mai e per questo diventa il padre di ogni processo rimasto “orfano”).

Alcune syscall ci ritornano l'identificatore del:

- `pid_t getpid();` → **processo chiamante**;
- `pid_t getppid();` → **padre** del processo chiamante;
- `uid_t getuid();` → **utente** del processo chiamante;
- `gid_t getgid();` → **gruppo** del processo chiamante.

Per quanto riguarda la **CREAZIONE** dei processi abbiamo la syscall **CreateProcess** (in Windows) e la syscall **fork** (in Unix/Linux). Noi vedremo meglio la fork, la quale genera un processo figlio clonato dal padre (quindi uguale eccetto per il PID, ritornato dalla fork). Dunque la fork viene richiamata 1 volta ma ritorna 2 volte (1 nel padre e 1 nel figlio). La fork [definita nella libreria <unistd.h> con `pid_t fork(void);`] torna quindi il **PID del figlio nel padre e PID 0 nel figlio** se termina correttamente, mentre -1 se non si può allocare un nuovo processo.

Qui sotto vediamo un **ESEMPIO** di programma concorrente in grado di generare un processo figlio, far terminare il processo padre prima del figlio e il processo figlio prima del padre (la syscall **sleep** (`unsigned int n`) mette il processo in wait per almeno `n` secondi) [con a fianco i risultati]:

```
#include <unistd.h>
...
printf ("Main : %s", argv[0]);
printf ("PID=%d; My parent PID=%d\n",
       getpid(), getppid());
...
pid = fork();
if (pid == 0){
    sleep (tC);
    printf ("Child : PIDreturned=%d ", pid);
    printf ("PID=%d; My parent PID=%d\n",
           getpid(), getppid());
} else {
    sleep (tF);
    printf ("Father: PIDreturned=%d ", pid);
    printf ("PID=%d; My parent PID=%d\n",
           getpid(), getppid());
}
```

Stato della shell
(ps: print process status)

Child awaits 2 secs
Father awaits 5 secs

Osservare i PID crescenti ...

Il child rimane **zombie** per 3 secondi

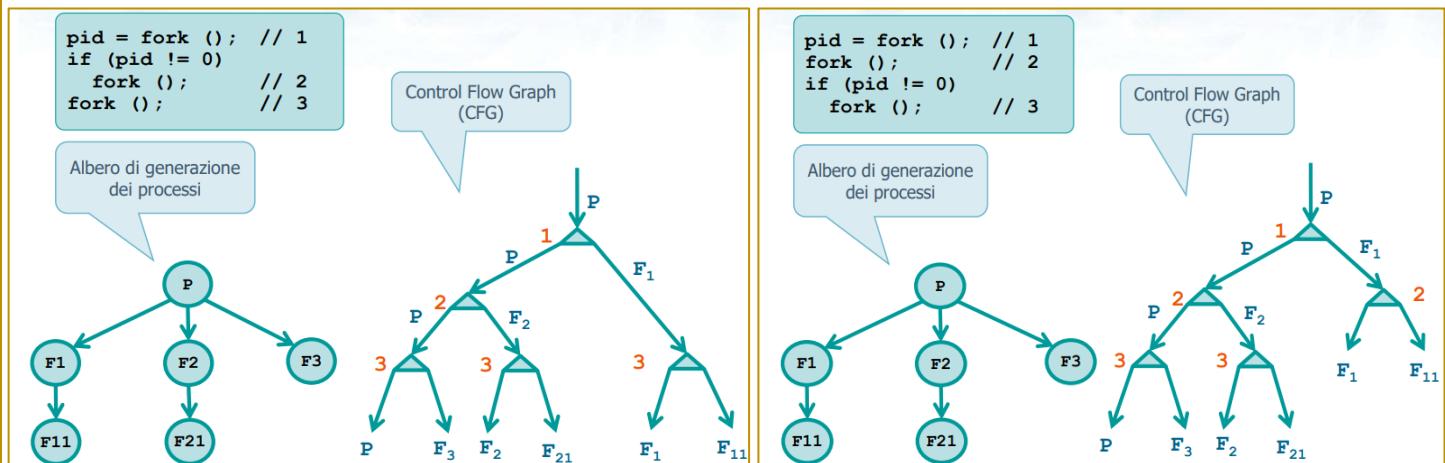
Stato della shell
(ps: print process status)

Child awakes 5 secs
Father awakes 2 secs

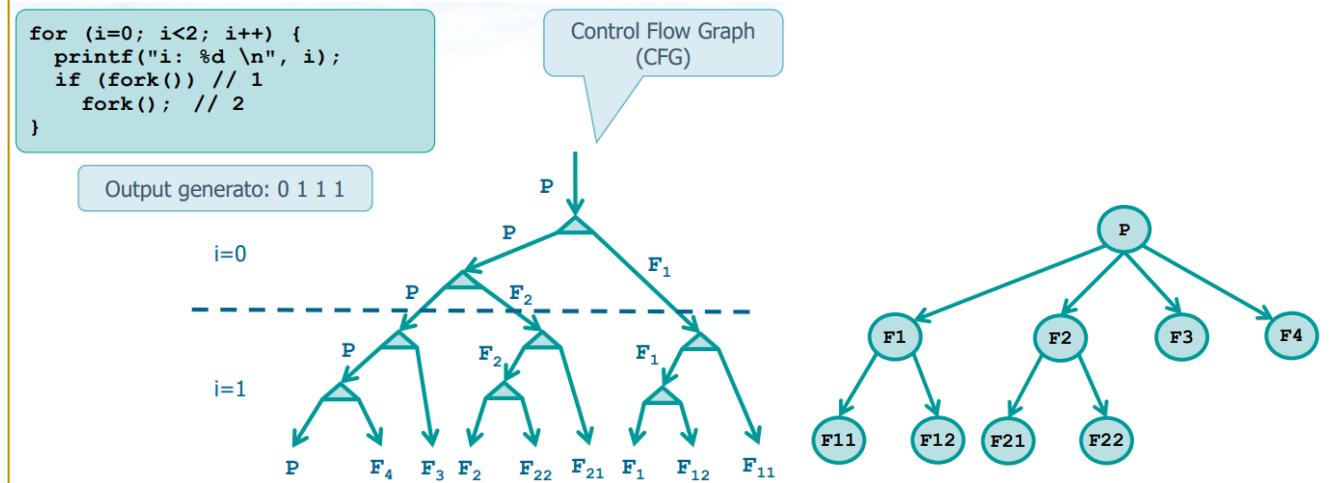
Osservare i PID crescenti ...

Il child rimane **orfano** e viene ereditato dal processo init

Altro ESERCIZIO è, dato il programma, disegnare il **CFG** (Control Flow Graph, cioè il **grafo del flusso di controllo**) e l'**albero di generazione dei processi** (ad ogni fork si fa biforcazione ma se ho **if (pid != 0)** significa che **il figlio non fa la fork ma la fa solo il padre**):



⚠ Se invece ho **if (fork())** fork significa anche qui che il figlio non fa la fork ma la fa solo il padre:



Altro ESERCIZIO è scrivere un programma concorrente che, dato un intero n, sia in grado di generare n processi figlio (né di più né di meno) [con ciascun processo figlio che visualizza il proprio PID e termina]; automaticamente ci verrebbe da scrivere il programma di sinistra, ma vengono generati **7 processi figli con n = 3** (perché vengono fatte le fork anche sui figli); perciò **bisogna fare la fork solo sul padre** (e non sui figli) [a destra]:

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

```
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);
```

Come abbiamo precedentemente detto, ogni processo è una entry nella **tabella dei processi**; **processo padre e processo figlio condividono il codice** (dopo la fork si ha una **copia dei dati identica**, ma poi possono **modificare i propri dati in maniera separata/indipendente**). In **UNIX/Linux** infatti padre e figlio:

- **CONDIVIDONO**: il codice sorgente [C], i descrittori dei file (file descriptor), lo user-id, il group-id, la root, la working directory, le risorse di sistema, i limiti di utilizzo, i segnali;
- **DIFERENZIANO**: il valore tornato dalla fork (padre conserva il proprio PID, il figlio ottiene il nuovo PID), lo spazio dati, l'heap e lo stack.

Ci sono diversi modi per **TERMINARE UN PROCESSO**:

- **5 METODI STANDARD:**

- return dalla funzione principale;
- exit;
- _exit o _Exit;
- return dal main dell'ultimo thread del processo;
- pthread_exit dall'ultimo thread del processo.

- **3 METODI ANOMALI:**

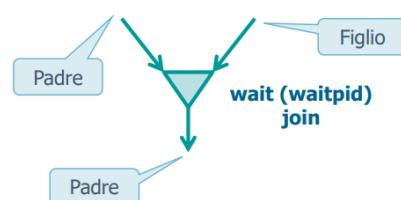
- funzione abort;
- ricevere un segnale (signal);
- cancellare l'ultimo thread del processo.

Quando un processo termina, il kernel invia un segnale (**SIGCHLD**) al padre e lui può decidere di **gestire la terminazione del figlio** o **ignorare la terminazione del figlio** (default). Se il padre decide di gestire la terminazione del figlio può farlo in maniera:

- **ASINCRONA** → con un gestore del segnale SIGCHLD (che analizzeremo in seguito con i segnali);
- **SINCRONA** → mediante le syscall **wait** o **waitpid** (l'opposto della fork, ovvero una **join**).

La syscall **wait** [definita in **<sys/wait.h>** con **pid_t wait(int *statLoc);**] ha effetti diversi a seconda dello stato dei processi figli del processo stesso:

- ritorna con un errore se il processo non ha figli (-1);
- blocca il processo se tutti i figli del processo sono ancora in esecuzione (bloccato fino a che 1 dei figli non terminerà, quando la wait ritornerà il **PID del figlio terminato** [valore di return] e lo **stato di terminazione del figlio** [parametro statLoc]);
- ritorna subito se almeno 1 dei figli è terminato (ritornando le stesse cose del caso prima).



⚠ Il parametro **statLoc** (puntatore ad un intero) indica lo **stato di terminazione del processo figlio terminato**, interpretabile confrontandolo con delle macro presenti in **<sys/wait.h>** (es. **WIFEXITED(statLoc)** è vera se la terminazione è stata corretta e **WEXITSTATUS(statLoc)** cattura gli 8 LSB del parametro passato a exit). **Esempio di processo in grado di sganciare un figlio e raccoglierne lo stato di terminazione:**

```
...  
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
```

La syscall **waitpid** si differenzia dalla **wait** in quanto può essere **non bloccante** (non blocca il padre in attesa della terminazione di almeno 1 figlio) e può attendere la **terminazione di un figlio specifico**. Definita in **<sys/wait.h>** con **pid_t wait(int *statLoc);** ha 3 parametri:

- **pid** → permette di attendere un figlio specifico se pid > 0, un figlio qualsiasi se pid = -1 (quindi come **wait**), un qualsiasi figlio il cui group ID è uguale a quello del chiamante se pid = 0 e il figlio il cui group ID è uguale a pid se abs(pid) < -1;
- **statLoc** → uguale alla **wait**;
- **options** → permette controlli aggiuntivi.

Definiamo **PROCESSO ZOMBIE** un processo terminato dove il **padre non ha ancora eseguito una wait** (il segmento dei 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**).

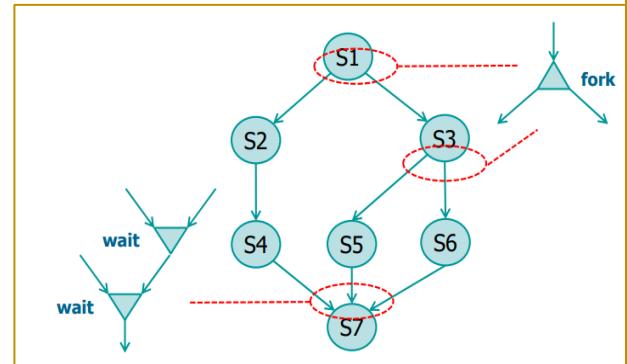
Definiamo **PROCESSO ORFANO** un processo dove il **padre termina prima di eseguire la wait** (questi vengono ereditati dal processo init [quello con PID = 1] e non diverranno più processi zombie).

ESERCIZIO – Scrivere un processo per effettuare quello che vediamo nel grafo

```

int main() {
    pid_t pid;
    printf("S1\n");
    if ((pid = fork()) == -1) err_sys("can't fork");
    if (pid == 0) {
        printf("S3\n");
        if ((pid = fork()) == -1) err_sys("can't fork");
        if (pid == 0) {
            printf("S6\n");
            exit(0);
        }
        else {
            printf("S5\n");
            while (wait((int*) 0) != pid);
        }
    }
    else {
        printf("S2\n");
        printf("S4\n");
        while (wait((int*) 0) != pid);
        printf("S7\n");
        exit(0);
    }
    return(1);
}

```



⚠ Quindi la regola è che si parte dal nodo più in alto e si prova a fare la fork: se non è possibile fine, altrimenti si fa la fork e si fa figlio (if(pid == 0)) a destra e padre (else) a sinistra; con il figlio facciamo exit(0) e con il padre facciamo la while(wait) per far confluire le ramificazioni (oppure nulla se sotto di lui non c'è una confluenza).

In UNIX/Linux ci sono dei **COMANDI DI SHELL** per eseguire processi in modo sequenziale (**FOREGROUND**) [comando1 → output1 → comando2 → output2]; questi comandi possono essere seguiti da un “&” che permette di eseguire quel processo in **BACKGROUND** (processo eseguito in modo indipendente dalla shell [lascia il terminale libero per altri lavori e permette di eseguire processi in parallelo]) [comando1 & → comando2 & → output1 → output2].

Ci sono 2 comandi principali per **visualizzare lo stato dei processi**:

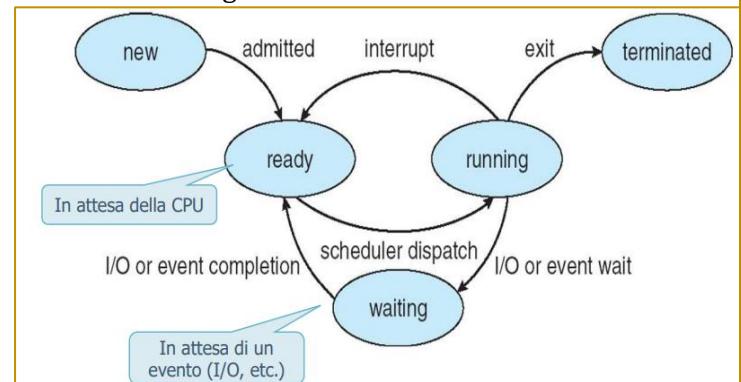
- **ps [opzioni]** → process status of active process, ovvero elenca i **processi attivi** (e i loro dettagli) [senza opzioni stampa lo stato dei processi con stesso user ID dell'utente da cui si effettua il comando];
- **top** → **visualizza informazioni sui processi in esecuzione**, aggiornate in run-time.

Opzioni			
Formato		Significato	Effetto
Compatto	Esteso		
-a			Elenca i processi di tutti gli utenti del sistema
-u			Visualizza informazioni più dettagliate (resident size, virtual size, etc.)
-u user			Visualizza i processi dell'utente <user>
-x			Aggiunge all'elenca i processi che non hanno un terminale di controllo (e.g. daemon)

Per inviare un segnale ad un processo dalla linea di comando di shell è possibile usare il comando **kill [-sig] pid** con **sig** = codice del segnale (default SIGTERM = **terminazione standard del processo**), **pid** = PID del processo target. Il comando **killall [-sig] name** termina tutti i processi di dato nome (utile per terminare tutti i processi generati dallo stesso programma senza specificare il pid).

L'**OS** tiene traccia (nella **tavella dei processi**) di ogni processo associandogli un insieme di dati:

- **stato del processo** (**new, ready, running, waiting, terminated**) → evoluzione di processo = **DIAGRAMMA DEGLI STATI**;
- **program counter** (indirizzo della successiva istruzione da eseguire);
- **registri della CPU**;
- informazioni utili per lo **scheduling della CPU** (priorità, puntatori alle code di scheduling, etc...);
- informazioni utili per la **gestione della memoria**;
- **tavella dei segnali** (signal handlers);



- o informazioni amministrative (tempo di utilizzo CPU, limiti etc...);
- o informazioni sullo **stato delle operazioni di I/O**.

⚠ 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 [**CONTEXT SWITCHING**] (il tempo dedicato a questo lavoro è overhead [non utile, quindi va ridotto al minimo]). Queste operazioni vengono controllate dalla **SCHEDULER della CPU**, che determina quando il processo corrente deve terminare la sua esecuzione e seleziona tra i processi disponibili il successivo da eseguire (ma ne parleremo meglio più avanti)!

→ **Debug Multi-Process e Thread:** la maggior parte dei debugger non da supporto per i **programmi concorrenti** (ovvero con le fork), in quanto continuano il debug del processo originale (padre) e ignorano l'esecuzione dei figli. Usando **GDB**, si può raggiungere il problema: possiamo inserire una **sleep** all'inizio dell'esecuzione dei processi padre e figlio, eseguire il programma in **background** da linea di comando e usare il comando **ps** per conoscere i PID dei processi di cui effettuare il debug; a livello di GDB, dobbiamo eseguire **1 istanza per ogni processo** di cui vogliamo fare il debug e **fornire i seguenti comandi** (in ciascuna istanza) **prima** del termine della sleep:

```
attach <pid>
file <nome dell'eseguibile>
n
```

Il debug di **programmi multi-thread** è teoricamente più semplice (inserendo un break-point nei vari thread oppure direttamente alcuni IDE permettono di seguire i thread separatamente).

5.1) CONTROLLO AVANZATO

Come già detto, la **fork** permette la duplicazione di un processo, ma esistono **2 applicazioni** di questo meccanismo:

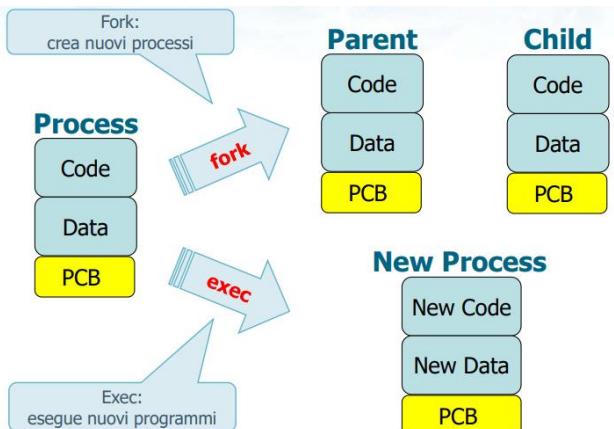
- **padre e figlio eseguono sezioni diverse di codice** (es. nei server di rete, dove all'arrivo di una richiesta il server si duplica [figlio gestisce la richiesta, mentre padre continua l'attesa]);
- **padre e figlio eseguono codici differenti** (es. nelle shell). Questo richiede l'uso dei comandi **exec**.

La syscall **exec** sostituisce il processo con un nuovo **programma** (ovvero sostituisce l'immagine del processo corrente con quelli di un processo nuovo [**il PID non cambia** perché **fork** = duplica processo esistente, **exec** = esegue nuovo programma]).

Esistono **6 versioni** della **exec**, ovvero:

```
int execl (char *path, char *arg0, ..., (char *)0);
int execlp (char *name, char *arg0, ..., (char *)0);
int execle (char *path, char *arg0, ..., (char *)0,
            char *envp[]);

int execv (char *path, char *argv[]);
int execvp (char *name, char *argv[]);
int execve (char *path, char *argv[], char *envp[]);
```



Dove le lettere dopo “**exec**” specificano:

- **l (list)** → la funzione riceve una lista di argomenti, ovvero in **char* arg0** troviamo il **nome** dell'eseguibile, mentre i successivi sono i **parametri** dell'eseguibile (simile alla stringa **argv[0]** in C e ai parametri come **argv[i]**);
- **v (vector)** → la funzione riceve un vettore di argomenti, simile alla matrice dinamica ****argv** del C;
- **p (path)** → la funzione riceve solo il nome del file (non il percorso [path]) e lo rintraccia con la variabile PATH, ovvero il campo **char* path** conterrà il **percorso nelle exec senza "p"** e può contenere **solo il nome nelle "exec...p"** [il **percorso di default** deve però essere contenuto nella variabile di sistema **PATH**];
- **e (environment)** → la funzione riceve un vettore di variabili di ambiente (environment), invece di usare l'environment corrente.

⚠ La funzione **exec** non ritorna in caso di successo (nessun valore di return) e torna **-1** in caso di errore!

Un **esempio** di programma (`./pgrm`) che richiama se stesso se riceve come parametro 1 o 2 (a destra l'**output** [con `n = 1`] più i comandi di shell (in blu) `ps -aux | grep 2471`, ovvero una **pipe** (`|`) che prende il comando di sinistra `ps -aux` [stampa le righe del programma in questo caso] e lo fa gestire dal comando di destra `grep 2471` [cerca nelle righe quella con PID 2471]):

```

n = atoi (argv[1]);
switch (n) {
    case 1:
        printf("#1:PID=%d;PPID=%d\n", getpid(), getppid());
        sleep (n*10);
        execl ("./pgrm", "./Pgrm", "2", (char *) 0);
        break;
    case 2:
        Anche argv[0]
        printf("#2:PID=%d;PPID=%d\n", getpid(), getppid());
        sleep (n*10);
        execl ("./pgrm", "ilMioPgmr", "3", (char *) 0);
        break;
    default:
        printf("#3:PID=%d;PPID=%d\n", getpid(), getppid());
        sleep (n*10);
        break;
}
return (1);

```

IL PID non cambia

```

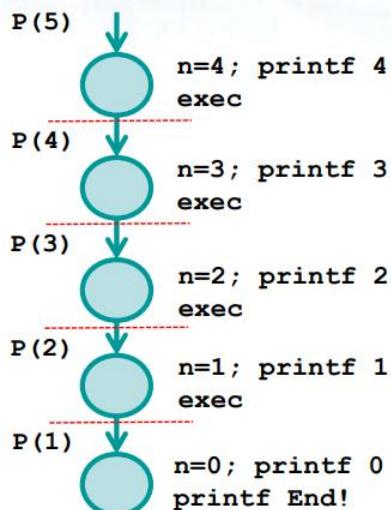
> ./pgrm 1 &
[2] 2471
#1: PID=2471; PPID=2045
> ps -aux | grep 2471
quer 2471 0.0 0.0 4192 352 pts/2 S 19:29 0:00 ./pgrm 1
#2: PID=2471; PPID=2045
> ps -aux | grep 2471
quer 2471 0.0 0.0 4192 356 pts/2 S 19:29 0:00 ./Pgrm 2
#3: PID=2471; PPID=2045
> ps -aux | grep 2471
quer 2471 0.0 0.0 4192 356 pts/2 S 19:29 0:00 ilMioPgmr 3
[2]+  Exit 1 ./pgrm 1

```

Comandi di shell (in blu)

Il nome cambia

Esercizio – Albero di generazione dei processi (con parametro passato su riga di comando, ovvero `argv[1] = 5`)



```

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

```

Output

4
3
2
1
0
End!

⚠ I 2 `argv[0]` nella `execl` sono (come già detto prima) il 1° = nome/percorso del programma da eseguire, mentre il 2° = nome "fittizio" che vogliamo dare al programma (ma solo in locale, infatti nel caso sopra vediamo che il nome del "case 1" non rimane poi nel "case 2" al posto del nome del programma).

Esercizio – Albero di generazione dei processi

```

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

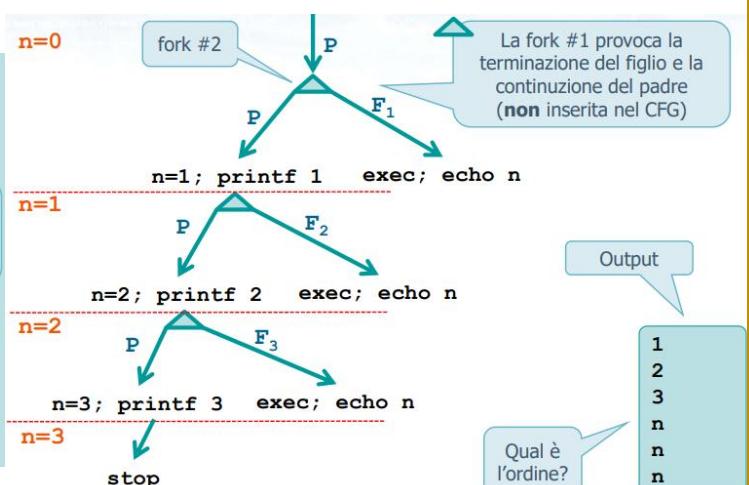
int main(){
    int n;
    n=0;
    while (n<3 && fork()){
        if (!fork())
            execl ("echo", "n++", "n", NULL);
        n++;
        printf ("%d\n", n);
    }
    return (1);
}

```

fork #1
Se 0 siamo nel figlio; il figlio termina subito

fork #2
Se 0 siamo nel figlio; il figlio fa la exec

"printf" di shell

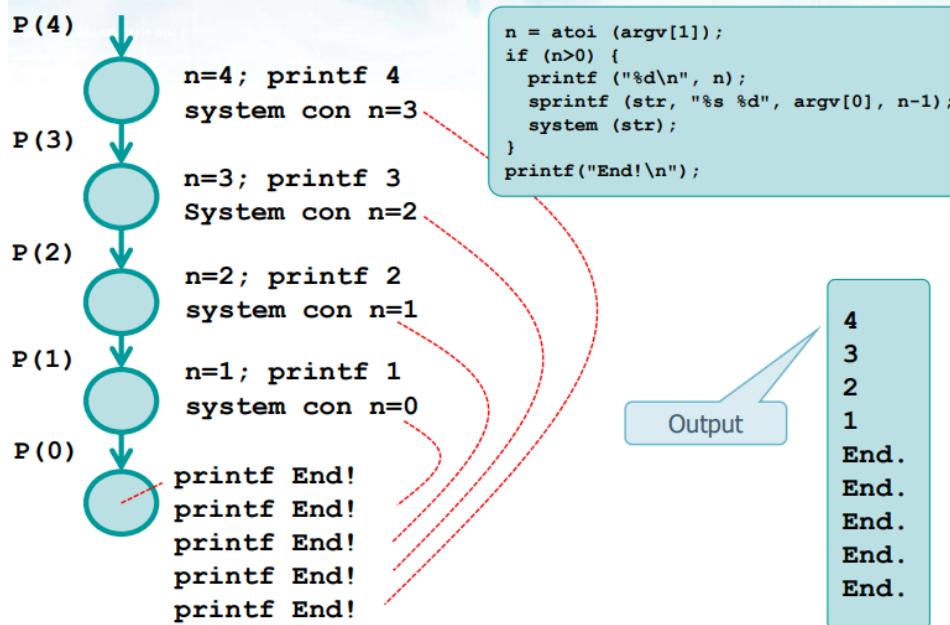


⚠ Un programma contenente delle wait se viene eseguito in foreground (**comando**), la shell deve aspettare la terminazione della wait, mentre in background (con **comando &**) non deve aspettare la wait.

La syscall **system** [ovvero `int system (const char* string);`] passa il comando di nome "string" all'ambiente affinché questo lo esegua; il controllo viene restituito al processo chiamante una volta che l'esecuzione del comando è terminata.

⚠ Ritorna il valore di terminazione della shell che esegue il comando, ma se fallisce torna -1 se fallisce la fork o la waitpid usata per realizzarla oppure torna 127 se fallisce la exec usata per realizzarla!

Esercizio – Albero di generazione dei processi (argv[1] iniziale = 4)

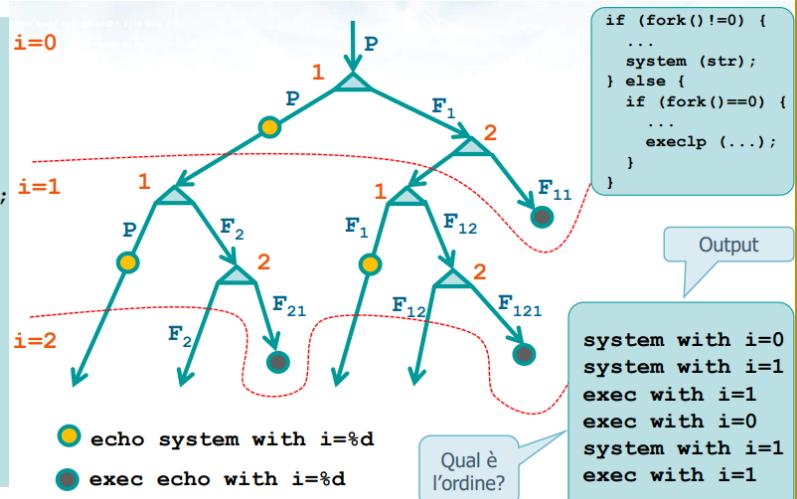


⚠ Quindi la **system** fa ripartire il programma con `argv[1] = n - 1`, ma alla fine tutti i processi interrotti dalla **system** vengono ripresi (in questo caso infatti si stampano tanti "End!" quanti sono i processi interrotti dalla **system** + quello finale).

Esercizio – Albero di generazione dei processi (+ complesso)

```

#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 with i=%d", i);
                execlp ("echo", "myPgrm", str, NULL);
            }
        }
    }
    return (0);
}
  
```



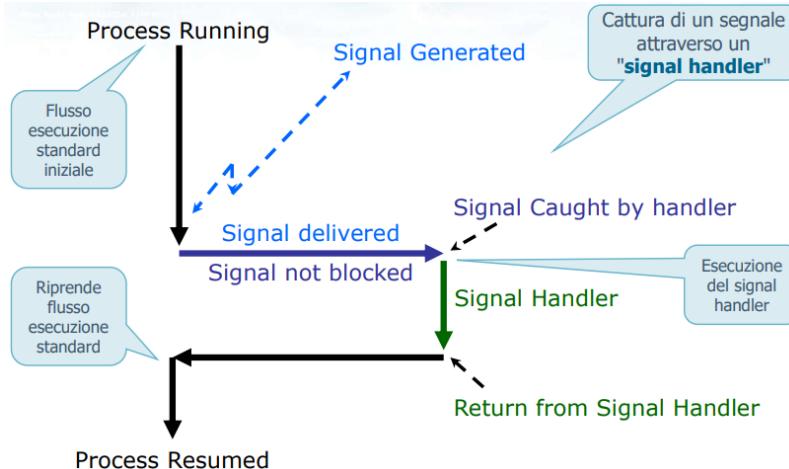
6) SEGNALI

Definiamo **SEGNALE** un interrupt software (INTERRUPT = interruzione dell'esecuzione corrente dovuto al verificarsi di un evento [es. un dispositivo hardware che invia una richiesta di servizio alla CPU (interrupt hardware) o processo che richiede un'operazione tramite una syscall (interrupt software)]), ovvero una notifica inviata dal kernel o da un processo ad un altro processo per dirgli che si è verificato un particolare evento. Permettono di gestire eventi asincroni e possiamo usarli per la comunicazione tra processi. Hanno nomi che iniziano con **SIG** (definiti nel file `signal.h`) [es. `SIGUSR1` o `SIGUSR2`, disponibili nelle applicazioni utente, usati per far terminare qualcosa].

La gestione di un segnale ha 3 fasi:

- GENERAZIONE → quando il kernel o il processo sorgente effettua l'evento;
- CONSEGNA → invio del segnale alla destinazione;

- **GESTIONE** → il processo destinazione deve dire al kernel cosa vuole fare nel caso riceva uno specifico segnale.



La gestione di un segnale si può fare con le seguenti **syscall**:

- **void (*signal (int sig, void (*func) (int)))(int);** → instanzia un gestore di segnali, ovvero specifica il segnale da gestire (**sig**) la cui ricezione attiverà la funzione per gestirlo (**func**) [cioè il gestore di segnali (**signal handler**)]. In caso di **successo**, ritorna il **puntatore alla funzione gestore del segnale (*func)**, mentre **SIG_ERR** in caso di **errore**. Inoltre, permette di impostare 3 comportamenti diversi:
 - lasciare che si verifichi il **comportamento di default** [**func = SIG_DFL**];
 - **ignorare il segnale** [**func = SIG_IGN**];
 - **catturare il segnale** (**func specificata da noi**).

⚠ Se non vogliamo fare una terminazione sincrona con la wait oppure asincrona con una funzione gestore apposita e non vogliamo far diventare i figli del processo dei processi zombie, allora dobbiamo impostare una **signal** (**SIGCHLD**, **SIG_IGN**) in modo che la **terminazione dei figli venga ignorata!**

- **int kill (pid_t pid, int sig);** → invia il segnale **sig** ad un processo o ad un gruppo di processi **pid**. Per inviare un segnale ad un processo occorre averne diritto. Torna **0 se successo, -1 se errore**. Parametri:

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

- **int raise (int sig);** → permette ad un processo di inviare un segnale **sig** a sé stesso (di default, se scrivo **raise(sig)** equivale a **kill(getpid(), sig);**)
- **int pause (void);** → sospende il processo chiamante fino all'arrivo di un segnale (ritorna -1 solo quando viene eseguito un gestore di segnali che termina la sua esecuzione);
- **unsigned int alarm (unsigned int seconds);** → attiva un timer di n° secondi = **seconds**; finito il countdown, viene generato il segnale SIGALRM (e di default, se SIGALRM viene ignorato, termina il processo [se **seconds = 0**, si disattiva il precedente allarme]). Il return è il n° di secondi rimasti prima dell'invio del segnale se ci sono state chiamate precedenti, oppure 0 in caso non ci siano state chiamate precedenti.

⚠ C'è un solo timer per ciascun processo, dunque le syscall **sleep** e **alarm** usano lo stesso timer!

Ci sono diversi **LIMITI** per i segnali:

- **MEMORIA** dei segnali “pending” (inviati ma non consegnati) è **LIMITATA** → si ha al max 1 segnale “pending” per ciascun tipo di segnale [quelli successivi dello stesso tipo si perdono]. Inoltre i segnali possono essere bloccati (ovvero si può evitare di riceverli) [con il comando **kill -9 <pid del processo>** si uccide un processo perché **-9 = SIGKILL**].
- Richiedono **FUNZIONI RIENTRANTI** → quando si esegue un segnale, una volta finita la gestione, il kernel sa dove riprendere il flusso delle istruzioni (bloccate in precedenza per eseguire il segnale). **Cosa succede però**

se il signal handler fa un'operazione non compatibile con il flusso di istruzioni bloccate? [es. interrompiamo una malloc o interrompiamo l'esecuzione di una funzione che usa una variabile statica]. Lo UNIX definisce una serie di **funzioni** (dette **rientranti**) che possono essere interrotte senza problemi (es. read, write, sleep, wait ...), ma la maggior parte delle funzioni di I/O del C non sono rientranti (es. printf, scanf...)!!!

- Producono **RACE CONDITIONS** → il comportamento di più processi che lavorano sugli stessi dati dipende dall'ordine di esecuzione. Questa cosa va gestita, altrimenti si generano dei problemi. Esempio:

- ❖ Si supponga due processi P_1 e P_2 vogliono 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...);
}

```

⚠ Nonostante i loro difetti, i segnali possono fornire un meccanismo di sincronizzazione!

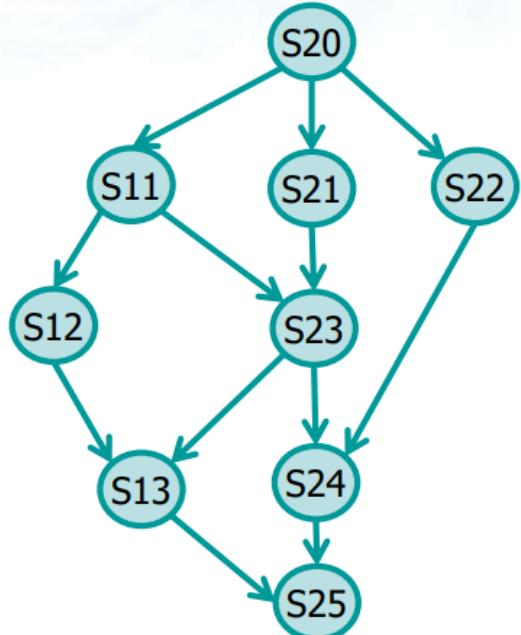
Esercizio - trascurando le race conditions realizzare il programma del seguente grafo di precedenza (CFG):

```
// Signal Handler
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;
}

void P1 (pid_t cpid) {
    printf("S11\n");
    kill(cpid, SIGUSR1);
    printf("S12\n");
    pause();
    printf("S13\n");
    return;
}

void P2 () {
    if (fork() > 0){
        printf("S21\n");
        pause();
        printf("S23\n");
        kill(getpid(), SIGUSR2);
        wait((int*) 0);
    } else {
        printf("S22\n");
        exit(0);
    }
    printf("S24\n");
    return;
}

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);
    }
    printf("S20\n");
    pid = fork();
    if(pid > (pid_t) 0){
        P1(pid);
        wait((int*) 0);
    } else {
        P2();
        exit(0);
    }
    printf("S25\n");
    return (0);
}
```



7) COMUNICAZIONE TRA PROCESSI

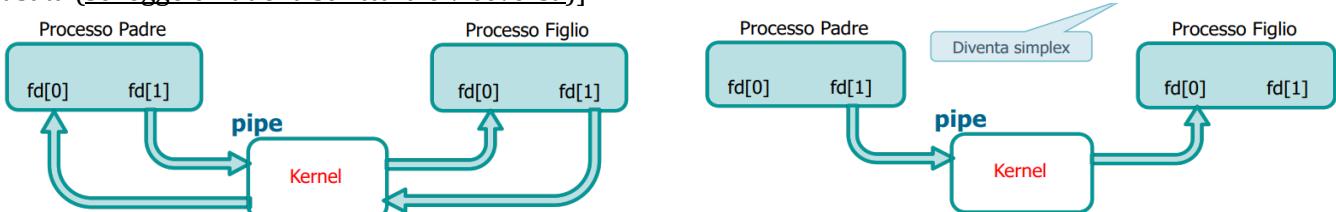
I processi concorrenti possono essere:

- **INDIPENDENTI** → non influenzati e non influenzano gli altri processi;
- **COOPERANTE** → influenzati e influenzano gli altri processi. La cooperazione può avvenire solo con scambio o condivisione di dati (IPC = InterProcess Communication); i modelli di comunicazione sono basati su:
 - **MEMORIA CONDIVISA** (normalmente l'OS impedisce ad un processo di accedere alla memoria di un altro processo, ma qui no) [usata per tanti dati];
 - **SCAMBIO DI MESSAGGI** = occorre instaurare un **canale di comunicazione** e richiede l'utilizzo delle system calls [usata per pochi dati].

Il **CANALE DI COMUNICAZIONE** viene caratterizzato da naming (denominazione [diretta o indiretta]), sincronizzazione [sincrona o asincrona] e capacità (ovvero la dimensione del buffer per i messaggi in attesa).

Noi vedremo solo le **half-duplex pipes** e i **semaphores (semafori)**!

- **HALF-DUPLEX PIPES** → canale di comunicazione **diretto, asincrono e a capacità limitata**. Una PIPE consiste in un **flusso di dati tra 2 processi**; viene gestita come un file ed è rappresentata tramite **2 descrittori** (1 per estremità). Sono **HALF-DUPLEX** in quanto i dati possono fluire in entrambe le direzioni, ma non insieme (quello accade nelle full-duplex) e possono essere per la **comunicazione tra processi con un parente comune**. La syscall **int pipe (int fileDescr[2]);** (definita in <unistd.h>) crea una pipe e ritorna i descrittori delle **2 estremità della pipe nel vettore fileDescr[2]** (l'output effettuato su **fileDescr[1]** corrisponde all'input ricevuto su **fileDescr[0]**). Il processo padre crea una pipe, effettua una fork per generare il figlio (il quale eredita i descrittori del file); quindi 1 dei 2 processi (default il padre) scrive nella pipe, mentre l'altro legge dalla pipe (default il figlio) [**il descrittore non usato può essere chiuso**, ovvero viene chiusa l'estremità non usata (**se leggo chiudo la scrittura e viceversa**)].



Le pipe vengono **manipolate come i file** in UNIX, ovvero si usano le funzioni **read** e **write**:

- **read** → si blocca se la pipe è vuota (**bloccante**), ritorna solo i caratteri disponibili (se la pipe contiene meno caratteri di quanto richiesto) e ritorna 0 se la pipe è stata chiusa all'altra estremità;
- **write** → si blocca se la pipe è piena (**bloccante**) e ritorna SIGPIPE se l'altra estremità è stata chiusa [per capire la dimensione di una pipe, si può usare infatti la write fino a che questa non si blocca, proprio perché è bloccante].

Esercizio – CODICE PER CAPIRE LA DIMENSIONE MASSIMA DELLA PIPE (con la shell affianco)

```
#define SIZE 524288
int fileDescr[2];
int main() {
    int i, n, nR, nW;
    char c = '1';
    setbuf(stdout, 0);

    pipe(fileDescr); // creo la pipe
    n = 0;
    if (fork()) {
        fprintf(stdout, "\nFather PID=%d\n", getpid());
        sleep(1);
        for (i = 0; i < SIZE; i++) {
            nW = write(fileDescr[1], &c, 1);
            n += nW;
            fprintf(stdout, "W %d\r", n);
        }
    } else {
        fprintf(stdout, "\nChild PID=%d\n", getpid());
        sleep(10);
    }
}
```

```
> ./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

```

        for (i = 0; i < SIZE; i++) {
            nR = read(fileDescr[0], &c, 1);
            n += nR;
            fprintf(stdout, "R %d\r", n);
        }
    }
}

```

⚠ Cosa succede se non si rispetta la gestione half-duplex di una pipe? Si possono invertire le operazioni di `read` e `write` se lo si fa correttamente; per quanto riguarda la caratteristica full-duplex, questo non accade con le half-duplex pipe (al massimo possono alternarsi molto velocemente tra chi scrive e chi legge, ma mai contemporaneamente)!

→ **PIPE E REDIREZIONE IN UNIX/LINUX:** la comunicazione tra processi può essere fatta con le pipe anche quando i processi vengono eseguiti con i comandi di shell; le pipe a livello di comandi shell sono rappresentate con il carattere “|”. La pipe crea un collegamento tra lo `stdout` del comando precedente e lo `stdin` del comando successivo (la pipe risiede completamente in memoria).



Esempio:

```

ls -la | more
ps | grep "main"
cat file1.txt file2.txt file3.txt | sort
ls -laR *.c | wc

```

⚠ Attenzione alla differenza tra il comando `wc *.c` (conta le parole in tutti i file C) e `ls -laR *.c | wc` (conta le parole dell'elenco dei file C, ovvero nei nomi/percorsi dei file)!

Altra cosa da vedere in UNIX/Linux è la **REDIREZIONE dell'I/O**, ovvero permettere ad un comando di leggere e scrivere i dati su terminali diversi da quelli predefiniti (default: `stdin = 0, stdout = 1, stderr = 2`). Lo si fa con:

- **comando < file** (redirezione dello `stdin` da un file), oppure, se voglio la redirezione del `stdin` da tastiera uso `comando << marker ...testo... marker;`
- **comando > file** (redirezione dello `stdout` su un file), dove se il file esiste viene cancellato e poi ricreato per essere usato nello `stdout`, oppure, se non voglio cancellare il file, posso usare `comando >> file` (l'output viene accodato sul file);
- **comando 2> file** (redirezione dello `stderr` su un file) e analogamente a sopra `comando 2>> file`.

⚠ Si può fare anche redirezione contemporanea dello `stdout` e dello `stderr` sullo stesso file (con `comando &> file` oppure `comando &>> file`) o su file diversi (`comando > fileOut 2> fileErr`)!

8) THREADS

Un **THREAD** è una sezione di un processo che viene **schedulata ed eseguita indipendentemente dal processo** (o thread) che l'ha generata. Un thread **condivide il proprio spazio di indirizzamento** con gli altri thread. Quindi il thread è un **processo “leggero”** (processo = unità che raggruppa le risorse, thread = unità di schedulazione della CPU). Rispetto agli altri thread, un thread ha:

- **DATI CONDIVISI** → sezione di codice, sezione di dati (variabili, descrittori dei file...), risorse dell'OS (segnali);
- **DATI PRIVATI** → PC (Program counter), registri hardware, stack (variabili locali e storia dell'esecuzione).

Quindi, per quanto riguarda la programmazione multi-thread:

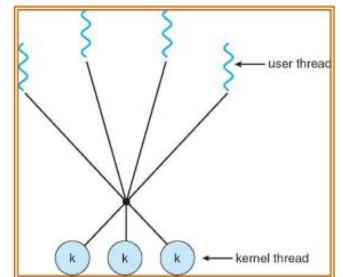
- ❖ **Pro:** tempi di risposta ridotti, risorse condivise, costi ridotti, maggiore scalabilità
- ❖ **Contro:** non esiste protezione tra thread (se i thread non sono sincronizzati, l'accesso ai dati condivisi non è “thread safe”) e non esiste gerarchia padre-figlio tra i thread (tutti paritetici)

Esistono **3 modelli di programmazione multi-thread**:

- **Kernel-Level** → thread gestiti dal kernel, ovvero l'OS conosce e manipola i thread (come manipola i processi) [tutte le operazioni sui thread vengono effettuate con le syscall]. L'OS inoltre mantiene per ciascun thread informazioni simili a quelle che mantiene per ogni processo (tabella dei thread).
 - ❖ **Pro:** l'OS conosce i thread, efficace nelle applicazioni che si bloccano spesso (es. read bloccante) e permette parallelismo

- ❖ **Contro:** a causa del passaggio a kernel mode (syscall), la gestione è lenta e inefficiente; è limitato il max numero di thread e l'OS deve mantenere troppe informazioni (tabella dei Thread e TCB, Thread Control Block)
- **User-Level** → gestiti run-time a livello utente (tramite una libreria), cioè kernel non conosce i thread (gestisce solo i processi); ogni processo ha però bisogno di una tabella personale dei thread in esecuzione (ma comunque meno informazioni richieste del Kernel-Level)
 - ❖ **Pro:** i thread si possono implementare in tutti i kernel (anche nei sistemi che non li supportano nativamente), non si richiedono modifiche dell'OS, gestione efficiente e il programmatore può generare tutti i thread che vuole
 - ❖ **Contro:** l'OS non conosce i thread e può portare ad inefficienza; inoltre bisogna comunicare informazioni tra kernel e utente, lo scheduler deve mappare i thread utente sull'unico thread kernel (ovvero se thread kernel si blocca, si bloccano anche tutti i thread utente)
- **Mista (soluzione ibrida)** → si tenta di combinare i vantaggi di entrambe: l'utente decide quanti thread utente eseguire e su quanti thread kernel mapparli, mentre il kernel conosce e gestisce solo i thread kernel (ogni thread kernel può essere usato a turno da diversi thread utente).

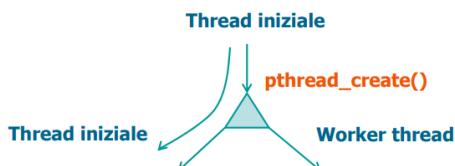
Come coesistono thread e processi? La coesistenza porta a diversi **problemi**:



- **Gestione dei segnali:** il comportamento di un processo alla ricezione di un segnale specifico è condiviso da tutti i thread e ogni segnale è consegnato ad un thread singolo all'interno del processo;
- **Utilizzo della fork:** in un processo multi-thread, una fork duplica solo il thread che richiama la fork (i dati privati dei thread non duplicati dalla fork possono non essere gestibili dall'unico thread duplicato);
- **Utilizzo della exec:** una exec effettuata da un thread sostituisce con il nuovo programma l'intero processo (non solo il thread che fa l'exec).

Vediamo ora la libreria **PTHREADS** ([Posix Threads](#), `#include <pthread.h>`), una libreria UNIX per la gestione dei thread; il thread viene gestito come una funzione che viene eseguita in maniera indipendente dal resto del programma. Questa libreria definisce **60 funzioni di gestione dei thread** (tutte con “`pthread_nomeFunzione`”). Per compilare un programma includendo questa libreria, bisogna scrivere `gcc -Wall -g -o <exeName> -pthread <file.c>`. Un thread è **identificato** in maniera univoca da un **identificatore** di tipo `pthread_t` (simile al tipo `pid_t`). Vediamo ora le funzioni:

- `int pthread_equal (pthread_t tid1, pthread_t tid2);` → **confronta 2 identificatori di thread** (torna valore ≠ 0 se thread uguali, altrimenti 0);
- `pthread_t pthread_self (void);` → **ritorna l'identificatore del thread chiamante** (usata con la equal per auto-identificarsi, utile per accedere ai propri dati personali);
- `int pthread_create (pthread_t *tid, pthread_attr_t *attr, void *(*startRoutine)(void*), void *arg);` → **crea un nuovo thread** (il numero massimo di create dipende dall'implementazione); ha come parametri il puntatore all'identificatore del thread generato, i suoi attributi [default = NULL], la routine C eseguita dal thread e l'argomento passato alla routine di inizio. Torna 0 = successo, errore = fallimento;

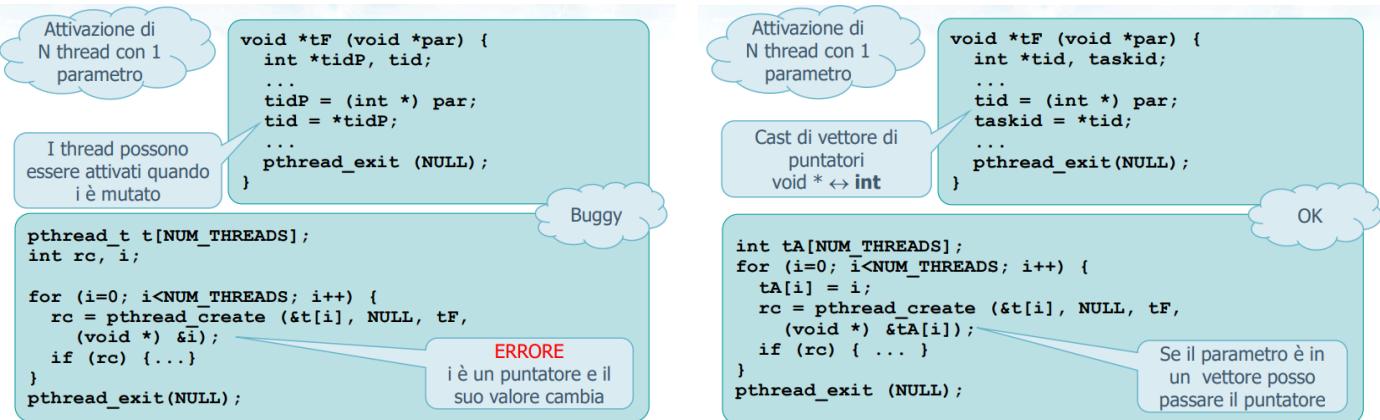


- `void pthread_exit (void *valuePtr);` → **permette ad un thread di terminare** restituendo il suo stato di terminazione. Il valore `valuePtr` è mantenuto dall'OS fino a quando un thread fa una `pthread_join` (il valore sarà disponibile a quel thread);

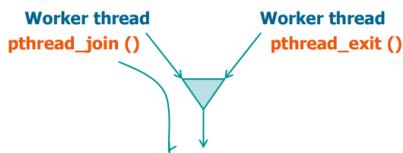
⚠ Un **intero processo** (con tutti i suoi thread) **termina** se un suo thread fa `exit`, il main effettua una `return` o un suo thread riceve un segnale la cui azione è terminare; invece un **singolo thread** **termina** se effettua una `return` dalla sua funzione di inizio, esegue una `pthread_exit` o riceve una `pthread_cancel` da un altro thread!

⚠ Nei programmi multi-thread bisogna **evitare di passare puntatori a valori** che variano nel nostro programma ai thread, perché nessuno ci assicura che, nel mentre che il thread lavora con quel valore passato con un

puntatore, il valore contenuto nel puntatore non cambi (problema di **SINCRONIZZAZIONE**)! Quindi è consigliabile passare il valore al thread e non il suo puntatore (contenitore), ma sarebbe ancora meglio se quel valore si trova in un vettore e io passo il puntatore (**vettore**):



- **void pthread_join (pthread_t tid, void **valuePtr);** → usata per fare la **wait su un altro thread**; i parametri sono il tid atteso e il valore valuePtr che riferirà il valore tornato dal thread. Torna 0 = successo, errore = fallimento;



⚠ Alla sua creazione un thread può essere dichiarato **"joinable"** (ovvero il suo stato di terminazione viene mantenuto fino a quando un altro thread esegue una pthread_join per quel thread) o **"detached"** (stato di terminazione subito dropato). Il thread che richiama la pthread_join rimane bloccato fino a che il thread richiesto non fa la pthread_exit!

- **int pthread_cancel (pthread_t tid);** → **termina il thread indicato** (come se fosse una pthread_exit con parametro PTHREAD_CANCELED). Il thread chiamante non attende la terminazione del thread che vuole cancellare (continua nel mentre). Torna 0 = successo, errore = fallimento;
- **int pthread_detach (pthread_t tid);** → **dichiara il thread come detached** (la memoria del thread che termina sarà restituita all'OS) per cui non sarà più possibile fare una join con quel thread. Torna 0 = successo, errore = fallimento.

ESERCIZIO – Realizzare con i thread il seguente grafo di precedenza

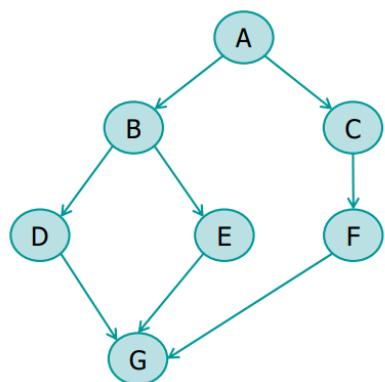
```

// funzione per wait con tempo random (per non far terminare il thread)
void waitRandomTime (int max){
    sleep ((int)(rand() % max) + 1);
}

int main (void) {
    // dichiaro i tid dei thread che dovrò passare alla funzione
    pthread_t th_cf, th_e;
    void *retval;
    // sid
    srand (getpid());
    waitRandomTime (10); printf ("A\n");
    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;
}

// funzione per i thread C e F
static void *CF () {
    waitRandomTime (10); printf ("C\n");
}

```



```

    waitRandomTime (10); printf ("F\n");
    return ((void *) 1); // Return code
}
// funzione per il thread E
static void *E () {
    waitRandomTime (10); printf ("E\n");
    return ((void *) 2); // Return code
}

```

→ **ESPRESSIONI REGOLARI e COMANDO FIND:** le **ESPRESSIONI REGOLARI** [vedremo le ERE (Extended Regular Expression)] sono un modo per **specificare un insieme di stringhe complesse** (es. **a | b*** indica l'insieme delle stringhe {a, Ø, b, bb, bbb, bbbb, ...}) [usate infatti nel matching di oggetti come i nomi di file]. Un'espressione regolare è un NFA (Automa finito non-deterministico). Nelle espressioni regolari troviamo **LETTERALI** (qualsiasi carattere usato nella ricerca del match), **METACARATTERI** (caratteri con significato speciale, es ***** = da 0 a ∞ simboli precedenti) e **SEQUENZE DI ESCAPE** (metodo per indicare che un metacarattere va usato come letterale, es. *****). Vediamo alcuni di questi metacaratteri:

- [...] → specifica un elenco o un intervallo di simboli (es. **[_a-zA-Z][_a-zA-Z0-9]*** specifica il nome di una variabile);
- (...) → raggruppa insiemi di simboli, gestisce la precedenza tra operatori (classiche tonde) e permette riferimenti ad espressioni precedenti (es. **a(b|c)** può essere abd o acd);
- | → or tra 2 espressioni regolari (es. **b|c** può essere b o c);
- \< → inizio parola;
- \> → fine parola;
- ^ → inizio riga;
- \$ → fine riga;
- * → elemento presente $[0, \infty]$ volte;
- + → elemento presente $[1, \infty]$ volte;
- ? → elemento presente $[0,1]$ volte;
- [abc] → 1 qualsiasi dei caratteri a, b o c;
- [a-z] → 1 qualsiasi dei caratteri nel range tra a e z;
- [^a-z] → 1 qualsiasi dei caratteri non nel range tra a e z;
- {n} → elemento presente esattamente n volte;
- {n,m} → elemento presente da n a m volte;
- c → un qualsiasi simbolo c (tranne quelli speciali);
- . → un qualsiasi carattere non '\n';
- \s → spazio o tabulazione;
- \d → una cifra (0-9);
- \D → non una cifra;
- \w → qualsiasi carattere tra 0-9, A-Z, a-z;
- \W → qualsiasi carattere non tra 0-9, A-Z, a-z;
- [a-zA-Z0-9] → una lettera o una cifra;
- (.)\1 → 2 caratteri identici (perché \N indica che viene preso il carattere alla posizione N);
- (.)(.)\2\1 → stringa palindroma di 5 caratteri (abcba perché \2 prende il 2° carattere e \1 prende il 1° carattere);

Esempi: ABC → la stringa "ABC"; ab? → solo a oppure ab; a{5,15} → da 5 a 15 ripetizioni di a; (ciao){3,9} → da 3 a 9 ripetizioni di "ciao"; .+ → una riga, ovvero qualsiasi sequenza non vuota; myfunc.*(.*) → una funzione il cui nome inizia per "myfunc"; \d{1,2}\/\d{1,2}\/\d{4} → data con formato gg/mm/aaa.

⚠ Il comando **tree** mostra l'albero di ciò che contiene una directory!

Il comando **find directory [opzioni] [azioni]** ricerca ed elenca file, direttori o link che soddisfano il match (ritorna il path relativo, non solo il nome). Sostanzialmente visita tutto l'albero a partire dal direttorio **directory** (recursive), trova l'elenco che soddisfa le **opzioni** e effettua per ogni file le **azioni** specificate. Le **opzioni** sono:

- -name **pattern** → **match con solo il nome del file** (il path iniziale è rimosso) [-iname pattern è identica ma è case insensitive];

- **-path pattern** → come il precedente ma specifica path+nome [-ipath identico ma case insensitive];
- **-regex expr** → specifica un'espressione regolare che deve avere un match con il path completo [anche qui iregex expr case insensitive];
- **-regextype expr** → specifica il tipo di regex usato e va messo prima di -regex expr;
- **-atime[+, -]n, -ctime[+, -]n, -mtime[+, -]n** → ultimo access, status o modification time; n=1 specifica da 0 a 24h fa; inoltre possiamo avere +n (che indica \leq) o -n (che indica \geq);
- **-size [+, -]n[bckwMG]** → indica la dimensione del file. Possiamo avere + (che indica \leq) o - (che indica \geq); il carattere dopo il segno indica l'unità di misura (b blocchi di 512 byte, c byte, k kByte, w word (2 byte), M MByte, G GByte);
- **-type tipo** → tipo di file (f = file, p = pipe, l = symbolic link, s = socket, d = directory);
- **-user nome, -group nome** → appartiene ad un certo user e/o group;
- **-readable, -writable, -executable** → leggibile, scrivibile, eseguibile;
- **-mindepth n, -maxdepth n, -quit** → sezione dell'albero in cui effettuare la ricerca (mindepth = profondità minima per la ricerca nell'albero e maxdepth = profondità massima); con -quit si smette la ricerca dopo il primo match.

Le azioni invece sono:

- **-print** → azione di default, stampa un nome per ciascuna riga;
- **-fprintf** → come prima, ma output su file;
- **-print0** → come print, ma non va a capo (stampa tutto su stessa riga);
- **-exec command** → esegue il comando, includendo nome e path (es. `find . -name "*.txt" -exec mv \{\} \{\}.back \;` → trova tutti i file.txt nella directory corrente e li sostituisce con i file.txt.back). I comandi avranno come argomento quello che trovo con la find se uso \{\};
- **-delete** → elimina ciò che trovo.

→ **FILTRI**: un FILTRO è un comando che riceve il proprio input da stdin, lo manipola (filtra, secondo determinati parametri e opzioni) e manda l'output su stdout [sostanzialmente sono comandi per manipolare testo]. Vediamo alcuni filtri:

- **cut [opzioni] file** → rimuove sezioni specifiche del file indicato (opzioni qui a destra);

Opzioni			
Formato		Significato	Effetto
Compatto	Esteso		
-c LIST	--characters=LIST		Seleziona solo i caratteri di posizione indicata
-f LIST	--fields=LIST		Indica la lista dei campi da selezionare (separati da virgola) Formato: n (=n), -n (=n), n- (=n), n1-n2 ($\geq n_1 \&& \leq n_2$) Esempi: 3, -3, 3-, 3-3
-d DELIM	--delimiter=DELIM		Usa DELIM per dividere i campi (il delimitatore di default è la tabulazione)

- **tr [opzioni] set1 [set2]** → copia lo stdin nello stdout, effettuando le sostituzioni oppure le cancellazioni specificate (va usato ridirigendo il suo input con l'output di altri comandi) [es. `cat file.txt | tr ab BA` → visualizzo su stdout le righe di file.txt, sostituendo "a" con "B" e "b" con "A"];

Opzioni			
Formato		Significato	Effetto
Compatto	Esteso		
-c, -C	--complement	Complema insieme	Utilizza il complemento del set ₁
-d	--delete	Cancella caratteri	Cancella i caratteri indicate nel set ₁
-s	--squeeze-repeats	Elimina ripetizioni	Sostituisce ogni sequenza di un carattere ripetuto incluso nel set ₂ con una occorrenza singola dello stesso carattere

- **uniq [options] [inFile] [outFile]** → riporta oppure elimina le righe ripetute nel file in ingresso (richiede che il file sia ordinato; senza opzioni elimina le righe duplicate);

Opzioni			
Formato		Significato	Effetto
Compatto	Esteso		
-c	--count		Stampa il numero di ripetizioni prima della riga
-d	--repeated		Visualizza solo le righe ripetute
-f N	--skip-fields=N		Ignora i primi N campi per il confronto
-I	--ignore-case		Case insensitive

- **basename nome [estensione]** → elimina il path (e il suffisso se uso anche l'estensione nel comando) dal nome di un file (utile nelle manipolazioni sui nomi dei file negli script shell);
- **dirname nome** → elimina il nome del file, tenendo solo il suo path;
- **sort [opzioni] [file]** → ordina i file in input in ordine alfabetico. Ha molte opzioni:

Opzioni			
Formato		Significato	Effetto
Compatto	Esteso		
-b	--ignore-leading-blanks		Ignora gli spazi iniziali
-d	--dictionary-order		Considera solo spazi e caratteri alfabetici
-f	--ignore-case		Trasforma caratteri minuscoli in maiuscoli (case insensitive)
-I	--ignore-case		Case insensitive
-n	--numeric-sort		Confronta utilizzando un ordine numerico
-r	--reverse		Ordine inverso

Opzioni			
Formato		Significato	Effetto
Compatto	Esteso		
-k c1[,c2]	--key=c1[,c2]		Ordina sulla base dei soli campi selezionati
-m	--merge		Merge file già ordinati (senza riordinare)
-o=f	--output=f		Scrive l'output nel file f invece che su standard output

⚠ Il campo dopo il k deve essere specificato numericamente (es. se ho un file fatto da 4 colonne e voglio ordinare in base alla 2^ (e alla 3^ se la 2^ non risolve), devo usare **sort -k 2,3**) [ricorda che **sort -k 2** ordina ricorsivamente sui campi delle colonne del file a partire dal campo 2, ovvero se campo 2 uguale vado al 3 e così via, ovvero non si ferma fino al campo che ordina].

- **grep [opzioni] pattern [file]** → cerca nel contenuto di file di input le righe che hanno un “match” con il pattern fornito e le visualizza su **stdout**. La grep però usa solo le regex basic, quindi se vogliamo usare le regex extended dobbiamo usare **egrep** o **grep -E**

Opzioni			
Formato		Significato	Effetto
Compatto	Esteso		
-A N	--after-context=N		Dopo ciascun match stampa ancora N righe (oltre alla riga in cui si è trovato il match). Inserisce un separatore (--) dopo ogni insieme stampato.
-H	--with-filename		Stampa il nome del file per ogni match
-I	--ignore-case		Case insensitive
-n	--line-number		Stampa il numero di riga del match
-r, -R	--recursive		Procede in maniera ricorsiva sul sottoalbero
-v	-inverse-match		Stampa solo le righe che non fanno match

Esempio:

- Visualizzare tutte le righe dei file di estensione “txt” che contengono una stringa palindroma: **grep -E -e “(.)(.).\2\1” *.txt;**
- Per ogni file di estensione “txt” presente nel direttorio 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 > stat.txt;**
- Trovare tutti i file di estensione “txt” nel direttorio “/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” ‘{}’\;.**

→ **SHELL**: è lo strato più esterno dell’OS (fornisce interfaccia utente [interpreta i comandi utente]), ma in **UNIX** non è parte del kernel (viene trattato come un normale processo utente). Ogni shell permette i **comandi** di linea e la scrittura di “script” (programmi, scritti per evitare di ripetere sequenze di comandi, automatizzando operazioni ripetitive). Noi vedremo la shell **BASH** (dentro **/bin/sh**). Una shell può terminare con il comando **exit** o il carattere **EOF** (di solito **ctrl+d**). All’avvio della shell abbiamo dei **file di avvio** (start-up files) che possono essere file di **login** (shell viene eseguita dopo la password, si attivano **/etc/profile** [script globale] o **~/.bash_profile** [script utente]) o di **non-login** (shell eseguita digitando la sua icona o menù di sistema, si attiva **~/.bashrc**).

Importante l’**ESPANSIONE DELLA SHELL**, ovvero ci sono alcuni **caratteri** che sono **particolari** nelle shell e possono essere usati per usare delle variabili:

- **PARENTESI** → se dichiaro **nome=Gian** e poi faccio:
 - o **echo \${nome}marco** ottengo **{Gian}marco** come output;

- o `echo ${nome}marco` ottengo **Gianmarco**.
- **QUOTING:**
 - o **Apici** (‘) → identificano una stringa al cui interno non abbiamo variabili espanso;
 - o **Virgolette**(“”) → identificano una stringa al cui interno abbiamo variabili espanso;
 - o **Backslash** (\) → identifica il carattere di escape (come C).

⚠ Negli script di shell sono importanti gli spazi (`var=ciao` è diverso da `var = ciao`); negli script, le variabili contengono solo stringhe! Posso usare queste variabili con il \$ per usarle come stringhe nei vari comandi; se invece scrivessi nel terminale semplicemente `$var`, allora la shell proverebbe ad eseguire la var come un comando di nome var. Inoltre i comandi sulla stessa riga devono essere separati dai “;”.

- **OUTPUT** → possiamo prendere l'output di un comando usando `$(...)` e memorizzarlo in una variabile [es. `d=$(date)` metto l'output del comando date dentro la variabile d];
- **HISTORY** → ogni shell ricorda l'elenco degli ultimi comandi digitati (elenco contenuto nel file `.bash_history`) e possiamo usare questo elenco con i comandi **history** [mostra l'elenco], `!str` [esegue l'ultimo comando che inizia con str] e `^str1^str2` [sostituisco nell'ultimo comando str1 con la str2 e lo eseguo nuovamente];
- **ALIAS** → nelle shell possiamo definire nomi nuovi per comandi esistenti con il comando **alias** [es. `alias ciao="cd ./.."` farà in modo che ogni volta che seguo il comando `ciao`, verrà eseguito il comando `cd ./..`].

⚠ Bash è utile per fare programmi da provare/eseguire velocemente (prima di scriverli in linguaggi di programmazione più corposi [scrittura software prototipale e “quick and dirty”]) e per eseguire in maniera automatica comandi linux. Come Python, Bash è un linguaggio compilato.

Gli **SCRIPT Bash** sono memorizzati in `file.sh` (anche se in Linux posso dare ogni estensione ai file perché non è importante l'estensione).

Gli script bash devono iniziare con la riga `#!/bin/bash` (i commenti in bash sono scritti come in python con #), terminare con `exit numero` (in generale `exit 0 = true`) [ricorda che il comando echo \$? può essere usato dopo il comando per darci il valore di ritorno del comando/script] e per eseguirlo devo dargli i permessi di esecuzione (`chmod +x file.sh`).

Posso eseguirli in maniera diretta (ovvero da terminale con il nome del file, classico modo) dove viene aperta una **nuova shell**; in maniera indiretta (ovvero con il comando `source file.sh`, che mi esegue lo script riga per riga e non richiede la riga iniziale dello script bash) invece, siamo nella **stessa shell** (infatti il comando exit contenuto nello script bash, mi chiuderà anche la shell chiamante).

⚠ Non esistono tool specifici per il debug di uno script bash, ma possiamo eseguire lo script in “debug mode” (completa se debuggo l'intero script, parziale solo alcune righe dello script [che scriverò tra `set -x` e `set +x` direttamente nel file.sh]) [come con flask -debug run].

Opzioni			
Formato		Significato	Effetto
Compatto	Esteso		
-o noexec		Check senza esecuzione	Esegue un controllo sintattico ma non esegue lo script
-n		Echo comandi	Visualizza i (fa l'eco dei) comandi eseguiti
-o verbose			
-v			
-o xtrace		Traccia lo script	Visualizza la traccia di esecuzione dell'intero script
-x			
-o nounset		Check errori	Riporta un errore per variabile non definita
-u			

I **parametri/argomenti** possono essere passati da linea di comando, ovvero come `argv` in C, ma qui saranno contenuti dentro `$0` = nome dello script (come `argv[0]` in C), `$1, $2`, etc... Il comando **shift** effettua uno shift dei parametri verso sinistra (`$0` rimane invariato). Parametri speciali sono `$* = intera lista dei parametri passati`, `$# = numero dei parametri` (tolto `$0`) e `$$ = PID del processo` (utile se devo eseguire lo script da più shell).

Le **VARIABILI** contengono stringhe e si assegnano con **variabile=“valore”** e si usano con **\$variabile**. Le variabili possono essere **LOCALI** (di shell, ovvero solo nella shell corrente) o **GLOBALI** (di ambiente, disponibili in tutte le shell). Il comando **set** mi elenca tutte la **variabili locali definite** e il loro valore; il comando **unset variabile** mi cancella il valore di una variabile. Il comando **export** trasforma una variabile locale in una variabile **globale** (ovvero permette la sua visibilità anche ad altri processi); le **variabili globali elencate** con comando **env**.

Alcune variabili di ambiente sono predefinite:

- `$?` → memorizza il valore di ritorno dell'ultimo comando (0 in caso di successo, diverso da 0 in caso di errore);
- `$SHELL` → indica la shell corrente;
- `$LOGNAME` → indica lo username usato per il login;

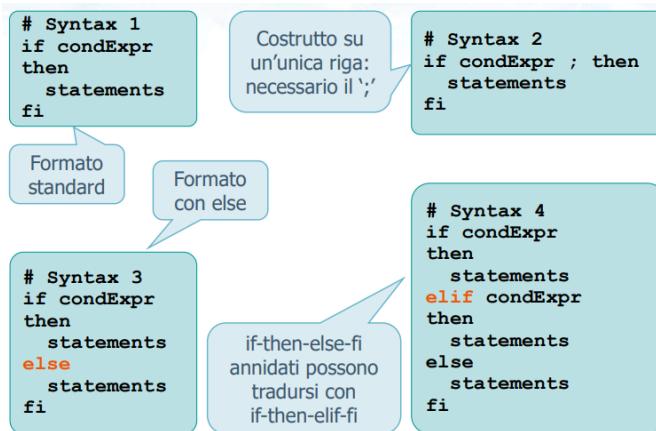
- **\$HOME** → indica la **home directory** dell'utente corrente;
- **\$PATH** → memorizza l'**elenco dei direttori** separati da ‘:

Il comando **read [opzioni] var1 var2 ... varN** legge una riga da **stdin** e salva le varie stringhe da riga di comando nelle variabili **var1, var2, ... varN**; le opzioni sono **-n nchars** (ritorna dalla lettura dopo nchars caratteri senza aspettare lo **\n**) e **-t nseconds** (ritorna 1 se non si introducono dati entro nseconds secondi).

I comandi di **output** sono **echo** e **printf**; noi useremo **echo** che visualizza i propri argomenti, separati da spazi e terminati con un carattere newline; le opzioni sono **-e** (interpreta i caratteri di escape, es. se negli argomenti ho “ciao\nciao” stampa ciao, va a capo e stampa ciao) e **-n** (sopprime il newline finale).

Per esprimere **espressioni matematiche** si usa il comando **let** (es. se ho **i=1** e poi scrivo **let "v1=i+1"** oppure **let v1=i+1** oppure **let v1=\$i+1**, mi stampa in tutti e 3 i casi il numero 2).

Il costrutto condizionale **if-then-fi** verifica se l'uscita di una sequenza di comandi è 0, e se lo è, li esegue (si può anche estendere con **if-then-else-fi** o anche annidandoli o anche con **if-then-elif-then-else-fi**):



Al posto di **condExpr** ovviamente va messa l'espressione da verificare; queste si possono scrivere con [**parametro operatore parametro**] e gli **operatori** possono essere:

Operatori per numeri		Operatori per file e direttori		Operatori per stringhe		Operatori logici	
-eq	==	-d	L'argomento è una directory	=	strcmp	!	NOT (in condizione singola)
-ne	!=	-f	L'argomento è una file regolare	!=	!strcmp	-a	AND (in condizione singola)
-gt	>	-e	L'argomento esiste	-n string	non NULL string	-o	OR (in condizione singola)
-ge	>=	-r	L'argomento ha il permesso di lettura	-z string	NULL (empty) string	&&	AND (in un elenco di condizioni)
-lt	<	-w	L'argomento ha il permesso di scrittura				OR (in un elenco di condizioni)
-le	<=	-x	L'argomento ha il permesso di esecuzione				
!	! (not)	-s	L'argomento ha dimensione non nulla				

⚠ Il comando **if [0]** (in generale **if [numero]** o anche **if [stringa]**) è sempre vero, mentre se ho **if []** allora ho il NULL, dunque falso!

**if ["\$a" -eq 24 -a "\$s" = "str"]; then
...
fi**

Formato equivalente

```
if [ "$a" -eq 24 ] && [ "$s" = "str" ]
if [[ "$a" -eq 24 && "$s" = "str" ]]
```

AND di condizioni

```
#!/bin/sh

echo -n "Is it morning (yes/no)? "
read string
if [ "$string" = "yes" ]; then
    echo "Good morning"
elif [ "$string" = "no" ]; then
    echo "Good afternoon"
else
    echo "Sorry, wrong answer"
fi

exit 0
```

Il costrutto iterativo **for var in [list] do...done** esegue i comandi specificati, una volta per ogni valore assunto dalla variabile **var** nella lista **list**:

```
for str in foo bar echo charlie tango
do
    echo $str
done
```

```
num="2 4 6 9 2.3 5.9"
for file in $num
do
    echo $file
done
```

```
n=1
for i in $* ; do
    echo "par #" $n = $i
    let n=n+1
done
```

C'è anche il costrutto iterativo **while [condizione] do...done** dove iteriamo fino a quando **condizione** è vera:

```
#!/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

```
#!/bin/bash
n=1
while read row
do
    echo "Row $n: $row"
    let n=n+1
done < in.txt > out.txt
exit 0
```

Nomi dei file costanti.
Possibile l'uso di parametri o variabili
:... < \$1 > \$var

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

⚠ Ci sono dei modi particolari di usare il **while per leggere un file** e per stampare cosa si legge, ovvero:

- **PAROLA PER PAROLA** (passo il nome del file in input da linea di comando in \$1):

```
for i in $(cat $1); do
    echo "WORD: $i"
done
```

- **RIGA PER RIGA** (passo il nome del file in input da linea di comando in \$1):

```
while read i; do
    echo "LINE: $i"
done < $1
```

- **CARATTERE PER CARATTERE** (passo il nome del file in input da linea di comando in \$1):

```
while read -n 1 i; do
    echo "CHAR: $i"
done < $1
```

⚠ Anche in bash ci sono **break** e **continue**; inoltre possiamo usare ":" per istruzione nulle (come continue)!

Per quanto riguarda i **VETTORI**, possiamo usare in bash solo **vettori monodimensionali** (non esiste limite alla dimensione di un vettore e non c'è vincolo sull'uso di indici contigui); un vettore si **definisce** con **vet[i] = "ciao"** (elemento per elemento) o con **vet=(ciao1 ciao2)** (con lista di valori separati da spazi). Con **\${vet[i]}** mi riferisco al **singolo elemento del vettore**, mentre con **\${vet[*]}** mi riferisco a **tutti gli elementi del vettore**. Con **\${#vet[*]}** trovo il **numero di elementi del vettore**, mentre con **\${#vet[i]}** trovo il **numero di caratteri dell'elemento i del vettore**. Il comando **unset vet[i]** **elimina l'elemento i del vettore** e **unset vet** **elimina l'intero vettore**.

→ Vediamo degli **ESEMPI**:

```
#!/bin/bash
while read x y z; do
    let f=3*x*x+4*y+5*z
    echo "$f"
done < $1
exit(0)
```

```
#!/bin/bash
for word in $(ls *.c); do
    egrep --quiet -e "POSIX" $word
    if [ $? -eq 0 ]
    then
        more $word
    fi
done
```

❖ Si scriva uno script in grado di calcolare i valori di una funzione $f(x)$ per tutte le terne di valori intere memorizzate in un file

➤ $f(x) = 3 \cdot x^2 + 4 \cdot y + 5 \cdot z$

➤ Esempio

1	1	2	17
2	1	3	31
1	3	4	35

Contenuto del file

Valori da calcolare e visualizzare

❖ Il nome del file sia dato sulla riga di comando

❖ Scrivere uno script bash in grado di visualizzare

➤ Tutti i file del directory corrente

➤ Con estensione ".c"

➤ Che contengono almeno una volta la stringa "POSIX"

9) SEZIONI CRITICHE (SINCRONIZZAZIONE)

Nella programmazione parallela (tramite Processi o Thread) bisogna manipolare dati condivisi, si possono verificare corse critiche e possono esistere tratti di codice non rientranti; dunque serve **SINCRONIZZAZIONE** tra processi/thread. Facendo riferimento allo **Stack** (LIFO), le funzioni di push e pop agiscono sulla stessa estremità dello stack, ma se non sincronizzo bene i processi che agiscono simultaneamente sullo stack, faccio push/pop nei momenti sbagliati (togliendo o aggiungendo dati nell'ordine errato) [in quanto la variabile "top" (cima dello stack) è condivisa]; analogamente accade su una **Queue** (FIFO).

Definiamo **SEZIONE CRITICA** (SC) [o regione critica, RC] una sezione di codice comune a più processi/thread, dove questi accedono ad oggetti comuni (competono per l'uso di risorse comuni); la soluzione è stabilire un **PROTOCOLLO DI ACCESSO** per forzare la **MUTUA ESCLUSIONE**: per entrare in una SC, il processo esegue un codice di prenotazione, mentre per uscire da una SC, il processo esegue un codice di rilascio. Dunque ogni SC è protetta da una sezione di ingresso (prenotazione o prologo) e da una sezione di uscita (rilascio o epilogo) [le sezioni non critiche non devono essere protette] [sezioni critiche indipendenti devono essere protette separatamente].

⚠ In questo corso vedremo come scrivere programmi paralleli (poi vedremo anche come renderli efficienti)!

Dunque ogni soluzione alle SC deve soddisfare dei **requisiti**:

- **MUTUA ESCLUSIONE** (ME) → 1 solo processo/thread per volta deve accedere alla SC;
- **PROGRESSO** → occorre evitare **DEADLOCK** tra processi (nessun processo fuori dalla SC può bloccarne altri);
- **ATTESA DEFINITA** → occorre evitare **STARVATION** di processi [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];
- **SIMMETRIA** → la selezione di chi deve accedere alla SC non dovrebbe dipendere dalla priorità relativa o dalla velocità relativa.

Ci sono **3 soluzioni possibili**:

1. **SOFTWARE** → fatte dal programmatore; si basano sull'uso della **variabili globali**. Vedremo il **caso con 2 soli processi/thread, denominati P_i e P_j** (dunque dato i allora $j = 1 - i$ e viceversa); vediamo le **varie soluzioni**:
 - ✓ Usare un vettore di flag di elementi `int flag[2] = {FALSE, FALSE}` che indica se la SC è "busy" oppure libera, ma non garantisce mutua esclusione; questo perché la variabile di lock viene controllata mediante 2 istruzioni a sé stanti, ovvero le 2 istruzioni sono interrompibili [non atomiche]. Oltre a ciò i cicli di controllo dei flag effettuano "attesa attiva" (**busy waiting** con spin lock, ovvero spreco di CPU time):

P_i / T_i

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

P_j / T_j

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

- ✓ Uguale a prima ma scambio l'ordine del test (ovvero il `while`) e il set (ovvero mettere il `flag` a `TRUE`); questo però potrebbe comportare **DEADLOCK** (può non esserci progresso) e presenta anch'essa il **busy waiting**:

P_i / T_i

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

P_j / T_j

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

- ✓ Usa una **variabile globale** che dice a chi spetta il turno (tra P_i e P_j); questa soluzione comporta **MUTUA ESCLUSIONE** (perché si usa il "turno"), ma anche **STARVATION** (attesa non definita) e **busy waiting**:

P_i / T_i

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

P_j / T_j

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

- ✓ Usare sia turn sia vettore flag;; questa assicura MUTUA ESCLUSIONE, elimina DEADLOCK (garantisce il progresso), elimina STARVATION (garantisce attesa definita) e garantisce SIMMETRIA (codici di i e j identici) [SOLUZIONE DI PETERSON]:

```

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

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

```

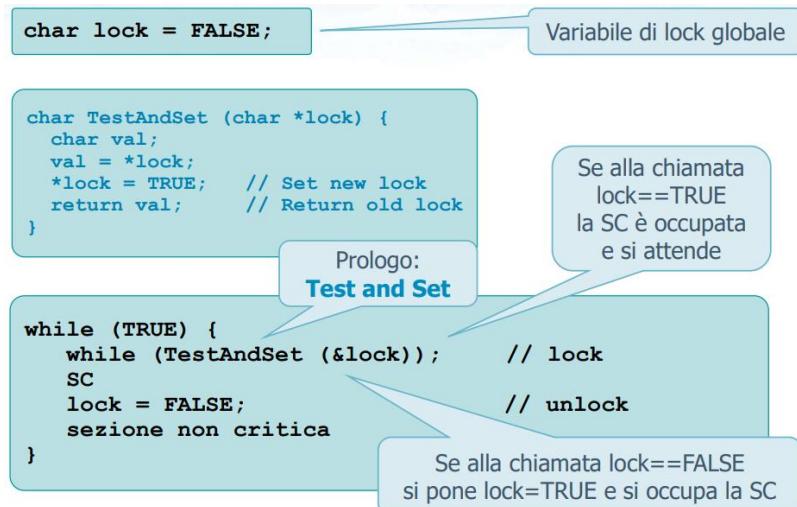
Questa soluzione comporta comunque busy waiting con spin lock (spreco di CPU time) ed è complessa!

2. HARDWARE → si basano su soluzioni architetturali che garantiscono ATOMICITÀ. Ci sono 2 tipi:

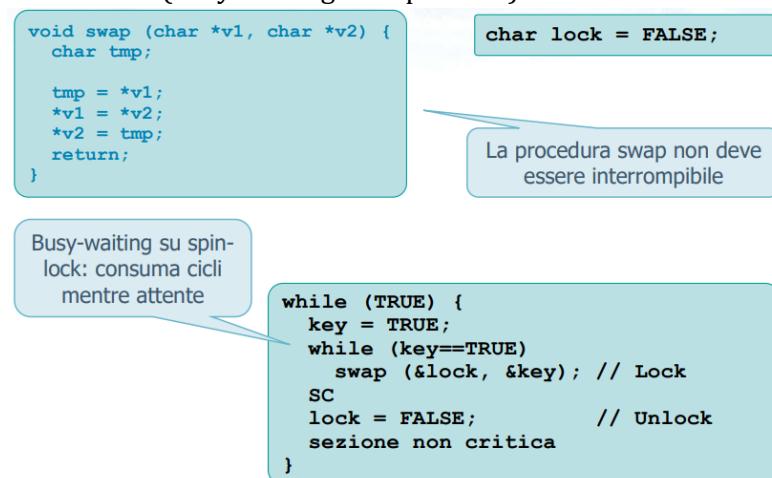
- ✓ Soluzioni senza diritto di prelazione = i processi in esecuzione nella CPU non possono essere interrotti e il controllo verrà lasciato al kernel solo quando il processo lo lascerà volontariamente; questo **elimina il problema della SC nei sistemi mono-processore** (ma noi usiamo sistemi multicore, quindi grave!!!).
- ✓ Soluzioni con diritto di prelazione = i processi in modalità kernel **possono essere interrotti** (interrupt sposta il controllo di flusso su un altro processo e poi terminerà il processo originario); questo comporta la **disabilitazione dell'interrupt** (grave!!!).

Quindi una strategia è emulare le soluzioni software, usando LOCK (lucchetti di protezione) e istruzioni ATOMICHE (indivisibili) per manipolare questi lock; ci sono **2 tipi di istruzioni atomiche di lock**:

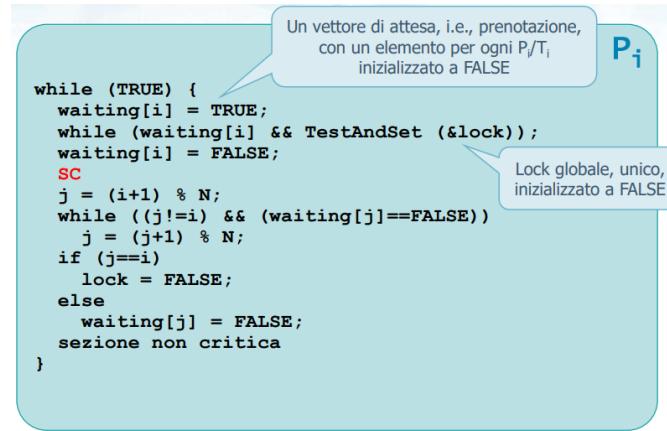
- a. **TEST-AND-SET** = setta e restituisce una variabile di lock globale. È molto più facile della soluzione SW ma deve essere atomica (non interrompibile) e lenta (busy-waiting con spin-lock):



- b. **SWAP** = scambia 2 variabili, di cui una di lock globale e una lock locale. È più facile della SW, ma anch'essa deve essere atomica e lenta (busy-waiting con spin-lock):



⚠ Entrambe le soluzioni HW potrebbero comportare starvation! Esiste una **3rd soluzione senza starvation**:



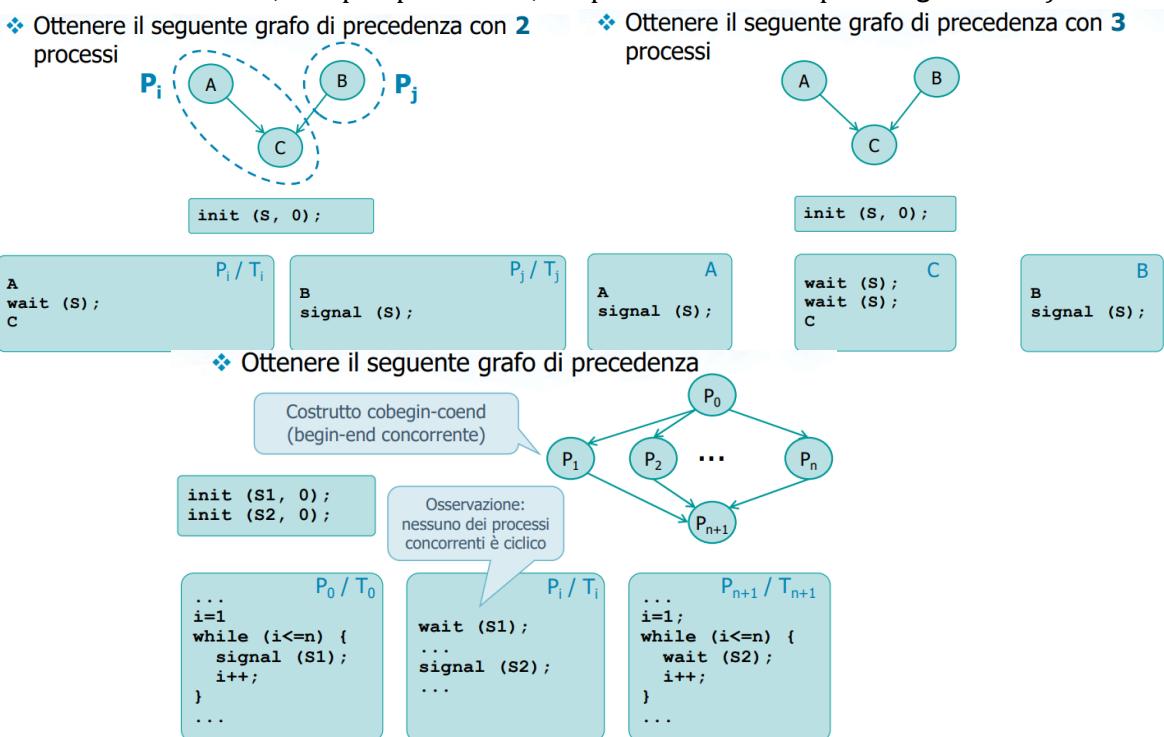
⚠ Dunque **Vantaggi HW**: utilizzabili nei multicore, estendibili a N processi (non solo 2 come le SW), semplici dal punto di vista SW (utente), simmetriche; al tempo stesso però **Svantaggi HW**: non facili dal punto di vista HW, starvation, busy-waiting con spin-lock!

3. **AD-HOC** → l'OS fornisce funzioni e strutture dati, mentre il programmatore le usa opportunamente; risultano più semplici da usare e sono più efficienti. Un **semaforo (S)** è una variabile intera condivisa (globale) protetta dall'OS, utilizzabile per inviare e ricevere segnali. L'atomicità delle operazioni sui semafori è garantita dall'OS. È impossibile per 2 processi eseguire operazioni contemporanee sullo stesso semaforo. Dato un semaforo (**S**), le operazioni standard fatte sul semaforo sono:

- ✓ **init (S, k)** = definisce (alloca) e inizializza il semaforo al valore **k**. Esistono 2 tipi di semafori (**binari** [0,1] e con conteggio [numeri interi]);
- ✓ **wait (S)** = permette (nella sezione di ingresso) di ottenere l'accesso della SC protetta dal semaforo S. Se il valore di **S** è negativo o nullo, blocca il processo chiamante (risorsa non disponibile); in ogni caso, decrementa il valore di S [da non confondere con la **wait** dei processi figli];
- ✓ **signal (S)** = permette (nella sezione di uscita) di uscire dalla SC protetta dal semaforo S. Incrementa il semaforo [da non confondere con la **signal** del gestore dei segnali];
- ✓ **destroy (S)** = libera il semaforo. Fa la free della variabile semaforica.

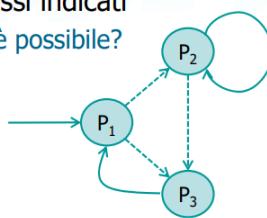
⚠ Quindi nei semafori binari posso far accedere solo 1 processo per volta nella SC (perché solo 0 e 1); nei semafori con conteggio invece, ipotizzando di voler far accedere 2 processi alla SC, devo fare **init(S,2)** così la **wait** del P1 (processo 1) mi decrementa **S=1** e la **wait** del P2 mi decrementa **S=0** (in questo modo se un altro processo P3 vuole entrare nella SC, non può perché **S=0**, ma potrà farlo solo dopo la **signal** di P1).

Esempi:



Esercizio:

- ❖ Siano dati i semafori e i processi indicati
➤ Quale **ordine** di esecuzione è possibile?



```
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 (S2);
}
...
```

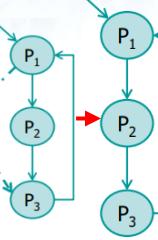
```
...
while (1) {
    wait (S2);
    SC di P3
    signal (S1);
}
...
```

- ❖ 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);
}
...
```

⚠ Nei grafi di precedenza (vedi es a dx), il tratteggio lo possiamo ignorare per la **proprietà transitiva** (perché se P2 fa wait di P1 e P3 fa wait di P2, allora P3 fa wait di P1 automaticamente [non devo mettere la wait(S2) dentro P3, altrimenti complicato]!)

⚠ Se i processi sono **ciclici**, dobbiamo usare un **semaforo diverso per ogni processo** sotto il processo iniziale (es. se da P0 partono P1, P2 e P3, devo usare S1, S2 e S3), **altrimenti posso usare lo stesso semaforo** per tutti e 3 i processi.

Nella pratica, i semafori vengono implementati con una struct contenente un contatore e una lista (coda) di processi (questo per non ricorrere all'attesa attiva "busy waiting" con spin-lock, delle altre soluzioni). Esistono diverse **implementazioni** reali dei semafori:

- **SEMAFORI TRAMITE PIPE** = data una pipe, il **contatore del semaforo** è realizzato con il **token**, la **signal** viene fatta con una **write di un token** sulla pipe (non bloccante) e la **wait** viene fatta con una **read di un token** dalla pipe (bloccante):

```
#include <unistd.h>
void semaphore_init (int *S) {
    if (pipe (S) == -1) {
        printf ("Error");
        exit (-1);
    }
    return;
}
```

```
#include <unistd.h>
void semaphore_signal (int *S) {
    char ctr = 'X';
    if (write(S[1], &ctr, sizeof(char)) != 1) {
        printf ("Error");
        exit (-1);
    }
    return;
}
```

```
#include <unistd.h>
void semaphore_wait (int *S) {
    char ctr;
    if (read (S[0], &ctr, sizeof(char)) != 1) {
        printf ("Error");
        exit (-1);
    }
    return;
}
```

- **SEMAFORI POSIX** = definiti nella libreria **POSIX**, 2 tipi: **Unnamed semaphores** [per la sincronizzazione dei thread in un processo, definiti nell'header `semaphore.h`] e **Named semaphores** (che non vedremo [per la sincronizzazione di processi]). Un semaforo è una variabile di tipo **sem_t**; le funzioni di manipolazione dei semafori (**sem_***) tornano **-1** in caso di errore; vediamole:

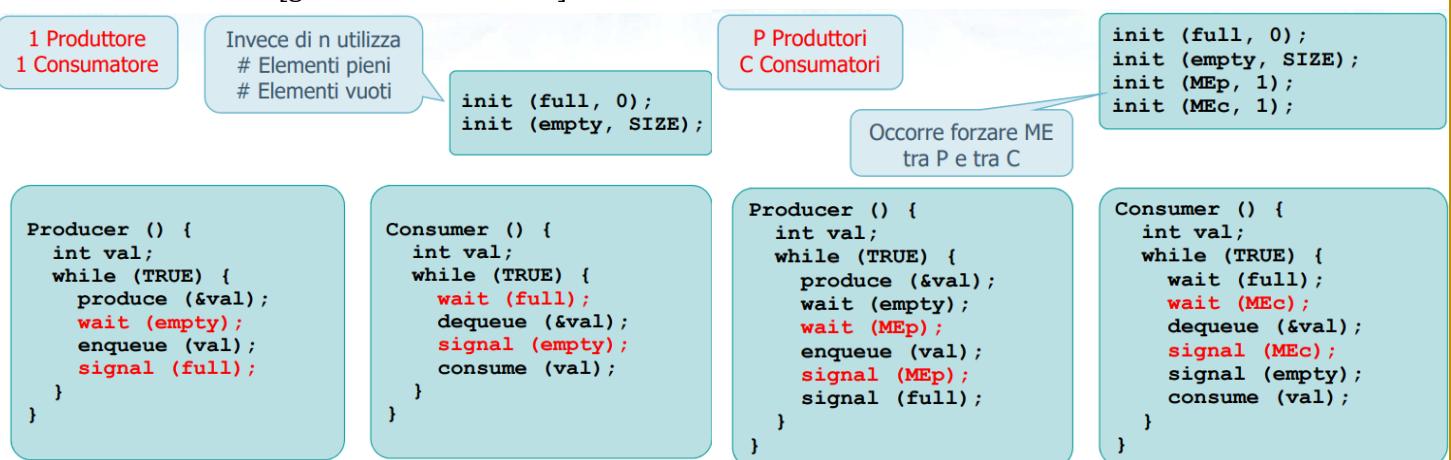
- **int sem_init (sem_t *sem, int pshared, unsigned int value);** = inizializza il semaforo al valore **value**, mentre se **pshared=0**, semaforo locale [condiviso tra threads], altrimenti semaforo condiviso [condiviso tra processi];
- **int sem_wait (sem_t *sem);** = come la wait vista prima, ovvero se **semaforo==0**, blocca il chiamante fino a quando può decrementare il valore del semaforo;
- **int sem_trywait (sem_t *sem);** = se **semaforo==0**, torna **-1** (ma non blocca il chiamante);
- **int sem_post (sem_t *sem);** = come la signal vista prima, ovvero incrementa il valore del semaforo;
- **int sem_getvalue (sem_t *sem, int *valP);** = esamina il valore del semaforo;
- **int sem_destroy (sem_t *sem);** = distrugge un semaforo (torna **-1** se cerca di distruggere un semaforo usato da un altro processo).

- **MUTEX PTHREAD** (MutEx → **mutua esclusione**) = semafori **binari** della libreria **Pthread** (tipo base `pthread_mutex_t`); funzioni:
 - **int pthread_mutex_init (pthread_mutex_t *mutex, pthread_mutexattr_t *attr);** = crea un nuovo mutex con attributi specificati [torna 0 se successo];
 - **int pthread_mutex_lock (pthread_mutex_t *mutex);** = controlla il valore del mutex e: blocca il chiamante se già locked, acquisisce il lock se non lo è [torna 0 se successo];

- `int pthread_mutex_trylock (pthread_mutex_t *mutex);` = come prima, ma non blocca il chiamante se il lock è già stato acquisito [torna 0 se lock è stato acquisito];
- `int pthread_mutex_unlock (pthread_mutex_t *mutex);` = rilascia il lock mutex al termine della SC [torna 0 se successo];
- `int pthread_mutex_destroy (pthread_mutex_t *mutex);` = rilascia la memoria occupata dal lock mutex (free) [torna 0 se successo].

PROBLEMI DI SINCRONIZZAZIONE TIPICI:

- **PRODUTTORE-CONSUMATORE** con memoria limitata → far lavorare in parallelo produttori (che mettono elementi in coda) e consumatori (che prendono elementi dalla testa) [quindi **produttori e consumatori lavorano sulle 2 estremità opposte del buffer**], senza arrivare ad avere la coda di oggetti prodotti piena o vuota; si possono avere 2 casi:
 - **1 P e 1 C** = occorre inserire un semaforo **full** che conta il numero di elementi pieni e un semaforo **empty** che conta il numero di elementi vuoti;
 - **n P e n C** = produttori e consumatori possono operare **contemporaneamente**, purchè il buffer non sia **pieno o vuoto**, ovvero devo garantire la mutua esclusione su 2 produttori consecutivi e su 2 consumatori consecutivi [garantire l'alternanza].



- **LETTORI-SCRITTORI** = condividere una base dati tra **2 insiemi di processi concorrenti**: una classe di processi (**Reader**) a cui è consentito accedere al database in concorrenza, una classe di processi (**Writer**) a cui è consentito accedere al database in mutua esclusione (sia con altri Writer, sia con i Reader). Ci sono 2 versioni del problema (entrambe rispettano la precedenza e massimizzano la concorrenza):
 - **DARE PRECEDENZA AI READER** = Reader non devono attendere a meno che un Writer sia nella SC: i Reader possono accedere in concorrenza al database; i Writer attendono fino a quando arrivano i Reader (quando anche l'ultimo Reader termina, allora si sveglia un Writer);



⚠ Si ha **nR** = variabile globale numero di reader nella SC, **meR** = semaforo di mutua esclusione per manipolare **nR**, **w** = semaforo di mutua esclusione per più writer (o per reader e writer), **meW** = semaforo di mutua esclusione per writer. I Writer sono soggetti a **STARVATION** (possono aspettare per sempre)!

- **DARE PRECEDENZA AI WRITER** = writer pronto deve attendere il meno possibile; ogni writer deve attendere che finiscano i reader, ma 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 ha **nR** e **nW** per il conteggio di Reader e Writer, **meR** e **meW** semafori di mutua esclusione per la manipolazione di **nR** e **nW**, **r** e **w** semafori di mutua esclusione reader/writer. I Reader sono soggetti a STARVATION (possono attendere per sempre).

- **TUNNEL A SENSO ALTERNATO** = permettere a più processi di procedere nella stessa direzione; se c'è traffico in una direzione, bloccare il traffico nella direzione opposta. Lo possiamo vedere come un'estensione del problema di Reader-Writer con 2 insiemi di reader; quindi avrò 2 variabili globali di conteggio (**n1** e **n2**, 1 per senso di marcia), 2 semafori (**s1** e **s2**, 1 per senso di marcia) e 1 semaforo globale di attesa (**busy**). Nella sua implementazione base può avere starvation dei processi in una direzione rispetto all'altra.

```
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 (right to left)
...
wait (s2);
n2--;
if (n2 == 0)
    signal (busy);
signal (s2);
```

- **5 FILOSOFI** = diverse risorse sono in comuni a più processi concorrenti (un tavolo ha 5 piatti di riso e 5 bastoncini, ciascuno tra 2 piatti; al tavolo ci sono 5 filosofi che pensano o mangiano, ma per mangiare hanno bisogno di 2 bastoncini e possono essere presi solo 1 alla volta). Si risolve introducendo 1 stato per ogni filosofo (**THINKING**, **HUNGRY**, **EATING**), 1 semaforo per ogni filosofo (per accedere al cibo) e 1 semaforo unico per l'accesso alla variabile di stato del filosofo stesso.

```
while (TRUE) {
    think ();
    takeForks (i);
    eat ();
    putForks (i);
}
```

```
test (int i) {
    if (state[i]==HUNGRY && state[LEFT]!=EATING &&
        state[RIGHT]!=EATING) {
        state[i] = EATING;
        signal (sem[i]);
    }
}
```

```
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);
}
```

⚠ Utili esercizi del file **u08-06-esercizi!**

10) SCHEDULING CPU

Ogni CPU viene assegnata a più processi/thread; lo **SCHEDULER** deve decidere l'algoritmo da usare per assegnare la CPU ad un processo/thread. Avevamo già parlato di quando un processo/thread venga interrotto (CONTEXT SWITCHING). Ci sono diversi tipi di scheduler:

- **SHORT-TERM** ("breve termine") → seleziona il processo a cui assegnare la CPU (interviene molto spesso);
- **MEDIUM-TERM** ("medio termine") → controlla il numero di processi in RAM (interviene spesso);
- **LONG-TERM** ("lungo termine") → controlla il grado di multiprogrammazione (interviene meno).

Lo scheduler gestisce i processi in attesa mediante la "CODA DI PROCESSI" (lista concatenata); il diagramma di accodamento specifica le gestione dei processi nelle varie code. Ci sono diversi ALGORITMI (con e senza prelazione [ovvero bloccare un processo e sottrarne la CPU]). Le prestazioni dello scheduler sono valutate usando una "funzione di costo":

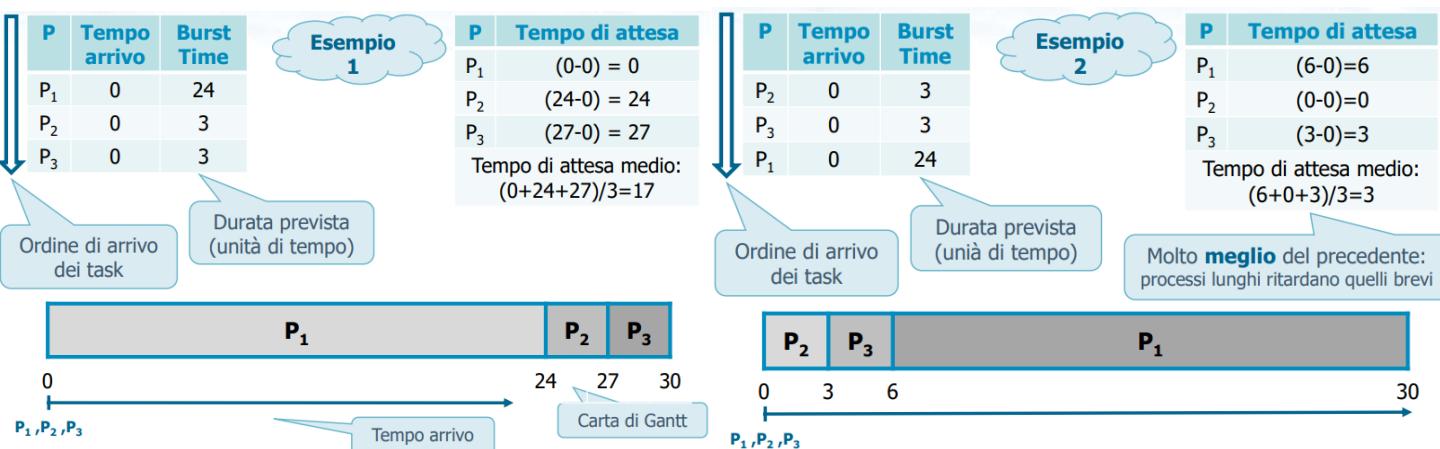
Funzione di costo	Descrizione	Ottimo
Utilizzo della CPU (CPU utilization)	Percentuale di utilizzo della CPU	[0-100%] Massimo
Produttività (Throughput)	Numero di processi completati nell'unità di tempo	Massimo
Tempo di completamento (Turnaround time)	Tempo che trascorre dalla sottomissione al completamento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

Vediamo gli **algoritmi** (li vediamo graficamente usando il DIAGRAMMA DI GANTT):

- **FCFS (First-Come First-Served)** = CPU assegnata ai task seguendo l'ordine in cui viene richiesta (FIFO).

Pro: facile

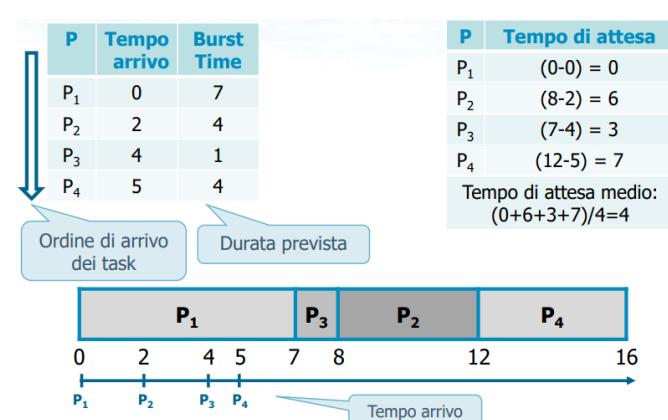
Contro: tempi di attesa lunghi, no prelazione (non utile) e effetto convoglio (task lunghi che bloccano corti)



- **SJF (Shortest-Job First)** = ad ogni processo viene associato la durata della sua prossima richiesta ("next CPU burst") e quindi lo scheduling è effettuato in ordine della durata della prossima richiesta di processi/thread.

Pro: algoritmo ottimo

Contro: starvation (attesa infinita) e difficoltà di applicazione a causa di non conoscere a priori il futuro, no prelazione (non utile)



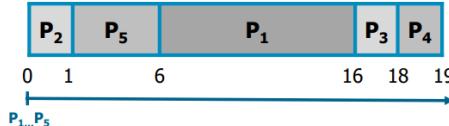
- PS (Priority Scheduling)** = ad ogni processo viene associata la sua priorità (a priorità maggiore corrisponde valore minore) e la CPU viene data al processo con la priorità maggiore (quindi come SJF ma con la priorità e non il burst time)

Contro: starvation (attesa indefinita), no prelazione (non utile)



P	Tempo arrivo	Priorità	Burst Time	P	Tempo di attesa
P ₁	0	3	10	P ₁	(6-0) = 6
P ₂	0	1	1	P ₂	(0-0) = 0
P ₃	0	4	2	P ₃	(16-0) = 16
P ₄	0	5	1	P ₄	(18-0) = 18
P ₅	0	2	5	P ₅	(1-0) = 1

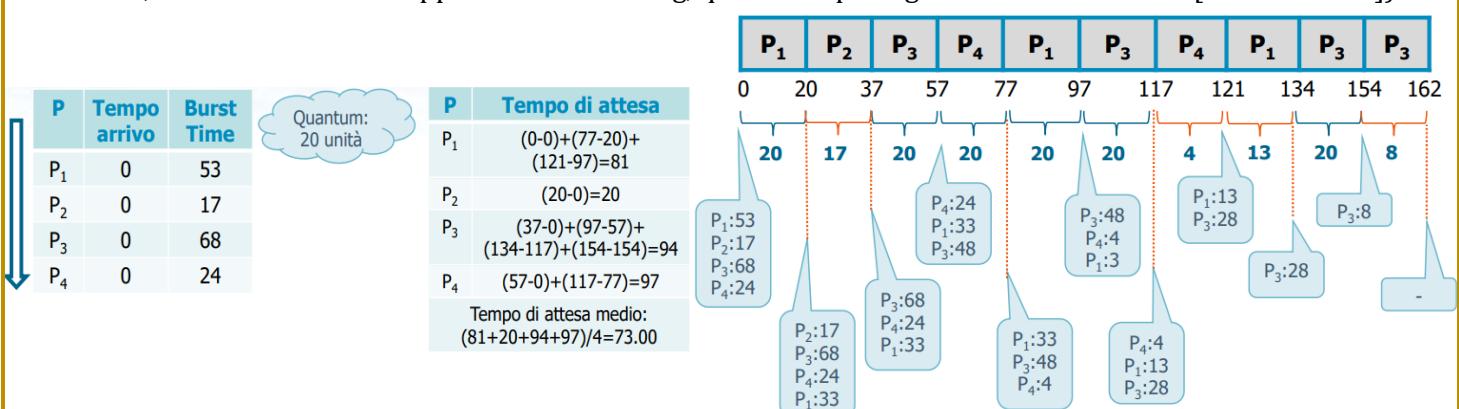
Tempo di attesa medio:
 $(6+0+16+18+1)/5=8.2$



- RR (Round Robin)** = FCFS con prelazione; l'utilizzo della CPU viene suddiviso in "time quantum" e ogni processo/thread tiene la CPU per 1 time quantum e viene poi reinserito nella coda di processi ready (gestita con modalità FIFO).

Pro: prelazione

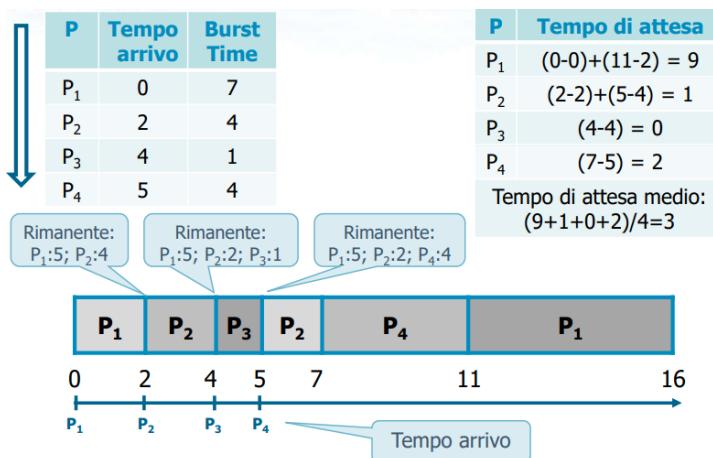
Contro: tempo di attesa medio lungo, prestazioni legate a quanto è lungo il time quantum (lungo → RR ≈ FCFS; corto → effettuati troppi context switching, quindi tempo di gestione molto elevato [overhead alto])



- SRTF (Shortest-Remaining-Time First)** = SJF con prelazione; se viene aggiunto un processo con burst time minore di quello in esecuzione, avviene prelazione.

Pro: task corti gestiti molto velocemente, context switching solo se arrivano nuovi processi con burst time minore (quindi overhead minimo)

Contro: starvation (attesa indefinita) e richiede stime accurate sul burst time dei processi/thread



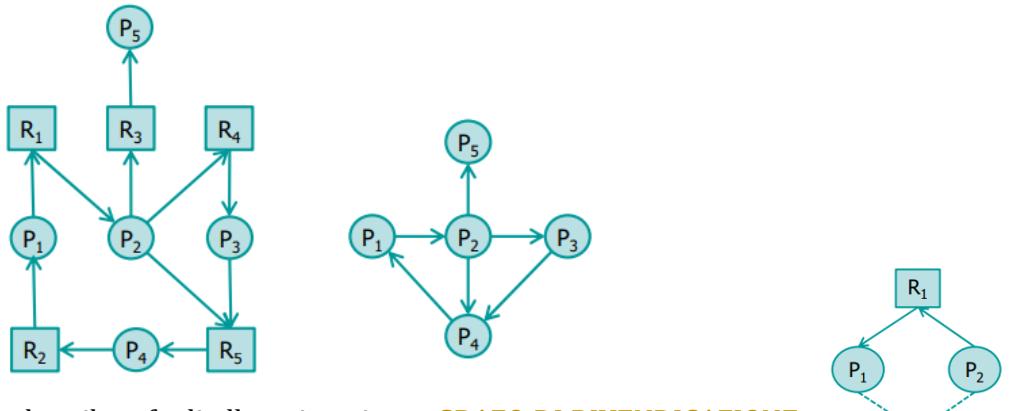
⚠ Se ho più task raggruppabili in gruppi diversi, si possono applicare più algoritmi di scheduling (MQS, Multilevel Queue Scheduling).

⚠ Lo SCHEDULING è un task che viene schedulato analogamente agli altri task; lo scheduling può essere fatto tramite processi o tramite thread (a seconda dell'OS). Inoltre finora abbiamo visto lo scheduling MONO-PROCESSORE, ma nella realtà abbiamo sistemi multicore: bisogna fare un BILANCIAMENTO del carico (ovvero delle task assegnate ai vari core).

11) DEADLOCK (STALLO)

Come abbiamo già visto lo STALLO (DEADLOCK) avviene quando un processo/thread (task) chiede una risorsa non disponibile, che ne comporta uno stato di attesa infinito (DEADLOCK implica STARVATION, non viceversa). Le CONDIZIONI affinché si verifichi il deadlock sono Mutua Esclusione, Possesso&Attesa, Impossibilità di Prelazione e Attesa Circolare (devono verificarsi tutte insieme).

Per analizzare il deadlock si usa un il GRAFO DI ALLOCAZIONE DELLE RISORSE $G = (V, E)$; i VERTICI V sono divisi in PROCESSI $P = \{P_1, \dots, P_n\}$ e RISORSE $R = \{R_1, \dots, R_m\}$ [tutti i processi e tutte le risorse sono identici, non c'è distinzione all'interno di processi e risorse]. Gli ARCHI E sono divisi in RICHIESTE $P_i \rightarrow R_j$ e ASSEGNAZIONI $R_i \rightarrow P_j$. In alcuni casi si può semplificare il grafo di allocazione in un GRAFO DI ATTESA (eliminando i vertici R e ricomponendo gli archi):



In alcuni casi invece è utile estendere il grafo di allocazione in un GRAFO DI RIVENDICAZIONE, ovvero si aggiungono delle risorse usate negli archi di reclamo (rivendicazione).

I deadlock possono essere GESTITI in vari modi:

- **A POSTERIORI:** si permette al sistema di entrare in stallo per poi intervenire; si hanno 2 fasi:
 - o **RILEVAZIONE** (deadlock detection) = se il grafo ha cicli troviamo stalli se esiste 1 sola istanza per ogni risorsa, mentre non ci sono stalli se il grafo è aciclico oppure se esiste più di 1 istanza per ogni risorsa:

❖ Processi

- P_1, P_2, P_3, P_4

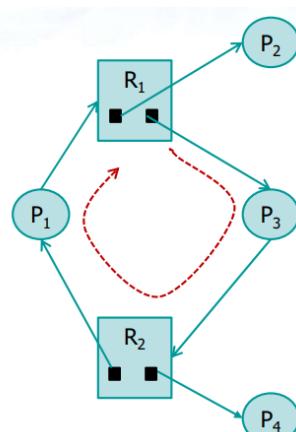
❖ Risorse

- 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



❖ Processi

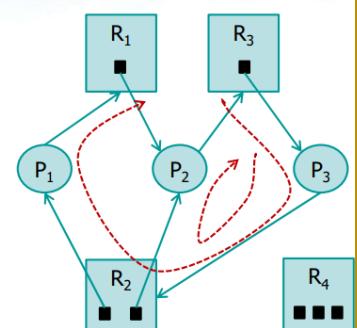
- P_1, P_2, P_3

❖ 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



Ogni fase di rilevazione ha un costo; le rilevazioni vengono effettuate ogni volta che un processo fa una richiesta non soddisfatta subito oppure ad intervalli di tempo fissi oppure quando l'uso della CPU scende sotto una certa soglia.

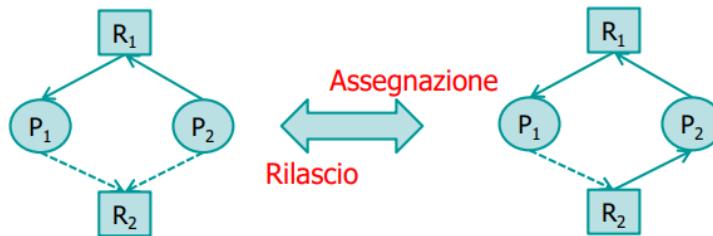
- **RIPRISTINO** (recovery del sistema) = ci sono 2 modi:
 - Agire sui **VERTICI** del grafo di assegnazione = terminare tutti i processi in stallo oppure terminare 1 processo alla volta tra quelli in stallo;
 - Agire sugli **ARCHI** del grafo di assegnazione = rimuovere le richieste verso risorse non concesse oppure prelazionare le risorse ad 1 processo alla volta.

⚠ Quindi rilevazione e ripristino sono complesse logicamente e onerose temporalmente!

- **PREVENZIONE** = cercano di prevenire il verificarsi di 1 delle 4 CONDIZIONI dello stallo; di particolare interesse (vita reale) è la prevenzione dell'**Attesa Circolare** (che avviene quando in un insieme di processi, ogni processo attende una risorsa posseduta da un altro processo), fatta imponendo un ordinamento totale tra le classi di risorse;
 - **EVITARE** = forzare i processi a fornire a priori informazioni sulle richieste che effettueranno nel corso della loro esistenza; gli algoritmi si differenziano per la quantità e il tipo di informazioni richieste, e si basano sul concetto di:
 - o **STATO SICURO** = il sistema può allocare le risorse richieste da tutti i processi in esecuzione, impedire il verificarsi di uno stallo e trovare una sequenza sicura;
 - o **SEQUENZA SICURA** = sequenza di schedulazione dei processi tale che, per ogni processo, le richieste che esso può ancora fare siano soddisfatte impiegando le risorse attualmente disponibili più le risorse liberate dai processi precedenti.

Il sistema parte in uno stato sicuro; la richiesta di una nuova risorsa sarà soddisfatta solo se lascerà il sistema in uno stato sicuro, altrimenti verrà ritardata (e il processo messo in attesa) [ricorda che stato non sicuro \neq stallo, ma semplicemente in uno stato non sicuro può avvenire uno stallo]. Abbiamo 2 algoritmi:

➤ **ALGORITMO PER ISTANZE UNITARIE** = tutte le richieste che saranno effettuate devono essere dichiarate all'inizio dell'esecuzione; quando viene fatta una richiesta, si trasforma il suo arco di reclamo in un arco di assegnazione (se non si generano cicli); quando viene rilasciata una risorsa, il suo arco di assegnazione torna ad essere arco di reclamo:



➤ **ALGORITMO PER ISTANZE MULTIPLE** (Dijkstra o “del banchiere”):

Esempio = trovare sequenza sicura:

P	Fine	Assegnate	Massimo	Necessità	Disponibili
		R ₀ R ₁ R ₂			
P ₀	T	0 1 0	7 5 3	7 4 3	3 3 2
P ₁	T	2 0 0	3 2 2	1 2 2	5 3 2
P ₂	T	3 0 2	9 0 2	6 0 0	7 4 3
P ₃	T	2 1 1	2 2 2	0 1 1	7 5 3
P ₄	T	0 0 2	4 3 3	4 3 1	10 5 5
					10 5 7

Esempio = la richiesta di $P_1(1,0,2)$ può essere soddisfatta? (ovvero si può fare subito P_1 ?) → Si si può fare:

Esempio = la richiesta di $P_4(3,3,0)$ può essere soddisfatta? No non c'è disponibilità:

P	Fine	Assegnate	Massimo	Necessità	Disponibili
		R ₀ R ₁ R ₂			
P ₀	F	0 1 0	7 5 3	7 4 3	2 3 0
P ₁	F	3 0 2	3 2 2	0 2 0	? 0 0
P ₂	F	3 0 2	9 0 2	6 0 0	
P ₃	F	2 1 1	2 2 2	0 1 1	
P ₄	F	0 0 2	4 3 3	4 3 1	1 0 1

Esempio = la richiesta di $P_0(0,3,0)$ può essere soddisfatta? No, lo stato non è sicuro:

P	Fine	Assegnate	Massimo	Necessità	Disponibili
		R ₀ R ₁ R ₂			
P ₀	F	0 1 0	7 5 3	7 4 3	2 3 0
P ₁	F	3 0 2	3 2 2	0 2 0	2 0 0
P ₂	F	3 0 2	9 0 2	6 0 0	?
P ₃	F	2 1 1	2 2 2	0 1 1	
P ₄	F	0 0 2	4 3 3	4 3 1	