-Copiare file cp [-fir] src1 src2 ... dest
-Cancellare file rm [-fir] file1 file2 …
-Spostare file mv [-fi] file1 file2 ... Dest
-Dir corrente pwd
-Creare dir mkdir <dir>
-cancellare dir rmdir <dir>
-Permessi ai file chmod permessi file
-Visualizza testo cat file , head [-n] file , tail [-n] file
-occupazione disco df [-k] [disco ...]
-Archivi tar –czvf <file>.tgz <dir> tar –xzvf <file>.tgz
-Conteggi: wc [options] [file...]
Valuta numero di: c byte w parole l linee
-diff tra 2 file/2 dir diff [-opzioni] file1 file2
Elenca le righe diverse(num di riga)
Opz: b Ignora spazi fine riga. i Ignora maius/min -w Ignora spaziatura

FIND
find directory options actions
OPT: -name pattern -regex expr + -> >= - -> <=
-size [+,-]n[bckwMG] dimensione del file -type tipo
-mindepth n a partire dalla profondità -maxdepth n sino alla profondità n
ACT:
find . –name "*.old" -type f –exec rm –f \{} \; Rimuove i file elencati
find / -user root –exec cat \{} \; Visualizza ifile elencati concatenati
find . –name "*.txt" –exec head –n 2 \{} \; le prime 2 righe dei file

Espressioni regolari
Le espressioni regolari sono pattern che descrivono un insieme di stringhe e sono utilizzate per effettuare l'accoppiamento (**match**) tra oggetti.
- . Un carattere qualsiasi
¬ [c1c2] Uno qualsiasi dei caratteri in parentesi
¬ [c1-c5] Uno qualsiasi dei caratteri nel range
¬ [^c1-c5] Uno qualsiasi dei caratteri **non** nel range
¬^ Inizioriga
¬$ Fine riga
¬ \+ Match con +
¬ \? Match con ?
¬ \< Inizio parola
¬ \> Fine parola
¬ \w un qualsiasi carattere in una parola
¬ \W un qualsiasi carattere non in una parola
¬ * Elemento presente [0, ∞] volte
¬ + Elemento presente [1, ∞] volte
¬ ? Elemento presente [0, 1] volte
¬ {n} Elemento presente esattamente n volte
¬ {n,} Elemento presente n o più volte
¬ {n1,n2} Elemento presente da n1 a n2 volte
¬ [a-zA-Z0-9] Una lettera o una cifra
- \(.\)\(.\).\2\1 stringa palindroma di 5 caratteri

FILTRI è un comando che:Riceve il proprio input da standard input,Lo manipola (lo filtra) secondo determinati parametri e opzioni, Produce il suo output su standard output.
Sostanzialmente sono comandi che:Permettono un qualche tipo di manipolazione di testi,sono spesso utilizzati come comandi in **pipeline(|)**

cut [options] [file]
-d " " il delimitatore è lo spazio(deafualt tabulatore
-f 1,3 seleziona i campi 1 e 3 di tutte le righe
-c a cancella tutte le a
tr [options] set1 [set2]
tr –d ab < file.txt Visualizza righe dovei eliminati i caratteri a, b

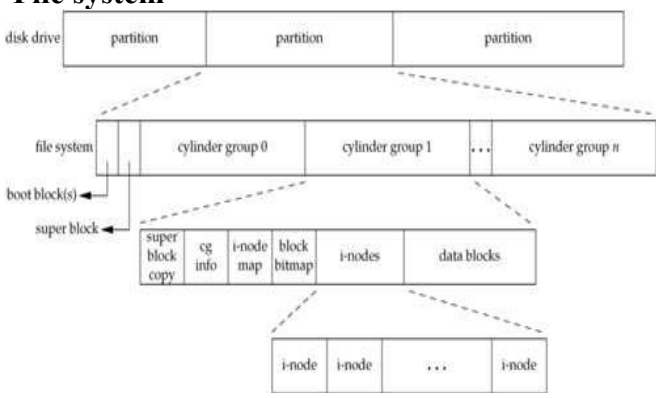
uniq [options] [inFile] [outFile]
-c Stampa numero ripetizioni prima della riga
-d Visualizza solo le righe ripetute
-f N Ignora i primi N campi per il confronto

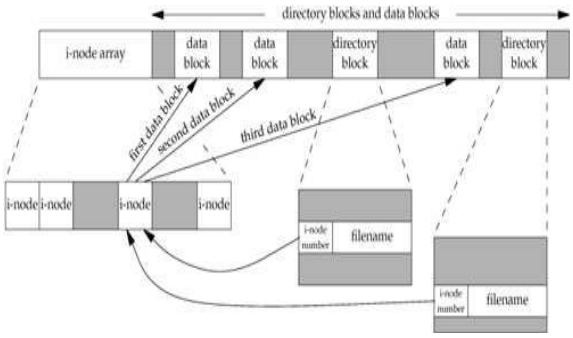
sort [opzioni] [file]
-b ignora gli spazi iniziali
-d Considera solo spazi e caratteri alfabetici
-f Trasforma caratteri minuscoli in maiuscoli
-n Confronta utilizzando un ordine numerico
-r Ordine inverso
-k c1[,c2] Ordina sulla base dei soli campi selezionati
-m Merge file già ordinati (senza riordinare)
-o=f Scrive l'output in f invece che su stdout

grep[opzioni] pattern [file]
-e PATTERN Specifica i pattern da ricercare (uno o più)
-A N Dopo ciascun match stampa ancora N righe
-H Stampa il nome del file per ogni match
-I Case insensitive
-n Stampa il numero di riga del match
-R Procede in maniera ricorsiva sul sottoalbero
-v Stampa solo le righe che non fanno match

grep –e "l." –e a file.txt
tutte le righe che contengono una l seguita da un altro carattere qualsiasi, oppure una a
grep –H –A 4 abc file.txt
● Stampa tutte le righe che contengono la stringa "abc", stampa tali righe e le 4

File system





Un i-node:È un record di lunghezza fissa che contiene la maggior parte di informazioni relative ai file E Ha un contataore che individua il numero didirettori che puntano ad esso

-Il **kernel** virtualizza diverse partizioni come se fossero tutte parte dello stesso albero.-A ogni file di qualunque tipo è associato un inode number, l'indice di una tabella in cui ogni elemento (inode) contiene le informazioni di un file. -Le directory sono delle tabelle che associano al nome di un file il suo inode number.-Tale associazione si chiama link.

- **Il file system** è organizzato in blocchi logici contigui:¬ Dimensione fissa di 1024, 2048 o 4096 byte
¬ Indipendente dalla dimensione del blocco fisico (generalmente 512 byte)
- Un blocco speciale, detto **super block**, in posizione fissa all'inizio del FS ne descrive le caratteristiche(Dimensioni, struttura, Max mount count, State)

LINK: Hard link -Directory entry che punta a un inode puntato da un'altra directory entry¬ No cross-file system
- Soft link¬ Directory entry che punta a un inode che punta a un blocco contenente il path name del file corrispondente
¬ un file che come unico blocco dati ha il nome di un altro file
¬ Nessun hard link a una directory¬ Nessun hard link a file su un altro file system¬ Un file è fisicamente rimosso solo quando tutti i suoi hard link sono stati rimossi

Il puntatore da un direttorio al rispettivo i-node è detto **hard-link.**

Inode:Ogni file o directory è associato ad un inode
Descrive le caratteristiche del file e Identifica i blocchi di cui è composto il file, Numerati a partire da 1, Alcuni sono riservati al sistema

Directory: È un file che contiene record di tipo directory entry,Una directory entry associa un nome file al suo inode

File descriptor: Individua univocamente un file aperto
- Ciascun processo può utilizzare fino a OPEN_MAX (**<limits.h>**) file descriptor.File descriptor di default
¬ 0 standard input¬ 1 standard output¬ 2 standard error

ln crea un hard-link
L'opzione **–s** permette al comando di creare un link simbolico

```
ln [opzioni] source_file [target_file]
ln source alias
ln –s source alias
```

rm rimuove il file solo se il numero dei link è uguale a 0
mv equivale a eseguire prima **ln** e poi **rm**

INPUT/OUTPUT POSIX
open, read, write, lseek, close
Tale tipologia di accesso Fa parte di POSIX e della Single UNIX Specification ma non di ISO C.Si indica normalmente con il termine di "unbuffered I/O" nel senso che ciascuna operazione di read o write corrisponde a una system call al kernel**.**

Nel kernel UNIX un "**file descripto**r" è un intero non negativo
¬ Standard input = descrittore 0 = STDIN_FILENO
¬ Standard output = descrittore 1 =STDOUT_FILENO
¬ Standard error = descrittore 2 = STDERR_FILENO
definiti nel file di header **unistd.h**

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

int open (const char *path, int flags, mode_t mode);
ϖ Apre un file definendone i permessi
ϖ Valori di ritorno
¬ Il descrittore del file in caso di successo
¬ Il valore -1 in caso di errore

Può avere 2 oppure 3 parametri
¬ Il parametro **mode** è opzionale
ϖ **Path** indica il file da aprire
ϖ **Flag** ha molteplici opzioni
¬ Si ottiene mediante l'OR bi-a-bit di costanti
presenti nel file di header **fcntl.h**
¬ Una delle tre seguenti costanti è obbligatoria
• O_RDONLY open for read-only access
• O_WRONLY open for write-only access
• O_RDWR open for read-write access

int read (int fd, void *buf, size_t nbytes);
Legge da file fd nbytes, memorizzandoli in buf
¬successo -1, 0 ERRORE se EOF

int write (int fd, void *buf, size_t nbytes);
Scrive nbytes byte di buf nel file fd
ritorno:¬ Il numero di byte scritti in caso di successo(nbytes)
Il valore -1 in caso di errore

int close (int fd); 0 successo -1 errore

Un esempio di R/W
#include <sys/stat.h> #include <fcntl.h> #include <unistd.h>
#define BUFFSIZE 4096

int nR, nW, fdR, fdW; char buf[BUFFSIZE];
fdR = open (argv[1], O_RDONLY);
fdW = open (argv[2], O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);
if (fdR==(-1) || fdW==(-1)) {fprintf (stdout, "Error \n");exit (1);}

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

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

Stat manipolazione dei direttori(FYLE SYSTEM)

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

-stat restituisce la struttura **stat** per il file (o riferimento) passato come parametro. 0 successo -1 errore
-lstat restituisce informazioni sul link simbolico non sul file puntato
-fstat restituisce informazioni su un file già aperto

struct stat {
mode_t st_mode; /* file type & mode */
ino_t st_ino; /* i-node number */
...};
-Il secondo argomento di **stat** è il puntatore alla struttura **stat -**Il campo **st_mode** codifica il tipo di file

ESEMPIO
#define N 100
struct stat statbuf; DIR *dp;
char fullName[N]; struct dirent *dirp; int i;

if (**lstat**(argv[1], &statbuf) < 0) {fprintf (stderr, "Error"); exit(1);}

if (S_ISDIR(statbuf.st_mode) == 0) {ERRORE}
if ((dp = **opendir**(argv[1])) == NULL) {ERRORE}
i = 0;
while ((dirp = **readdi**r(dp)) != NULL)
{
sprintf (fullName, "%s/%s", argv[1], dirp->d_name);
if (**lstat**(fullName, &statbuf) < 0){ERRORE}

if (S_ISDIR(statbuf.st_mode) == 0) {
fprintf (stdout, "File %d: %s\n", i, fullName);
}
else { fprintf (stdout, "Dir %d: %s\n", i, fullName);}
i++;
} if (**closedir**(dp) < 0) {ERRORE}

char *getcwd (char *buf, int size); // dir corrente
int chdir (char *path); // cambia dir
Ottiene il path del direttorio di lavoro
ritorno ¬ getcwd Il buffer buf se è OK; NULL se c'è errore
ritorno ¬ chdir Il valore 0 se è OK; il valore -1 se c'è errore

int mkdir (const char *path, mode_t mode);
int rmdir (const char *path);
Creano un nuovo direttorio (vuoto) o lo cancellano (se vuoto)
ritorno¬ Il valore 0 se è OK¬ Il valore -1 se c'è errore

#include <dirent.h>
DIR *opendir (const char *filename);
Apre un direttorio in lettura
valori di ritorno: Il puntatore al direttorio se corretta o NULL

struct dirent {
inot_t d_no;
char d_name[NAM_MAX+1];
...}
La truttura **dirent** (**DIR ***) ritornata
¬ Dipende dall'implementazione ¬ Contiene almeno i campi indicati:
• Il numero di inode• Il nome del file (null-terminated)

struct dirent readdir (DIR *dp);
ϖ Valori di ritorno ¬ Il puntatore alla struttura dirent se corretta

int closedir (DIR *dp);
ϖ Valori di ritorno ¬ 0 successo -1 errore

TIPO FILE:
¬ **S_ISREG** regular file, **S_ISDIR** directory,
S_ISBLK block special file, **S_ISCHR** characterspecial file,
S_ISLNK symbolic link

PROCESSO
Sequenza di operazioni effettuate da unprogramma in esecuzione su un determinato insieme di dati di ingresso.

Processo sequenziale
Esiste un totale relazione di ordinamento.
Dato un stesso input produce sempre lo stesso output.
Comportamento deterministico.

Processi concorrenti
Due o più processi sono concorrenti se le loro operazioni si sovrappongono nel tempo.
Comportamento non deterministico
La concorrenza reale è attuata solo nei sistemi multi-processore.

Thread
Istanza attiva di un processo.L 'evoluzione temporale di un processo può essere analizzata attraverso la sua traccia.

Esecuzione processi: in concorrenza o con sincronizzazione.

Identificatore processi: 0->schedulatore processi 1->processo init

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

fork:¬ Genera un processo detto "**processo figlio**"
Il figlio è una copia identica al padre tranne che per il Process ID

● Il padre riceve l'ID del figlio ● Il figlio riceve il valore 0
Il figlio Condivide con il padre: il codice (C) sorgente,Possiede una propria copia dello spazio dati.

#include <unistd.h>
pid_t fork (void);
¬ Se l'operazione si conclude correttamente
● Il PID del figlio nell'istanza di codice del padre
● Il PID zero nel figlio
¬ Il valore -1 se non è possibile allocare un nuovo process: si è raggiunto il limite sul numero diprocessi

DIFFERENZA tra padre e figlio
¬ Il valore ritornato dalla fork ¬ Il loro PID
¬ Il PID del relativo padre
● Il PID del padre del figlio è quello del padre stesso
● Il PID del padre del padre non cambia

Terminazione processo:
standard: return, exit
anormale: abort, signal di terminazione.

Quando un processo termina tanto in maniera normale quanto anomala:
Il kernel invia un segnale (**SIGCHLD**) al padre
L'evento è ovviamente asincrono e il padre può decidere di:

● Gestire la terminazione del figlio (e/o il segnale)
● Ignorare tale evento
¬ L'azione di default è ignorarlo

¬ Se decide di gestirlo può effettuarne la gestione:
● Mediante un gestore del segnale **SIGCHLD** (vedere segnali)
● Mediante chiamate a wait o waitpid

Quando un processo effettua una wait
¬ Si ferma se tutti i suoi figli sono ancora in esecuzione
¬ Ottiene immediatamente lo stato di terminazione di un figlio, se almeno un figlio è terminato ed è in attesa che il suo stato di tarminazione sia recuperato
¬ Ritorna con un errore se non ha figli

ZOMBIE
Un processo terminato per il quale il padre non ha ancora eseguito una wait si dice zombie.
¬ Il segmento dati del processo non viene rimosso dalla process table per tenere traccia dello stato di uscita
¬ L'entry viene rimossa solo dopo che il padre ha eseguito una **wait**
¬ Se il padre termina prima di eseguire la **wait** ilprocesso figlio viene ereditato dal processo init(quello con PID=1) e il figlio non diviene piùzombie alla terminazione

#include <sys/wait.h>
pid_t wait (int *statLoc);
¬ Blocca il processo che la chiama se non ci sono figli terminati
¬ Ritorna subito se almeno uno dei figli del processo è già terminato o non appena una dei figli termina se tutti i figli sono in esecuzione.

Il parametro statLoc
¬ È un puntatore a un intero
● Se non è NULL specifica lo stato di uscita del processo figlio (valore restituito dal figlio)

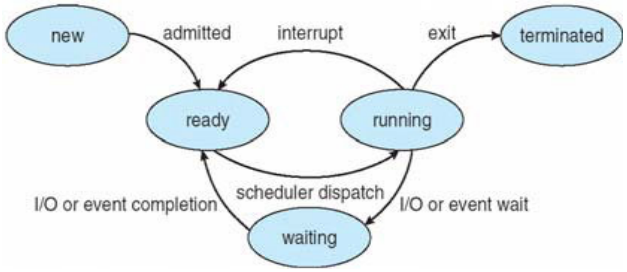
¬ Le informazione di stato sono
● Implementatio dependend
● Interpretabili con delle macro presenti in **<sys/wait.h>** (WIFEXITED, WIFSIGNALED, etc.)
Restituisce¬ Il PID del processo figlio terminato

La **wait** si sblocca quando un qualsiasi processo figlio termina.
Figlio specifico con wait:
controllare il pide del figlio terminato.
Memorizzare pid del fiflio terminato nella lista dei processi figlio terminati.
Effettuare un altra wait sino a quando termina il figlio desiderato.

`pid_t waitpid (pid_t pid,int *statLoc,int options);`
ϖ Se il parametro pid
¬ È ==-1 attende un qualsiasi figlio (waitpid==wait)
¬ È >0 attende il figlio con quel PID
Il parametro options permette controlli aggiuntivi

Programma: entità passiva, sequenza di linee di codice.
Processo: Programma in esecuzione,entità passiva.

STATO PROCESSO (durante la sua esecuzione)
¬ **New**: il processo viene creato e sottomesso al SO
¬ **Running**: in esecuzione
¬ **Ready**: Logicamente pronto ad essere eseguito, in attesa della risorsa processore.
¬ **Waiting**: in attesa della disponibilità di risorse da parte del sistema.
¬ **Terminated**: Il processo termina e rilascia le risorse utilizzate



PCB process control block
Il SO tiene traccia di ogni processo associando:
¬ Stato del processo ● New, Ready, Running, Waiting,Terminated
¬ Program counter ● Indirizzo succ istruzione da eseguire
-Registri della CPU● In numero e tipo dipendentedall'hardware
¬ Informazioni utili per loscheduling della CPU
¬ Informazioni utili per la gestione della memoria
- Informazioni amministrative varie● Tempo di utilizzo CPU, limiti,
¬ Informazioni sullo stato delle operazioni di I/O

I processi possono essere classificati in
¬ **I/O-bound**
● Passano più tempo effettuando I/O che calcoli● Richiedono molti servizi corti da parte della CPU
¬ CPU-bound
● Pssano più tempo effettuando calcoli che I/O● Richiedono pochi servizi molto lunghi da parte dellaCPU

CONTEXT SWITCHING
Quando la CPU viene assegnata ad un altro processo, il kernel deve:
¬ Salvare lo stato del processo running
¬ Caricare lo stato salvato precedentemente per il nuovo processo

-Il tempo dedicato al context switching è overhead, cioè lavoro non utile direttamente ad alcun processo
- Il tempo che un sistema usa per il context switching dipende dall'hardware a disposizione

PROCESS SCHEDULING QUEUES
L'obiettivo della **multiprogrammazione** è quello di massimizzare l'utilizzo della CPU da parte dei processi

-Il SO gestisce quindi i processi mediante uno **scheduler:**
¬ Lo scheduler inserisce ciascun processo in una **coda**
¬ Esistono diverse code (ready, I/O)

Il **diagramma di accodamento** specifica la gestione dei processi (transizioni) nelle varie code:
¬ Coda dei "processi pronti"
Code dei dispositivi

Schedulatori
Scheduler a breve termine
¬ Seleziona prevalentemente processi per la CPU
¬ Interviene molto frequentemente
¬ Rischedula a tempi dell'ordine dei millisecondi
¬ Deve essere molto veloce

Scheduler a lungo termine
¬ Interviene molto meno frequentemente
¬ Rischedula a tempi dell'ordine dei secondi/minuti
¬ Seleziona quale processo inserire nella ready list e pronti all'esecuzione in memoria centrale
¬ In pratica controlla il grado dimultiprogrammazione

I comandi di shell (in **foreground**) permettonodi eseguire processi in modo **sequenziale**
¬ Esegue il processo in maniere indipendente dalla shell
¬ Lascia il terminale libero per altri
¬ Esegue processi in parallelo

Un job è un¬ Concetto di "shell"¬ Una shell raggruppa uno o più processi in un job¬ In altre parole un job è un processo eseguito da
una shell¬ La shell tiene traccia dei job in una "job table"

Il comando **ps** (**process status**)
¬ Elenca i processi attivi e i relativi dettagli
● ps <opzioni>¬ Permette le seguenti opzioni● –e Elenca tutti i processi
● –f Visualizza il formato esteso● –u <user> Processi dell'utente <user>

JOB:
è un concetto di shell, una shell raggruppa uno o piu processi in un job ovvero un job è un processo eseguito da una shell.
La shell tiene traccia dei job in una **job table**

Ogni job è caratterizzato da un hob identifier.
Il comando jobs mostra la lista dei bob attivi in background eseguiti da quella shell.

PS processus status
Elenca i processi attivi e i relativi dettagli
ps opzioni
-e elenca tutti i proc
-f visualizza formato esteso
-u user processi dell utente x

job background: bg %jobid
termina processo: kill -9 pid

Fork e Exec

fork

¬ Un processo desidera duplicarsi in modo che padre e figlio possano eseguire sezioni diverse del codice
¬ Un processo vuole eseguire un programma differente

exec sostituisce il processo con un nuovo programma che incomincia l'esecuzione in maniera standard (dal main):
¬ Non crea un nuovo processo, ovvero il PID del processo non cambia
¬ Sostituisce l'immagine del processo corrente (i.e., ilsuo codice, i suoi dati,) con quelli di un processo nuovo

durante la exec¬ Vengono mantenuti tutti i file descriptor esistenti
-Questo comportamento è necessario per ereditareeventuali redirezioni impostate nei comandi di shell una volta eseguita la exec

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

-La system invoca il comando string all'interno di una shell

¬ exec**l** , exec**lp**, exec**le** ¬ exec**v**, exec**vp**, exec**ve**
¬ **l** (list): la funzione riceve una lista di parametri
¬ **v** (vector): la funzione riceve un vettore di argomenti argv[]
¬ **p** (path): la funzione riceve il nome del file e lo rintraccia tramite la variabile d'ambiente PATH
¬ **e** (environment): la funzione riceve un vettore di environment envp[] inveci di utilizzarel'environment corrente

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

```
execl ("ls -laR", "myLs", "/home", NULL);
execl ( "/bin/sh","sh","-c", cmd, (char *)0);
execv ("/bin/ls", cmd);
execlp (command, arg0, ..., argn, 0);
execvp (command, argv);
execle (path, arg0, ..., argn, 0, env);
execve (path, argv, env);
```

```
path: programma da eseguire
lista argomenti o il path
variabili di ambiente
RITORNA: nessuno in caso successo invece -1 errore
```

CONCORRENZA

Il flusso di esecuzione sequenziale può essere alterato per aumentare l efficenza di elaborazione.
Sistemi multi processori permetto esecuzioni in concorrenza

grafo di precedenza è un grafo aciclico diretto in cui (Vincoli di precedenza rappresenati grazie al grafo)

¬ I vertici corrispondono a ● Istruzioni singole i
● Processi
¬ Gli archi corrispondono a condizioni di precedenza:
● Un arco dal vertice A al vertice B significa che B può essere eseguito solo una volta terminato A

Condizioni per la concorrenza

Dato un'istruzione S (statement/processo)definiamo
¬ R (S)
● Read set di S,insieme delle variabili referenziate dall'istruzione S
¬ W (S)
● Write set di S, ovvero l'insieme d

CONDIZIONI DI BERNSTEIN

Due processi Si e Sj possano essere eseguiti in parallelo
¬ R (Si) ∩ W (Sj) = 0
¬ W (Si) ∩ R (Sj) = 0
¬ W (Si) ∩ W (Sj) = 0

ϖ In caso contrario si hanno errori dipendenti dalla schedulazione scelta
ϖ Quando le condizioni di Bernstein non sono soddisfatte i processi operano in **Mutua Esclusione**

Siano date le seguenti operazioni aritmetiche(Esemopio)

S1. a= x+y
S2. b= z+1
S3. c= a-b
S4. w=c+1

Il flusso di esecuzione sequenziale è 1->2->3->4 può essere alterato in quanto:
-istruzione 3 deve essere eseguita dopo 1 e 2
-istruzione 4 deve essere esguita dopo 3
-le istruzioni 1 e 2 possono essere eseguite in parallelo.

ϖ Data la sequenza di istruzioni(esempio)Concorrenza
¬ S1. a = x + y
¬ S2. b = z + 1
¬ S3. c = a – b
¬ S4. w = c + 1
ϖ Si ha che
¬ R (S3) = {a, b}
¬ W (S3) = {c}
¬ R (S4) = {c}
¬ W (S4) = {w}

Data la sequenza di istruzioni(ESEMPIO) BERNESTEIN
S1: a = x + y
S2: b = z + 1
S3: c = a – b
S4: w = c + 1
ϖ Si ha che
¬ R (S1) ∩ W (S2) = 0
¬ R (S2) ∩ W (S1) = 0
¬ W (S1) ∩ W (S2) = 0
…

SEGNALI
Un segnale è
¬ Un interrupt software/ Un evento di sistema inviato a un processo da un altro processo

I segnali permettoo di gestire eventi asincroni
¬ Sono utilizzati per notificare il verificarsi di eventi particolari
● Condizioni di errore, violazioni di accesso in memoria.
-Possono venire utilizzati per la comunicazione tra processi

SEGNALI PRINCIPALI
Il file **signal.h** definisce i nomi dei segnali
¬ SIGABRT ● generato chiamando la funzione abort
¬ SIGALRM● Alarm clock, generato dalla funzione alarm
¬ SIGILL ● Illegal instruction
¬ SIGINT ● Terminal interrupt
- SIGKILL● Kill (non mascherabile)
¬ SIGPIPE ● Write on a pipe with no reader
¬ SIGQUIT ● Terminal quit
¬ SIGCHLD ● Child process stopped or exited
¬ **SIGUSR1**● Il comportamento di default è la terminazione
¬ **SIGUSR2**● Simile a SIGUSR1

I segnali sono:
¬generati quando il processo sorgente effettua l'evento necessario
¬ consegnati quando il processo destinatario assume le azioni richieste dal segnale

ϖ Un segnale non consegnato risulta pendente
ϖ Un segnale ha un tempo di vita che va dalla sua generazione alla sua consegna

Gestione di un segnale
Per gestire un segnale un processo deve dire al kernel che cosa intende fare nel caso riceva tale segnale. Ogni processo che prevede di ricevere un segnalepuò decidere di:
1)Lasciare che si verifichi il comportamento di default
● Ogni segnale ha un comportamento di default definito dal sistema
● La maggior parte dei compostamenti di defaultconsistono nel terminare il processo

2)Ignorare esplicitamente il segnale
● Tale comportamento può essere applicato alla maggior parte dei segnali tranne che per i segnali SIGKILL e SIGSTOP che non possono essere ignorati
● SIGKILL e SIGSTOP permettono al kernel e allo superuser di avere un qualche controllo su tutti i processi
3)Catturare (catch) il segnale
● Viene realizzato dicendo al kernel di richiamare una funzione specifica nel caso in cui si verifichi il segnale

GEMERARE ALARM
```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
void myAlarm (int sig) {
printf ("Alarm\n");
}

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

void (*signal (int sig,void (*func)(int)))(int);
La system call **signal** consente di instanziare un gestore di segnali, i.e., di settare un signal handler.
Sig: Segnale da intercettare
func:L'indirizzo (i.e., il puntatore) della funzione da invocare quando si presenta il segnale restituisce la funzione che gestiva il segnale **sig** in precedenza.
(deafaul: sig_dfl ignora: sign_ign cattura: signalhandlerfunc)

int kill (pid_t pid, int sig);
Invia un segnale a un processo Se il valore di pid è:
>0● Invia il segnale al processo di PID=pid
==0● Invia il segnale a tutti i processi di uguale group id
<0 invia segnale a tutti i processi di group id uguale a pid
==-1 invia segnale a tutti i processi del sistema
(raise(sig) // invia segnale a se stesso)

int pause (void);
Sospende il processo chiamante sino all'arrivo di un signale
Ritorna solo quando viene eseguito un gestore di segnali e al termine della sua esecuzione questo ritorna

unsigned int alarm (unsigned int seconds);
Permette di attivare un timer che terminerà a un preciso istante futuro.Quando il count-down termina il segnale **SIGALARM** viene generato

LIMITI DEI SEGNALI
Nessun tipo di dato viene consegnato con i segnali tradizionali.
La memoria dei segnali"pending" è limitata
Richiedono funzioni rientranti.
Producono race conditions

Funzioni rientranti: il comportamento in presenza di un segnale prevede interruzione del flusso di istruzioni corrente, esecuzone del signal handler,ritorno al flusso standard alla terminazione sel signal handler. Quindi il kernel sa dove riprendere il flusso di istruizioni precente ma il signal handler non sa dove esso è stato interroto.

Una **race conditions** avviene quando il comportamento di più processi che lavorano su dati comuni dipende dall'ordine di esecuzione.Utilizzo di segnali tra processi può generare race conditions che pornano il programma a non funzionare nel modo desiderato.

ESEMPIO LANCIARE 2 SEGNALI
```
void manager (int sig) {
printf ("Ricevuto il segnale %d\n", sig);
// (void) signal (SIGINT, manager);return;}

int main() {
(void) signal (SIGINT, manager);
while (1) {
printf ("main: Hello!\n");sleep (1);}}
void manager (int sig) {
if (sig==SIGUSR1)
printf ("Ricevuto SIGUSR1\n");
else if (sig==SIGUSR2)
printf ("Ricevuto SIGUSR2\n");
else printf ("Ricevuto %d\n", sig);
return;}
int main () {
...(void) signal (SIGUSR1, manager);
(void) signal (SIGUSR2, manager);
...}
```

Le Pipe

I processi concorrenti possono essere
1)indipendente se :
Non può essere infuenzato dagli altri processi in esecuzione nel sistema ¬ **Non** può influenzare l'esecuzione di altri processi
2)cooperante se ¬ Può essere influenzato dagli altri processi inesecuzione ¬ Può influenzare l'esecuzione di altri processi

Ogni processo che condivide dati con altri processi è un processo cooperante .
Vantaggi della cooperazione tra processi:
Condivisione delle informazioni, Incremento della velocità di calcolo,Modularità

Comunicazione tra processi
ϖ Processi cooperanti necessitano di meccanismi che consentano la condivisione di dati

ϖ Si parla di **IPC I**nter**P**rocess **C**ommunication
¬ Modello a memoria condivisa(Una regione della memoria è condivisa, I dati da trasferire sono scritti e letti in tale memoria)
¬ Modello basato sul passaggio di messaggi(La comunicazione avviene mediante lo scambio di messaggi)

Esempio

```
#include <unistd.h>#include <stdlib.h>#include
<stdio.h>#include <string.h>
int main() {
int n;int file[2];char cR = 'X';char cW;pid_t pid;
if (pipe(file) == 0) {
pid = fork ();
if (pid == -1) { fprintf(stderr, "Fork failure");
exit(EXIT_FAILURE);}

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

else {// parent writes
close (file[0]);
n = write (file[1], &cW, 1);
printf ("Wrote %d bytes: %c\n", n, cW);}
}
exit(EXIT_SUCCESS);}
```

Una pipe è un **flusso dati** tra due processi ovvero Il flusso di dati in una pipe è **half-duplex**

Le pipe possono essere utilizzate per la comunicazione tra processi con un **parente comune**

ϖUna volta creata una pipe ciascun processo accede a uno degli estremi della pipe attraverso un file descriptor.
ϖ Dato che i file descriptor devono essere comuni ai due processi comunicanti tali processi devono avere un antenato comunem call pipe ( )

#include <unistd.h>
int pipe (int fileDescr[2]);
ϖ La funzione ritorna due descrittori di file
ϖ Il vettore **fileDescr** contiene i due nuovi descrittori, tali che:
¬ fileDescr[0]: Aperto per la lettura dalla pipe
¬ fileDesrc[1]: Aperto per la scrittura sulla pipe
¬ L'output su fileDesc[1] corrisponde all'input su fileDescr[0]
ϖ Valore di ritorno – 0 ok -1 errore

ϖ Lettura e scrittura da e su pipe vengono effettuate mediante **read** e **write**
¬ Il descrittore della pipe è un intero

¬ La system **call read**
● Ritorna solo i caratteri disponibili, se la pipe contiene meno caratteri di quanto richiesto
● Si blocca se la pipe è vuota (**è bloccante**)
● Ritorna 0 se la pipe è stata chiusa all'altra estremità

¬ System **call write**
● Si blocca se la pipe è piena (bloccante)
● Ritorna SIGPIPE se l'altra estremità è stata chiusa

SHELL

Fornisce l'interfaccia utente, ovvero interpreta icomandi degli utenti passandoli al kernel
Diverse variabili sono riservate

- Quando una shell viene eseguita alcune variabili sono inizializzate a partire da valori dell'"environment"
- In genere tali variabili sono definite con lettere maiuscole per distinguerle da quelle utente
- Si visualizzano con il comando **printenv**

Variabili d'ambiente

- Un elenco parziale include le seguenti
  - $SHELL● Indica la shell in uso corrente
  - $LOGNAME● Indica lo username utilizzato per il login
  - $HOME● Indica la home directory dell'utente corrente
  - $PATH● Memorizza l'elenco dei direttori separati da ':' utilizzato per la ricerca dei comandi (eseguibili)
- La shell conserva un elenco di alias, che sono quindi locali alla shell utilizzata
- I comandi per manipolare gli alias sono
  - alias¬ unalias
  - alias nome="valore"
- Un alias viene eliminato dalla shell tramite il comando¬ unalias nome

read [opzioni] var_1 var_2 ... var_n
La funzione **read** permette di eseguire dell'input interattivo, in altre parole permette di leggere una riga da stdin
Il comportamento è il seguente¬ Ogni parola (separata da spazi) viene messa inuna variabile¬ Nel caso ci siano più stringhe in ingresso che variabili l'ultima variabile contiene tutte le rimanenti parole¬ Nel caso ci siano più variabili di stringhe le variabili in eccesso non vengono assegnate¬ Nel caso non siano specificate variabili si utilizza lavariabile di default REPLY

- Parametri posizionali
  - $0 indica il nome dello script● $1, $2, $3, ... indicano I parametri passati allo scriptsulla riga di comando
- Parametri speciali● $* indica l'intera lista (stringa) dei parametri
  - $# memorizza il numero di parametri ● $$ memorizza il PID del processo

Il "quoting" serve per¬ Disabilitare il significato speciale di caratteri estringhe ¬ Prevenire il riconoscimento di parole riservate tre tecniche di quoting

Gli apici ' '¬ Identificano una stringa al cui interno **non** sono espanse le variabili
virgolette " "¬ Identificano una stringa al cui interno **sono** espanse le variabil
Il backslash \¬ Indentifica il carattere di escape

Comandi principali

- ⇑, ⇓, history● Mostrano i comandi eseguiti precedentemente
- !n● Esegue il comando numero n nel buffer
- !-n● Esegue l'n-ultimo comando
- !stringa● Esegue l'ultimo comando che inizia con stringa
- ^stringa1^stringa2● Sostituisce nell'ultimo comando la stringa1 con la stringa2 (anche !!:s/vecchio/nuovo)
- !$● Indica l'ultimo parametro del comando precedente
- !*● Indica tutti i parametri del comando precedente
- ctrl-r com● Filtra la history mostrando solo i comandi che contengono la stringa com

Variabili d'ambiente

- $PS1● Contiene il prompt comandi (di solito '$')
- $PS2● Specifica un prompt ausiliario, si solito utilizzato per ricevere altro input (di solito '>')
- $IFS● Elenca i caratteri utilizzati per separe le stringhe lette da input (vedere comando read della shell)

```
#!/bin/bash
# String length
echo "Type a word: "
read word
# echoing without newline | word count chars
l=$(echo -n $word | wc -c)
echo "Word $word is $l characters long"
```

Confronto per file

- La condizione è vera (uguale a 0) se
  - -d l'argomento è una directory● -f l'argomento è un file regolare
  - -e l'argomento esiste (talvolta non portabile,usare –f)
  - -r l'argomento ha il permesso di lettura● -w l'argomento ha il permesso di scrittura● -x l'argomento ha il permesso di esecuzione
  - -s l'argomento ha dimensione non nulla

```
if [ -f foo.c ]
```

Confronto tra stringhe

La condizione è vera (uguale a 0) se

- = le due stringhe sono uguali
- != le due stringhe non sono uguali
- -n string la stringa non è NULL
- -z string la stringa è NULL (empty string)

```
if [ $string = "abc" ]
```

Operatori logici

- All'interno di una condizione
  - ! NOT
  - -a AND
  - -o OR
- In un elenco di condizioni
  - && AND
  - || OR

La condizione è vera (uguale a 0) se

- -eq i due numeri sono uguali
- -ne i due numeri non sono uguali
- -gt il primo numero è maggiore
- -ge il primo numero è maggiore o uguale
- -lt il primo numero è minore
- -le il primo numero è minore o uguale
- ! negazione (!expt ◊ not(expr))

Thread è una sezione di un processo che viene schedulata e eseguita indipendentemente dal processo (thread) che l'ha generata
¬ Un thread viene a essere l'unità elementare diutilizzo della CPU e agevola la risposta delprocesso a eventi asincroni
¬ Condivide con gli altri thread che appartengono allo stesso processo● Sezione di codice● Sezione di dati

¬ Possiede dei dati privati
● Program counter ● Insieme di registri● Stack
¬ Ovvero, i dati privati di un thread sono costituiti dai registri e dalle variabili locali (nello stack) e globali (nell'heap) **non** condivise dal thread

VANTAGGI

¬ Avere tempi di risposta ridotti¬ Condividere le risorse
¬ Economizzare le risorse¬ Maggiore scalabilità

Svantaggi

¬ Non esiste protezione tra i thread di uno stesso processo in quanto tutti sono eseguiti nello stesso spazio degli indirizzi
¬ Se i thread **non** sono sincronizzati l'accesso a dati comuni non è **thread safe** ϖ Anche se al thread creante viene normalmente restituito l'identificatore del thread creato i thread in genere **non** hanno relazione gerarchica padrefiglio1

Modelli One-to-One

Per ogni thread a livello utente ne esiste uno a livello kernel.La gestione dei thread:Permette un effettivo parallelismo ¬ La creazione di un thread utente richiede la creazione di un thread a livello kernel(Minori prestazioni,L'utente deve controllare il numero di thread generati)

Modelli Many-to-One

I Thread sono gestiti a livello utnte e nel kernel esiste un solo thread.La gestione di thread risulta efficente in quanto viene effettuata a livello utente, permette al programmatore di generare i thread desiderati, no realizza parallelismo vero in quanto lo scheduler deve mappare i thread utente sull'unici thread kernel.Se il thread si blocca tutti i thread utente si bloccano.

Modelli Many-to-Many

Gli n thread a livello utente corrispondono a m thread a livello kernel (m<n)
La gestione dei thread¬ Risulta la più efficiente in quanto da un lato l'utenteè libero di creare tutti i thread che desidera dall'altro si può avere effettivo parallelismo a livello di thread kernel

Modelli di programmazione 3

1)User thread

IIl kernel non è a conoscenza dell'esistenza dei thread e gestisce solo processi standard
ϖ Vantaggi
¬ Context switch veloce tra threads di uno stesso task
¬ Si possono implementare sopra un kernel qualsiasi
ϖ Svantaggi
¬ Se un thread effettua una system call bloccante, sono bloccati tutti i threads dello stesso processo
¬ Esiste solo un thread in run per task anche in un sistema multiprocessore

2)Kernel thread

I thread sono gestiti dal kernel
ϖ Le informazioni sui thread in esecuzione
¬ Sono le stesse mantenute nel caso della gestione di thread utente
¬ Vengono gestite dal kernel
Vantaggi
¬ I thread ready possono essere schedulati anche se appartengono allo stesso task di un thread che ha chiamato una system call bloccante
¬ In un sistema multiprocessore si possono eseguire thread multipli per task
ϖ Svantaggi
¬ Context switch più costoso perché richiede il passaggio al modo kernel
¬ Limitazione nel numero massimo di thread per task/sistema

3)Implementazione ibrida

ϖ L'implementazione mista tenta di cobinare i vantaggi di entrambi gli approcci
¬ L'utente decide quanti thread utente eseguire e su quanti thread kernel mapparli
ϖ Il kernel è a conoscenza solo dei thread kernel e gestisce solo tali thread
ϖ Ogni thread kernel può essere utilizzato a turno da diversi thread utente

Un thread è una funzione (procedura) del programma che viene eseguita in maniera indipendente dal resto del programma stesso
¬ Un processo concorrente costituito da più thread può essere visto come un insieme di funzioni (procedure) in esecuzione indipendente che condividono le risorse del processo

int pthread_create (pthread_t *tid,const pthread_attr_t *attr,void *(*startRoutine)(void *),void *arg);
genera un nuovo thread
int pthread_equal (pthread_t tid1,pthread_t tid2,);
Confronta due identificatori di thread, 0 se uguali

LIFO-STACK
i
```
void push (int val) {
if(top<0)return;
stack[top] = val;
top--;
return;
}
```
J
```
int pop (int *val) {
if(top>SIZE-2)return;
top++;
*val=stack[top];
return;
}
```
ϖ Le funzioni **push** e **pop** agiscono sulla stessa estremità dello stack (vettore)
ϖ La variabile **top** è condivisa da push e pop

FIFO – Queue – Buffer Circolare
i
```
void enqueue(int val)
{if(n>SIZE) return;
queue[tail] = val;
tail=(tail+1)%SIZE;
n++;
return;}
```
j
```
int dequeue(int *val)
{if(n<=0) return;
*val=queue[head];
head=(head+1)%SIZE;
n--;
return;}
```
ϖ Le funzioni enqueue e dequeue agiscono su estremità "diverse" della coda (vettore) usando variabili tail e head
ϖ La variabile n è comunque condivisa

Sezioni critiche
ϖ Una sezione critica (**SC**) è una sezione di codice, comune a N processi, nella quale
¬ I processi possono accedere (in lettura e scrittura) a variabili comuni
ϖ Ovvero nella quale
¬ I processi competono (in lettura e scrittura) per l'uso di una risorsa comune e/o di dati condivisi
ϖ Problema
¬ Assicurarsi che le condizioni di **Bernstein** siano verificate
ϖ Ovvero che ¬ Quando un processo sta eseguendo il codice nella sua SC, nessun altro processo può fare altrettanto
¬ Il codice nella SC deve essere eseguito da un singolo processo (thread) alla volta ¬ L'esecuzione del codice nella SC deve essere effettuato in **mutua esclusione**
Definizione del problema Protocollo di accesso
ϖ Soluzione ¬ Occorre stabilire un protocollo di accesso in mutua esclusione alla SC
ϖ Ovvero
¬ Per entrare nella propria SC un processo esegue un codice di prenotazione dell'accesso che lo blocca finché la SC è utilizzata da un altro processo
¬ All'uscita dalla SC, un processo esegue un codice per il rilascio della regione occupata

Condizioni da soddisfare
Ogni soluzione al problema delle SC **deve** soddisfare
1) **Mutua esclusione** • Un solo processo alla volta deve ottenere l'accesso
2) **Progresso** • Se nessun processo si trova nella SC solo i processi in fase di prenotazione possono partecipare alla selezione per chi accederà in futuro e tale selezione deve durate un tempo definito • Evitare i **deadlock**
3) Attesa definita
• Deve esistere un numero definito di volte per cui altri processi riescano ad accedere alla SC prima che un processo che ha fatto richiesta possa farlo
• Evitare la **starvation** di un processo

4)Ogni soluzione dovrebbe inoltre essere simmetrica
¬ Non dipendere dalla priorità relativa tra i processi per la determinazione di chi accede alla risorsa
¬ Essere indipendente dalla velocità relativa dei processi

SOLUZIONE SOFTWARE

ϖ Il problema della SC per via software può essere risolto mediante l'utilizzo di variabili condivise
ϖ Analizzeremo il caso con due soli processi
ϖ I due processi verranno denominatiovvero P_i e P_j
¬ Quindi se **i=1** allora **j=0** e viceversa
¬ Quindi dato **i** allora **j=1-i** e viceversa

Soluzione 1

```
while (TRUE) {
while (turn==j);
SC di Pi
turn = j;
sezione non critica
}
```

Soluzione 1

```
while (TRUE) {
while (turn==i);
SC di Pj
turn = i;
sezione non critica
}
```

Progresso non assicurato: P_i e P_j devono entrarenella SC ma in maniera alternata

ϖ La soluzione 2 prevede l'utilizzo di una variabile che specifica di chi è il "turno"
ϖ L'azione di controllare continuamente **turn** viene detto **busy waiting**
ϖ Un lock che utilizza il busy waiting viene denominato **spin lock**
flag vettore inizializzato a false

Soluzione 2

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

Soluzione 2

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

Mutua esclusione non assicurata: P_i e P_j possono entrare entrambi nella SC

La soluzione 3 tenta di risolvere il problema della soluzione 2 con un approccio simmetrico al test di un flag e al set del turno
ϖ Oltre a presentare deadlock ha gli stessi svantaggi della soluzione 2

Soluzione 3

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

Soluzione 3

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

Attesa non definita: P_i e P_j possono rimanere bloccati (deadlock)

Conclusioni:
In generale le soluzioni software al problema delle SC risultano complicate e inefficienti
¬ Un processo può assegnare un valore a una variabile o controllarlo, ma questa operazione è "invisibile" agli altri processi
¬ Le operazioni di controllo e modifica non sono "atomiche", quindi si possono avere reazioni al valore presunto di una variabile invece che a quello reale
ϖ In generale le soluzioni software sono estendibili a N processi con difficoltà

Soluzione 4

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

Soluzione 4

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

Soluzione corretta, rispetta tutte le condizionirelative alle SC

SOLUZIONE HARDWARE

Utilizzo di
- **"Lock"**, i.e., lucchetti di protezione
- Istruzione "**atomiche**"

Tra le soluzioni hw è però possibile differenziare i kernel che
- Non permettono il diritto di **prelazione**
- Permettono il diritto di **prelazione**

Sistemi senza diritto di prelazione
- Un processo in esecuzione nel kernel non può essere interrotto
- Il controllo del kernel verrà rilasciato solo quando il processo lo lascerà volontariamente

Sistemi con diritto di prelazione

In un sistema con diritto di prelazione
- Un processo in esecuzione in modalità di sistema può essere interrotto
- Di fatto l'arrivo di un **interrupt** sposta il controllo del flusso su un altro processo
- Il processo originario verrà terminato in seguito

è possibile risolvere il problema della SC mediante abilitazione/disabilitazione dell'interrupt
- Disabilitare l'interrupt all'entrata della SC
- Abilitare l'interrupt all'uscita della SC

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

- Il problema delle sezioni critiche nei sistemi con diritto di prelazione è applicabile solo
  - A livello dei soli processi kernel
  - Su sistemi mono-processori
  - Su sistemi non real-time

Meccanismi di lock – unlock

utilizzare meccanismi di: Lock e Istruzioni atomiche
- In generale vengono implemetate due istruzioni atomiche di lock
  - Test-And-Set su una variabile di lock
  - Swap di due variabili, di cui una di lock
- Ogni SC è protetta dal relativo **lock**
  - Il **lock** viene acquisito per entrare nella SC
  - Il **lock** viene rilasciato all'uscita della SC

```
while (TRUE) {
acquisisci il lock (lock)
SC
rilascia il lock (unlock)
sezione non critica
}
```

Test-And-Set
- Effettua un test seguito da un se di una variabile di lock globale (di tipo char, ovvero di 1 singolo byte)
  - Se il byte è FALSE la SC è libera
  - Se il byte è TRUE la SC è occupata
  - In ogni caso al byte viene assegnato valore TRUE

```
char TestAndSet (char *lock) {
char val;
val = *lock;
*lock = TRUE; // Set new lock to TRUE
return val; // Return old lock
}
```

Mutua esclusione con Test-And-Set

```
char TestAndSet (char *lock) {
char val;
val = *lock;
*lock = TRUE; // Set new lock
return val; // Return old lock
}


char lock = FALSE; // GLOBAL lock
...
while (TRUE) {
while (TestAndSet (lock)); // lock
SC
lock = FALSE; // unlock
sezione non critica
}
```

Swap
- Effettua uno scambio tra due parole di memoria
  - Utilizza nuovamente una variabile globale di lock di tipo char (1 singolo byte)
  - La variabile lock si inizializza al valore FALSE

```
void swap (char *v1, char *v2) {
char = *tmp;
*tmp = *v1;
*v1 = *v2;
*v2 = *tmp;
return;
}
```

Mutua esclusione con Swap

```
void swap (char *v1, char *v2) {
char = *tmp;
*tmp = *v1;
*v1 = *v2;
*v2 = *tmp;
return;
}
char lock = FALSE; // Global
lock
...
while (TRUE) {
key = TRUE;
while (key==TRUE)
swap (&lock, &key); // Lock
SC
lock = FALSE; // Unlock
sezione non critica
}
```

Mutua esclusione senza starvation

Non fa attendere troppo un processo per evitare che "Muore"

```
while (TRUE) {
waiting[i] = TRUE;
key = TRUE;
while (waiting[i] && key)
key = TestAndSet (lock);
waiting[i] = FALSE;
```

- Vantaggi delle soluzioni hardware
  - Utilizzabili in ambienti multi-processore
  - Facilmente estendibili a N processi
  - Semplici da utilizzare dal punto di vista software,cioè dal punto di vista utente
- Svantaggi delle soluzioni hardware
  - Non facili da implementare a livello hardware
  - Busy Waiting su spin-lock

```
SC di Pi
j = (i+1) % N;
while ((j!=i) and (waiting[j]==FALSE))
j = (j+1) % N;
if (j==i) lock = FALSE;
else waiting[j] = FALSE;
sezione non critica
}
```

- Starvation
- Deadlock

SEMAFORI

Un semaforo S è

- Una variabile intera condivisa
- Protetta dal sistema operativo
- Che può essere utilizzata per inviare e ricevere segnali

Le operazioni su **S s**ono sempre eseguite in maniera atomica

- init (S, k)
  - Inizializza il semaforo S
  - Esistono due tipi di semafori
    - Semafori binari
      - Il valore di inizializzazione k deve essere uguale a 0oppure a 1
      - Il valore del semaforo in un istante qualsiasi è ugualea 0 oppure a 1
      - Anche noti come "mutex lock" (mutex = mutualexclusion)
    - Semafori con conteggio
      - Il valore di inizializzazione k è un intero qualsiasi
      - Il valore del semaforo in un istante qualsiasi ricadenell'intervallo [0, k]

- wait (S)
  - Decrementa la variabile semaforica e **blocca** il processo chiamante se il suo valore è negativo o nullo
  - Attende se la risorsa non è disponibile
  - Originariamente era denominata P() dall'olandese "probeer te verlagen", i.e., "try to decrease"

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

- signal (S)
  - Incrementa la variabile semaforica e **sblocca** larisorsa
  - Originariamente denominata V(), dall'olandese"verhogen", i.e., "to increment"
  - Da **non** confondere con la system call signal utilizzata per instanziare un gestore di segnali

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

- destroy (S)
  - Rilascia la memoria occupata dal semaforo S
    - Le implementazioni reali di un semaforo richiedono molto di più di una semplice variabile globale per definire un semaforo
  - Presente nelle implementazioni reali, non viene utilizzata negli esempi/esercizi successivi

Mutua escusione con un semaforo

```
init (S, 1);
while (TRUE) {
wait (S);
SC di Pi
signal (S);
sezione non critica
}
while (TRUE) {
wait (S);
SC di Pj
signal (S);
sezione non critica
}
```

I semafori vanno implementati

- Senza ricorrere all'attesa attiva "busy waiting" su"spin-lock"
- Definiamo un semaforo con
  - Un contatore
  - Una lista di processi

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

init (semaphore_t S, int k) {
S.cnt = k; // k>=0
}

wait (semaphore_t S) {
S.cnt--; // May be <0
if (S.cnt<0) {
push P to S.head
block P}}

signal (semaphore_t S) {
S.cnt++;
if (S.cnt<=0) {
pop P from S.head
wakeup P
}
}

destroy (semaphore_t S) {
while (S.cnt>=0) {
free S.head
S.cnt--;
}
}
```

SEMAFORI PIPE

- Il semaforo è una variabile di tipo contatore
- L'operazione **wait** • Decrementa il valore del contatore
  • È bloccante se il contatore è uguale a 0
- L'operazione **signal** • Incrementa il valore del contatore
- Un tale comportamento può essere realizzato mediante una pipe tra un processo padre e un processo figlio
- Data una pipe
- Il contatore di un semaforo è realizzato tramite il concetto di **token**
- La **signal** è effettuata con un'operazione di **write** di un token sulla pipe (non bloccante)
- La **wait** è effettuata con un'operazione di **read** di un token dalla pipe (bloccante)

ESEMPIO

```
int main() {
int S[2];
pid_t pid;
semaphoreInit (S);
pid = fork();
// Check for correctness
if (pid == 0) { // child
semaphoreWait (S);
printf("Wait done.\n");
} else { // parent
printf("Sleep 3s.\n");
sleep (3);
semaphoreSignal (S);
printf("Signal done.\n");
}
return 0;
}
```

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

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

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

- Legge un carattere dalla pipe (read bloccante)

```
int sem_init (
sem_t *sem,
int pshared,
unsigned int value
);
```

- Inizializza il semaforo al valore value
- Il valore di **pshared** identifica il tipo del semaforo
- Se uguale a 0, allora il semaforo è **locale** al processo corrente
- Altrimenti, il semaforo può essere **condiviso** tra diversi processi (padre che inizializza e figli creati di seguito)

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

Operazione di wait standard

- Se il semaforo è uguale a 0,blocca il chiamante sino a quando può decrementare il valore del semaforo

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

- Operazione di wait senza blocco (non-blocking wait)
- Se il semaforo ha un valore diverso da 0 (>0), lo decrementa e ritorna 0
- Se il semaforo è uguale a 0, ritorna -1 (invece di bloccare il chiamante coma la wait)

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

- Classica operazione signal
- Incrementa il valore del semaforo

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

- Distrugge un semaforo creato precendetemente
- Può ritornare -1 se si cerca di distruggere un semaforo utilizzato da un altro processo

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

Permette di esaminare il valore di un semaforo

- Il valore del semaforo viene assegnato a *valP (i.e., valP è il puntatore all'intero che indica il valore del semaforo dopo la chiamata)
- Se ci sono processi in attesa, a *valP si assegna 0o un numero negativo il cui valore assoluto è uguale al numero diprocessi in attesa

ESEMPIO:

```
...
#include "semaphore.h"
...
sem_t *sem;
...
sem = (sem_t *) malloc(sizeof(sem_t));
sem_init (sem, 0, 0);
...
... create sub processes or threads ...
...
sem_wait (sem);
... SC ...
sem_post (sem);
```

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

- Crea un nuovo lock (mutex variable) mutex
- Restituisce mutex al chiamante
- attr specifica gli attributi di mutex (default=NULL)
- Valore di ritorno
- Se l'operazione ha avuto successo 0
- Il codice di errore altrimenti

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

- Controlla il valore del lock mutex e
- Blocca il chiamante se è già locked
- Acquisisce il lock se non è locked
- Valore di ritorno
- Se l'operazione ha avuto successo 0
- Il codice di errore altrimenti

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

- Simile a pthread_mutex_lock ma nel caso il lock sia già stato acquisito ritorna senza bloccare il chiamante
- Valore di ritorno
- Se il lock è stato acquisito 0
- Se il mutex era posseduto da un altro thread, codice di errore EBUSY

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

- Rilascia il lock mutex al termine della SC
- Valore di ritorno
- Se l'operazione ha avuto successo 0
- Il codice di errore altrimenti

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

- Rilascia la memoria occupata dal lock mutex
- Tale lock non sarà non più utilizzabile
- Valore di ritorno
- Se l'operazione ha avuto successo 0
- Il codice di errore altrimenti

ERRORI timedependent

¬ Questi errori si possono presentare anche nel caso in cui un processo si comporti correttamente a causa di un altro processo

¬ Questi errori dipendono dalla cooperazione dei diversi processi e sono difficili da debuggare

REGIONE CRITICA

Per eliminare errori time-dependent

ϖ La definizione richiede

¬ La dichiarazione di una variabile condivisa v di tipoT

¬ L'utilizzo del costrutto **region** per accedere in mutua esclusione a tale variabile all'interno di una sezione critica SC

Le regioni critiche sono gestite dal compilatore per generare il codice corrispondente utilizzando dei semafori

Regioni critiche condizionali

ϖ Una regione critica condizionale ritarda eventualmente un processo solo all'entrata della sua SC

ϖ Non si può utilizzare nel caso in cui la sincronizzazione sia in punti interni alla SC

```
region v do
begin
S1;
await (B);
S2;
end;
```

Produttore-Consumatore

Produttore e consumatore con memoria limitata

- Utilizza un buffer circolare di dimensione **SIZE** per memorizzare gli elementi prodotti e da consumare
- Il buffer circolare implementa una coda FIFO

- La soluzione è simmetrica
  - Il produttore produce posizioni piene
  - Il consumatore produce posizioni vuote
- Può essere facimente **estesa** al caso in cui coesistano più produttori e più consumatori
  - Produttori e consumatori operano su estremità opposte del buffer e possono farlo **contemporaneamente**
  - Due produttori oppure due consumatori devono invece agire in **mutua esclusione**

Accesso concorrente

```
init (full, 0);
init (empty, MAX);

Producer () {
Message m;
while (TRUE) {
produce (m);
wait (empty);
enqueue (m);
signal (full);
}
}
Consumer () {
Message m;
while (TRUE) {
wait (full);
m = dequeue ();
signal (empty);
consume (m);
}
}
```

```
dequeue (message m) {
m = buffer[head];
head = (head+1)%SIZE;
return;
}
enqueue (message m) {
queue[tail] = m;
tail = (tail+1)%SIZE;
return;
}
#define SIZE ...
...
message buffer[SIZE];
int tail, head;
...
init () {
tail = 0;
head = 0;
}
```

Accesso concorrenti di più P & C

```
init (full, 0);
init (empty, MAX);
init (MEp, 1);
init (MEc, 1);

Consumer () {
Message m;
while (TRUE) {
wait (full);
wait (MEc);
m = dequeue ();
signal (MEc);
signal (empty);
consuma m;
}
}
Producer () {
Message m;
while (TRUE) {
produce m;
wait (empty);
wait (MEp);
enqueue (m);
signal (MEp);
signal (full);
}
}
```

Accesso concorrente con RC

```
init (full, 0);
init (empty, MAX);
shared T p, c;

Consumer () {
Message m;
while (TRUE) {
wait (full);
region c do
m = dequeue ();
signal (empty);
consuma m;
}
}
Producer () {
Message m;
while (TRUE) {
produce m;
wait (empty);
region p do
enqueue (m);
signal (full);
}
}
```

Accesso concorrente con RC Condizionali

```
init (full, 0);
init (empty, MAX);
shared T p, c, count;
Consumer () {
Message m;
while (TRUE) {
region count
when count>0 do {
region c do
m = dequeue ();
count--;
}
consuma m;
}
}
Producer () {
Message m;
while (TRUE) {
produce m;
region count
when count<MAX do {
region p do
enqueue (m);
count++;
}
}
}
```

Semaforo empty: conteggia il numero di elementi vuoti e blocca il produttore nel caso il buffer sia pieno.
Semaforo full: conteggia il numero di elementi pieni e bloccka il consumantore nel caso il buffer sia vuoto
il semaoro meC: forza la mutua esclusione tra i diversi consumatori
il semaoro meP: forza la mutua esclusione tra i diversi Produttori

Readers & Writers

¬ Condividere un base dati tra due insiemi di processi concorrenti
• Una classe di processi detta **Reader** a cui è consentito accedere a un data-base in **concorrenza**
• Una classe di processi detta **Writer** a cui è consentito accedere al data-base in **mutua esclusione** sia con altri processi Writers sia con i processi Readers
ϖ Esistono due versioni del problema
¬ Primo problema o problema con precedenza ai reader
¬ Secondo problema o problema con precedenza ai writer
ϖ Obiettivi comuni¬ Rispettare il protocollo di precedenza ¬ Rispettare le condizioni di Bernstein ¬ Massimizzare la concorrenza

Precedenza ai reader

ϖ Dare precedenza ai reader significa
¬ Privilegiare l'accesso dei reader rispetto a quello dei writer ovvero
¬ I reader non devono attendere a meno che un writer sia nella SC
ϖ Protocollo di accesso
¬ I reader possono accedere in concorrenza al database
¬ Sino a quando arrivano reader i writer attendono
¬ Quando anche l'ultimo reader termina allora si può svegliare un writer (o un reader ... dipende dallo scheduler)

```
nR = 0;
init (meR, 1);
init (w, 1);

wait (meR);
nR++;
if (nR==1)
wait (w);
signal (meR);
...
lettura
...
wait (meR);
nR--;
if (nR==0)
signal (w);
signal (meR);


wait (w);
...
scrittura
...
signal (w);
```

La soluzione utilizza
¬ Una variabile globale (nR) che conta il numero di reader nella SC
¬ Un semaforo di mutua esclusione per la manipolazione della variabile nR (meR)
¬ Un semaforo di mutua eslusione per più writer o per un reader e i writer (w)
ϖ I writer sono soggetti a **starvation**, in quanto possono attendere per sempre
ϖ Sono possibili soluzioni più complesse senza starvation

Precedenza ai writer

ϖ **Dare precedenza ai writer significa**
¬ Un writer pronto deve attendere il meno possibile
ϖ Protocollo di accesso
¬ Ogni writer deve attendere che finiscano i reader
¬ Ogni writer ha priorità su tutti i reader

```
nR = nW = 0;
init (w, 1); init (r, 1);
init (meR, 1); init (meW, 1);

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);
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);
```

ϖ La soluzione utilizza
¬ Due variabili globali per il conteggio dei reader e dei writer
¬ Due semafori di mutua esclusione (meR e meW) per la manipolazione delle variabili nR e nW
¬ Due semafori di mutua eslusione reader/writer
ϖ I reader sono soggetti a starvation, in quanto possono attendere per sempre
ϖ Sono possibili soluzioni più complesse senza starvation

I 5 filosofi

ϖ Modello del caso in cui diverse risorse sono comuni a diversi processi concorrenti
¬ Un tavolo è imbandito con • 5 piatti di di riso
• 5 bastoncini (cinesi) ciascuno tra due piatti
¬ Intorno al tavolo siedono 5 filosofi
¬ I filosofi pensano oppure mangiano
• Per mangiare ogni filosofo ha bisogno di due bastoncini • I bastoncini possono essere ottenuti uno alla volta

Modello 1

Utilizzare un unico semaforo binario (mutex) per proteggere l'unica risorsa "cibo"
¬ Annulla la concorrenza
¬ Un solo filosofo mangia (potrebbero farlo in due)
init (mutex, 1);

while (true) {
Pensa ();
wait (mutex);
Mangia ();
signal (mutex);
}

Modello 2

Avere un semaforo per bastoncino

- Può causare deadlock

```
init (chopstick[0], 1);
...
init (chopstick[4], 1);

while (true) {
Pensa ();
wait (chopstick[i]);
wait (chopstick[(i+1)mod5]);
Mangia ();
signal (chopstick[i]);
signal (chopstick[(i+1)mod5]);
}
```

Soluzione

Struttura dati

- Uno stato per ogni fisoloso (THINKING, HUNGRY,EATING)
- Un semaforo per ogni filosofo (per l'accesso a cibo)
- Un semaforo ulteriore unico per l'accesso alla variabile di stato del filosofo stesso

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

```
takeForks (int i) {
wait (mutex);
state[i] = HUNGRY;
test (i);
signal (mutex);
wait (sem[i]);
}
int state[N]
init (mutex, 1);
init (sem[0], 0); ...; init (sem[4], 0);
test (int i) {
if (state[i]==HUNGRY && state[LEFT]!=EATING &&
state[RIGHT]!=EATING) {
state[i] = EATING;
signal (sem[i]);
}
}
putForks (int i) {
wait (mutex);
state[i] = THINKING;
test (LEFT);
test (RIGHT);
signal (mutex);
```

Il "Tunnel a senso alternato"

In un tunnel a senso alternato

- Estensione del problema dei Readers-Writers con due insiemi di reader
- Struttura dati
- Due variabili globali di conteggio (n1 e n2), una per ciascun senso di marcia
- Due semafori (s1 e s2), uno per ciascun senso di marcia
- Un semaforo globale di attesa (busy)
- Nella sua implementazione base può provocare starvation delle auto 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);

rigth2left
wait (s2);
n2++;
if (n2 == 1)
wait (busy);
signal (s2);
...
Run (left to right)
...
wait (s2);
n2--;
if (n2 == 0)
signal (busy);
signal (s2);
```

SED

Le espressioni regolari

- c simbolo c
- * da 0 a ∞
- \+ da 1 a ∞
- \? 0 o 1
- \{n\} n
- \{n1,n2\} da n1 a n2 (inclusi)
- \{n,\} da n a ∞
- \{n,\} da n a ∞
- \(…\) raggruppa quanto tra parentesi
- . qualsiasi carattere
- ^ inizio riga
- $ fine riga
- […] elenco di simboli
- \n new line
- Solo a oppure ab ‘a\?b’
- Una o più a seguite da una o più b ‘a\+b\+’
- Qualsiasi sequenza (non vuota) ‘.*’ OPPURE ‘.\+’
- Una funz il cui nome inizia per myfunc ‘myfunc.*(.*)’
- Una riga che inizia con ABC ‘^ABC’
- Una lettera o una cifra ‘[a-zA-Z0-9]’
- Nove carratteri seguiti da una A ‘.\{9\}A’

Azioni

q Termina l’elaborazione
d Elimina la riga corrente
p Stampa la riga corrente
= Stampa il numero della riga corrente

Inserisce una (o più) riga(he)

- \i Prima della riga corrente
- \a Dopo la riga corrente
- \c Al posto della riga corrente

- Sostituisce la stringa **str1** con la stringa **str2**
  - s/str1/str2/flag

flag può essere g(globale),un intero quale sostituzione effettuare nella riga,se non metto nulla prende solo la prima occorrenza

```
#!/usr/bin/sed -nf
# Conta le righe di un file: ‘wc -l’
$=

#Stampa il num di righe per le righe vuote
/^$/=

#Aggiunge ‘\n’ e 80 spazi alla fine di una riga
# G: active = active + ‘\n’ + auxiliary
G
# Tiene i primi 81 caratteri (80 + newline)
s/^\(.\{81\}\).*$/\1/
# \2 match la meta’ degli spazi
# ... i quali vengono spostati all’inizio
s/^\(.*\)\n\(.*\)\2/\2\1/
```

AWK

```
# Il record deve contenere "foo"
/foo/
# Il field 2 e’ esattamente "foo"
$2 == "foo"
# Il field 1 deve contenere il carattere ‘J’
$1 ~ /J/
# Il record contiene “2400” e “bar”
/2400/ && /bar/
# Il record non deve contenere “xxx”
!~ /xxx/
```

```
# Visualizza tutte le righe di lunghezza
# maggiore di 80 caratteri
awk ‘length($0) > 80‘ in.txt

# Setta il separatore di field (FS) a ‘:’
# Prende il primo field e lo stampa
# ordinandolo alfabeticamente
awk 'BEGIN { FS = ":" }
{ print $1 | "sort" }' /etc/passwd
```

```
#!/usr/bin/awk -f
# Inverte le righe di un file
# (l’ultima diventa la prima e viceversa)
BEGIN {
n = 1
}
{
array[n] = $0
n++
}
END {
for (i=n-1; i>0; i--)
print array[i]
}
```

```
# Visualizza la trasposta di un matrice
{
if (maxNC < NF) maxNC = NF
maxNR = NR
for (i=1; i<=NF; i++) matrix[NR, i] = $i
}
END {
for (c=1; c<=maxNC; c++) {
for (r=1; r<=maxNR; r++)
printf("%s ", matrix[r, c])
printf("\n")
}
}
```

SLIDE 8 STALLO

Deadlock(stallo)

Condizione per avvenire stallo:

- Un processo richiede una risorsa che non è disponibile
- Entra in uno stato di attesa
- L’attesa non termina mai

Consiste quindi in un insieme di processi che attendono tutti il verificarsi di un evento che può essere causato solo da un altro processo dello stesso insieme

Un deadlock implica **starvation** ma non il contrario

- La starvation di processo implica che tale processo attende indefinitamente ma gli altri processi possono procedere.
- Tutti i processi in deadlock sono in starvation

Esempio Stallo

Elaboratore con 2 unità periferiche

- Pi legge da HD e scrive su DVD Pj legge da DVD e scrive su HD

Per gestire la mutua eslusione

- Si utilizzano due semafori hd e dvd, inizializzati a 1
- I semafori risolvono il problema della mutua escusione ma causano **deadlock**

Pi	**Pj**
wait (hd);	**wait (dvd);**
...	...
wait (dvd);	wait (hd);
...	...

Modello(Descrivere condizione di deadlock)

ϖ Processi

¬ I processi sono indistinguibili e in numeroindefinito● P_1, P_2, ..., P_n

¬ Ogni processo utilizza una risorsa facendone accesso mediante protocollo standard.

ϖ Risorse

¬ Le risorse sono suddivise in classi (tipi)● R_1, R_2, . . ., R_m

¬ Ogni risorsa di tipo R_i ha W_i istanze

¬ Tutte le istanze di una classe sono "**indentiche**"

GRAFO G = (V, E)

¬ Grafo G, con insieme di vertici V e insieme di archi E

ϖ V è partizionato in due tipi ¬ P ={P_1, P_2, ..., P_n} processi sistema

¬ R = {R_1, R_2, ..., R_m} Risorse nel sistema

ϖ E è partizionato in due tipi ¬ Archi di richiesta e di assegnazione

Rilevazione di un deadlock

¬ Se il grafo non contiene cicli allora non c'è deadlock

¬ Se il grafo contiene uno o più cicli allora

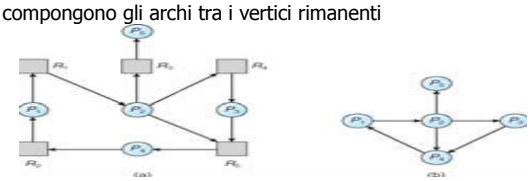
● Se esiste solo un'istanza per tipo di risorsa, c'è sicuramente deadlock

● Se esiste più di un'istanza per tipo di risorsa, c'è lapossibilità di deadlock

GRAFO ATTESA

¬ Si eliminano i vertici di tipo risorsa

¬ Si compongono gli archi tra i vertici rimanenti



Condizioni necessarie per deadlock

Può avvenire **solo se** si verificano 4 condizioni:

1)Mutua esclusione

¬ Deve esserci almeno una risorsa **non condivisibile**, ovvero tale che può essere utilizzata da un processo alla volta

¬ Ulteriore richieste di quella risorsa non possonoessere soddisfatte

2)Possesso e attesa

¬ Un processo che mantiene almeno una risorsa attende di aquisire altre risorse mantenute da altriprocessi

3)Impossibilità di prelazione

¬ Non esiste il diritto di prelazione di una risorsa

¬ Una risorsa può essere rilasciata solo dal processo che la utilizza, dopo che ha completato il suo task

4)Attesa circolare

¬ Esiste un insieme {P_1, ..., P_n} di processi tale che P_1 attende una risorsa tenuta da P_2, P_2 attende una risorsa enuta da P_3, ..., P_{n-1} attende una risorsa tenuta da P_n e P_n attende una risorsa tenuta da P_1

Metodi di gestione

Le condizioni di stallo possono essere gestite con 4 strategie

Strategia dello struzzo ● Si ignora il problema supponendo la probabilità di un deadlock nel sistema sia bassissima

Prevenire ● Controllare le modalità di richiesta per prevenire il verificarsi di uno stato di deadlock

Evitare● Fornire informazioni aggiuntive sulle richieste che verranno effettuate per evitare che il sistema entri in uno stato di deadlock

Ripristinare ● Permettere che il sistema entri in uno stato di deadlock per poi effettuare un'operazione di ripristino (recovery)

Prevenire le situazioni di stallo

Le techinche di **prevenzione** cercano di c**ontrollare** le modalità di richiesta per prevenire il verificarsi di **almeno una** delle condizioni.

Mutua esclusione

Uno stallo si verifica quando un processo rimane indefinitivamente in attesa su una risorsa noncondivisibile

Quindi uno stallo potrebbe essere evitato se

¬ Non esistessero risorse non condivisibili o¬ Non si potesse rimanere in attesa di una risorsa non condivisibile

ϖ Strategia 1 ¬ Proebire risorse non condivisibili

ϖ Strategia 2¬ Proebire l'attesa su risorse non condivisibili

Possesso e attesa

Una condizione di possesso e attesa si verificac quando un processo possiede una o più risorse ene chiede di ulteriori

Quindi una condizione di possesso e attesa potrebbe essere evitata imponendo che un processo chieda una risorsa solo quando non ne possiede altre

Strategia 1¬ **Request All First** (**RAF**)

¬ I processi devono acquisire tutte le risorse di cui necessitano prima di iniziare l'attività di elaborazione ¬ Problema: scarso utilizzo delle risorse **e attesa**

ϖ Strategia 2¬ **Release Before Request** (**RBR**)

¬ Ai processi è consentito richiedere risorse solamente se non ne hanno già acquisite in precedenza

¬ Prima di ogni nuova richiesta ogni processo deve rilasciare le risorse già possedute

¬ Problema: possibilità di starvation

Impossibilità di prelazione

Impossibilità di prelazione significa che una risorsa non può essere sottratta a un processo

ϖ In generale è complesso sottrarre risorse a altriprocessi in esecuzioneϖ Però si potrebbe ottenere un effetto simile

ϖ Strategia 1F

¬ Se un processo, che mantiene alcune risorse, ne chiede un'altra che non può essere allocata immediatamente, allora è costretto a rilasciare tutte le risorse mantenute (preemption)

¬ Le risorse liberate sono aggiunte alla lista delle risorse che il processo attende di acquisire ¬Il processo sarà svegliato solo quando potrà riacquisire tutte le sue vecchie risorse e quelle nuove ke richiede

ϖ Strategia 2

¬ Se la risorsa richiesta non è disponibile si verifica quale processo la possiede ¬ Se il processo che la possiede è a sua volta inattesa si sottrae a tale processo la risorsa richiesta assegnandola al processo che ne ha fatto richiesta¬ In caso contrario il processo viene messo in attesae in futuro potranno essere prelazionate le sue risorse

Attesa circolare

Una condizione di attesa circolare implica che esiste un insieme di processi tale pe cui ogniprocesso dell'insieme attende una risorsa posseduta da un altro processo dell'insieme

Per evitare tale condizione si potrebbe imporre un ordinamento totale tra tutte le classi di risorse

ϖ Strategia

¬ **Hierarchical Resource Usage** (**HRU**)

¬ Impone una relazione di ordinamento totale tra i vari tipi di risorse, associando a ciascuno di essi unnumero intero

¬ Forza ogni processo a richiedere le risorse con un ordine crescente di enumerazione

hru HRU

Sia F la funzione che impone un ordine univoco tra tutte le classi di risorse R_i del sistema

ϖ Un processo P

¬ Abbia precedentemente richiesto una istanza della risorsa R_{old} e faccia richiesta di una istanza di R_{new}

¬ Se $F(R_{new}) > F(R_{old})$ ¬ Se $F(R_{new}) \leq F(R_{old})$

È possibile dimostrare che tale condizione è sufficiente per evitare l'attesa circolare.

In generale la verifica dell'applicazione dell'algoritmo HRU

¬ Può essere lasciata al programmatore

¬ Può essere effettuata dal sistema operativo

Condizione di attesa circolare implica che esiste un insieme di processi tale pe cui ogni processo dell'insieme attende una risorsa posseduta da un altro processo dell'insieme relazione di ordinamento totale tra i vari tipi di risorse, associando a ciascuno di essi un numero intero

Se F(Rnew) > F(Rold)

La risorsa viene concessa

HRU

$F(Rk) < F(Rk+1), \forall\ k = 0 .. n - 1$

cioè

$F(R0) < F(R1) < ... < F(Rn) < F(R0)$

$F(R0) < F(R0)$

che è assurdo

EVITARE DEADLOCK

le tecniche di prevenzione: ¬ Prevedono che i processi forniscano al sistema operativo informazioni aggiuntive a priori
¬ Tali informazioni permetteranno al sistema operativo di schedulare i processi in modo che non si verifichino deadlock
ϖ Gli principali algoritmi di prevenzione
¬ Si differenziano per la quantità e il tipo di informazioni richieste
¬ Causano un ridotto utilizzo delle risorse e una minore efficienza del sistema
¬ Si basano sul concetto di **stato sicuro** e di **sequenza sicura**

STATO SICURO si dice quando
¬ Il sistema è in grado di : Allocare le risorse richieste a tutti i processi in esecuzione e Impedire il verificarsi di uno stallo
¬ Esiste una **sequenza sicura**, ovvero una sequenza di processi {P_1, P_2, …, P_n} tale che per ogni P_i le richieste che esso può ancora effettuare possono essere soddisfatte impiegando le risorse attualmente disponibili più le risorse liberate daiprocessi P_j con j<i

STATO NON SICURO si dice quando
¬ Non necessariamente è uno stato di stallo
¬ Potrebbe condurre in uno stallo

STRATEGIA di prevenzione consiste quindi nell'assicurarsi che il sistema rimanga sempre inuno stato sicuro
¬ All'inizio il sistema è in uno stato sicuro
¬ Ogni richiesta di una risorsa da parte di un processo sarà soddisfatta solo se tale richiesta lascerà il sistema in uno stato sicuro
¬ In caso una richiesta non lasci il sistema in uno stato sicuro il processo che l'ha effettuata verrà posto in attesa

2 STRATEGIE:
¬ Con risorse aventi istanze unitarie
• Applicano algoritmi per la determinazione dei cicli sul grafo di assegnazione delle risorse
¬ Con risorse aventi istanze multiple
• Si occupano di valutare se le risorse disponibili sono sufficienti per portare a termini tutti i processi

Algoritmo per istanze unitarie

Si modificano i grafi di assegnazione delle risorse aggiungendo un nuovo tipo di arco
¬ Arco di **reclamo** (
¬ $P_i \rightarrow R_j$ • Indica che il processo P_j richiederà la risorsa R_j
¬ È rappresentato mediante linea tratteggiata
ϖ Quando un processo
¬ Richiede una risorsa, un arco di reclamo diventa un arco di assegnazione
¬ Rilascia una risorsa, l'arco di assegnazione ritorna a essere un arco di reclamo
ϖ I deadlock si evitano rilevando i cicli nel garfo prima di accettare una nuova richiesta
ϖ La presenza di cicli può essere verificata mediante visita in profondità
ϖ La tecnica è applicabile sono con risorse con una sola istanza per classe

Algoritmo per istanze multiple
algoritmo del banchiere

ϖ Lo stato del sistema è rilevato in base al numero di risorse disponibili, assegnate e massimorichiesto
ϖ Ogni processo
¬ Deve dichiarare a priori il massimo uso di risorse
¬ Quando richiede una risorsa può essere bloccato
¬ Quando ottiene le risorse che gli servono deve garantire che le restituirà in un tempo finito
ϖ Siano dati
¬ Un insieme di processi P_i con cardinalità **n**
¬ Un insieme di risorse R_j con cardinalità **m**
ϖ L'algoritmo del banchiere è costituito da dueparti, per che
¬ Uno stato sia sicuro
¬ Una nuova richiesta possa essere soddisfatta rimanendo in uno stato sicuro

strutture dati

Nome	Dim.	Contenuto e significato
Fine	[n]	Fine[r]=false indica che P_r non ha terminato
Assegnate	[n][m]	Assegnate[r][c]=k P_r possiede k istanze di R_c
Massimo	[n][m]	Massimo[r][c]=k P_r può richieder max k istanze di R_c
Necessità	[n][m]	Necessità[r][c]=k P_r ha bisogn di altre k istanze di R_c ∀i∀j Necessità[i][j]=Massimo[i][j]-Assegnate[i][j]
Disponibili	[m]	Disponibili[c]=k disponibilità pari a k per R_c

Banchiere ALGORITMO

Verifica di una richiesta da parte di P_i
se
∀j Richieste[i][j]≤Necessità[i][j]
AND
∀j Richieste[i][j]≤Disponibili[j]
ALLORA
∀j Disponibili[j]=Disponibili[j]-Richieste[i][j]
∀j Assegnate[i][j]=Assegnate[i][j]+Richieste[i][j]
∀j Necessità[i][j]=Necessità[i][j]-Richieste[i][j]
se lo stato risultante è sicuro si conferma tale assegnazione
altrimenti si ripristina lo stato precedente

1.(verifica di uno stato)
∀i∀j Necessita[i][j]=Massimo[i][j]-Assegnate[i][j]
∀i Fine [i]=false
2.
Trova i per cui
Fine[i]=falso AND ∀j Necessità[i][j]≤Disponibili[j]
Se tale i non esiste goto step 4
3.
∀j Disponibili[j]=Disponibili[j]+Assegnate[i][j]
Fine[i]=true
goto step 2
4.
Se ∀i Fine[i]=true il sistema è in uno stato sicuro

Ripristinare una situazione di stallo

1) Rilevazione (della condizione di stallo) Uno stallo si può rilevare mediante algoritmi simili a quelli analizzati per evitarlo¬ Risorse con istanze singole• Si applica un algoritmo di rilevazione dei cicli sulgrafo delle assegnazioni
¬ Risorse con istanze multiple• Si applica l'algoritmo del banchiere
ϖ Ogni fase di rilevazione ha un costo non indifferente, rivelazionie effettuate quando processo fa una richiesta che non soddisfa immediatamente o a intervalli di tempo fissi.
2)¬ Ripristino (del sistema) Per ripristinare un corretto funzionamento sono possibili diverse strategie
ϖ Strategia 1¬ Terminare tutti i processi in stallo
ϖ Strategia 2¬ Terminare un processo alla volta tra quelli in stallo sino a eliminare la condizione di stallo
ϖ Strategia 3¬ Prelazionare le risorse a un processo alla volta sino a interrompere lo stallo

Conclusioni

ϖ Rilevazione e ripristino sono operazioni
¬ Complesse logicamente
¬ Onerose temporalmente
ϖ In ogni caso se un processo richiede molte risorse è possibile causare starvation
¬ Lo stesso processo viene ripetutamente scelto come vittima e incorre in rollback ripetuti