Introduzione al SO

giovedì 14 marzo 2019 10.14

Concetti base:

Multiutenza
Il sistema presenta ad ogni utente una macchina virtuale completamente dedicata in termini di CPU o altre risorse.

Interattività
Per garantire un'accettabile celocità di reazione alle richieste dei singoli utenti, il SO interrome l'esecuzione di ogni job dopo un intervallo di tempo prefissato, detto quanto di tempo (o **time slice**), assegnando la CPU ad un altro job.

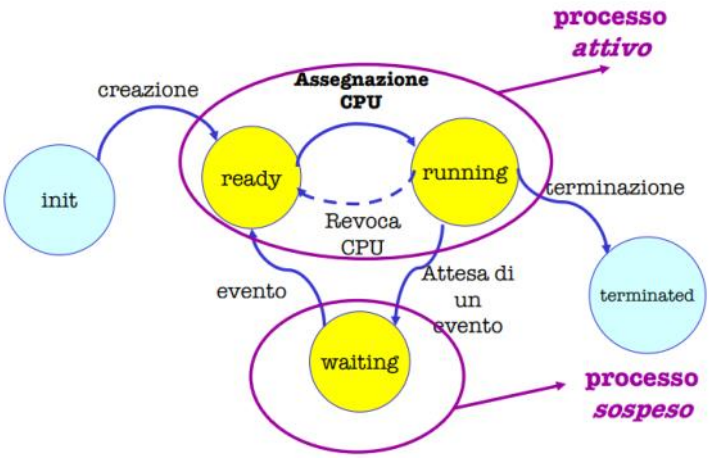
Sistemi multiprogrammati (o time-sharing)
I sistemi time-sharing sono sistemi interattivi, in cui cioè la CPU è dedicata a job diversi che si alternano ciclicamente.
I requisiti per la realizzazione di un sistema time-sharing sono:

- gestione e protezione della memoria
- scheduling CPU
- sincronizzazione/comunicazione tra job
- interazione con file system e accesso on-line

Processo
Programma in esecuzione, unità di esecuzione. Un SO multiprogrammato consente l'esecuzione concorrenziale di processi. Il processo è un'attività attiva (esecutore). Il programma un'attività passiva (descrittore).
Il processo è rappresentato da:

- codice
- dati
- program counter (PC)
- registri
- stack
- risorse di sistema

Stati di un processo:



Process control block (PCB)
Struttura dati (descrittore) per mantenere le informazioni relative ad un processo:

- stato processo
- identificatore processo
- program counter
- contenuto dei registri della CPU
- informazioni di scheduling
- ...

Thread
Sono processi leggeri che hanno un PCB più piccolo ma possono condivider i dati tra loro.

Task
Un insieme di thread che riferiscono lo stesso codice e gli stessi dati.

Introduzione al SO

giovedì 9 maggio 2019 22.22

Strutture del sistema operativo

Struttura monolitica

Il sistema operativo è costituito da un unico modulo eseguibile contenente un insieme di procedure, che realizzano le varie componenti.
Esempi: MS-DOS, UNIX, GNU/Linux.

VANTAGGI	SVANTAGGI
• Basso costo di interazione tra componenti	• Non è ideale per garantire estendibilità, manutenibilità, riutilizzo, portabilità, affidabilità

Struttura modulare

Le varie componenti del SO vengono organizzate in moduli caratterizzati da interfacce ben definite.
Esempio:
Sistemi stratificati come il THE, pensato da Dijksta nel 1968, in cui il sistema operativo è costituito da livelli sovrapposti, ognuno dei quali realizza un insieme di funzionalità.

VANTAGGI	SVANTAGGI
• Astrazione: ogni livello è un oggetto astratto e fornisce ai livelli superiori una visione astratta (Macchina Virtuale) • Modularità: possibilità di sviluppo, verifica, modifica in modo indipendente dagli altri livelli	• Organizzazione gerarchica non sempre possibile • Scarsa efficienza: costo di attraversamento dei livelli

Struttura a microkernel

La struttura del **nucleo**, quella parte del sistema operativo che si interfaccia direttamente con l'hardware, è ridotta a poche funzionalità di base. Il resto del SO è rappresentato da processi utente.
Esempi: Minix, L4, Mach, Hurd, BeOS/Haiku

VANTAGGI	SVANTAGGI
• Affidabilità: separazione tra processi • Modularità: possibilità di estensioni e personalizzazioni	• Scarsa efficienza: molte chiamate a system call

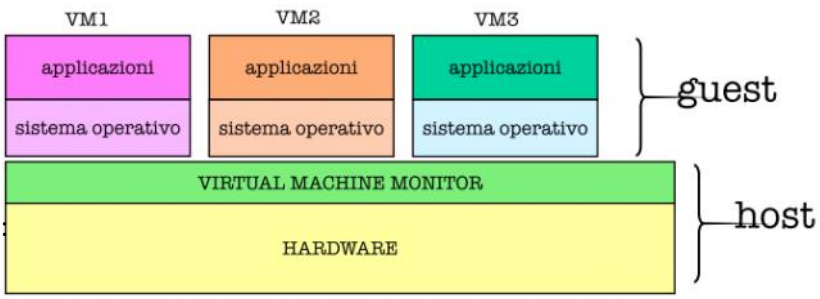
Kernel ibridi

Microkernel che integrano a livello kernel funzionalità non essenziali atte a ridurre le chiamate a system call da parte degli altri processi utente. Essi posseggono i vantaggi della struttra a microkernel e ne compensano gli svantaggi.
Esempi: Windows, XNU

Virtualizzazione

La macchina fisica è trasformata in N macchine virtuali.
Su ogni VM è possibile installare un SO.
Il Virtual Machine Monitor (VMM) è il mediatore tra le macchine virtuali e l'HW sottostante. Esso garantisce isolamento tra le Vm e stabilità del sistema e può essere:

- di sistema (in fig.) *Esempi: Vmware, Kvm*
- ospitato *Esempi: VirtualBox*



Introduzione al SO

sabato 11 maggio 2019 09.46

Interruzioni

Le varie componenti HW e SW del sistema operativo interagiscono mediante interruzioni asincrone (**interrupt**).
Ad ogni interruzione è associata una routine di servizio (**handler**) per la gestione dell'evento associato.

Interruzioni hardware

Le interruzione hardware sono generate dai dispositivi, che inviano segnali alla CPU pe notificare particolari eventi al SO.

Interruzioni software (**trap**)

Le interruzioni software sono generate da programmi in esecuzione per:

- segnalare operazioni non lecite
- richiedere l'esecuzione di servizi (si parla di **system call**)

Gestione delle interruzioni

Alla ricezione di un'interruzione, il SO:

1. interrompe la propria esecuzione
2. attiva la routine di servizio all'interruzione
3. ripristina lo stato salvato

Input / Output

Controller: interfaccia HW delle periferiche verso il bus di sistema. E' dotato di:

- registro dati
- registri speciali per memorizzare le specifiche delle operazioni I/O da eseguire

Driver: componente SW del SO che interagisce direttamente con il dispositivo.

Quando un job richiede un'operazione di I/O:

1. La CPU scrive nei registri speciali del dispositivo coinvolto le specifiche dell'operazione
2. Il controller esamina l'operazione scritta e provvede a trasferire i dati richiesti
3. Invio di un interrupt alla CPU per segnalare il completamento del trasferimento
4. La CPU esegue l'operazione di I/O tramite la routine di servizio

Protezione

Per garantire protezione molte architetture CPU prevedono un duplice modo di funzionamento (**dual mode**) attraverso il bit di modo:

- kernel mode: 0
- user mode: 1

Istruzioni privilegiate possono essere eseguite soltanto se il sistema operativo si trova in kernel mode.
Se dunque un utente necessita di eseguire operazioni privilegiate, l'unico modo è quello di invocare una system call:

1. invio di un'interruzione software al SO
2. salvataggio dello stato del programma chiamante e trasferimento del controllo al SO
3. esecuzione dell'operazione in kernel mode
4. ritorno del controllo al programma chiamate in user mode

Unix & Linux

giovedì 14 marzo 2019 10.17

Storia di Unix
Nel 1970 si ha la prima versione di UNIX.
Nel 1988 l'IEEE definisce lo standard POSIX (Portable Operating System Interface for uniX) che elenca le caratteristiche relative alle modalità si utilizzo del sistema operativo.

GNU (GNU is Not Linux)
Nel 1984 Richard Stallman avvia un progetto di sviluppo si un sistema operativo livero compatibile con Unix.
Nel 1991 Linus Tornvalds realizza Linux, un kernel che viene integrato nel progetto GNU: nasce il sistema operativo GNU/Linux.
Caratteristiche:

- Muliprogramato time-sharing
- Multiutente
- Multi-threaded
- Kernel monolitico con caricamento dinamico dei moduli
- Estendibile

Distribuzioni GNU/Linux

- Redhat / Fedora
- Slackware
- Debian / Ubuntu
- SUSE

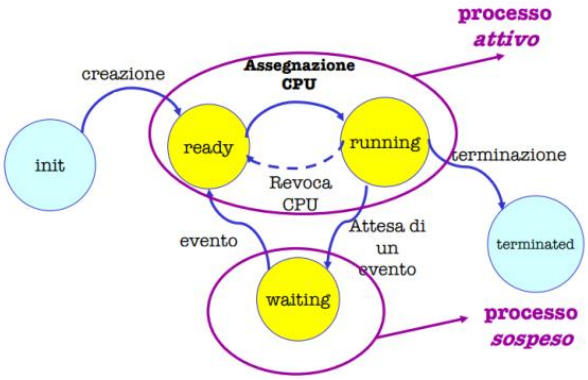
Comandi della shell
Per ogni comando da eseguire la shell crea uno shell figlio dedicato all'esecuzione del comando.

```
~$ date         data
~$ who          info utenti connessi
~$ whoami       info utente corrente
~$ pwd          print working direcory
~$ cd           change directory
~$ ls           list
~$ man          manual
```

Processi - Introduzione

giovedì 16 maggio 2019 15.30

Il **processo** è un programma in esecuzione. E' l'unità di esecuzione all'interno del SO. Un SO multiprogrammato permette l'esecuzione concorrente di più processi.
Un processo è rappresentato da: codice del programma, dati (variabili globali), PC, registri CPU, stack e possono essere associate risorse di SO (file, connessioni di rete, dispositivi I/O).



Ad ogni processo viene associata una struttura dati (descrittore) chiamata **Process control block (PCB)** per mantenere le informazioni relative ad un processo:

- stato processo
- identificatore processo
- program counter
- registri CPU
- limiti di memoria
- file aperti
- …

Scheduling dei processi

Lo **scheduling** è l'attività mediante la quale il SO effettua scelte tra i processi riguardo a caricamento in memoria centrale e assegnazione CPU. In generale vi sono tre attività di scheduling:

a breve termine (o di CPU)	Selezione dei processi a cui assegnare la CPU dalla coda dei processi pronti (**ready queue**) e cambio di contesto (**context switch**): salvataggio dello stato di P_i, ripristino dello stato di P_{i+1}.
a medio termine (o **swapping**)	Trasferimento temporaneo di (parti di) processi in memoria secondaria in condizioni critiche di spazio libero in memoria centrale.
[a lungo termine]	Selezione dei programmi dalla memoria seconaria da eseguire per caricarli in memoria centrale. Non è presente nei sistemi **time-sharing**.

Gerarchie di processi

Un processo (padre) può generare altri processi (figli) che possono essere a loro volta padri di altri processi.
Se un processo termina, il padre può rilevare il suo stato di terminazione (il SO mantiene il riferimento al padre nel PCB).
Caratteristiche:

- concorrenza: Il padre e il figlio possono procedere in parallelo (caso UNIX) oppure il padre può sospendersi in attesa della terminazione del figlio.
- condivisione di risorse: Le risorse del padre possono essere condivise con i figli (caso UNIX) oppure usate solo su esplicita richiesta.
- spazio degli indirizzi: Lo spazio degli indirizzi può essere *duplicato*, cioè essere una copia di quello del padre (es. fork() in UNIX), oppure *differenziato*, cioè con codice e dati diversi (es. exec() in UNIX).

Processi leggeri (o Thread)

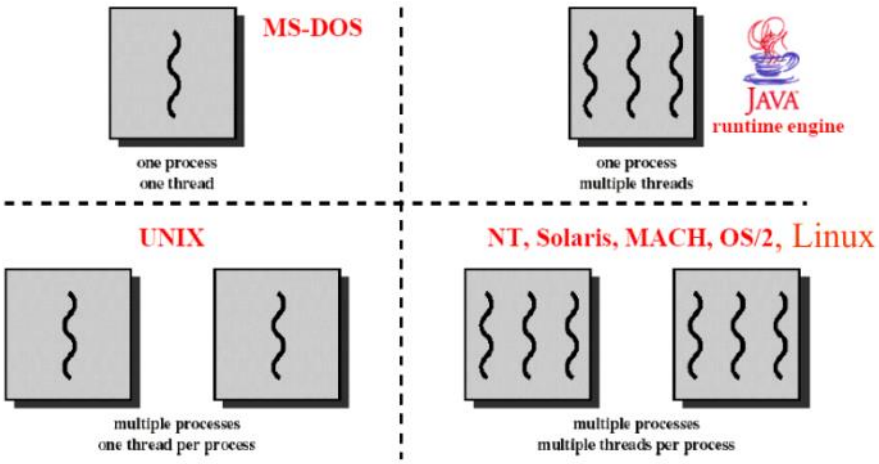
giovedì 16 maggio 2019 16.37

Processi leggeri (Thread)

Un **thread** (o processo leggero) è un'unità di esecuzione che condivide codice e dati con altri thread associati ad esso.
E' detto **task** l'insieme dei thread che riferiscono lo stesso codice e gli stessi dati. Un processo è un task con un solo thread.

VANTAGGIO	SVANTAGGIO
Lo scheduling di thread comporta un minor costo di context switch poichè il PCB non contiene informazioni su codice e dati globali.	Minore protezione: thread dello stesso task possono modificare dati in comune tra loro

Approcci adottati dai vari sistemi:

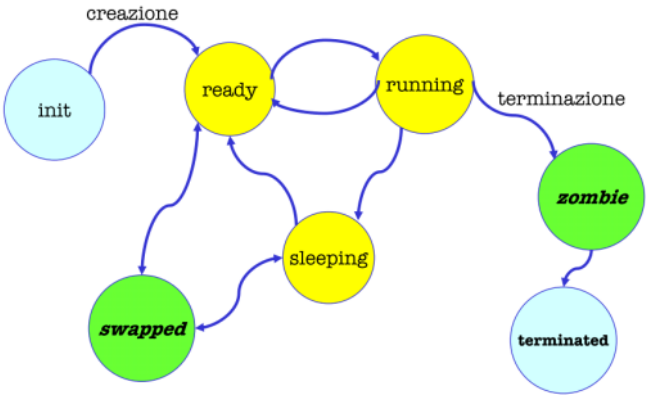


Processi UNIX

giovedì 14 marzo 2019 10.22

In UNIX ogni processo ha un proprio spazio di indirizzamento locale e non condiviso (modello ad ambiente locale o a scambio di messaggi), l'unica eccezione è per il codice che invece può essere condiviso.

Stati di un processo UNIX



Zombie: il processo è terminato ma è in attesa che il padre ne rilevi lo stato di terminazione

Swapped: il processo (o parte di esso) è temporaneamente trasferito in memoria secondaria

Lo scheduler a medio termine, o **swapper**, quando la memoria è in condizioni critiche, trasferice dei processi (tipicamente quelli in stato di sleeping e lunghi) in memoria secondaria (swap out) per liberare spazio in quella centrale.

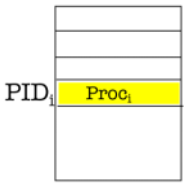
Rappresentazione dei processi UNIX

Il Process Control Block (PCB), la struttura contenente le informazioni necessarie a gestire la schedulazione di un processo, è rappresentato da:

1. **Process Structure**: info necessarie al sistema per la gestione del processo

PID (Process IDentifier)	stato	puntatori a aree dati / stack / codice	info scheduling	riferimento al padre	puntatore a User Structure

Tutte le Process Structure sono organizzate in un vettore: **Process Table**



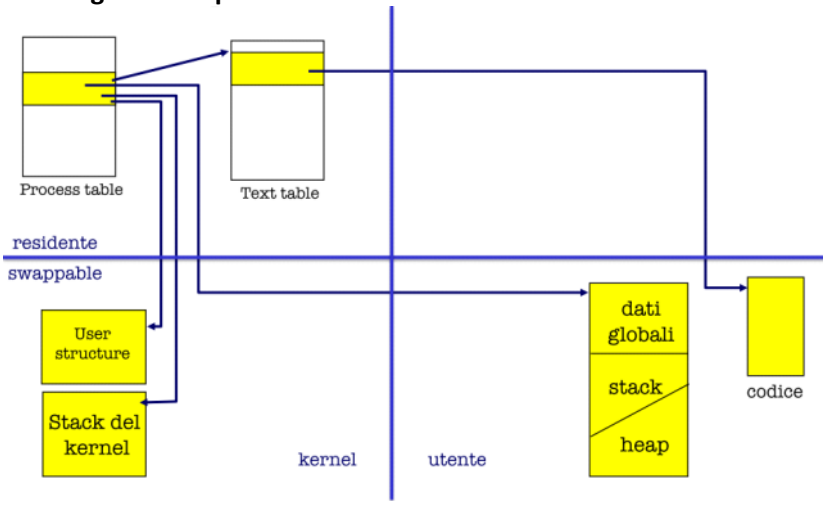
Process table: 1 elemento per ogni processo

2. **User Structure**: info necessarie se il processo è in memoria centrale

registri di CPU	file aperti	info su gestione di segnali	ambiente del processo (direttorio corrente, utente, gruppo, argc/argv)

Immagine di un processo UNIX

L'**immagine di un processo** è l'insieme delle aree di memoria e strutture dati associate al processo.



utente	accessibile dall'utente
kerner	non accessibile dall'utente
swappable	parte dell'immagine che può essere trasferita in memoria secondaria
residente	parte dell'immagine che non può essere trasferita in memoria secondaria

Processi UNIX - System Call

giovedì 14 marzo 2019 10.55

FORK

`#include <unistd.h>`

```
int fork(void);
```

Consente ad un processo di generare un processo figlio. Il figlio eredita una copia dei dati del padre (PS e US).

return:

- = 0 se siamo nel processo figlio creato
- > 0 se siamo nel processo padre e indica il PID del figlio
- < 0 se si è riscontrato qualche errore nella creazione

EXIT

`#include <unistd.h>`

```
void exit(int status);
```

Consente di terminare il processo corrente e comunicare lo stato di chiusura al padre.

params:

`int`	`status`	stato di terminazione del processo corrente da comunicare al padre

WAIT

`#include <sys/wait.h>`

```
int wait(int* status);
```

Consente al padre di aspettare che il figlio termini per ottenere lo stato di ritorno

params:

`int*`	`status`	l'indirizzo della variabile in cui viene memorizzato lo stato di terminazione del figlio

return:

- > 0 indica il PID del processo terminato
- < 0 se il padre non ha figli (utile per controllare quando tutti i figlio hanno terminato l'esecuzione)

EXEC

Con la fork i processi padre e figlio condividono il codice. In UNIX è possibile differenziare il codice dei due processi mediante una system call della famiglia exec:

`execl`	`execle`	`execlp`	`execv`	`execve`	`execvp`	…

Il meccanismo è quello di sostituire codice ed eventuali argomenti di invocazione con coice e argomenti di un programma specificato come parametro.

```
int execl(char* pathname, char* arg0, ..., char* argN, (char*)0);
```

params:

`char*`	`pathname`	percorso (assoluto o relativo) dell'eseguibile da caricare
`char*`	`arg0`	nome del programma da eseguire
`char*`	`arg1,...,argN`	argomenti da passare al programma
	`(char*)0`	è il puntatore nullo che termina la lista

```
int execve(char* pathname, char* argV[], char* env[]);
```

params:

`char*`	`pathname`	percorso (assoluto o relativo) dell'eseguibile da caricare
`char*`	`argV[]`	vettore degli argomenti del programma da eseguire
`char*`	`env[]`	vettore delle variabili d'ambiente da sostituire nell'ambiente del processo nel formato "VAR=val"

PERROR

In caso di fallimento ogni system call ritorna un valore negativo, in aggiunta UNIX prevede la variabile globale **errno** alla quale il kernel assegna il codice di errore generato dall'ultima system call eseguita. Per interpretarlo è possibile usare: `<sys/errno.h>`

```
void perror(const char* str);
```

che stampa la stringa seguita dal numero errore presente nella variabile globale errno.

Processi - Interazione

martedì 19 marzo 2019 09.23

Tipologie di processi

Processi interagenti se l'esecuzione di un processo è influenzata dall'esecuzione dell'altro.
Processi indipendenti se l'esecuzione di un processo non è influenzata dall'esecuzione dell'altro.

Tipi di interazione:

- **Cooperazione** interazione prevedibile, desiderata, insita nella logica del programma.
- **Competizione** interazione prevedibile ma non desiderata, i processi interagiscono per sincronizzarsi nell'accesso alle risorse comuni.
- **Interfefenza** interazine non prevista e non desiderata, potenzialmente deleteria (es. condivisione di stampante)

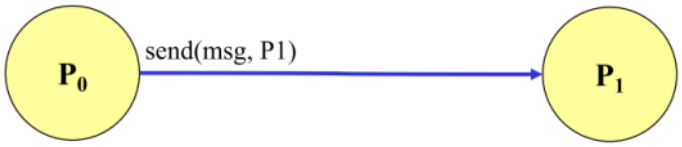
L'interazione può avvenire mediante:

- **Comunicazione - Memoria condivisa** (modello ad ambiente globale) *thread* : condivisione di variabili
- **Sincronizzazione - Scambio di messaggi** (modello ad ambiente locale) *processi pesanti* : meccanismi di trasmissione / ricezione di messaggi (eventi)

Modello ad ambiente locale

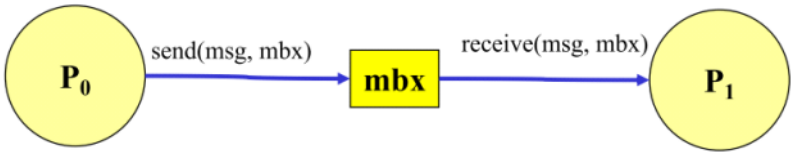
- Comunicazione
  - Comunicazione diretta

    Al messaggio viene associato l'identificatore del processo destinatario (naming esplicito).



  - Comunicazione indiretta

    Il messaggio viene indirizzato a una mailbox dalla quale il destinatario preleverà il messaggio




```
Processo produttore P:
pid C =....;
main()
{ msg M;
  do
  { produco(&M);
    ...
    send(C, M);
  }while(!fine);
}
```

```
Processo consumatore C:
pid P=....;
main()
{ msg M;
  do
  { receive(P, &M);
    ...
    consumo(M);
  }while(!fine);
}
```

Es. produttore - consumatore

    - I processi cooperanti non sono tenuti a conoscersi
    - La mailbox è un canale utilizzabile da più processi che funge da contenitore di messaggi.
    - Tale canale ha una capacità, un numero massimo di messaggi che è in grado di contenere contemporaneamente ed è gestito con polititca FIFO.

  ### Comunicazione tra processi UNIX

  Pipe: comunicazione indiretta, canale unidirezionale molti-a-molti e bufferizzata (con capacità limitata).

- Sincronizzazione

Processi UNIX - Comunicazione

martedì 2 aprile 2019 10.11

I processi in Unix non possono condividere memoria (modello ad amiente locale). L'interazione può dunque avvenire:

- condivizione di file
- strumenti di Inter Process Communication:
  - sulla stessa macchina

    pipe tra processi della stessa gerarchia

    fifo qualunque insieme di processi (canale unidirezionale con nome nel file system)
  - sulla stessa rete

    socket

PIPE

- comunicazione indiretta
- unidirezionale o bidirezionale
- molti-a-molti
- capacità limitata
- comunicazione asincrona
- omogenerità con i file

```
int pipe(int fd[2]);
```

params:

`int[2]`	`fd`	`fd[0]` rappresenta il lato di lettura della pipe `fd[1]` rappresenta il lato di scrittura della pipa

return:

- = 0 in caso di successo
- < 0 in caso di errore

Il canale (la pipe) ha capacità limitata. può essere piena o vuota. *Sincronizzazione automatica*: read e write da/verso pipe possono essere sospensive (a meno che relativamente tutti i canali di scrittura/lettura non siano chiusi).
Soltanto i processi apartenenti a una stessa gerarchia possono scambiarsi messaggi mediante pipe.
E' bene che ogni processo chiuda il lato della pipe che non usa con una close. Un estremo della pipe viene detto effetivmente chiuso quando tutti i processi che ne avevano visibilità hanno compiuto una close.

Se un processo P:

- tenta una lettura da pipe vuota in cui il lato di scrittura è effettivamente chiuso:
  `read` ritorna 0
- tenta una scrittura da pipe il cui lato di lettura effettivamente chiuso:
  `write` ritorna -1 e vine inviato a P un segnale `SIGPIPE`

DUP

```
int dup(int fd);
```

params:

`int`	`fd`	`file descriptor del file da duplicare`

return:

- > 0 il file descriptor del file aperto copiato
- - 1 in caso di errore

FIFO

```
int mkfifo(char *pathname, int mode);
```

params:

`char*`	`pathname`	`il nome della fifo`
`int`	`mode`	`permessi`

return:

- = 0 in caso di successo
- < 0 in caso di errore

Processi UNIX - Sincronizzazione in ambiente locale

martedì 19 marzo 2019 11.02

La sincronizzazione permette di imporre vincoli sull'ordine di esecuzione delle operazioni dei processi interagenti.
La sicronizzazione può realizzarsi mediante un **segnale**: un'interruzione software che notifica un evento in modo asincrono al processo che lo riceve.

Segnali UNIX

Kernel → Proc. Utente	Proc. Utente → Proc. Utente

Quando un processo riceve un segnale può:

- **gestire** il segnale con una funzione handler
- **eseguire** un'azione predefinita del SO (azione di default)
- **ignorare** il segnale

}

1. interruzione esecuzione
2. esecuzione azione associata (handler o default)
3. ritorno all'istruzione successiva del processo interrotto

Gestione di segnali in C

```
#include <signal.h>
#define SIGxxx --
```
→ molti già definiti in signal.h

SIGNAL

```
void (* signal(int sig, void (*func)()))(int);
```

Consente di eseguire una funzione alla ricezione di un dato segnale

params:

`int`	`sig`	il codice del segnale da gestire
`void`	`func`	il puntatore alla funzione da eseguire

return:

un puntatore a funzione

- al gestore del segnale, in caso di successo
- SIG_ERR(-1), in caso di errore

Esempio di utilizzo:

```
#include <signal.h>
void handler(int signum) {
    <istruzioni per la gestione del segnale>
    return;
}
main() {
    ...
    signal(SIGxxx, handler);
    ...
}
```

KILL

```
int kill(int pid, int sig);
```

Consente di inviare un segnale ad un processo

params:

`int`	`pid`	Destinatario del segnale • > 0 l'unico processo destinatario • = 0 tutti i processi appartenenti al gruppo del mittente • < -1 tutti i processi con groupID uguale al valore assoluto di pid
`int`	`sig`	il codice del segnale da gestire

Processi UNIX - Sincronizzazione in ambiente locale

domenica 24 marzo 2019 23.48

SLEEP

```
unsigned int sleep(unsigned int N);
```

Provoca la sospensione del processo per (al massimo) N secondi.

params:

`unsigned int`	`N`	numero di secondi per cui si vuole sospendere il processo a meno che non riceva un segnale

return:

- = 0 se la sospensione non è stata interrotta da segnali
- > 0 il numero di secondi rimanenti dopo l'interruzione

ALARM

```
unsigned int alarm(unsigned int N);
```

Imposta un timer che dopo N secondi invierà al processo il segnale `SIGALRM`.

params:

`unsigned int`	`N`	numero di secondi tra cui si vuole ricevere il segnale

return:

- = 0 se la sospensione non è stata interrotta da segnali
- > 0 il numero di secondi rimanenti dopo l'interruzione

PAUSE

```
int pause(void);
```

Sospende il processo fino alla ricezione di un qualunque segnale

params:

`unsigned int`	`N`	numero di secondi tra cui si vuole ricevere il segnale

return:

- -1 (errno = EINTR)

File System

martedì 26 marzo 2019 09.23

Il **file system** è quella componente del SO che fornisce i meccanismi di accesso e memorizzazione delle informazioni allocate in memoria di massa. Realizza i concetti astratti di file, direttorio e partizione. La sua struttura può essere organizzata a livelli:

Applicazioni	
Struttura logica	presenta alle applicazioni una visione astratta delle informazioni memorizzate
Accesso	definisce i meccanismi di accesso e protezione dei file
Organizzazione fisica	allocazione dei file su dispositivo, rappresentazione della struttura logica sul dispositivo
Dispositivo virtuale	presenta una vista astratta del dispositivo: una sequenza di blocchi di dimensione costante
Hardware (memoria secondaria)	

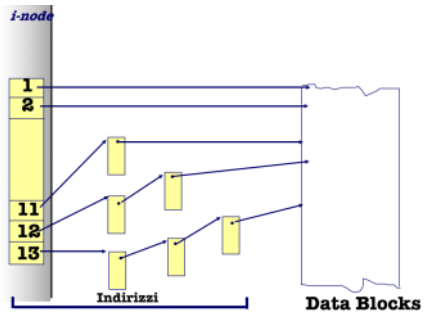
Record logico: unità di trasferimento logico nelle operazioni di accesso al file. Di dimensione variabile.
Blocco: unità di trasferimento fisico nelle operazioni di I/O da/verso il dispositivo. Di dimensione fissa.

Metodi di allocazione

- Metodo di allocazione contigua
  Ogni file è mappato su un insieme di blocchi fisicamente contigui, questo da una parte costituisce un vantaggio in tempo di ricerca (possibilità di accesso sequenziale o diretto), dall'altra parte comporta problemi nell'aumento dinamico delle dimensioni di un file che conduce a frammentazione esterna.
- Allocazione a lista concatenata
  I blocchi che compongono un file sono organizzati in una lista concatenata, questo risolve il problema di riallocazione in caso di aumento dinamico delle dimensioni e frammentazione esterna. Aumenta però il costo di ricerca di un blocco.
- Metodo di allocazione ad indice (a più livelli di indirizzamento):
  Simile a lista concatenata ma tutti i puntatori ai blocchi utilizzati per un determinato file sono concentrati in un unico blocco detto blocco indice.

File System di Unix

- Organizzazione ad albero/grafo
- Tutto è visto come file: file ordinari, direttori, dispositivi fisici (/dev) detti partizioni
- Ad ogni file possono essere associati uno o più nomi ma un solo descrittore (**i-node**)
- Allocazione ad indice a più livelli di indirizzamento:
  L'allocazione del file non è su blocchi fisicamente contigui; nell'i-node sono contenuti puntatori a blocchi, ad esempio 13, dei quali i primi 10 riferiscono blocchi dati, l'11° riferisce un blocco contentente indirizzi di blocco dati (1 livello di indirettezza), il 12° un blocco contentente indirizzi di blocchi di indirizzi di blocco dati (2 livelli di indirettezza) e il 13° uguale con 3 livelli di indirettezza. Considerata dimensione 512 bytes e blocchi da 128 indirizzi si può calcolare il massimo: $10\times512+128\times512+128^2\times512+128^3\times512 = 5\text{KB} + 64\text{KB} + 8\text{MB} + 1\text{GB}$.



- Formattazione del disco in blocchi fisici (dimensione [512 - 4096] bytes)
- Il file system è partizionato in 4 regioni:

Boot block	Contiene le procedure di inizializzazione del sistema
Super block	Fornisce i limiti delle 4 regioni e puntatori alla lista degli i-node liberi e alla lista dei blocchi liberi
i-List	Lista di tutti i descrittori (i-node) dei file, direttori e dispositivi. La posizioni vengono dette **i-number**
Data blocks	Area di disco disponibile per memorizzazione dei file. I blocchi liberi sono organizzati in una lista

- **Virtual File System**: Linux prevede l'integrazione con filesystem diversi da Ext grazie al VFS che intercetta ogni system call relativa al file system e all'occorrenza provvede al collegamento con file system "esterni"

File System - Accesso ai file

venerdì 29 marzo 2019 09.12

File

Ogni file è individuato almeno da un nome simbolico. Il **Descrittore del file**, la struttura dati che contiene gli attributi del file:

- tipo (ordinario, direttorio, speciale)
- indirizzo/i
- dimensione
- utente proprietario, gruppo
- data e ora (di creazione e/o modifica)
- bit di protezione per i diritti di accesso
  In UNIX 12 bit

bit di permessi			permessi utente			permessi gruppo			permessi per gli altri		
SUID	SGID	STICKY	r	w	x	r	w	x	r	w	x

  - SUID (Set User ID) se settato, associa al processo che esegue il file l'user id del proprietario
  - SGID (Set Group ID) se settato, associa al processo che esegue il file il group id del proprietario
  - STICKY (Save Text Image) se settato, l'immagine del processo viene mantenuta in area di swap anche dopo aver finito il proprio compito per una maggiore velocità di riavvio

Operazioni su file:

- creazione
- lettura
- scrittura
- cancellazione
- apertura
- chiusura

Accesso ai file

L'accesso ai file può avvenire secondo varie modalità:

- Accesso **sequenziale**: Il file è una sequenza di record logici. Per accedere ad un determinato record occorre prima scorrere tutti quelli che lo precedono. Vi è un puntatore che punta alla posizione corrente.
- Accesso **diretto**: Il file è un insieme di record logici numerati. Si può accedere direttamente ad un particolare record. Utile con file di grosse dimensioni per modifiche di piccola entità
- Accesso ad **indice**: Ad ogni file è associata una struttura dati contenente l'indice delle informazioni contenute. Per ogni accesso si esegue una ricerca nell'indice.

Strutture dati

- Tabella dei file aperti di processo (User area del processo)
  Tabella dei file aperti dal processo allocata nella user structure. Ha dimesione limitata (tipicamente 20 elementi). Ogni elemento è individuato da un indice intero detto **file descriptor**.
- Tabella dei file aperti di sistema (kernel)
  Metodo di accesso sequenziale: contiene un elemento per ogni sessione di acceso da parte dei processi a file nel sistema. Ogni elemento contiene:
  - l'**I/O pointer** che indica la posizione corrente all'interno del file
  - Un puntatore al'**i-node** del file nella tabella dei file attivi
- Tabella dei file attivi (kernel)
  Contiene un elemento per ogni file aperto nel sistema.

File e programmazione concorrente

Caso 1) Faccio la fork e successivamente apro un file:
I due processi hanno I/O pointer allo stesso file ma distinti.

Caso 2) Apro un file e successivamente faccio la fork:
I due processi condividono l'I/O pointer e dunque se uno dei due si sposta all'interno del file anche l'alto processo si troverà nel punto in cui era arrivato l'altro processo

Apertura di file

L'apertura di un file provoca:

1. l'inserimento di un elemento nella prima posizione libera della tabella dei file aperti del processo (fig.1)
2. L'inserimento di un nuovo record nella tabella dei file aperti di sistema
3. La copia dell' i-node nella tabella dei file attivi e memory mapping del file

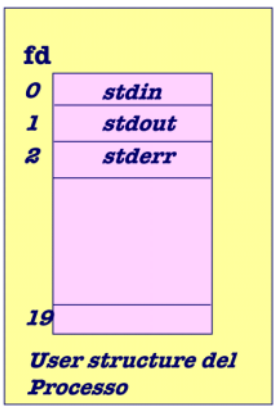


fig.1

File System - Accesso ai file

venerdì 29 marzo 2019 09.50

OPEN

```
#include <fcntl.h>
```

```
int open(char nomefile[], int flag, [int mode]);
```

params:

char*	nomefile	Nome del file
int	flag	Eprime il modo di accesso: • O_RDONLY (=0) accesso in lettura • O_WRONLY (=1) accesso in scrittura • O_RDWR • O_CREAT (=2) creare file • O_TRUNK E' possibile combinarli mediante il connettore ' | '
int	mode	Richiesto soltanto se l'apertura determina la creazione del file. In tal caso specifica i bit di protezione

return:

- > 0 è il file descriptor del associato al file
- -1 in caso di errore

CREAT

```
int creat(char nomefile[], int mode);
```

params:

char*	nomefile	Nome del file
int	mode	Richiesto soltanto se l'apertura determina la creazione del file. In tal caso specifica i bit di protezione

return:

- > 0 è il file descriptor del associato al file
- -1 in caso di errore

Chiusura di file

La chiusura di un file provoca:

1. La memorizzazione del file su disco
2. L'eliminazione dell'elemento di indice fd dalla tabella dei file aperti del processo
3. L'eliminazione degli elementi corrispondenti dalla tabella dei file aperti di sistema e dei file attivi

CLOSE

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

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

params:

int	fd	File descriptor del file da chiudere

return:

- 0 in caso di successo
- -1 in caso di errore

File System - Accesso ai file

lunedì 10 giugno 2019 17.21

Lettura da file

READ

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

```
int read(int fd,char* buf, int n);
```

params:

int	fd	File descriptor del file
char*	buf	l'area in cui trasferire i byte letti
int	n	numero di caratteri da leggere

return:

- > 0 && < n in caso di successo, indica il numero di caratteri effettivamente letti
- 0 in caso non si sia riuscito a leggere nessun carattere
- -1 in caso di errore

Scrittura su file

WRITE

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

```
int write(int fd,char* buf, int n);
```

params:

int	fd	File descriptor del file
char*	buf	l'area in cui trasferire i byte letti
int	n	numero di caratteri da leggere

return:

- > 0 && ==n in caso di successo, indica il numero di caratteri effettivamente scritti
- -1 in caso di errore

Caratteristiche Lettura / Scrittura:

- accesso mediante il file descriptor
- ogni operazione di lettura (o scrittura) agisce sequenzialmente sul file, a partire dalla posizione corrente del puntatore (I/O pointer)
- possibilità di alternare operazioni di lettura e scrittura
- atomicita` della singola operazione
- operazioni sincrone, cioè con attesa del completamento dell'operazione.

File System - Accesso ai file

venerdì 29 marzo 2019 10.26

LSEEK

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

```
int lseek(int fd, int offset, int origine);
```

params:

int	fd	File descriptor del file da chiudere
int	offset	Spostamento in byte rispetto all'origine
int	origine	• SEEK_SET (=0) inizio file • SEEk_CUR (=1) posizione corrente • SEEK_END (=2) fine file

return:

- > 0 in caso di successo, rappresenta la nuova posizione
- < 0 in caso di errore

UNLINK

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

```
int unlink(char* name);
```

Decrementa di 1 il numero di link del file (nell' i-node), eliminando il nome specificato dalla struttura logica del file system.

params:

char*	name	Nome del file

return:

- = 0 in caso di successo
- < 0 in caso di errore

LINK

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

```
int link(char* oldname, char* newname);
```

Incrementa di 1 il numero di link del file (nell' i-node), aggiorna il direttorio.

params:

char*	oldname	Nome del file esistente
char*	newname	Nome associato al nuovo link

return:

- 0 in caso di successo
- -1 in caso di errore

ACCESS

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

```
int access(char* pathname, int amode);
```

Serve a verificare i diritti di un utente di accedere a un file.

params:

char*	pathname	Nome del file
int	amode	Diritto da verificare: • 04 read access • 02 write access • 01 execute access • 00 existence

return:

- 0 in caso di successo
- -1 in caso di errore

File System - Accesso ai file

martedì 2 aprile 2019 09.31

STAT

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

```
int stat(const char *path, struct stat *buf);
```

Serve a verificare i diritti di un utente di accedere a un file.

params:

`const char*`	`path`	Nome del file
`struct stat *`	`buf`	Puntatore a una struttura di tipo stat nella quale vengono restituiti gli attributi del file

return:

- 0 in caso di successo
- -1 in caso di errore

```
struct stat {
    dev_t  st_dev;      ID of device containing file
    ino_t st_ino;       i-number
    mode_t st_mode;     protection & file type
    nlink_t st_nlink;   number of hard links
    uid_t st_uid;       user ID of owner
    gid_t st_gid;       group ID of owner
    dev_t st_rdev;      device ID (if special file)
    off_t st_size;      total size, in bytes
    blksize_t st_blksize; blocksize for file system I/O
    blkcnt_t st_blocks; number of blocks allocated
    time_t st_atime;    time of last access
    time_t st_mtime;    time of last modification
    time_t st_ctime;    time of last status change
};
```

CHMOD

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

```
int chmod(char *pathname, char *newmode);
```

Per modificare i bit di protezione di un file.

params:

`char*`	`pathname`	Nome del file
`char*`	`newnode`	Nuovi diritti

return:

- 0 in caso di successo
- -1 in caso di errore

CHOWN

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

```
int chown(char *pathname, int owner, int group);
```

Per cambiare il proprietario e il gruppo di un file.

params:

`char*`	`pathname`	Nome del file
`int`	`owner`	UID del nuovo proprietario
`int`	`group`	GID del gruppo

return:

- 0 in caso di successo
- -1 in caso di errore

File System - Gestione dei direttori

martedì 2 aprile 2019 09.50

Directory

La directory è uno strumento per organizzare i file all'interno del file system. Può essere:

- a un livello (una sola directory per file system)
- a due livelli (una direcotry principale che contiene una directory per utente ad un livello)
- ad albero (gerarchie di directory)
- a grafo aciclico (uno stesso file può essere riferito da link da diverse posizioni)

Operazioni sui direttori

- creazione / cancellazione
- aggiunta / rimozione dei file
- listing
- attraversamento
- ricerca

CHDIR

```
int chdir(char *nomedir);
```

Per cambiare il direttorio

params:

char*	nomedir	Nome della directory

return:

- 0 in caso di successo
- -1 in caso di errore

OPENDIR

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

```
DIR *opendir(char *nomedir);
```

Per aprire un direttorio

params:

char*	nomedir	Nome della directory

return:

- un puntatore a DIR al direttorio
- NULL in caso di errore

CLOSEDIR

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

```
int closedir(DIR *dir);
```

Per chiudere un direttorio

params:

DIR*	dir	Puntatore alla directory

return:

- un puntatore a DIR al direttorio
- NULL in caso di errore

File system - Gestione direttori

lunedì 10 giugno 2019 17.37

READDIR

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

```
struct dirent *readdir(DIR *dir);
```

Per leggereun direttorio

params:

DIR*	dir	Puntatore alla directory

return:

- un puntatore ad una struttura di tipo `dirent` dichiarata in `dirent.h`
- NULL in caso di errore

```
struct dirent {
    long d_ino;                 i-number
    off_t d_off;                offset del prossimo
    unsigned short d_reclen;    lunghezza del record
    unsigned short d_namelen;   lunghezza del nome
    char *d_name;               nome del file
}
```

MKDIR

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

```
int mkdir(char *pathname, int mode);
```

Per aprire un direttorio

params:

char*	pathname	Nome della directory da creare
int	mode	I bit di protezione

return:

- = 0 in caso di successo
- < 0 in caso di errore

Scheduling a breve termine (o di CPU)

venerdì 5 aprile 2019 10.45

Scheduling della CPU (a breve termine):
Commutazione dell'uso della CPU tra i vari processi.

Scheduler della CPU o Dispatcher :
È quella parte del SO che seleziona dalla coda dei processi pronti il prossimo processo al quale assegnare l'uso della CPU.

Coda dei processi pronti (ready queue)
Contiene i descrittori (PCB) dei processi pronti. La ready queue è realizzata mediante politiche di scheduling.

Processo CPU bound: Processi che hanno attività prevalentemente di computazione (CPU burst)
Processo I/O bound: Processi che hanno attività prevalentemente di I/O (I/O burst)

Pre-emption
Allo scheduling è permesso sottrarre CPU ad un processo ed assegnarla ad un altro. Il processo può essere cioè p*relazionato*.

POLITICHE DI SCHEDULING

1. Utilizzo della CPU: percentuale di utilizzo CPU (Ideale : 100%)
2. Throughput: numero di processi completati nell'unità di tempo (Ideale : >>)
3. Tempo di attesa: tempo totale trascorso nella code dei processi ready (Ideale : <<)
4. Turnaround: Durata vita del processo (Ideale : <<)
5. Tempo di risposta: Durata tra inizio e prima risposta (Ideale : <<)

ALGORITMO DI SCHEDULING **FCFS** (First Come First Served)

La coda dei processi viene gestita in modo FIFO. Non pre-emptive.
problematiche:

- Elevato tempo di attesa nel caso di processi con lunghi CPU burst
- Scarso utilizzo di CPU se si accodano molti processi I/O bound

ALGORITMO DI SCHEDULING **SJF** (Shortest Job First)

Dà priorità ai job con CPU burst corti. Può essere non pre-emptive oppure pre-emptive Shortest Remaining Time First (**SRTF**) cioè, se nella coda arriva un processo con CPU burst minore del CPU burst rimasto al processo running, esso viene schedulato
problematica: stimare la lunghezza di CPU burst. Tecnica utilizzata exponential averaging:

t_n: lunghezza attuale dell'n-esimo CPU burst
τ_{n+1}: valore predetto per il prossimo CPU burst
α: un valore compreso tra 0 e 1

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

Starvation
Si verifica quando uno o più processi di priorità bassa vengono lasciati indefinitamente nella coda dei processi pronti perchè vi è sempre almeno un processo pronto di priorità più alta.
Soluzione: Modifica dinamica della priorità dei processi. Es. gestionde UNIX:

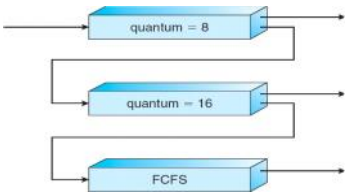
- la priorità decresce al crescere del tempo di CPU già utilizzato (feedback negativo o *aging*)
- la priorità cresce dinamicamente con il tempo di attesa del processo (feedback positivo o *promotion*)

ALGORITMO DI SCHEDULING **RR** (Round Robin)

Algoritmo preemptive tipicamente usato in sistemi time sharing. La ready queue viene gestita come una coda FIFO circolare (FCFS) e ad ogni processo viene assegnata la CPU per un intervallo Δt costante; al termine del quanto di tempo, se non ha completato l'esecuzione, il processo viene reinserito in coda. Risolve la problematica di starvation.
problematica: Dimensionamento del quanto di tempo: Un Δt troppo piccolo o troppo grande può comportare un peggioramento delle prestazioni.

APPROCCIO MISTO **MLFQ** (Multi Level Feedback Queue)

L'obiettivo è privilegiare i processi brevi: Più code (160), ognuna associata ad un tipo di job diverso. Ogni coda ha diversa priorità e viene gestita con un algoritmo di scheduling diveso (RR o FCFS). I processi possono muoversi da una coda all'altra in base alla loro storia.



Gestione della memoria

giovedì 18 aprile 2019 16.55

Il SO deve gestire la memoria in modo da consentire la presenza contemporanea di più processi. Ogni sistema è equipaggiato con un unico spazio di memoria accessibilie rispettamente da CPU e dispositivi: la memoria centrale.
I compiti del SO sono:

- *Allocare memoria* ai processi
- *Deallocare memoria*
- *Protezione*: separare gli spazi di indirizzi associati ad indirizzi diversi
- *Binding*: realizzare i collegamenti tra gli indirizzi logici e la memoria fisica
- *Memoria virtuale*: gestire spazi logici di dimensioni superiori allo spazio fisico

La memoria centrale è un vettore di celle, ognuna univocamente individuata da un indirizzo che può essere:

- *simbolico*: variabili → sorgente
- *logico*: riferimento a celle nello spazio logico di indirizzamento del processo → eseguibile rilocabile
- *fisico*: riferimento assoluto a livello HW → eseguibile

L'associazione tra indirizzo logico e fisico viene detta **binding**. Il binding può essere:

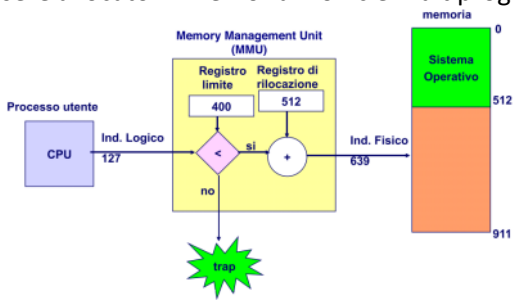
- *statico*:
  - ◇ a tempo di compilazione: il compilatore genera indirizzi assoluti
  - ◇ a tempo di caricamento: il compilatore genera indirizzi relativi che vengono converititi in assoluti dal loader
- *dinamico*: a tempo di esecuzione

TECNICHE DI ALLOCAZIONE:

ALLOCAZIONE CONTIGUA

- *partizione singola*

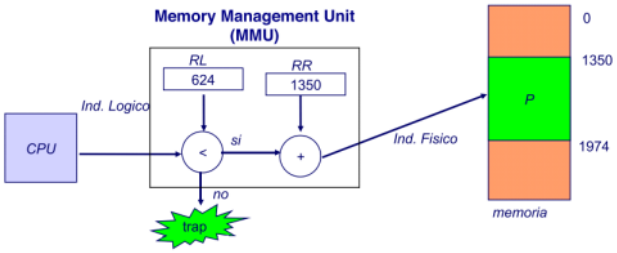
  Un solo processo alla volta può essere allocato in memoria: non c'è multiprogrammazione.



- *partizioni multiple*

  Ad ogni processo viene associata un'area di memoria distinta (partizione) definita da una coppia di valori $<V_{RR},V_{RL}>$:
  - RR : valore del registro di rilocazione
  - RL : valore del registo limite

  Quando un processo viene schedulato, il **dispatcher** carica i valori di RR e RL.



  - ○ partizioni fisse (MFT, Multiprogramming with Fixed number of Tasks)

    La dimensione di ogni partizione è fissata a priori.

Pro:	Contro:
▪ velocità	▪ **frammentazione interna** (sottoutilizzo della partizione) ▪ multiprogrammazione limitata al numero di partizioni ▪ dim. massima limitata alla dim. della partizione più estesa

  - ○ partizioni variabili (MVT, Multiprogramming with Variable number of Tasks)

    Ogni partizione è dimensionata in base alla dimensione del prcesso da allocare.

Pro:	Contro:
▪ eliminazione frammentazione interna ▪ grado di multiprogrammazione variabile ▪ dim. massima pari a dim. spazio fisico	▪ scelta dell'area in cui allocare: best fit, worst fit, first fit… ▪ **frammentazione esterna** (necessità di compattazione)

Gestione della memoria

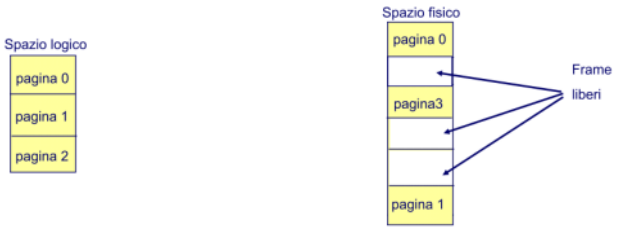
martedì 9 aprile 2019 10.52

ALLOCAZIONE NON CONTIGUA

- *Paginazione*

L'idea di base è quella di partizionare la memoria in pagine (**frame**) di dimensione costante (es. 4KB) sulle quali vengono mappate porzioni dei processi da allocare.

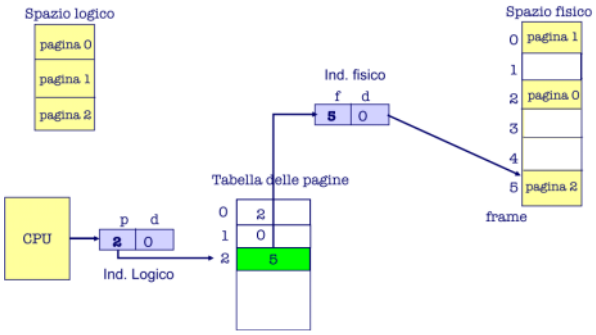
Pro:	Contro:
▪ eliminazione della frammentazione esterna ▪ riduzione della frammentazione interna ▪ è possibile caricare in memoria un sottoinsieme delle pagine logiche di un processo	



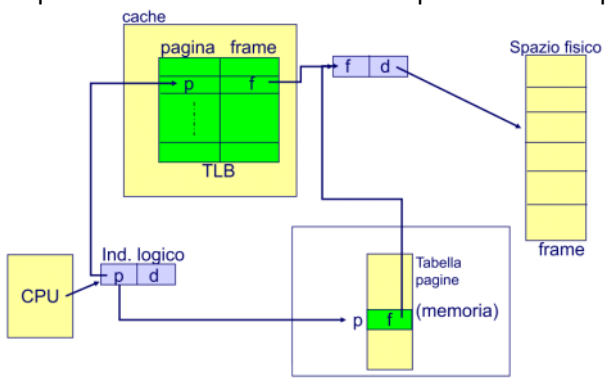
Struttura indirizzo logico	**Struttura indirizzo fisico**
p | d	f | d
p: numero di pagina logica d: offset della cella rispetto all'inizio della pagina	f: numero di pagina fisica (frame) d: offset della cella rispetto all'inizio del frame

Tabella delle pagine

Il binding tra indirizzi logici e fisici viene realizzato mediante la **tabella delle pagine** (associata al processo).



La tabella delle pagine è indirizzata dal registro **PTBR** (Page Table Base Register).
Poiché la tabella delle pagine può essere molto grande e il binding deve essere il più veloce possibile una soluzione è il **TLB** (Translation Look-aside Buffers) che consiste nell'allocare la tabella in memoria centrale e copiare la parte relativa alle pagine accedute più frequentemente in memoria cache per un accesso più veloce.



La tabella delle pagine ha dimensione fissa e non sempre viene utilizzata completamente. Per questo motivo per ogni entry della tabella delle pagine ci sono inoltre:

- *Bit di valità*:
  0) entry non valida
  1) entry valida (la pagina appartiene allo spazio logico del processo corrente)
- *Bit di protezione* che esprime le modalità di accesso alla pagina

→ Il registro **PTLR** (Page Table Length Register) contiene il numero degli elementi validi nella tablella delle pagine

Gestione della memoria

martedì 16 aprile 2019 15.42

Miglioria

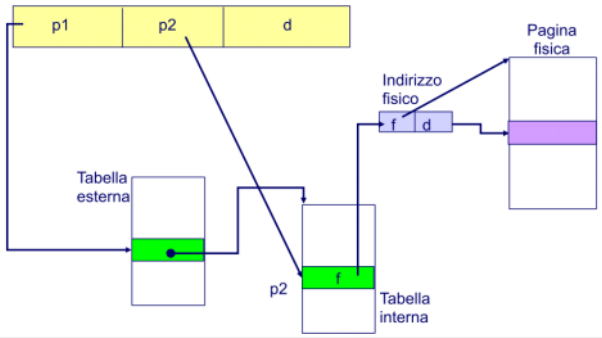
Paginazione a più livelli

Consiste nell'applicare la paginazione anche della tabella delle pagine. Questo perché lo spazio logico di indirizzamento di un processo può essere molto esteso, dunque la tabella delle pagine risulterebbe molto grande.

Struttura indirizzo logico

p1	p2	d

p1: indice di pagina nella tabella esterna
p2: offset cella nella tabella interna
d: offset cella all'interno della pagina fisica



Pro:	Contro
▪ possibilità di indirizzare spazi logici di dimensioni elevate ▪ possibilità di mantenere in memoria soltanto le tabelle utili	▪ tempo di accesso più elevato (num accessi pari al numero di livelli di paginazione)

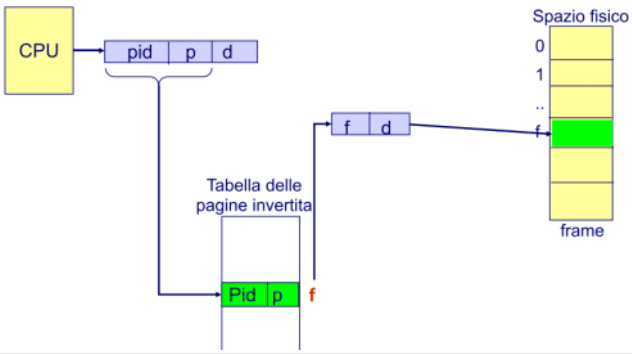
Tabella delle pagine invertita

Per limitare l'occupazione di memoria si una un'unica struttura dati globale chiamata **tabella delle pagine invertita** che ha un elemento per ogni frame.

Struttura indirizzo logico

pid	p	d

pid: identificatore del processo a cui è assegnato il frame
p: numero di pagina logica
d: offset cella all'interno della pagina fisica



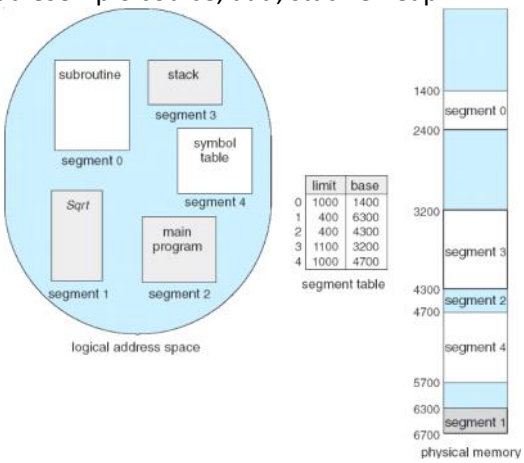
Pro:	Contro
	▪ tempo di ricerca elevato nella tabella invertita ▪ Non si può associare uno stesso frame a pagine logiche di diversi processi

Gestione della memoria

giovedì 18 aprile 2019 18.47

- ## Segmentazione

La segmentazione di basa sul partizionamento dello spazio logico degli indirizzi di un processo in parti (**segmenti**):
Vi è una divisione semantica, ad esempio codice, dati, stack e heap.



Pro:	Contro:
• migliore per motivi di protezione	▪ frammentazione esterna Soluzione: ▪ allocazione dei segementi con tecniche best fit, worst fit, first fit… ▪ compattazione

Struttura indirizzo logico

s	d

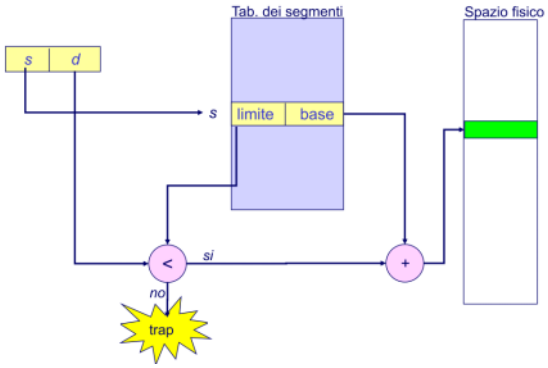
s: segmento (numero che individua il segmento nel sistema)
d: offset cella al'interno del segmento

Tabella dei segmenti

Il binding tra indirizzi logici e fisici viene realizzato mediante la **tabella dei segmenti**. Essa ha una entry per ogni segmento che ne descrive l'allocazione in memori fisica mediante la coppia:

base	limite

base: indirizzo prma cella del semento nello spazio fisico
limite: dimensione del segmento



La tabella dei segmenti è indirizzata dal registro **STBR** (Segment Table Base Register).
Il registro **STLR** (Segment Table Length Register) contiene il numero degli elementi validi nella tablella dei segmenti.

◇ Segmentazione paginata

Segmentazione e paginazione possono essere combinate (es. Intel x86) :

- spazio logico segmentato
- ogni segmento suddviso in pagine (in Linux la paginazione è a 3 livelli)

Così facendo si combinano i vantaggi dei due metodi:
eliminazione di frammentazione esterna e possibilità di caricare in memoria solo le pagine necessarie di un segmento

Strutture dati:

- tabella dei segnemti
- una tabella delle pagine per ogni segmento

Gestione della memoria

martedì 16 aprile 2019 15.56

Problematica: In generale la memoria disponibile può non essere sufficiente ad accogliere codice e dati di un processo.

Overlay

La tecnica di overlay ha l'obiettivo di mantenere in memoria istruzioni e dati che:

- vengono utilizzati più frequentemente
- sono necessari nella fase corrente

allocando/deallocando le relative parti dello spazio di indirizzi. La realizzazione è a carico del programmatore.

Memoria virtuale

E' un modo di gestione della memoria che consente l'esecuzione di processi non completametne allocati in memoria.
Vantaggi:

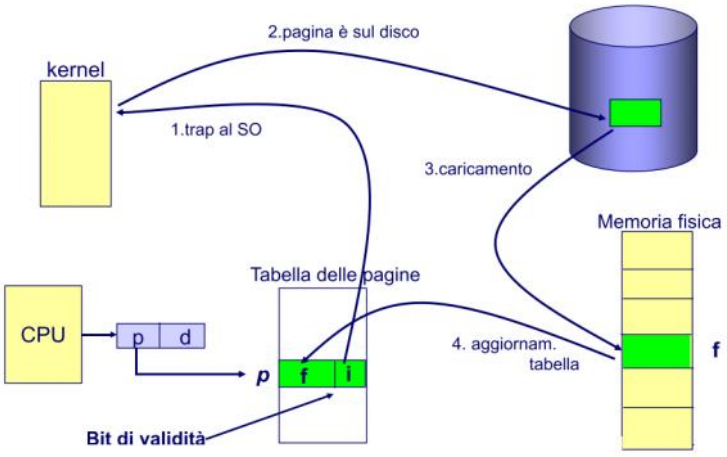
- dimensione spazio logico non vincolata dall'estensione della memoria
- grado di multiprogrammazione indipendente dalla dimensione della memoria fisica
- efficientza nel caricamento e swapping di un processo
- astrazione

Di solito la memoria virtuale è realizzata mediante tecniche di **paginazione su richiesta**, ossia tutte le pagine di un processo risiedono in memoria di massa, durante l'esecuzione alcune vengono trasferite in memoria centrale dal **pager** (un modulo del SO che funge swapper di singole pagine). Per la realizzazione sono necessarie tabella delle pagine e memoria secondaria.
Si utilizza il **bit di validità** nella tabella delle pagine per distinguere se la pagina:

- (1) è in memoria centrale
- (0) è in memoria secondaria oppure è invalida, cioè non appartiene allo spazio logico del processo.

Se la pagina richiesta ha bit di validità a 0 viene generata un'interruzione di **page fault** al SO. Il kernel, alla ricezione dell'interruzione:

1. Salva il contesto di esecuzione del processo (registri, stato, tabella delle pagine)
2. Verifica il motivo del page fault mediante una tabella interna al kernel
   - riferimento illegale → terminazione del processo
   - riferimento legale → la pagina risiede in memoria secondaria
3. Copia la pagina in un frame libero
4. Aggiorna la tabella delle pagine
5. Ripristina il processo all'esecuzione interrotta dal page fault



Problematica: Potrebbe succedere che non ci siano frame liberi per caricare la pagina in memoria centrale (**sovrallocazione**).

Una possibile soluzione può esere la sostituzione di una pagina vittima P_{vitt} allocata in memoria centrale con la pagina P_{new} da caricare. I metodi per decidere quale pagina rendere vittima sono svariati, per ottimizzare tale processo viene introdotto il **bit di modifica** (dirty bit) che individua se la pagina è stata modificata rispetto alla copia residente in memoria secondaria; in caso negativo è preferibile scegliere tale pagina poiché si può evitare di aggiornare il valore in memoria secondaria.

Altri algoritmi:

- LFU (Last Frequently Used) : viene individuata come vittima la pagina usata meno frequentemente
- FIFO: viene individuata come vittima la pagina caricata da più tempo in memoria
- LRU (Last Recently Used): viene individuata come vittima la pagina usata meno recentemente

Gestione della memoria

martedì 30 aprile 2019 09.26

Page fetching & trashing

La paginazione su domanda carica una pagina soltanto se strettamente necessario.
Ciò può portare al fenomeno di **trashing** ossia il processo impiega più tempo per la paginazione che per l'esecuzione.

Soluzione:
working set: è una tecnica basata su pre-paginazione, cioè si caricano precedentemente il set di pagine di cui il processo ha bisogno per la prossima fase di esecuzione.
Si è osservato infatti che un processo, in una certa fase di esecuzione usa un sottoinsieme piccolo delle sue pagine logiche e tale sottoinsieme varia lentamente nel tempo. In particolare si ha:
Località spaziale: alta probabilità di accedere a locazioni vicine
Località temporale: alta probabilità di accedere a locazioni accedute di recente (algoritmo LRU)
Viene solitamente usato un working set basato su località temporale mantenendo una **finestra** delle **Δ** pagine più recenti.
Data la dimensione corrente del working set WSS_i di ogni processo P_i, si definisce **richiesta totale di frame** $D = \sum_i WSS_i$.
Sia m il numero totale di frame liberi:

- $D < m$ può esserci spazio per l'allocazione di nuovi processi
- $D > m$ è necessario lo swapping di uno o più processi

UNIX

Prime versioni

Spasio logico segmentato con allocazione contigua dei sementi seguendo la politica di first fit.
Ogni 4 secondi lo swapper viene attivato per provvedere a swap-in e swap-out dei processi.

Da BSDv3 in poi

Segmentazione paginata con allocazione non contigua seguendo la politica di memoria virtuale tramite paginazione su richiesta. Tecnica di pre-paginazione ma anche con pagine non strettamente necessarie nei frame liberi (No working set).
Presenza della **core map** che descrive lo stato di allocazione dei frame.
La tecnica di sostituzione è un LRU modificato: ogni pagina ha un **bit di uso** inizializzato a 0, quando la pagina viene acceduta questo è settato a 1, nella fase di ricerca di una vittima se tale bit è a 1 viene riportato a 0, se è 0 allora la pagina viene selezionata come vittima. La sostituzione viene eseguita dal pager **pagedeamon** (pid=2) ogni 250ms o più.
La sostituzione viene attivata quando il numero totale di frame liberi è ritenuto insufficiente (< *lotsfree*).
Viene attivato lo swapper di processi quando il numero totale i frame liveri è minore di quelli necessari (< *minfree*)

MS Windows XP

Paginazione con clustering delle pagine (**page cluster**): in caso di page fault, viene caricato tutto un gruppo di pagine attorno a quella mancante.
Ogni processo ha:

- un working set minimo (numero minimo di pagine sicuramente mantenute in memoria)
- un working set massimo (massimo numero di pagine mantenibile in memoria)

Qualora la memoria fisica libera scenda sotto una soglia, il SO automaticamente ristabilisce la quota desiderata di frame liberi (working set trimming), che elimina pagine di processi che ne hanno in eccesso rispetto a working set minimo.

Introduzione a UNIX shell e file comandi

venerdì 12 aprile 2019 09.18

La shell è un programma che permette di far interagine l'utente con il SO tramite comandi.
E' un interprete di comandi evoluto.

Per ogni comando la shell genera un processo figlio dedicato alla sua esecuzione.
Il processo padre attende la terminazione del comando (se l'esecuzione è in *foreground*) oppure prosegue in parallelo (se l'esecuzione è in *background* : il comando termina con '&').

Comandi shell Linux: filtri

`grep <testo> [<file>...]`	Ricerca di testo
`tee <file>`	Scrivere l'input sia su file che su canale di output
`sort [<file>]`	Ordina alfabeticamente le linee
`rev <file>`	Inverte l'ordine delle linee di un file
`cut [-options] <file>`	Seleziona colonne da file

Redirizione di input output

Ridirezione dell'input:

```
comando < file_input
```

Ridirezione dell'output:

```
comando > file_output          (nuovo o sovrascritto)
comando >> file_output         (append)
```

Piping

L'output di un comando può esser diretto a diventare l'input di un altro comando:

```
comando1 | comando2
```

Comando awk

E' un comando per la ricerca di testo ed esecuzione di azioni

```
awk '<pattern> {action}' [<file>...]
```

Esempi:

`'/str/ {print}'`	Tutte le linee che contengono 'str'
`'/^In/ {print}'`	Tutte le linee che cominciano con 'In'
`'/^In/ {print $2}'`	Stampa solo la seconda parola per ogni linea che comincia con 'In' assumendo come carattere delimitatore lo spazio
`-F';' {print $4}`	Stampa il quarto campo usando ';' come delimitatore

Variabili nella shell

Le variabilii sono stringhe. I riferimenti delle variabili si fanno con l carattere speciale '$'.

```
X=1 + 3        #niente spazi nell'assegnazione
echo $X
1 + 3
X=`expr 1 + 3`
echo $X
4
```

Introduzione a UNIX shell e file comandi

lunedì 15 aprile 2019 23.07

Metacaratteri

La shell riconosce dei caratteri speciali

*	una qualunque stringa di zero o più caratteri in un nome di file
?	un qualunque carattere in un nome di file
[a-d,z,f,c]	un qualunque carattere tra quelli nell'insieme
#	commento
#!	direttiva di interprete
\	carattere di escape: non viene interpretato il carattere successivo
' '	impediscono l'espansione
" "	impediscono l'espansione con l'eccezione di $, \ e ```

Parsing della shell

Ridirezione output → sostituzione dei comandi → sostituzione di variabili e parametri → sostituzione metacaratteri.

Comando set

Il comando set a linea di comando permette di visualizzare le variabili d'ambiente

File comandi

La prima riga deve specificare quale shell si vuole utilizzare:

```
#!/bin/bash
```

Gli argomenti passati al programma sono variabili posizionali $0, $1, $2, …

Comando shift

il comando shift fa scorrere verso sinistra tutti i paramentri del programma tranne $0

Comando set

Il comando set nel file bash permette di riassegnare i valori agli argomenti:

```
set exp1 exp2 exp3
```

Comando read

Il comando read serve a leggere qualcosa da standard input

```
read var
```

$*	insieme di tutte le variabili posizionali
$#	numero di argomenti passati ($0 è escluso)
$?	valore int restituito dall'ultimo comando eseguito
$$	pid

Introduzione a UNIX shell e file comandi

lunedì 15 aprile 2019 23.52

Comando test

Il comando test permette la valutazione di un'espressione (viene generata una shell figlia che fa una exec)

```
test [expression]
```

valore = 0 → true
valore ≠ 0 → false

```
test -f <nomefile>              esistenza di file
test -d <nomefile>              esistenza di directory
test -r | -w | -x <nomefile>    diretto di lettura/scrittura/esecuzione sul file

test -z <stringa>               vero se stringa nulla
test <stringa>                  vero se stringa non nulla
test <stringa1>  = <stringa2>   uguaglianza tra stringhe
test <stringa1> != <stringa2>   disuguaglianza tra stringhe

test <val1> -gt <val2>          >
test <val1> -lt <val2>          <
test <val1> -ge <val2>          >=
test <val1> -le <val2>          <=

test <cond1> -a <cond2>         and
test <cond1> -o <cond2>         or
```

In alternativa al comando test è possibile utilizzare le parentesi, a patto di mantenere gli spazi a inizio e fine espressione

```
[ ... ]
```

Un'altra possibile alternativa sono le doppie parentesi, le quali impongono però che la condizione venga valutata nella shell stessa. Gli spazi sono comunque significativi

```
[[ ... ]]
```

I vantaggi sono:

- prestazioni
- possibilità di usare gli operatori logici && , ||
- possibilita di usare > e < ma solo per confrontare stringhe, non per redirezioni I/O

Struttura di controllo if

```
if <condition>
then
    ...
[elif <condition>
then
    ...]
[else
    ...]
fi
```

```
if <condition>; then
    ...
[elif <condition>; then
    ...]
[else
    ...]
fi
```

Alternativa multipla

```
case <var> in
    <pattern-1>)
        ...
    ;;
    ...
    <pattern-n>)
        ...
    ;;
esac
```

Introduzione a UNIX shell e file comandi

martedì 16 aprile 2019 00.26

Ciclo for

```
for <var> [in <list>]
do
    ...
done
```

Ciclo while

```
while <condizione>
do
    ...
done
```

Ciclo until

Condizione opposta a quella del while

```
until <condizione>
do
    ...
done
```

Programmaz. concorrente - Modello ad Ambiente Globale

martedì 30 aprile 2019 10.35

Se la macchina è organizzata secondo il modello ad ambiente globale (o a memoria comune) allora:

- processo = thread
- comunicazione attraverso risorse comuni (condivise)
- necessità di sincronizzare gli oggetti condivisi (meccanismo di controllo degli accessi)

Una **risorsa** è un qualunque oggetto, fisico o logico, di cui un processo necessita per portare a termine il suo compito.
La regola di **mutua esclusione** impone che le operazioni con le quali i processi accedono alle risorse comuni non si sovrappongano nel tempo (altrimenti deadlock).
La sequenza di istruzioni con le quali un processo accede e modifica un insieme di variabili comuni prende il nome di **sezione critica**.

Deadlock

E' importante che una sola sezione critica di una classe sia in esecuzione ad ogni istante
Un insieme di processi è in **deadlock** se ogni processo dell'insieme è in attesa di un evento che può essere causato solo da un altro processo dell'insieme (fig.). Per descrivere il deadlock si utilizzano i **grafi di allocazione** delle risorse.

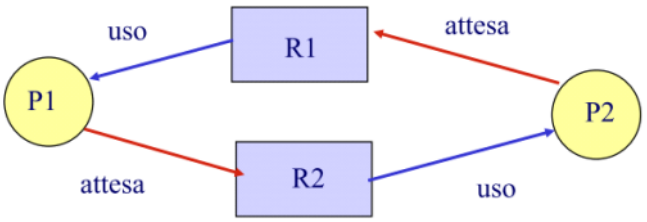


fig.

Dato un insieme di processi, essi si trovano in uno stato di deadlock se e solo se sono verificate 4 condizioni:

- mutua esclusione: le risorse sono utilizzate in modo mutuamente esclusivo
- possesso e attesa: ogni processo che possiede una risorsa può richiederne un'altra
- impossibilità di interazione: una volta assegnata ad un processo, una risorsa non può essere sottratta (no preemption)
- attesa circolante: fig.

soluzioni

- prevenzione: si evita a priori il verificarsi di situazioni di blocco critico:
  - *prevenzione statica*: si impongono vincoli sulla struttura che garantiscano a priori il che una delle condizioni necessarie per il deadlock non si verifichi.
  - *prevenzione dinamica*: ogni volta che un processo richiede una risorsa il SO verifica se l'allocazione può portare a situazioni di blocco critico e in tal caso mette in attesa il processo. Un esempio è l'algoritmo del banchiere (Dijkstra, 1965): l'obiettivo è individuare una sequenza di processi salva tra tutte le possibili.

	necessità max	uso attuale	possibili richieste
P0	10	5	5
P1	4	2	2
P2	9	2	7

    Sequenza salva: {P1, P0, P2}
- rilevazione: se il SO rileva la presenza di deadlock avvia un algoritmo di ripristino (rollback)

Programmaz. concorrente - Modello ad Ambiente Globale

venerdì 3 maggio 2019 09.47

Mutua esclusione e soluzioni

Una corretta soluzione al problema della mutua esclusione deve soddisfare i seguenti requisiti:

1. Sezioni critiche della stessa classe devono essere eseguite in modo mutuamente esclusivo.
2. Quando un processo si trova all'esterno di una sezione critica non può rendere impossibile l'accesso alla stessa sezione (o a sezioni della stessa classe) ad altri processi.
3. Assenza di deadlock.

Prologo : Serie di istruzioni precendenti alla sezione critica che garantiscono l'accesso esclusivo alla risorsa se questa libera, oppure ne impediscono l'accesso e mettono il processo in attesa se questa è già occupata.

Epilogo : Sequenza di istruzioni successive alla sezione critica che servono per dichiarare libera la stessa.

- ### Soluzioni algoritmiche

  Sfrutta la possibilità di condivizione di variabili. Si realizza l'attesa con dei cicli dunque durante l'attesa si usa inutilmente la CPU: *attesa attiva* (busy waiting).

  Algoritmo di Dekker

```
int busy1 = 0;
int busy2 = 0;
int turno = 1;
```

```
/* processo P1: */
main() {
...
      busy1 = 1;
      while(busy2 == 1){
            if (turno == 2) {
                  busy1 = 0;
                  while(turno != 1);
                  busy1 = 1;
            }
      }
      <sezione critica A>;
      turno = 2;
      busy1 = 0;
...
}
```

```
/* processo P2: */
main() {
...
      busy2 = 1;
      while(busy1 == 1){
            if (turno == 1) {
                  busy2 = 0;
                  while(turno != 2);
                  busy2 = 1;
            }
      }
      <sezione critica A>;
      turno = 1;
      busy2 = 0;
...
}
```

  Algoritmo di Peterson

```
int busy1 = 0;
int busy2 = 0;
int turno = 1;
```

```
/* processo P1: */
main() {
...
      busy1 = 1;
      turno = 2;
      while (busy2 && turno == 2);
      <sezione critica A>;
      busy1 = 0;
...
}
```

```
/* processo P2: */
main() {
...
      busy2 = 1;
      turno = 1;
      while (busy1 && turno == 1);
      <sezione critica A>;
      busy2 = 0;
...
}
```

  problematica:

  Attesa attiva (**busy waiting**): Durante l'attesa ogni processo usa inutilmente la CPU

Programmaz. concorrente - Modello ad Ambiente Globale

martedì 7 maggio 2019 09.30

- Soluzioni Hardware-based

  Il supporto è fornito dall'architettura HW (es. disabilitazione delle interruzioni, lock/unlock):

  1. Molti processori prevedono istruzioni che consentono di esaminare e modificare il contenuto di una parola in un unico ciclo (es: test-and-set)

```
int test-and-set(int *a) {
      int R;
      R=*a;
      *a=0;
      return R;
}
```

  2. Mediante l'istruzione test-and-set è possibile realizzare il meccanismo di lock / unlock:

```
void lock(int *x) {
      while (!*x);
      *x=0;
}
void unlock(int *x) {
      *x=1;
}
```

già realizzate in linguaggio macchina

```
lock(x):
      tsl register, x
      cmp register, 1
      jne lock
      ret
unlock(x):
      move x,1
      ret
```

```
/* processo P1: */
main() {
...
      lock(&x);
      <sezione critica A>;
      unlock(&x);
...
}
```

```
/* processo P2: */
main() {
...
      lock(&x);
      <sezione critica B>;
      unlock(&x);
...
}
```

Programmaz. concorrente - Modello ad Ambiente Globale

domenica 12 maggio 2019 12.51

- Soluzioni basate su strumenti di sincronizzazione

Prologo ed epilogo sfruttano strumenti software per la sincronizzazione realizzati dal nucleo del sistema operativo:

Semaforo

Un semaforo è un dato astratto rappresentato da un intero positivo, al quale è possibile accedere soltanto tramite le due operazioni atomiche:

p(s): while(!s); s--;	v(s): s++;

Dato un semaforo S:

- Se il valore di S è 0: l'esecuzione di p provocherà l'attesa del processo che la invoca (semaforo ROSSO)
- Se il valore di S è maggiore di 0: la chiamata di p provocherà il decremento di S e la continuazione dell'esecuzione (semaforo VERDE).

Esempio realizzazione

```
typedef struct{
      unsigned int value;
      queue Qs;
      int lock = 1;
} semaphore;

void p(semaphore *s) {
      lock(s-> lock);
      if( s->value == 0 ){
            unlock(s->lock);
            <sospensione processo in s->Qs>
            lock(s->lock);
      }else
            s->value--;
      unlock(s->lock);
}

void v(semaphore *s) {
      lock(s->lock);
      if (!empty(s->Qs))
            <il descrittore del primo processo viene rimosso dalla coda ed il suo stato modificato in pronto>
      else
            s->value++;
      unlock(s->lock);
}
```

```
semaphore mutex;
mutex.value = 1;
```

```
/* processo P1: */
...
      p(&mutex);
      <sezione critica>;
      v(&mutex);
...
```

```
/* processo P2: */
...
      p(&mutex);
      <sezione critica>;
      v(&mutex);
...
```

Esempio: caso produttore-consumatore

Consideriamo il caso di un buffer condiviso tra due processi: un produttore e un consumatore. Per la realizzazione occorrono 2 semafori:

- `spazio_disp` (v.i. n, num. elementi liberi nel buffer)
- `msg_disp` (v.i. 0, num. messaggi presenti nel buffer)

Il produttore invoca `p(&spazio_disp)`, decrementando così il numero di spazi liberi ad ogni inserimento fino a bloccarsi quando arriva a 0, e invoca `v(&msg_disp)` ad ogni inserimento sbloccando il semaforo al consumatore. Viceversa il consumatore chiamerà `p(&msg_disp)` per consumare la risorsa precedentemente prodotta e successivamente invocherà `v(&spazio_disp)` liberando lo spazio al produttore per altri inserimenti.

Nel caso di più produttori e cosumatori è necessario garantire che:

- non più di un produttore alla volta acceda al buffer (mutex_P)
- non più di un consumatore alla volta acceda al buffer (mutex_C)

Thread in Java - pt. 1

venerdì 3 maggio 2019 10.34

Realizzazione

Due modalità per implementare i thread in Java:

1. estendendo la classe java.lang.*Thread*

```
class SimpleThread extends Thread {
      public void SimpleThread() { <costruttore> }
      public void run() {
            <sequenza di istruzioni eseguita da ogni thread di questa classe>
      }
}

public class EsempioConDueThreads {
      public static void main (string[] args) {
            SimpleThread t1=new SimpleThread();
            t1.start();
      }
}
```

2. implementando l'interfaccia java.lang.*Runnable*

```
class EsempioRunnable extends MiaClasse implements Runnable {
      public void run() {
            <sequenza di istruzioni eseguita da ogni thread di questa classe>
      }
}
public class Esempio {
      public static void main(String args[]) {
            EsempioRunnable e=new EsempioRunnable();
            Thread t=new Thread(e);
            t.start();
      }
}
```

Sincronizzazione

Differenti thread condividono lo stesso spazio di memoria (heap). Servono meccanismi di sincronizzazione per impedire che più thread possano accedere contemporaneamente allo stesso oggetto. Se ne distunguono due tipologie:

- Interazione di tipo competitivo (mutua esclusione)

  In java ad ogni oggetto viene associato automaticamente dalla JVM un lock, che rappresenta lo stato dell'oggetto (libero / occupato). E' possibile denotare le sezioni critiche tramite la parola chiave **synchronized**.

```
public class contatore {
      private int i=0;
      public synchronized void incrementa() {
            i++;
      }
      public synchronized void decrementa() {
            i--;
      }
}
```

  Se un thread tenta di eseguire un'operrazione synchronized su un oggetto che attualmente è in uso, il thread viene aggiunto ad una coda detta **entry set**. Al termine della sezione critica, se non ci sono thread in attesa, il lock viene reso libero mentre, se ci sono thread in attesa, il lock rimane occupato e viene scelto uno di questi nella entry set.

Thread in Java - pt. 1

domenica 12 maggio 2019 10.34

- ## Interazione di ripo cooperativo (Semafori)

  Dalla versione 5.0, in Java è disponibile la classe java.util.concurrent.*Semaphore* tramite la quale si possono creare semafori e operare tramite i metodi:

  - `acquire();` `// implementazione di p()`
  - `release();` `// implementazione di v()`

```
public class ThreadP extends Thread{ //produttori
	int i = 0;
	Risorsa r; //buffer condiviso
	public threadP(Risorsa R){
		this.r = R;
	}
	public void run(){
		try{
			System.out.print("Thread PRODUTTORE) il mio ID è: " + getName() + "\n");
			while(i < 100){
				sleep(100);
				r.inserimento(i);
				i++;
			}
		}catch(InterruptedException e){}
	}
}
```

```
public class ThreadC extends Thread{ //consumatori
	int msg;
	Risorsa r;
	public threadC(Risorsa R){
		this.r = R;
	}
	public void run(){
		try{
			System.out.print("Thread CONSUMATORE) il mio ID è: " + getName() + "\n");
			while(true){
				msg = r.prelievo();
			}
		}catch(InterruptedException e){}
	}
}
```

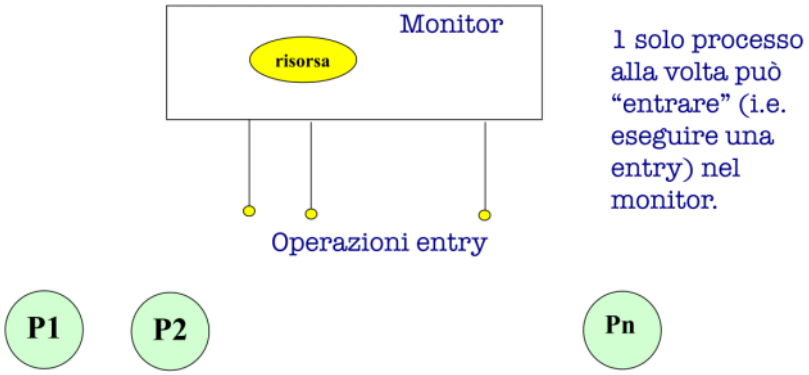
```
import java.util.concurrent.Semaphore;
public class Risorsa {
	final int N = 30;
	int lettura, scrittura;
	int[] buffer;
	Semaphore sP; /* sospensione dei Produttori   v.i. N */
	Semaphore sC; /* sospensione dei Consumatori  v.i. 0 */
	Semaphore sM; /* semaforo di mutua esclusione v.i. 1 */

	public risorsa(){
		lettura = 0;
		scrittura = 0;
		buffer = new int[N];
		sP = new Semaphore(N); /* v.i. N */
		sC = new Semaphore(0); /* v.i. 0 */
		sM = new Semaphore(1); /* v.i. 1 */
	}
	public void inserimento(int M){
		try{
			sP.acquire();   //se il valore associato al semaforo è > 0 allora decrementa e passa, altrimenti si blocca
			sM.acquire();
			buffer[scrittura] = M;
			scrittura = (scrittura + 1) % N; // num. di scritture se scritture != N, altriementi 0
			sM.release();
			sC.release();
		}catch(InterruptedException e){}
	}
	public int prelievo(){
		int messaggio = -1;
		try{
			sC.acquire();    //se il valore associato al semaforo è > 0 allora decrementa e passa, altimenti si blocca
			sM.acquire();
			messaggio = buffer[lettura];
			lettura = (lettura + 1) % N;
			sM.release();
			sP.release();
		}catch(InterruptedException e){}
		return messaggio;
	}
}
```

```
import java.util.concurrent.*;
public class prodcons{
	public static void main(String args[]){
		Risorsa R = new Risorsa();
		ThreadP TP = new threadP(R);
		ThreadC TC = new threadC(R);
		TC.start();
		TP.start();
	}
}
```

Monitor

martedì 14 maggio 2019 10.05

Il **monitor** è un oggetto utile per la programmazione concorrenziale. Le operazioni *entry* definite nel monitor sono mutuamente esclusive. Solitamente al monitor è associata una risorsa con lo scopo di poter controllare l'accesso a tale risorsa da parte di processi concorrenti, in accordo a determinate politiche.



Il monitor garantisce due livelli di sincronizzazione:

1. Garantische che un solo processo alla volta posso avere accesso al monitor (sospensione processi nell' entry queue).
2. Controlla l'ordine con il quale i processi hanno accesso alla risorsa in base ad una condizione logica di sincronizzazione. Tale logica può essere definita attraverso un nuovo strumento chiamato variabile condizione

Variabile condizione

Una **variabile condizione** rappresenta una coda nella quale i processi possono sospendersi se la condizione di sincronizzazione non è verifiata.

```
condition cond;
```

E' possibile effettuare due operazioni sulle variabili condizione:

- `wait(cond)`
  Sospende il processo introducendolo nella coda `cond` e liberando il monitor.
  Al risveglio il processo riacquisisce l'accesso esclusivo al monitor e riprende l'esecuzione.
- `signal(cond)`
  Riattiva un processo in attesa nella coda `cond`, se non vi sono processi non produce effetti.

Come conseguenza della signal, entrambi i processi, quello segnalante Q e quello segnalato P, possono proseguire la loro esecuzione. Il monitor però limita a 1 il numero di processi che eseguono al suo interno.
Possibili strategie:

- signal_and_wait: P riprende immediatamente l'esecuzione e Q viene sospeso nella entry queue.
- signal_and_continue: Q prosegue l'esecuzione mentre P viene posto nella entry queue. Poiché altri processi prima di P (lo stesso Q) potrebbero modificare la condizione di sincronizzazione, tale condizione va ritestata. Occorre un while:

```
while(!B)
    wait(cond);
<accesso alla risorsa>
```

- signal_and_urgent_wait: P riprende immediatamente l'esecuzione e Q viene sospeso in una coda interna al monitor detta **urgent queue**.

Thread in Java - pt. 2

venerdì 17 maggio 2019 09.26

Wait set

In Java, il **wait set** è una coda di thread associata ad ogni oggetto ed inizialmente vuota.
I thread entrano e escono dal wait set utilizzando rispettivamente i metodi `wait()` e `notify()` che possono essere invocati solo all'interno di un blocco o di un metodo sincronizzati.

wait()

Comporta il rilascio della lock, la sospensione del thread ed il suo inserimento nel wait set.

notify()

Comporta l'estrazione di un threa dal wait set e il suo inserimento nell'entry set. Non provoca il rilascio del lock, il thread risvegliato deve attendere che il lock sia libero (signal&continue).

notifyAll()

Comporta l'estrazione di tutti i thread dal wait set ed il loro inserimento nell'entry set. Non provoca il rilascio del lock, i thread risvegliati devono attendere che il lock sia libero (signal&continue).

Esempio:

```
//Mailbox di capacita` N
public class Mailbox {
	private int[] contenuto;
	private int contatore, testa, coda;
	public mailbox(){ //costruttore
		contenuto = new int[N];
		contatore = 0;
		testa = 0;
		coda = 0;
	}
	public synchronized int preleva (){
		int elemento;
		while (contatore == 0)
			wait();
		elemento = contenuto[testa];
		testa = (testa + 1)%N;
		--contatore;
		notifyAll();
		return elemento;
	}
	public synchronized void deposita (int valore){
		while (contatore == N)
			wait();
		contenuto[coda] = valore;
		coda = (coda + 1)%N;
		++contatore;
		notifyAll();
	}
}
```

Nelle versioni più receti i Java è possibile utilizzare le variabili condizione mediante l'interfaccia definita in `java.util.concurrent.lock`

```
public interface Condition{
    void await() throws InterruptedException;
    void signal();
    void signalAll();
}
```

```
public interface Lock{
    void lock();
    void unlock();
    Condition newCondition();
}
```

Esempio:

```
Lock lockvar = new Reentrantlock();   //Reentrantlock è un classe che implementa Lock
Condition C = lockvar.newCondition(); //C è una condition associata al lock lockvar
```

Java non offre un costrutto equivalente al monitor, ma con gli strumenti vistri possiamo definire classi che implementano il concetto di monitor.

Thread in Java - pt. 2

venerdì 17 maggio 2019 10.04

Esempio di gestione di un buffer circolante con meccanismo di monitor:

```
public class Mailbox{
	//Dati:
	private int[] contenuto;
	private int contatore, testa,coda;
	private Lock lock = new ReentrantLock();
	private Condition non_pieno = lock.newCondition();
	private Condition non_vuoto = lock.newCondition();

	//Costruttore:
	public Mailbox(){
		contenuto=new int[N];
		contatore=0;
		testa=0;
		coda=0;
	}

	//metodi "entry":

	public int preleva() throws InterruptedException{
		int elemento;
		lock.lock();
		try{
			while(contatore== 0)
				non_vuoto.await();
			elemento=contenuto[testa];
			testa=(testa+1)%N;
			--contatore;
			non_pieno.signal();
		}finally{
			lock.unlock();
		}
		return elemento;
	}

	public void deposita(int valore) throws InterruptedException{
		lock.lock();
		try{
			while (contatore==N)
				non_pieno.await();
			contenuto[coda] = valore;
			coda=(coda+1)%N;
			++contatore;
			non_vuoto.signal();
		} finally{
			lock.unlock();
		}
	}
}
```

```
public class Produttore extends Thread {
	int messaggio;
	Mailbox m;
	public Produttore(Mailbox M){
		this.m = M;
	}
	public void run(){
		while(1) {
			<produci messaggio>
			m.deposita(messaggio);
		}
	}
}
```

```
public class Consumatore extends Thread{
	int messaggio;
	Mailbox m;
	public Consumatore(Mailbox M){
		this.m = M;
	}
	public void run(){
		while(1){
			messaggio = m.preleva();
			<consuma messaggio>
		}
	}
}
```

```
public class BufferTest{
	public static void main(String args[]){
		Mailbox M = new Mailbox();
		Consumatore C = new Consumatore(M);
		Produttore P = new Produttore(M);
		C.start();
		P.start();
		...
	}
}
```

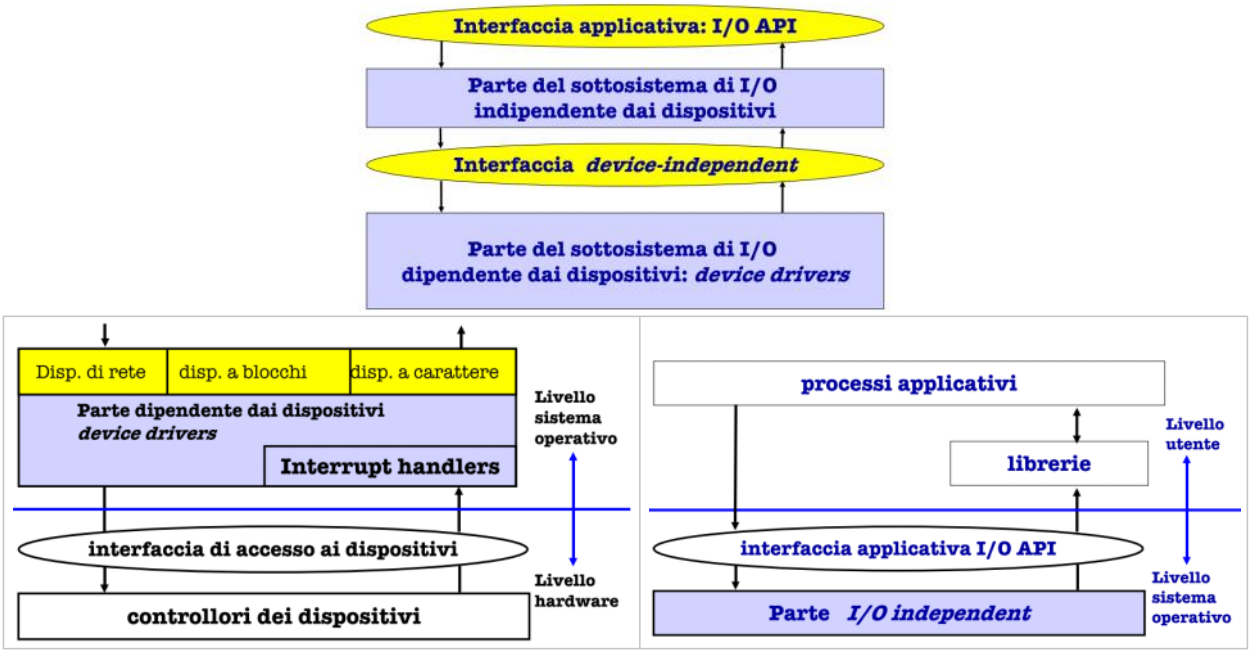
Gestione I/O

martedì 21 maggio 2019 09.45

Compiti del sottosistema di I/O

- *Nascondere* al programmatore i dettagli delle interfacce hardware dei dispositivi
- *Omogeneizzare* la gestione di dispositivi diversi
- *Gestire i malfunzionamenti* che si possono verificare durante un trasferimento di dati
- Definire lo spazio dei nomi (***naming***) con cui vengono identificati i dispositivi
- Garantire la corretta *sincronizzazione* tra ogni processo applicativo che ha attivato un trasferimento dati e l'attività del dispositivo (è possibile gestire il trasferimento in maniera sincrona o asincrona).

Architettura sottositema di I/O

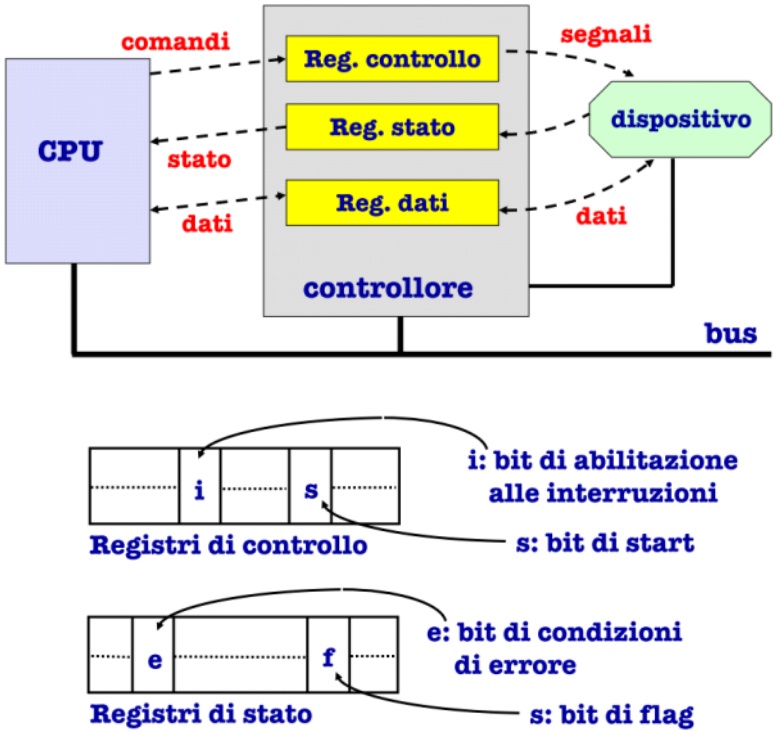


Buffering

Per ogni operazione di I/O il sistema operativo riserva un'area di memoria "tampone" (buffer), per contenere i dati oggetto del trasferimento. Le motivazioni di tale strategia sono:

- differenza di velocità tra processo e periferica (disaccoppiamento)
- quantità di dati da trasferire

Controllori dei dispositivi



Registro di controllo:
Viene scritto dalla CPU per controllare la periferica; ogni comando da impartire alla periferica viene scritto, tramite il driver, nel registro di controllo.
La periferica viene attivata tramite il settaggio del bit di start.

Registro di stato:
La periferica usa il registro di stato per comunicare l'esito di ogni comando eseguito ed, eventualmente, per notificare eventuali errori occorsi.
Il termine del comando viene notificato tramite il bit di flag.

Registro Dati:
Viene utilizzato per il trasferimento dei dati letti/scritti dal dispositivo

Gestione I/O

venerdì 24 maggio 2019 10.02

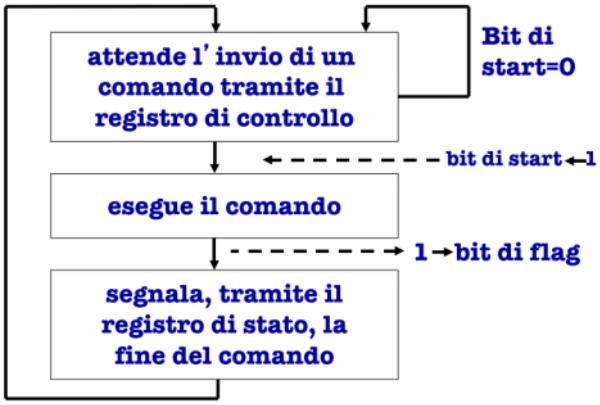
Modalità di gestione di un dispositivo

La sincronizzazione tra CPU e periferica può avvenire secondo 2 modelli:

- gestione a controllo di programma (o **polling**)
- gestione basata su **interruzioni**

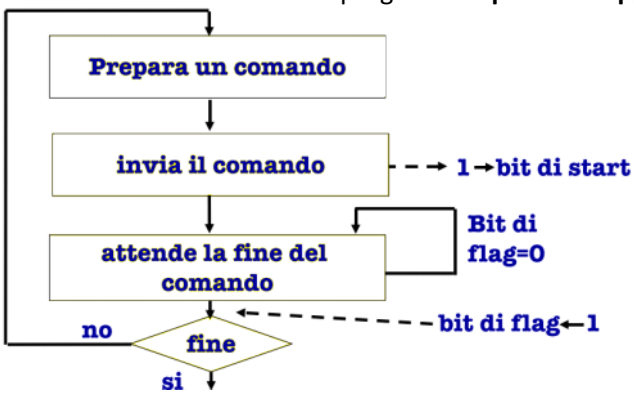
Polling

1. Gestione a controllo di programma: **processo esterno**




```
processo esterno // descrive l'attività del dispositivo
{
     while (true){
          do{;}while(start == 0); //stand-by
          <esegue il comando>;
          <registra l'esito del comando ponendo flag = 1>;
     }
}
```

2. Gestione a controllo di programma: **processo applicativo**




```
processo applicativo
{
.........................
     for (int i=0; i++; i<n) {
          <prepara il comando>;
          <invia il comando>;
          //ciclo di attesa attiva
          do{;}while(flag == 0);
          <verifica l'esito>;
     }
.........................
}
```

Interruzioni

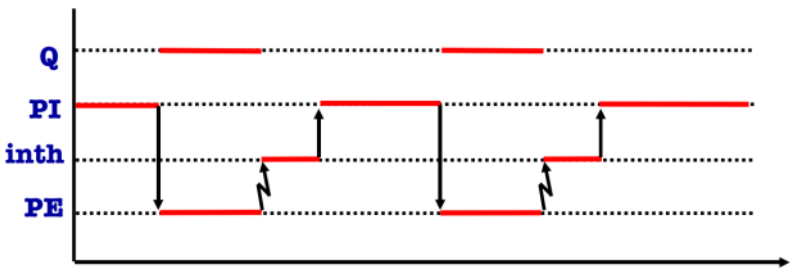
Lo schema "a controllo di programma" non è adatto per i sistemi multiprogrammati a causa dei cicli di attesa attiva. Per evitare l'attesa attiva occorre riservare per ogni dispositivo un semaforo:

```
semaphore dato_disponibile = 0;
```

che esso può risvegliare attraverso la generazione di un'interruzione.

```
processo applicativo
{ ....

     for (int i=0; i++; i<n) {
          <prepara il comando>;
          <invia il comando>;
          p(dato_disponibile);
          <verifica l'esito>;
     }
  ....
}
```

```
interrupt_handler
{ ....
     v(dato_disponibile);
  ....
}
```

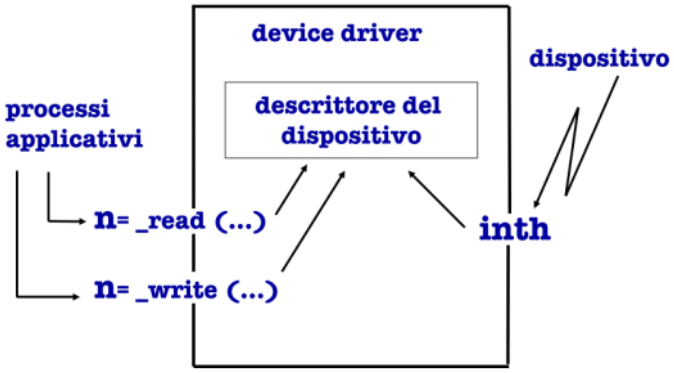


PI: processo applicativo che attiva il dispositivo
PE: processo esterno
Inth: routine di gestione interruzioni
Q: altro processo applicativo

Gestione I/O

martedì 28 maggio 2019 09.13

Astrazione di un dispositivo



indirizzo registro di controllo
indirizzo registro di stato
indirizzo registro dati
semaforo dato_disponibile
contatore dati da trasferire
puntatore al buffer in memoria
esito del trasferimento

Esempio caso lettura:

```
int _read(int disp, char* buf, int cont){
     descrittore[disp].contatore = cont;
     descrittore[disp].puntatore = buf;
     <attivazione dispositivo>;
     p(descrittore[disp].dato_disponibile);
     if(descrittore[disp].esito == <cod.errore>)
          return -1;
     return (cont - descittore[disp].contatore);
}

void inth(){
     char b;
     <legge il valore del registro di stato>;
     if (<bit di errore> == 0){
          < b = registro dati >;
          *(descrittore[disp].puntatore) = b;
          descrittore[disp].puntatore++;
          descrittore[disp].contatore--;
          if (descrittore[disp].contatore != 0){
               <riattivazione dispositivo>;
          }else{
               descrittore[disp].esito = <codice di terminazione corretta>;
               <disattivazione dispositivo>;
               v(descrittore[disp].dato_disponibile);
          }

     }else{
          if (<errore non recuperabile>){
               descrittore[disp].esito = <codice errore>;
               v(descrittore[disp].dato_disponibile);
          }
     }
     return;
}
```

Flusso di controllo durante un trasferimento

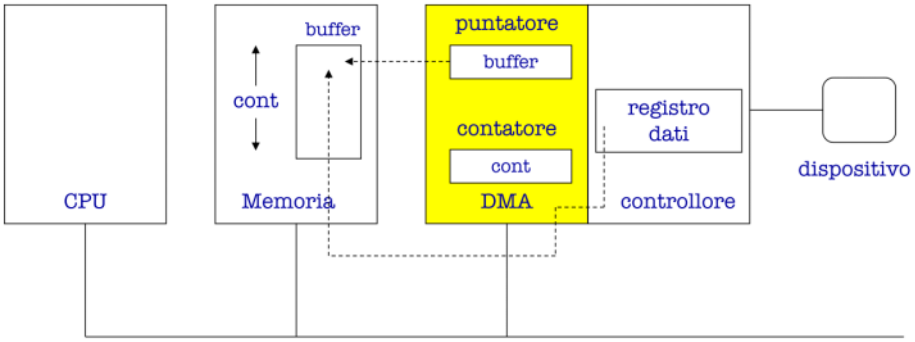
1. Il processo fa una system call al metodo read eseguibile in kernel mode al SO.
2. Nel metodo read viene chiamato il metodo _read dove il processo viene sospeso in attesa della risposta del dispositivo e posto nella coda del relativo semaforo.
3. L'interrupt handler inth, alla ricezione dell'interruzione, provvede a trasferire i dati sul buffer e a riattivare il processo.

Gestione I/O

martedì 28 maggio 2019 11.20

Gestione di un dispositivo in DMA

Il **DMA** (Direct Memory Access) è quel meccanismo che permette ad esempio alle periferiche di accedere direttamente alla memoria per scambiare dati senza coinvolgere l'unità di controllo per ogni byte.

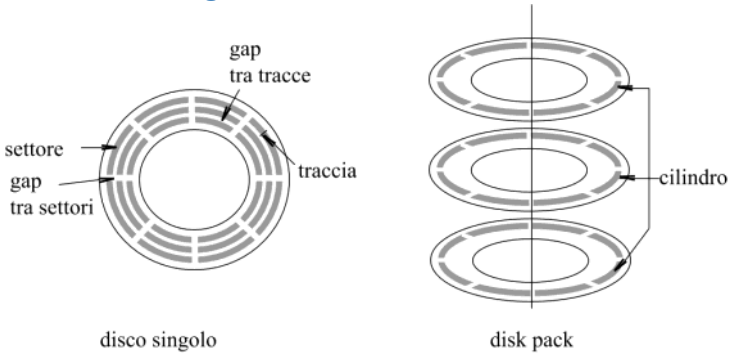


Un possibile caso che ci aiuta a capire l'utilità di questo meccanismo è la gestione di un dispositivo temporizzatore, il quale genera interruzioni periodiche a frequenze stabilite e sono utili per servizi quali:

- aggiornamento della data
- gestione del quanto di tempo (sistemi time-sharing)
- gestione delle system call alarm o sleep
- gestione del time-out (watchdog)

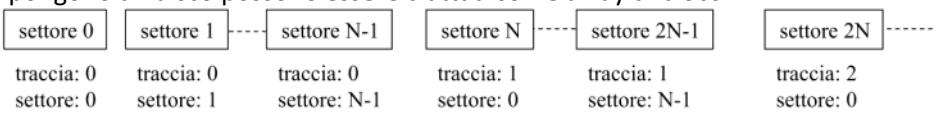
In tali casi non è necessario che la CPU gestisca tutte le interruzioni ricevute dal temporizzatore, ma sarà il DMA a notificarla una volta raggiunto il numero di interruzioni stabilito.

Gestione e organizzazione fisica dei dischi



Indirizzo di un settore (f, t, s):
f - faccia
t - traccia
s - settore

Tutti i settori che compongono un disco possono essere trattati come array di blocchi:



Indicando con M il numero di tracce per faccia e con N il numero di settori per traccia, l'idice i che rappresenta il blocco è:

$$i = f * M * N + t * N + s$$

Politiche di scheduling delle richieste

Il tempo medio di trasferimento su un settore è dato dalla formula:

$$TF = TA + TT = ST + RL + TT$$

Dove:

- TA - tempo medio di accesso (posizionamento della testina all'inizio del settore considerato) e comprende
  - ST - tempo di seek (posizionare la testina sopra la traccia)
  - RL - rotational latency (ruotare il settore per posizionarlo sotto la testina)
- TT - tempo di trasferimento dati del settore

Le richieste in coda ad un dispositivo possono essere servite secondo diverse politiche:

- First-Come-First-Served (**FCFS**)
- Shortest-Seek-Time-First **(SSTF**)
- **SCAN** algorithm -> La testina si porta ad una estremità del disco e si sposta verso l'altra estremità, servendo le richieste man mano che si raggiungono le tracce indicate ,quindi viene invertita la direzione (Minimizzazione ST).
- **C-SCAN** (Circular-SCAN) -> E' una variante dello SCAN, nel quale l'insieme delle tracce viene scansionato sempre nella stessa direzione in modo che le richieste più vecchie non siano lontane poiché una volta terminato il giro ricomincia da capo.