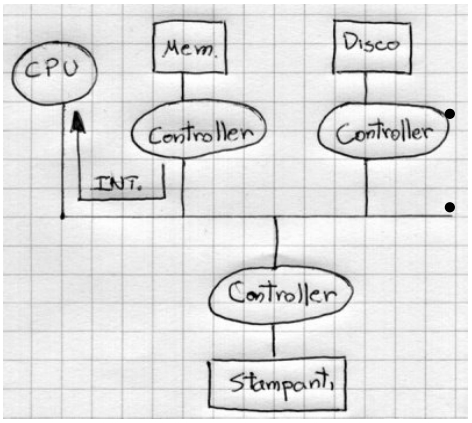
Appunti di sistemi operativi

Appunti per il corso universitario di sistemi operativi, riferito a sistemi Unix/Windows.
Si discute su problemi di sincronizzazione, memoria e scheduling dei processi.

ARGOMENTI

INPUT/OUTPUT

controller = processore dedicato all'I/O (si occupa di gestire un componente del calcolatore, sollevando quindi da tale compito la CPU)

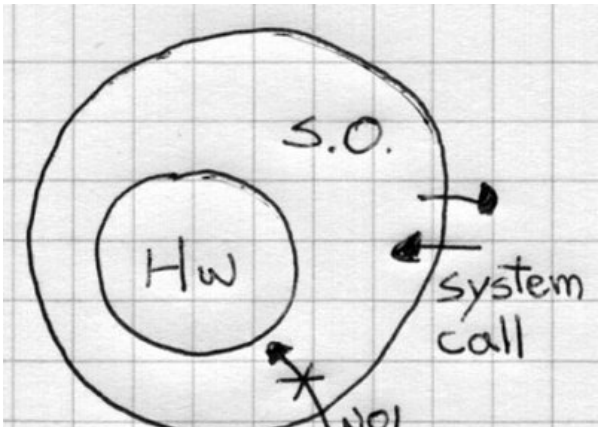


interrupt = segnale mandato alla CPU; può essere causato da:

- eventi **interni** (trap) = causati da istruzioni / errori -> eccezioni
- eventi **esterni**: es. dai controller di periferica -> interruzioni (vengono chiamati interrupt asincroni)

trap = chiamate al sistema operativo (al supervisore) delle istruzioni che richiedono un servizio dal S.O.

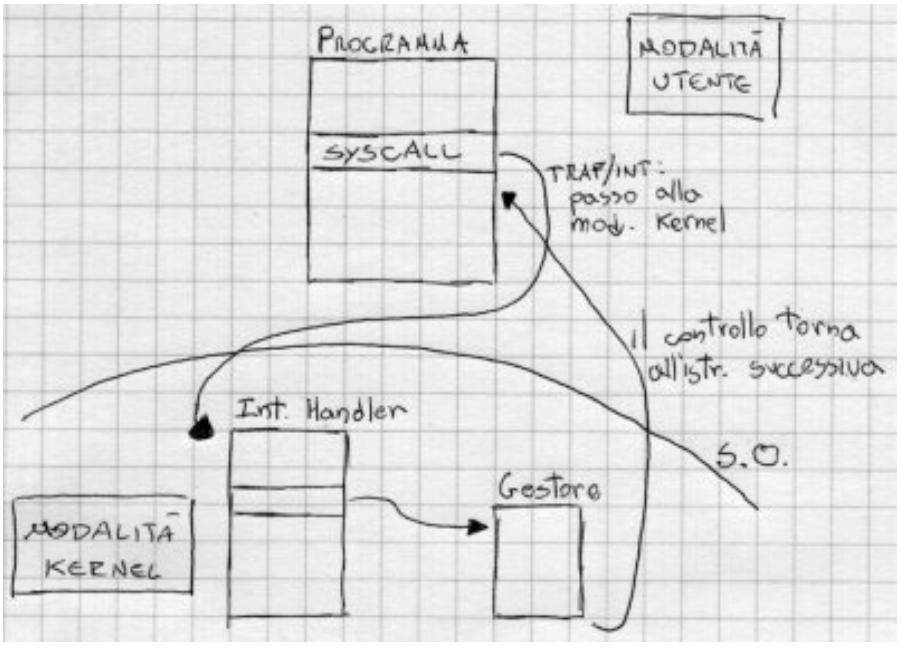
in un sistema ben progettato non possiamo fare I/O senza passare attraverso il sistema operativo; il motivo per cui viene fatta l'interruzione è per tenere separati i livelli programma e sistema operativo



Il programma è in **modalità utente**, nel momento in cui avviene un interruzione il processore passa automaticamente in **modalità kerne**l

in particolare, quando c'è un interruzione:

1. programma fa chiamata a sistema
2. viene generato un segnale hardware che fa passare la CPU in modalità kernel
3. viene eseguito l'handler
4. la CPU torna in modalità utente
5. continua l'esecuzione del programma



potrei farlo come una chiamata a procedura ma non avrei la stessa protezione (avveniva nel DOS: potevo comunicare con l'hardware senza passare per il S.O.)

Il meccanismo delle modalità serve a **proteggere le parti sensibili** del sistema: in modalità utente non si possono eseguire **istruzioni "privilegiate"**, ossia **istruzioni che possono causare danni allo stato del sistema**, come ad es. le istruzioni di I/O eseguite direttamente sull'hardware; in modalità kernel invece, posso eseguire tutto ciò che voglio

Fasi di una chiamata:

1. la prima cosa che l'hardware deve fare è **salvare** (lo copia) il PC (program counter), il PSW (program status word)
2. viene invocato l'handler (che sovrascrive il PC e il PSW) che:

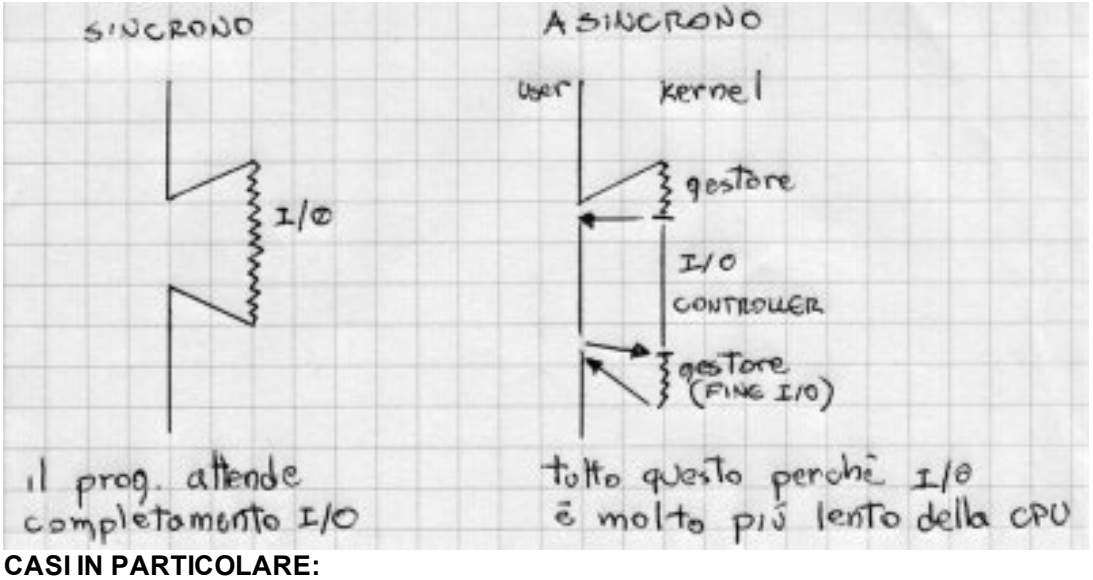
- salva i registri
- gestisce la syscall
- ripristina i registri
- invoca "fine interruzione" (istruzione macchina che ripristina PC e PSW e fa il fetch dell'istruzione successiva)

EVENTI ESTERNI

I/O sincrono = il programma attende il completamento dell'I/O

I/O asincrono:

1. normale esecuzione
2. parte trap
3. viene eseguito un gestore
4. parte l'I/O (nel frattempo l'esecuzione continua)
5. finisce l'I/O
6. parte un interrupt asincrono
7. viene eseguito un gestore di fine I/O
8. si ritorna all'esecuzione del programma principale



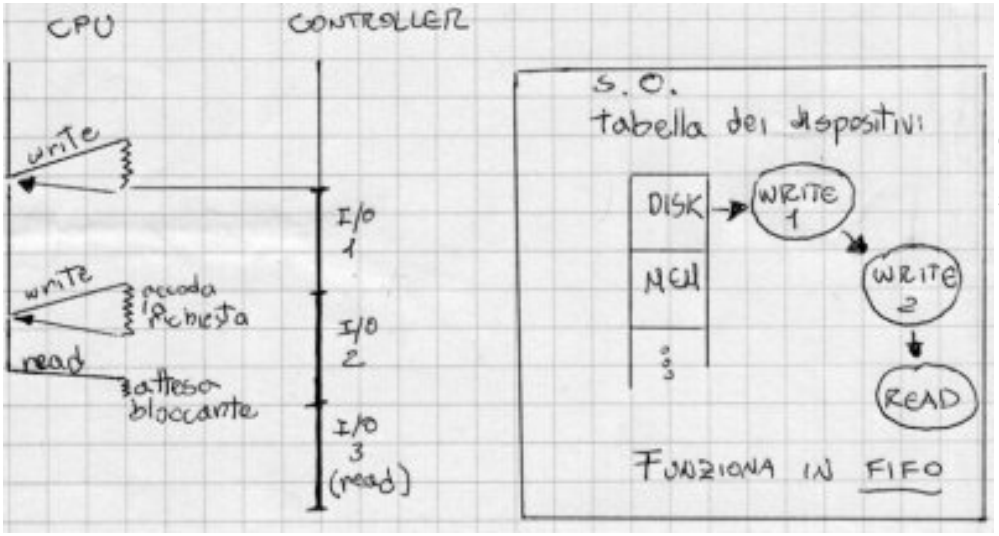
CASI IN PARTICOLARE:

es.

un programma esegue una write -> parte un gestore -> inizia l'I/O

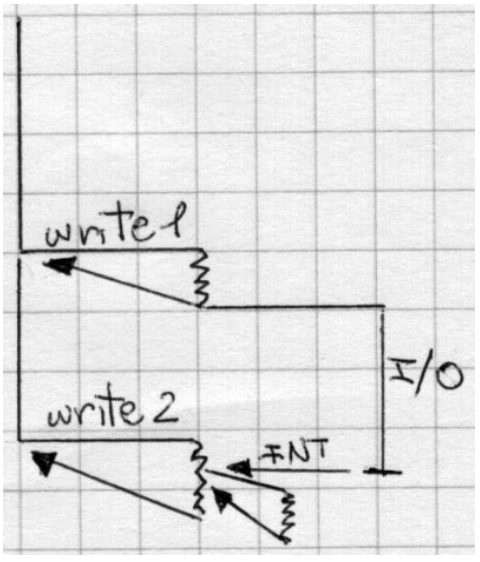
supponiamo che parta un'altra write: il controller è occupato -> mette la richiesta in una coda (FIFO) e continua il suo lavoro, le varie richieste verranno eseguite quindi in ordine

se parte una read sono costretto a fermare l'esecuzione e attendere che le istruzioni precedenti vengano completate -> perdo i vantaggi della gestione asincrona dell'I/O (è qui che ho bisogno della multi-programmazione, che mi permette di eseguire un altro job)



es. 2

sempre considerando due write che vengono eseguite in successione potrebbe esserci il rischio di sovrascrivere i valori salvati durante l'esecuzione della prima write; quindi abbiamo bisogno di uno **stack**, e anziché salvare i valori andremo ad eseguire delle push (e successivamente delle pop per ripristinarli), questo avviene in hardware !!



una soluzione per gestire strutture dati in modo corretto è dare **differenti priorità alle varie interruzioni** (buona per i sistemi **real-time**)

un'altra è la "**disabilitazione delle interruzioni**" = durante la gestione di un interruzione vengono disabilitate le altre (soluzione più semplice per i sistemi **all-purpose**)

quando arriva un'interruzione durante la gestione di un'altra interruzione, il S.O. o decide di metterla in coda (e in questo caso bisognerà cercare di ridurre il più possibile i tempi di gestione) oppure può gestire l'annidamento

questa seconda scelta crea problemi di coerenza (mutua esclusione sulle strutture dati del S.O.) -> devo avere meccanismi di sezione critica non con semafori (perché sto gestendo strutture dati del S.O. stesso) -> utilizzerò quindi busy waiting con istruzioni hardware (come test and set)

ARCHITETTURE DI PROTEZIONE

parte hardware che permette di proteggere il sistema (e l'esecuzione stessa dei programmi)

tecniche:

- differenziare le modalità di esecuzione della CPU
- porre dei **limiti alla zona di memoria che possiamo vedere**: abbiamo 2 registri, BASE e BOUND (l'accesso a questi è un istruzione privilegiata) che indicano quale parte di memoria è visibile al programma; quando il programma accede alla memoria, se l'accesso avviene in una zona non assegnata dal sistema operativo, viene interrotto e parte una trap (darà un messaggio di errore); nel caso di più programmi in esecuzione, il sistema aggiorna i registri BASE e BOUND a quelli del programma attualmente in esecuzione

TEMPO DI CPU

in un sistema batch:

job1, job2, job3...

se un job si ferma il sistema si blocca

per evitare questo utilizzo un interrupt timer: allo scadere di un timer viene inviato un interrupt (devo caricare il timer con un tempo appropriato altrimenti rischio di interrompere un job che non è in stallo ma semplicemente lungo)

in un sistema time-sharing quando un processo va in loop, il sistema continua ad eseguirlo ma esegue anche gli altri programmi, quindi il sistema rallenta ma non si blocca

continua..