ESERCIZIO Round Robin con priorità CON prerilascio $\Delta t = 2$

Ma esecuzione A=4 e priorità D=3

		B	C			D	E									
CPU	A	B	B	B	B	B	B^X	A	A	D	D	A^X	D^X	C	C^X	E^X
Stato della coda	–	a^3	a^3	a^3	a^3	a^3	a^3	a	d	d	a	a	d	c	e	e
	–		c^2	c^2	c^2	c^2	d^3	d	c	c	c	c	c	e		
	–						c^2	c	e	e	e	e	e			
	–							e								

Scade Δt e ora c'è D che ha la stessa priorità

era arrivato prima A

T. attesa 6,8
T. risposta 4,8
T. turn around 10,2

ESERCIZIO SRTN

	t_0	Es.	Pr.
A	0	3	3
B	1	7	5
C	2	2	2
D	6	3	4
E	7	1	1

		B							D	E						
CPU	A	A	A	A^X	C	C^X	B	D	E^X	D	D^X	B	B	B	B	B	B^X
Coda		b	c	c	b	b		b	d	b	b						
		b	b						b								

si riordina perché Es(C) < Es(B)

Es(D) = 3
Es(B) = 6 (1 già fatta)

⇒ Prerilascia

ad A rimangono 2 unità, a B ne servono 7
NON prerilascia

T. attesa 2
T. risposta 1
T. turn around 5,2

RIASSUNTO dei fondamenti

Gerarchia di memoria:

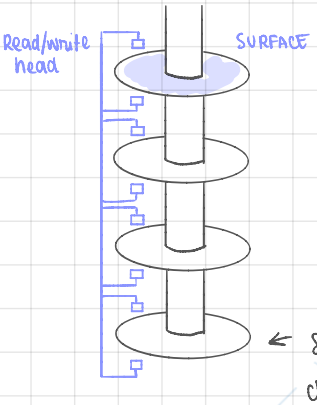
access time		capacity
1nsec	Registers	< 1kB
2nsec	cache	16MB
10nsec	main memory RAM	16GB
10msec	magnetic disk	1TB
100 sec	magnetic tape	1TB

Registers: interni alla CPU
Processori 32 e 64 bit ⇒ dimensione del registro

Cache controllata da un hardware e suddivisa in blocchi (linee)
- L1 (dentro CPU)
- L2 vicina ⇒ come organizzare un eventuale accesso condiviso?
- L3 un po' più lontana

HIT ≠ MISS + write through o write(copy) back

Dischi magnetici capienza 100 volte superiore e costo 100 volte inferiore rispetto alla RAM (main memory) ma MOLTO PIU' LENTA ❌



Hard disk

← Se l'hard-disk cade potrebbe succedere che la testina tocchi la superficie

Solid State Disk (SSD)
Velocizzano e risolvono i problemi del collo di bottiglia dell'hard disk.
✅ Sono più veloci, silenziosi, solidi, non ha parti fisiche in movimento, meno energia (2W/6W), leggeri ↗ aka difficile che si rompa
❌ Costosi, meno capienti, scritture e riscritture usurano le celle

SOLUZIONI IBRIDE
Una parte molto veloce di SSD per il software base e un HDD molto capiente per applicazioni e file
Risultato: macchina molto veloce all'avvio

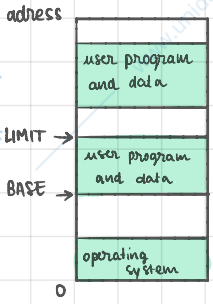
PROGRAMMI & MEMORIA

CMOS = memoria volatile alimentata da una piccola batteria (↑ tampone per evitare perdita)
↳ si perde il contenuto non appena manca corrente

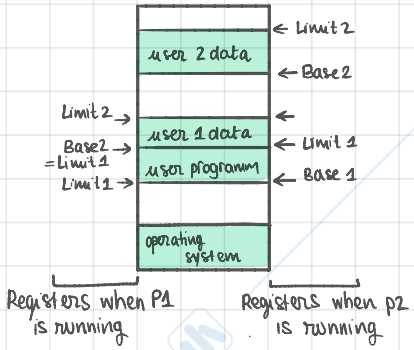
Memorizza l'ora corrente e
le impostazioni BIOS ≠ default

ITERAZIONE MEMORIA - PROGRAMMI

- Come proteggere i programmi tra di loro ed evitare che si sovrascrivano l'uno con l'altro
- come fare interagire il kernel con i programmi
- come gestire la rilocazione
  ↳ infatti i processi usano spazi virtuali che poi devono essere trasformati in qualcosa di fisico = ALLOCAZIONE (RAM o altro)
- Quando si compila un programma non si sa in che area della memoria verrà caricato
  - soluzione hardware, due registri (base & limite) ⇒ so che virtuale va da 0 a k, da k a n è per la fisica
  - verifica e somma base+indirizzo costa qualche ciclo di CPU



Base - Limite
singolo per processo



Base - Limite
distinto per dati e programma

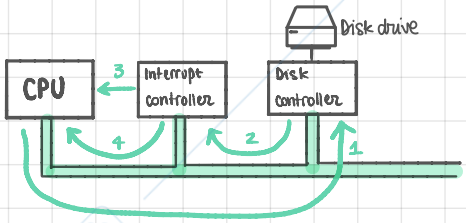
Base - Limite sono una base MOLTO importante perché definiscono come tradurre gli indirizzi
Gestione spazio di memoria virtuale dei processi = MMU che è un dispositivo posto tra CPU e memoria ed è gestito a sua volta dal sistema operativo

❌ Dopo ogni context switch la cache è piena dei dati del processo precedente

GESTIONE DEGLI INTERRUPT

Interrupt segnale inviato per vari motivi tipo periferica che ha finito un compito e avvisa il processore che ha finito.

I. Driver dice a controller cosa fare
II. Segnale finito su certe linee bus
III. Informa il controllore delle interruzioni
IV. Mette nome dispositivo su bus



Quando arriva un interrupt, viene fermato il flusso della CPU ma si decide dopo se effettivamente prenderlo in carico un processo o se si può rimandare

L'uso delle interruzioni evita il ricorso al polling

ARRIVO DI UN' INTERRUZIONE

Leggera valutazione per capire cosa fare (gestirla subito o anche dopo?)

1. I registri PC e PSW sono posti sullo stack del processo corrente ⇒ tolgo un attimo in modo da mandare in esecuzione le istruzioni per fare la verifica sull'interrupt
2. La CPU passa al modo operativo protetto
3. Il parametro principale che denota l'interruzione serve come indice nel vettore delle interruzioni (in modo da capire quale gestore dovrà gestire l'interruzione)
4. La parte immediata del gestore eseguo nel contesto del processo interrotto altrimenti, se è meno urgente viene differita

plug & play

INTEL
MICROSOFT

Compro una periferica, la collego con dei cavi al mio elaboratore e questo immediatamente funziona ed è pronto per l'utilizzo

Prima del PLUG & PLAY bisognava stabilire per ogni periferica il suo interrupt perché ogni periferica ha bisogno di un segnale identificativo
↳ assegnato in modo fisso

Se due periferiche avevano lo stesso interrupt ⇒ una delle due/entrambe non funzionano e quindi bisognerebbe spegnere, cambiare interrupt, riaccendere e andare a tentativi

Con **plug & play** è il S/O ad assegnare ⇒ non c'è rischio che due periferiche abbiano lo stesso interrupt

BIOS

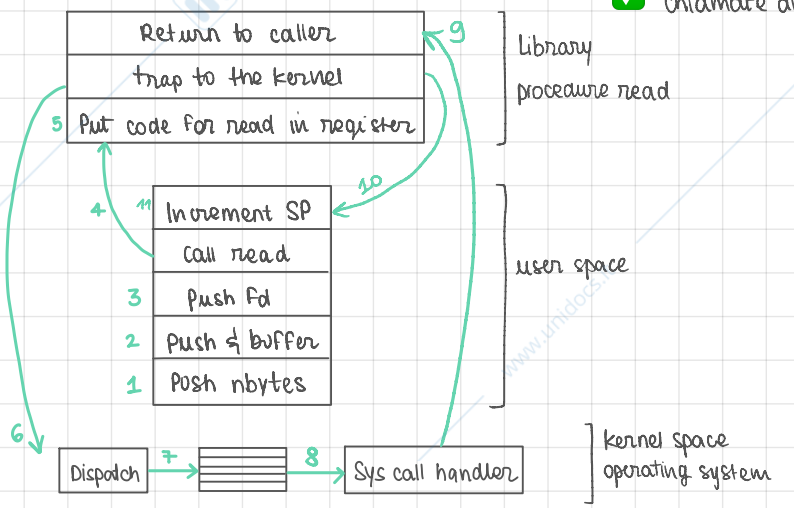
Basic Input Output System

- Nasconde le difficoltà di gestione all'utente
- E' caricato all'avvio del computer
- Alcune verifiche vengono stampate durante l'avvio.
- Fa uno scan dei bus ISA e PCI per rilevare dispositivi connessi
  ↳ se dispositivi vecchi senza plug & play, vengono registrati per primi
  dispositivi plug & play
  dispositivi nuovi dopo ultimo avvio ⇒ configurati
- Determina il dispositivo di boot dalla memoria CMOS
  ↓
  tabella che ci mostra come è partizionata la memoria del calcolatore
  ↳ caricato un secondo boot loader che legge il sistema operativo e lo esegue
  S/O interroga il BIOS per ottenere informazioni sulla configurazione e per ogni dispositivo controlla l'esistenza del driver ⇒ carica nel **kernel**
  → LOGIN dell'utente

Chiamate di sistema = invocazioni esplicite di processi che richiedono l'aiuto del S/O
Per effettuarle, nel codice vanno aggiunte delle librerie in modo da far avvenire la chiamata in modo corretto

Prima istruzione di una chiamata di sistema viene chiamata **trap** e deve attivare la modalità più adatta
- inizia l'esecuzione e viene assegnato un parametro alla chiamata ⇒ SIMILE alla GESTIONE delle INTERRUZIONI
  solo che ❌ interruzioni asincrone
  ✅ chiamate di sistema sincrone

Esempio
count = read (fd, buffer, nbytes)



ALCUNE CHIAMATE di SISTEMA (POSIX) standard

Process management

Call	Description
pid = fork( ) istruzioni sono uguali	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image Rimpiazza parte di codice
exit(status)	Terminate process execution and return status

PID:
0 = processo figlio
un numero > 0 = processo genitore
numero < 0 = errore

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system tipo montare una USB
s = umount(special)	Unmount a file system toglie file system aggiuntivo

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal) parametro -9 = TERMINA	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

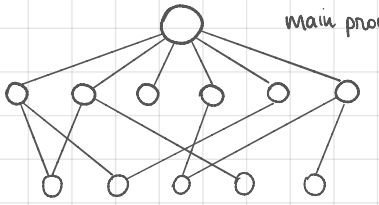
SHELL BASE tramite Fork in modo che non si chiuda la finestra quando finisce il p. figlio

```
while (TRUE) {                                  /* repeat forever
*/                          ↙ segnale dove inserire input
type_prompt( );                                 /* display prompt */
read_command (command, parameters)              /* input from terminal */
                  ↗ Boom creato processo figlio
if (fork() != 0) {          ↙ infatti se non è uguale a 0 vuol dire che sta eseguendo il processo genitore
/* Parent code */                               /* fork off child process */
waitpid( -1, &status, 0);                       /* wait for child to exit */
} else {                    ↙ se è 0 è il figlio in esecuzione
/* Child code */
execve (command, parameters, 0);                /* execute command */
}
```

In questo modo genitore/figlio fanno due cose diverse

Nel momento in cui viene creato un processo figlio, non si sa quale processo andrà in esecuzione per primo anche se di solito un processo appena creato ha un boost di priorità

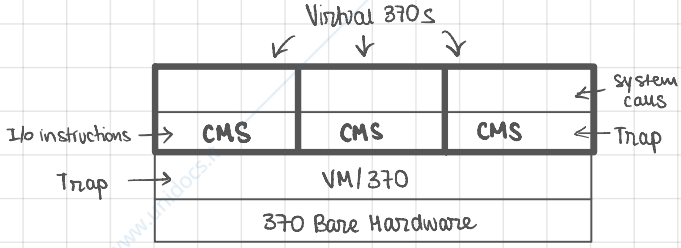
STRUTTURA MONOLITICA



L'architettura monolitica non ha struttura
S/O = collezione piatta di procedura
Unica struttura riconoscibile è la convenzione delle chiamate di sistema

ORGANIZZAZIONE

Programma principale invoca ee procedure di servizio richieste
Insieme di procedure che eseguono le chiamate di sistema
insieme di procedure di utilità che sono di ausilio per le proc. di servizio
GENERALIZZAZIONE ⇒ Struttura a Strati di Dijkstra



Struttura logica sviluppata da IBM con lo scopo di fornire un sistema a time sharing con i sistemi batch dell'azienda

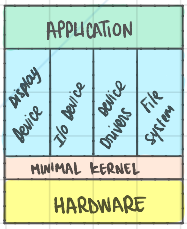
Basato su due fondamentali funzioni: multiprogrammazione e virtualizzazione dell'elaboratore fisico

In questo modo si riesce ad offrire copie identiche di macchine virtuali

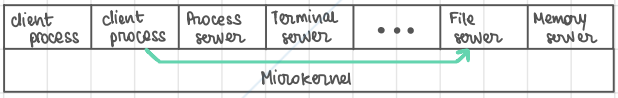
illudere più utenti di avere ognuno la macchina a propria disposizione

semplificare la macchina per l'utilizzo

CMS = Conversational Monitor System
(parte della multiprogrammazione)
Idea partita dal VM/370 ma poi ha avuto un grande seguito

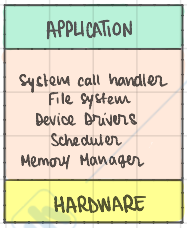


STRUTTURA CLIENTE-SERVENTE



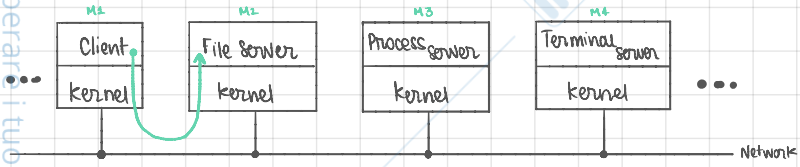
Rallentamento della macchina a causa del continuo passaggio da una modalità all'altra

Semplificazione della macchina (in caso di problemi per esempio, non si blocca tutta la macchina ma solo quel determinato servizio e basta riavviarlo.



Può essere chiamato anche **microkernel** perché infatti è molto ristretto e semplificato
processi di sistema = serventi
processi utente = clienti

Generalizzazione ⇒ **STRUTTURA DISTRIBUITA**



Macchine separate collegate da una rete

MONOLITICO
più semplici da realizzare e da mantenere
LINUX

VS

MICROKERNEL
consentono una gestione più flessibile
MA problemi di sincronizzazione tra le varie componenti ⇒ rallentamento
MINIX

WINDOWS/MAC OS/X
ibridi

GESTIONE della MEMORIA

Memoria: capiente, veloce e permanente (non volatile)
NESSUNA memoria fisica soddisfa tutte e tre, solo l'intera gerarchia di memoria nel suo insieme
È il gestore della memoria che poi decide come soddisfare le esigenze di memoria dei processi

CLASSI di SISTEMI di GESTIONE

- Per processi allocati in modo fisso
- Orientati a processi soggetti a migrazione da memoria principale a disco durante l'esecuzione
  - molto più efficiente e flessibile, dinamicamente la CPU effettua degli spostamenti e posso eseguire programmi molto più ampi
  - ✗ Causa un rallentamento

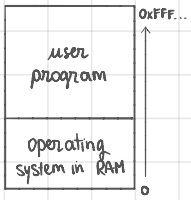
In entrambi i casi il processo avviene in memoria virtuale ma poi deve essere allocato

La memoria disponibile è in generale inferiore a quella necessaria per tutti i processi attivi simultaneamente

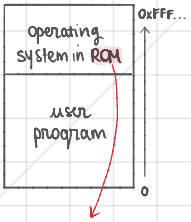
SISTEMI MONOPROGRAMMATI

Eseguono un solo processo alla volta e non si toglie dal processore fino a che non termina
memoria = sistema operativo e quell'unico processo
Parte del S/O ospitata in RAM è solo quella che contiene l'ultimo comando invocato dall'utente
↳ il resto del S/O va allocato da qualche altra parte, unica scelta progettuale

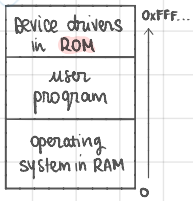
main Frame



embedded



MS-DOS



Perché sono sistemi molto specifici e quindi non serve che il S/O venga aggiornato o modificato

FRAMMENTAZIONE indica la suddivisione dei dati in più parti

interna	VS	esterna
Memoria divisa in blocchi uguali tra loro. Quando un processo viene allocato può essere che un blocco non venga interamente riempito. E i byte liberi NON li posso usare per un altro processo.		Memoria divisa in blocchi di dimensione variabile. Rimangono delle aree sparse e libere non assegnabili ad altri blocchi.

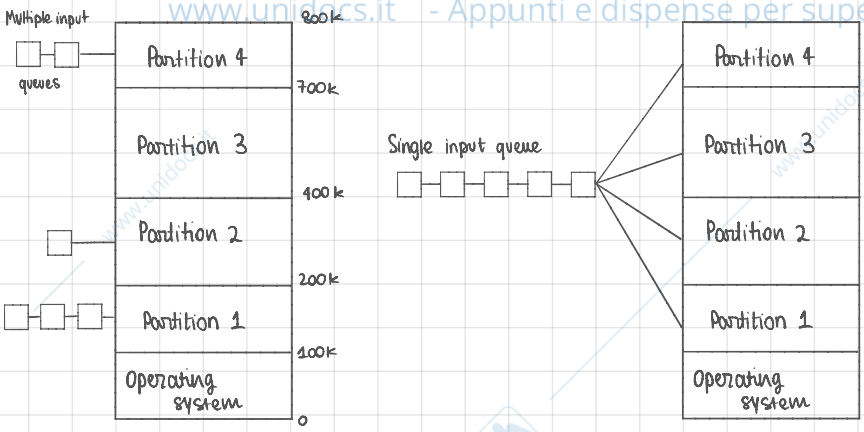
⇒ **SPRECO di MEMORIA**
(o spreco di tempo se voglio provare a ricompattare)

SISTEMI MULTIPROGRAMMATI

Posso avere più processi che vanno in esecuzione in concorrenza sul processore.
La competizione per l'accesso alla memoria è tra i vari processi + sistema operativo
Forma più rudimentale = **PARTIZIONE STATICA** decisa all'avvio del sistema
↓
Spazi di dimensione diverse

PROBLEMA = assegnare dinamicamente gli spazi ai processi mano a mano che occupano il processore cercando di **minimizzare** la frammentazione interna aka spreco di RAM
Ad ogni nuovo processo viene assegnata la partizione di dimensione più appropriata processo 12 ⇒ partizione 13 p.e. MA NON 11
+ **coda di processi per partizione** oppure **selezione opportunistica** avvero scorro la coda e scelgo il processo migliore

Processi piccoli vengono discriminati



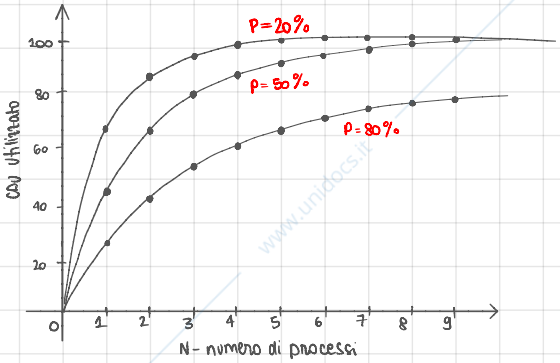
Una coda per ogni partizione

Una sola coda per tutte le partizioni

VALUTAZIONE MULTIPROGRAMMAZIONE

- Valutazione probabilistica di quanti processi debbano eseguire in parallelo per ottimizzare la CPU avendo ipotizzato che:
  - ogni processo impegna P% del tempo in I/O
  - N processi simultaneamente in memoria

tempo stimato = $1 - P^N$



"almeno un processo in I/O" = 1 - "tutti in I/O"

Computer: 32MB e 80% I/O
- 16 MB per sistema operativo
- 4 MB ∀ processo ⇒ Posso avere 4 processi simultaneamente

P = 80% = 0,8 ⇒ $1 - 0{,}8^4 = 60\%$

+16MB altri 4 processi ⇒ $1 - 0{,}8^8 = 83\%$

+16MB ⇒ $1 - 0{,}8^{12} = 93\%$

RILOCAZIONE: interpretazione degli indirizzi emessi in relazione alla sua collocazione in memoria. (↑ per esempio indirizzo 10 di A ≠ indirizzo 10 di B perché fanno riferimento alla loro memoria VIRTUALE)

riferimenti assoluti vs relativi da rilocare

PROTEZIONE: assicurazione che ogni processo operi soltanto nello spazio di memoria ad esso permissibile

soluzione adottata da IBM: Memoria divisa in blocchi (2kB) con un codice di protezione (4 bit)

PSW = codice di protezione univoco

S/O blocca ogni tentativo di accedere a blocchi con un codice di protezione != PSW corrente

SOLUZIONE COMBINATA

Do la possibilità ad un processo di accedere solo ad una determinata parte di memoria compresa tra una base e un limite. Basta prendere la base, sommarci la quantità di memoria e verificare che NON superi il limite.