

# Caso di studio: il kernel di Unix

# Sommario

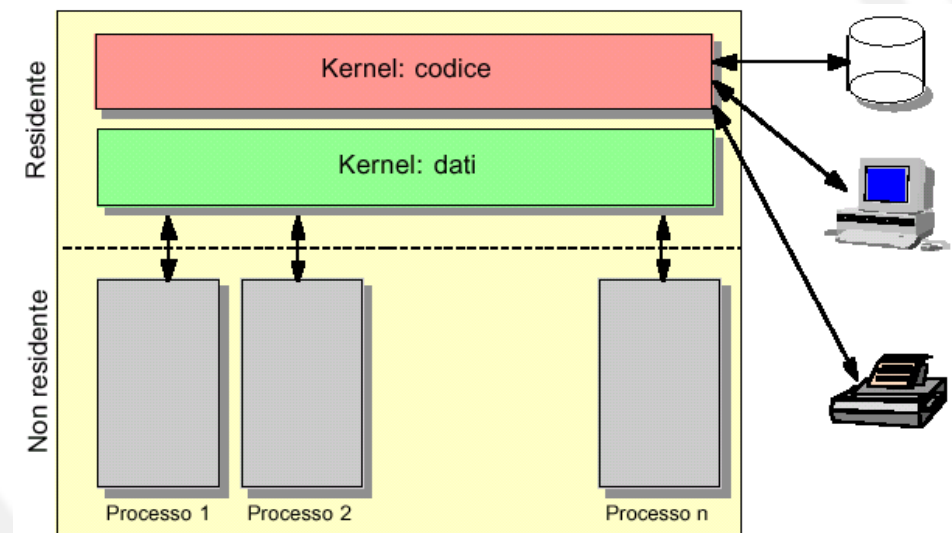
- Storia
- Architettura del kernel
- Strutture dati del kernel
  - Gestione dei processi
  - Gestione del file system
  - Gestione della memoria
- Per approfondimenti:
  - M.J. Bach, “The Design of the Unix Operating Systems”, Prentice Hall, 1986
  - Ed. italiana: “Unix: Architettura di sistema” Gruppo Editoriale Jackson, 1988

# Storia

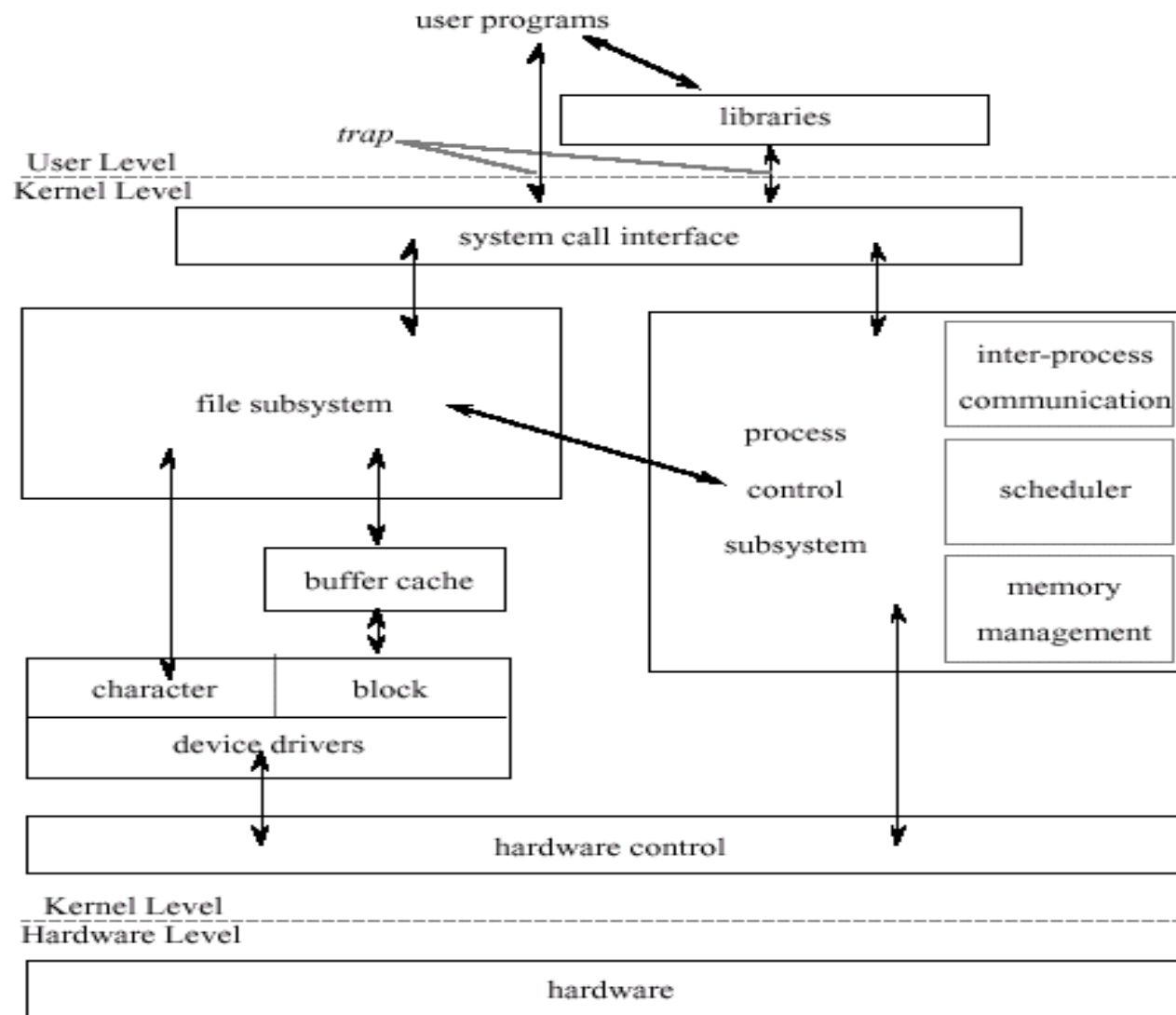
- Ver. 1: Ken Thompson (Bell Labs, 1969)
- Ver. 3: riscritto in C (1973)
  - Aggiunta la multiprogrammazione
- Ver. 6: rilasciata fuori dai Bell Labs (1976)
- Ver. 7: antenato del moderno Unix (1978)
- System V: memoria virtuale paginata (1983)
- 3BSD: univ. of Berkeley (1978)
- 4BSD: supportato dalla DARPA
  - Protocollo TCP/IP per comunicazione di rete
- ...
- 1a versione Linux: Linus Torvalds (1991)

# Architettura del kernel

- Kernel consiste di
  - Una parte residente
    - Kernel vero e proprio
    - Codice esegue in modalità kernel
    - Opera su strutture dati private
  - Una parte non residente
    - Esegue come processo utente in modalità user



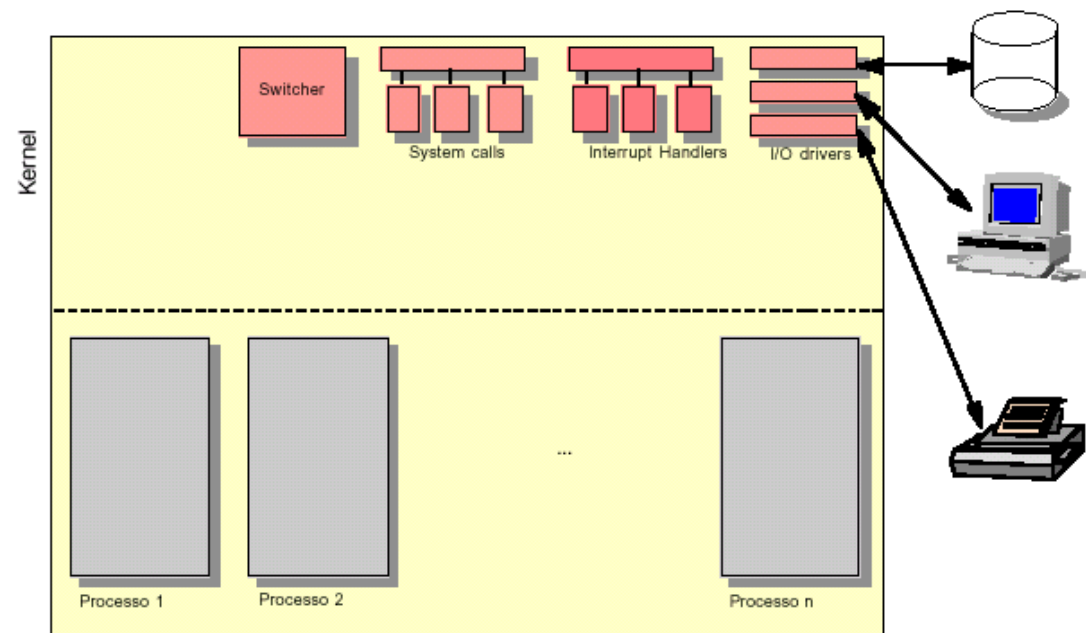
# Architettura del kernel



# **MODULI E STRUTTURE DATI DEL KERNEL**

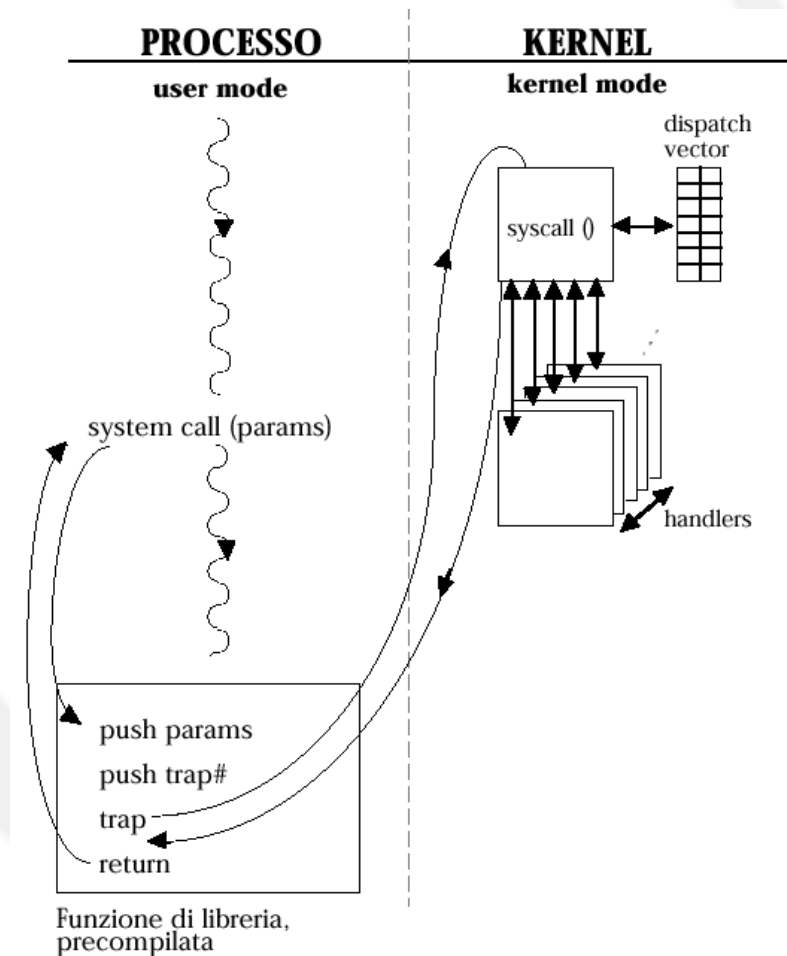
# Kernel: moduli principali

- Gestore delle system call
- Gestori di interrupt
- Driver
- Switcher (scheduler + dispatcher)



# Come funzionano le system call

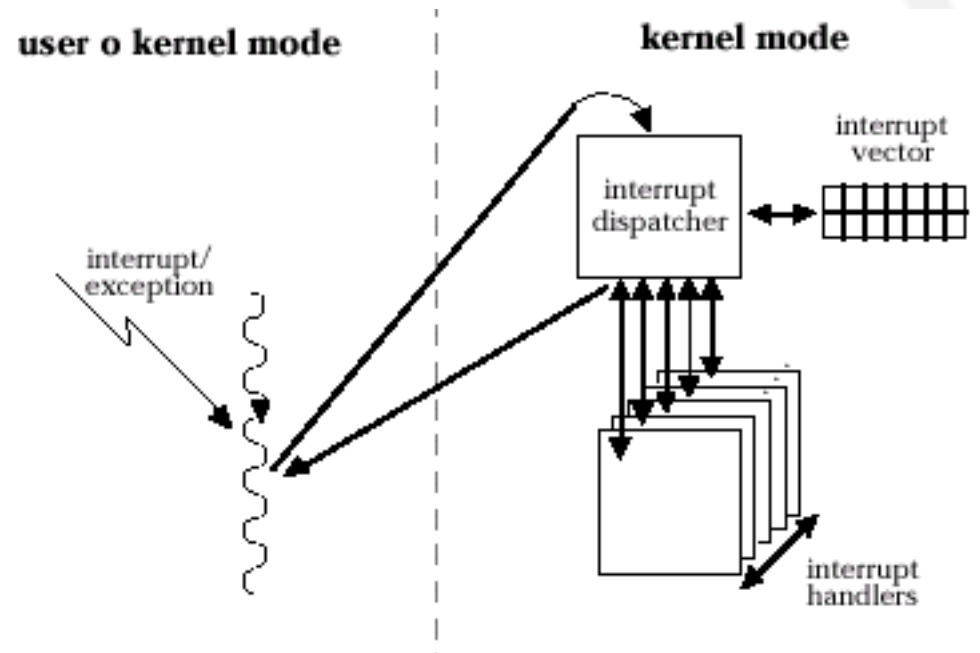
- System call genera una trap per attivare in modo rapido il codice della chiamata
- La trap
  - mette nello stack i parametri e il codice identificativo della trap
  - fa entrare in modalità kernel
- Il gestore delle trap seleziona la routine corrispondente a trap#





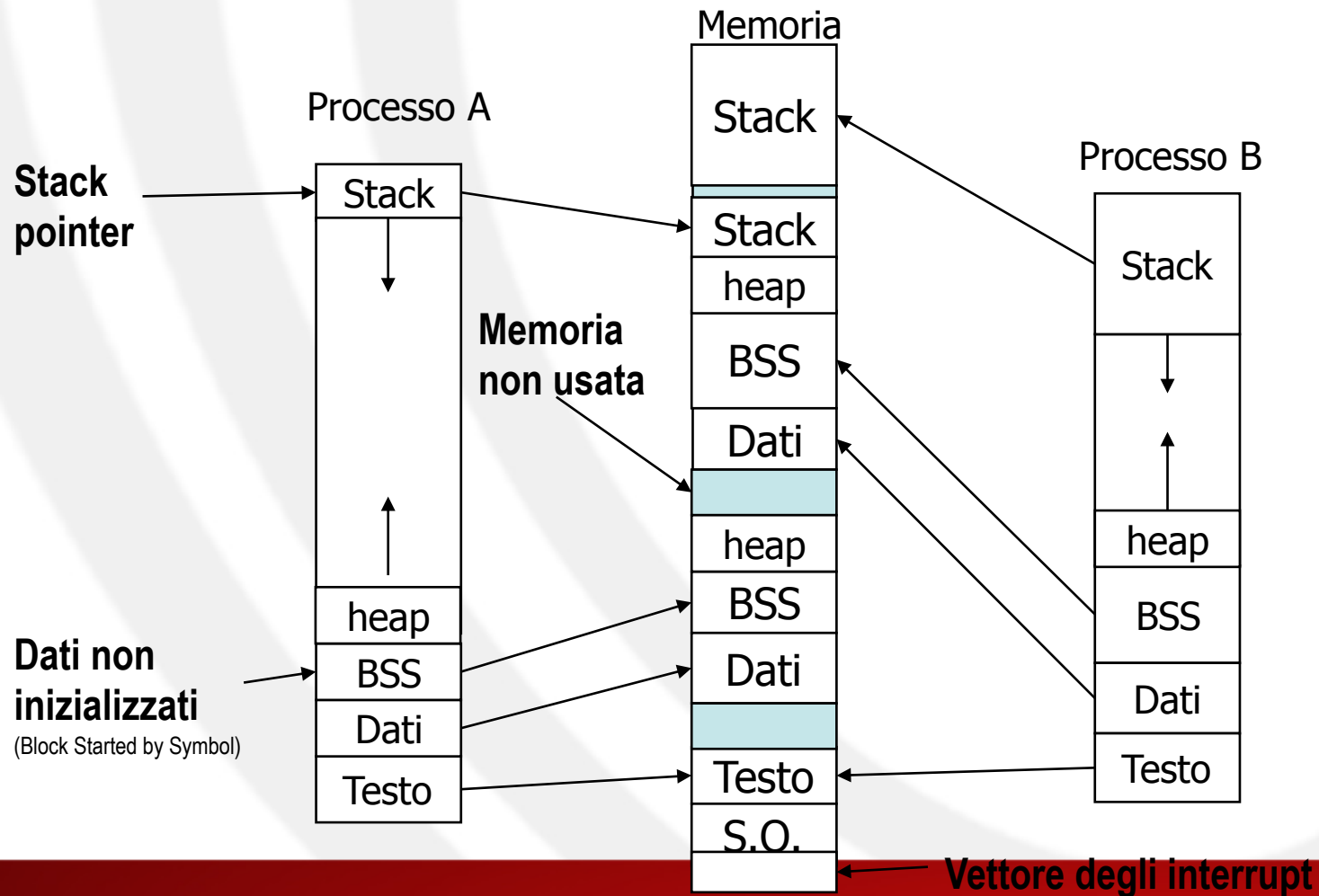
# Gestori di interrupt

- Componente essenziale
  - **Trattamento di**
    - interruzioni generate da device periferici
    - eccezioni generate dall'hardware (es. stack overflow, div. per 0 ...)

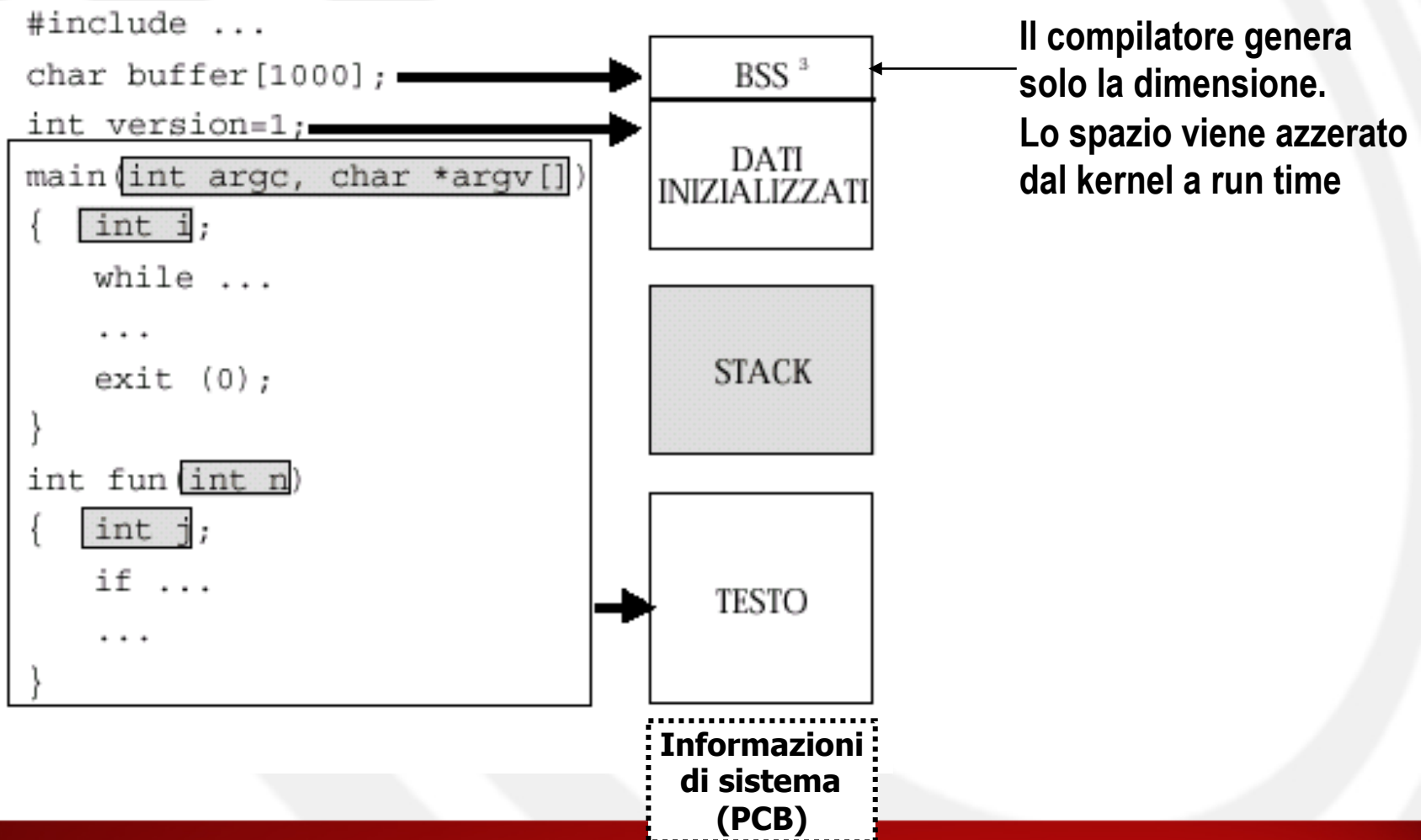


# GESTIONE DEI PROCESSI

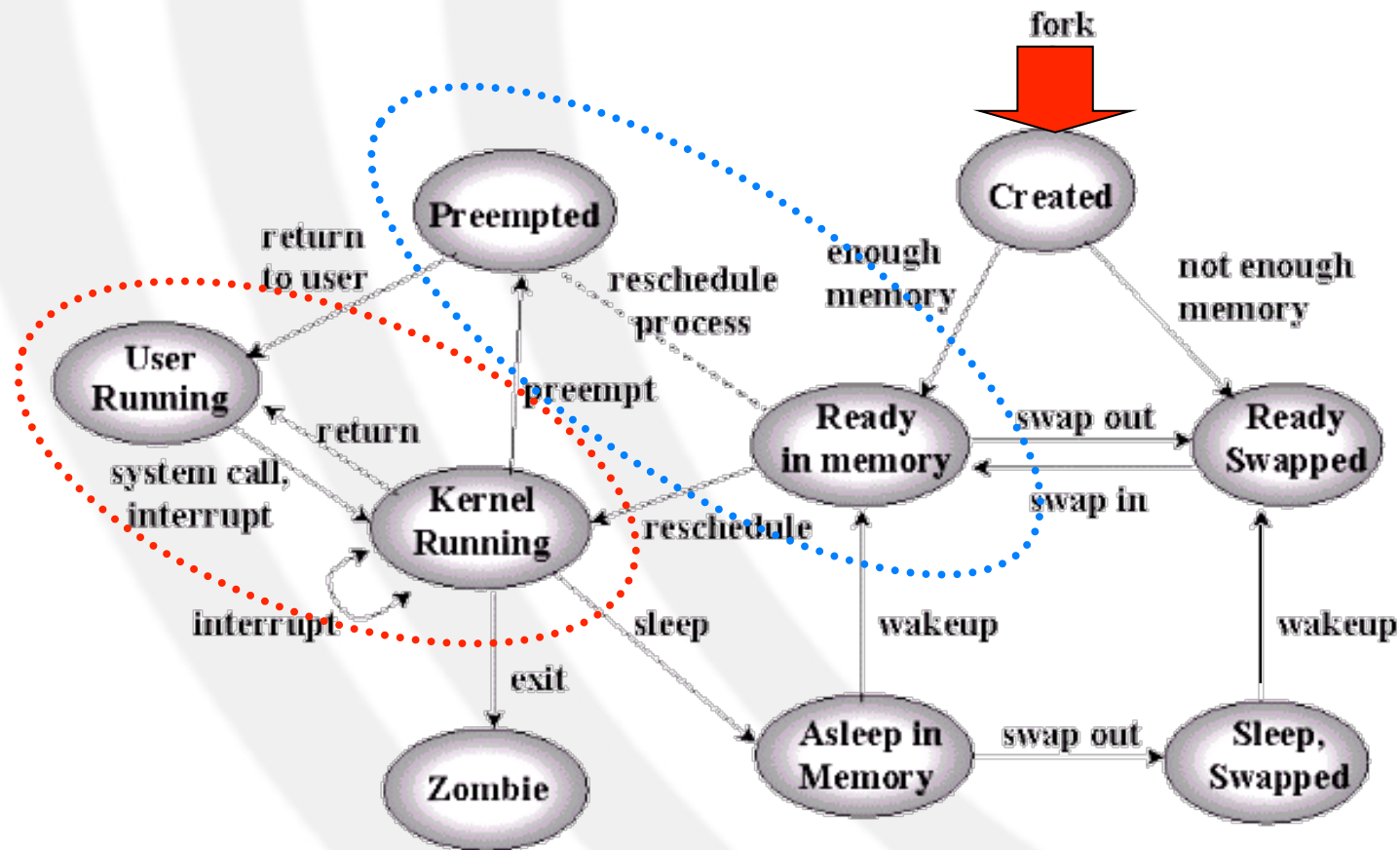
# Immagine in memoria



# Programmi e processi



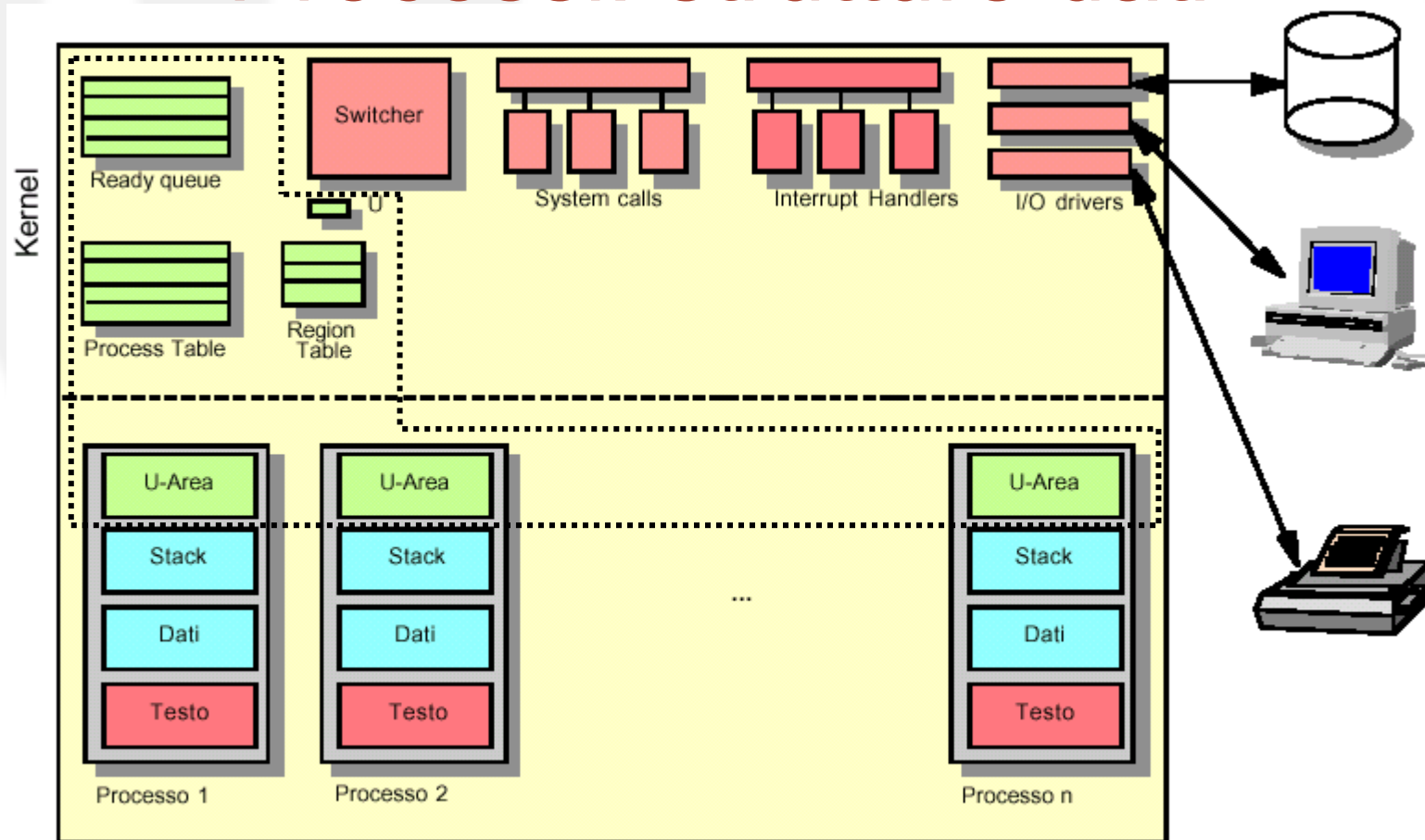
# Unix: stati di un processo



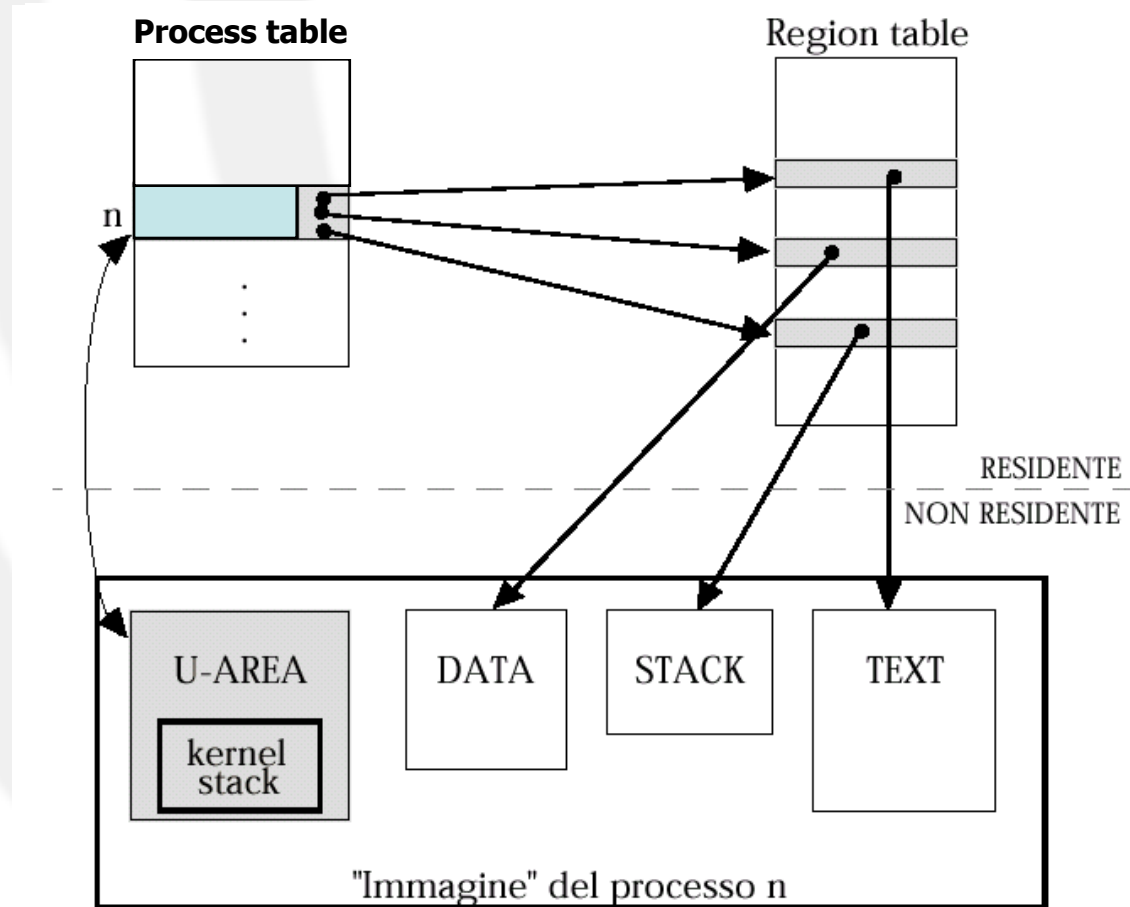
# Processi: strutture dati

- Process Table
  - informazioni (residenti) sui processi
- U-area
  - informazioni (non residenti) sul processo
- Region Table
  - indirizzi delle regioni
- Ready Queue
  - code dei processi pronti
- U
  - puntatore al processo running

# Processi: strutture dati



# Processi: relazione tra strutture dati





# Process table

- Una entry per ogni processo, allocata alla creazione del processo, e deallocata alla sua terminazione
- Contiene informazioni utili anche quando il processo non è attivo
  - PID (process ID), PPID (parent PID), UID (user ID)
  - stato del processo
  - dimensioni del processo
  - parametri di schedulazione
    - priorità del processo, tempo di CPU usato, tempo di attesa, ...
  - puntatore al prossimo processo nella coda di scheduling
  - maschere per i segnali (che specificano come trattarli)
  - evento/i su cui il processo è in attesa
  - ...

# User area (U-area)

- Estensione della entry della process table
- Contiene informazioni che servono solo quando il processo è in esecuzione
- Utilizzata solo dal kernel
- Campi principali
  - Puntatore alla process table entry
  - Area salvataggio registri e program counter
  - Informazioni relative alla system call corrente
  - UID reale ed effettivo (per determinare i privilegi)
  - File descriptors dei file aperti dal processo
  - Parametri per operazioni di I/O
  - Directory corrente e root directory
  - Informazioni per l'accounting
  - Stack del kernel (kernel stack)

# Regioni e Region table

- Regione (o segmento)
  - area contigua dello spazio di indirizzamento di un processo, che viene trattata dal S.O. come oggetto atomico
- Ogni processo ha 3 regioni
  - text
  - data
  - stack
- Porzioni di memoria condivisa (shared memory) possono essere considerati come parti della regione data

## Region table

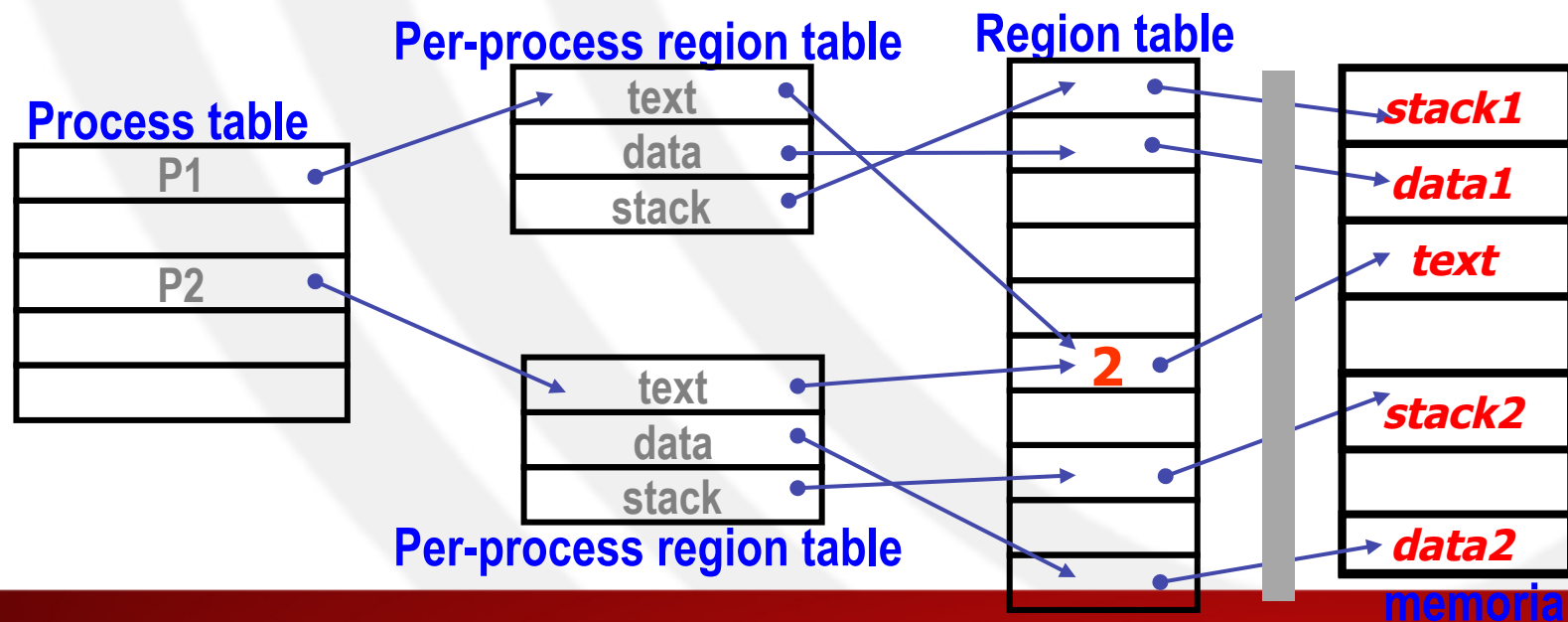
- Una entry per ogni regione, allocata alla creazione della regione e deallocata alla terminazione dell'ultimo processo che la usa
- Campi principali
  - indirizzo di memoria
  - indirizzo su disco (per regioni swappate)
  - dimensioni della regione
  - numero di riferimenti da processi
  - puntatore alla page table

## Region table

- In realtà l'associazione tra processi e spazio di memoria avviene in modo indiretto tramite tabella "intermedia" associata ai singoli processi detta per-process region table (p-region table)
- Questa indirectione permette la condivisione delle regioni da parte di processi diversi

# P-region table

- Allocata nella process table, nella u-area, o separatamente (dipende dall'implementazione)
- Contiene anche info relative al modo di accesso (r,w,x)
- Schema completo



# Processi

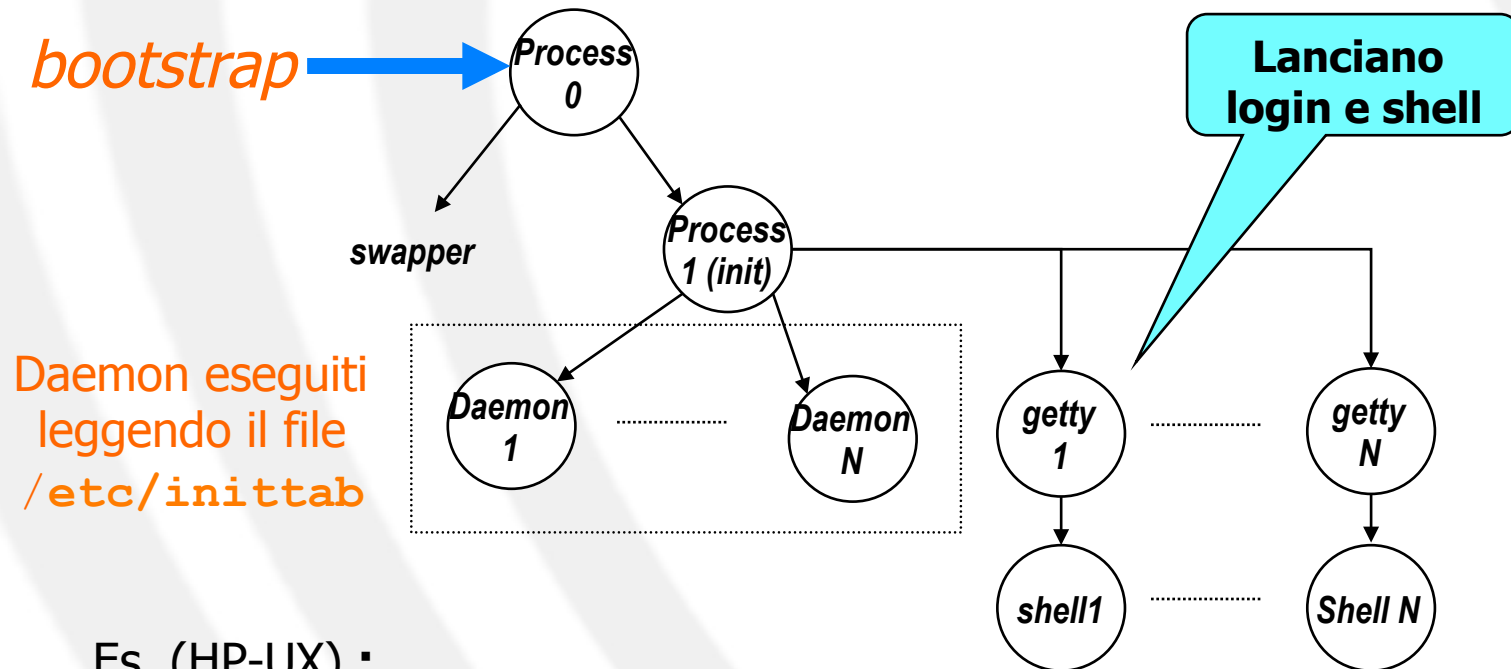
- Creazione
  - fork()
  - exec()
- Terminazione
  - exit()
  - wait()

# Processi

- Il processo 1 è l'init
  - Viene creato dal kernel dopo il boot
- Tutti i processi utente sono discendenti di init
- Oltre a init esistono pochi altri processi creati al boot, tra cui:
  - Swapper (sched, PID = 0)
    - Gestisce lo swap-in e swap-out dei processi nella/dalla memoria
    - In Linux si chiama kswapd
  - Pagedaemon (pageout, PID = 2)
    - Gestisce i page-fault
- Nuovi processi vengono creati mediante fork/exec



# Albero dei processi UNIX



Es. (HP-UX) :

UID	PID	PPID	C	STIME	TTY	TIME	COMMAND
root	0	0	0	Jun 9	?	0:21	swapper
root	1	0	0	Jun 9	?	0:00	init
root	2	0	0	Jun 9	?	0:00	vhand
root	3	0	0	Jun 9	?	0:49	statdaemon
...							
root	1553	1	0	Jun 9	console	0:00	/usr/sbin/getty

# fork()

- Esecuzione di una fork()
  - Alloca una entry nella process table per il nuovo processo
  - Assegna un PID unico al nuovo processo e inizializza i campi della entry della process table
  - Crea una copia dell'immagine del processo padre
    - il text non viene duplicato, ma si incrementa un reference count
  - Incrementa opportuni contatori per i file aperti
  - Copia la u-area dal padre e inizializza i contatori di accounting
  - ...
  - Pone il nuovo processo nella ready-queue
  - Restituisce il PID del figlio al padre, e 0 al figlio

# exec()

- Esecuzione di una exec()
  - Recupera il file eseguibile
  - Verifica i permessi di esecuzione (e il fatto che sia un eseguibile)
  - Se il file ha SUID (o SGID) a 1, cambia l'effective UID (o GID) con quello del proprietario del file
  - Copia argomenti e ambiente nello spazio del kernel perché lo spazio utente sta per essere distrutto
  - Libera il vecchio spazio di indirizzamento e ne crea uno nuovo
  - Ripristina argomenti e ambiente nel nuovo user stack
  - Imposta i gestori dei segnali alle azioni di default
  - Inizializza il contesto HW
    - Registri a 0
    - Program counter inizializzato con l'indirizzo della prima istruzione del nuovo programma

# Perché fork ed exec sono separate?

- In applicazioni client-server il server può creare (fork) numerosi processi che eseguono il suo stesso codice
- A volte un processo potrebbe voler eseguire un nuovo programma senza creare un nuovo processo
- Tra una fork e una exec possono essere eseguite operazioni di inizializzazione per il nuovo processo
  - Ridirezione di stdin, stdout, stderr
  - Chiusura di file ereditati dal padre e non necessari
  - ...

# exit()

- Esecuzione di una exit()
  - Invocata tramite system call oppure su ricezione di un segnale di kill
  - Chiude i file aperti
  - Rilascia tutte le risorse del processo
  - Mette il processo nello stato di zombie
    - La process table non è rimossa in attesa che il padre recuperi exit status del figlio morto e informazioni sull'uso delle risorse
  - Rilascia lo spazio di indirizzamento
  - Notifica al padre tramite SIGCHLD
    - Se il padre è interessato alla morte del figlio si mette in ascolto per ricevere SIGCHLD, altrimenti il segnale è ignorato
  - Chiama swtch() per schedulare un nuovo processo

# wait()

- Esecuzione di una wait()
  - Se il processo chiamante non ha figli, restituisce un codice di errore
  - Se il processo chiamante ha figli nello stato zombie
    - Ne seleziona uno a caso
    - Ne elimina il descrittore dalla Process Table
    - Ne restituisce al chiamante il PID e l'exit code
  - Se il processo chiamante ha figli in uno stato non-zombie sospende il chiamante
- Il padre dovrebbe attendere la terminazione di tutti i suoi figli prima di terminare
  - Altrimenti i figli rimangono orfani e vengono “adottati” da init che libera la loro process table quando terminano
- Problema: se un figlio muore prima del padre e il padre non esegue la wait il figlio rimane zombie fino al reboot → spreco di risorse

# SCHEDULING

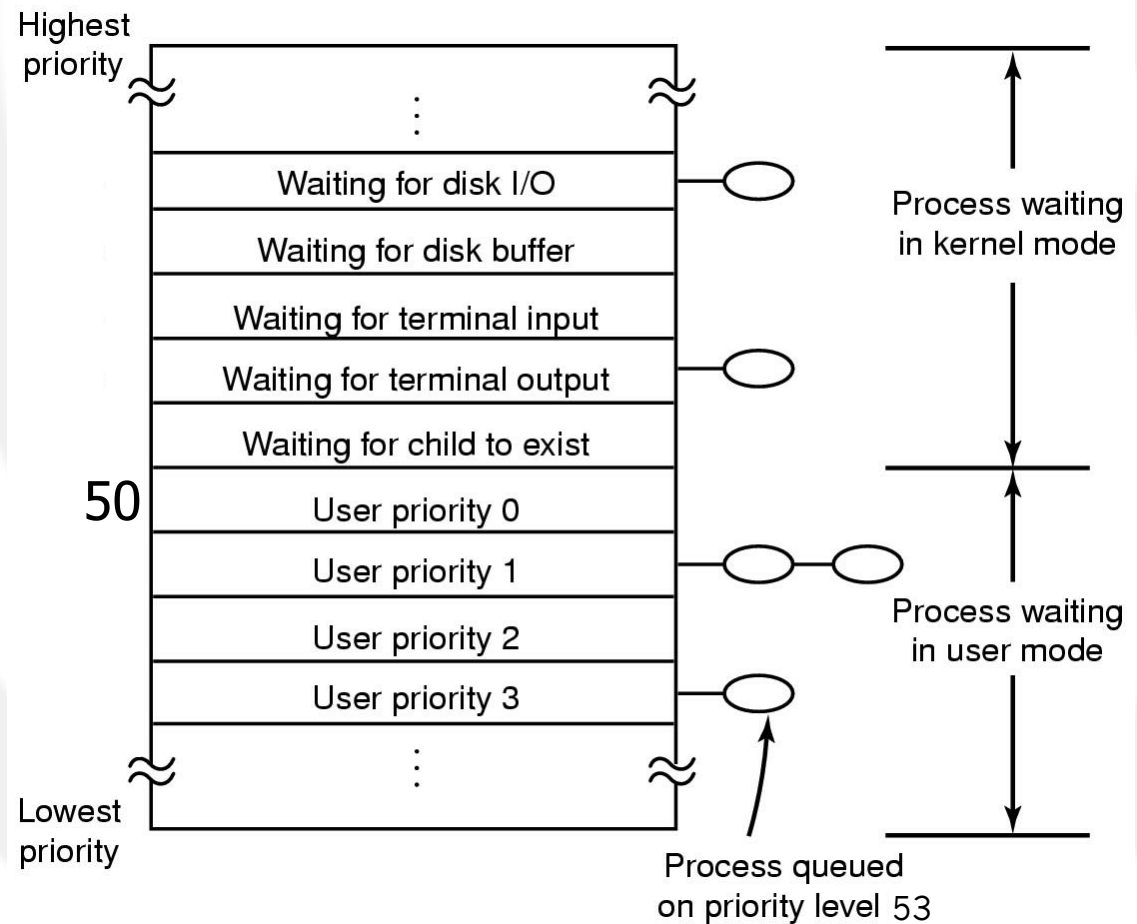
# Scheduling

- Unix è un sistema time-sharing
- Ad ogni processo è associata una priorità dinamica
  - Priorità ricalcolata a intervalli regolari
    - Es.: 4.3BSD 1 volta ogni secondo
- Una coda di scheduling per ogni livello di priorità
- Algoritmo di scheduling
  - Priorità con multilevel feedback
  - Processi a priorità + elevata prelaiono quelli a priorità + bassa
  - Round-Robin all'interno della stessa coda
    - Es.: Time slice 100 ms per 4.3BSD



# Code di scheduling

- Processi che eseguono in modo kernel hanno priorità maggiore di quelli che eseguono in modo user
- Richieste di I/O hanno priorità elevata per garantire elevato time-sharing



## Clock HW

- Clock HW interrompe (interrupt) il sistema a intervalli regolari (tick)
  - **Esempio**
    - In Linux un tick ogni 10 ms
      - La costante HZ definita in `/usr/include/asm/param.h` è impostata a 100 (100 interrupt al secondo)
- Priorità del clock interrupt è seconda solo alla priorità dell'interrupt che segnala power failure

# Clock HW

- Il gestore del clock interrupt:
  - Carica il clock HW (se necessario)
  - Aggiorna le statistiche sull'uso della CPU per il processo corrente
  - Ricalcola le priorità e gestisce time-slice expiration
  - Uccide (segnale SIGXCPU) il processo corrente se ha superato il limite massimo per l'uso della CPU
  - Aggiorna la data e l'ora
  - Gestisce le callout
    - Funzioni che il kernel deve eseguire ad un certo istante futuro
  - Sveglia swapper e pagedemon
  - Gestisce gli alarms
- Non tutte queste operazioni vengono svolte ad ogni tick
  - Es. 4.3BSD ricalcola le priorità ogni secondo

# Priorità

- Numero intero da 0 a 127 (0 = max)
  - Processi kernel = 0..49
  - Processi user = 50..127
  - 32 ready-queue, una ogni 4 valori di priorità
- Calcolato in funzione di
  - Quantità di tempo di CPU usato
  - Valore di nice

# Priorità

- Utilizzo della CPU (C )
  - Ad ogni tick viene incrementato (fino ad un valore max di 127) se il processo è in esecuzione
    - C cresce → processi che hanno usato molto la CPU perdono priorità
  - Ricalcolata all'inizio di ogni time slice ( $C = C * \text{decay}$ ,  $\text{decay} < 1$ )
    - C cala → processi che attendono da molto tempo acquisiscono priorità
      - SVR3:  $\text{decay} = \frac{1}{2}$
      - 4.3BSD:  $\text{decay} = (2 * \text{load\_average}) / (2 * \text{load\_average} + 1)$ 
        - »  $\text{Load\_average} = \#$  processi "runnable" nell'ultimo secondo
- Valore di nice (N )
  - Aumentabile dall'utente (system call nice())
    - Solo superuser può decrementare nice
  - Compreso tra 0 e 39 (default 20)

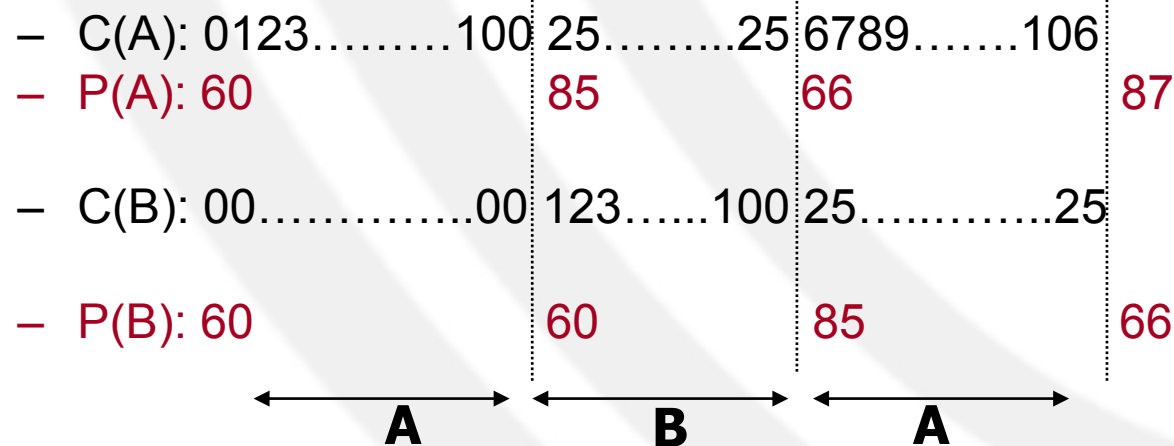
# Priorità

- Calcolo della priorità (4.3BSD)
  - $\text{Priority} = \text{BASE} + N/2 + C \cdot \text{decay}$
  - $\text{BASE} = 50$
- Obiettivo
  - Job interattivi e I/O bound avvantaggiati
  - Job CPU bound e job lanciati in background non vanno in starvation
- Recenti versioni di Unix hanno ulteriormente migliorato l'algoritmo di scheduling per gestire real-time

# Priorità

- Esempio:
  - 2 processi: A,B (A in esecuzione inizialmente)
  - Valore iniziale  $P = 50$
  - decay =  $1/4$
  - Ricalcolo ogni 100ms
  - Clock interrupt = 1 ogni 1ms
  - $N = 20$  fisso (BASE +  $N/2 = 60$ )

$$P = 60 + C/4$$



# GESTIONE DEL FILE SYSTEM



# File system

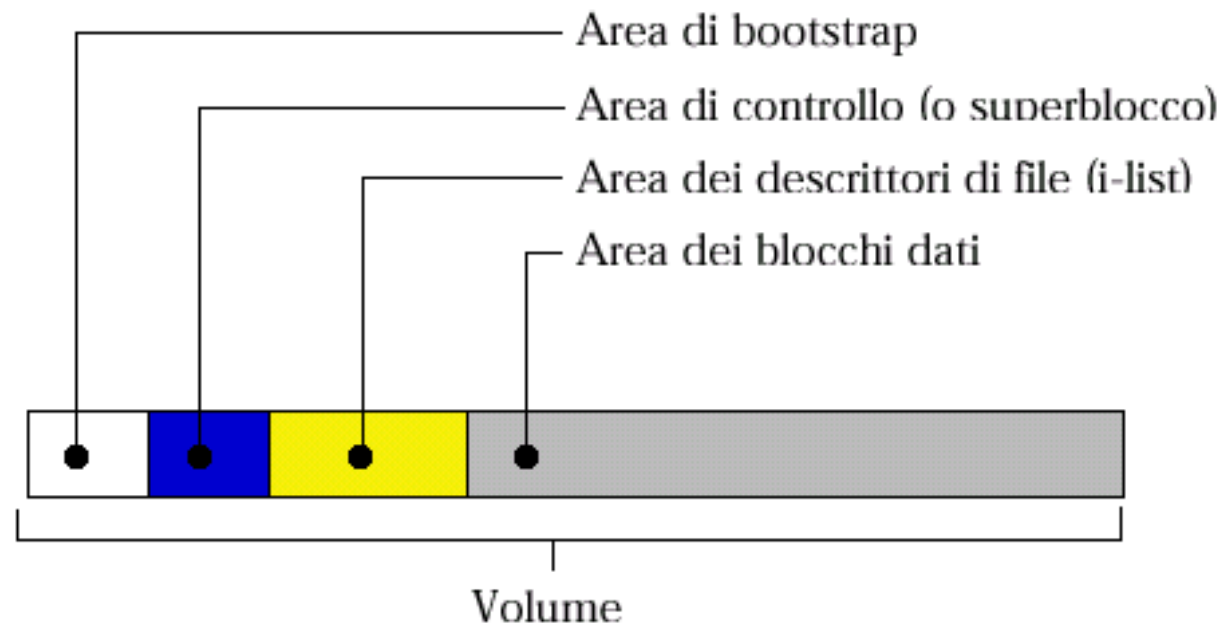
- s5fs (System V file system)
    - Preso come riferimento per le prossime slide
  - ffs (Fast file system, 4.2BSD)
  - ufs (Unix file system, estensione di ffs)
  - afs (Andrew file system)
  - nfs (Network file system)
  - ext2 (The 2nd extended file system)
  - ext3 (The 3rd extended file system)
  - Ext4 (the 4th extended file system)
  - vnode/vfs interface per gestire file system diversi sulla stessa macchina (Sun)
- Per trasferimento  
file tra host remoti
- Linux
- Implementa journaling
  - Migliorata l'efficienza di ext3 e supporto per volumi più grandi

# Struttura del file system

- Ogni disco è suddiviso in zone contigue dette volumi o partizioni
- Ogni file system è interamente contenuto in una partizione
- Ogni partizione può contenere un solo file system
- Per creare un file system si usa il comando mkfs
  - La dimensione delle partizioni è fissa e specificata al momento della creazione del file system
  - Ogni partizione è suddivisa in blocchi di dimensione fissa (512-4K a seconda della versione di sistema)

# Struttura del file system

- Organizzazione fisica: 4 aree distinte
  - Dimensioni definite all'atto della creazione



## Area di bootstrap

- Contiene il programma di bootstrap per l'inizializzazione del sistema
- Si usa tipicamente l'area di bootstrap del volume che contiene il root file system ...
- ... ma per uniformità ogni volume contiene l'area di bootstrap

# Superblocco

- Singolo blocco che contiene informazioni globali sul volume
  - Dimensione del volume (n° di blocchi)
  - Dimensione della lista degli i-node (i-list)
  - Nome del file system
  - Identificazione del dispositivo
  - Data dell'ultima modifica al superblocco
  - Informazioni per gestione blocchi liberi
    - Numero dei blocchi liberi nel volume
    - Inizio della lista dei blocchi liberi
  - Informazioni per gestione i-node liberi
    - Numero degli i-node liberi
    - Cache degli i-node liberi

Modificati  
spesso

# Superblocco

- Superblocco modificato frequentemente
  - Per motivi di efficienza, una copia del superblocco si trova in memoria
  - Quando viene modificato, il S.O. registra le modifiche
- Il superblocco è un'area critica, senza la quale l'intero file system non è più accessibile
- In versioni recenti (es. FFS e successivi), esistono alcune copie di backup del superblocco

## Area dei descrittori di dati (i-node list)

- Tabella di descrittori di file detti i-node (index-node)
- Ogni i-node individua un unico file
  - i-node accessibile tramite il suo indice nella tabella (i-node number)
  - Contiene gli attributi di un file
- Dimensione della tabella fissata al momento della creazione del file system

# Contenuto di un i-node

- Tipo di file
  - 0 = i-node libero
- Numero di hard link
- UID del proprietario
- GID del proprietario
- Permessi di accesso
- Dimensione del file (byte)
- Tabella degli indirizzi dei blocchi dati
- Ora/data ultimo accesso
- Ora/data ultima modifica
- Ora/data ultima modifica dell'inode





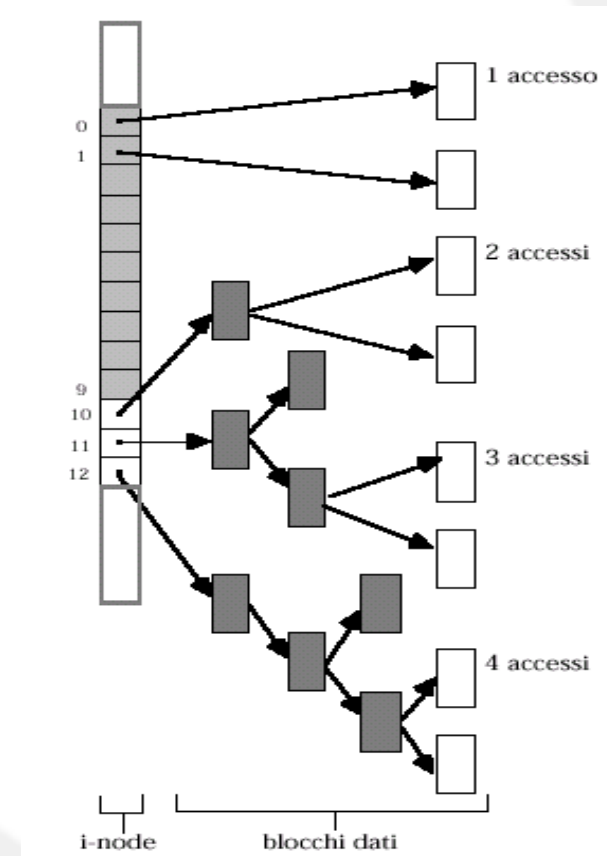
# UNIX i-node

Field	Bytes	Description
Mode	2	File type, protection bits, setuid, setgid bits
Nlinks	2	Number of directory entries pointing to this i-node
Uid	2	UID of the file owner
Gid	2	GID of the file owner
Size	4	File size in bytes
Addr	39	Address of first 10 disk blocks, then 3 indirect blocks
Gen	1	Generation number (incremented every time i-node is reused)
Atime	4	Time the file was last accessed
Mtime	4	Time the file was last modified
Ctime	4	Time the i-node was last changed (except the other times)

**Totale = 64**

# i-node

- Indirizzi ai blocchi dati:
  - Indirizzamento indicizzato con vari livelli di indirezione
  - 13 indirizzi per i-node
    - 10 diretti
    - 1 indiretto
    - 1 indiretto doppio
    - 1 indiretto triplo
  - Privilegiati i file piccoli
    - Richiedono un unico accesso
  - 48% file < 1KB
  - 85% file < 8KB

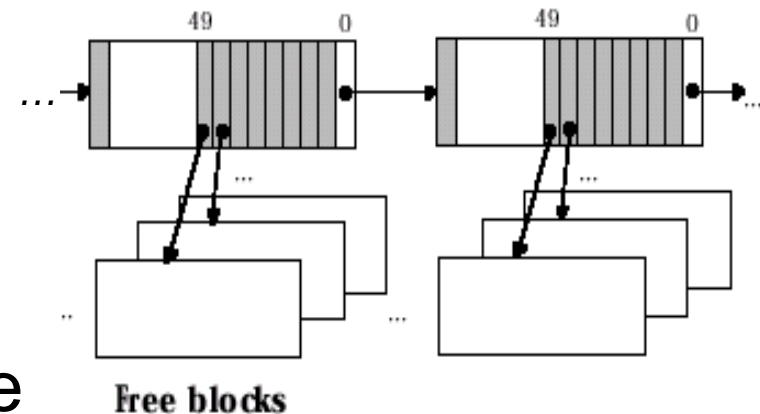


## i-node – esempio

- Dimensione massima indirizzabile con blocchi da 1KB e indirizzi da 4 byte:
  - $10 * 1 \text{ Kb} = 10 \text{ Kb} +$   
 $256 * 1 \text{ Kb} = 256 \text{ Kb} +$   
 $256 * 256 * 1 \text{ Kb} = 64 \text{ Mb} +$   
 $256 * 256 * 256 * 1 \text{ Kb} = 16 \text{ Gb}$   
 $> 16 \text{ GB per file}$
  - N.B. con 32 bit indirizzo fino a 4GB → file grande al max 4GB
    - I moderni file system (etx2, ext3) supportano file più grandi ( $2^{64}$ )
- Come accedere al byte 12500 di un file?
  - $12500/1024 = 12 \rightarrow$  blocco 12
  - Resto = 212 → offset (in byte) all'interno del blocco 12

## Lista dei blocchi liberi

- Organizzata come una lista di elementi ciascuno contenente una serie di puntatori a blocchi liberi
  - Inizio della lista contenuto nel superblocco
- Struttura di un elemento della lista:
  - Contatore (1...50)
  - 50 puntatori a blocchi
    - Il primo ([0]) punta al resto della lista
- Possibilità di allocare/liberare più blocchi liberi alla volta con un solo accesso a disco

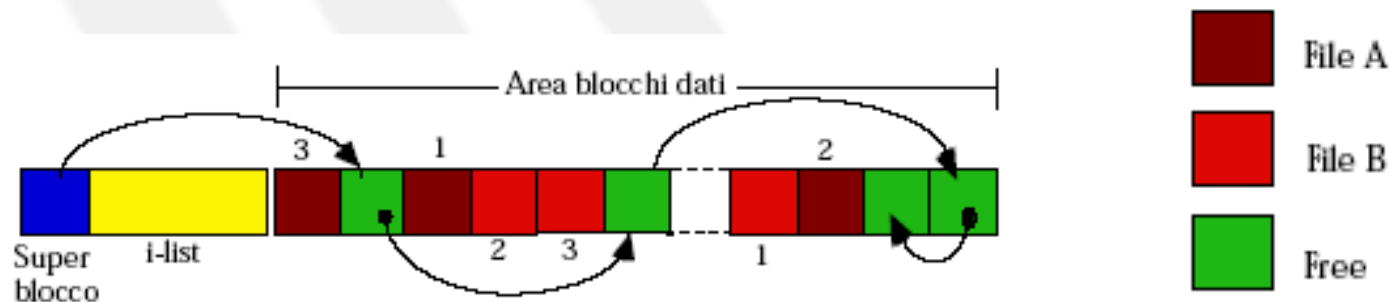


## Caching degli i-node liberi

- Nel superblocco viene anche mantenuto un elenco parziale di i-number di i-node liberi (“caching”)
  - Sottoinsieme di tutti gli i-node liberi
- Gli i-node liberati vengono riutilizzati
  - Quando serve un i-node libero, dalla cache viene prelevato un i-number (se c'è)
  - Quando la cache è vuota, la si rialimenta scandendo la i-list per cercare altri (uno o più) i-node liberi

## Area dei blocchi dati

- Contiene i blocchi dati
  - Indirizzi dei blocchi sono nell'i-node
- Contiene i blocchi liberi
  - Collegati in una free-block list la cui testa si trova nel superblocco
- Blocchi di un file non sono necessariamente contigui



# S5fs: Vantaggi & Svantaggi

- Vantaggi
  - Semplicità
- Svantaggi
  - Se si rovina superblocco l'intero file system va perso
  - Allocazione fisica non ottimizzata
    - i-node tutti a inizio volume
    - Blocchi dati tutti in fondo
  - Dimensione del blocco fissa
    - Grande → migliori prestazioni, ma aumenta frammentazione
    - Piccolo → frammentazione ridotta, ma prestazioni ridotte
  - Nome file da max 14 caratteri
  - Al max 65535 file

## Miglioramenti di ffs

- Nomi file da 255 caratteri
- Partizioni suddivise in gruppi di cilindri contigui ciascuno con informazioni che permettono di tenere vicini dati correlati
- Superblocco replicato
- File system diversi possono avere blocchi di dimensione differente
- Un blocco è suddiviso in frammenti di max 512 byte
- L'ultimo blocco di un file può essere parziale (1 o + frammenti) → frammentazione ridotta

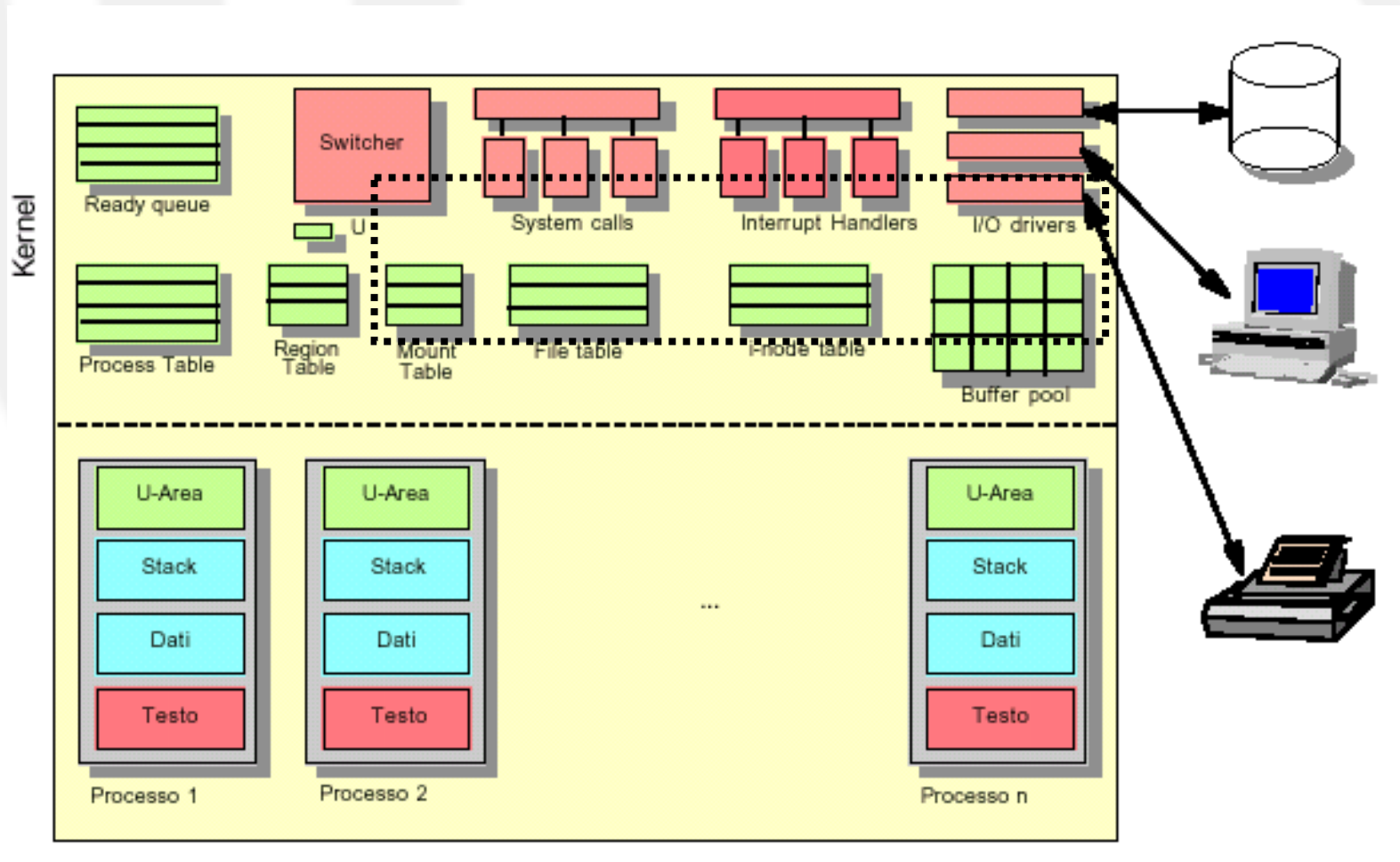


# **STRUTTURE DATI DEL FILE SYSTEM IN MEMORIA**

# File system: strutture dati

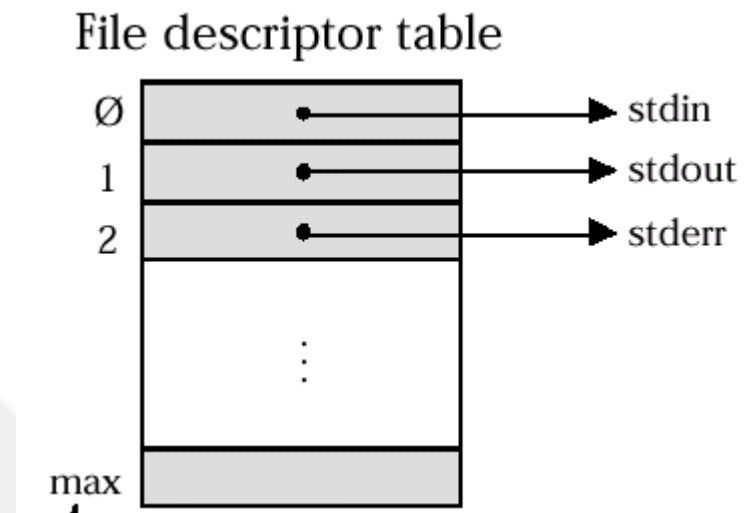
- Nella U-area di ogni processo
  - File Descriptor Table
- Globali a tutti i processi e residenti
  - File Table
  - I-node table
  - Mount Table

# File system: strutture dati



# File descriptor table

- Una tabella per processo, contenuta nella U-area
- Vettore di file descriptor
  - File descriptor = riferimento al file aperto
    - Usato dalle system call (es.: read, write) per accedere al file
  - Dimensione tabella = numero max di file apribili contemporaneamente per processo
  - Un file descriptor per ogni file aperto dal processo



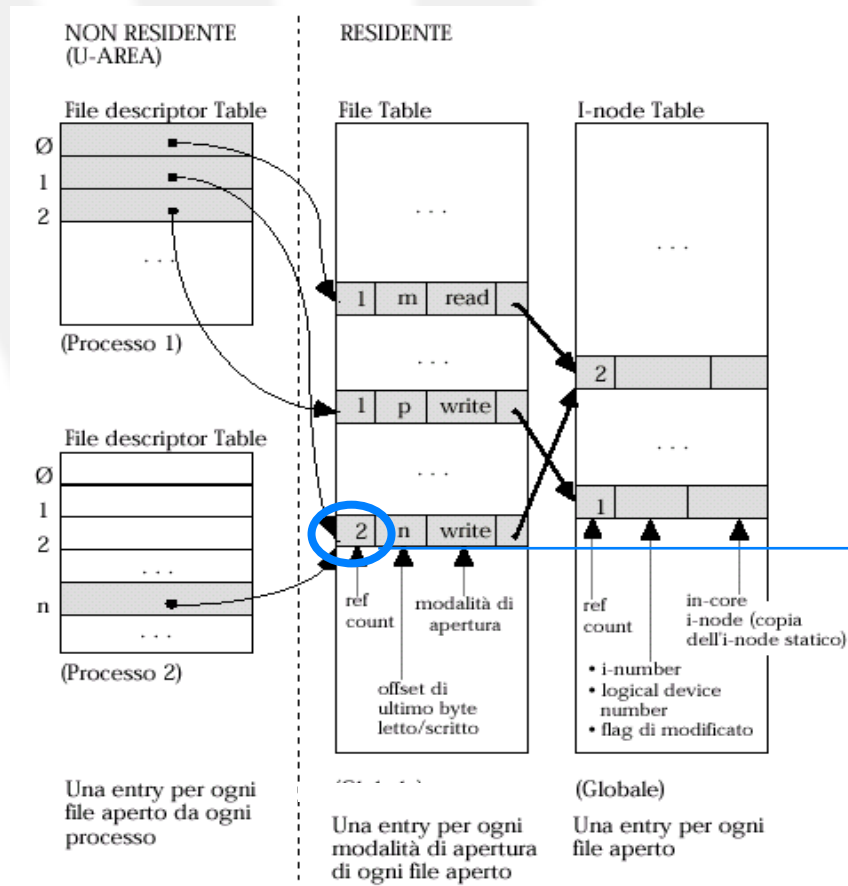
# File table

- File table contiene
  - Una entry per ogni modalità di apertura di ogni file aperto
  - Una entry contiene
    - Numero di riferimenti
      - Quanti processi hanno aperto il file
    - Offset di ultimo accesso
    - Tipo di apertura (r/w/a)

# I-node table

- I-node table contiene
  - Una entry per ogni file aperto
  - Una entry contiene
    - Numero di riferimenti
      - Numero di link
    - i-number
    - Numero di device associato
    - Flag di modificato per la sincronizzazione con la cache
    - Copia dell'i-node su disco (in-core copy)

# Accesso a file: figura d'insieme



**NOTA:** esiste una entry nella File Table per ogni entry nella File Descriptor Table

File Table serve per permettere la condivisione del puntatore di lettura/scrittura

(per es., `dup()` e `fork()`)

# Mount table

- Tabella sempre residente che contiene, per ogni file system montato
  - Major e minor number del dispositivo montato
    - Major number: numero che identifica il tipo del dispositivo ( il driver)
    - Minor number: numero che identifica il dispositivo specifico
  - Puntatore all'area superblocco del file system montato
  - Puntatore alla root del file system su cui è montato
  - Puntatore alla directory su cui è montato
- La mount table è aggiornata dai comandi (o system call) mount e umount



# open()

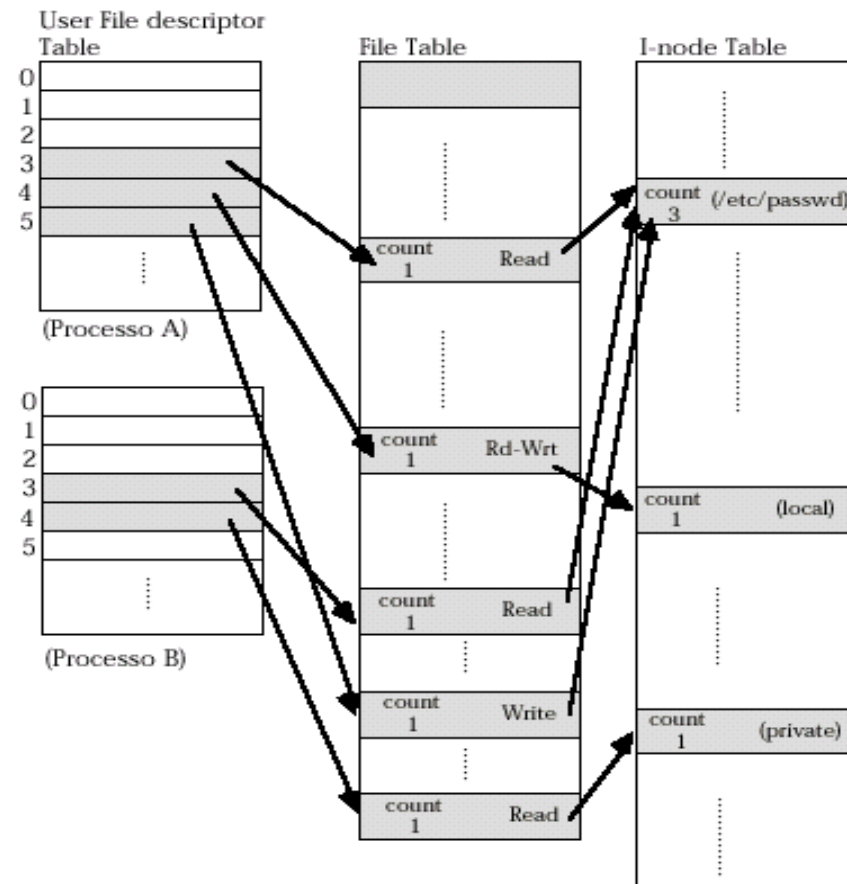
- *fd = open(path, mode)*
- Cerca l'i-node corrispondente a path
- Se il file non esiste o non è accessibile nella modalità mode restituisce al chiamante un codice di errore
- Altrimenti:
  - se il file non è già aperto: copia l'i-node nella in-core i-node table e la completa
  - crea una entry nella file table, e inizializza modalità di apertura, reference count, offset
  - crea una entry nella file descriptor table del processo utilizzando la prima entry libera
  - restituisce al chiamante l'indice di tale entry (fd)

## close()

- *s=close(fd)*
- Dealloca la entry della file descriptor table allocata da open()
- Se il reference count nella file table è >1:
  - decrementa e conclude
- Se il reference count nella file table è =1:
  - dealloca la entry nella file table
  - Se il ref. count delle inode table è 1
    - Libera l'in-core inode

# open - esempio

- Processo A
  - fd1=open("/etc/passwd",O\_RDONLY);
  - fd2=open("local",O\_RDWR);
  - fd3=open("/etc/passwd",O\_WRONLY);
- Processo B
  - fd1=open("/etc/passwd",O\_RDONLY);
  - fd2=open("private",O\_RDONLY);

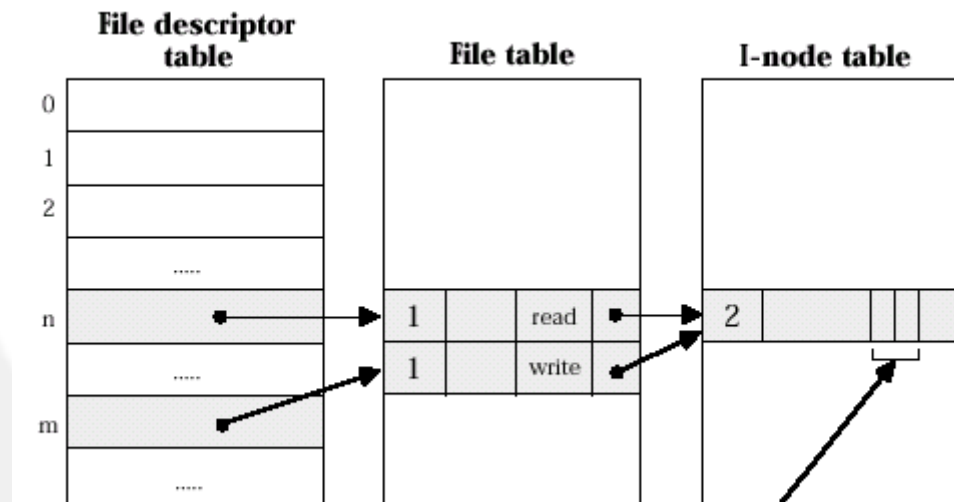


# read()

- $m = \text{read}(fd, buf, n)$
- Accede alla entry della file table a partire da fd e controlla la modalità di apertura
- Accede ed effettua il lock dell'in-core i-node
- Trasforma l'offset nella entry della file table in un indirizzo di disco:
  - (indirizzo blocco + offset all'interno del blocco) attraverso la tabella di indirizzamento dell'in-core i-node
- Legge i blocchi richiesti (attraverso la buffer cache) e copia gli n byte richiesti in buf
- Aggiorna l'i-node e ne effettua l'unlock
- Aggiorna l'offset nella entry della file table
- Restituisce al chiamante il numero di bytes letti m

# pipe()

- `s = pipe(fd)`
- Accesso simile a file
- Contenuto delle tabelle diverso



*Offset di r/w contenuto nella i-node table perché deve essere visto da entrambi i processi (no `lseek()`)*

```
fd_pipe  | n | m |
          | 0 | 1 |
```

# Buffer cache

- Cache dei blocchi su disco mantenuta dal kernel
  - Se il blocco e' nella cache, evita accesso a disco
  - Il kernel alloca una porzione della RAM (10%) per la buffer cache in fase di inizializzazione
- Struttura di un buffer:
  - Buffer header (intestazione)
    - Numero di device (file system)
    - Numero di blocco (blocco dati su disco)
    - Stato (locked/unlocked, valid/invalid, read/write, etc.)
    - Puntatori usati dall'algoritmo di allocazione dei buffer
      - (2) Puntatori alla free buffer list
      - (2) Puntatori alla hash queue (coda di hash)
  - Disk block
    - I dati del blocco corrispondente

# Buffer cache

- Buffer della cache sono organizzati in due “liste”
  - Lista dei buffer liberi (free buffer list)
    - Lista doppiamente linkata per mantenere l'ordine LRU
  - Hash queue
    - Per l'accesso ai buffer
    - Concettualmente una hash table
      - In pratica un insieme di code (default 64) doppiamente linkate
      - Chiave hash: coppia (#device,#blocco)
- Concetto: il kernel cerca
  - Uno specifico numero di blocco nella hash queue
  - Un blocco qualsiasi nella free list

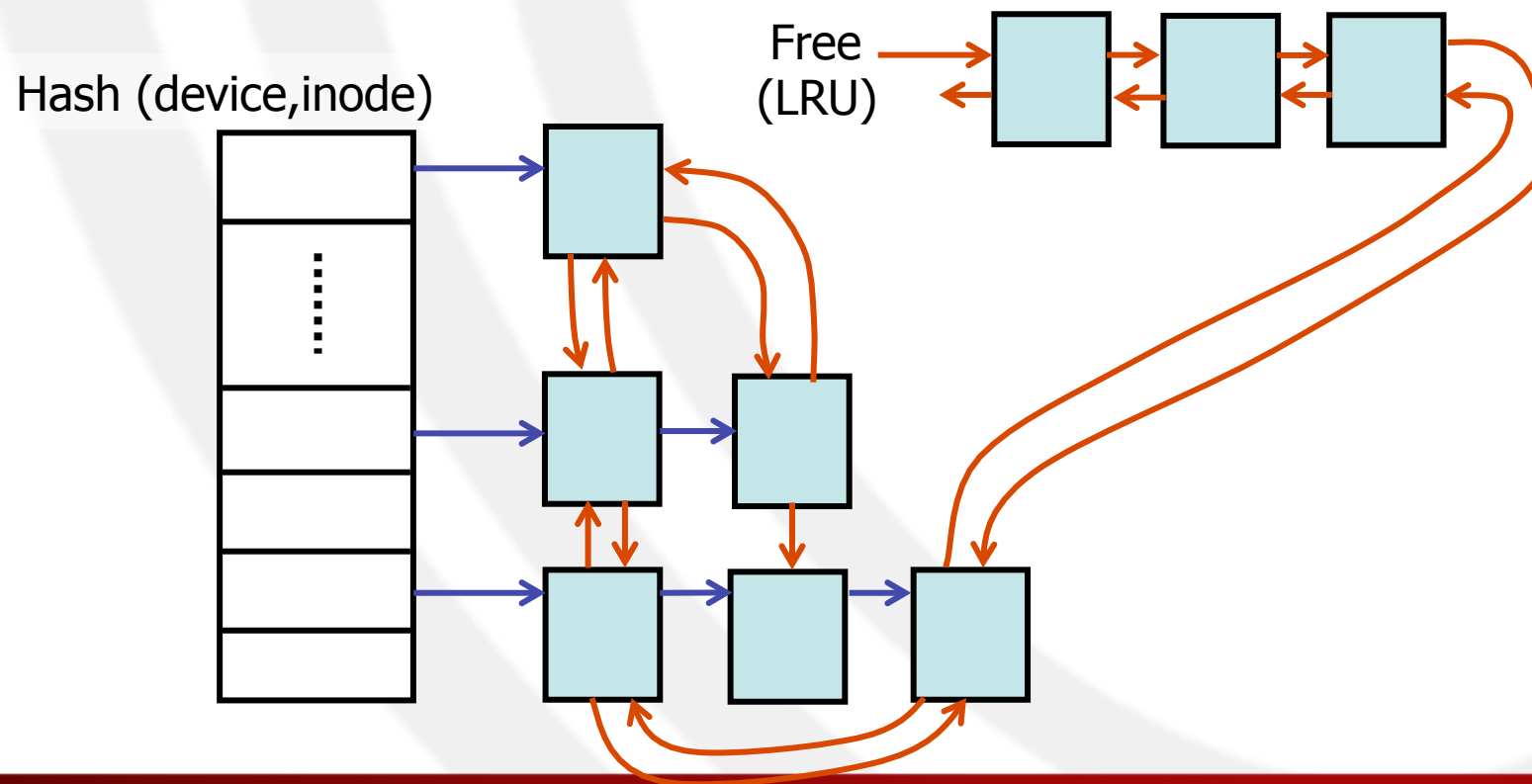
# Buffer cache

- Quando un processo vuole leggere/scrivere un blocco:
  - Si cerca nella buffer cache
    - Se non c'è, si sceglie un buffer nella buffer cache e il blocco viene letto dal disco
    - Se il blocco viene modificato si setta il dirty flag
      - Quando il buffer viene scelto per essere rimpiazzato, se il dirty flag è attivo il blocco viene scritto su disco
    - Quando il blocco è in uso viene bloccato (locked)
      - Un'operazione su un blocco locked deve attendere
  - I blocchi non locked vengono tenuti in una free-list ordinata secondo l'algoritmo LRU
    - Quando il kernel necessita di un buffer preleva quello LRU
    - Quando un buffer viene letto/scritto viene messo in fondo alla free-list



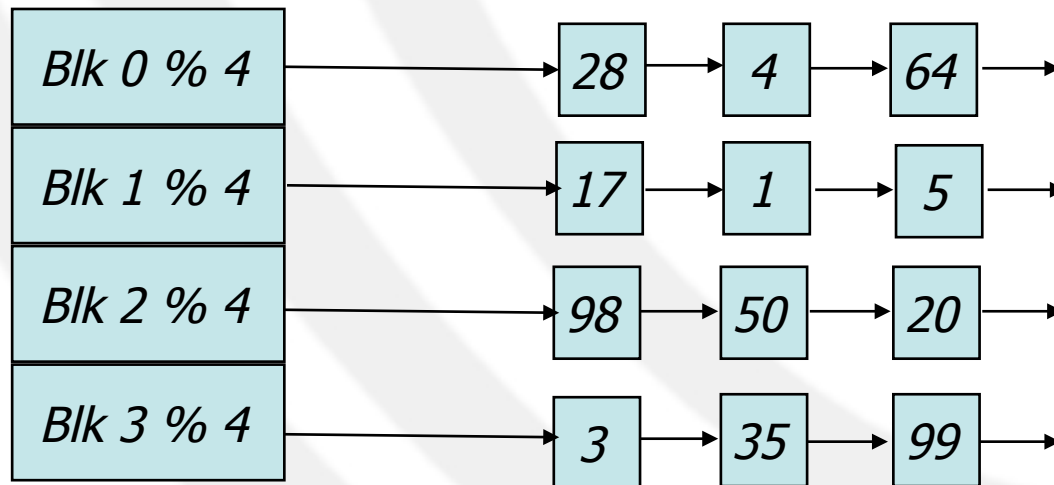
# Buffer cache

- Schema complessivo



# Buffer cache

- Esempio di hash queue:
  - Funzione di hash dipende solo da # di blocco
    - 4 code di hash  $\Rightarrow f(b) = b \% 4$
    - Vantaggio: tempo medio di ricerca da  $N/2$  a  $N/M$  (con  $M$  code)
    - Ogni buffer esiste in una ed una sola coda di hash
    - Può però esistere anche nella free list



# GESTIONE DELLA MEMORIA IN UNIX

# Gestione della memoria

- Sistemi UNIX moderni usano paginazione con memoria virtuale
- Per eseguire un processo è sufficiente che in memoria siano presenti almeno:
  - la sua u-area
  - la sua tabella delle pagine
- Se u-area e tabella delle pagine non sono in memoria, ve le porta il processo swapper
- Le pagine di testo, dati e stack sono portate dinamicamente in memoria, dal kernel, una alla volta quando servono (cioè a seguito di un page fault)

# Strutture dati

- Quattro strutture dati fondamentali:
  - Page table
    - [Disk block descriptor]
  - Page frame data table
  - Swap-use table

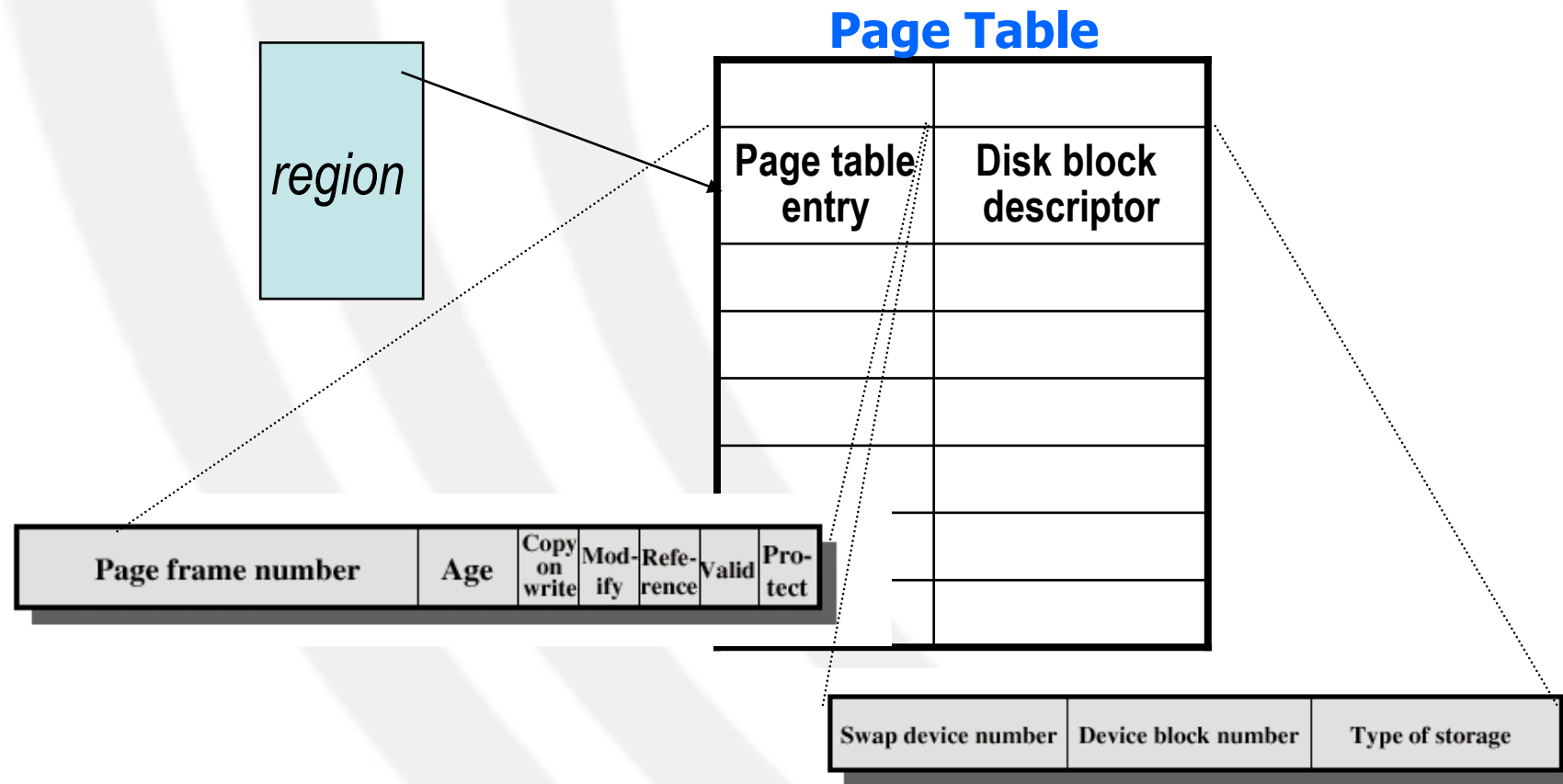
# Page table

- Regioni puntano a page table
  - Una tabella per ogni processo con una entry per ogni pagina del processo
- Contenuto della entry
  - Numero del frame corrispondente
  - Serie di bit
    - Reference: la pagina è stata recentemente usata
    - Dirty: la pagina è stata modificata
    - Permessi di lettura/scrittura
    - Valid: la pagina è in memoria principale
    - Age: da quanto tempo la pagina è in memoria senza essere usata
    - Copy on write: per gestire la consistenza dopo la scrittura di una pagina condivisa da parte di un processo (messo a uno quando più di un processo condivide la stessa pagina)
  - Disk block descriptor

# Disk block descriptor

- Associa alla pagina l'informazione che descrive la copia fisica su disco della pagina
- Contiene
  - Numero del dispositivo (disco) che contiene la pagina
    - Può esserci + di un dispositivo di swap
  - Blocco (del disco) su cui si trova la pagina
  - Tipo di supporto di memorizzazione per lo swap
    - Text su file system
    - Data su spazio di swap

# Page table – struttura complessiva





# Page frame data table (pfddata table)

- Descrive ogni frame della memoria fisica
- Indicizzata dal numero del frame
- Usata nel rimpiazzamento delle pagine
- Contiene:
  - Stato della pagina (libero/allocato)
  - Numero di processi che fanno riferimento alla pagina
  - Dispositivo e numero di blocco che contiene una copia della pagina
  - Puntatore ad altre entry della pfddata table

Page state	Reference count	Logical device	Block number	Pfddata pointer
------------	-----------------	----------------	--------------	-----------------

## Page frame data table (pfdata table)

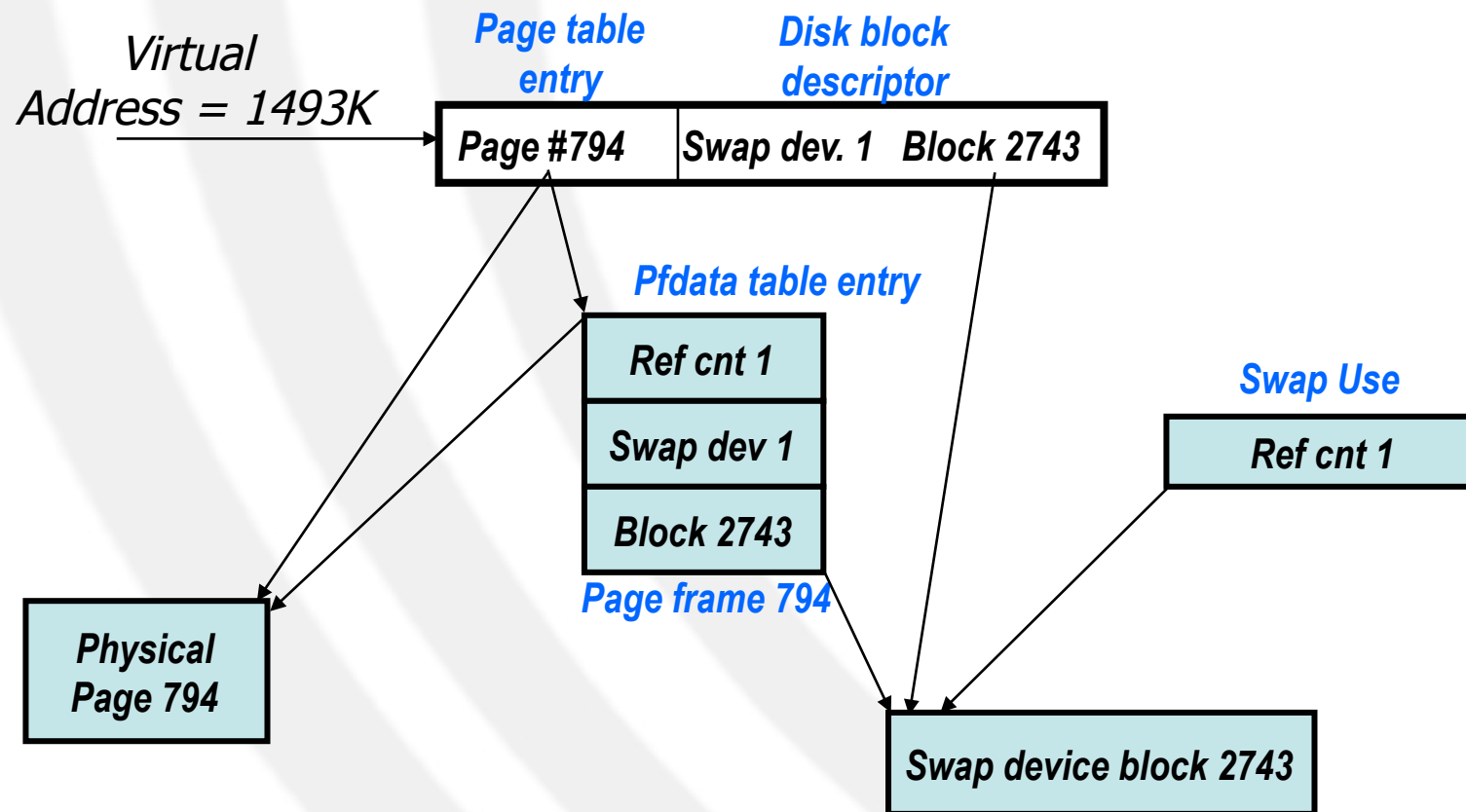
- Il kernel collega entry della pfdata table secondo una lista di frame liberi ed una lista di hash (tipo buffer cache)
- Allocazione di frame liberi:
  - Preleva la pagina LRU dalla testa della lista
  - Inserisce la nuova pagina nella lista di hash
    - funzione di hash basata su (device, blocco)

## Swap-use table

- Una tabella per ogni dispositivo di swap
- Una entry per ogni pagina presente sul dispositivo di swap
- Contiene:
  - Numero di entry delle tabelle delle pagine che puntano ad una pagina presente sul dispositivo di swap
  - Identificatore della pagina

Reference count	Page/storage unit number
--------------------	-----------------------------

# Vista d'insieme

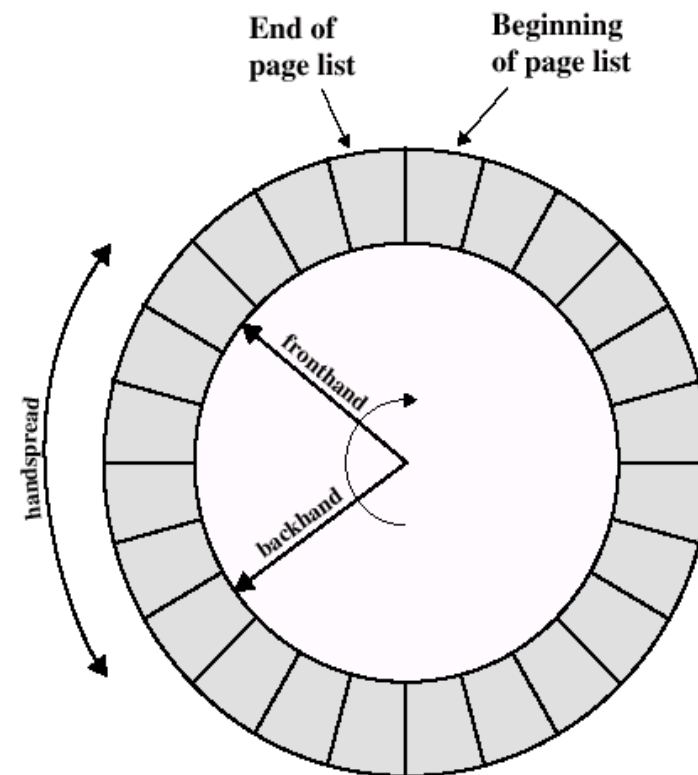


# Rimpiazzamento delle pagine

- L'algoritmo di rimpiazzamento delle pagine in parte implementato nel kernel, in parte nel processo pagedaemon
- Usa la pfdata table
- Algoritmo: variante del clock algorithm
  - Algoritmo dell'orologio a due lancette (two-hands clock algorithm)

# Rimpiazzamento delle pagine

- Usa reference bit
  - Messo a 0 quando la pagina viene portata in memoria
  - Messo a 1 al primo riferimento
- Prima lancetta:
  - Mette a 0 il reference bit
- Seconda lancetta:
  - Se bit=1, riferimento “recente”
  - Se bit=0, inserita in lista di candidati per il rimpiazzamento
- Parametri
  - Scanrate: scansione [pag/sec]
  - Handsread: distanza tra le lancette



## Il processo pagedaemon

- Demone di sistema che viene eseguito periodicamente, per controllare il numero di pagine libere in memoria
  - se è troppo basso, libera qualche pagina, con l'algoritmo di rimpiazzamento visto prima
  - se è ok, ritorna inattivo
- Tipicamente, PID=2

# Il processo swapper

- Demone di sistema che:
  - Si attiva se troppo spesso il numero di pagine libere è sotto la soglia
  - Quando si attiva:
    - rimuove uno o più processi dalla memoria (in genere, quelli inattivi da più tempo)
    - verifica periodicamente l'esistenza di processi pronti su disco e, se possibile, li carica in memoria (solo U-area e tabella delle pagine)
      - in genere, quelli fuori da più tempo, a meno che non siano troppo grossi
- Tipicamente PID=0