# IL SISTEMA UNIX

## Parte Terza

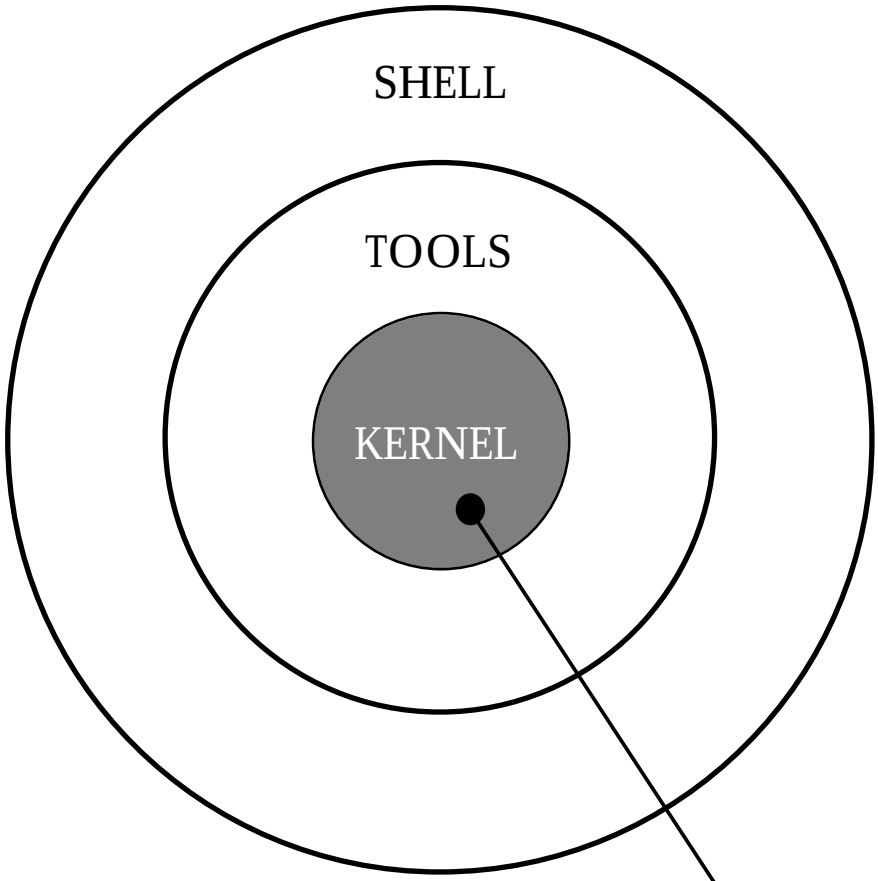
## Struttura interna

**Roberto Polillo**

**Corso di Sistemi Operativi**
**Corso di Laurea in Informatica**
**Università di Milano**

**Ultimo aggiornamento: giugno 1997**

# LIVELLO DI VISIBILITÀ




Struttura interna

# INDICE

# RIFERIMENTI

- G.Glass
  Unix for Programmers and Users
  Prentice Hall, 1993

## Per approfondimenti:

- M.J. Bach
  The Design of the Unix Operating Systems[1]
  Prentice Hall, 1986
  Ed. italiana: Unix: Architettura di sistema
  Gruppo Editoriale Jackson,1988

- U.Vahalia
  Unix Internals - The New Frontiers[2]
  Prentice Hall, 1996

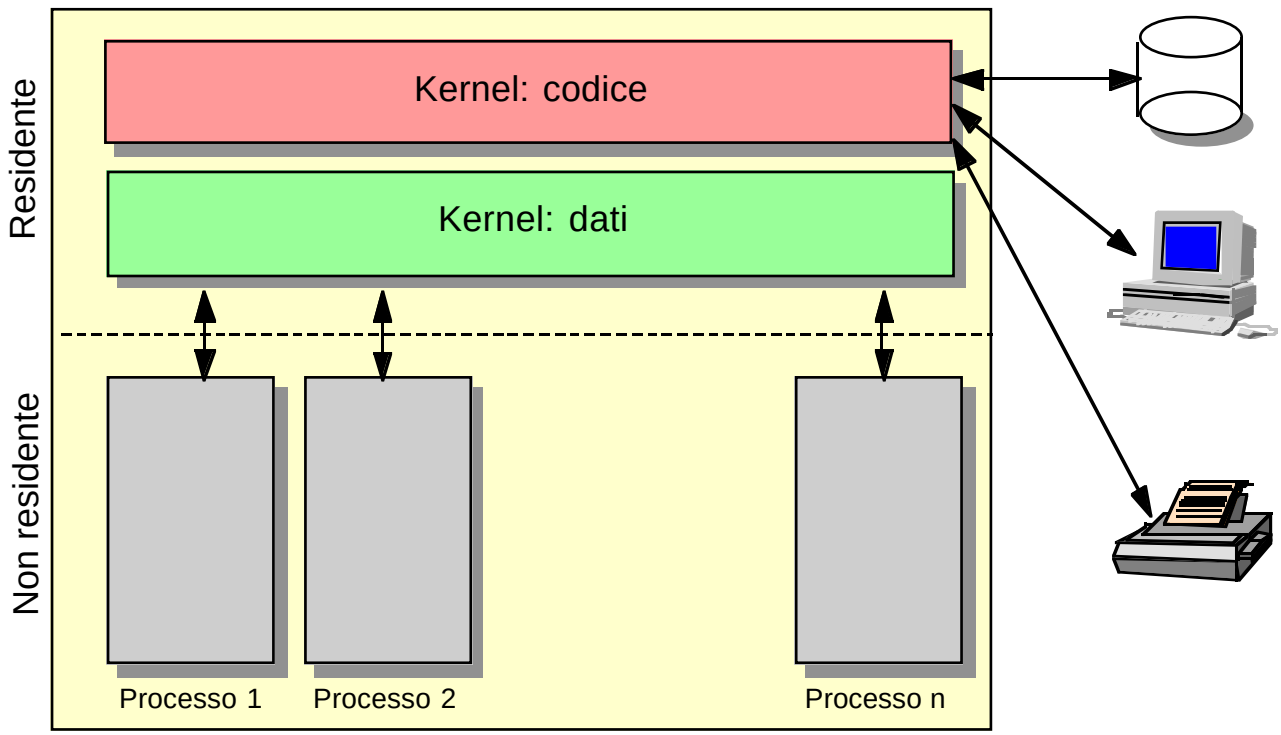
1 Basato essenzialmente su System VR2-R3

2 Confronto di versioni recenti di Unix (SVR4, 4.4BSD, ...)

# 1. INTRODUZIONE

# IL KERNEL

Il kernel è il sistema operativo vero e proprio, che gestisce direttamente l'hardware della macchina, è sempre residente in memoria e fornisce i necessari servizi ai processi in esecuzione




3

[3] Il kernel è normalmente in `/unix`

# PROCESSI

- Nei sistemi Unix tradizionali, ogni processo esegue un singolo programma, ed ha un singolo *thread*: un unico *program counter* specifica la prossima istruzione da eseguire
- Ogni processo è indipendente dagli altri, e può interagire con essi solo mediante opportune system call
- Ogni utente può avere diversi processi attivi

# DEMONI

Sono processi di sistema, che partono automaticamente in background quando il sistema è attivato

**Esempi:**

- gestione del tempo (`cron`)

  si attiva ogni minuto per controllare se c'è qualcosa da fare, nel qual caso lo fa, e si sospende fino al prossimo controllo (es.: back-up periodici)

- gestione delle code di stampa

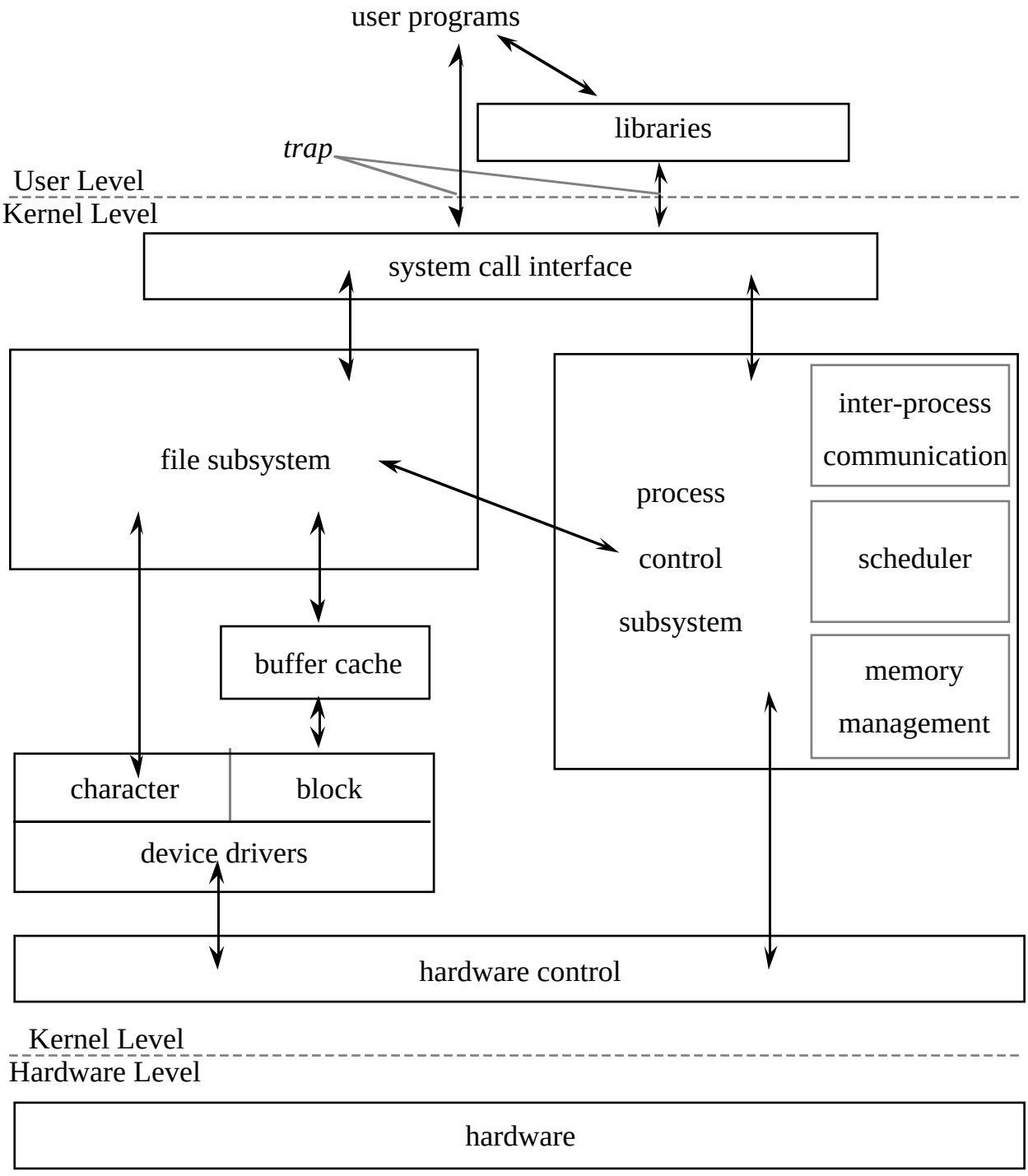
- gestione della posta in partenza e in arrivo

- gestione della memoria (pagedaemon, swapper)
- ...

# STATI (PRINCIPALI) DI UN PROCESSO


Unix è un sistema **time-sharing**: un processo esegue per un quanto di tempo (es. 100 msec) o finchè non si blocca in attesa di un evento

# ARCHITETTURA DEL KERNEL



# 2. GESTIONE PROCESSI

# SPAZI DI INDIRIZZAMENTO

I sistemi Unix recenti usano la **memoria virtuale**

Pertanto, ogni processo ha un suo proprio **spazio virtuale di indirizzi**, e non può in alcun modo accedere allo spazio di indirizzi di un altro processo

# MODALITÀ DI ESECUZIONE

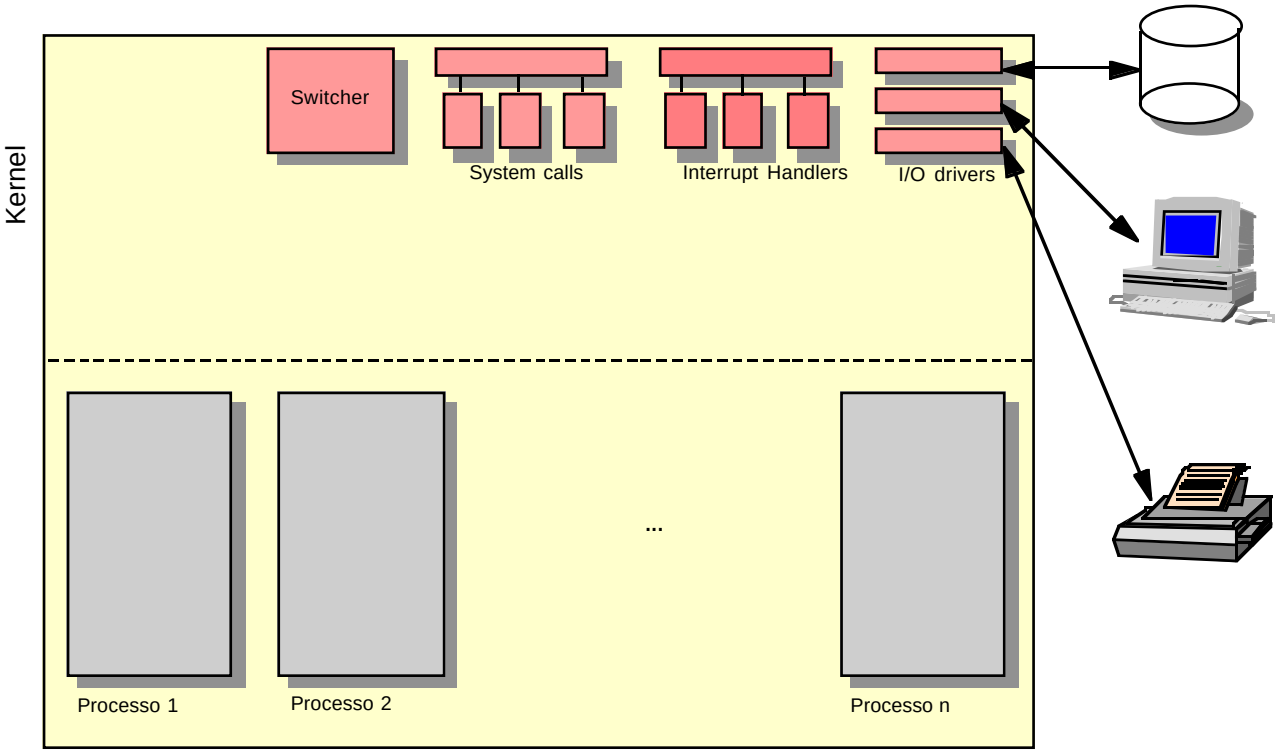
Un programma può essere eseguito in due modalità, distinguibili via hardware: **kernel mode** e **user mode**

Il kernel viene sempre eseguito in kernel mode, mentre il codice degli altri programmi viene eseguito in user mode

La distinzione ha scopo di **protezione**, infatti in user mode:

- non è accessibile lo spazio di indirizzi del kernel
- non sono eseguibili alcune istruzioni particolari

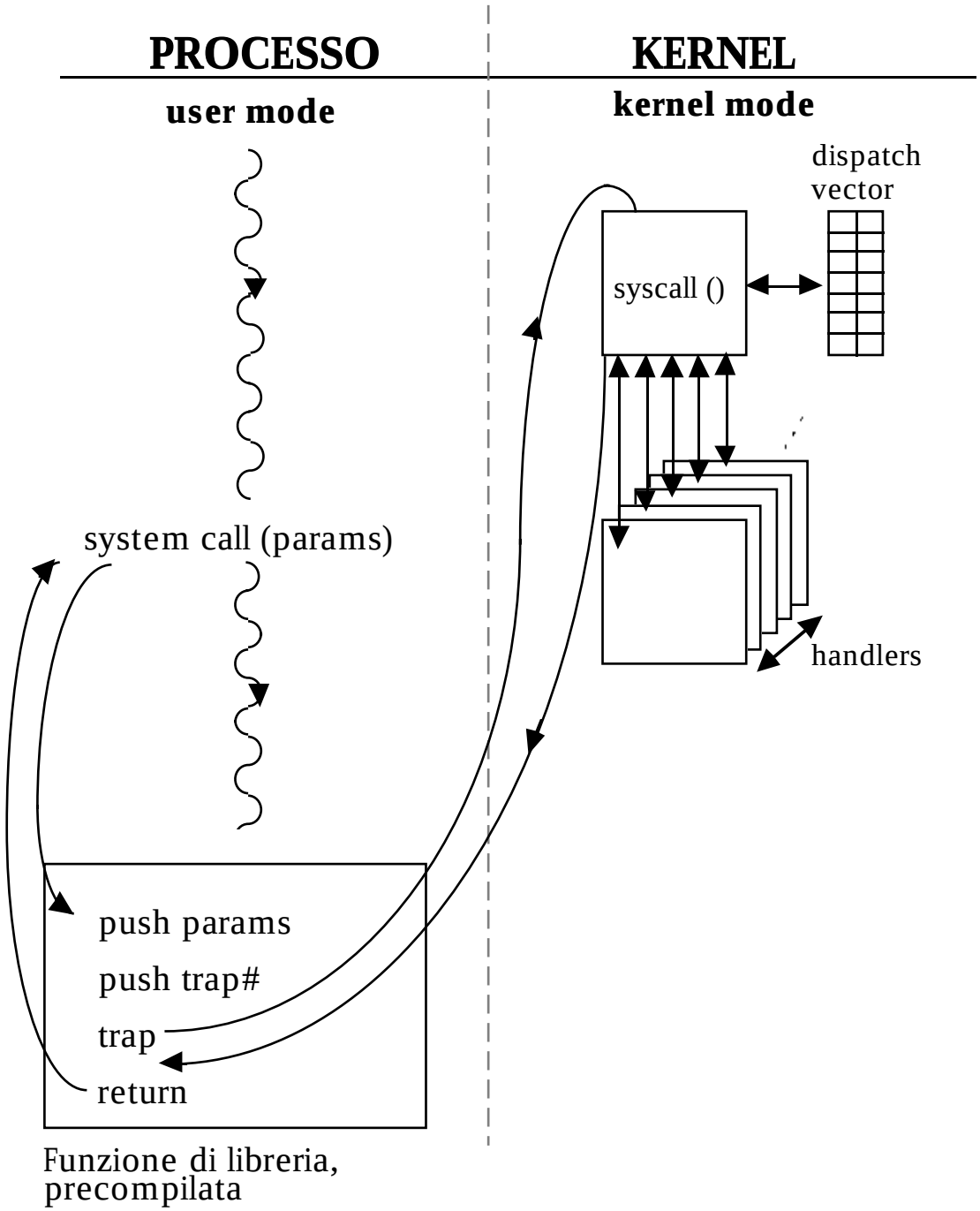
# KERNEL: ALCUNE ROUTINES PRINCIPALI




- switcher (è lo scheduler che seleziona il processo da mandare in esecuzione)
- routines per la gestione di tutte le system call e di tutte le interruzioni
- drivers per tutte le periferiche

# SYSTEM CALL: COME FUNZIONANO

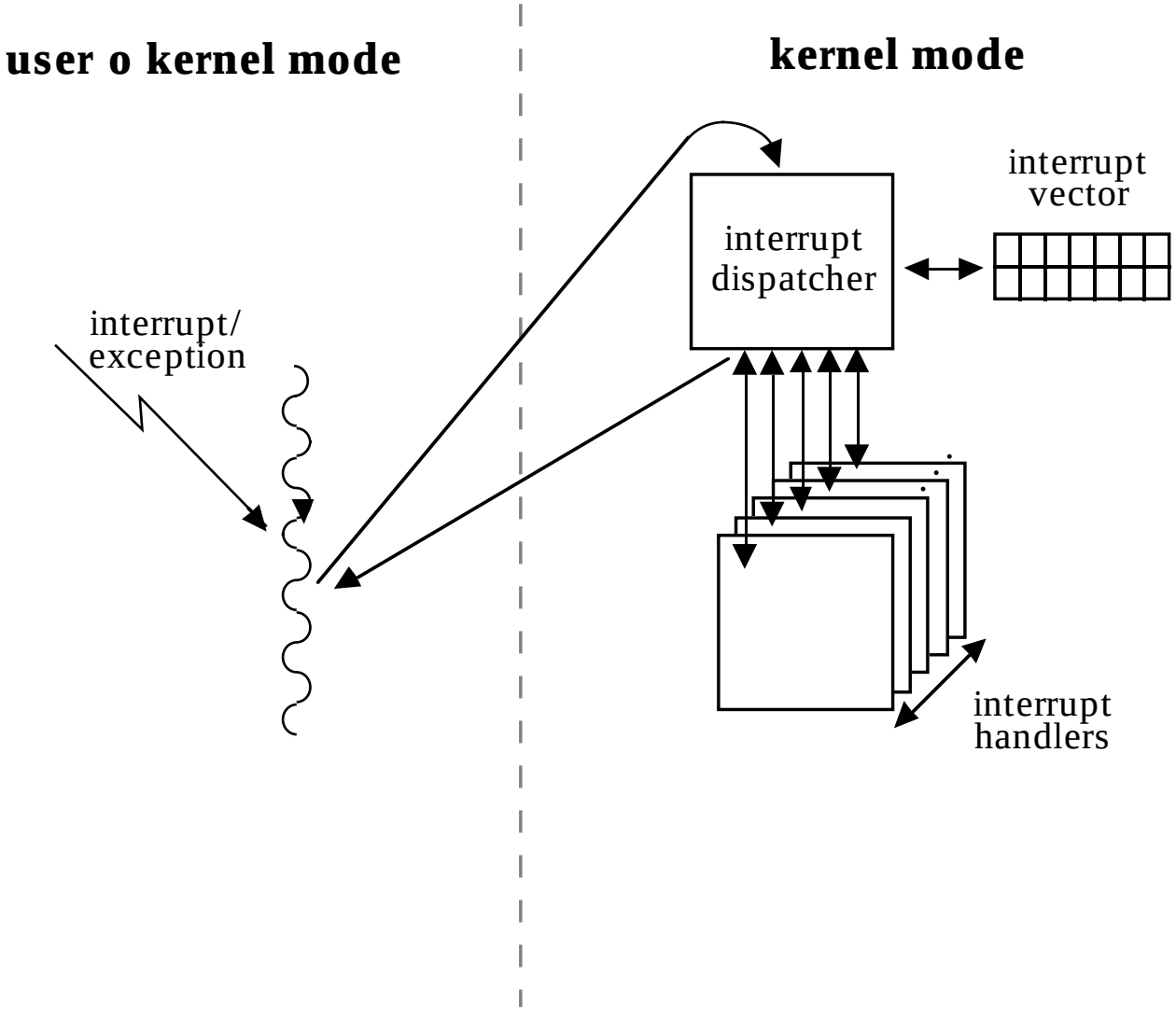
Una system call provoca la esecuzione di una **trap** per attivare in modo veloce la routine relativa



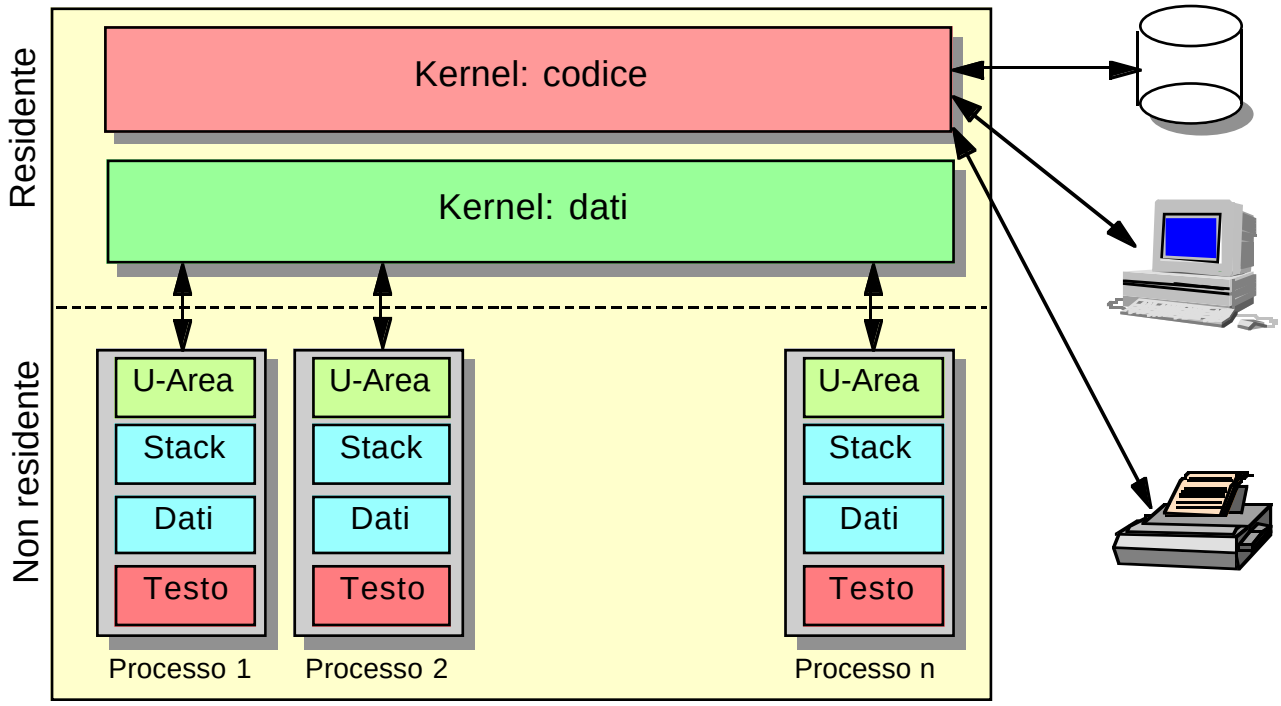
# INTERRUZIONI

Un'altra importante funzione del kernel è il trattamento di eventi che occorrono durante la esecuzione del processo corrente:

- **interruzioni** generate da device periferici
- **eccezioni** generate dall'hardware (es. stack overflow, divisione per zero, ...)



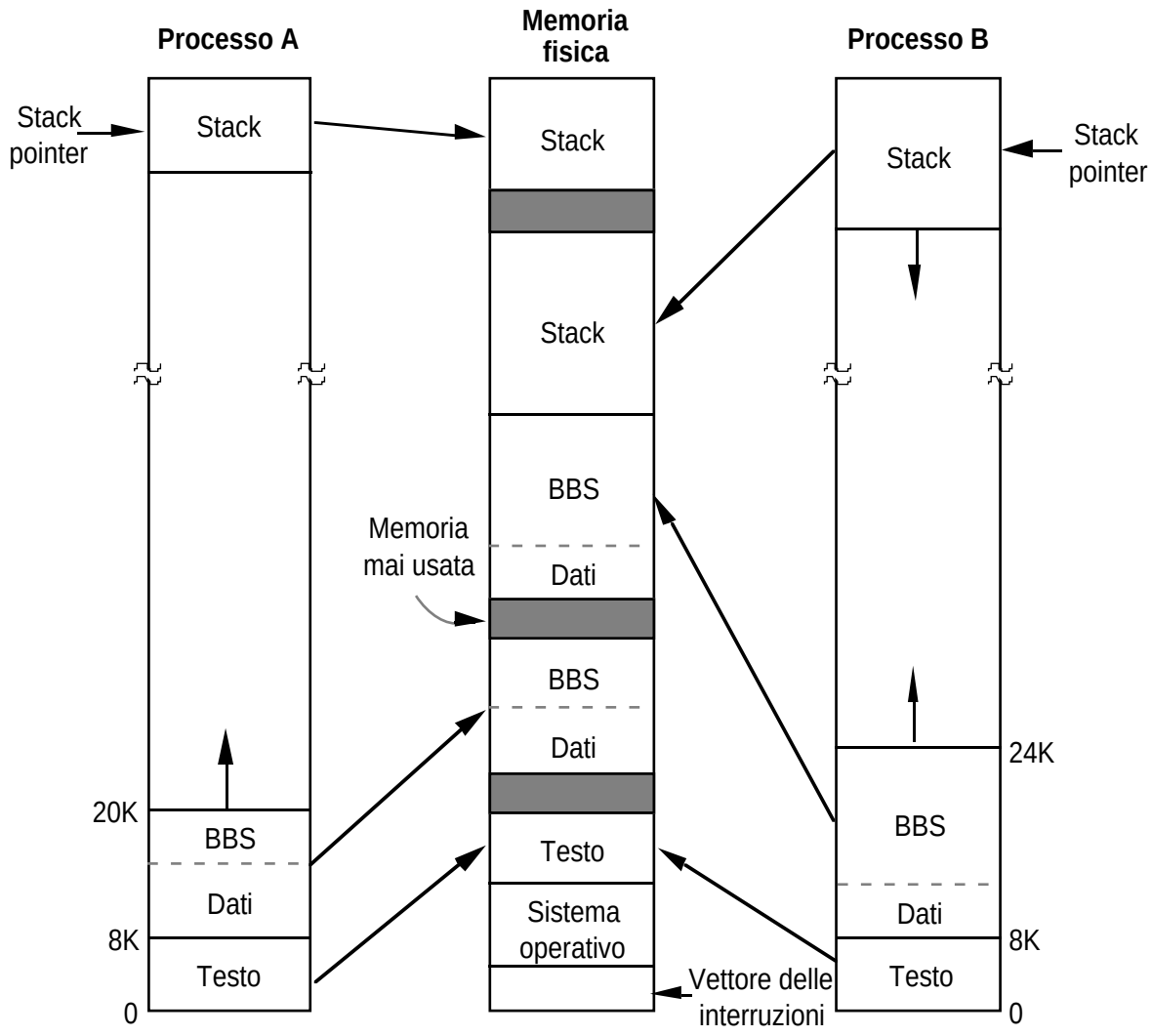
# IMMAGINE DI UN PROCESSO




Ogni processo ha uno spazio degli indirizzi formato da tre segmenti (“regions”):

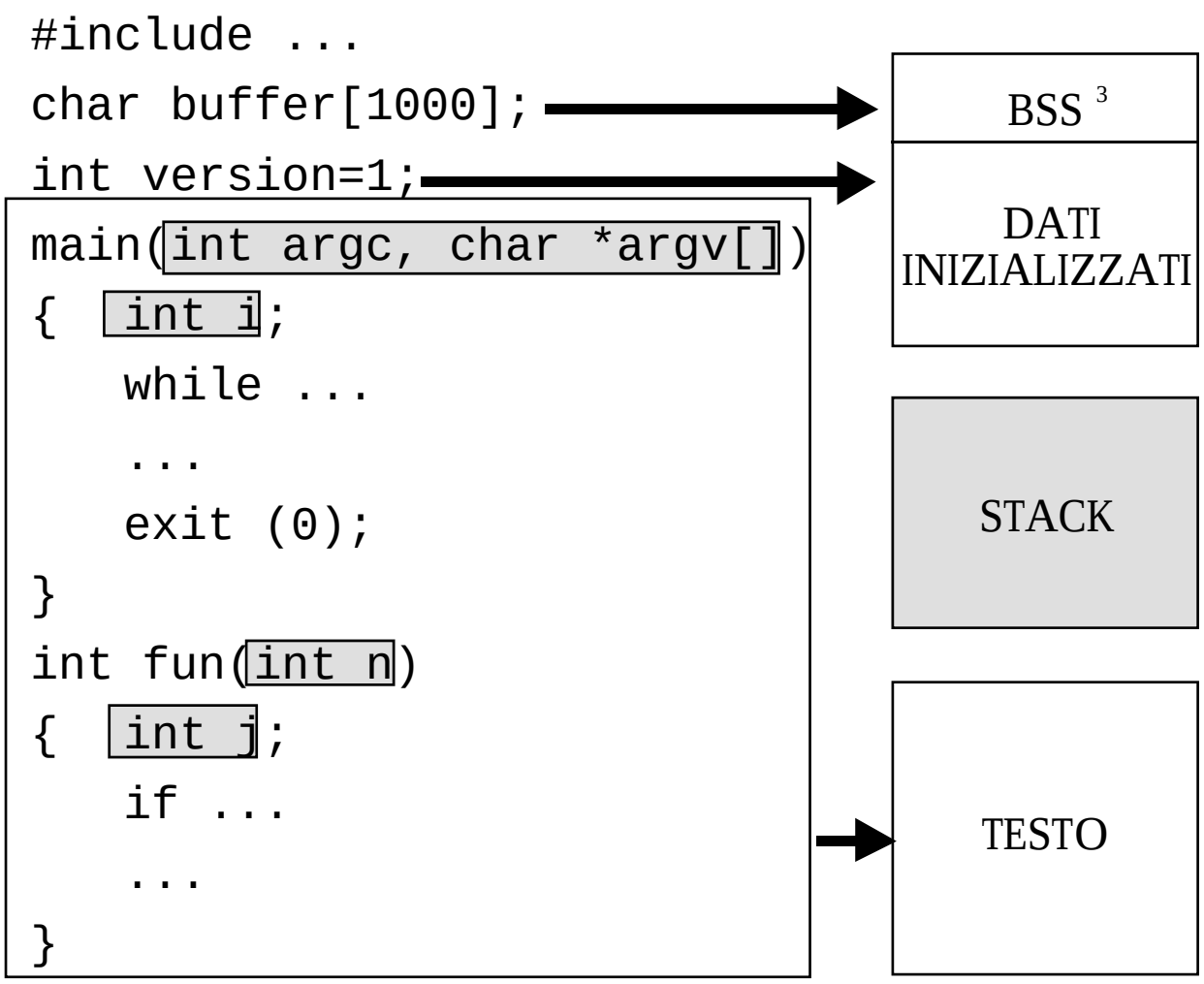
- **testo**: codice eseguibile, rientrante (condiviso)
- **dati**: inizializzati e non
- **stack**

# IMMAGINE DI UN PROCESSO: ESEMPI




I dati non inizializzati si chiamano BSS, da “Block Started by Symbol” (pseudo istruzione assembler IBM 7090)

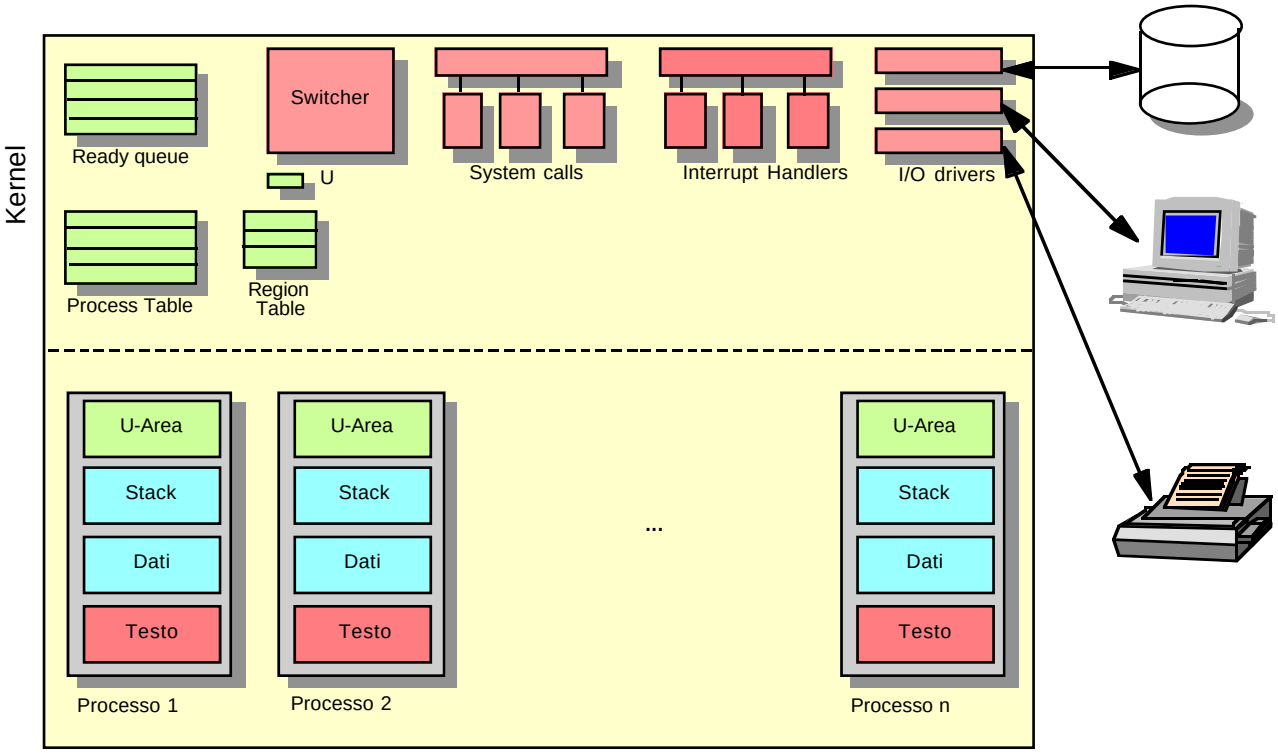
# ESEMPIO:
# CHE COSA GENERA IL COMPILATORE C




: allocati nello stack a run-time

[3] Il compilatore genera solo la dimensione: lo spazio verrà azzerato dal kernel all'esecuzione

# TABELLE PER LA GESTIONE DEI PROCESSI



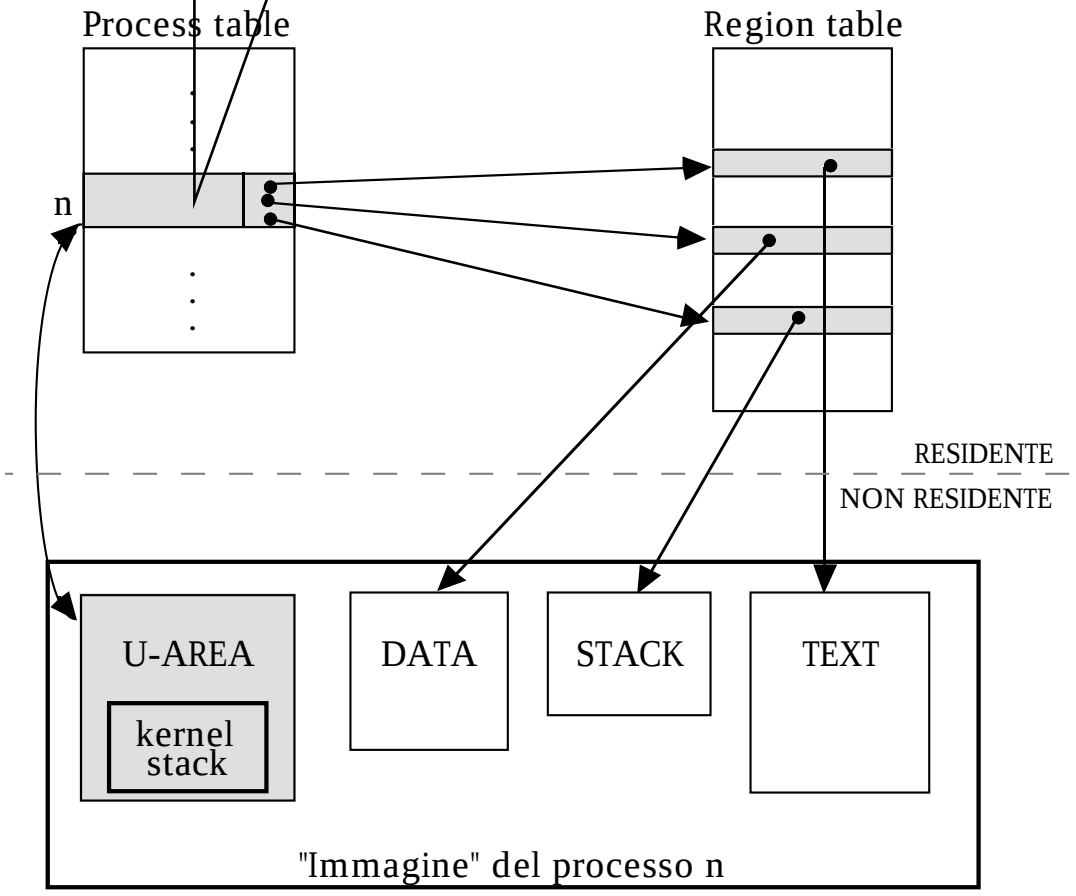

- **Process Table**: informazioni (residenti) sui processi
- **U-area**: informazioni (non residenti) sul processo
- **Region Table**: indirizzi delle region
- **Ready Queue**: code dei processi pronti
- **U**: puntatore al processo running

# PROCESS TABLE

Una entry per ogni processo (allocata alla creazione del processo, e deallocata alla sua terminazione)

Contiene informazioni che servono anche quando il processo non è attivo:

- **PID, PID del genitore, UID del proprietario**
- **stato del processo e flag di residenza**
- **dimensioni del processo**
- **parametri di schedulazione** priorità del processo, tempo di CPU usato, tempo di attesa, ...
- **maschere per i segnali** (che specificano come trattare i vari segnali)
- **evento** su cui il processo è in attesa
- .....

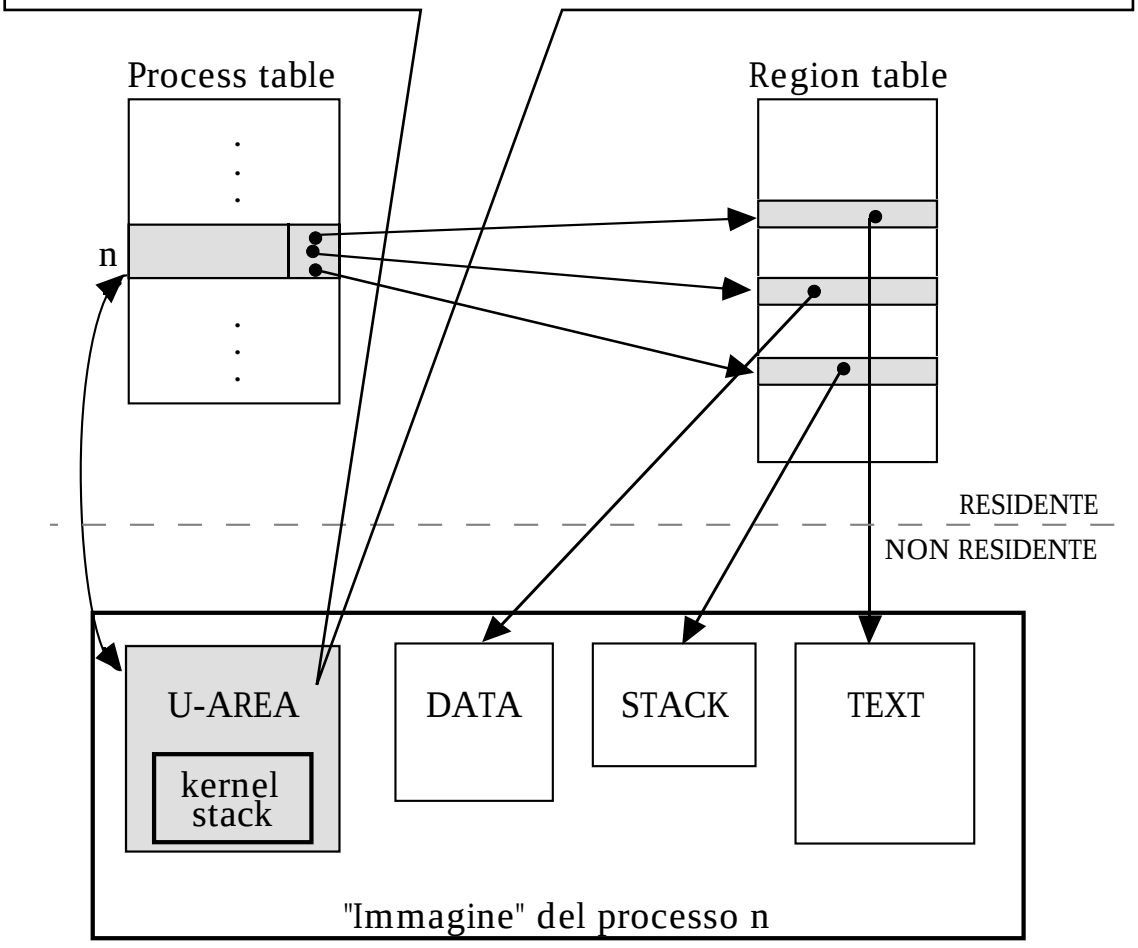


# U-AREA (O STRUTTURA UTENTE)

È una estensione della Process Table entry, e contiene informazioni che servono solo quando il processo è in esecuzione

Viene utilizzata solo dal kernel

Campi principali:
- Area salvataggio registri e program counter
- Informazioni relative alla system call corrente
- Puntatore alla Process table entry
- UID reale ed effettivo
- 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

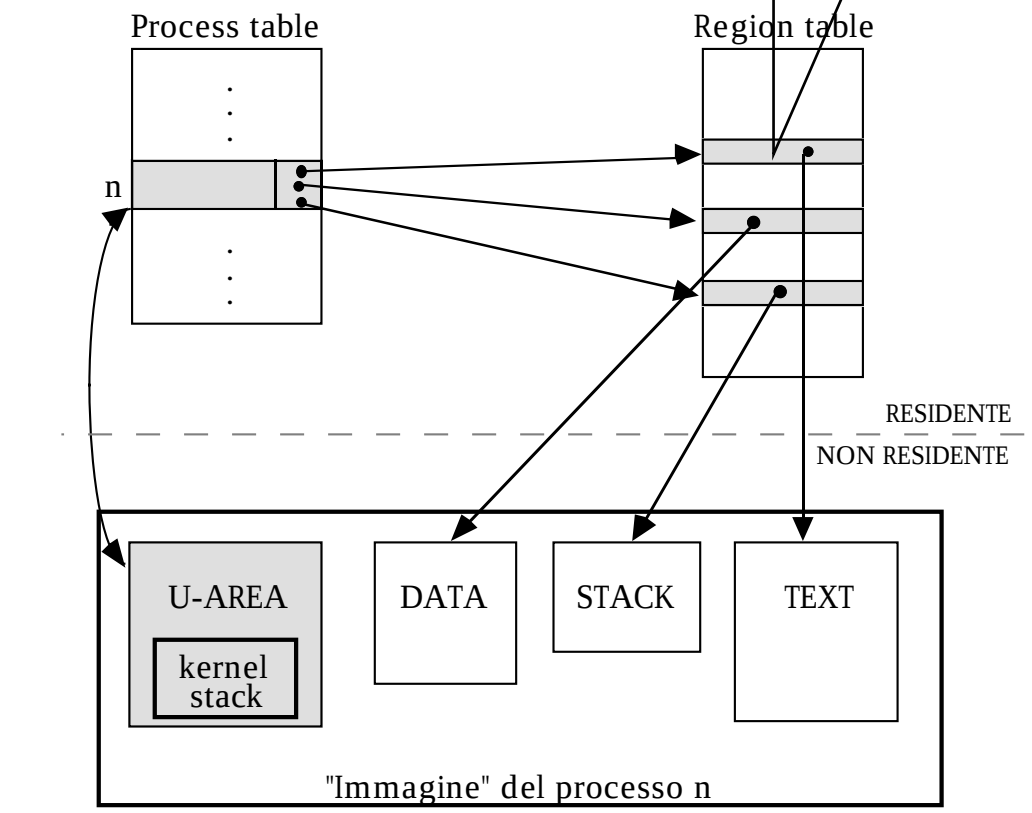


# REGION TABLE

Una entry per ogni region allocata alla creazione della region, deallocata alla terminazione dell'ultimo processo che la usa

Campi principali:

- indirizzo di memoria
- indirizzo su disco (per region swappate)
- dimensioni della region
- numero di riferimenti da processi



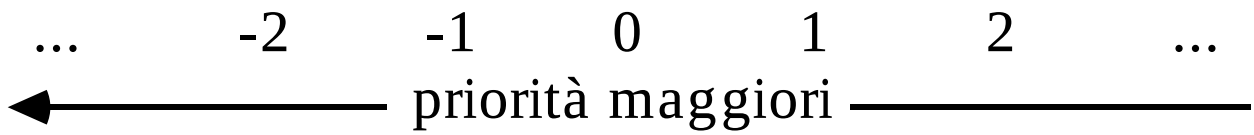

- La indirettezza via Region table permette di avere regioni testo condivise
- Il kernel vede solo la u-area del processo in esecuzione, e da questa accede alle altre strutture

# CHE COSA FA LA `fork`

- alloca una entry nella Process table per il nuovo processo
- assegna un PID unico al nuovo processo e inizializza i campi della Process table entry
- crea una copia della immagine del processo padre (il testo non viene duplicato, ma si incrementa un reference count)
- incrementa opportuni contatori per i file aperti
- inizializza i contatori di accounting nella u-area del nuovo processo
- pone il nuovo processo nello stato di pronto
- restituisce il PID del figlio al padre, e 0 al figlio

# SCHEDULAZIONE: PRIORITÀ

- Ogni processo ha un **livello di priorità**:



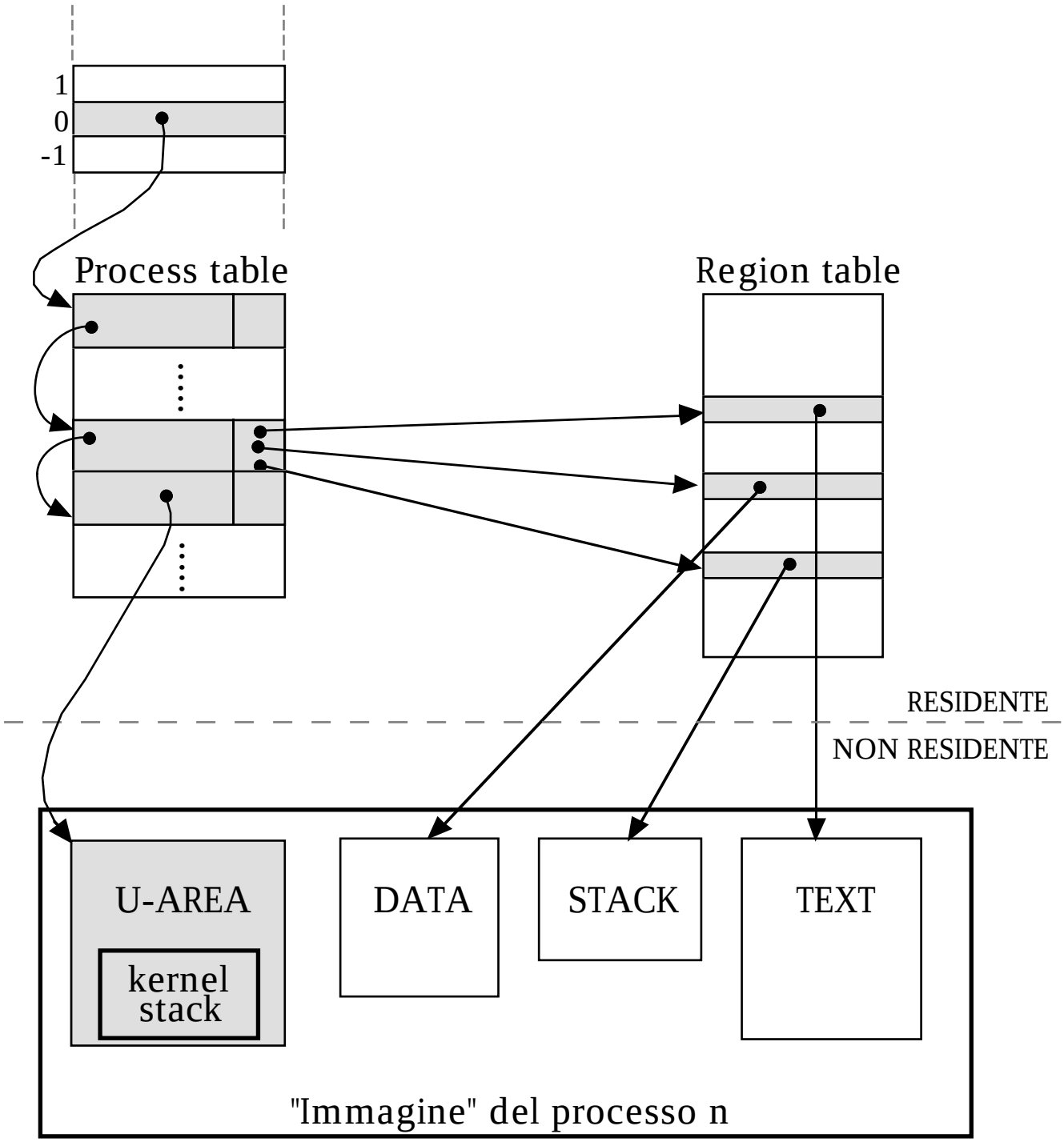

- Le priorità vengono attribuite dinamicamente

- Processi che eseguono in modo kernel hanno priorità maggiore di quelli in modo user (così restano il meno possibile in modo kernel)

- Processi di pari priorità sono schedulati a turno ("round-robin")

# READY QUEUE

Un processo ready viene inserito in una **coda** associata al suo livello di priorità



# CALCOLO DELLE PRIORITÀ

La priorità di un processo ha una componente statica (attribuita tramite il comando `nice`) e una dinamica che dipende dal tempo di CPU ricevuto:

- Ad ogni tic del clock viene incrementato il contatore di utilizzo della CPU nella Process table entry del processo in esecuzione

- Ogni secondo, le priorità di tutti i processi vengono ricalcolate secondo la formula:

  nuova priorità = base + utilizzo CPU/2

- La base è di solito 0, ma può essere incrementata con il comando `nice`, che *peggiora* la priorità

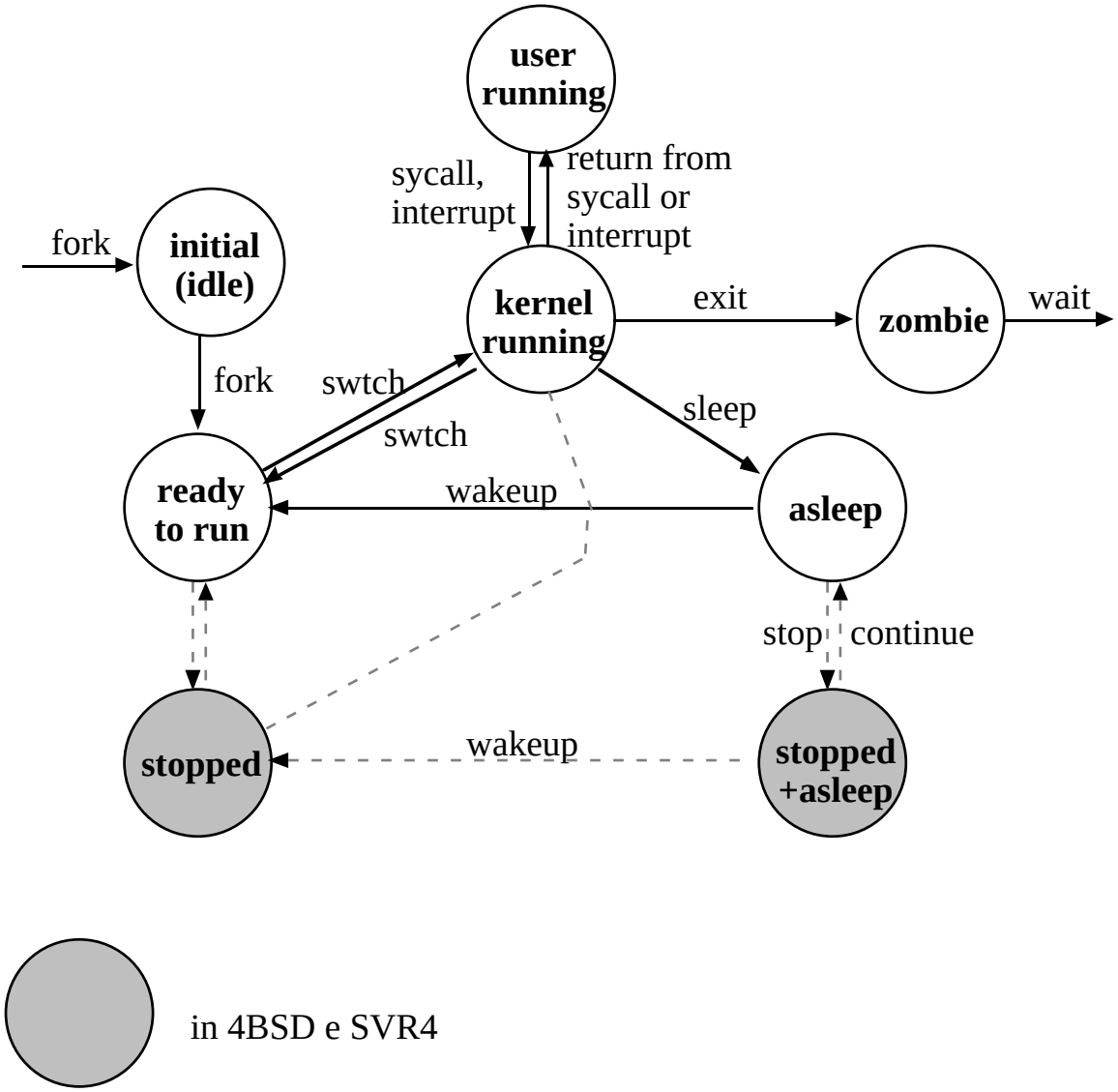
- In tal modo, vengono “premiati” i processi pronti che non hanno avuto molta CPU, e “penalizzato” quello running

# SCHEDULAZIONE: SINTESI

L’algoritmo di schedulazione dei processi pronti è stato progettato per fornire buoni tempi di risposta ai processi interattivi

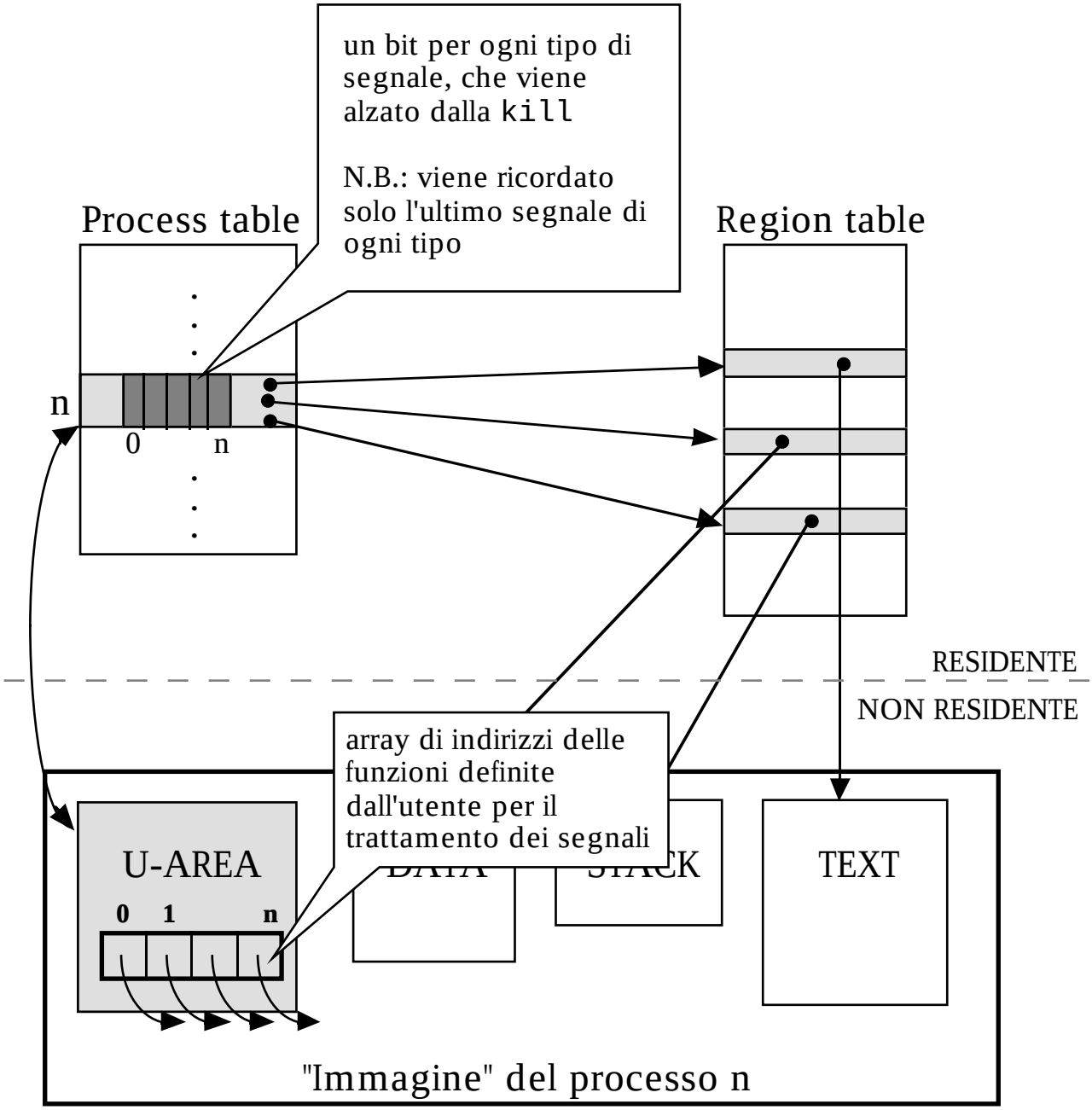
In pratica: i processi CPU-bound sono serviti solo quando tutti i processi I/O-bound e quelli interattivi sono bloccati

# STATI DI UN PROCESSO: PIÙ IN DETTAGLIO...




idl: image definition and loading

# STRUTTURE PER LA GESTIONE DEI SEGNALI




- Il kernel tratta i segnali occorsi solo quando un processo ritorna da kernel mode a user mode (quindi, un segnale non ha effetto istantaneo su un processo running in kernel mode)

# GESTIONE DEI SEGNALI

- **signal**

  assegna all'apposito array nella U-area del processo l'indirizzo dell'handler del segnale (oppure 0 se ignore, 1 se default)

- **kill**

  alza il bit corrispondente al segnale specificato nel descrittore del processo destinatario.

  (Se il processo è asleep a una priorità opportuna, viene svegliato affinchè possa trattare il segnale)

- **trattamento del segnale**

  E' asincrono: i segnali pendenti per un processo vengono "sentiti" quando si ritorna al processo da una system call o quando il processo entra/esce nello/dallo stato asleep

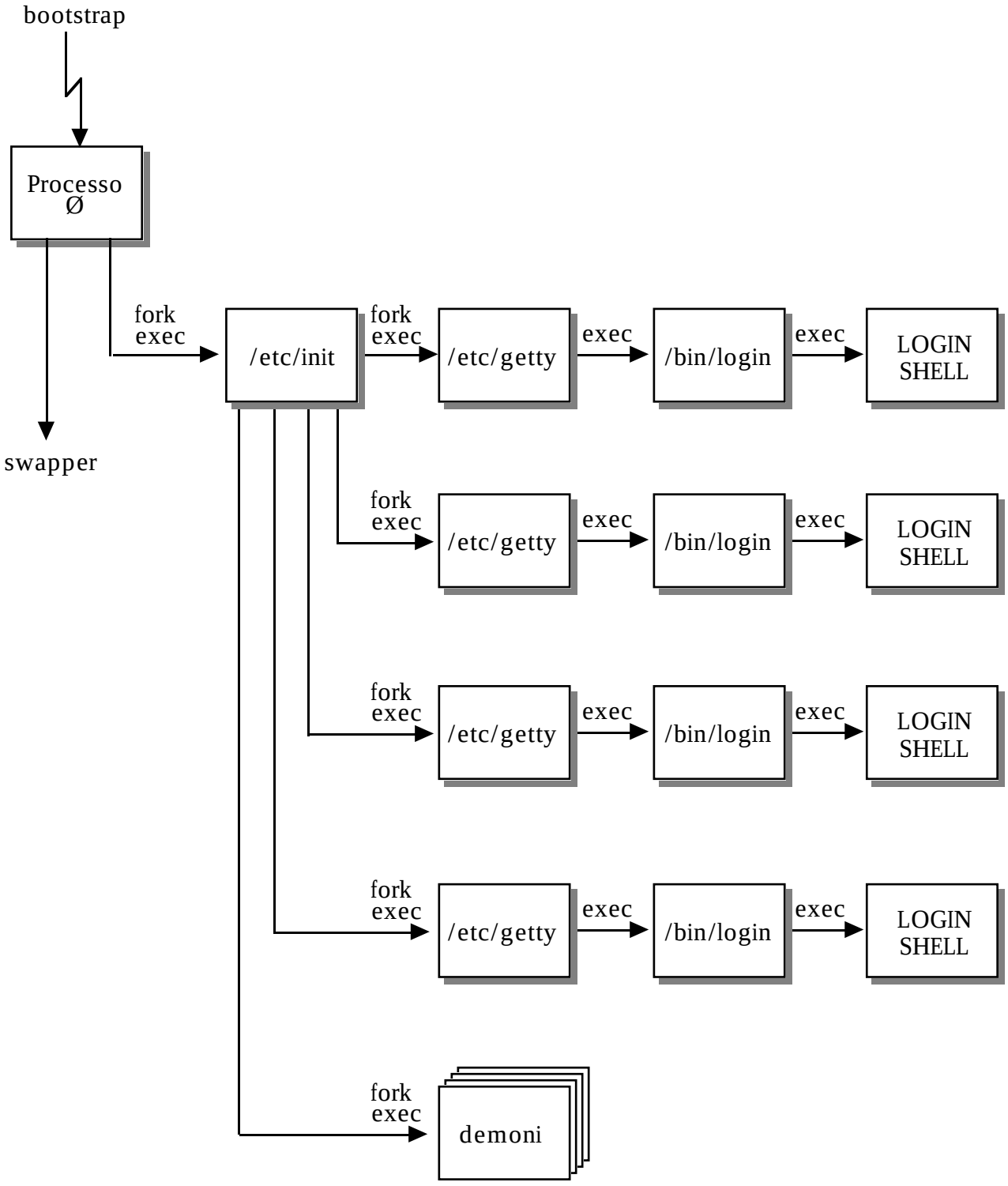
# TERMINAZIONE DI UN PROCESSO

- **exit**

  deposita l'exit code in apposito campo della process table, pone il chiamante nello stato zombie, dealloca le sue regioni, e invia al padre un segnale di SIGCHLD

- **wait**
  - se il chiamante non ha figli, restituisce un codice di errore
  - se il chiamante ha dei figli zombie, ne seleziona uno a caso, ne elimina il descrittore dalla Process Table, e ne restituisce al chiamante il PID e l'exit code
  - altrimenti sospende il chiamante

# INIZIALIZZAZIONE DEL SISTEMA



# Memory management

# PAGINAZIONE

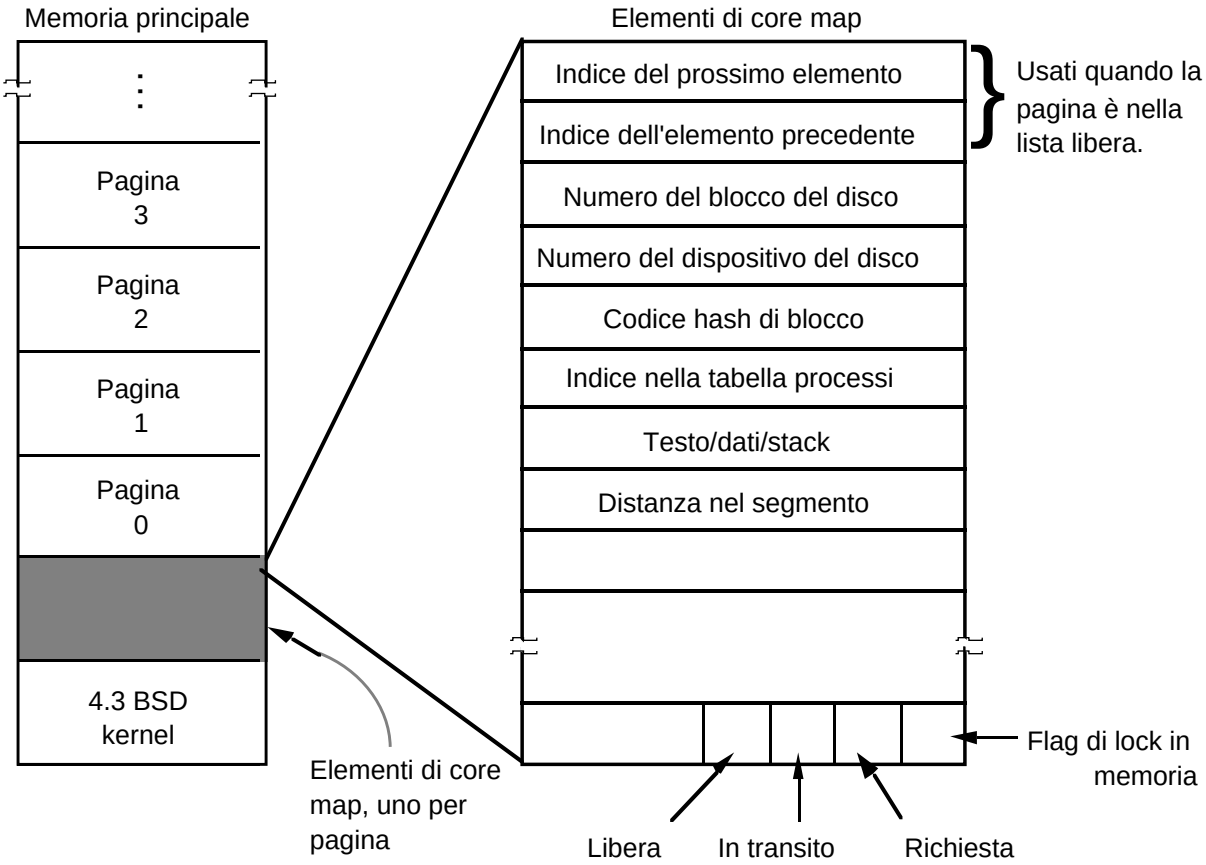
Le versioni recenti di Unix utilizzano tecniche di **paginazione**:

- Per eseguire un processo è sufficiente che in memoria siano presenti almeno:
  - la sua **u-area**
  - la sua **tabella delle pagine**

- 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)

- Se u-area e tabella delle pagine non sono in memoria, ve le porta il processo **swapper**

# 4BSD: ORGANIZZAZIONE DELLA MEMORIA:




Le pagine libere sono collegate in una free list bidirezionale

# PAGEDAEMON

È un processo 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 dell'orologio
- se è ok, ritorna inattivo.

# SWAPPER

È un processo di sistema che:

- se le pagine libere sono sotto il minimo rimuove uno o più processi dalla memoria

  (in genere, quelli inattivi da più tempo)

- verifica periodicamente la 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)

# 3. FILE SYSTEM

# INTRODUZIONE

Il file system Unix, negli anni, si è evoluto da sistema di files **locali** a sistemi di files **distribuiti** (ad esempio `nfs`)

L'interfaccia (system calls) verso il file system è rimasta invece abbastanza stabile

Oggi normalmente il kernel è in grado di supportare file system **multipli**, (coesistenti sulla stessa macchina)

Nel seguito si descrive `s5fs` (il file system originale di System V)

# Strutture dati su disco

# STRUTTURA DI UN DISCO

Ogni disco è suddiviso in zone contigue dette **dischi logici** (o **volumi**, o **partizioni**)

Ogni **file system** è interamente contenuto in un volume, ed ogni volume può contenere un solo file system

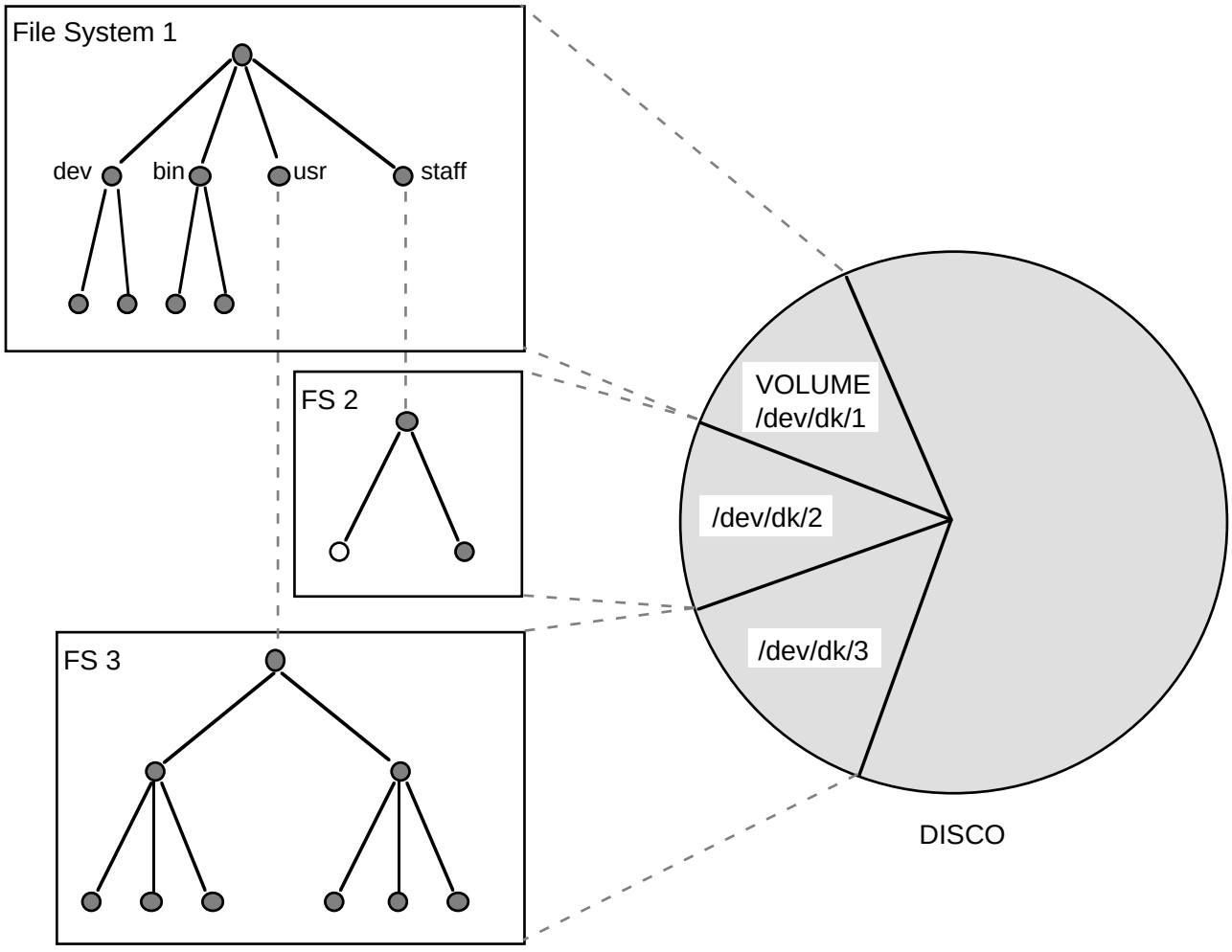
Per creare un file system si usa il comando `mkfs`

La dimensione dei volumi è fissa, e specificata al momento della creazione del file system

# MONTAGGIO E SMONTAGGIO

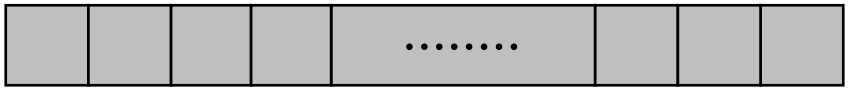
Un file system può essere montato su altri file system mediante il comando `mount` (e smontato mediante il comando `umount`)

## Esempio:



# BLOCCHI

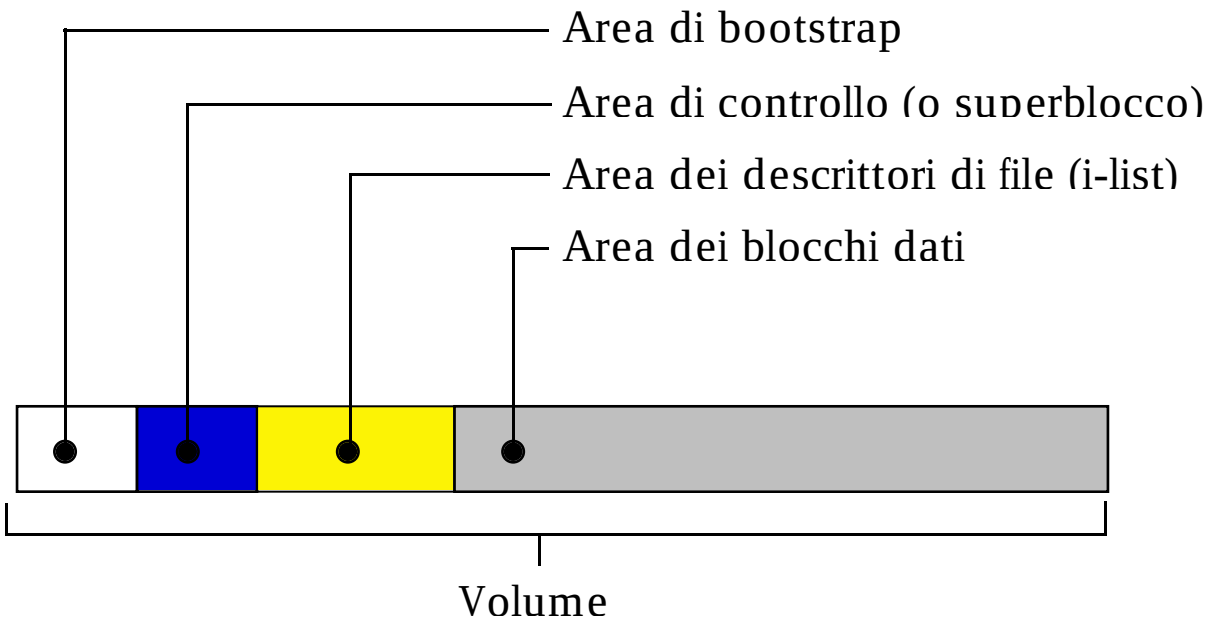
Ogni volume è suddiviso in **blocchi di dimensione fissa** (512, 1024, 2048 o più bytes a seconda della versione di sistema)




Un blocco costituisce la unità di allocazione a un file e la unità di accesso al disco

# STRUTTURA DI UN VOLUME

Un volume è suddiviso in quattro aree distinte:




Le aree hanno dimensione fissa definita al momento della configurazione del sistema

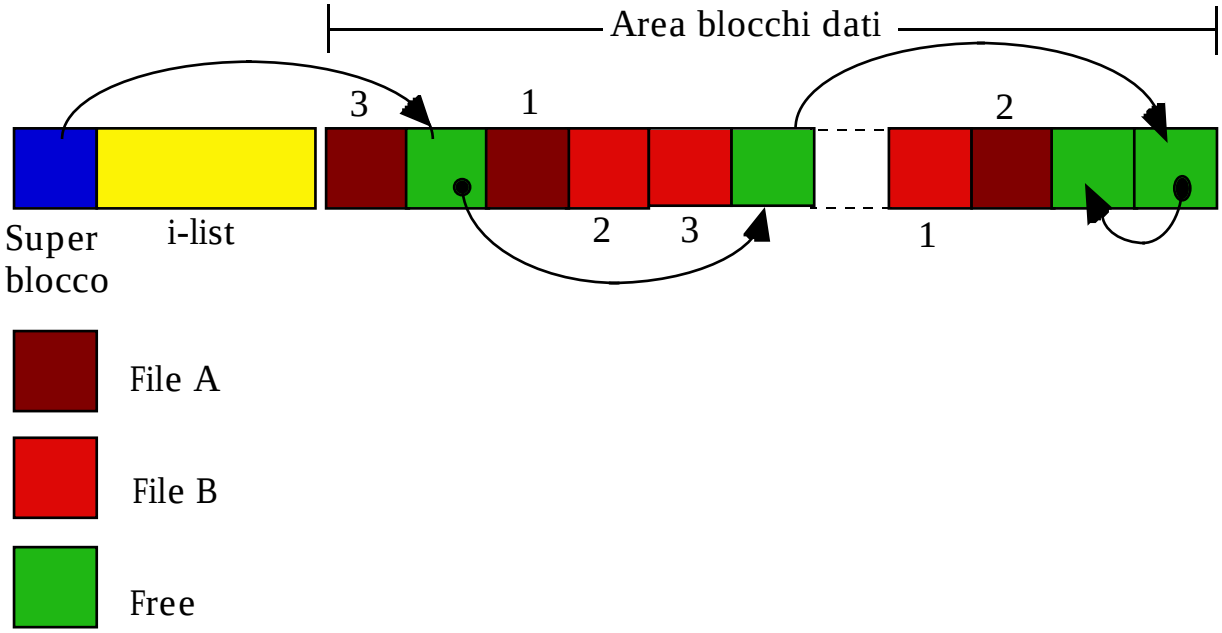
# AREA DI BOOTSTRAP

Non serve per gestire i files: contiene il **programma di bootstrap** per la inizializzazione del sistema

Si usa l'area di bootstrap del volume che contiene il root file system (ma per uniformità c’è su ogni volume)

# AREA DEI BLOCCHI DATI

- Contiene i **blocchi dati** di tutti i file del file system, e i **blocchi liberi**, che possono essere allocati ai files
- I blocchi vengono allocati ai vari file e deallocati quando necessario: pertanto i blocchi di un file non sono necessariamente contigui
- Tutti i blocchi liberi sono collegati in una **free-block list**, la cui testa è nel superblocco

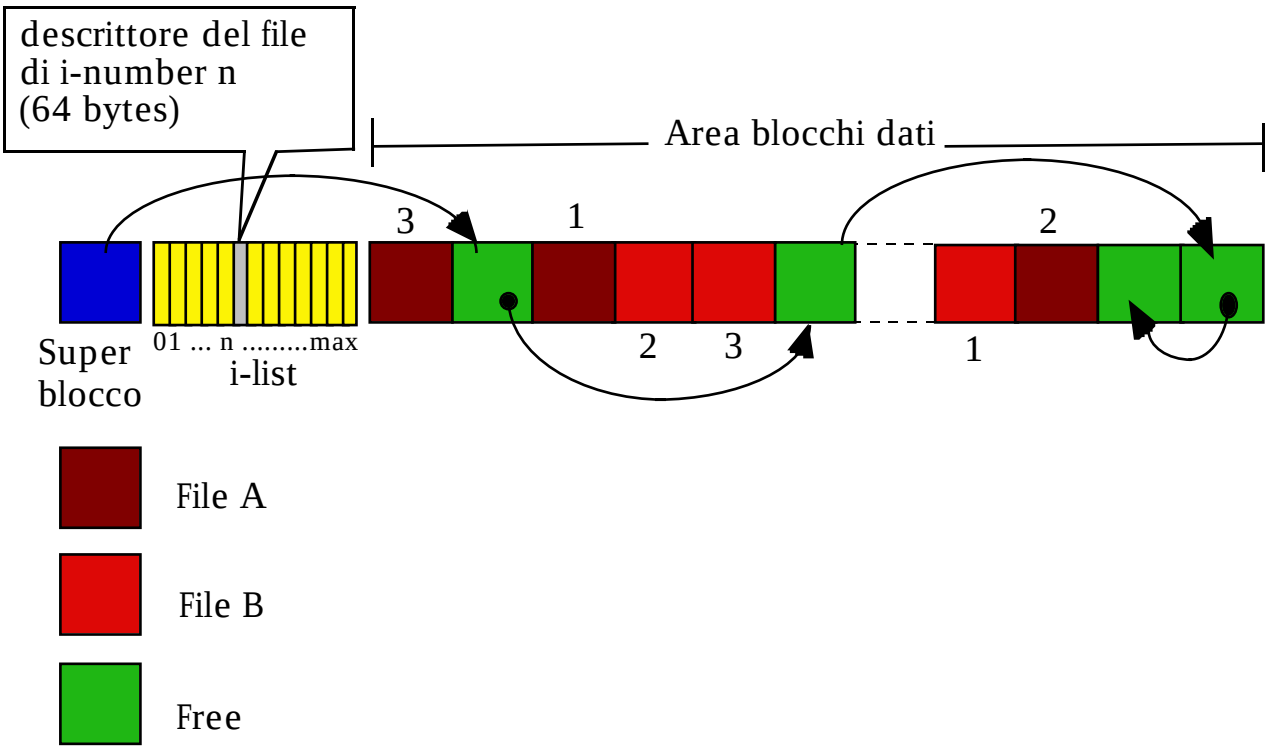



- Gli indirizzi dei blocchi dati di un file sono nel descrittore del file (contenuto nella i-list)

# AREA DEI DESCRITTORI DI FILE (I-LIST)

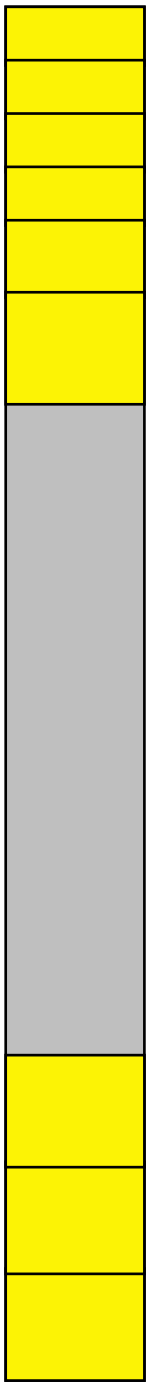
- È una tabella di descrittori di file denominati **i-node** (“index-node”)

- Ciascun i-node è accessibile attraverso il suo indice nella tabella (**i-number**):



# I-NODE

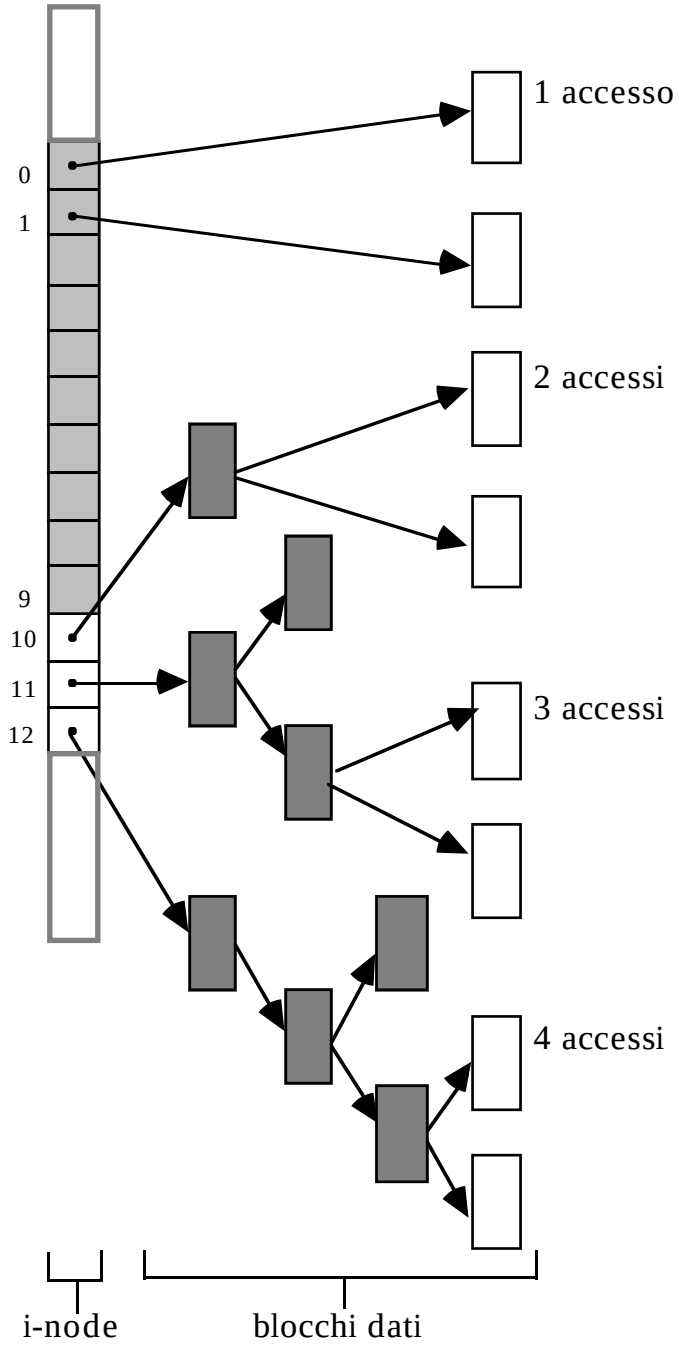
Un i-node contiene gli **attributi di un file**, e cioè:



- Tipo di file
- Numero di (hard) link al file
- UID del proprietario
- GID del proprietario
- Permessi di accesso
- Dimensione del file in bytes
- Tabella degli indirizzi dei blocchi dati
- Ora e data dell'ultimo accesso
- Ora e data dell'ultima modifica
- Ora e data dell'ultima modifica dell'i-node

# TABELLA DEGLI INDIRIZZI AI BLOCCHI DATI

È composta di 13 indirizzi, e si basa su una struttura a quattro livelli:



# ESEMPIO

**Ipotesi:** blocco: 1 Kb

indirizzo: 4 bytes

1024/4 = 256 indirizzi per blocco

## Come accedere al byte 12500 di un file?

12500/1024 = 12 con resto 212

quindi devo accedee al 212-esimo byte del 13-esimo blocco del file

# CAPACITÀ DI INDIRIZZAMENTO

**Ipotesi:** blocco: 1 Kb

indirizzo: 4 bytes

1024/4 = 256 indirizzi per blocco

**Dimensione massima indirizzabile:**

| | | |
|---|---|---|
| 10 * 1 Kb | = 10 Kb | + |
| 256 * 1 Kb | = 256 Kb | + |
| 256 * 256 * 1 Kb | = 64 Mb | + |
| 256 * 256 * 256 * 1 Kb | = 16 Gb | = |

**> 16 GB per file**

NB: In ogni caso, poichè 32 bit indirizzano solo fino a 4 Gb ($2^{32}$), la dimensione è limitata a 4 Gb per file

# PRESTAZIONI

La struttura di indirizzamento a un file privilegia i file piccoli rispetto ai file grandi

**Nota:**

In una tipica installazione Unix, la maggior parte dei file ha dimensioni molto piccole.

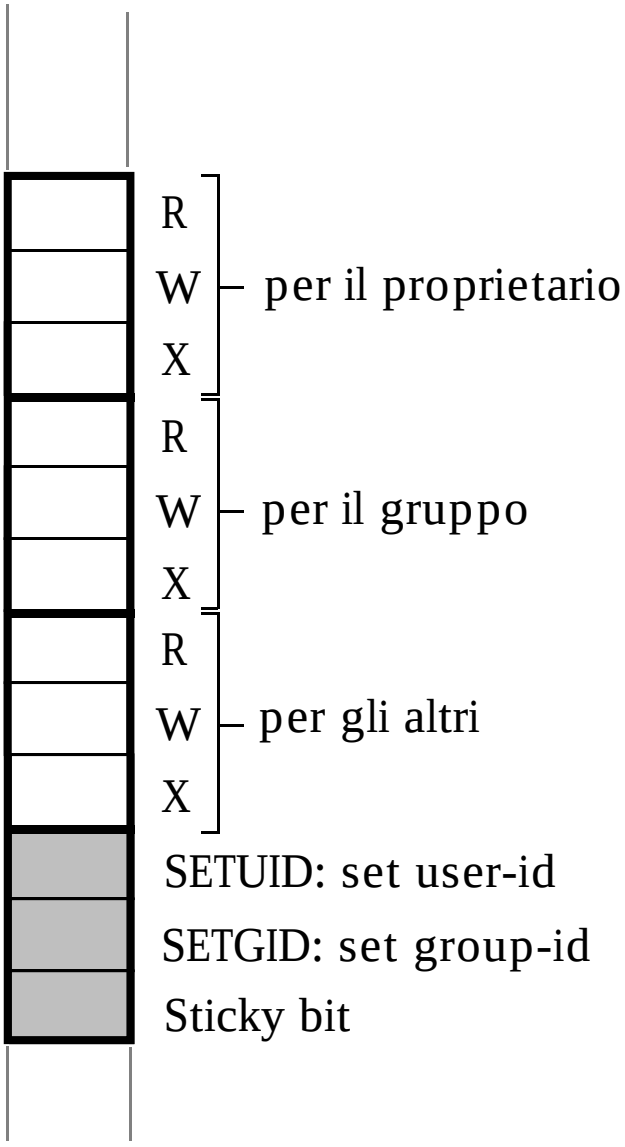
Ad esempio, uno studio[4] su un campione di circa 20.000 files ha dato questi risultati:

| | | |
|---|---|---|
| 48% | dei files: | <= 1 Kb |
| 85% | dei files | <= 8 Kb |

[4] Mullender, Tanenbaum, Immediate Files, in Software Practice and Experience, Aprile 1984

# PERMESSI

12 bit:




**Sticky bit**: permette di richiedere al kernel che l'immagine di un processo resti allocata nell'area di swap anche dopo la sua terminazione (utile per programmi frequentemente utilizzati, es. `vi`)

# AREA DI CONTROLLO (O SUPERBLOCCO)

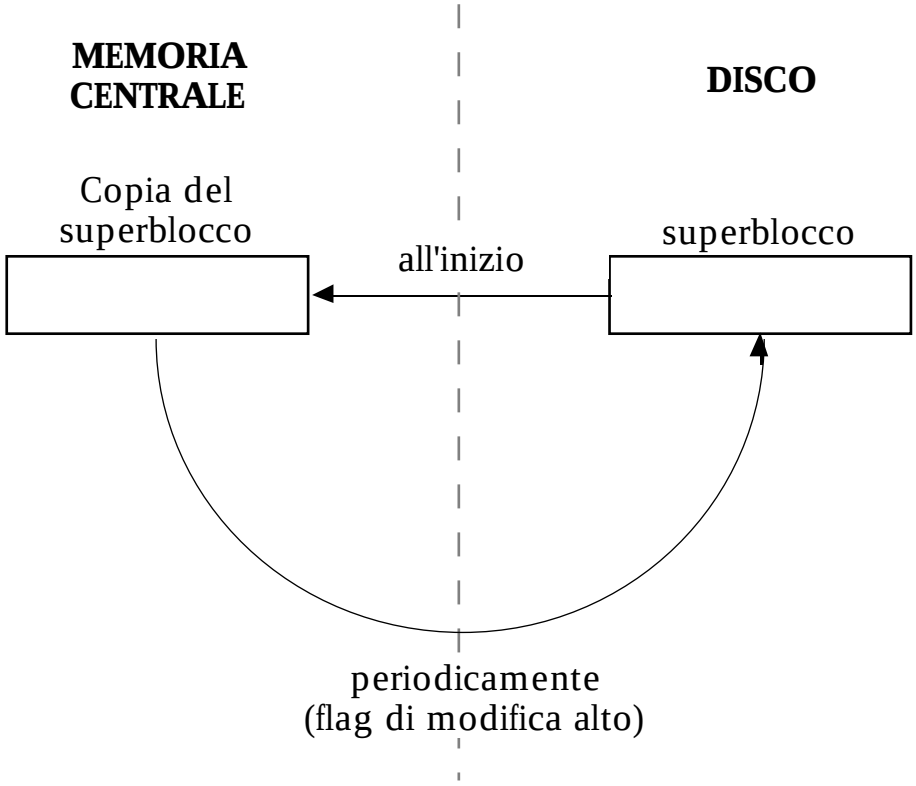
È un singolo blocco, e contiene informazioni globali sul volume:

- **dimensione del volume** (# blocchi)
- **informazioni per gestione blocchi liberi** (numero dei blocchi liberi nel volume, testa della free-block list, flag di lock della free-block list
- **dimensione della i-list**
- **informazioni per gestione i-nodes liberi** (numero degli i-node liberi, cache degli i-node liberi, flag di lock per la lista degli i-node liberi)
- **flag che indica se il superblocco è stato modificato**

# AGGIORNAMENTO DEL SUPERBLOCCO

Per motivi di efficienza, copia del superblocco sta normalmente in memoria...

... e il sistema lo registra periodicamente su disco, se è stato modificato



# SUPERBLOCCO: CRITICITÀ

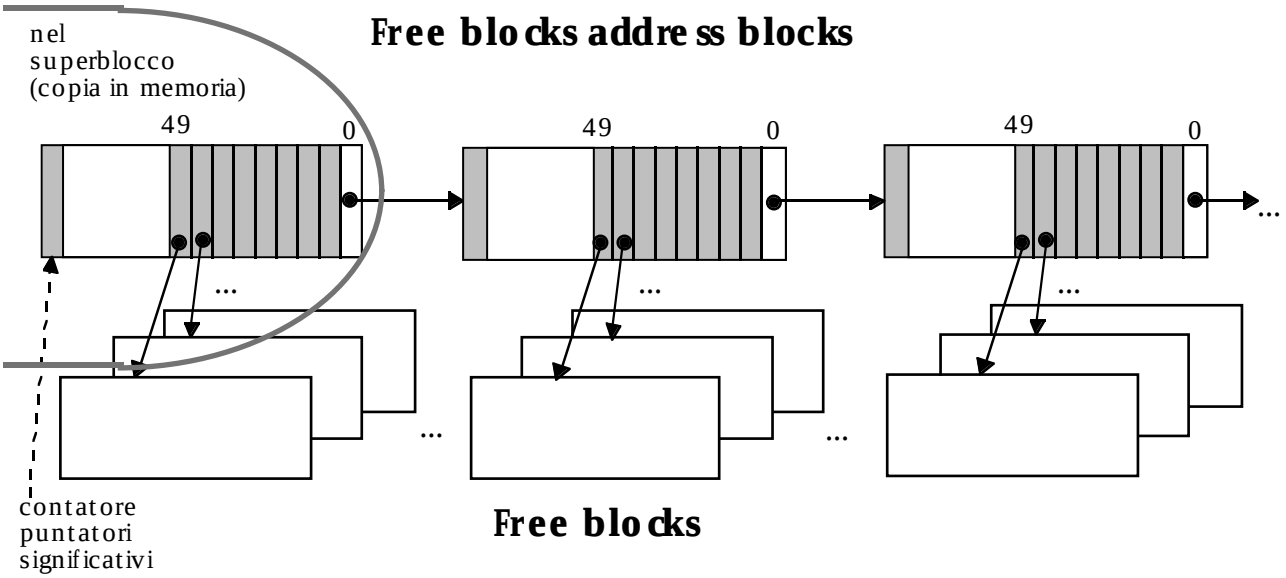
Il superblocco è un'area critica, senza la quale l'intero file system non è più accessibile

Che cosa può capitare:

1. Il blocco fisico di disco su cui risiede può rovinarsi
2. Il sistema può cadere prima dell'allineamento fra superblocco in memoria e su disco

**Architettura molto pericolosa!**

# FREE-BLOCK LIST




## Vantaggi:

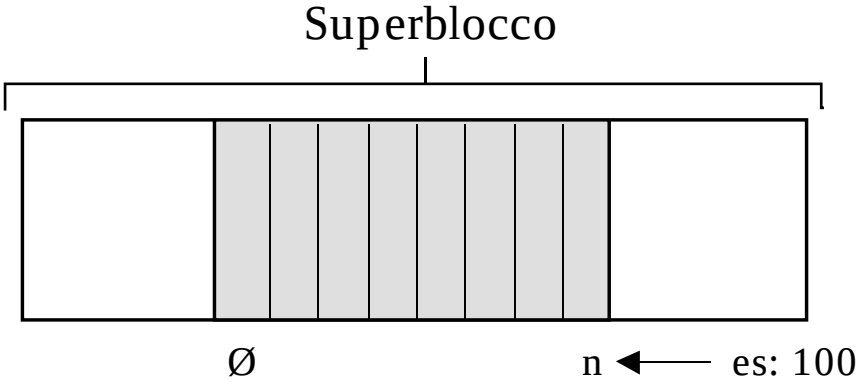
Con 1 accesso al disco (al massimo) posso:

- ottenere gli indirizzi di **molti** blocchi liberi
- liberare **molti** blocchi

(Una normale lista lineare permetterebbe di ottenere un solo blocco alla volta)

# CACHING DEGLI I-NODES LIBERI

Nel superblocco viene mantenuto un elenco di i-number di i-node liberi (“caching”)




- 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 i-node liberi (campo “tipo-file” a 0)

  (Ci si ricorda l’ultimo i-node letto nella i-list, e la prossima ricerca riparte da lì)
- In tal modo, gli i-node liberati vengono riutilizzati

(Questo approccio non è utilizzabile per i blocchi liberi, perché questi non sono riconoscibili)

# s5fs: VANTAGGI E SVANTAGGI

**Vantaggi:**

- semplicità

**Svantaggi:**

- **superblocco**: se si rovina, l'intero file system diventa inutilizzabile
- **allocazione fisica**: non è ottimizzata (-> lunghi tempi di seek), ad es.:
  - i-node tutti a inizio volume in ordine casuale, dati tutti in fondo
  - frammentazione dei files crescente con l'uso
- **block size fisso:** se cresce, migliorano le prestazioni ma peggiora l'utilizzo dello spazio disco (in media, ogni file spreca metà dell'ultimo blocco)
- **limiti:** sui filename (14 crt) e sul numero massimo di files (65535)

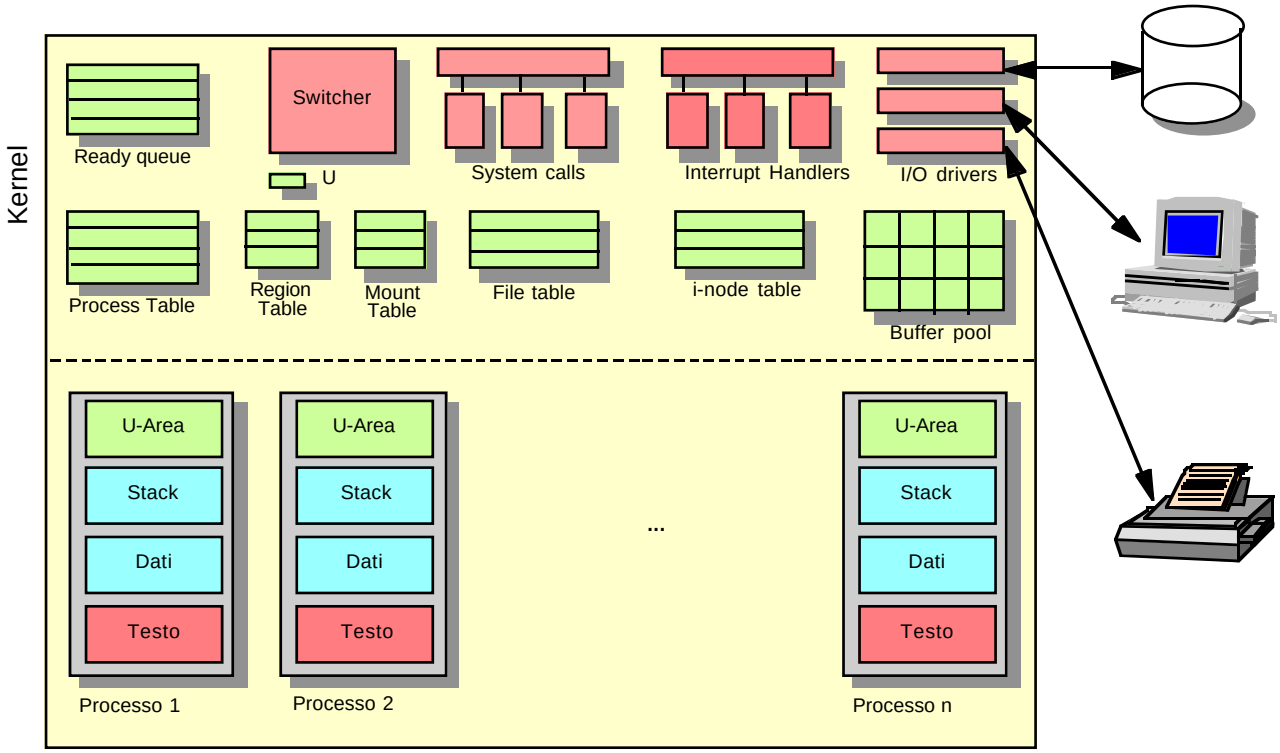
# IL FAST FILE SYSTEM (FFS) DI BERKELEY

Principali miglioramenti rispetto a `s5fs`:

- filename fino a 256 caratteri
- ogni partizione è suddivisa in **gruppi di cilindri** contigui, ciascuno con proprie informazioni di servizio: ciò permette di adottare particolari strategie per tenere vicini dati correlati
- il superblocco è spezzato in più parti, che sono duplicate per sicurezza
- file system diversi possono avere block size diversi ($2^n$, non inferiore a 4096 bytes)
  (es. due livelli di indirettezza bastano per 4 Gb)
- un blocco é suddiviso in n **frammenti** di almeno 512 bytes (n fisso per ogni file system)
- l'ultimo blocco di un file può essere parziale (1 o più frammenti)

# Strutture dati in memoria

# TABELLE PER LA GESTIONE DEI FILES




Nella U-area di ogni processo:

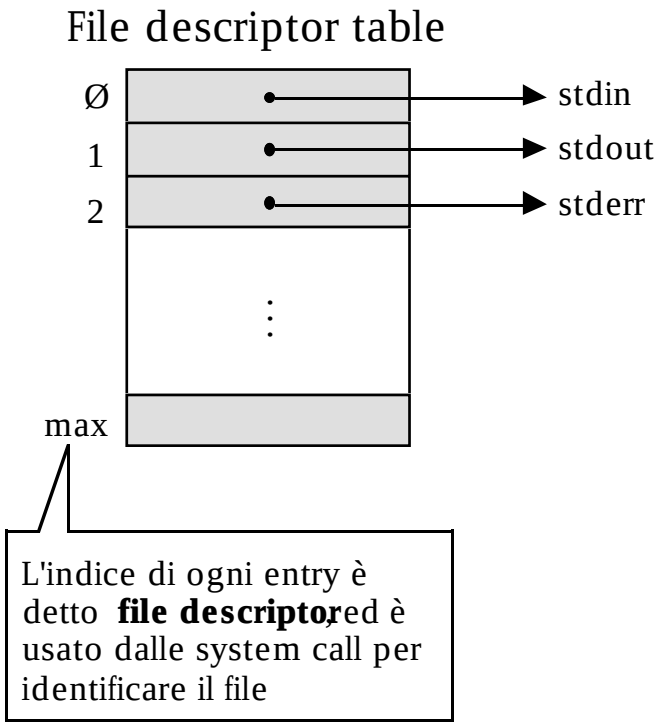
- **File Descriptor Table**

Globali a tutti i processi e residenti:

- **File Table**
- **I-node table**
- **Mount Table**

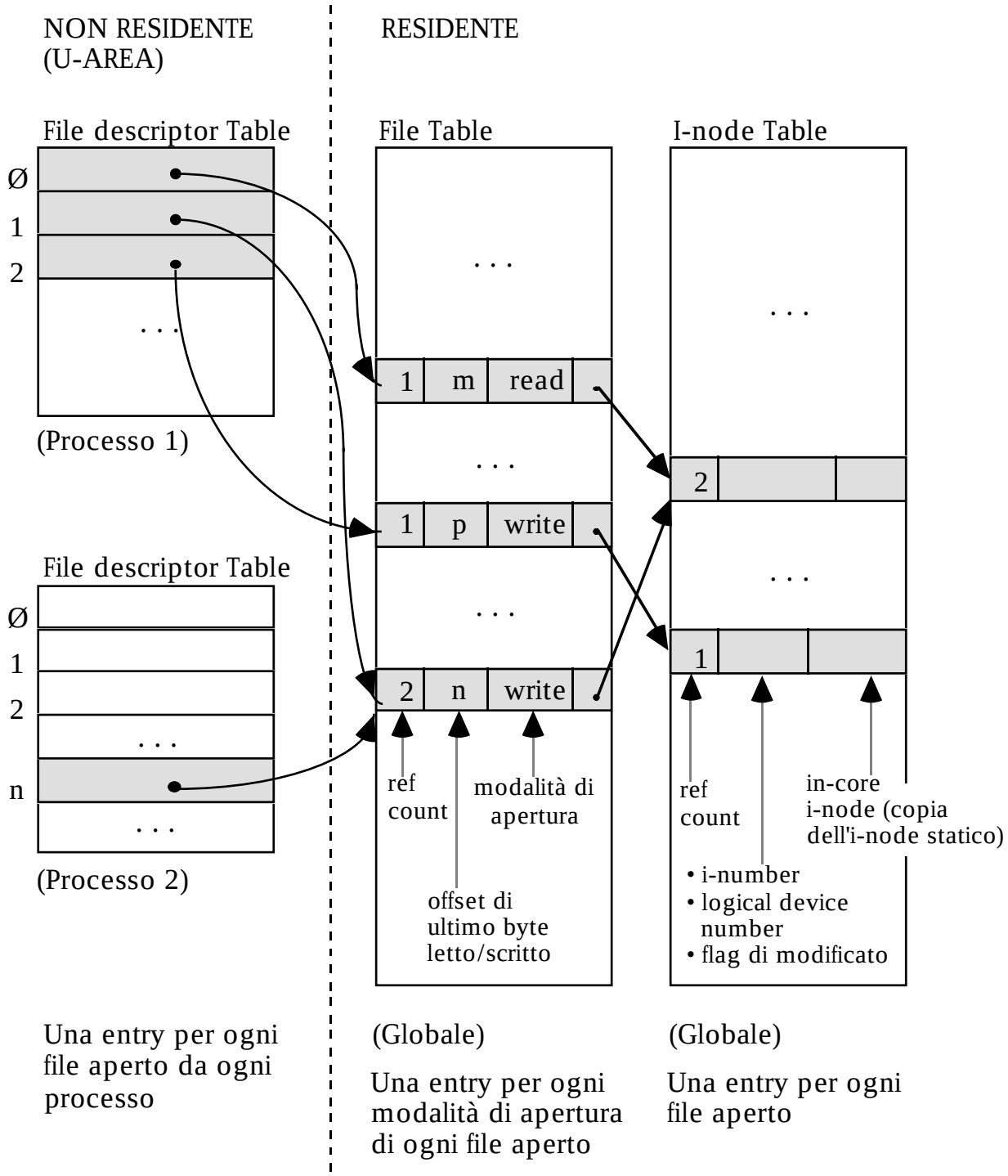
# FILE DESCRIPTOR

Nella U-AREA di ogni processo esiste la tabella dei file aperti dal processo:



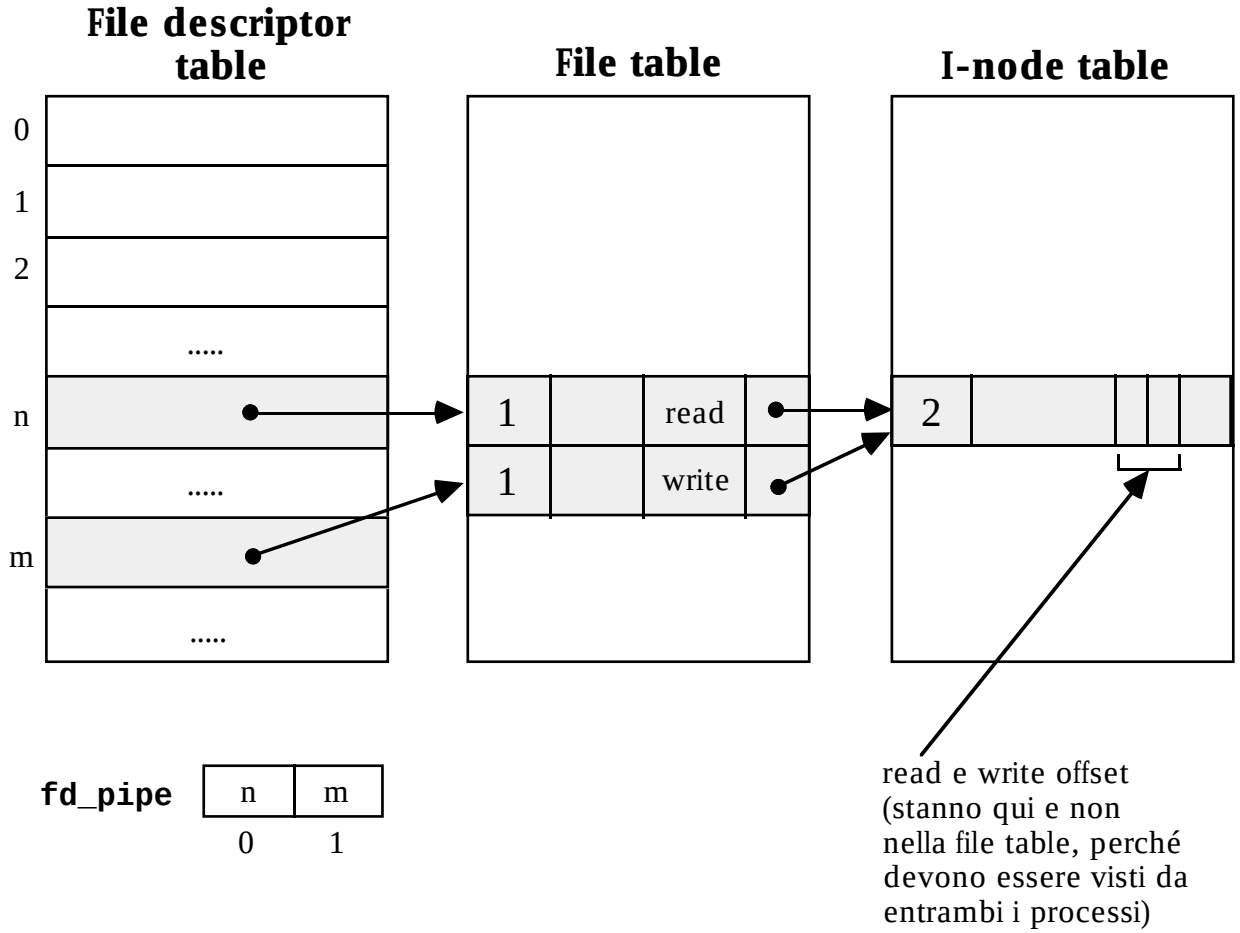

Questo sta alla base del meccanismo di redirezione dell'input/output

# ALTRE STRUTTURE DI MEMORIA



# PIPES

```
pipe(fd_pipe);
```



# MOUNT TABLE

Tabella sempre residente che contiene, per ogni file system montato:

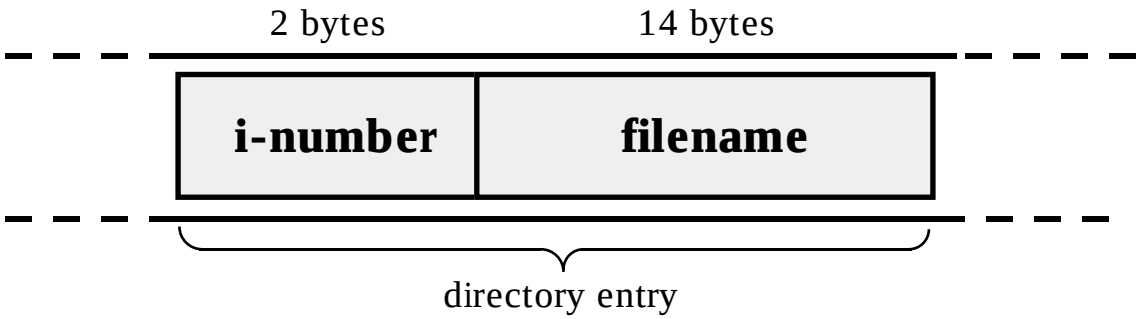
- major e minor number del device montato
- puntatore all'area superblocco del file system montato
- puntatore alla root del file system a cui è montato
- puntatore alla directory su cui è montato

La Mount table è aggiornata da `mount` e `umount`

# Visione dinamica

# STRUTTURA DI UNA DIRECTORY

## In System V:



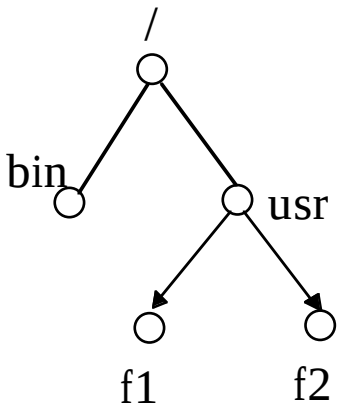

Quindi il numero massimo di file è 65535 (0 non è un i-number valido)

Più in generale (Posix):

```
struct dirent {
    ino_t  d_ino;              /* i-number */
    char d_name[NAME_MAX+1]    /*null-
                                terminated
                                filename */
};
```

# ACCESSO A UN FILE: ESEMPIO

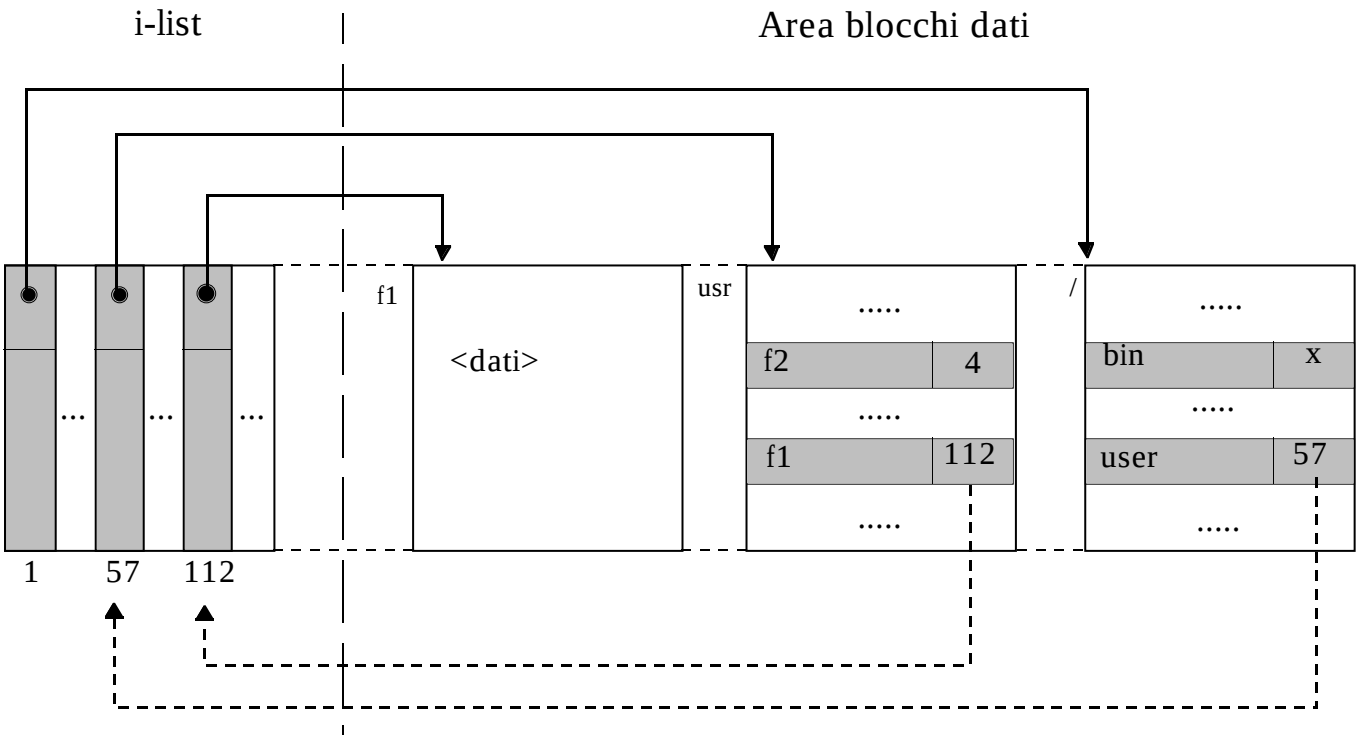
## Visione logica:




## Pathname del file da accedere: /usr/f1

## Visione fisica:

NB: Si suppone per semplicità che tutti i file e la i-list occupino un solo blocco



# CHE COSA FA LA open

`fd=open(pathname,mode)`

Tramite il pathname del file cerca il suo i-node

Se il file non esiste o non è accessibile nella modalità richiesta 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`)

# CHE COSA FA LA `close`

```
s=close(fd)
```

Dealloca la File descriptor table entry

Se il reference count nella File table è >1 lo decrementa e conclude

Altrimenti:

- dealloca la File table entry
- Se il reference count nell'in-core i-node >1
- lo decrementa e conclude
- Altrimenti: dealloca l'in-core i-node

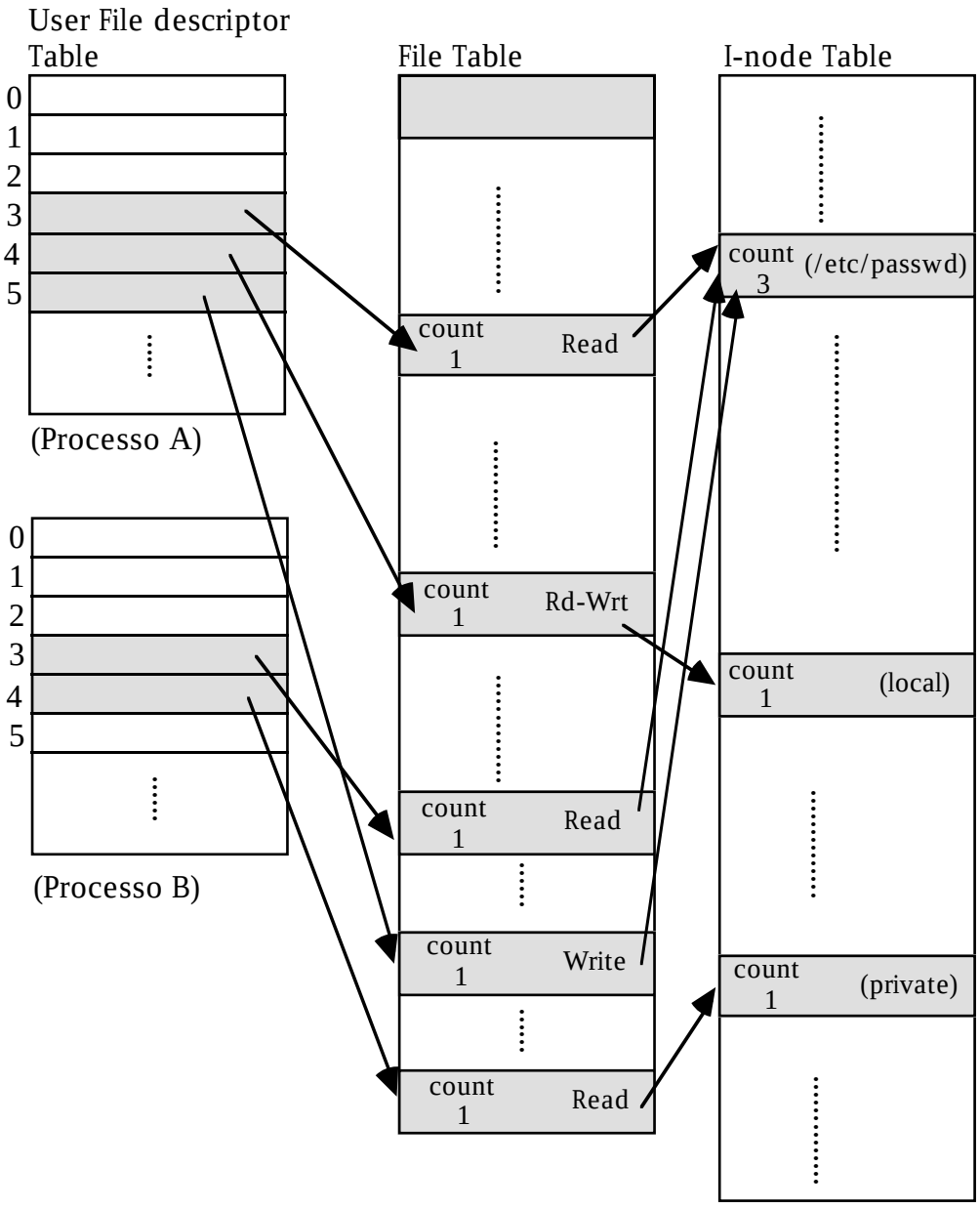
# ESEMPIO

**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);
```

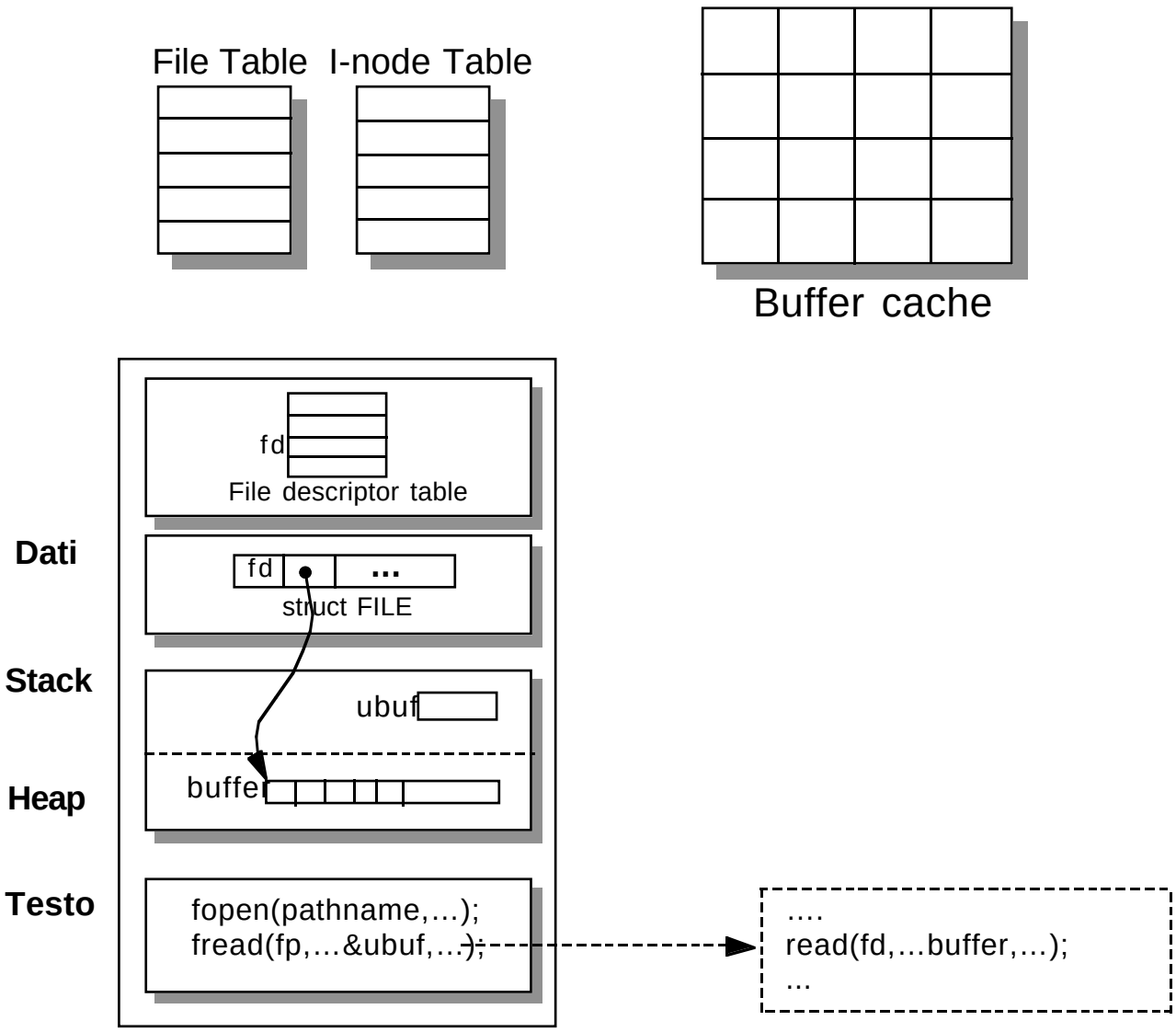


# CHE COSA FA LA `read`

```
n=read(fd, buffer, nbytes)
```

- Accede alla File table entry a partire da `fd` e controlla la modalità di apertura
- Accede all'in-core i-node; lock l'in-core i-node
- Trasforma l'offset nella File table entry in un indirizzo di disco (indirizzo blocco, + offset all'interno del blocco), attraverso la tabella di indirizzamento dell'in-core i-node
- Legge il blocco/blocchi dati richiesto/i (attraverso la buffer cache) e copia gli `nbytes` richiesti in `*buffer`
- Aggiorna l'i-node; unlock i-node
- Aggiorna l'offset in File table entry
- Restituisce al chiamante il numero di bytes letti

# BUFFERING: VISIONE DI INSIEME




**NB**: • `buffer` è il buffer (non visibile all'utente) gestito dalle funzioni della C Standard Library (`fopen`, `fread`) e allocato automaticamente dalla `fopen` nello heap;

• `ubuf` è il buffer definito dall'utente (dove vuole lui)

# BUFFERING: ANOMALIE

Varie situazioni possibili:

- Se il processo esegue una `exit`, questa fa una flush di `buffer`[5] nella buffer cache
- Se il processo viene interrotto (es.: illegal instruction), il controllo passa al kernel, il quale chiude i file (via File Descriptor Table nella U-area), ma non conosce `buffer`, il cui contenuto quindi viene perso
- Se cade il sistema, si perde il contenuto della buffer cache

[5] Si intende il buffer allocato dalle primitive della C standard library (vedi figura precedente)

# Sottosistema di I/0

# IL SISTEMA DI I/O