# Interazione e sincronizzazione

# Interazione tra processi

***Processi (e thread) possono interagire.***

**Classificazione:**

- **processi indipendenti**: due processi P1 e P2 sono indipendenti se l'esecuzione di P1 non è influenzata da P2, e viceversa.
- **processi interagenti**: P1 e P2 sono interagenti se l'esecuzione di P1 è influenzata dall'esecuzione di P2, e/o viceversa.

# Interazione tra processi

## L'interazione tra processi* e` caratterizzata da diversi parametri:

- **Motivazioni: perche` i processi interagiscono.**
- **Modalita`: come avviene l'interazione**
- **Modello: quale modello per i processi**
- **Strumenti: attraverso quali strumenti si concretizza l'interazione**

* il termine processo viene utilizzato in questo contesto per indicare sia processi "pesanti" che thread

# Motivazioni

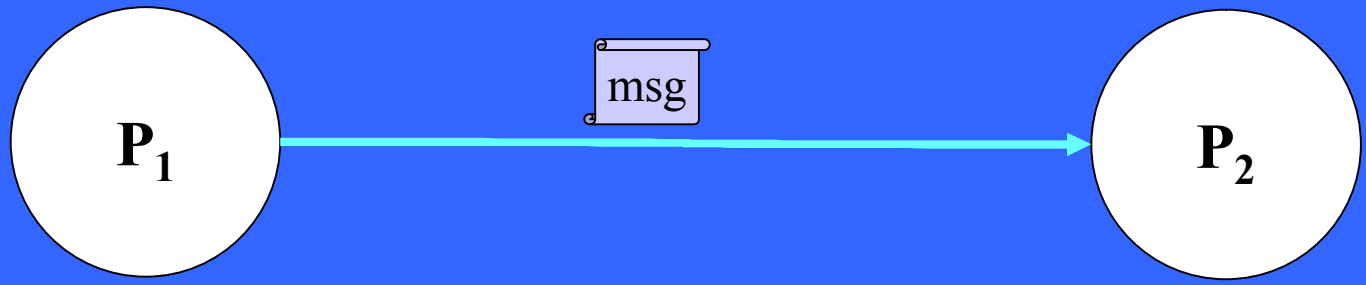
## Due processi possono interagire per diverse ragioni:

- **Competizione:** i processi interagiscono per *coordinarsi* nell'accesso a risorse comuni. Di solito, mediante strumenti di *sincronizzazione*.

- **Cooperazione:** i processi interagenti concorrono alla soluzione dello stesso problema (ad esempio thread dello stesso task, applicazioni client/server,ecc.). Tipicamente mediante:
  - *comunicazione*
  - *sincronizzazione*

# Modalita`: Comunicazione e Sincronizzazione

- **Comunicazione:** scambio di informazioni tra i processi interagenti. Ad esempio P1 trasferisce a P2 l'informazione msg.




- **Sincronizzazione:** imposizione di vincoli temporali sull'esecuzione dei processi.

  Ad esempio, l'istruzione K del processo P1 puo` essere eseguita soltanto dopo l'istruzione J del processo P2

# Modello

**A seconda del modello di processo adottato nel S.O. l'interazione puo` essere supportata da astrazioni diverse:**

- **processo "pesante"(*modello ad ambiente locale*): non vi sono dati condivisi; il supporto all'interazione e` completamente a carico del Sistema Operativo. L'interazione avviene mediante chiamate al sistema operativo.**

- **thread (*modello ad ambiente globale*): la condivisione di memoria rende possibile l'utilizzo di variabili condivise a supporto dell'interazione:**
  - **» PRO: costo minore**
  - **» CONTRO: il programmatore deve risolvere i problemi di sincronizzazione nell'accesso alle variabili condivise; uso di strumenti di sincronizzazione (mutex, semafori, variabili condizione, monitor, ecc.)**

# Strumenti

**L'interazione si puo` realizzare attraverso diversi *strumenti*:**

- **mezzi di comunicazione (esempio: socket, pipe, fifo..)**
- **scambio di eventi (esempio, segnali)**
- **semafori, mutex, variabili condizione**
- **buffer condivisi**
- **file**
- **...**

# Strumenti di interazione

**In base al** modello di processo:

- **modello ad ambiente locale** (processi pesanti): non c'è condivisione di variabili
  - la **comunicazione** avviene attraverso risorse esterne (canali, porte, file..)
  - la **sincronizzazione** avviene attraverso meccanismi esterni che consentono lo scambio di eventi (*segnali*)
- **modello ad ambiente globale**: condivisione dello spazio di indirizzamento
  - la **comunicazione** si realizza mediante l'accesso concorrente a variabili condivise, controllato mediante strumenti di sincronizzazione (mutex, semafori, ecc.)
  - la **sincronizzazione** avviene attraverso strumenti di sincronizzazione

# Ruolo del Sistema Operativo

- **modello ad ambiente locale**:
  - il sistema operativo fornisce i meccanismi per la **comunicazione** (canali, porte, file..) e la **sincronizzazione (***segnali***)** che i processi interagenti possono utilizzare mediante system call.
- **modello ad ambiente globale**:
  - fornisce i meccanismi di **sincronizzazione**

# Esempio: cooperazione nel modello ad ambiente locale

***Hp:***

- **modello ad ambiente locale (processi pesanti)**
- **cooperazione mediante comunicazione**

- **Il Sistema Operativo offre meccanismi (*risorse* e *operazioni*) a supporto della comunicazione tra processi (*Inter Process Communication*, o IPC).**
  - **Risorse: canale di comunicazione**
  - **Operazioni *(system call):*** *send, receive*

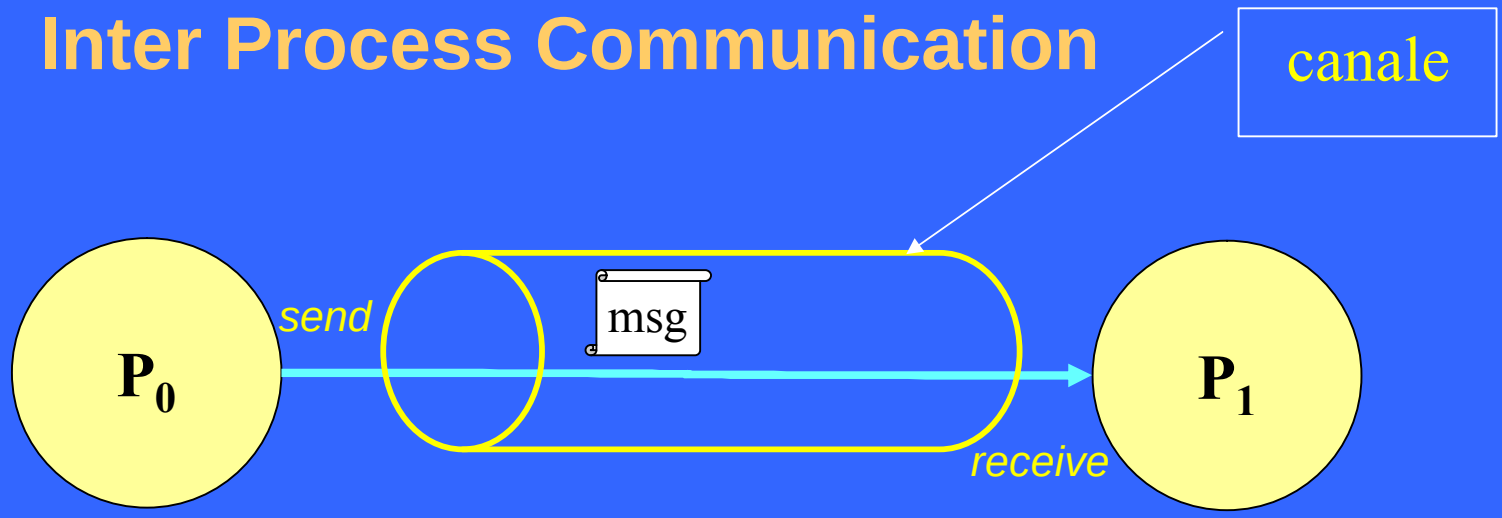
# Esempio: cooperazione nel modello ad ambiente globale

***Hp:***

- **modello ad ambiente locale**
- **thread comunicanti**

- **La comunicazione tra i processi sfrutta variabili condivise, accessibili direttamente dai processi interagenti. L'accesso deve essere opportunamente coordinato facendo uso di meccanismi di sincronizzazione (es: semafori). Le politiche di sincronizzazione sono a carico del programmatore.**

# Inter Process Communication




Lo scambio di messaggi tra due processi avviene mediante un ***canale di comunicazione.***

**Operazioni**: *send (..msg..)* *receive(..msg..)*

**Caratteristiche del canale:**

- monodirezionale, bidirezionale
- uno-a-uno, uno-a-molti, molti-a-uno, molti-a-molti
- capacità
- creazione: automatica, non automatica
- ..

# Meccanismi di comunicazione tra processi

**Aspetti caratterizzanti**:

- caratteristiche del messaggio:
  - » dimensione
  - » tipo
- caratteristiche della comunicazione:
  - » diretta o indiretta
  - » simmetrica o asimmetrica
  - » bufferizzata o no
  - » ...

# Naming

## In che modo viene specificata la destinazione di un messaggio?

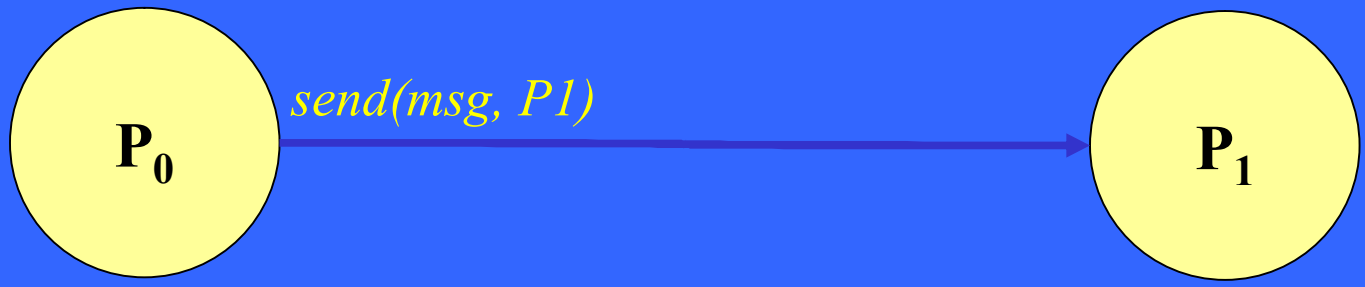
- **Comunicazione diretta**: al messaggio viene associato l'identificatore del processo destinatario (naming esplicito)

  ***send(Proc, msg)***

- **Comunicazione indiretta**: il messaggio viene indirizzato a una mailbox (contenitore di messaggi) dalla quale il destinatario preleverà il messaggio:

  ***send(Mailbox, msg)***

# Comunicazione diretta




- Il canale è creato automaticamente tra i due processi che devono *conoscersi* reciprocamente:
  - canale punto-a-punto
  - canale bidirezionale:

    p0: send(query, P1);

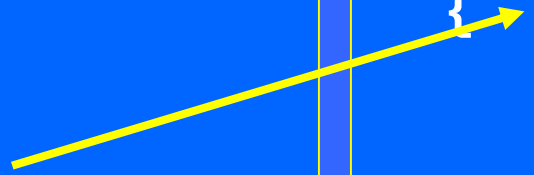
    p1: send(answ, P0)
  - per ogni coppia di processi esiste un solo canale(<P0, P1>)

# Esempio

**Processo mittente P1**:

```
pid P2 =....;
main()
{ msg M;
   do
   { produco(&M);
      ...
      send(P2, M);
   }while(!fine);
}
```

```
Processo destinatario P2:
pid P1=...;
main()
{ msg M;
   do
   { receive(P1, &M);
      ...
      consumo(M);
   }while(!fine);
}
```



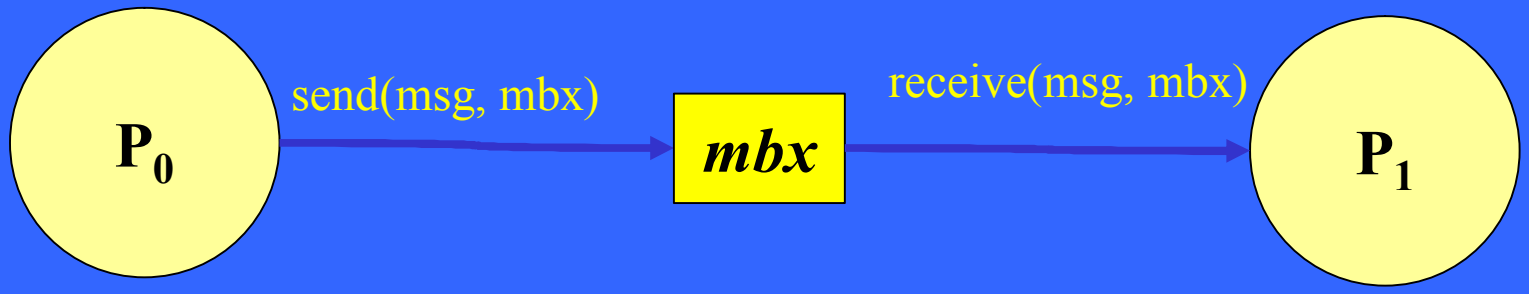
**Comunicazione diretta:** mittente e destinatario usano ***naming*** esplicito.

# Comunicazione diretta

**Problema:**

- **scarsa modularità**:
  - la modifica del nome di un processo implica la revisione di tutte le operazioni di comunicazione
  - difficoltà di riutilizzo
  - utilità di servizi di directory (name server)

# Comunicazione indiretta: *mailbox*




- La **mailbox** (o ***porta***) è una risorsa astratta accessibile da più processi che rappresenta un contenitore di messaggi.
  - » processi cooperanti non sono tenuti a conoscersi reciprocamente: si scambiano messaggi depositandoli/prelevandoli da una **mailbox** *condivisa.*

# Mailbox

## Proprietà:

- il canale di comunicazione è rappresentato dalla mailbox (non viene creato automaticamente)
- il canale può essere associato a più di 2 processi:
  - » mailbox di sistema: molti-a-molti
  - » mailbox del processo destinatario (***porta***): molti-a-uno
- canale bidirezionale:

  client: send(req, mbx)

  server: send(answ, mbx)
- per ogni coppia di processi possono esistere più canali (uno per ogni mailbox condivisa)

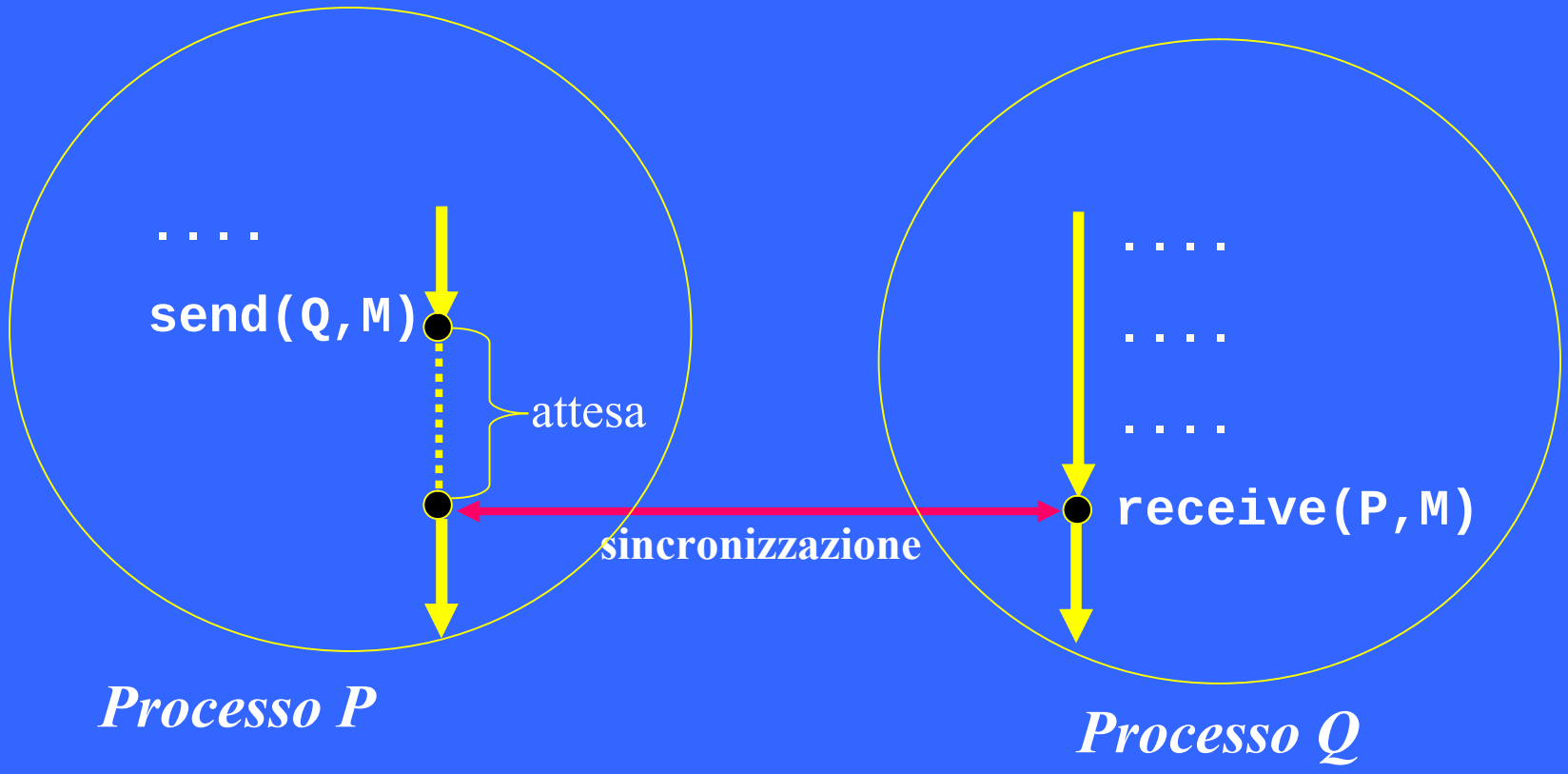
# Buffering del canale

Ogni canale di comunicazione è caratterizzato da una **capacità** che rappresenta il numero dei messaggi che è in grado di gestire contemporaneamente:

**capacita` nulla:**

- il canale non è in grado di gestire messaggi in attesa
- processo mittente e destinatario devono sincronizzarsi all'atto di spedire (send) / ricevere (receive) il messaggio: comunicazione sincrona o *rendez vous*
- send e receive possono essere sospensive

# Canale a capacita` nulla



# Buffering del canale

- **Capacità non nulla**: il canale puo` contenere fino a N messaggi contemporaneamente, di solito gestiti in modo FIFO:
  - se il canale **non è pieno**, un nuovo messaggio viene posto in fondo
  - se il canale **è pieno**: la send è sospensiva
  - se il canale **è vuoto**: la receive può essere sospensiva

# Caratteristiche della comunicazione tra processi unix

Vari meccanismi di comunicazione:

- **pipe**: mailbox per comunicazione locale (nell'ambito della stessa gerarchia di processi)
- **fifo:** comunicazione locale tra qualunque insieme di processi (System V)
- **socket**: comunicazione in ambiente locale e distribuito

**pipe**: caratteristiche

- » **comunicazione indiretta** (senza naming esplicito)
- » canale **unidirezionale** molti-a-molti
- » **bufferizzata** (capacità limitata): possibilità di sospensione sia per mittenti che per destinatari.
- » **omogeneita` con i file**

# Sincronizzazione tra processi

**La sincronizzazione permette di imporre vincoli sulle operazioni dei processi interagenti.**

## Ad Esempio:

**Nella cooperazione**:

- Per imporre un particolare ordine  cronologico alle azioni eseguite dai processi interagenti.
- per garantire che le operazioni di comunicazione avvengano secondo un ordine prefissato.

**Nella competizione**:

- per garantire il rispetto di politiche di accesso (ad esempio, mutua esclusione) dei processi nell'accesso alla risorsa condivisa.

# Sincronizzazione tra processi nel modello ad ambiente locale

**In questo ambiente non vi e` la possibilita` di condividere memoria:**

- **Gli accessi alle risorse "condivise" vengono controllati e coordinati dal sistema operativo.**
- **La sincronizzazione avviene mediante meccanismi offerti dal sistema operativo che consentono la notifica di “eventi” asincroni (privi di contenuto informativo) tra un processo ed altri.**
- **Per esempio, Linux/*nix:**
  - ***segnali***

# Sincronizzazione tra processi nel modello ad ambiente globale

**In un sistema multi-threaded:**

- **cooperazione**: lo scambio di messaggi avviene attraverso strutture dati condivise (che implementano il concetto di *mailbox*)
- **competizione**: le risorse sono rappresentate da variabili condivise (ad esempio, puntatori a file)

- In entrambi i casi è necessario sincronizzare i processi per coordinarli nell'accesso alla memoria condivisa:

**problema della mutua esclusione**

# Esempio: comunicazione in ambiente globale con mailbox di capacita` MAX

```
typedef struct {
        coda mbx;
     int num_msg; } mailbox;
shared mailbox M;
```

- **Processo mittente:**

```
...
   <crea messaggio m>
   if (M.num_msg < MAX)
    /* send: */
   { inserisci(M.mbx,m);
     M.num_msg++;}
   /* fine send*/
...
```

- **Processo destinatario:**

```
...
   if (M.num_msg >0)
   /* receive: */
   { estrai(M.mbx,m);
     M.num_msg--;}
   /* fine receive */
<consuma messaggio m>
...
```

# Esempio: Esecuzione

Hp: al tempo $T_i$ **`M.num_msg=MAX;`**
SO time sharing

**Processo mittente:**

$T_{i+4}$:**`<crea messaggio m>`**
$T_{i+5}$: **`if (M.num_msg < MAX)`** *falso*

**Processo destinatario:**

$T_{i+1}$: **`if (M.num_msg >0)`** *vero*
$T_{i+2}$: **`estrai(M.mbx,m);`** *libero un posto*
$T_{i+3}$: **`<cambio di contesto>`**

l'inserimento
non viene eseguito!
**`M.num_msg=MAX`**

- La correttezza della gestione della mailbox dipende dall'ordine di esecuzione dei processi
- E` necessario imporre la **mutua esclusione** dei processi nell'accesso alla variabile M

# Il problema della mutua esclusione

In caso di condivisione di risorse (*variabili*) può essere necessario impedire accessi concorrenti alla stessa risorsa (*variabile*).

**Sezione critica**:

è la sequenza di istruzioni mediante le quali un processo accede e può aggiornare una o piu` variabili condivise.

**Mutua esclusione:**

ogni processo esegue le proprie sezioni critiche in modo mutuamente esclusivo rispetto agli altri processi

# Esempio: sezioni critiche

sezione critica

```
typedef struct {
        coda mbx;
      int num_msg; } mailbox;
shared mailbox M;
```

sezione critica

- **Processo mittente:**

```
...
   <crea messaggio m>
   if (M.num_msg < MAX)
    /* send: */
   { inserisci(M.mbx,m);
     M.num_msg++;}
   /* fine send*/
...
```

- **Processo destinatario:**

```
...
   if (M.num_msg >0)
   /* receive: */
   { estrai(M.mbx,m);
     M.num_msg--;}
   /* fine receive */
<consuma messaggio m>
...
```

# Mutua esclusione

Necessita` di garantire la mutua esclusione nell'esecuzione delle sezioni critiche (accesso e aggiornamento di M).

Soluzione: esclusione; percio` ogni sezione critica e`:

- preceduta da un **prologo** (***entry section***), mediante il quale il processo ottiene l'autorizzazione all'accesso in modo esclusivo
- seguita da un **epilogo** (***exit section***), mediante il quale il processo rilascia la risorsa

```
<entry section>
<sezione critica>
<exit section>
```

➔ prologo ed epilogo fanno uso di strumenti di sincronizzazione: ad esempio, ***semafori***.

# Semafori (Dijkstra, 1965)

## Definizione di Semaforo:

- **E` un tipo di dato astratto al quale sono applicabili solo due operazioni (*primitive*):**
  - ***wait (s)***
  - ***signal (s)***
- **Al semaforo s sono associate:**
  - **una variabile intera *s.value* non negativa con valore iniziale >= 0.**
  - **una coda di processi *s.queue***

➢ **Il semaforo puo` essere condiviso da 2 o più processi per risolvere problemi di sincronizzazione (es.mutua esclusione)**

# Operazioni sui semafori: definizione

```
void wait(s)
{ if (s.value == 0)
   <il processo viene sospeso e il suo
   descrittore   inserito in  s.queue>
  else s.value=s.value-1;
}

void signal(s)
{ if (<esiste un processo nella coda s.queue>)
   <il suo descrittore viene estratto da s.queue
   e il suo stato modificato in pronto>
  else s.value=s.value+1;
}
```

# wait/signal

- **Wait:**
  - la **wait(s),** in caso di **s.value=0**, implica la sospensione del processo che la esegue (stato running -> waiting**)** nella coda **s.queue** associata al semaforo.

- **Signal:**
  - l'esecuzione della **signal (s)** non comporta concettualmente nessuna modifica nello stato del processo che l'ha eseguita, ma può causare il risveglio a un processo waiting nella coda **s.queue**.
  - La scelta del processo da risvegliare avviene secondo una politica FIFO (il primo processo della coda).

# Atomicità di wait e signal

Le primitive wait e signal agiscono su variabili condivise e pertanto sono a loro volta **sezioni critiche.**

- Il Sistema Operativo realizza **`wait`** e **`signal`** come operazioni non interrompibili (*system call*):
  - uso di istruzioni tipo test-and-set
  - disabilitazione delle interruzioni

# Mutua esclusione con semafori: Esempio

- Consideriamo due processi P1 e P2 che condividono una struttura dati D sulla quale vogliamo quindi imporre il vincolo di mutua esclusione:

```
shared data D;
```

```
P1:
…
/*sezione critica: */
Aggiorna1(&D);
/*fine sez.critica: */
   …
```

```
P2:
…
/*sezione critica: */
Aggiorna2(&D);
/*fine sez.critica: */
   …
```

Aggiorna1 e Aggiorna2 sono sezioni critiche e devono essere eseguite in modo mutuamente esclusivo.

**Soluzione:** uso di un semaforo (binario) **`mutex`**, il cui valore e` inizializzato a 1.

```
shared data D;
semaphore mutex=1;
mutex.value=1;
```

```
P1:
…
wait(mutex);
Aggiorna1(&D);
signal(mutex);
…
```

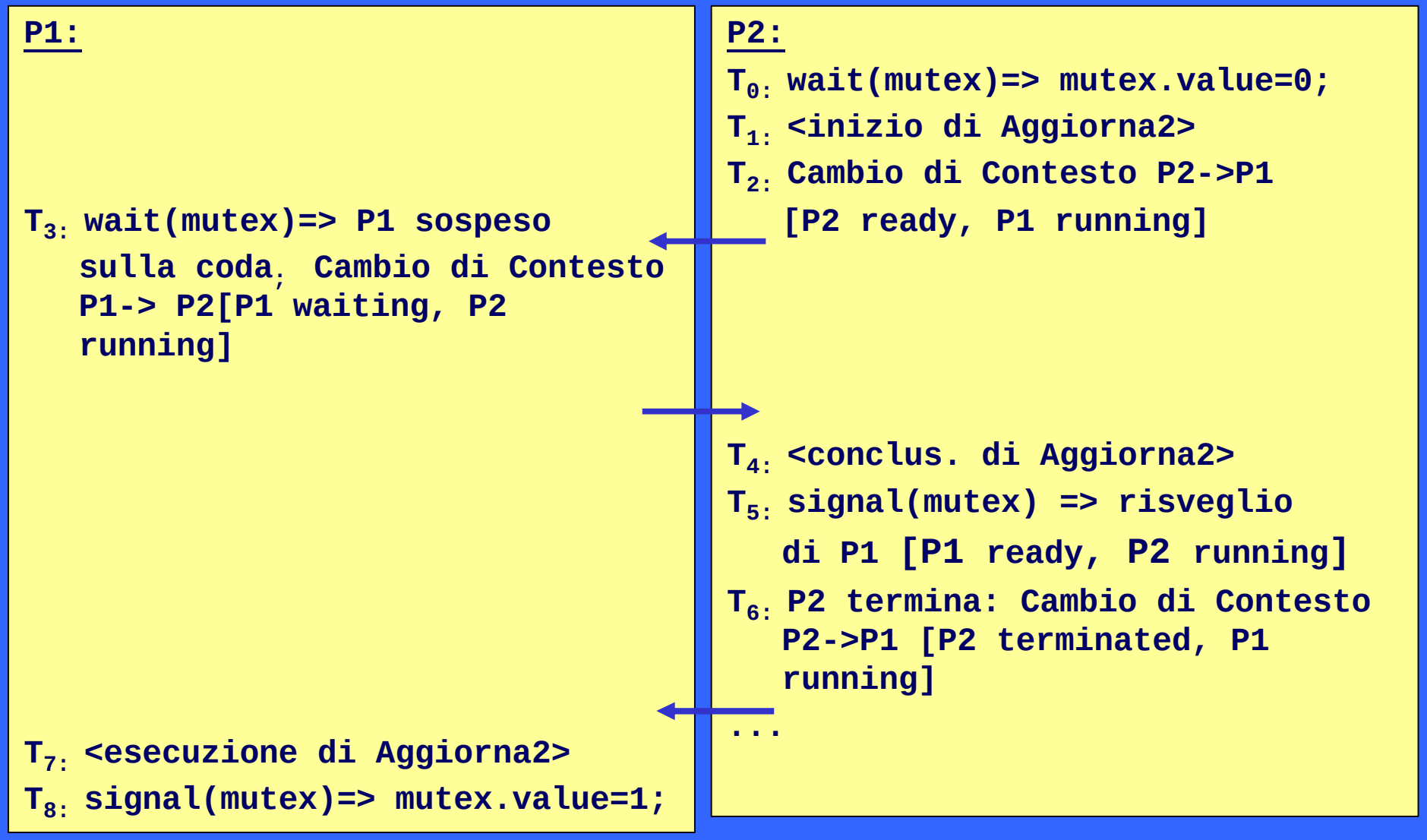
```
P2:
…
wait(mutex)
Aggiorna2(&D);
signal(mutex);
 …
```

- Il valore del semaforo mutex può assumere soltanto due valori (0,1): semaforo ***binario***

➢ la soluzione è sempre **corretta**, indipendentemente dalla sequenza di esecuzione dei processi (e dallo scheduling della CPU).

# Esecuzione

**Ad es:** verifichiamo la seguente sequenza di esecuzione:



# Semafori

Il semaforo puo` utilizzato per risolvere qualunque problema di sincronizzazione:

- mutua esclusione
- lettori/scrittori
- produttore/consumatore
- dining philosophers
- sleeping barber
- .....

# Sincronizzazione di processi cooperanti

- Mediante i semafori possiamo imporre **vincoli temporali** sull'esecuzione di processi

```
P1:
…
/*fase A : */
faseA(…);
/*fine fase A */ …
```

```
P2:
…
/*fase B: */
faseB(…);
/*fine fase B */ …
```

**Obiettivo:** l'esecuzione della fase A (in P1) deve precedere sempre l'esecuzione della fase B (in P2)

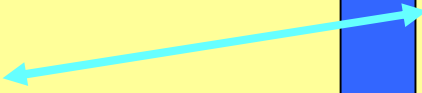
# Sincronizzazione di processi cooperanti

**Soluzione:**

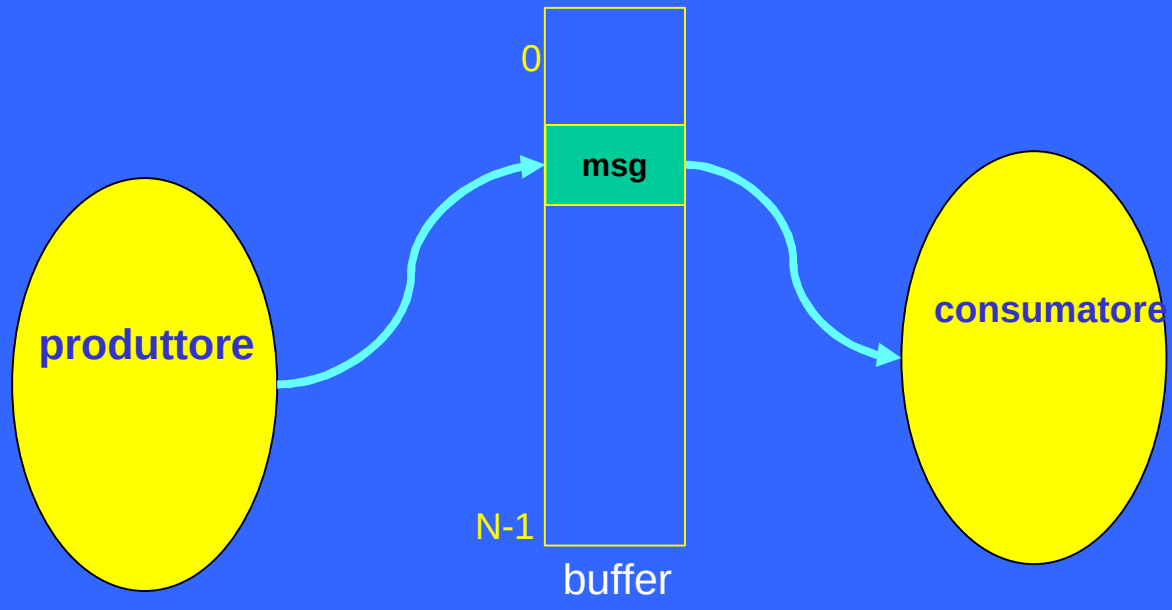
```
semaphore sync;
sync.value=0
```

```
P1:
…
faseA(…);
signal(sync);
…
```

```
P2:
…
wait(sync);
faseB(…);
…
```



# Produttore & Consumatore




***HP*:** Buffer (*mailbox*) limitato di dimensione N

**Requisiti di sincronizzazione:**

- ***mutua esclusione*** nell'esecuzione delle sezioni critiche (accesso e aggiornamento del buffer)
- quando il ***buffer è vuoto*** il consumatore non può prelevare messaggi
- quando il ***buffer è pieno*** il produttore non può depositare messaggi

# Produttore & Consumatore con semafori

- Problema di mutua esclusione:
  - produttore e consumatore non possono accedere contemporaneamente al buffer:
    - introduciamo il semaforo *binario* **`mutex`**, con valore iniziale a 1;

- Problema di sincronizzazione:
  - il produttore non può scrivere nel buffer se questo è pieno:
    - introduciamo il semaforo **`vuoto`**, con valore iniziale a N;
  - il consumatore non può leggere dal buffer se questo è vuoto:
    - introduciamo il semaforo **`pieno`**, con valore iniziale a 0;

```
shared msg Buff [N];
shared semaforo mutex; mutex.value=1;
shared semaforo pieno; pieno.value=0
shared semaforo vuoto; vuoto.value=N
```

```
/* Processo prod. P:*/

main()
{msg M;
 do
 {produco(&M);
  wait(vuoto);
  wait(mutex);
  inserisco(M, Buff);
  signal(mutex);
  signal(pieno);
 }while(true);}
```

```
/* Processo cons. C:*/
main()
{ msg M;
   do
   { wait(pieno);
     wait(mutex);
     prelievo(&M, Buff);
     signal(mutex);
     signal(vuoto);
      consumo(M);
   }while(true);}
```

# Strumenti di Sincronizzazione

- **Semafori:**
  - consentono una efficiente  realizzazione di politiche di sincronizzazione tra processi
  - La correttezza della realizzazione e` completamente a carico del programmatore
- **Alternative:** esistono strumenti di piu` alto livello (costrutti di linguaggi di programmazione) che eliminano a priori il problema della mutua esclusione sulle variabili condivise
  - Variabili Condizione
  - Regioni critiche
  - Monitor
  - ...

# Strumenti di interazione in Linux

Linux e` *multi-threaded*:

- interazione ***intra-task***: uso di strumenti di sincronizzazione
  - semafori
  - mutex, cioe` semafori binari (*pthread* Posix):
  - variabili condizione (*pthread* Posix):

- interazione ***inter-task***: uso dei tradizionali strumenti Unix (pipe, socket, segnali..)

# Approfondimenti

- Concetti generali:
  - Silbershatz: cap.7
  - Andrews: "Foundations of Multithreaded, Parallel and Distributed Programming", Addison-Wesley (cap I e II)
  - P.Ancilotti, M.Boari: "Principi e Tecniche di programmazione Concorrente", II edizione, UTET
- Interazione in Unix:
  - R.W. Stevens: Advanced programming in the Unix Environment, cap.11 e 14
- Linux:
  - Tigran Aivazian, Linux Kernel 2.4 Internals, cap 5. *http://www.linuxdoc.org/guides.html*

# Il file system

# *File system*

**È quella parte del Sistema Operativo che fornisce i meccanismi di accesso e memorizzazione delle informazioni (programmi e dati) allocate in memoria di massa.**

**Realizza :**

- **file: unità logica di memorizzazione**
- **direttorio: insieme di file (e direttori)**
- **partizione: insieme di file associato ad un particolare dispositivo fisico (o porzione di esso)**

➢ **Le caratteristiche di file, direttorio e partizione sono del tutto indipendenti dalla natura e dal tipo di dispositivo utilizzato.**

# Il file

- Ogni file è individuato da (almeno) un **nome** simbolico mediante il quale può essere riferito (ad esempio, nell'invocazione di comandi o system call).
- Ogni file è caratterizzato da un insieme di **attributi**. Solitamente:
  - **tipo**: stabilisce l'appartenenza a una classe (eseguibili, batch, testo, etc)
  - **indirizzo**: puntatore/i a memoria secondaria
  - **dimensione**: numero di byte contenuti nel file
  - **data e ora** (di creazione e/o di modifica)
  - utente **proprietario**
  - **protezione**: diritti di accesso al file per gli utenti del sistema
  - ..

# Descrittore del file

***Descrittore* del file:**

è la struttura dati che contiene gli attributi di un file.

Ogni descrittore di file deve essere memorizzato in modo persistente:

- il S.O. mantiene l'insieme dei descrittori di tutti i file presenti nel file system in apposite strutture in memoria secondaria (ad es. Unix: ***i-list***)

# Operazioni sui file

Compito del S.O. è consentire l'accesso *on-line* ai file, mediante opportune operazioni:

- **Creazione** : allocazione di un file in memoria secondaria ed inizializzazione dei suoi attributi.
- **Lettura** di record logici dal file.
- **Scrittura**: inserimento di nuovi record logici all'interno di file.
- **Cancellazione**: eliminazione del file dal file system.
- ...

➢ ogni operazione richiederebbe la localizzazione di informazioni su disco; ad esempio:

- gli indirizzi dei record logici a cui accedere
- gli altri attributi del file
- i record logici

# Operazioni sui file

## Per migliorare l'efficienza:

- il S.O. mantiene in memoria una tabella che registra i file attualmente in uso (file ***aperti***): per ogni file aperto:

  ***{puntatore al file, posizione su disco,..}***

- *memory mapping* dei file aperti:

  i file aperti vengono temporaneamente copiati in memoria centrale -> accessi più veloci.

## Operazioni necessarie:

- **Apertura** : introduzione di un nuovo elemento nella tabella del file aperti e memory mapping del file.
- **Chiusura**: salvataggio del file in memoria secondaria e eliminazione dell'elemento corrispondente dalla tabella dei file aperti.

# Struttura interna dei file

Ogni dispositivo di memorizzazione secondaria viene partizionato in **blocchi** (o ***record fisici***):

- **Blocco**: unità di trasferimento nelle operazioni di I/O da/verso il dispositivo; la sua dimensione è fissa.

L'utente vede il file come un insieme di **record logici:**

- **Record logico**: unità di trasferimento nelle operazioni accesso al file (es. lettura, scrittura di blocchi); la sua dimensione può variare.

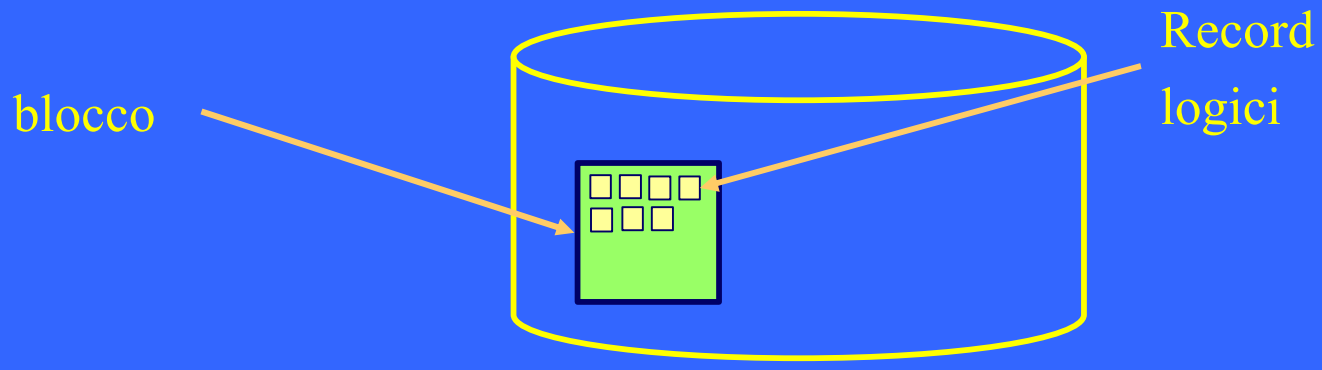
# Blocchi & record logici

Uno dei compiti del Sistema Operativo (del file system) è stabilire una **corrispondenza tra record logici e blocchi**.

**Di solito:**

**Dimensione(blocco) >> Dimensione(record logico)**

- *impaccamento* di record logici all'interno di blocchi.



# Metodi di accesso

L’accesso a file può avvenire secondo varie modalità:

- accesso **sequenziale**
- accesso **diretto**
- accesso a **indice**

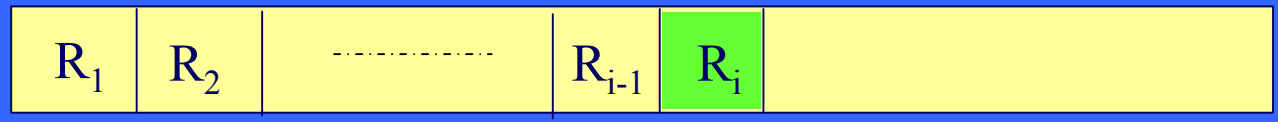
Il metodo di accesso è indipendente:

- **dal tipo di dispositivo** utilizzato;
- **dalla tecnica di allocazione** dei blocchi in memoria secondaria.

# Accesso sequenziale

Il file è una **sequenza** [$R_1$, $R_2$,.. $R_N$] di record logici:

- per accedere ad un particolare record logico $R_i$ , è necessario accedere prima agli (*i-1*) record che lo precedono nella sequenza:




- in questo caso le operazioni di accesso sono del tipo:
  ***readnext***: lettura del prossimo record logico della sequenza
  ***writenext***: scrittura del prossimo record logico

- ogni operazione di accesso (lettura/scrittura) posiziona il **puntatore al file** sull'elemento successivo a quello letto/scritto.

- **Linux** prevede accesso sequenziale.

# Accesso diretto

Il file è un **insieme non ordinato** $\{R_1, R_2, .. R_N\}$ di record logici:

- si può accedere direttamente ad un particolare record logico specificandone l'identificatore.

- in questo caso le operazioni di accesso sono del tipo:

  ***read i***: lettura del record logico i

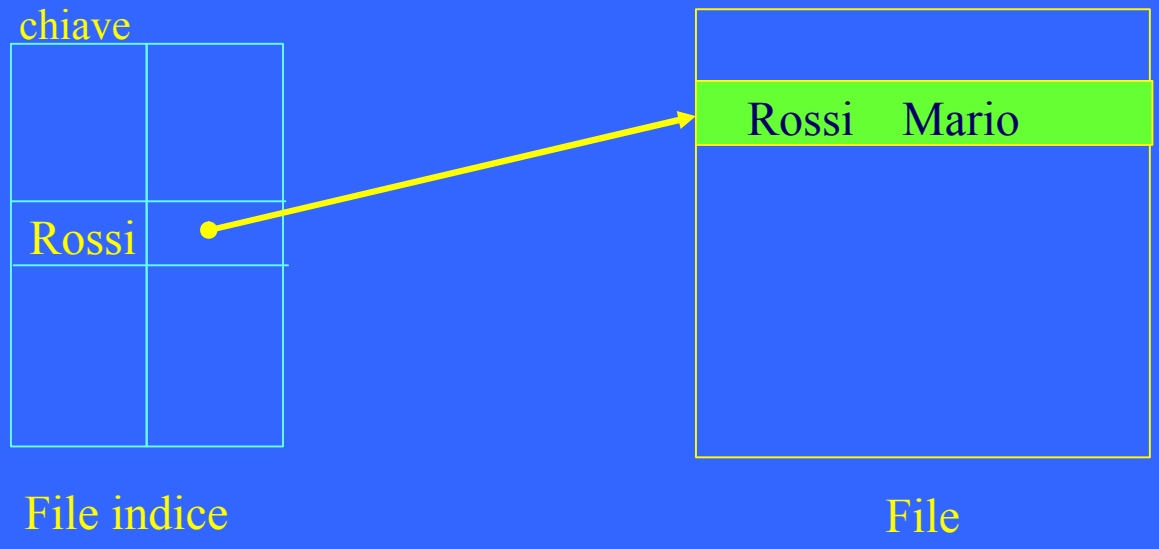
  ***write i***: scrittura del record logico i

- Utile, quando si vuole accedere a grossi file, per estrarre/aggiornare poche informazioni (es. database).

# Accesso a indice

Ad ogni file viene associata una struttura dati (*di solito, un file*) contenente l'***indice*** delle informazioni contenute nel file.

- per accedere a un record logico, si esegue una ricerca nell'indice (utilizzando una *chiave*):



# Il direttorio

È lo strumento per organizzare i file all'interno del file system:

- un direttorio può contenere più files
- è realizzato mediante una struttura dati che associa al nome di ogni file la posizione del file nel disco.

## Operazioni sui direttori:

- **Creazione** e **cancellazione** di direttori
- **Aggiunta/cancellazione** di file.
- **Listing**: elenco di tutti i file contenuti nel direttorio.
- **Attraversamento** del direttorio.
- **Ricerca** di file nel direttorio.

# Tipi di direttorio

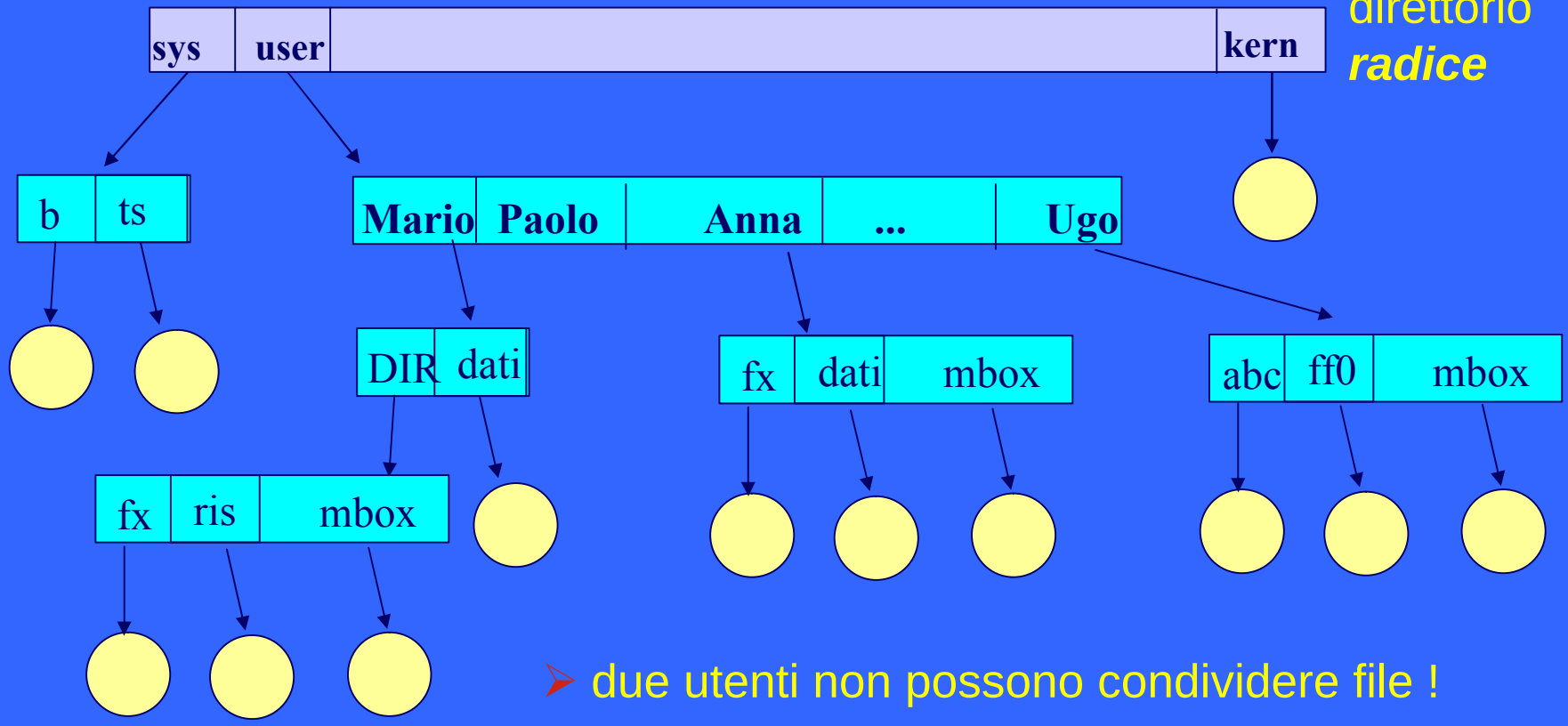
La struttura logica del direttorio può variare a seconda del Sistema Operativo

**Schemi più comuni:**

- ad **albero**
- a **grafo aciclico**

# Tipi di direttorio

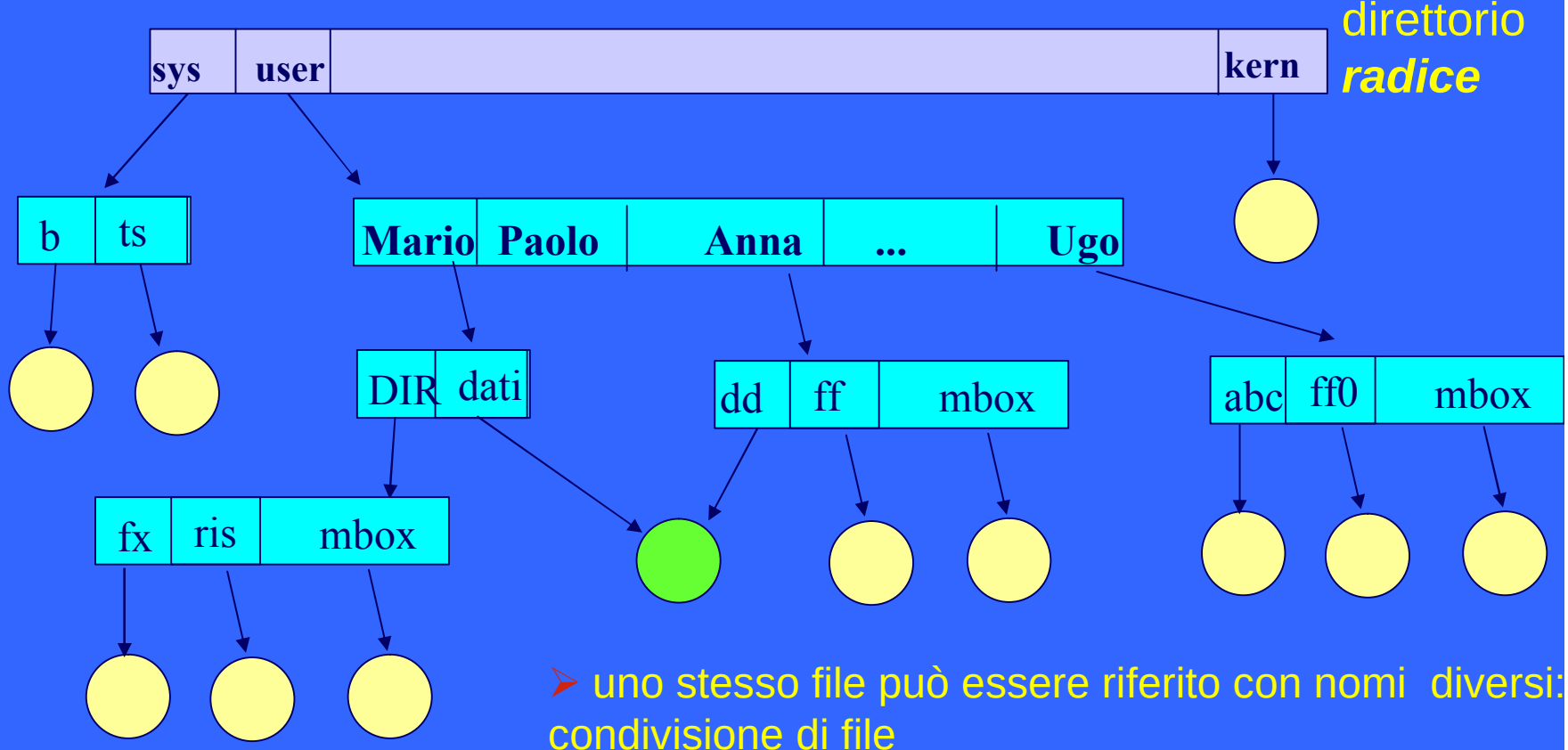
**Struttura ad albero:** ogni direttorio può contenere file e altri direttori. Organizzazione gerarchica a N livelli.




- due utenti non possono condividere file !

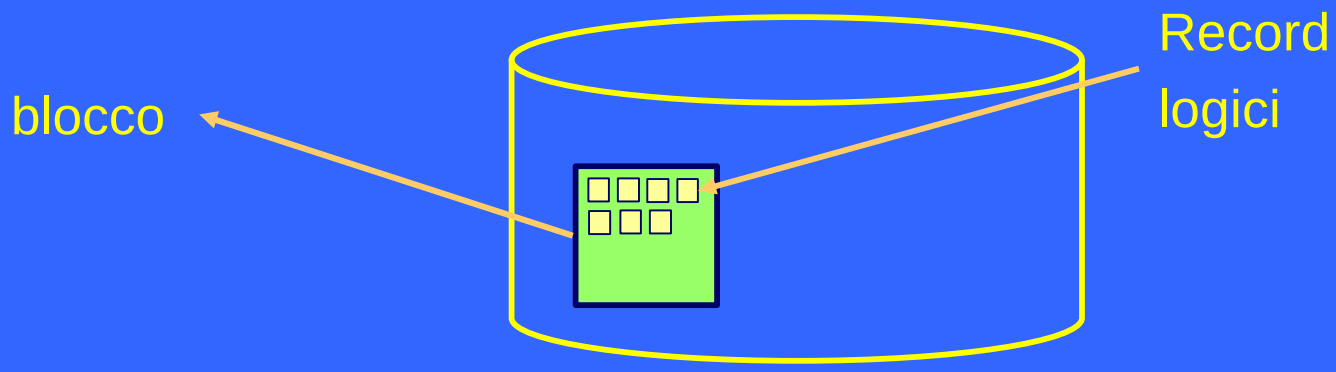
# Tipi di direttorio

**Struttura a grafo aciclico (es. UNIX):** estende la struttura ad albero per permettere la **condivisione di file.**




- uno stesso file può essere riferito con nomi diversi: condivisione di file

# Tecniche di Allocazione



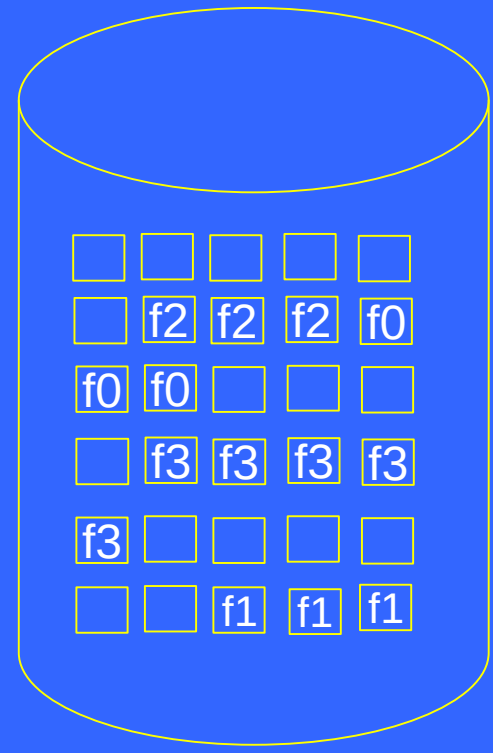

Ogni **blocco** contiene un insieme di **record logici contigui.**

Tecniche più comuni per l'allocazione dei blocchi sul disco:

- allocazione **contigua**
- allocazione a **lista**
- allocazione a **indice**

# Allocazione contigua

Ogni file è allocato su un insieme di blocchi fisicamente contigui.




**Pro**:

- costo della ricerca di un blocco
- efficienza nell'accesso sequenziale e diretto

**Contro**:

- individuazione dello **spazio libero** per l'allocazione di un nuovo file
- **Frammentazione** esterna: man mano che si riempie il disco, rimangono zone contigue sempre più piccole, a volte inutilizzabili:
  - **compattazione**
- **Aumento** dinamico delle **dimensioni** di file

# Allocazione a lista

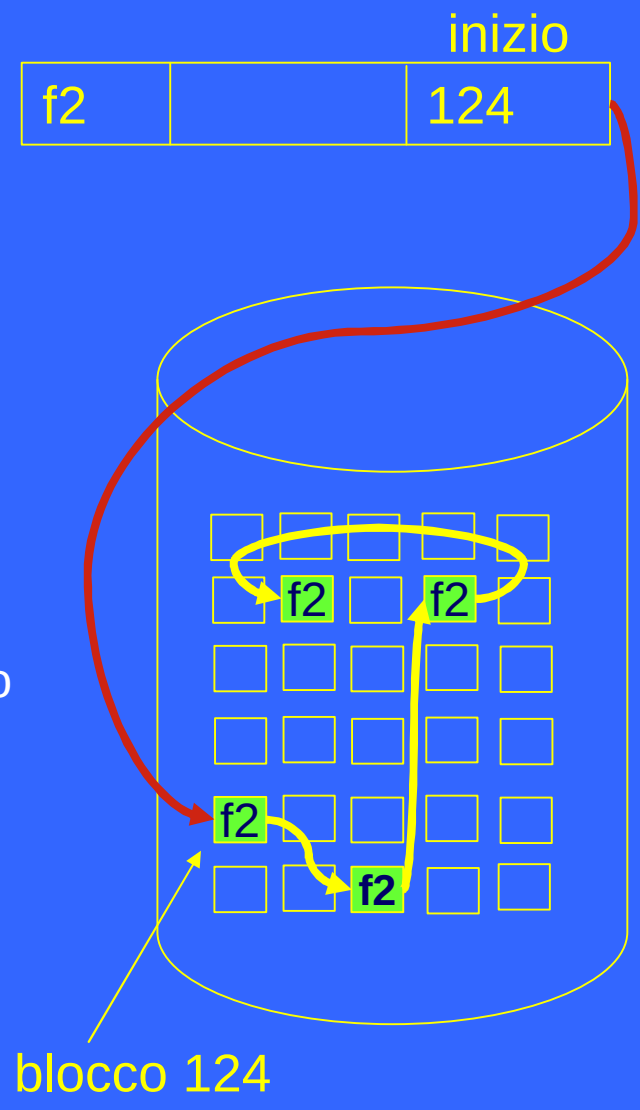
I blocchi sui quali viene mappato ogni file sono organizzati in una lista concatenata.

**Pro**:

- non c'è frammentazione esterna
- minor costo di allocazione

**Contro**:

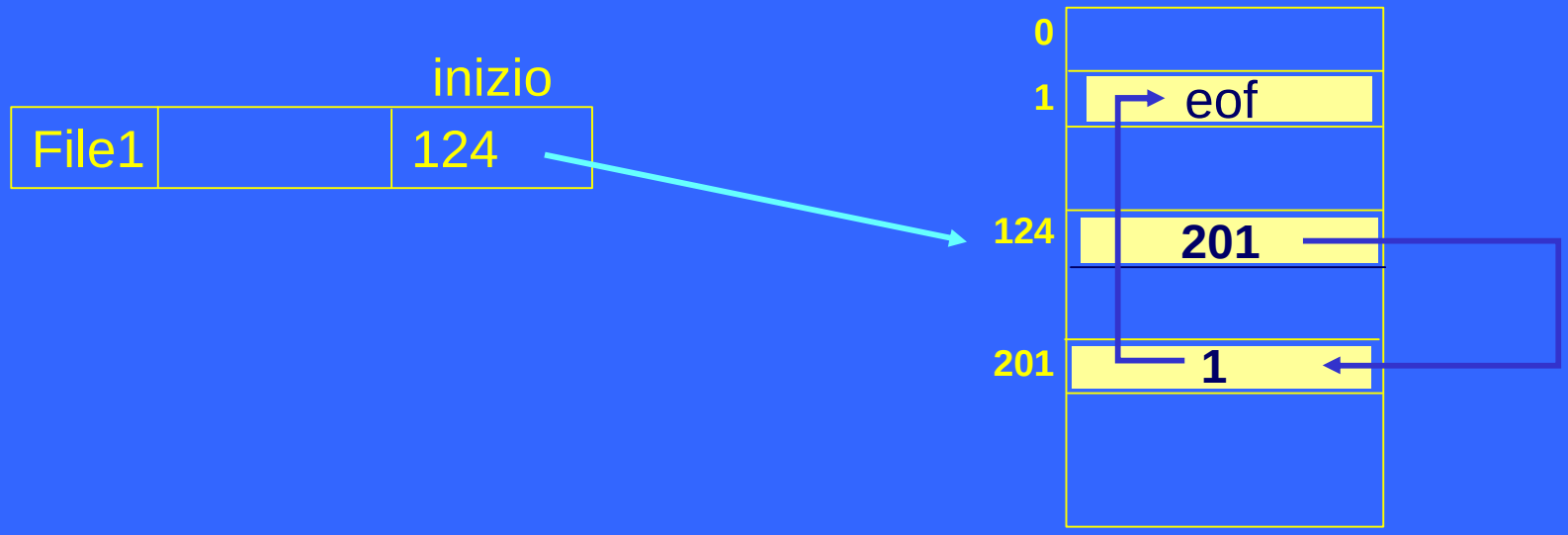
- possibilità di errore se un link viene danneggiato
- maggior occupazione (spazio occupato dai puntatori)
- difficoltà di realizzazione dell'accesso diretto
- costo della ricerca di un blocco



# Tabella di allocazione dei file (*FAT*)

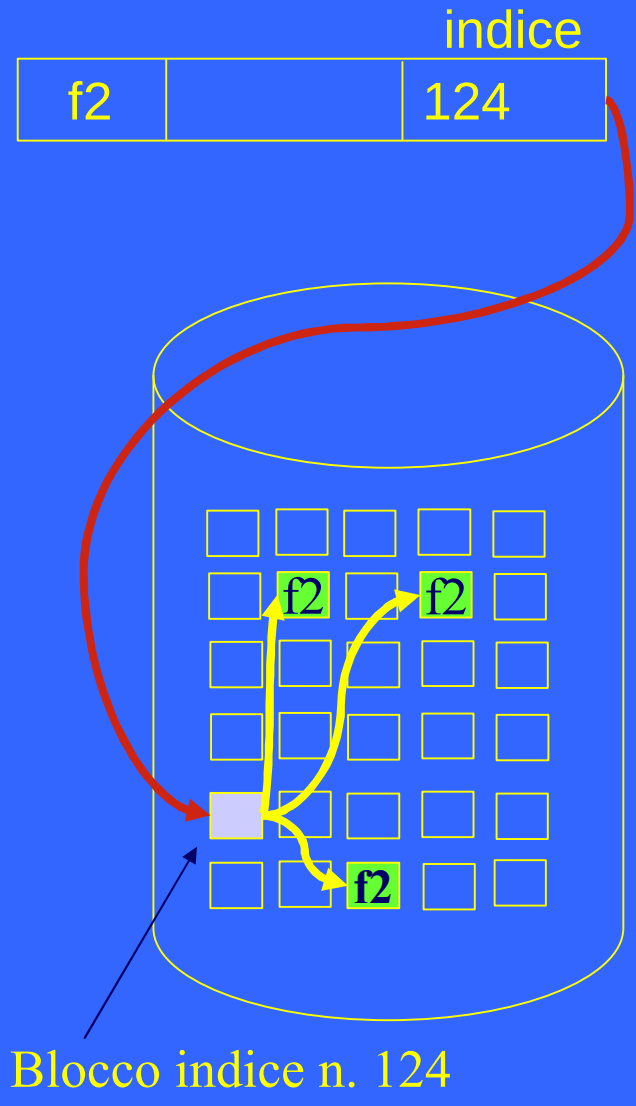
Alcuni sistemi operativi (ad es. DOS e OS/2) realizzano l'allocazione a lista in modo più efficiente e robusto:

- per ogni partizione, viene mantenuta una tabella (FAT) in cui ogni elemento rappresenta un blocco fisico.
- Il concatenamento dei blocchi sui quali è allocato un file è rappresentato nella FAT:



# Allocazione a indice

A ogni file è associato un **blocco** (***indice***) in cui sono contenuti tutti gli indirizzi dei blocchi su cui è allocato il file.




**Vantaggi**:

- gli stessi dell'allocazione a lista, più
  - efficienza nell' accesso diretto
  - maggiore velocità di accesso (rispetto a liste)

**Svantaggi**:

- scarso utilizzo dei blocchi indice

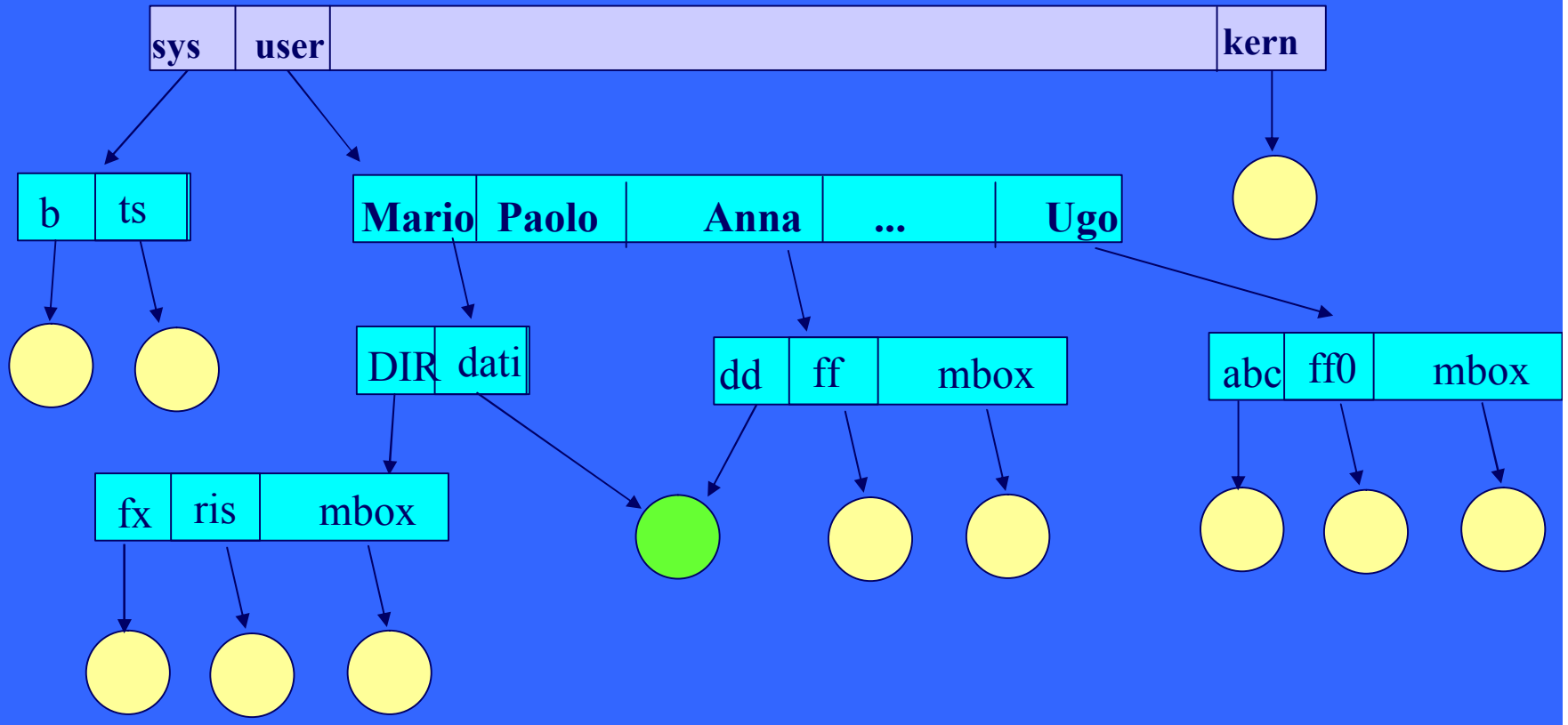
# Metodi di Allocazione

- Riassumendo, gli aspetti caratterizzanti sono:
  - grado di **utilizzo** della memoria
  - tempo di **accesso** medio al blocco
  - realizzazione dei **metodi di accesso**

  - **Esistono sistemi operativi che adottano più di un metodo di allocazione; spesso:**
    - file **piccoli** ➔ allocazione contigua
    - file **grandi** ➔ allocazione a indice

# Il file system nei sistemi *nix

- omogeneita` (tutto e` file)
- organizzazione logica a DAG



# Nome, *i-number, i-node*

- ad ogni file possono essere associati **uno o più nomi simbolici**

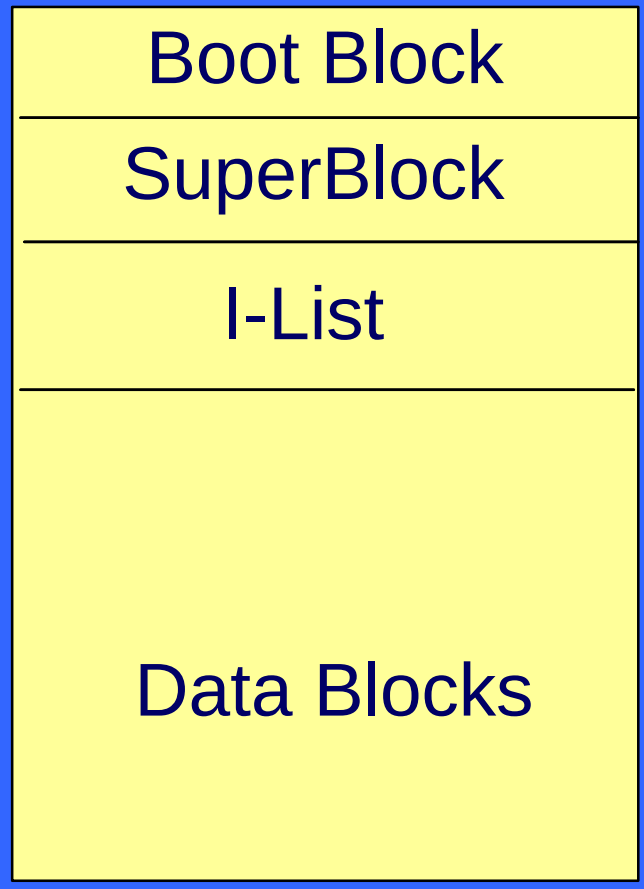
*ma*

- ad ogni file è associato **uno ed un solo descrittore** (***i-node)***, univocamente identificato da un **intero** (***i-number***)

# Organizzazione Fisica del file system di Unix

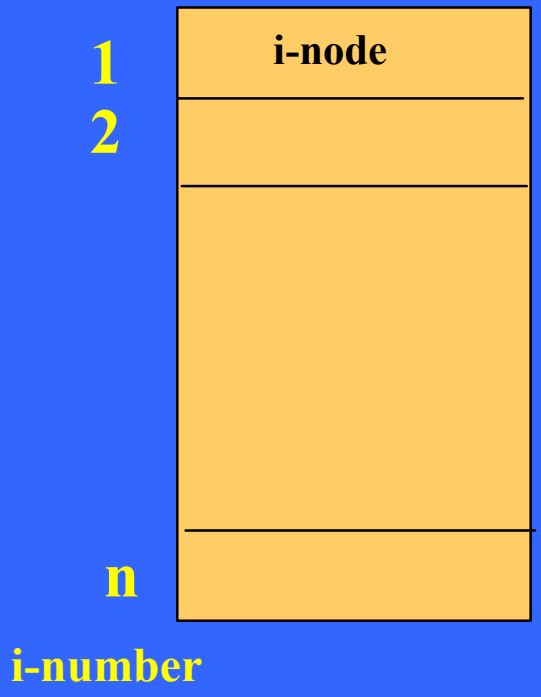
- Il metodo di allocazione utilizzato in Unix è ad **indice (***a più livelli di indirizzamento***)**
- formattazione del disco in **blocchi di dimensioni** 512- 4096 *Bytes*.
- La superficie del disco File System è partizionata in **4 regioni:**
  - **boot block:** contiene le procedure di inizializzazione del sistema (da eseguire al *bootstrap*)
  - **super block:** descrive la mappa del disco (i limiti delle 4 regioni, il puntatore alla **lista dei blocchi liberi,** ecc.)
  - **i-list:** contiene tutti i descrittori (***i-node***) dei file, direttori e dispositivi presenti nel file system
  - **data blocks:** l'area del disco effettivamente disponibile per la memorizzazione dei file; contiene i blocchi allocati e i blocchi liberi (organizzati in una lista)

# Il File System
## Organizzazione Fisica



# Il File System
## Organizzazione Fisica

- ***i-List***: contiene la lista di tutti i descrittori (***i-node)*** dei file, direttori e dispositivi presenti nel file system del file system (accesso con l'indice ***i-number***)



# i-node

È il **descrittore** del file.

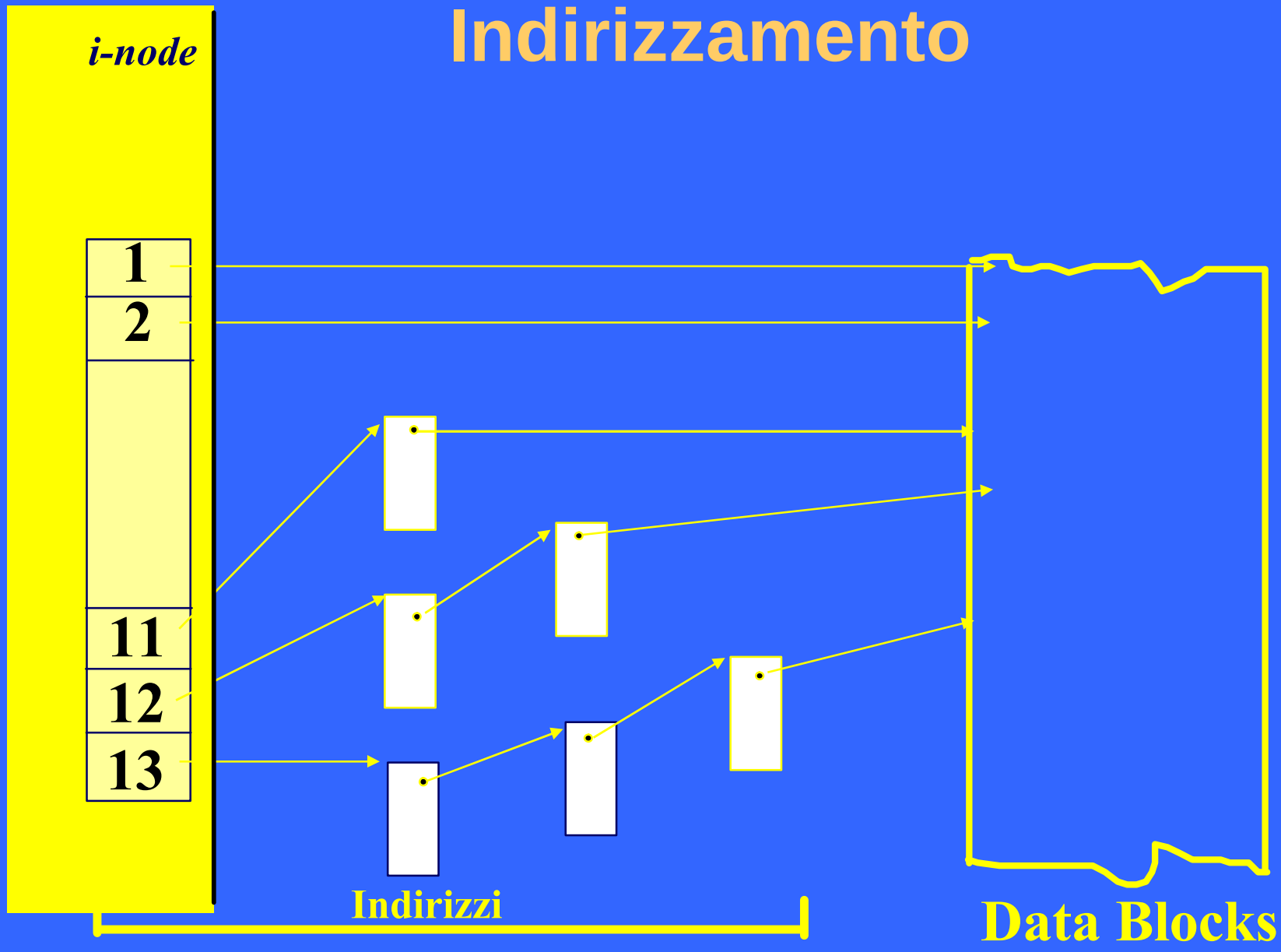
- Tra gli **attributi** contenuti nell'*i-node* vi sono:
  - **tipo** di file:
    - » **ordinario**
    - » **direttorio**
    - » file **speciale**, per i dispositivi.
  - **proprietario, gruppo** (user-id, group-id)
  - **dimensione**
  - **data**
  - 12 bit di **protezione**
  - numero di **links**
  - **13 -15 indirizzi** di blocchi ( a seconda della realizzazione)

# 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 indirizzi**: riferiscono blocchi di dati (indirizzamento *diretto*)
- **11° indirizzo** : indirizzo di un blocco contenente a sua volta indirizzi di blocchi dati (1 livello di *indirettezza*)
- **12° indirizzo**: due livelli di *indirettezza*
- **13° indirzzo**: tre livelli di *indirettezza*

# Indirizzamento



# Indirizzamento

**Hp**: dimensione del blocco **512** byte=0,5 KB

indirizzi di 32 bit (4 byte)

- 1 blocco contiene **128** indirizzi

## Quindi:

- 10 blocchi di dati sono accessibili **direttamente**
  - file di dimensioni minori di (10*512)byte=5120 byte =5 KB sono accessibili direttamente
- 128 blocchi di dati sono sono accessibili con ***indirezione singola*** (mediante il puntatore 11): **(128*512)byte = 65536 byte=64KB**
- 128*128 blocchi di dati sono accessibili con ***indirezione doppia*** (mediante il puntatore 12): **128*128*512 byte= 8MB**
- 128*128*128 blocchi di dati sono accessibili con ***indirezione tripla*** (mediante il puntatore 13): **128*128*128*512 byte= 1GB**

# Indirizzamento

- la dimensione massima del file è dell'ordine del **Gigabyte** :

**Dimensione massima = 1GB+ 8MB+64KB+5KB**

➔ **l'accesso a file di piccole dimensioni è più veloce rispetto al caso di file grandi**

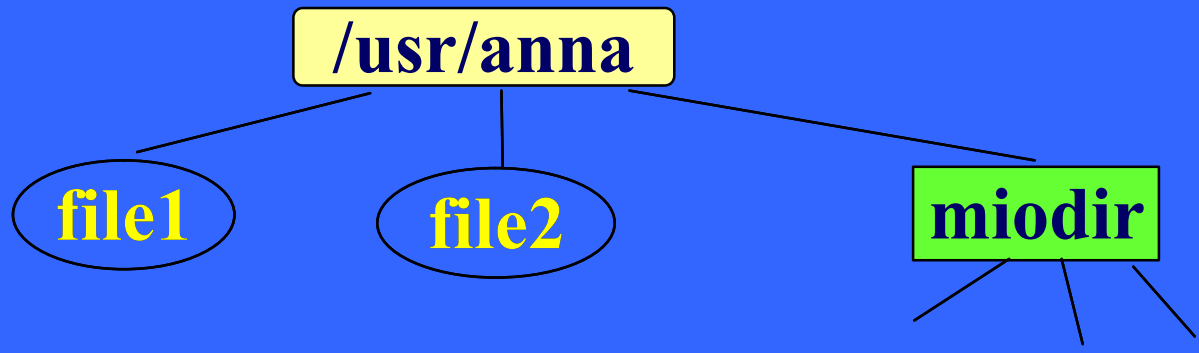
# Il Direttorio

- Anche il direttorio è rappresentato nel *file system* da un file.
- Ogni file-direttorio contiene un insieme di record logici con la seguente struttura:

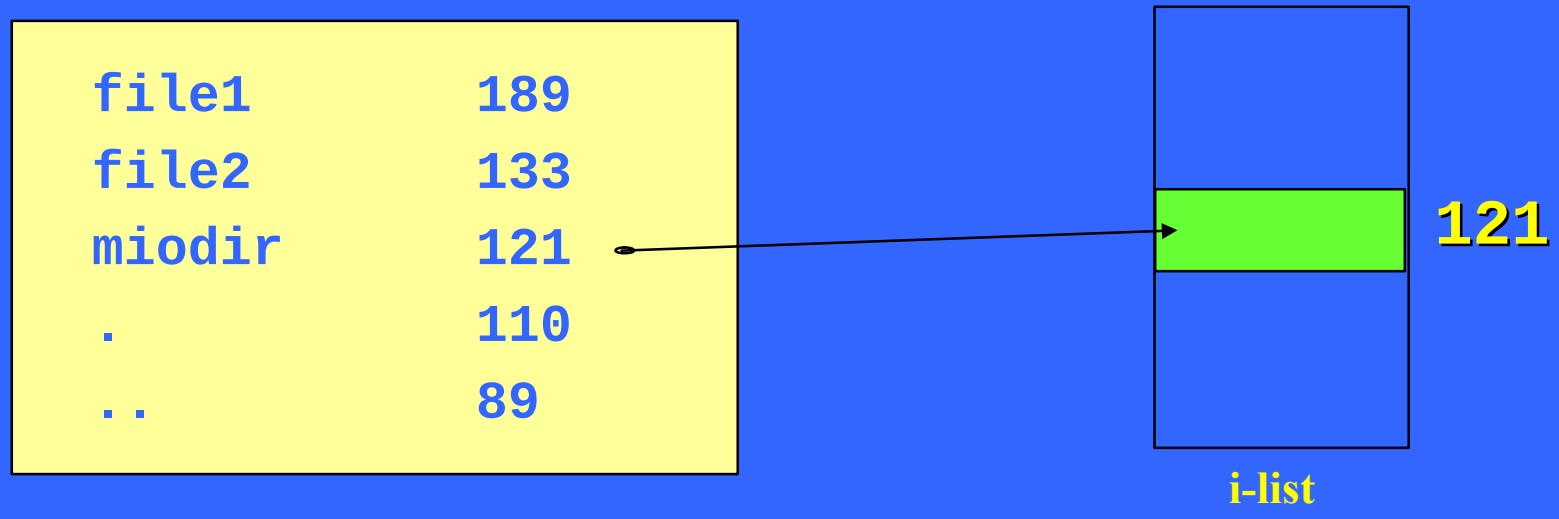
| nomerelativo | i-number |
|---|---|

- ogni record rappresenta un file appartenente al direttorio:
  - ➔ per ogni file (o direttorio) appartenente al direttorio considerato, viene memorizzato il suo nome relativo, al quale viene associato il valore del suo ***i-number*** (che lo identifica univocamente)

# Il Direttorio



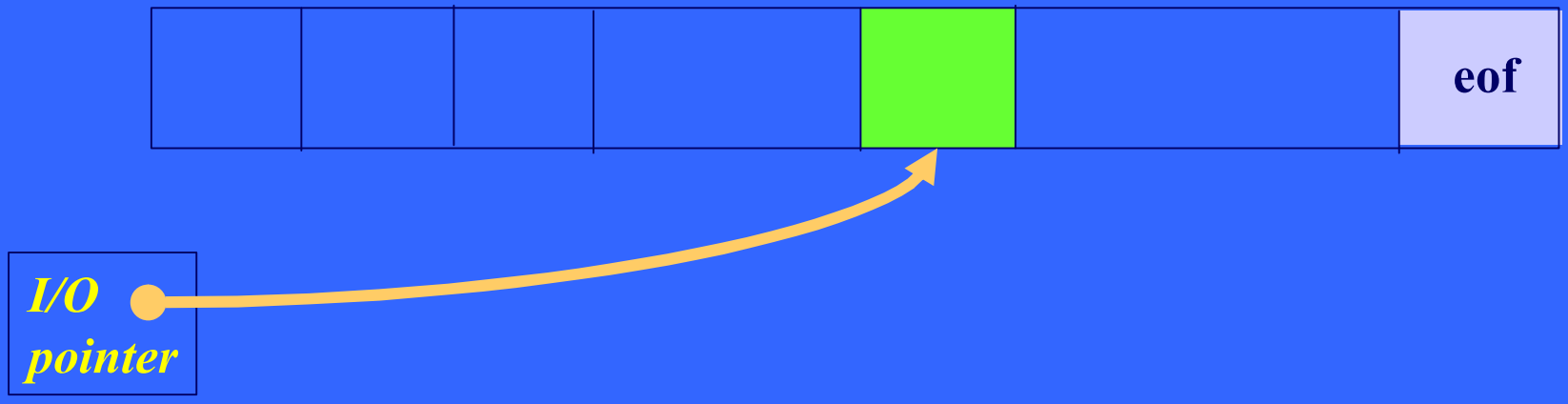

il file (direttorio) **`/usr/anna`** contiene:



# Meccanismi di accesso

- accesso sequenziale
- assenza di strutturazione:

  **file = sequenza di bytes (*stream*)**
- posizione corrente: **I/O Pointer**



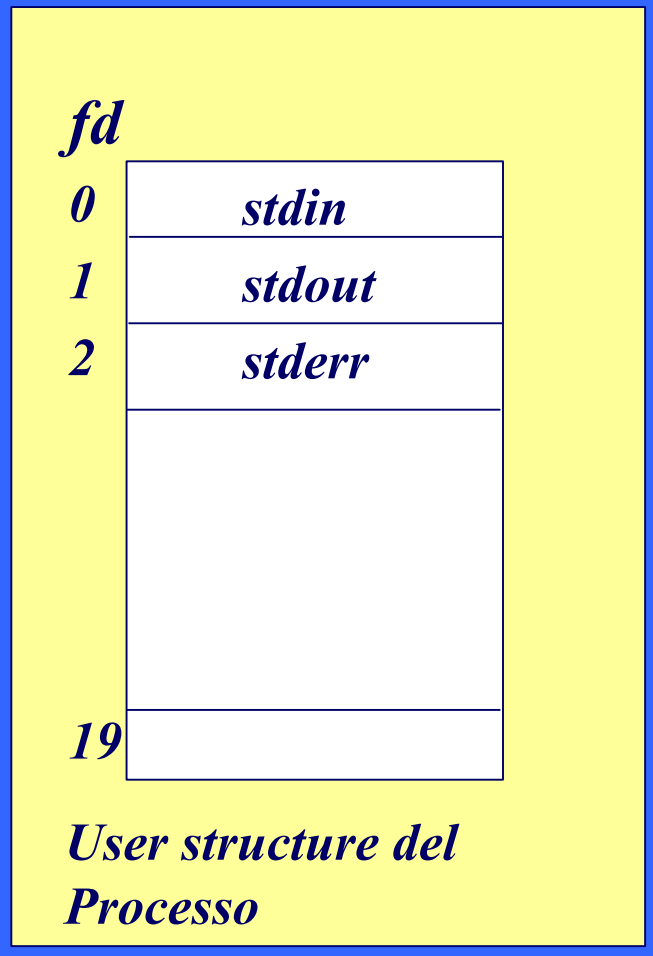
# Gestione di File

## *File Descriptor*

- A ogni processo è associata una **tabella dei file aperti** di dimensione limitata (tipicamente, 20 elementi)
- ogni elemento della tabella rappresenta un file aperto dal processo ed è individuato da un indice intero:

***file descriptor***

- la tabella dei file aperti del processo è allocata nella sua **user structure**
- i file descriptor **0,1,2** individuano rispettivamente standard input, output, error (aperti automaticamente)



# Gestione di File
## Strutture dati del Kernel

Per realizzare l'accesso ai file, il sistema operativo utilizza **due strutture dati** globali, allocate nell'area dati del kernel:

- la **tabella dei file attivi**: per ogni file aperto, contiene una copia del suo i-node:
  - in questo modo si rendono più efficienti le operazioni di accesso evitando accessi al disco per ottenere attributi dei file acceduti.

- la **tabella dei file aperti di sistema:** ha un elemento per ogni operazione di apertura relativa ad un file aperto; ogni elemento contiene:
    - » **l'I/O pointer**, che indica la posizione corrente all'interno del file
    - » un puntatore all'***i-node*** del file nella tabella dei file attivi
  - se due processi aprono separatamente lo stesso file *F*, la tabella conterrà due elementi distinti associati a *F*.

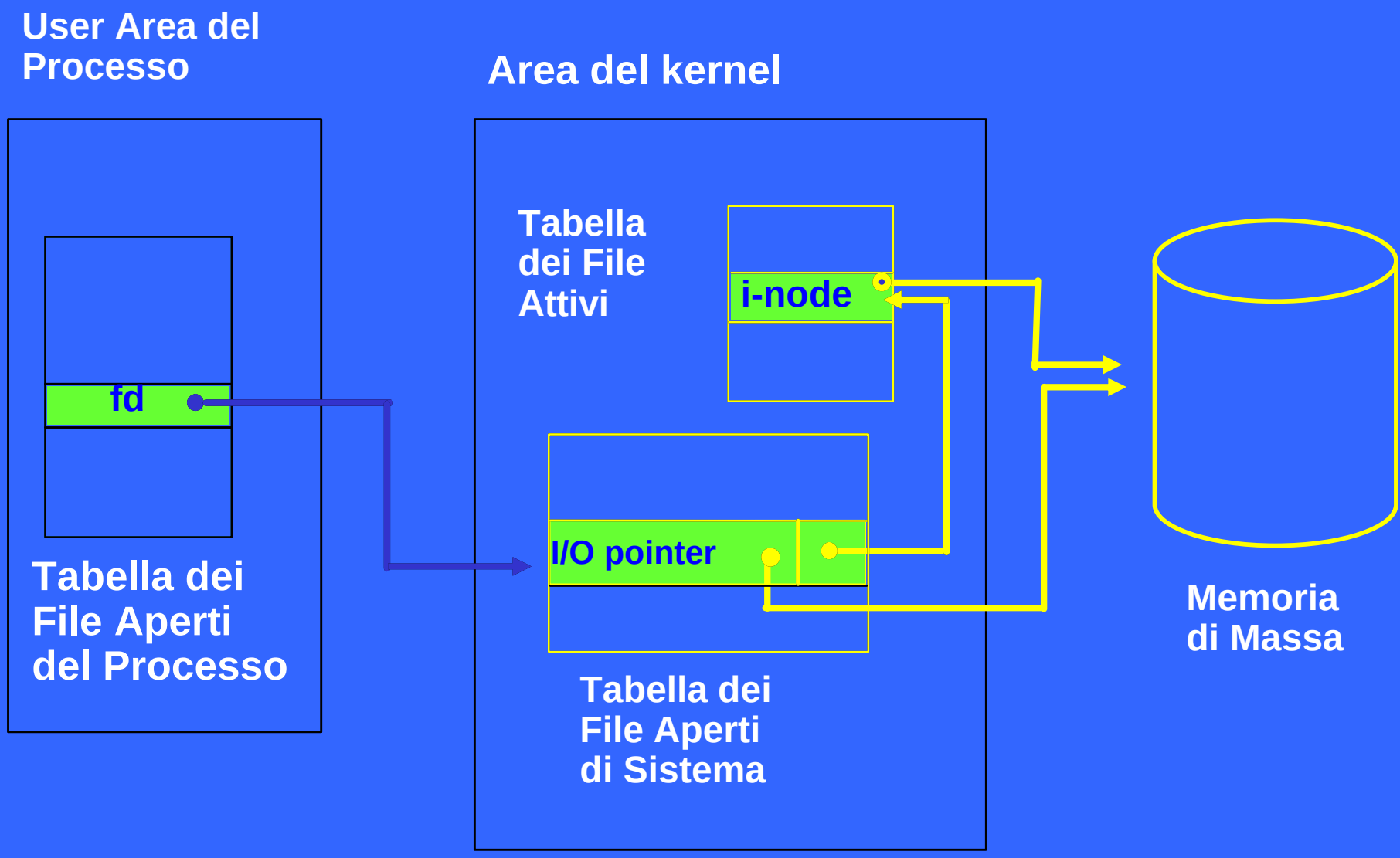
# Gestione di File
## Strutture dati

## Riassumendo:

- **tabella dei file aperti di processo:** nella user area del processo, contiene un elemento per ogni file aperto dal processo
- **tabella dei file aperti di sistema:** contiene un elemento per ogni sessione di accesso a file nel sistema
- **tabella dei file attivi**: contiene un elemento per ogni file aperto nel sistema

**Quali sono le relazioni tra queste strutture?**

# Strutture Dati



# Gestione di File

## Tabella dei File Aperti di Sistema

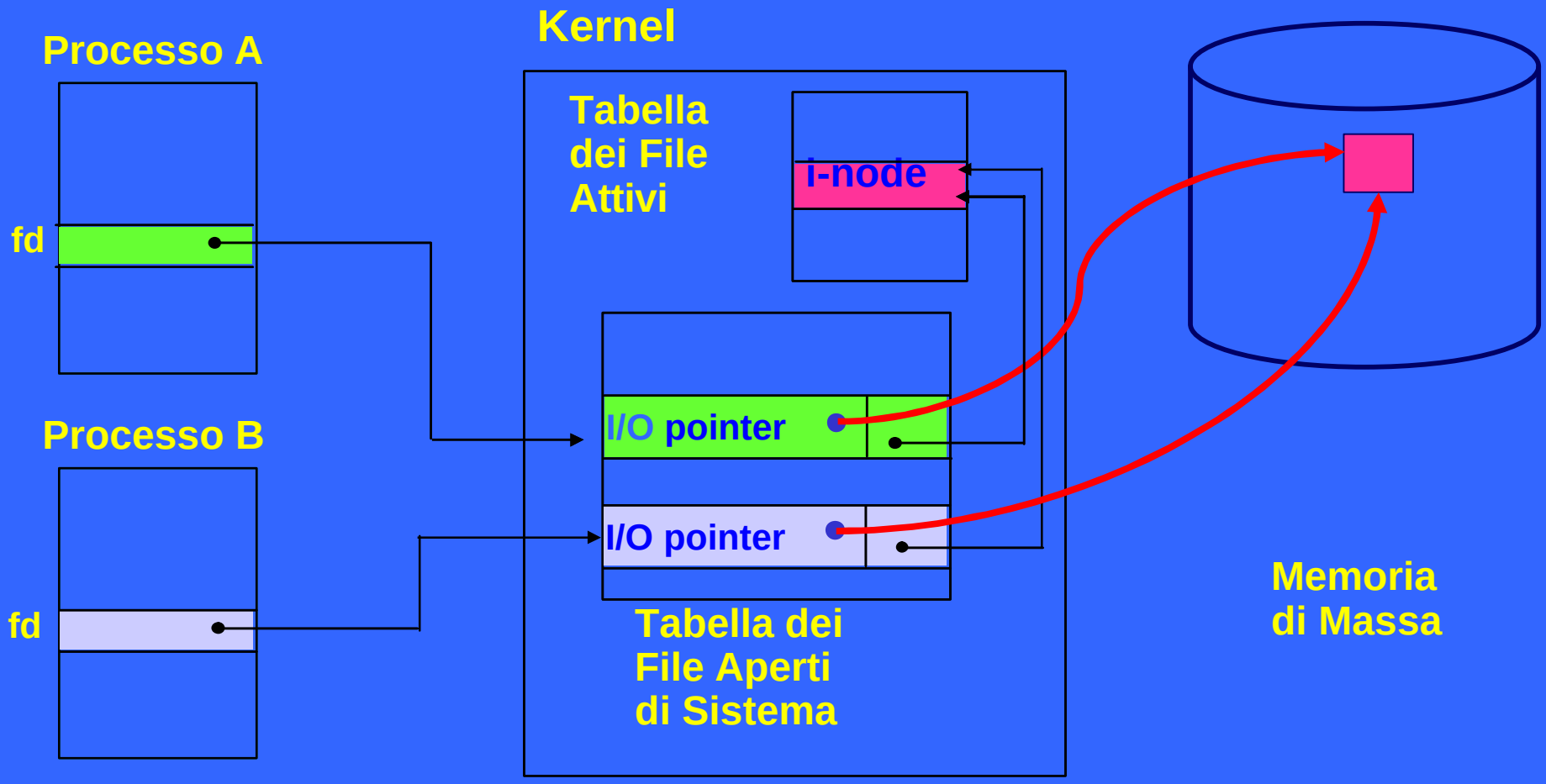
- un elemento per ogni "**apertura**" di file: a processi diversi che accedono allo stesso file, corrispondono entry distinte
- ogni elemento contiene il puntatore alla posizione corrente (I/O pointer)

più processi possono accedere contemporaneamente allo stesso file, ma con **I/O pointer distinti**

# Gestione di File
## Tabella dei File Attivi

- l'operazione di **apertura** provoca la **copia** dell'i-node in memoria centrale (se il file non è già in uso)
- la tabella  dei file attivi contiene gli ***i-node*** di tutti i file aperti
- il numero degli elementi è pari al numero dei file aperti (anche da più di un processo)

# Esempio: i processi A e B (indipendenti) accedono allo stesso file, ma con I/O pointer distinti

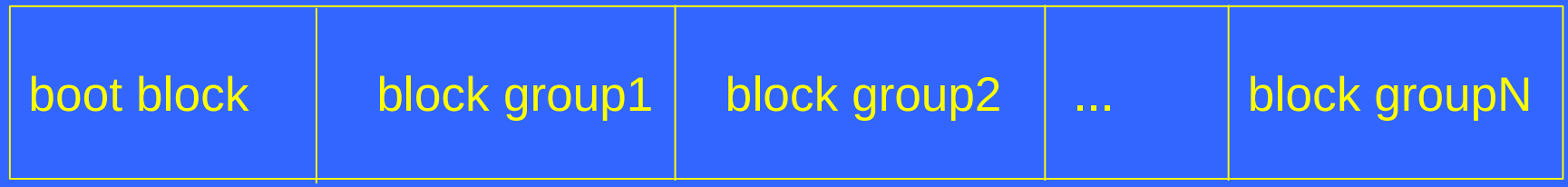


# File system Linux

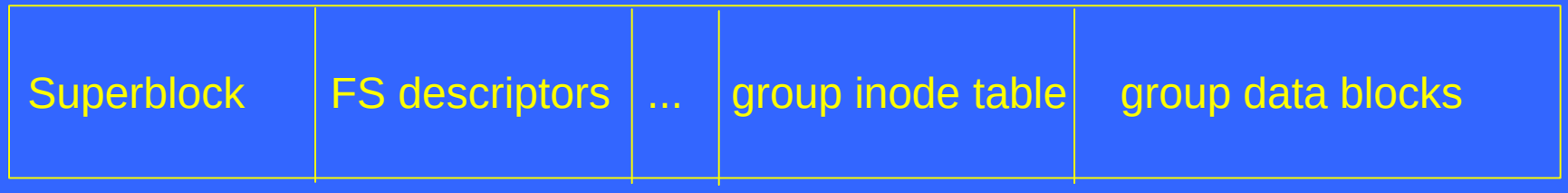
- Prime versioni: implementazione del file system di Minix (unix-like): limitazioni su:
  - indirizzabilita` (64 MB per file e per filesystem!)
  - nomi dei file (16 caratteri)
  - gestione dei blocchi inefficiente
- **Ext FS (1992): indirizzabilita` 2GB/2GB**
- **Ext 2 (1993): 2GB/4TB**
  - Flessibilita`: l'amministratore puo` decidere
    - dimensione del blocco (1024-4096 bytes)
    - dimensione i-list
  - Efficienza:
    - ***Gruppi di blocchi***: ogni gruppo include data blocks e inode memorizzati in tracce adiacenti e una copia delle strutture di controllo (superblock e descrittore filesystem) -> affidabilita`
    - ***fast symbolic links: il link e` memorizzato direttamente nell'i-node***
    - ***Preallocazione***: il filesystem prealloca blocchi a file prima che vengano effettivamente scritti

# EXT2 Block groups:

- il file system e` organizzato fisicamente in gruppi di blocchi:




- La struttura di ogni gruppo e` la seguente:



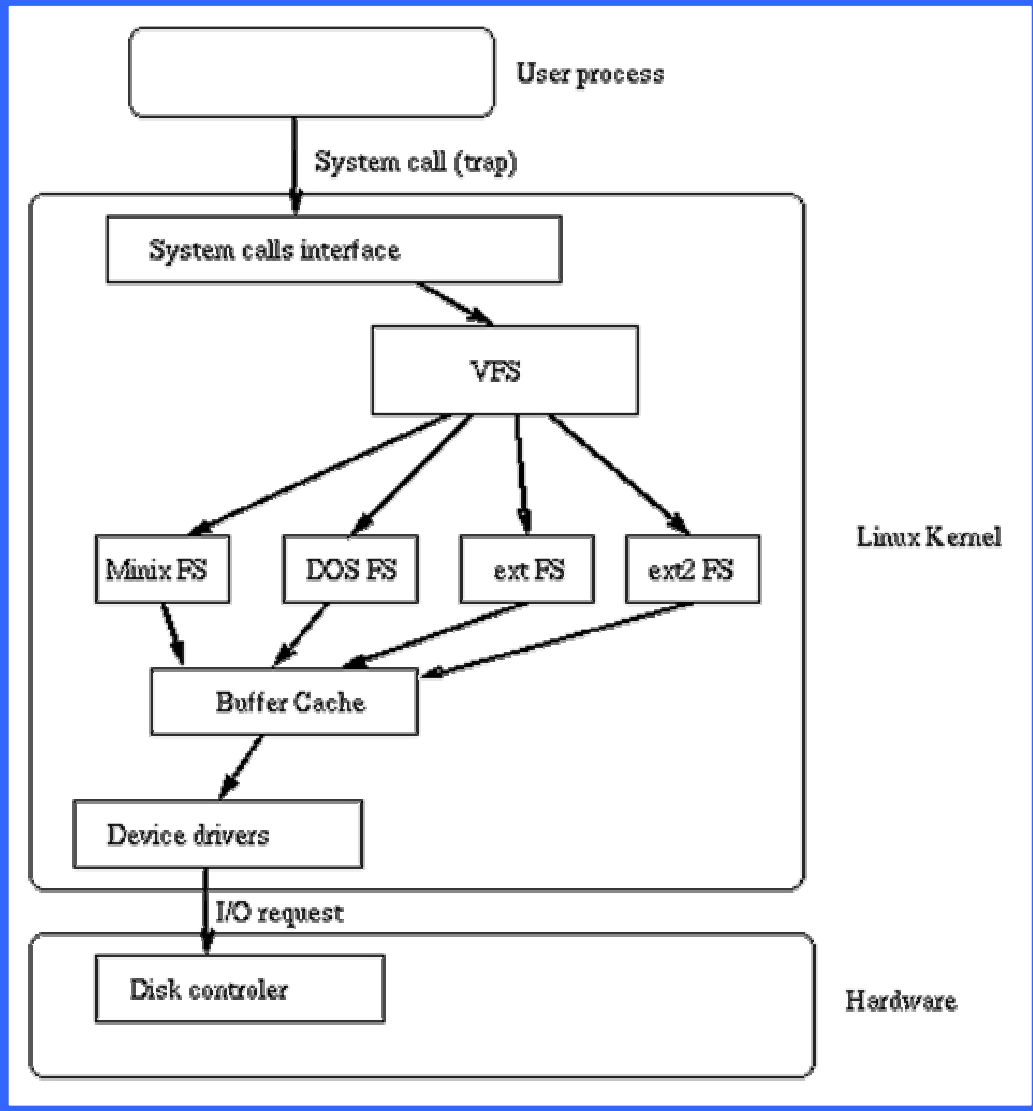

- localita` di inode e relativi file: tempi di accesso ridotti
- localita` dei blocchi allocati a uno stesso file

# Virtual File System

Linux prevede l'integrazione con filesystem diversi da Ext2, grazie al Virtual File System (VFS):

- intercetta ogni system call relativa all'accesso al file system e, all'occorrenza, provvede al collegamento con file system "esterni":
- file e filesystems sono mappati su *internal objects* nel kernel, ognuno dei quali include informazioni specifiche sulla risorsa che rappresenta:
  - superblock objects
  - i-node objects
  - file objects

# VFS


User process
System call (trap)
System calls interface
VFS
Minix FS
DOS FS
ext FS
ext2 FS
Linux Kernel
Buffer Cache
Device drivers
I/O request
Disk controler
Hardware

# VFS: file system supportati

- Disk File system:
  - unix-like FS: SystemV, BSD, ecc.
  - Microsoft-like: DOS, VFAT (Win98), NTFS..
  - HFS (Apple Macintosh)
  - ...
- Network file systems:
  - NFS
  - SMB (Microsoft)
  - NCP (Novell Net Ware Core Protocol)
  - ...

# Approfondimenti

- Concetti generali sul file system:
  - Silbershatz, Galvin: Cap. 10-11
- File system Unix:
  - R.W. Stevens: Advanced programming in the Unix Environment, cap.4
- File system di Linux: http://e2fsprogs.sourceforge.net/ext2intro.html