Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica

Sistemi operativi e reti

A.A. 2016-17

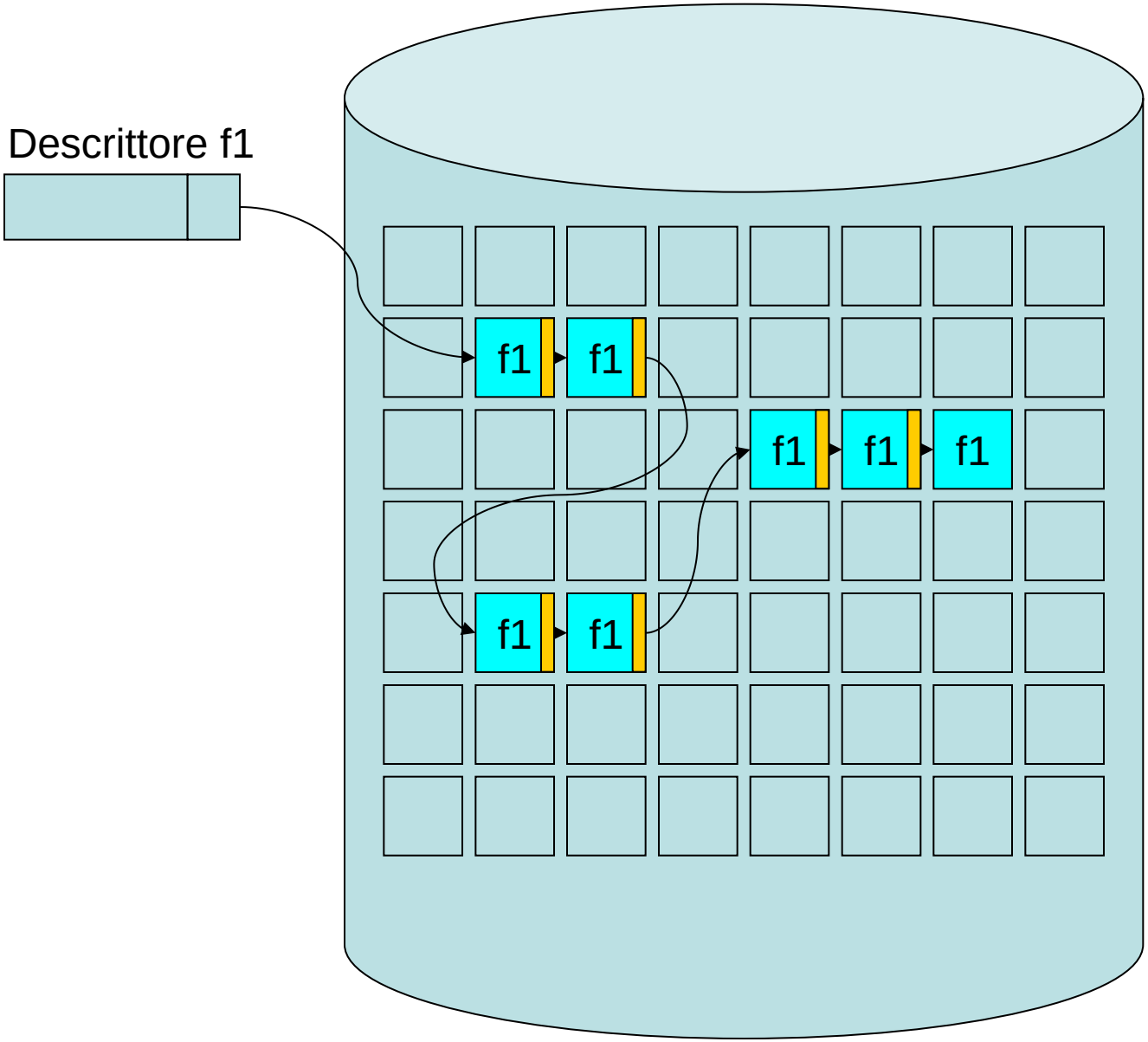
Pietro Frasca

Lezione 18

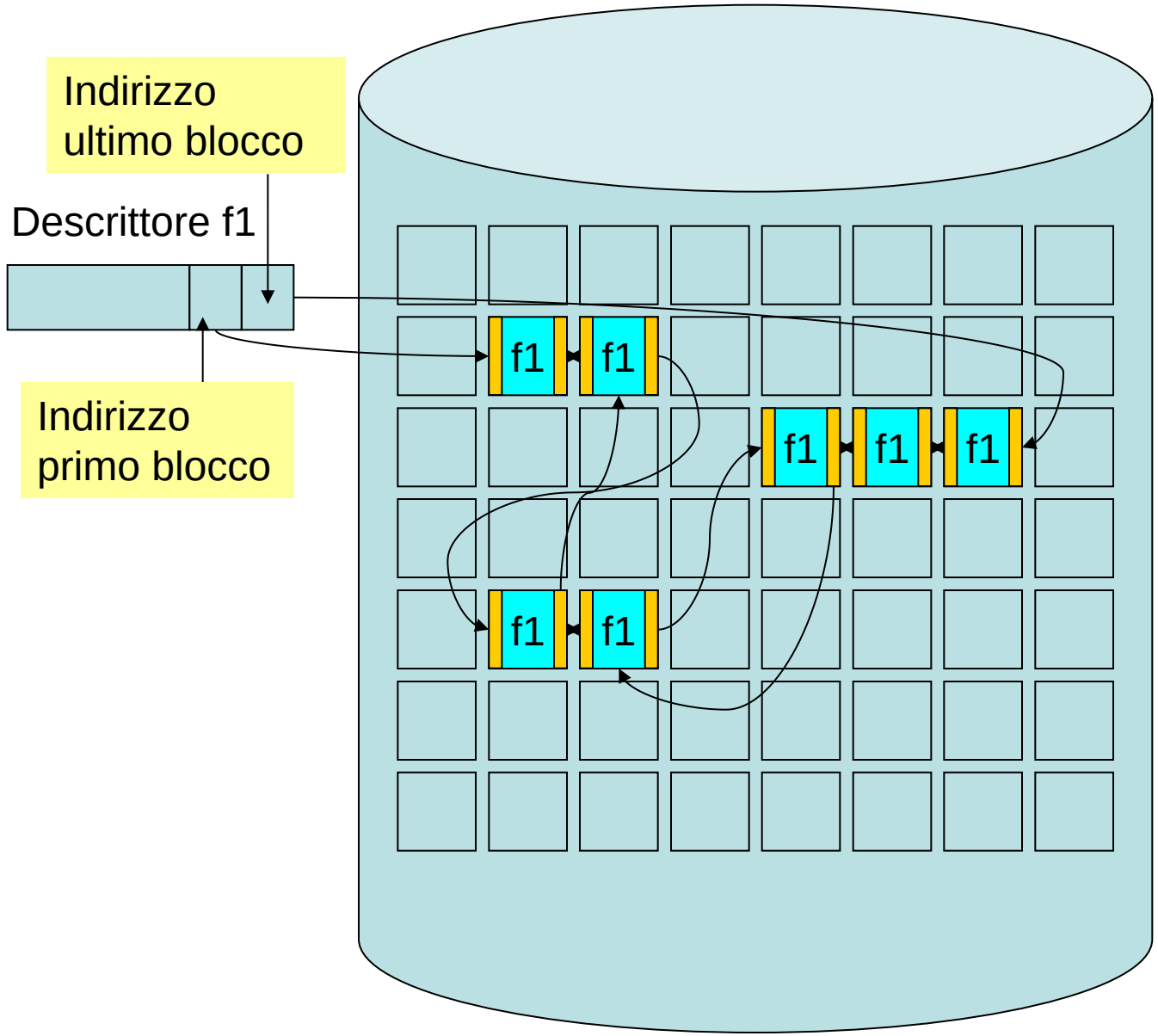
Giovedì 15-12-2016

Allocazione a lista concatenata

- Con questa tecnica il file è memorizzato in un insieme di blocchi non necessariamente adiacenti, organizzati in una lista concatenata.
- Il descrittore del file contiene il puntatore al primo blocco utilizzato. In ogni blocco è memorizzato, oltre ai dati, il puntatore al blocco successivo.
- Il metodo di accesso sequenziale è efficiente, mentre l'accesso diretto è invece lento, in quanto per accedere al blocco nel quale è allocato il record **Ri** sono necessari **i/Nb** (Nb = Db/Dr) accessi prima di arrivare al blocco desiderato.
- Questa tecnica ha il vantaggio di eliminare la frammentazione del disco eliminando il problema della ricerca dei blocchi liberi come nel caso dell'allocazione contigua. D'altra parte, le operazioni di accesso sono più lente, perché i blocchi sono sparsi sul disco.
- Il file system **è poco affidabile** poiché il guasto di un blocco implica la perdita del puntatore al blocco successivo, rendendo impossibile accedere all'intero file.

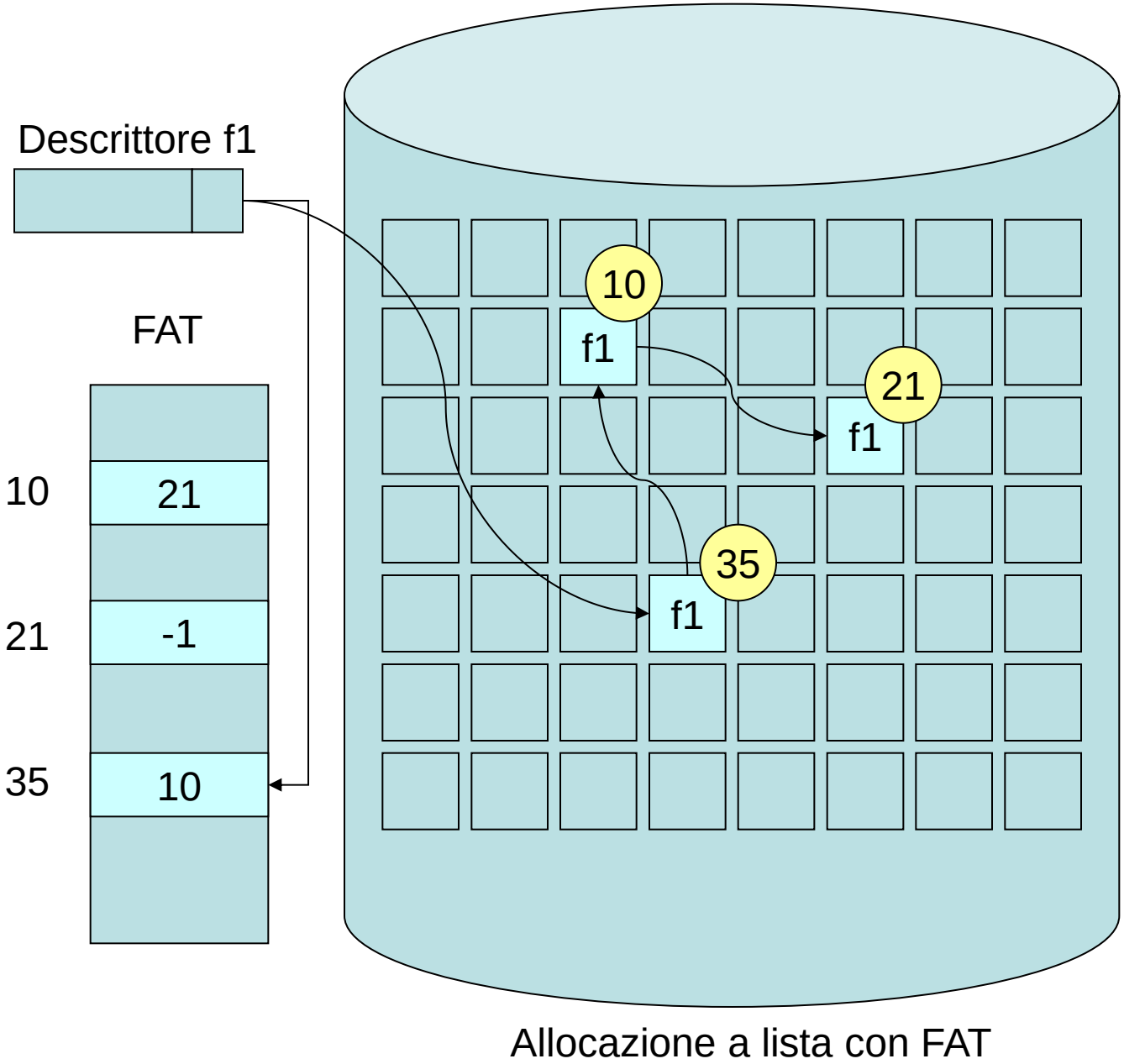


Allocazione a lista concatenata



Allocazione a lista con doppio collegamento

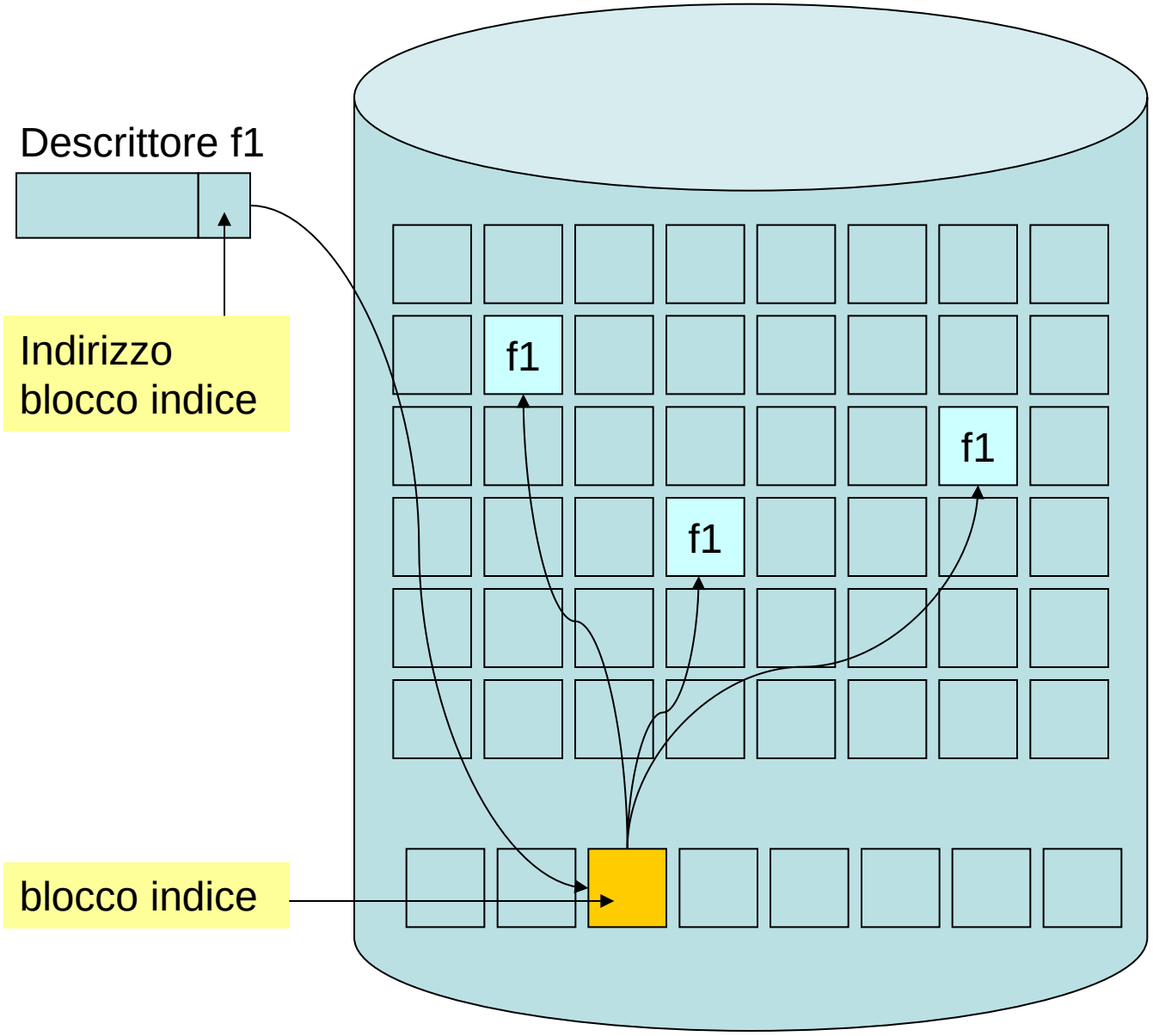
- Un miglioramento della tecnica a lista concatenata si ha utilizzando una struttura dati nella quale è descritta la mappa di allocazione di tutti i blocchi (**Tabella di allocazione dei file – File Allocation Table – FAT**). La FAT è memorizzata in una posizione fissa su disco (spesso è replicata su due zone del disco).
- La FAT contiene un elemento per ogni blocco del disco, il cui valore indica se il blocco è libero oppure contiene l'indice dell'elemento della tabella che rappresenta il blocco successivo nella lista.
- Se un puntatore si perde, si può ripristinare la concatenazione mediante la FAT. Copiando la FAT in memoria o in una cache si **velocizza notevolmente l'accesso diretto**, poiché l'indirizzo del blocco cui accedere può essere recuperato dalla RAM evitando continui accessi al disco.



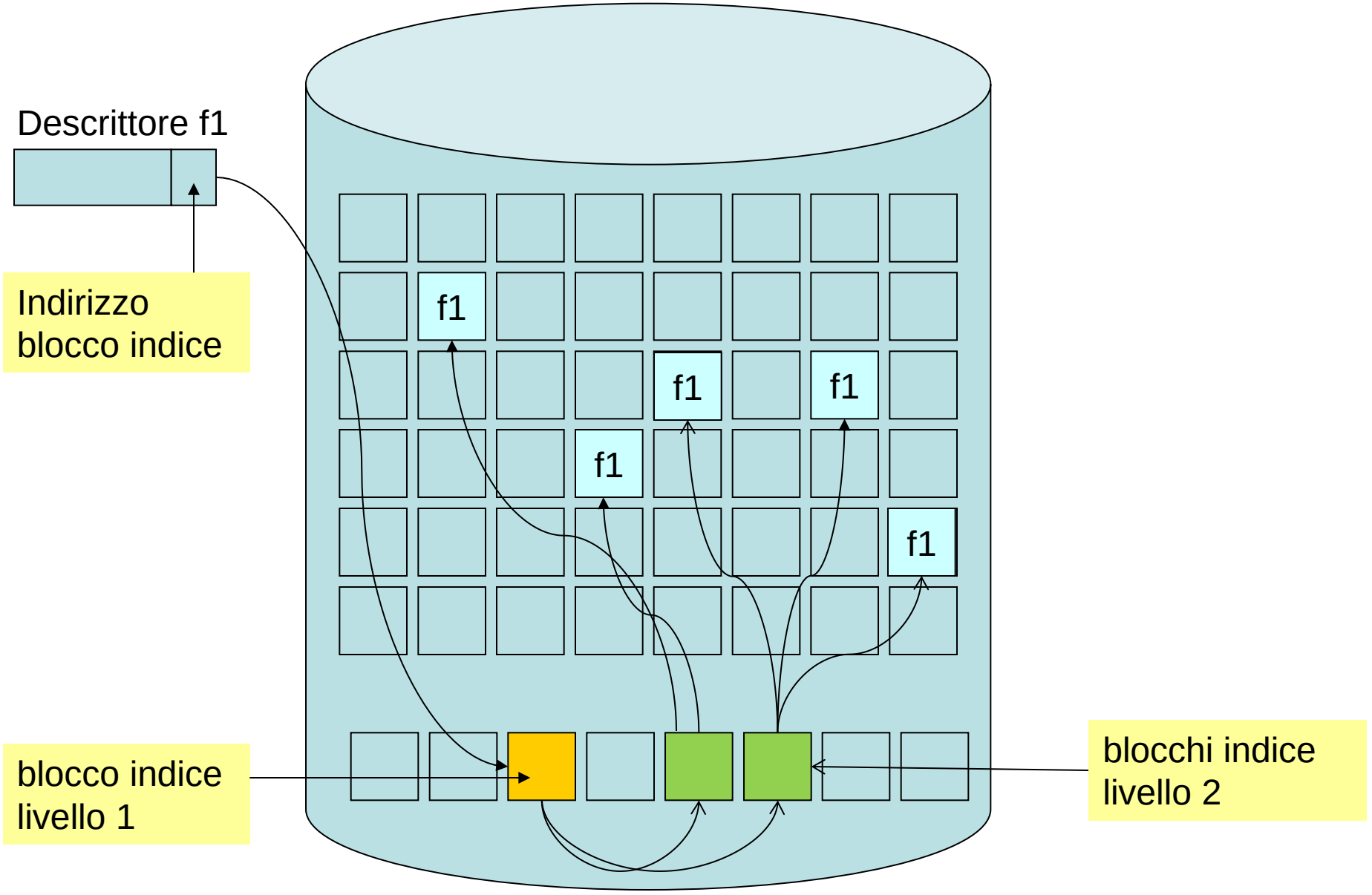
Allocazione a lista con FAT

Allocazione a indice

- Ad ogni file si associa un blocco indice che contiene gli indirizzi dei blocchi utilizzati per l'allocazione del contenuto del file.
- Il descrittore del file contiene l'indirizzo del blocco indice, accedendo al quale si ottengono gli indirizzi degli altri blocchi utilizzati per l'allocazione del file.
- Questo metodo è efficiente sia per l'accesso sequenziale che diretto.
- Uno svantaggio evidente è dato dalla limitata dimensione del blocco indice. Per eliminare questo limite si utilizzano vari livelli di indice. Ad esempio nel caso di indici a 2 livelli, ad ogni file è associato un indice di primo livello che contiene indirizzi di altri blocchi indice (di secondo livello), i quali contengono gli indirizzi dei blocchi utilizzati per l'allocazione dei file.
- Unix, ad esempio, utilizza un metodo a tre livelli di indice.



Allocazione a indice



Allocazione a 2 livelli di indice

Il livello dispositivo virtuale

- Interagisce direttamente con l'hardware e ha il compito di **partizionare il disco** in un insieme di blocchi fisici di dimensione fissa, facendo vedere al livello soprastante il dispositivo come fosse costituito da un vettore lineare di blocchi fisici.

Protezione di file e directory

- Fondamentale requisito per i SO multiutente: consente di stabilire regole di accesso ai file e directory.
- Le soluzioni più comuni per l'accesso ai file si basano sui concetti di ***risorsa*** e ***dominio di protezione.***
- Le **risorse** sono gli oggetti da proteggere (file, directory, dispositivi …).
- I **domini di protezione** sono definiti come un insieme di coppie ***<risorsa, diritti>*** e generalmente sono associati ad un utente. I diritti specificano il tipo di operazione che è possibile eseguire per la risorsa. Ad esempio per i file i diritti specificano se si ha diritto di lettura, scrittura o esecuzione.
- Il sistema operativo mantiene in opportune strutture dati le risorse e i domini. Concettualmente si può pensare ad una **matrice di protezione**.

Utente\risorsa	**file1**	**file2**	**file3**	**dir1**	**stampante1**
Lino	rw	r	rw	rw	w
Mario	r	rwx		r	w
Eva				rw	

Matrice di protezione

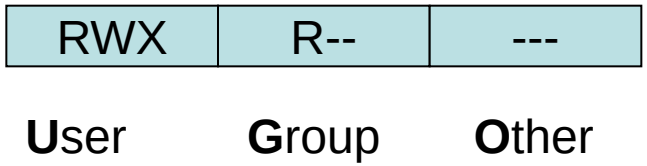
- La matrice non è una struttura dati adatta in pratica per realizzare la corrispondenza tra risorsa e diritti, per via dell'elevato numero di utenti e soprattutto di risorse che sono presenti in un sistema. Esse sarebbero matrici sparse e di enormi dimensioni.
- Le due soluzioni più comuni consistono in:
  - **Liste di controllo degli accessi (Access Control List – ACL)**
  - **Liste di capacità (Capability List o C-list)**

- Una **ACL** esprime la politica di protezione associata ad una risorsa, rappresenta quindi una **colonna della matrice di protezione**: per ogni risorsa **R** si elencano i permessi che ciascun utente possiede per essa. Ad esempio per la **risorsa file2** l'ACL è:

Utente\risorsa	file1	file2	file3	dir1	stampante1
Lino	rw	r	rw	rw	w
Mario	r	rwx		r	w
Eva				rw	

[lino:r,mario:rwx]

- Sia Windows che Unix adottano tecniche di protezione basate su **ACL**.
- Unix usa una tecnica di ACL più semplice poiché per ogni risorsa si specificano tre classi in cui gli utenti sono raggruppati: ***proprietario***, ***gruppo*** e tutti gli ***altri*** utenti. Pertanto, ogni ACL occupa solo 9 bit:



- Una **C-List**, invece corrisponde ad una riga della matrice di protezione. Quindi, il sistema mantiene una lista di permessi relativi alle risorse che il processo P ha diritto ad accedere. In relazione alla matrice di protezione dell'esempio si ha per un processo **P** dell'utente **lino**:

  [file1:rw, file2:r, file3:rwx, dir1:rw, stampante1:w]

Utente\risorsa	**file1**	**file2**	**file3**	**dir1**	**stampante1**
Lino	rw	r	rwx	rw	w
Mario	r	rw		r	w
Eva				rw	

Storia di UNIX

- Nel 1969 primo prototipo sviluppato nei laboratori Bell dell'AT&T.
- L'obiettivo principale era di realizzare un SO multiprogrammato portabile per macchine di media dimensione.
- Nel 1970 viene realizzata la prima versione di UNIX, multitasking e monoutente, sul calcolatore PDP-11, interamente realizzata in assembler.
- Successivamente furono aggiunte varie funzionalità tra cui la multiutenza.
- Nel 1973 venne implementato prevalentemente in linguaggio C (circa il 60%), conferendo al SO una buona portabilità.
- L'uso del linguaggio C per lo sviluppo di UNIX portò ad una grande diffusione del linguaggio.

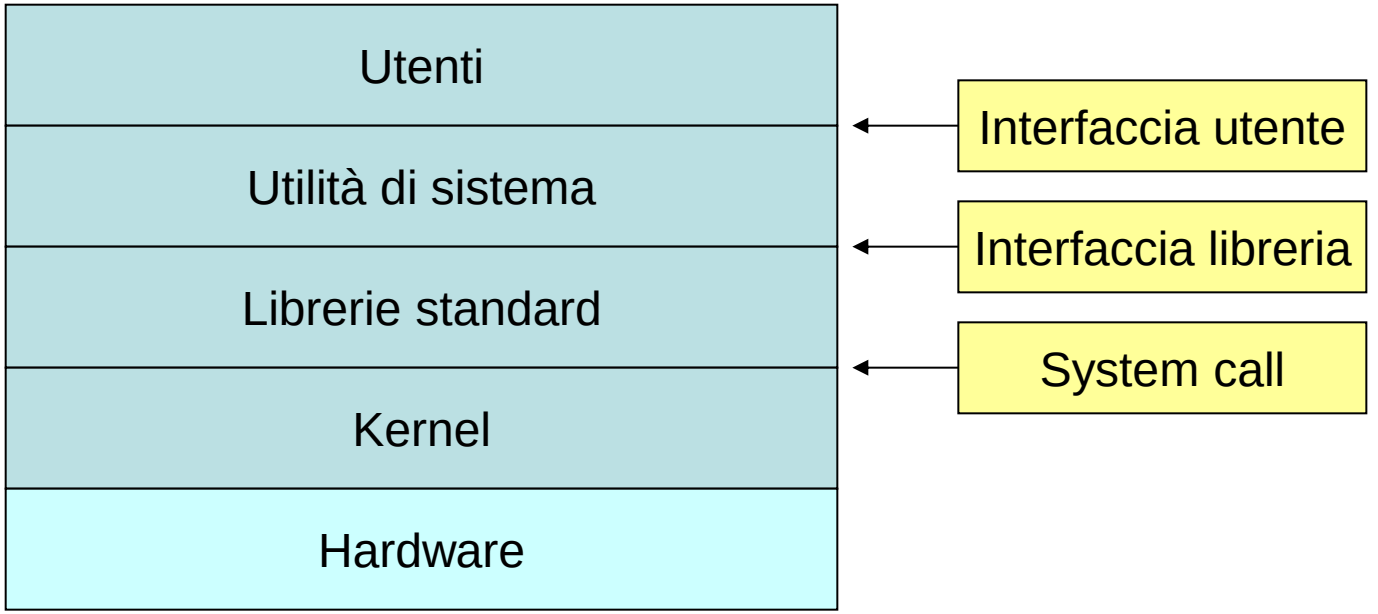
- Negli anni 80 sono state sviluppate molte versioni di UNIX, spesso con sensibili differenze tra loro, che hanno impedito una buona portabilità delle applicazioni.
- In questo periodo si sono affermate:
  - **UNIX system V dell'AT&T**
  - **UNIX Berkeley Software Distribution (BSD) dell'Università di Berkeley in California.**
- Nel 1988 l'IEEE (**I**nstitute of **E**letrical and **E**lectronic **E**ngineers) definisce lo standard **POSIX** (**P**ortable **O**perating **S**ystem **I**nterface UNI**X**) che definisce le API del SO UNIX.
- Nel 1990 POSIX viene riconosciuto dall'ISO (International Standard Organization). POSIX venne successivamente recepito da System V e BSD.

Linux

- Linux è una delle realizzazioni di UNIX attualmente più diffuse, un SO di pubblico dominio, nato per i PC è ora disponibile per molte piattaforme hardware come SUN, PowerPC, Alpha, etc..
- Linux è stato realizzato nel 1991 da Linus Torvalds, uno studente dell'università di Helsinki. Torvald, nella realizzazione di Linux, per alcuni aspetti trasse spunto dal kernel di MINIX ideato da Tanenbaum. A differenza di Tanenbaum, per Linux Torvald scelse un'architettura monolitica, mentre **MINIX** aveva un'architettura a microkernel.
- La libera diffusione del codice sorgente delle prime versioni di Linux consentì a molti programmatori di estendere il sistema.
- Linux attualmente è un ottimo e affidabile SO, molto diffuso ed è concorrente agli attuali SO commerciali.
- Linux attualmente presenta rilevanti differenze da UNIX.

Architettura di UNIX

- UNIX è un SO multiutente, multitasking e time-sharing
- Il sistema di gestione della memoria si basa su paginazione e segmentazione



Architettura di UNIX

- L’architettura è a livelli. Ogni livello presenta una specifica interfaccia.
- nel kernel sono implementate tutte le funzionalità del SO, tra cui la **gestione dei processi**, **della memoria, dell’ I/O** e del **file system**.

Interazione con l’utente

- L’interazione con l’utente avviene mediante:
  - **Shell (interprete di comandi)**
  - **Interfaccia grafica**

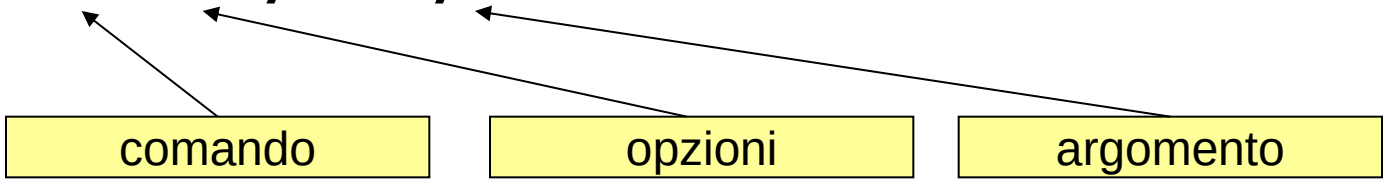
La Shell

- Esistono varie shell per UNIX. La shell originale è **sh** (realizzata da Bourne), che attualmente è ancora molto diffusa. Altre shell molto usate sono la **cshell** e la **bash** (**B**ourne **A**gain **sh**ell) che è la shell predefinita per Linux.
- La shell può essere usata in modo interattivo, digitando i comandi a riga di comando, oppure in modo batch eseguendo script del linguaggio proprio della shell.
- La sintassi di un comando ha il seguente formato:

$ **nome_comando [opzioni][argomenti del comando]**

Esempio:

$ ls –lax /user/lino



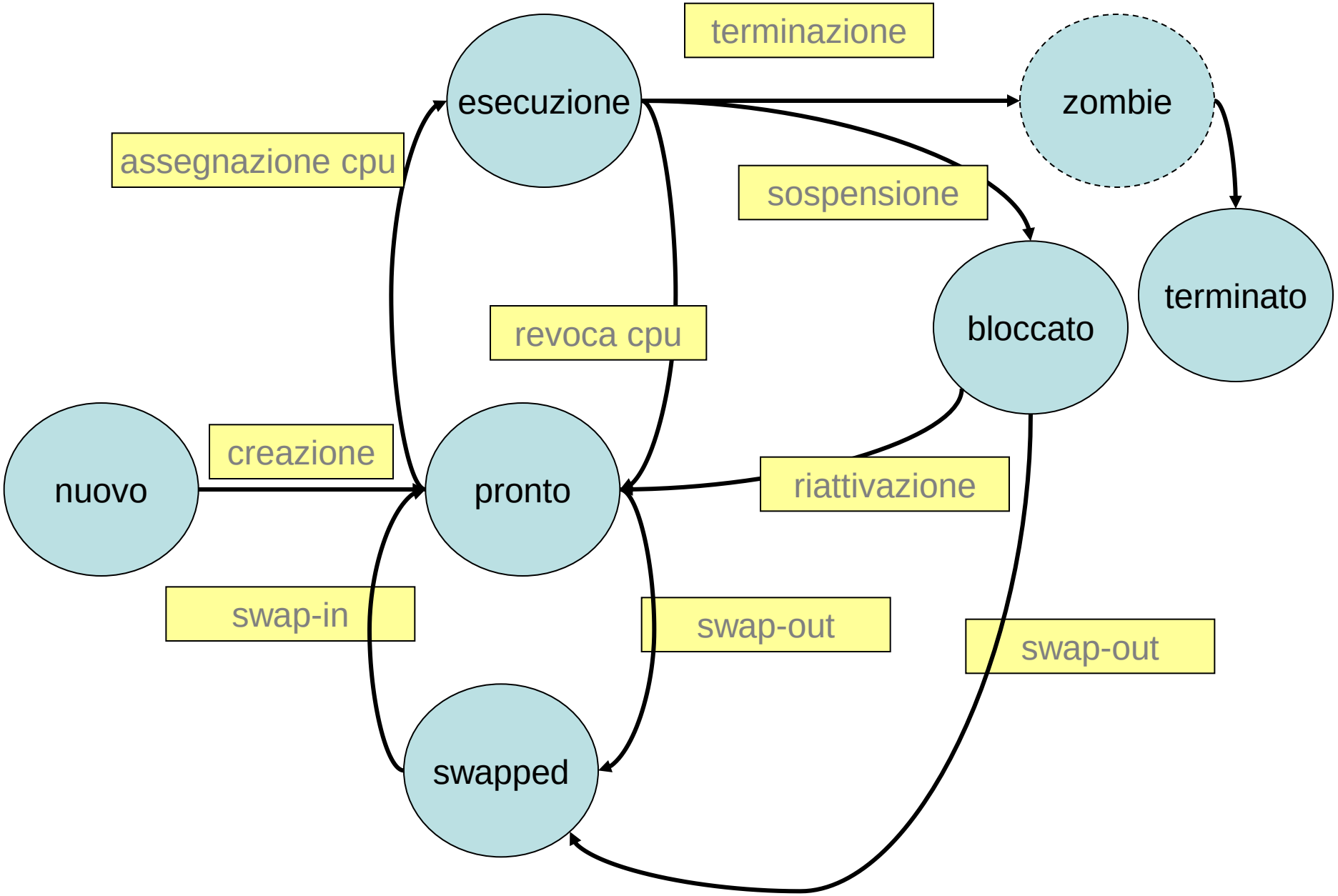
Processi e thread

- Unix è una famiglia di SO multitasking basata sui processi e thread. In origine Unix non supportava il multithreading a livello di kernel. Un processo unix conteneva quindi un solo thread.
- Attualmente molte versioni Unix supportano i thread a livello kernel. Varie librerie, come ad esempio pthread, consentono di realizzare applicazioni multithread in accordo con il livello di thread supportato dal sistema operativo (a livello di utente e/o kernel).
- Il processo unix ha spazi di indirizzamento (segmenti) separati per dati, codice e stack.

- Il segmento dei dati è privato. Questo implica che più processi non possono condividere variabili. La comunicazione tra processi avviene mediante opportune chiamate di sistema che consentono la condivisione di aree memoria tra processi oppure mediante scambio di messaggi.
- Il segmento del codice (text) è condiviso tra più processi (rientranza del codice).

Diagramma degli stati

- in unix la politica di assegnazione della CPU ai processi è basata sulla **divisione di tempo (time-sharing)**.
- Il diagramma degli stati è mostrato in figura:

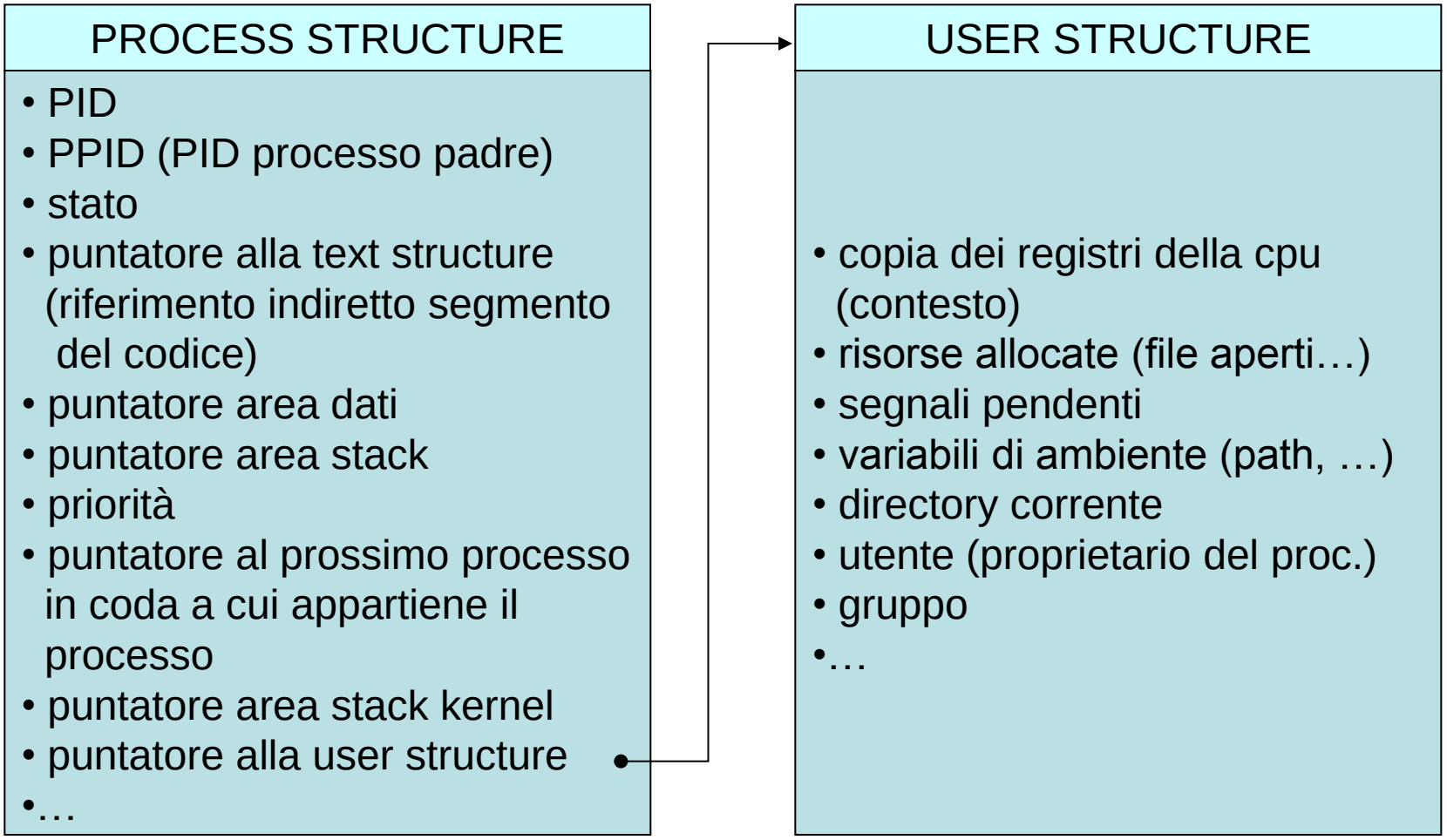

terminazione
esecuzione
zombie
assegnazione cpu
sospensione
terminato
bloccato
revoca cpu
nuovo
creazione
pronto
riattivazione
swap-in
swap-out
swap-out
swapped

- Oltre agli stati già noti, è presente lo stato **zombie**. Un processo entra in questo stato quando è terminato, ma non può essere eliminato dalla tabella dei processi poiché la sua immagine è ancora necessaria.
- Ad esempio, quando un processo figlio termina prima del padre e il padre non ha rilevato il suo stato di terminazione, ad esempio con la wait o la waitpid.
- Il processo figlio uscirà dallo stato zombie quando il padre rileverà tale stato. Se il padre non esegue le operazioni di rilevazione della terminazione, il figlio uscirà dallo stato zombie dopo la terminazione del processo padre.
- In unix è presente uno **scheduler a medio termine** (**swapper**) che periodicamente, se necessario, sposta pagine dalla memoria alla swap-area su disco.

- Le pagine da sostituire sono revocate dapprima a processi bloccati (sleeping), con criteri basati su parametri come **tempo di attesa, tempo di permanenza in memoria e dimensione del processo**. Spesso è usata una politica **basata sulla dimensione del processo** che revoca le pagine a processi che occupano più memoria.
- Per l'operazione di swap-in, lo swapper selezionerà pagine ancora in base a delle politiche proprie, ad esempio selezionerà pagine di **processi che hanno minore dimensione**.

Immagine di un processo unix

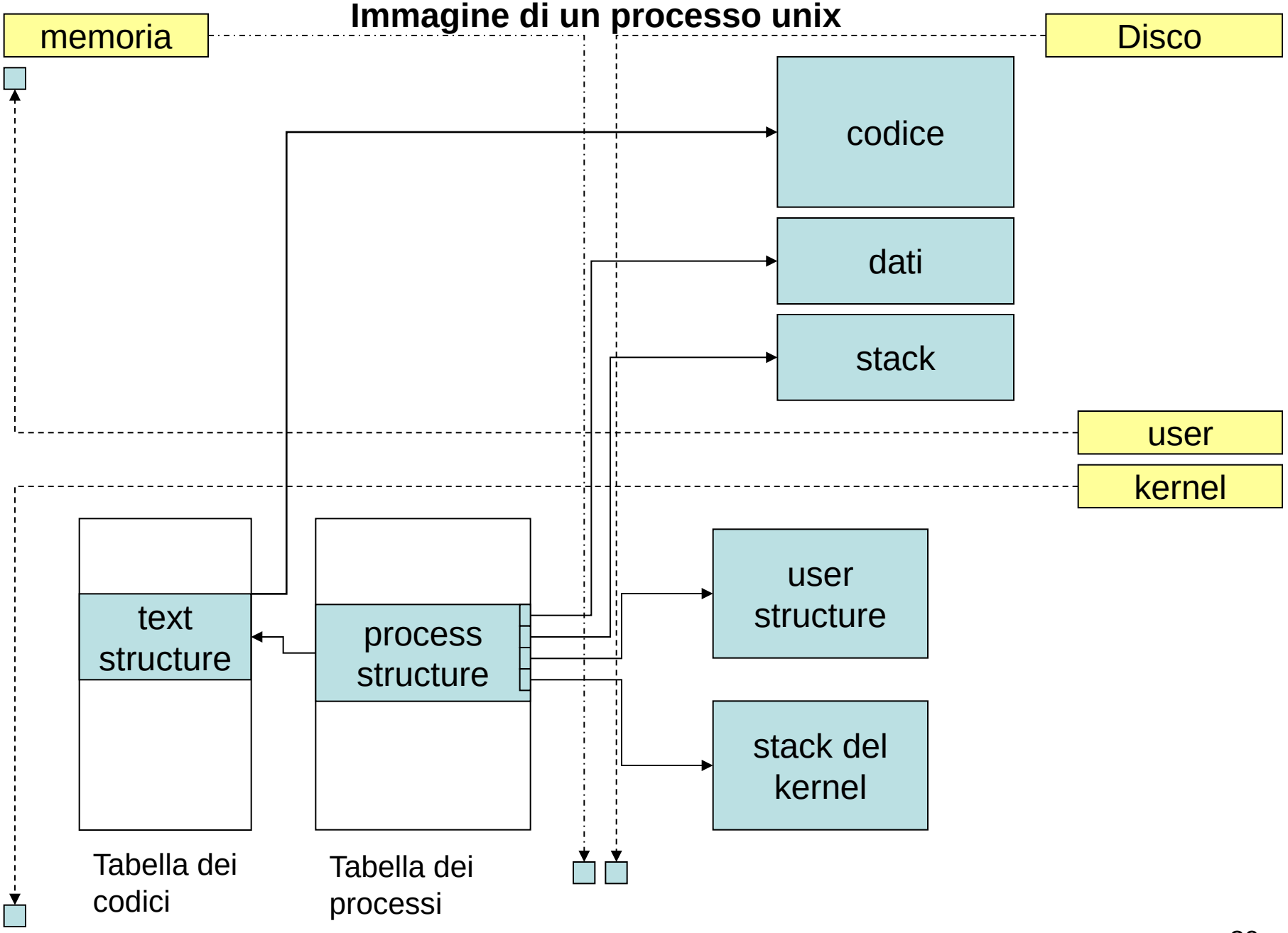
- Il descrittore del processo (**PCB - process control box)** è suddiviso in due strutture dati distinte:
  - **process structure**
  - **user structure**
- la **process structure** contiene dati fondamentali per la gestione del processo sia che esso sia in memoria, nello stato di pronto, o sia nello stato swapped. La **process structure** deve pertanto essere sempre **residente in memoria**.
- La **user structure**, invece, contiene dati necessari al sistema per la gestione del processo, solo nel caso che questo sia in memoria, non si trovi nello stato swapped. Per tale motivo la **user structure** potrà essere ***swappata***.



Descrittore di un processo UNIX

- Il kernel mantiene in memoria **la tabella dei processi** i cui elementi sono costituiti dalle **process structure** di tutti i processi.
- Il kernel gestisce una struttura dati globale, **la tabella dei codici** (**text table**) nella quale ogni elemento contiene un puntatore al segmento del codice del processo (se il codice è stato swappato contiene l'indirizzo relativo all'area su disco). Nella **process structure** è quindi presente un riferimento indiretto al segmento del codice.
- La figura seguente mostra le parti dell'immagine di un processo in base alla visibilità user/kernel e alla possibilità che le varie parti hanno di essere swappate.
- Alcune parti sono ovviamente protette e accedute solo dal kernel.

Immagine di un processo unix



System call per la gestione dei processi

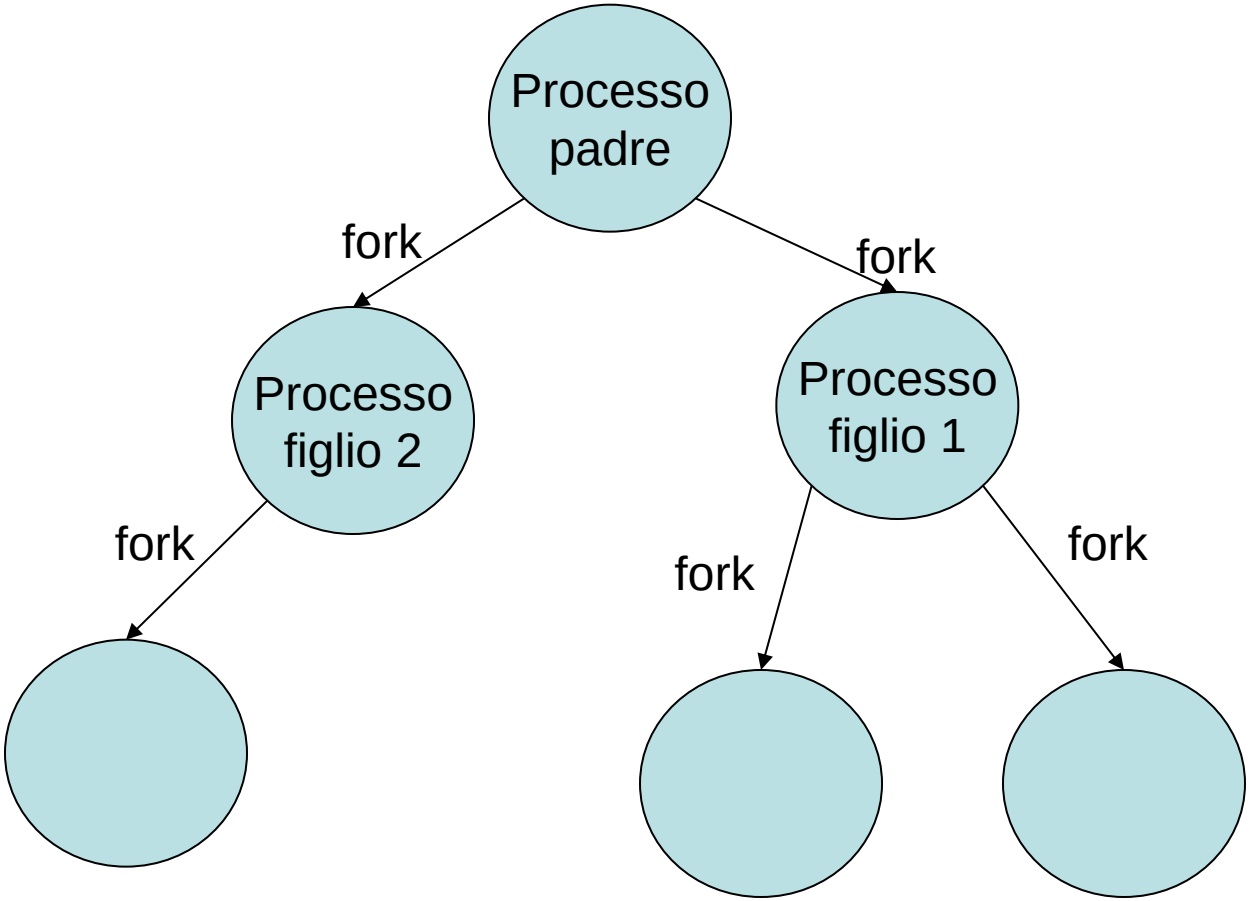
Creazione di processi

- In UNIX un processo può creare un altro processo mediante la chiamata di sistema **fork.**
- Il processo che chiama la funzione fork è detto processo **padre** e il nuovo, processo **figlio**.
- Al processo figlio sono assegnati un segmento dati e un segmento stack privati, mentre condivide il segmento del codice con il processo padre.
- Ogni nuovo processo può creare a sua volta processi figli.
- La fork ha la seguente sintassi:

```
int fork(void);
```

- La fork non ha parametri e restituisce al processo padre, se la creazione è avvenuta con successo, un valore intero che è il PID del processo figlio.

- Il processo figlio condivide il codice con il padre e una **copia** dei segmenti **dati** e **stack** e della user structure (quindi eventuali file aperti, variabili di ambiente, etc.) del padre. Pertanto, ogni variabile del figlio è inizializzata al valore che aveva nel processo padre prima della fork.
- Dato che il processo figlio riceve una copia della user structure del padre, la quale contiene il contesto del processo, compreso quindi il valore del PC, eseguirà la prima istruzione del codice successiva alla fork che lo ha generato.
- La fork viene chiamata una volta, dal padre, ma restituisce due valori diversi al padre e al figlio.
- La fork ritorna il **valore 0** al figlio e un **valore > 0** al padre, che è il **PID** del figlio.
- La fork ritorna -1 in caso di errore.
- In base a questi diversi valori che la fork ritorna è possibile differenziare il comportamento del padre dal figlio.



Gerarchia di processi

```
#include <stdio.h>
main(){
  int pid;
  pid=fork();
  if (pid==0) {
      // codice del figlio
      printf("sono il figlio pid: %d \n",getpid());
  }
  else if (pid > 0) {
      //codice del padre
      printf("sono il padre di pid: %d \n",pid);
      printf("il mio pid è: %d \n",getpid());
  }
  else
      printf("fork fallita");
}
```