

Deadlocks

Deadlocks

- Modello di Sistema
- Caratterizzazione dei deadlock
- Metodi di gestione dei deadlock
- Prevenzione deadlock
- Evitamento deadlock
- Rilevamento deadlock
- Ripristino dai deadlock

Obiettivi

- Descrivere i deadlock, che impediscono ad un insieme di thread di eseguire il loro compito
- Presentare metodi per prevenire, identificare, evitare, recuperare i deadlock

Esempio in Posix

□ Esempio di due thread in deadlock con mutex Posix

```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);

/* thread_one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}
```

Livelock

- Altra forma di fallimento di Liveness
- Un gruppo di thread non bloccato ma non procede
 - Continuo tentativo di eseguire un'azione che fallisce ed impedisce di avanzare
 - ▶ Es. Due persone in un corridoio che non riescono ad evitarsi

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
    int done = 0;

    while (!done) {
        pthread_mutex_lock(&first_mutex);
        if (pthread_mutex_trylock(&second_mutex)) {
            /**
             * Do some work
             */
            pthread_mutex_unlock(&second_mutex);
            pthread_mutex_unlock(&first_mutex);
            done = 1;
        }
        else
            pthread_mutex_unlock(&first_mutex);
    }

    pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
    int done = 0;

    while (!done) {
        pthread_mutex_lock(&second_mutex);
        if (pthread_mutex_trylock(&first_mutex)) {
            /**
             * Do some work
             */
            pthread_mutex_unlock(&first_mutex);
            pthread_mutex_unlock(&second_mutex);
            done = 1;
        }
        else
            pthread_mutex_unlock(&second_mutex);
    }

    pthread_exit(0);
}
```

In alcuni casi si può risolvere con randomizzazione (es. periodo backoff in protocolli di rete)

Caratterizzazione Deadlock

Un deadlock avviene se le seguenti proprietà sono verificate contemporaneamente

- **Mutual exclusion:** almeno una risorsa deve essere tenuta in modalità esclusiva: solo un processo alla volta può usare la risorsa
- **Hold and wait:** almeno un thread deve mantenere almeno una risorsa ed essere in attesa di avere una risorsa aggiuntiva tenuta da altri processi
- **No preemption:** le risorse non possono essere prelazionate - una risorsa può essere rilasciata solo dal processo che la detiene dopo che tale processo ha completato il suo task
- **Circular wait:** esiste un insieme $\{P_0, P_1, \dots, P_n\}$ di processi in attesa mutua, tali che P_0 è in attesa di una risorsa che è tenuta da P_1 , P_1 attende la risorsa di P_2, \dots, P_{n-1} attende la risorsa di P_n , e P_n attenda la risorsa di P_0 .

Le condizioni non sono tutte indipendenti (ultima implica hold and wait), ma è utile considerarle tutte

Modello del Sistema

- I sistemi forniscono risorse
- Tipi di risorse R_1, R_2, \dots, R_m
Cicli CPU, spazio di memoria, dispositivi I/O
- Ogni tipo di risorsa R_i ha W_i istanze.
- Ogni processo utilizza una risorsa come segue:
 - **request**
 - **use**
 - **release**

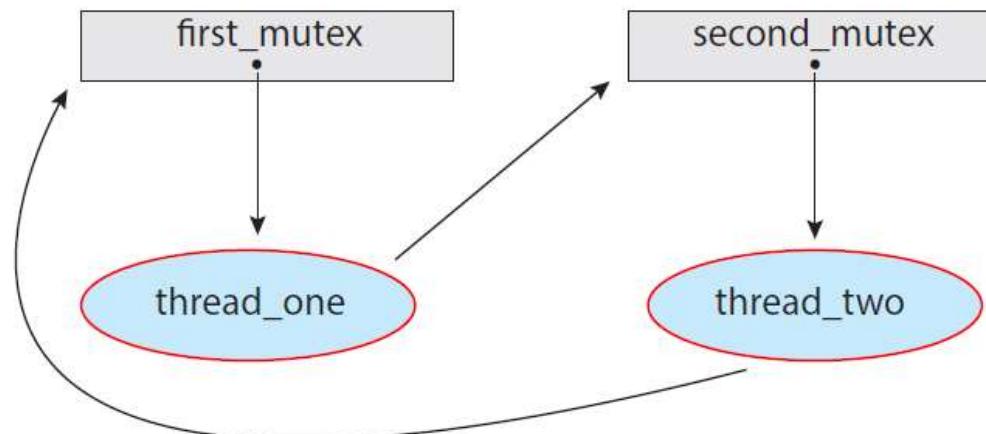
Deadlock con Lock Mutex

- Deadlock possono avvenire con system call, locking, etc.

Grafo di Allocazione delle Risorse

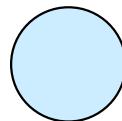
Insieme di nodi V e insieme di archi E .

- V partizionata in due tipi:
 - $P = \{P_1, P_2, \dots, P_n\}$, insieme di tutti i processi nel sistema
 - $R = \{R_1, R_2, \dots, R_m\}$, insieme di tutti i tipi di risorse del sistema
- **request edge** – archi diretti $P_i \rightarrow R_j$
- **assignment edge** – archi diretti $R_j \rightarrow P_i$

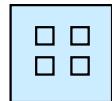


Grafo di Allocazione delle Risorse

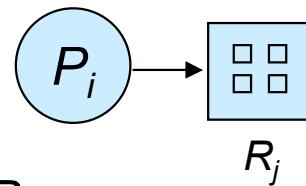
- Processo



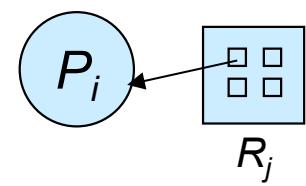
- Tipo di risorsa con 4 istanze



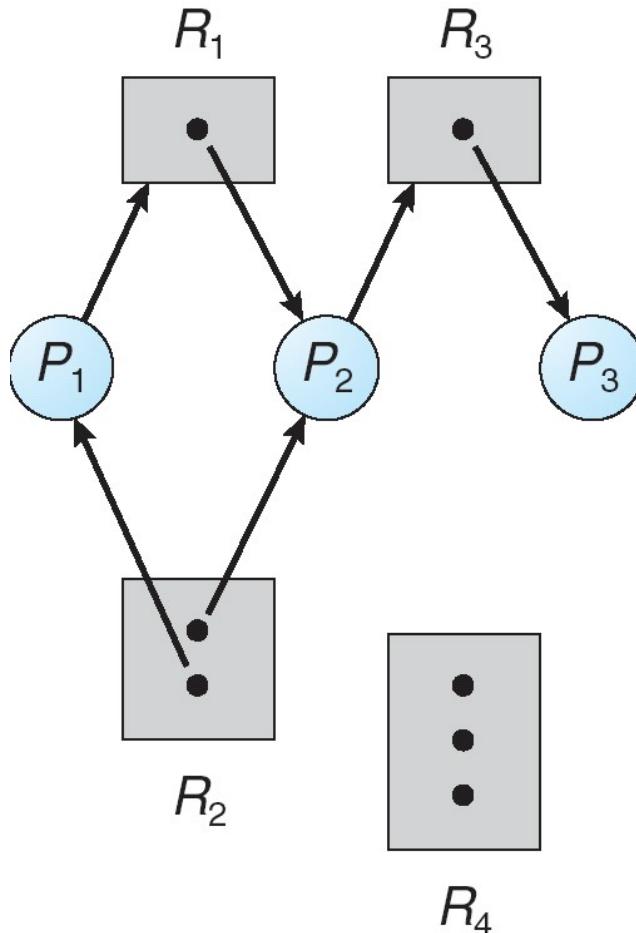
- P_i richiede istanza di R_j



- P_i detiene un'istanza di R_j



Esempio di un Grafo di Allocazione di Risorse



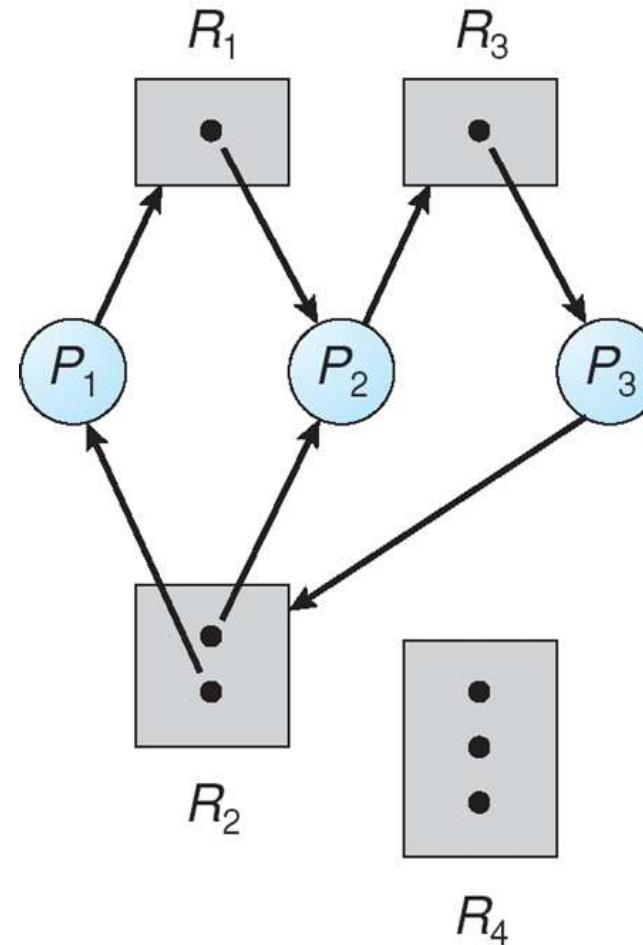
- $T = \{T_1, T_2, T_3\}$
- $R = \{R_1, R_2, R_3, R_4\}$
- $E = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3\}$

Senza cicli non c'è deadlock

Ciclo è condizione necessaria ma non sufficiente

Se risorse uniche allora necessaria e sufficiente

Grafo di Allocazione di Risorse con un Deadlock



Si introduca un ciclo

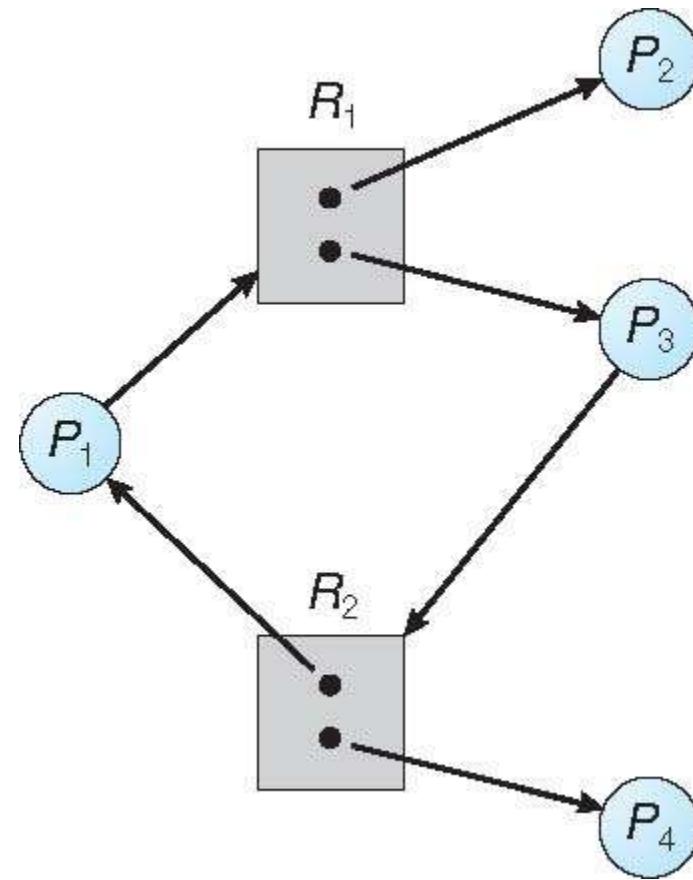
I tre thread sono in deadlock

$$\begin{array}{l} T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1 \\ T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_2 \end{array}$$

Grafo con un ciclo ma senza deadlock

... anche in questo
caso ciclo

Deadlock?

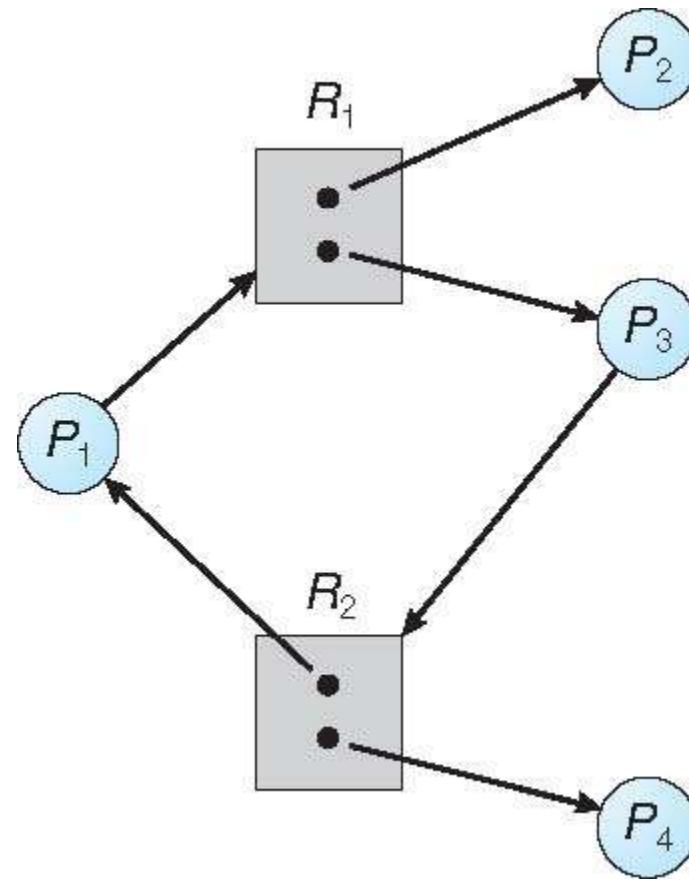


Grafo con un ciclo ma senza deadlock

... anche in questo
caso ciclo

Deadlock?

P₄ può rilasciare R₂
rompendo il ciclo per
consentire a P₃
l'esecuzione



Fatti di Base

- Se il grafo non ha cicli \Rightarrow non ha deadlock
- Se il grafo contiene un ciclo \Rightarrow
 - Se esiste una sola istanza per tipo di risorsa, allora deadlock
 - Se molte istanze per tipo di risorsa, c'è possibilità di un deadlock, ma non si ha necessariamente il deadlock

Metodi di Gestione Deadlocks

Approcci al problema della gestione del deadlock

- Ignorare il problema assumendo che i deadlocks non si presentino mai nel sistema;
 - usato dalla maggior parte dei sistemi operativi, incluso UNIX
- Assicurare che il sistema non entri **mai** in un deadlock state:
 - Prevenzione di deadlock (deadlock prevention)
 - Limitare i modi in cui si fanno le richieste per evitare i deadlock
 - Evitamento del deadlock (deadlock avoidance)
 - Valutare le richieste per evitare situazioni pericolose
- Permettere al sistema di entrare in una stato di deadlock per poi recuperare (deadlock recovery)

Prevenzione Deadlock

Limitare i modi in cui può essere fatta la richiesta

- **Mutual Exclusion** – non richiesta per risorse condivisibili (e.g., read-only files)
 - .. però non si può limitare le richieste a risorse non condivisibili
- **Hold and Wait** – deve garantire che quando un processo richiede una risorsa, non detiene altre risorse
 - Richiedere al processo di richiedere e allocare tutte le sue risorse prima che inizi l'esecuzione
 - Consentire al processo di richiedere risorse solo quando al processo non ne è stata allocata alcuna
 - Non pratico: basso utilizzo delle risorse; possibile starvation

Prevenzione Deadlock

□ Non cosentire prelazione di risorse –

- Se un processo che detiene risorse richiede un'altra risorsa che non può essere immediatamente data allora tutte le risorse devono essere rilasciate
- Le risorse prelazionate sono aggiunte alla lista delle risorse per le quali il processo è in attesa
- Il processo verrà riavviato solo quando può ottenere le vecchie e le nuove risorse
- Oppure si cerano risorse dei processi in attesa
- Può funzionare solo per risorse facilmente recuperabili (CPU, registry, DB, etc.), non per mutex o semafori

□ Attesa circolare –

- imponi un ordine totale a tutti i tipi di risorse e richiedi che ogni processo richieda le risorse in un ordine di enumerazione crescente
- Date le risorse $R = \{R_1, R_2, \dots, R_m\}$ si assegna un numero di ordine $F(R)$
- Un processo può richiedere risorse solo rispettando l'ordine delle risorse
 - ▶ Se nuova risorsa $F(R_j) > F(R_i)$

Esempio Deadlock

```
/* thread one runs in this function */

void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /** * Do some work */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    pthread_exit(0);
}

/* thread two runs in this function */

void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /** * Do some work */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);
    pthread_exit(0);
}
```

$F(\text{first_mutex}) = 1$
 $F(\text{second_mutex}) = 5$

Esempio di Deadlock con Lock Ordering

```
void transaction(Account from, Account to, double amount)
{
    mutex lock1, lock2;
    lock1 = get_lock(from);
    lock2 = get_lock(to);
    acquire(lock1);
    acquire(lock2);
    withdraw(from, amount);
    deposit(to, amount);
    release(lock2);
    release(lock1);
}
```

Transaction 1 e 2 eseguite concorrentemente. Transaction 1 transferisce \$25 da A a B mentre Transaction 2 transferisce \$50 da B ad A

```
transaction(checking_account, savings_account, 25.0)
transaction(savings_account, checking_account, 50.0)
```

Deadlock Avoidance

Il Sistema deve avere informazione *a priori* addizionale:

- Il modello più semplice ed utile richiede ai processi di dichiarare il **massimo numero** di risorse necessarie per tipo
- L'algoritmo di deadlock-avoidance esamina dinamicamente lo stato di allocazione delle risorse per assicurare che non si entri mai in una situazione di circular-wait
- Lo stato di allocazione delle risorse (resource-allocation **state**) è dato dal numero di risorse disponibile e allocate, e dalle richieste massime per processo

Stato Sicuro

- Quando un processo richiede una risorsa disponibile, il sistema deve decidere se l'allocazione immediata lascia il sistema in uno stato sicuro
- Un sistema è in uno **stato sicuro** se esiste una sequenza $\langle P_1, P_2, \dots, P_n \rangle$ di TUTTI i processi nel sistema tali che per ogni P_i , le risorse che P_i può ancora richiedere possono essere soddisfatte dalle risorse correntemente disponibili + le risorse tenute da tutti i P_j , con $j < i$
- Cioè:
 - Se le risorse richieste da P_i non sono immediatamente disponibili, allora P_i può aspettare finché tutti i P_j hanno finito
 - Quando P_j ha finito, P_i può ottenere le risorse richieste, eseguirle, rilasciando le risorse allocate e terminare
 - Quando P_i termina, P_{i+1} può ottenere le risorse necessarie, e così via ...

Stato Sicuro

- Esempio: 12 risorse e 3 thread

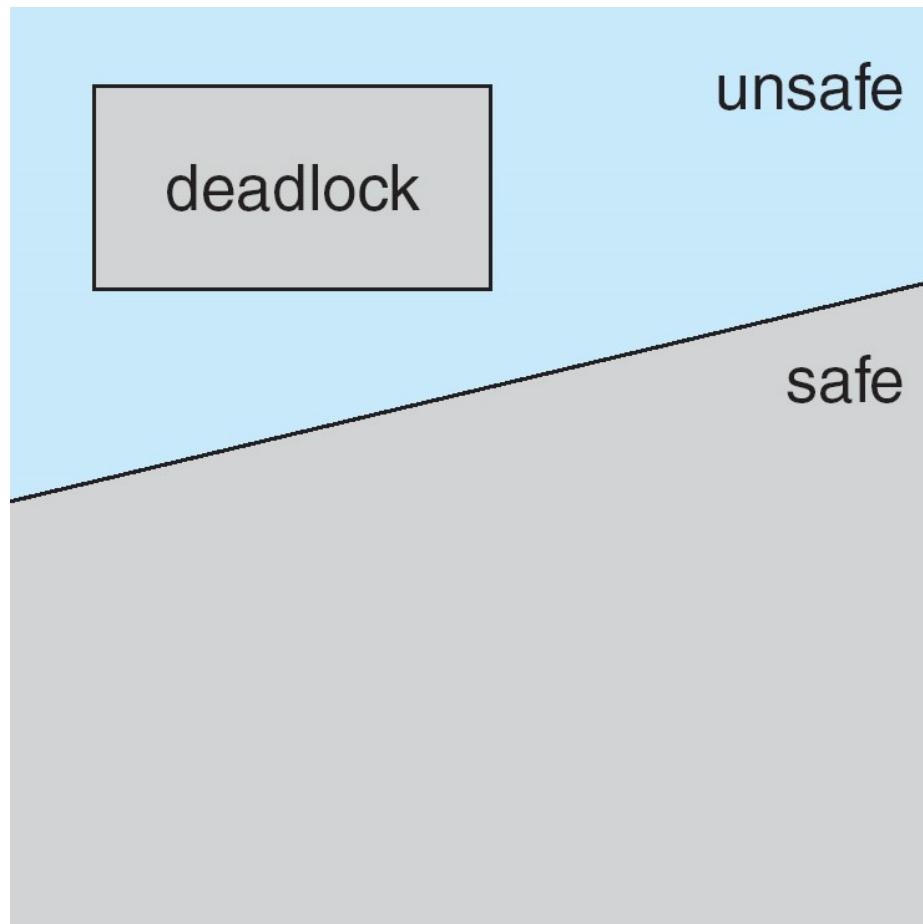
| | <u>Maximum Needs</u> | <u>Current Needs</u> |
|-------|----------------------|----------------------|
| T_0 | 10 | 5 |
| T_1 | 4 | 2 |
| T_2 | 9 | 2 |

- Il Sistema è in stato sicuro
 - Allocate 9 ne rimangono 3
 - La sequenza $\langle T_1, T_0, T_2 \rangle$ soddisfa il requisite
 - ▶ T_1 ne prende 2 e restituisce 4
 - ▶ T_0 ne prende 5 e restituisce 10
 - ▶ T_2 ne prende 7 e finisce
 - Se invece si alloca a T_2 un'ulteriore risorsa al tempo t_1 lo stato non è più safe

Fatti di Base

- Se un sistema è in stato sicuro \Rightarrow non deadlock
- Se un sistema è in stato non sicuro \Rightarrow possibilità di deadlock
- Evitamento \Rightarrow assicurare che un sistema non entri mai in uno stato non sicuro

Safe, Unsafe, Deadlock State



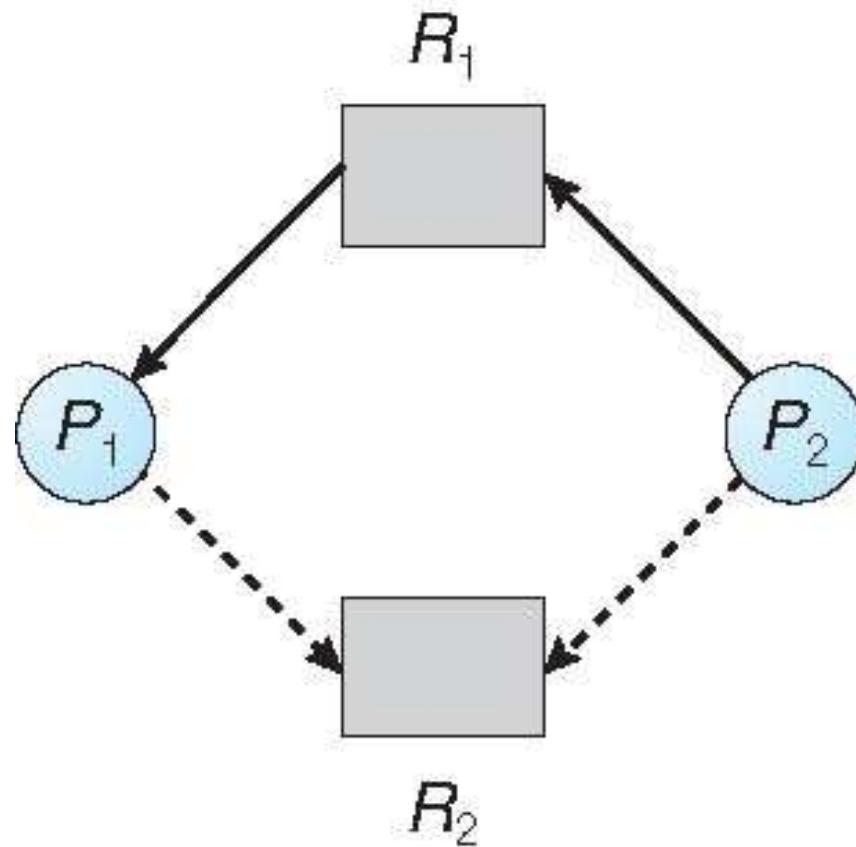
Algoritmo di Avoidance

- Singola istanza di un tipo di risorsa
 - Usa un resource-allocation graph
- Multiple istanze di tipo di risorsa
 - Usa l'algoritmo del banchiere

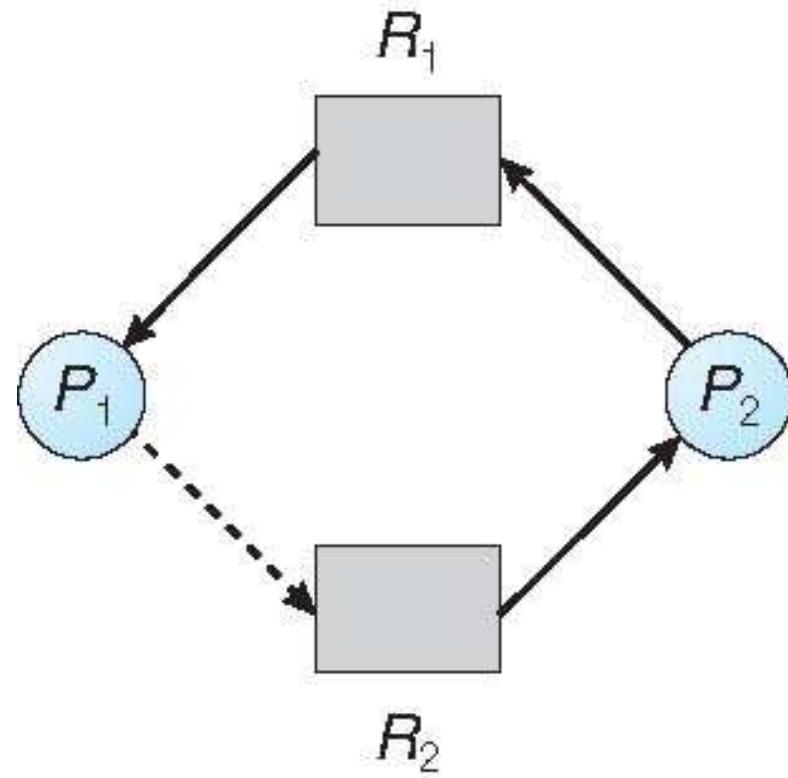
Schema con grafo di allocazione

- Si introduce la **claim edge** $P_i \rightarrow R_j$ che indica che il processo P_i potrebbe richiedere la risorsa R_j ; (linea tratteggiata)
- La claim edge si converte in request edge quando un processo richiede la risorsa
- La request edge si converte in assignment edge quando la risorsa è allocata al processo
- Quando la resource è rilasciata dal processo, l'assignment edge si riconverte in claim edge
- Le risorse devono essere richieste *a priori* nel sistema

Resource-Allocation Graph



Stato unsafe in Resource-Allocation Graph



Algoritmo di Resource-Allocation Graph

- Supponi che il processo P_i richieda una risorsa R_j
- La richiesta può essere garantita solo se trasformando l'arco di richiesta in un assegnamento non porta alla formazione di un ciclo nel grafo di allocazione delle risorse
- Altrimenti la richiesta viene messa in attesa

- Complessità di rilevare un ciclo è n^2 con n numero di thread del sistema

Algoritmo del Banchiere

- Multiple istanze
- Ogni processo deve dichiarare a priori il massimo utilizzo di risorse
- Quando un processo richiede una risorsa potrebbe dover aspettare
- Quando un processo ottiene tutte le sue risorse deve restituirle in un tempo finito

Algoritmo del Banchiere

- Quando un processo richiede un insieme di risorse, il Sistema deve controllare se l'assegnazione lascia il Sistema in uno stato sicuro
- Se lo stato è sicuro le risorse sono allocate altrimenti la richiesta viene messa in attesa finché qualche processo libera delle risorse aggiuntive
- Si introducono diverse strutture dati per rappresentare lo stato del sistema

Strutture Dati per l'Algoritmo del Banchiere

Sia n = numero di processi ed m = numero di tipi di risorse.

- **Available:** Vettore di lunghezza m . Se $\text{available}[j] = k$, ci sono k istanze della risorsa di tipo R_j disponibili
- **Max:** matrice $n \times m$. Se $\text{Max}[i,j] = k$, allora il processo P_i potrebbe richiedere al massimo k istanze della risorsa di tipo R_j
- **Allocation:** matrice $n \times m$. Se $\text{Allocation}[i,j] = k$ allora P_i ha correntemente allocate k istanze di R_j
- **Need:** matrice $n \times m$. Se $\text{Need}[i,j] = k$, allora P_i potrebbe aver bisogno di k più istanze di R_j per compiere il suo task

$$\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$$

Algoritmo di Safety

1. Siano **Work** e **Finish** vettori di lunghezza m ed n , rispettivamente.
Inizializza:

Work = Available

Finish [i] = false for $i = 0, 1, \dots, n-1$

2. Trova un indice i tale che entrambi:
 - (a) **Finish [i] = false**
 - (b) **Need_i ≤ Work**

Se non esiste i , vai al passo 4
3. **Work = Work + Allocation_i**,
Finish[i] = true
vai al passo 2
4. Se **Finish [i] == true** per tutti gli i , allora il sistema è in uno stato sicuro (safe state)

L'algoritmo verifica se lo stato è sicuro e può richiedere un ordine di $m \times n^2$ operazioni

Algoritmo di Richiesta di Risorsa per il Processo P_i

Algoritmo per determinare se una richiesta può essere accordata in sicurezza.

Request_i = vettore richieste per il processo P_i .

Se $\text{Request}_i[j] = k$ allora il processo P_i vuole k istanze delle risorse di tipo R_j

1. Se $\text{Request}_i \leq \text{Need}_i$ vai al passo 2. Altrimenti, segnala la condizione di errore dal momento che il processo ha superato il suo massimo dichiarato
2. Se $\text{Request}_i \leq \text{Available}$ vai al passo 3. Altrimenti P_i deve aspettare perché le risorse non sono disponibili
3. Prova ad allocare le risorse richieste per P_i , modificando lo stato come segue:

$$\text{Available} = \text{Available} - \text{Request}_i;$$

$$\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i;$$

$$\text{Need}_i = \text{Need}_i - \text{Request}_i;$$

- Verifica se safe
- Se safe \Rightarrow le risorse sono allocate a P_i ,
- Se unsafe $\Rightarrow P_i$ deve aspettare e il vecchio stato di allocazione delle risorse viene recuperato

Esempio Algoritmo Banchiere

- 5 processi da P_0 a P_4 ;
3 tipi di risorse:
 A (10 istanze), B (5 istanze) e C (7 istanze)
- Stato al tempo T_0 :

| | <u>Allocation</u> | | | <u>Max</u> | | | <u>Available</u> | | |
|-------|-------------------|----------|----------|------------|----------|----------|------------------|----------|----------|
| | <i>A</i> | <i>B</i> | <i>C</i> | <i>A</i> | <i>B</i> | <i>C</i> | <i>A</i> | <i>B</i> | <i>C</i> |
| P_0 | 0 | 1 | 0 | 7 | 5 | 3 | 3 | 3 | 2 |
| P_1 | 2 | 0 | 0 | 3 | 2 | 2 | | | |
| P_2 | 3 | 0 | 2 | 9 | 0 | 2 | | | |
| P_3 | 2 | 1 | 1 | 2 | 2 | 2 | | | |
| P_4 | 0 | 0 | 2 | 4 | 3 | 3 | | | |

Esempio

- Il contenuto della matrice **Need** è definito come **Max – Allocation**

| | <u>Need</u> | | |
|-------|-------------|---|---|
| | A | B | C |
| P_0 | 7 | 4 | 3 |
| P_1 | 1 | 2 | 2 |
| P_2 | 6 | 0 | 0 |
| P_3 | 0 | 1 | 1 |
| P_4 | 4 | 3 | 1 |

- Il Sistema è in un stato sicuro (safe state) perché la sequenza $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ soddisfa i criteri di safety

Esempio: P_1 Richiede (1,0,2)

- Controlla che Request \leq Available (cioè, $(1,0,2) \leq (3,3,2)$ \Rightarrow true)

| | <u>Allocation</u> | | | <u>Need</u> | | | <u>Available</u> | | |
|-------|-------------------|----------|----------|-------------|----------|----------|------------------|----------|----------|
| | <i>A</i> | <i>B</i> | <i>C</i> | <i>A</i> | <i>B</i> | <i>C</i> | <i>A</i> | <i>B</i> | <i>C</i> |
| P_0 | 0 | 1 | 0 | 7 | 4 | 3 | 2 | 3 | 0 |
| P_1 | 3 | 0 | 2 | 0 | 2 | 0 | | | |
| P_2 | 3 | 0 | 2 | 6 | 0 | 0 | | | |
| P_3 | 2 | 1 | 1 | 0 | 1 | 1 | | | |
| P_4 | 0 | 0 | 2 | 4 | 3 | 1 | | | |

- Eseguendo l'algoritmo di safety si vede che la sequenza $< P_1, P_3, P_4, P_0, P_2 >$ soddisfa i criteri
- Può la richiesta per $(3,3,0)$ di P_4 essere garantita?
- Può la richiesta per $(0,2,0)$ di P_0 essere grantita?

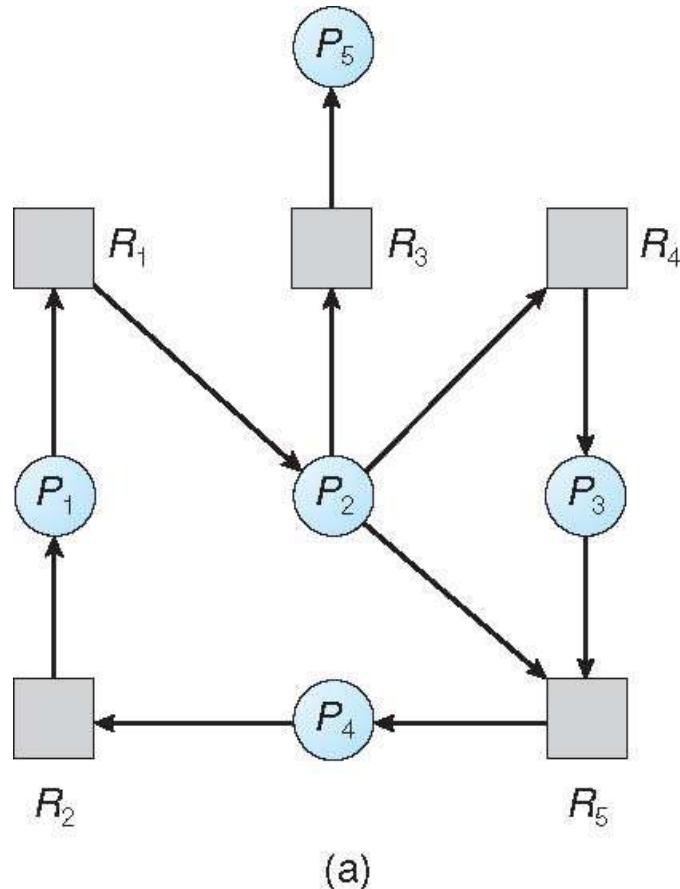
Rilevazione Deadlock

- Permetti al sistema di entrare nello stato di deadlock
- Servono quindi:
 - Algoritmi di deadlock detection
 - Metodi di deadlock recovery
- Costi:
 - Costo del monitoraggio (strutture dati ed algoritmi)
 - Potenziale perdita di dati durante il recovery
- Consideriamo due casi:
 - Istanza singola di risorsa
 - Istanze multiple di risorsa

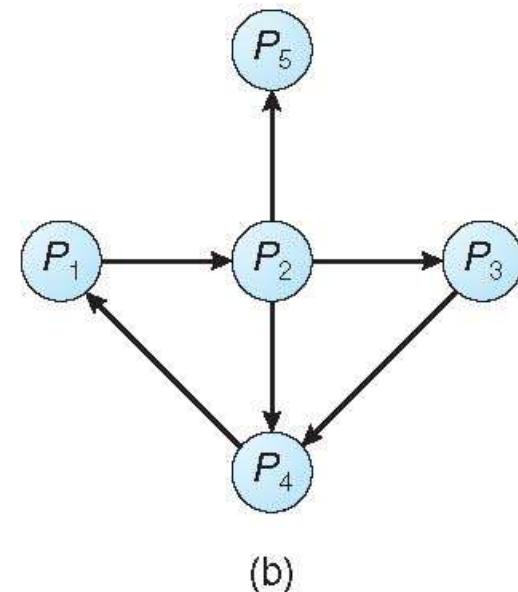
Istanza Singola di Ogni Tipo di Risorsa

- Si usa una variante del grafo di allocazione chiamato grafo **wait-for** (grafo delle attese)
 - Si rimuovono le risorse e si bypassano con archi tra processi
 - Nodi sono solo processi
 - $P_i \rightarrow P_j$ se P_i sta aspettando P_j

Grafo di Allocazione delle e Grafo delle Attese



Grafo di allocazione delle risorse



Grafo delle Attese corrispondente

Istanza Singola di Ogni Tipo di Risorsa

- Si usa una variante del grafo di allocazione chiamato grafo **wait-for** (grafo delle attese)
 - Si rimuovono le risorse e si bypassano con archi tra processi
 - Nodi sono solo processi
 - $P_i \rightarrow P_j$ se P_i sta aspettando P_j
- Periodicamente invoca un algoritmo che cerca i cicli nel grafo. Se c'è ciclo allora c'è un deadlock
- Un algoritmo per rilevare un ciclo in un grafo richiede un ordine di n^2 operazioni, dove n è il numero di vertici nel grafo
- Esempio
 - BCC toolkit su Linux mette a disposizione un `deadlock_detector` che traccia l'uso dei `pthread_mutex_lock` e `pthread_mutex_unlock` costruendo un grafo delle attese e segnalando i deadlock

Molte Istanze di un Tipo di Risorsa

Nel caso di più istanze occorrono più strutture dati come per l'algoritmo del banchiere

- **Available:** un vettore di lunghezza m che indica il numero di risorse disponibili per ogni tipo di risorsa
- **Allocation:** una matrice $n \times m$ definisce il numero di risorse per tipo correntemente allocato ad ogni processo
- **Request:** una matrice $n \times m$ indica la richiesta corrente di ogni processo. Se $\text{Request}[i][j] = k$, allora il processo P_i sta richiedendo k più istanze di risorsa di tipo R_j .

Algoritmo di Rilevamento

1. Siano **Work** e **Finish** vettori di lunghezza m ed n

Inizializza:

- (a) **Work = Available**
- (b) For $i = 1, 2, \dots, n$, if $\text{Allocation}_i \neq 0$, then
Finish[i] = false; otherwise, **Finish[i] = true**

2. Trova un indice i tale che entrambi:

- (a) **Finish[i] == false**
- (b) **Request $_i \leq Work$**

Se non esiste tale i , vai al passo 4

Algoritmo di Rilevamento

3. **$Work = Work + Allocation$,
 $Finish[i] = true$**
vai al passo 2
4. Se $Finish[i] == \text{false}$, per qualche i , $1 \leq i \leq n$, allora il sistema è in uno stato di deadlock. Inoltre, se $Finish[i] == \text{false}$, allora P_i è in deadlocked

Rilasciate le risorse dopo il completamento, si assume che non siano necessarie altre risorse da allocare

Algoritmo richiede un ordine di $O(m \times n^2)$ operazioni per rilevare se il sistema è in stato di deadlock

Esempio di Algoritmo di Rilevamento

- Cinque processi da P_0 a P_4 ; tre tipi di risorsa A (7 istanze), B (2 istanze), and C (6 istanze)
- Stato al tempo T_0 :

| | <u>Allocation</u> | | | <u>Request</u> | | | <u>Available</u> | | |
|-------|-------------------|---|---|----------------|---|---|------------------|---|---|
| | A | B | C | A | B | C | A | B | C |
| P_0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| P_1 | 2 | 0 | 0 | 2 | 0 | 2 | | | |
| P_2 | 3 | 0 | 3 | 0 | 0 | 0 | | | |
| P_3 | 2 | 1 | 1 | 1 | 0 | 0 | | | |
| P_4 | 0 | 0 | 2 | 0 | 0 | 2 | | | |

- La sequenza $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ porterà a $Finish[i] = \text{true}$ per tutti le i

Esempio

- P_2 richiede un'istanza addizionale di tipo **C**

Request

| | A | B | C |
|-------|---|---|---|
| P_0 | 0 | 0 | 0 |
| P_1 | 2 | 0 | 2 |
| P_2 | 0 | 0 | 1 |
| P_3 | 1 | 0 | 0 |
| P_4 | 0 | 0 | 2 |

- Stato del sistema?
 - Può rilasciare le risorse tenute dal processo P_0 , ma le risorse sono insufficienti per soddisfare le richieste degli altri processi
 - Esiste un deadlock per i processi P_1 , P_2 , P_3 , e P_4

Algoritmo Rilevamento Uso

- Quanto spesso invocare il rilevamento?
- Dipende da:
 - Quanto spesso un deadlock è probabile che avvenga?
 - Se frequente va invocato frequentemente
 - Può coinvolgere più processi
 - Nel caso estremo può essere invocato ogni volta che una richiesta non può essere soddisfatta
 - In questo caso si può identificare anche chi ha causato il deadlock
 - Però dispendioso, quindi check ad intervalli fissi
 - Quanti processi sarà necessario recuperare?
 - uno per ogni ciclo disgiunto
- Se l'algoritmo di detection invocato in modo arbitrario
 - possibili molti cicli nel grafo delle risorse
 - complicato risalire a quali dei tanti processi abbia “causato” il deadlock.

Ripristino dal Deadlock

- Quando viene trovato un deadlock va gestito:
 - Avvertire operatore che gestisce a mano
 - Gestione automatica
- Due modalità:
 - Terminare processi
 - Aggiungere risorse da processi nel deadlock

Ripristino dal Deadlock: Terminazione di Processo

- Terminazione
 - Abort di tutti i processi in deadlock
 - ▶ Molto dispendioso tutta la computazione persa
 - Abort dei processi, uno alla volta finché il ciclo di deadlock cycle è eliminato
 - ▶ Invocato il deadlock check ad ogni passo (overhead)
 - In quale ordine dovremmo sceglierli per fare l'abort?
 - ▶ Si dovrebbe scegliere il costo minimo, ma diversi criteri:
 1. Priorità del processo
 2. Per quanto tempo ha lavorato e quanto manca al completamento
 3. Quante e quali risorse il processo ha già usato (si possono prelazionare?)
 4. Quante risorse il processo necessita per completare
 5. Quanti processi necessiteranno di essere terminati
 6. È un processo interattivo o batch

Ripristino dal Deadlock: Prelazione di Risorse

- Si selezionano risorse da prelazionare finché il deadlock non è risolto
- Per la selezione della risorsa occorre:
 - **Selezionare una vittima** – quale risorsa e quale processo prelazionare? Minimizza il costo: il numero di risorse trattenute, il tempo di computazione impiegato, etc.
 - **Rollback** – se prelazionato un processo che farne? Deve ritornare in qualche safe state e ripartire da quello stato. Metodo più semplice ripartire da capo. Altrimenti da stato intermedio di computazione, però va mantenuto.
 - **Starvation** – come evitare lo starvation: se c'è una cost function per la scelta, lo stesso processo può sempre essere scelto come vittima. Per evitarlo si può includere il numero dei rollback nei fattori di costo

Riassunto

- Definizione di deadlock
- Condizioni per avere un deadlock
- Grafi di allocazione delle risorse e cicli
- Prevenzione dei deadlock in base alle condizioni
- Metodi di evitamento del deadlock per risorse singole (grafo delle allocazioni) e multiple (algoritmo del banchiere)
- Metodi di deadlock detection per risorse singole e multiple
- Metodi di deadlock recovery

File System: Dettagli

Obiettivi

- Approfondire i dettagli dei file system e la loro implementazione.
- Analizzare la procedura di avvio e la condivisione dei file.
- Descrivere i file system remoti, usando NFS come esempio

Dettagli File System

- Dettagli File System
- Montare un File-System
- Partizioni e Mounting
- File Sharing
- Virtual File System
- Remote File System
- Consistency Semantic
- NFS

File e Partizione

- A seconda del gestore di volumi, un volume può estendersi su più partizioni
- Un SO può avere più file system
 - Per esempio Solaris ha una dozzina di FS
 - Oltre ai FS general purpose, ci sono quelli specializzati

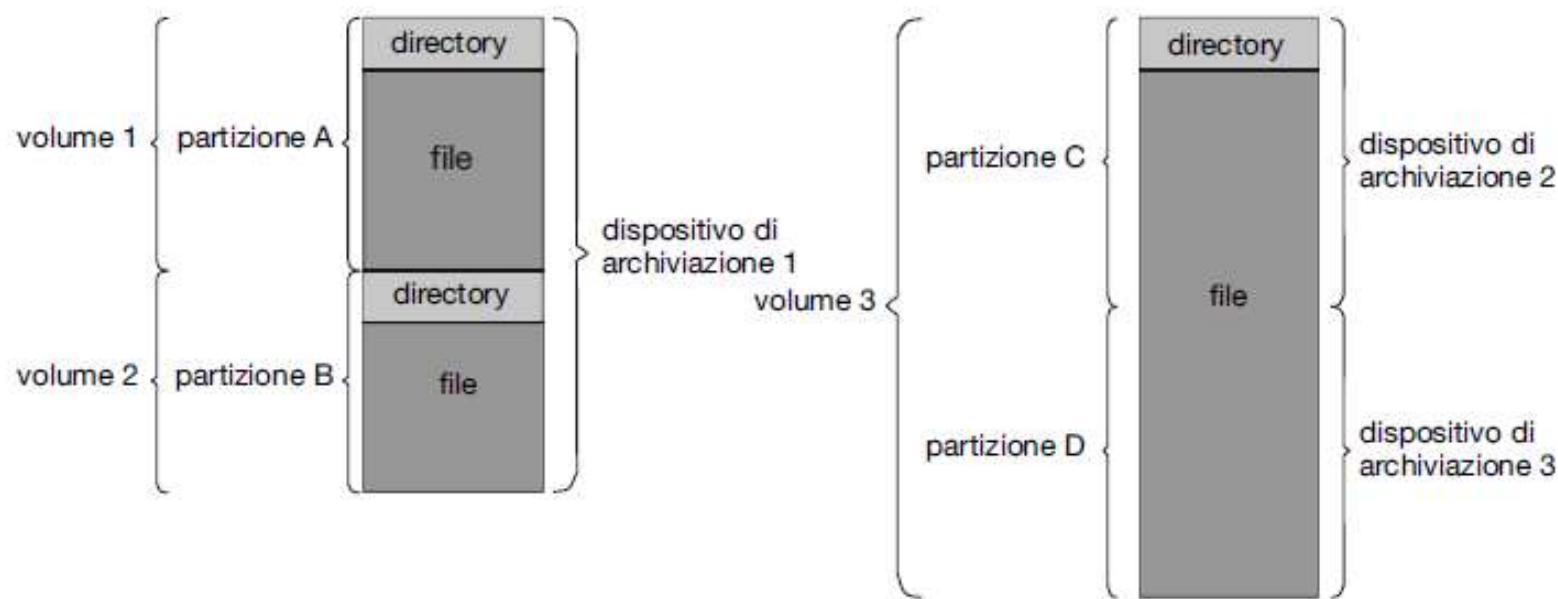


Figura 15.1 Tipica organizzazione di un file system.

File e Partizione

Il numero di file system in un sistema informatico può variare da zero a molti, e possono essere di tipo diverso.

Per esempio, un tipico sistema Solaris può avere molti file system di una dozzina di tipi diversi.

tmpfs—FS "temporaneo" in memoria volatile

objfs—FS "virtuale" (interfaccia al kernel che sembra un file system) dà accesso ai simboli del kernel

ctfs—FS virtuale

lofs— FS "loop back" permette ad un FS

l'accesso al posto di un altro

- procfs—FS virtuale che presenta informazione su i processi come un FS

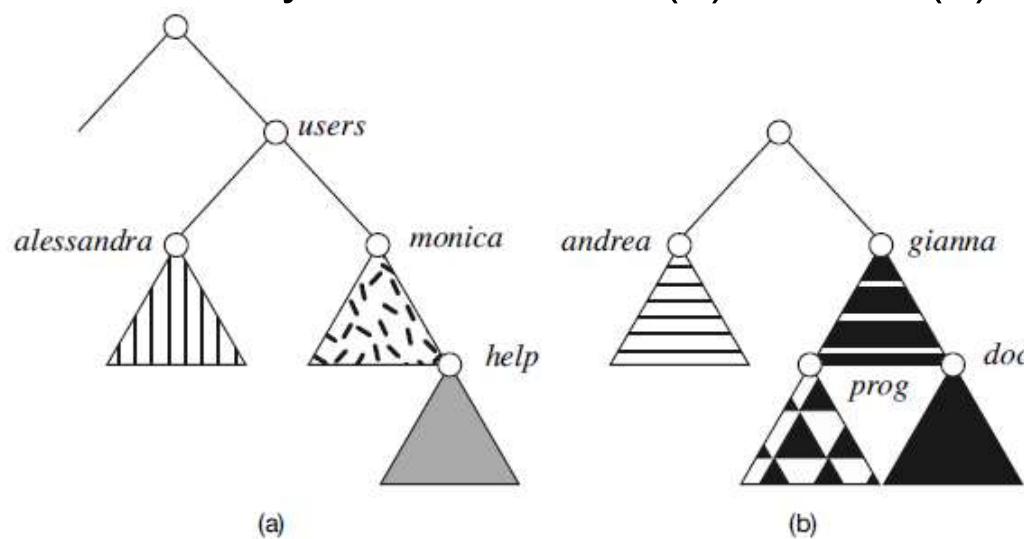
- ufs, zfs—FS general-purpose

| | |
|-------------------|--------|
| / | ufs |
| /devices | devfs |
| /dev | dev |
| /system/contract | ctfs |
| /proc | procfs |
| /etc/mnttab | mntfs |
| /etc/svc/volatile | tmpfs |
| /system/object | objfs |
| /lib/libc.so.1 | lofs |
| /dev/fd | fdfs |
| /var | ufs |
| /tmp | tmpfs |
| /var/run | tmpfs |
| /opt | ufs |
| /zpbge | zfs |
| /zpbge/backup | zfs |
| /export/home | zfs |
| /var/mail | zfs |
| /var/spool/mqueue | zfs |
| /zpbg | zfs |
| /zpbg/zones | zfs |

Figura 15.2 File system in Solaris.

Montaggio

- Come un file deve essere aperto per poter essere usato, un file system deve essere montato perché sia disponibile per i processi
 - La struttura delle directory costituita da più FS che devono essere montati per conividere il name-space
 - Montati da un punto di montaggio (directory vuota solitamente)
 - Quindi verifica che sia valido, si chiede al device driver di leggere le directory per verificare che il formato sia corretto
 - A questo punto si possono connettere e percorrere le directory
- I triangoli rappresentano i sottoalberi di directory di interesse. Si può accedere solo ai file del file system esistente (a). Invece (b) su /device/dsk



Montaggio

- I triangoli rappresentano i sottoalberi di directory di interesse. Si può accedere solo ai file del file system esistente (a). Invece (b) su /device/dsk
- A destra gli effetti dell'operazione di montaggio del volume residente in /device/dsk al punto di montaggio /users. Se si smonta il volume, il file system ritorna alla situazione rappresentata a sinistra.

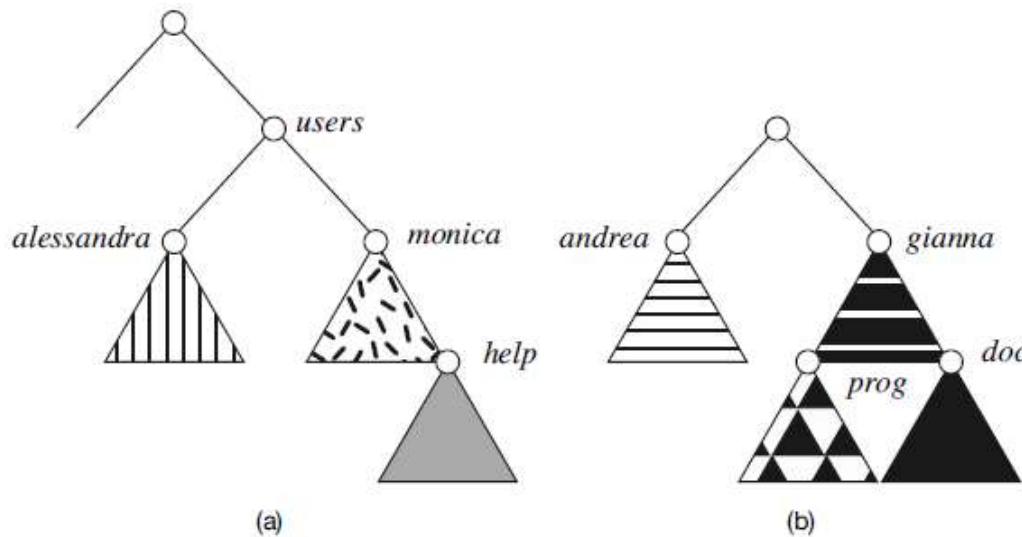


Figura 15.3 File system; (a) sistema esistente; (b) volume non montato.

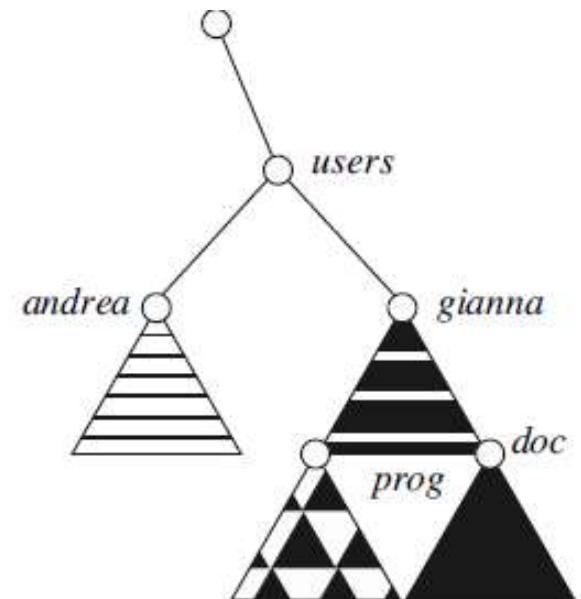


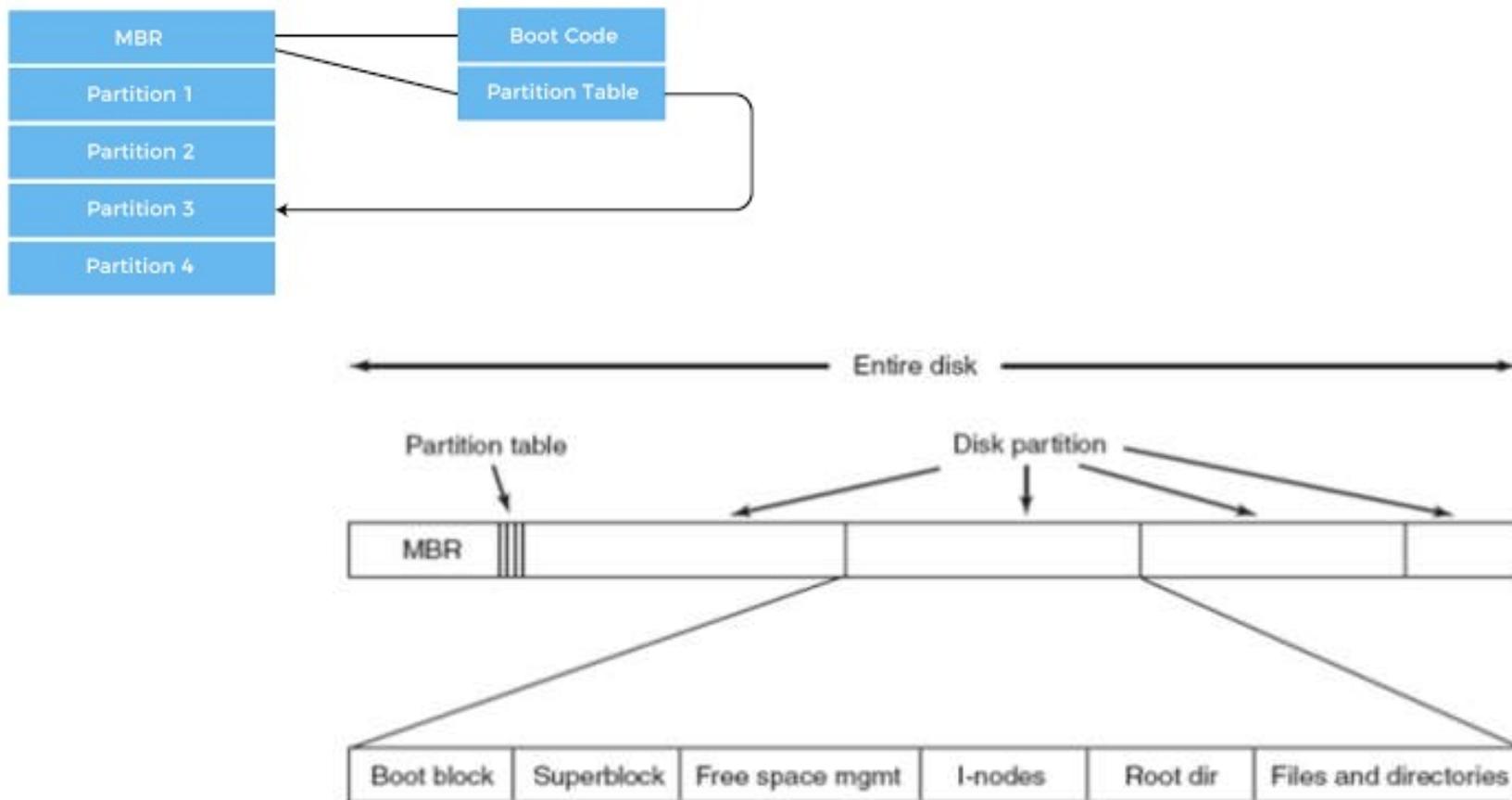
Figura 15.4 Volume montato sulla directory /users.

Partizioni e Montaggio

- Partizione può essere un volume che contiene un file system (“cotta”) or **raw** (cruda)– una sequenza di blocchi senza un file system
- Il boot block può puntare a un volume di boot; il bootstrap loader carica codice per caricare il kernel dal file system
 - ... o un programma boot management per boot con multipli SO
 - Il boot loader deve interpretare il formato del FS per poter caricare i blocchi
- **Partizione root** contiene il SO, altre partizioni possono contenere altri sistemi, altri file system, o possono essere raw
 - Montata a tempo di boot
 - Altre partizioni si possono montare automaticamente o manualmente
 - Su win montato su volume differenti name-space (lettere F:, E:, etc.)
- A tempo di mount, viene controllata la consistenza del file system
 - I metadata sono corretti?
 - ▶ se non lo sono, aggiusta e riprova
 - ▶ se lo sono, aggiungi la tabella del montaggio, permetti l'accesso

Partizioni e Montaggio

- Il boot block può puntare a un volume di boot
 - Il codice nel boot block consente di caricare il Sistema
 - Il superblock contiene parametri del file system



A possible file system layout.

File Sharing

- Se utenti multiple importante gestire la condivisione dei file
- Definito owner e groups (attributi del file)
- Permessi e protezioni

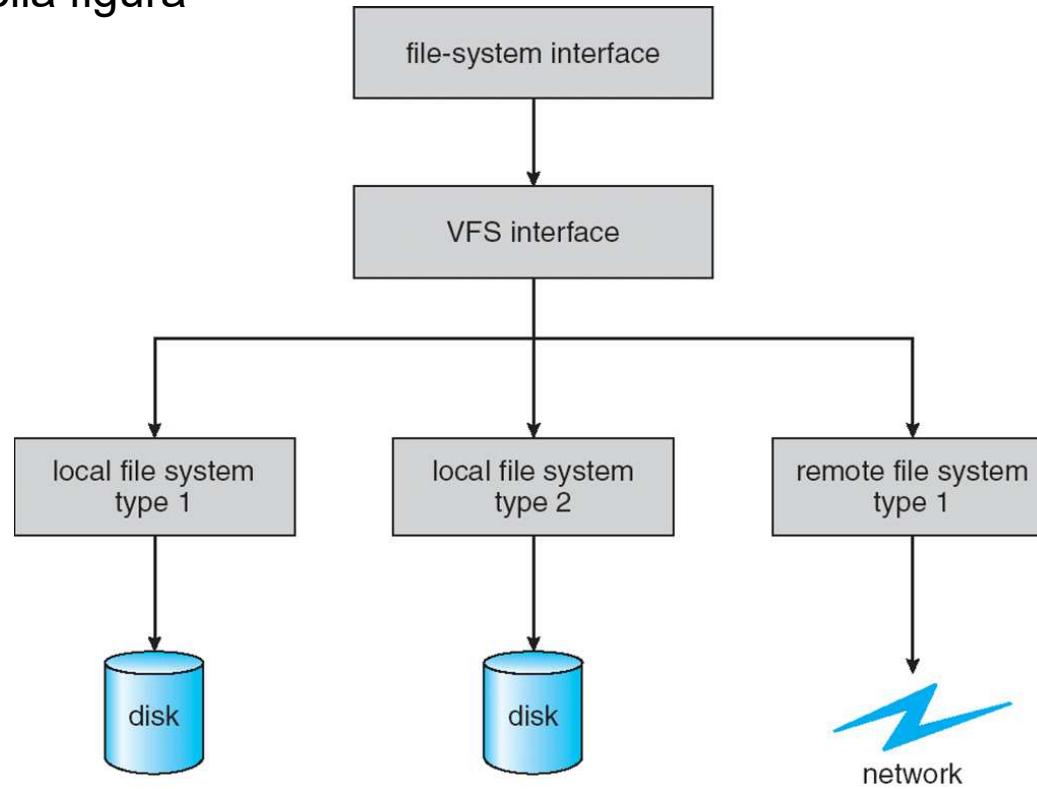
File System Virtuale

- Gli SO moderni supportano più sistemi operative integrati
- Come? Algoritmi e strutture replicate?
- **Virtual File System (VFS)** su Unix forniscono un approccio object-oriented per implementare multipli file system
- VFS permette di usare la stessa interfaccia per system call (API) per differenti tipi di file system
 - Separa operazioni dai dettagli implementativi
 - L'implementazione può riguardare molti tipi di FS, o network file system
 - Implementa **vnodes** che contengono inodes o dettagli su network file
 - Quindi assegna le operazioni sul FS alle routine implementate

File System Virtuale

- Le API sono per l'interfaccia VFS invece che per uno specifico FS

Per isolare le funzioni di base delle chiamate di sistema dai dettagli di implementazione si usano apposite strutture dati e procedure. In questo modo la realizzazione del file system si articola in tre strati principali, riportati in modo schematico nella figura



Implementazione File System Virtuale

- Per esempio, Linux ha quattro tipi di oggetti:
 - inode, file, superblock, dentry
- VFS definisce un insieme di operazioni sugli oggetti che devono essere implementati
 - Ogni oggetto ha un pointer ad una function table
 - Function table ha indirizzi di routine quella funzione su quell'oggetto
 - Per esempio gli oggetti sono:
- **inode object**, rappresenta un file
- **file object**, rappresenta un open file
- **superblock object**, rappresenta un file system
- **dentry object**, rappresenta una entry di una directory

Implementazione File System Virtuale

- Per esempio, Linux ha quattro tipi di oggetti:
 - inode, file, superblock, dentry
- VFS definisce un insieme di operazioni sugli oggetti che devono essere implementati
 - Ogni oggetto ha un pointer ad una function table
 - Function table ha indirizzi di routine quella funzione su quell'oggetto
 - Per esempio:

`int open(. . .)`—Open a file

`int close(. . .)`—Close an already-open file

`ssize_t read(. . .)`—Read from a file

`ssize_t write(. . .)`—Write to a file

`int mmap(. . .)`—Memory-map a file

Implementazione File System Virtuale

1. Separa le operazioni generiche del file system dalla loro realizzazione definendo un'interfaccia VFS uniforme
2. Permette la rappresentazione univoca di un file su tutta una rete
 - Il VFS distingue i file locali da quelli remoti, e distingue i file locali a seconda del relativo tipo di file system.
 - Il VFS, a seconda del tipo di file system, attiva le operazioni specifiche di un file system per gestire le richieste locali, e invoca le procedure del protocollo NFS per le richieste remote.

Semantica della Coerenza

- La **semantica della coerenza** è un importante criterio per la valutazione di qualsiasi file system che consenta la condivisione dei file.
- Una volta che la condivisione dei file è resa possibile, deve essere scelto e implementato un **modello di semantica della coerenza** che permetta la gestione di accessi simultanei a uno stesso file:

la semantica
UNIX

la semantica
delle sessioni

la semantica
dei file condivisi
immutabili

Semantica UNIX

- UNIX file system usa la seguente semantica:
 - Scrittura su file aperti da un utente visibile immediatamente da parte degli altri utenti con file aperto
 - Una modalità di sharing permette agli utenti di condividere il puntatore della locazione corrente del file
 - il pointer di un utente ha effetto su tutti gli utenti che condividono.
 - Un file ha una singola immagine che interfoglia tutti gli accessi
- Nella semantica UNIX un file è associato a una singola immagine fisica a cui si ha accesso esclusivo
 - Contese per questa singola immagine causa ritardi nei processi utente

Semantica delle Sessioni

- Andrew file system (OpenAFS) usa la semantica che segue:
 - Scritture su open file di utenti non visibili immediatamente da altri utenti che hanno lo stesso file aperto
 - Una volta chiuso il file le modifiche sono visibili solo nella sessione successivo. Le istanze aperte dei file non rispecchiano i cambiamenti
- Secondo questa semantica un file può avere temporaneamente molte immagini allo stesso tempo.
 - Utenti diversi possono eseguire read e write concorrentemente sui loro file immagine senza ritardi. Non ci sono vincoli sulla schedulazione degli accessi

Semantica Immutable Shared Files

- Un altro approccio è quello degli immutable shared file. Quando un file è dichiarato shared dal suo creatore, non può essere modificato.
- Un immutable file ha due proprietà:
 - il suo nome non può essere riusato
 - il suo contenuto non può essere alterato
 - Quindi i contenuti del file sono fissati
- L'implementazione di questa semantica su un sistema distribuito è semplice perché la disciplina dello sharing è read-only.

File System Remoto

- I metodi con i quali i file si condividono in una rete sono cambiati molto, seguendo l'evoluzione della tecnologia delle reti e dei file.
- Trasferimento dei file richiesto in modo esplicito dagli utenti → **ftp**
- **File System Distribuito** (*distributed file system, DFS*), che permette la visibilità nel calcolatore locale delle directory remote.
- **World Wide Web** → da un certo punto di vista, un ritorno al primo.
- **Cloud computing**

File System Distribuito

Un **DFS (file system distribuito)** è un servizio di file system i cui **client, server e dispositivi di memorizzazione sono sparsi tra i siti di un sistema distribuito.**



L'attività di servizio si deve eseguire per mezzo della rete e invece di un unico sito di memorizzazione dei dati centralizzato vi sono più dispositivi di memorizzazione indipendenti.

File System Distribuito

- Il **modello client-server** consente la condivisione trasparente dei file tra uno o più client.

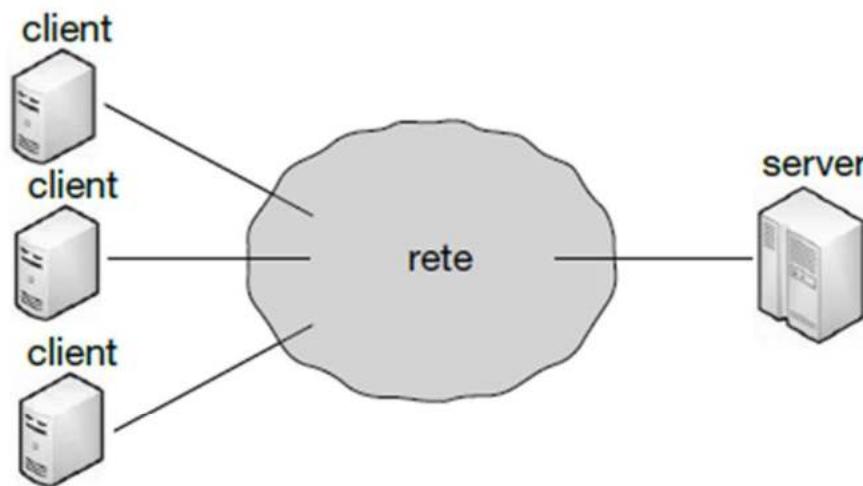


Figura 19.12 Modello client-server di DFS.

- Il **modello client-server** di DFS, per come è progettato, può soffrire di un singolo punto di guasto in caso di arresto anomalo del server.
- Inoltre, il **server** rappresenta un **collo di bottiglia** per tutte le richieste sia di dati sia di metadati, con conseguenti problemi di scalabilità e larghezza di banda.

Modello Client Server

- Nel caso distribuito, FS su server e client richiede l'accesso
 - Il server dichiara le directory ed i file a disposizione, più complicato identificare il client (con ID non sicuro), metodi insicuri i più usati
 - Un client può essere specificato dal network name o altri ID come un IP address, ma possono essere falsificati o imitare
 - ▶ In Linux, Network File System (NFS) autenticazione basata sull'info dell'host del client
- Una volta montato il file system remoto, vengono inviate le richieste di operazioni sui file al server tramite il protocollo DFS.
 - ▶ Es., richiesta di apertura file insieme all'ID dell'utente richiedente.
 - ▶ Il server applica i controlli di accesso per determinare se l'utente dispone delle credenziali per accedere al file nella modalità richiesta
 - ▶ Se è consentito, viene restituito un handle di file all'applicazione client
 - ▶ Il client chiude il file al termine dell'accesso

Naming e Trasparenza

- Le architetture client-server utilizzano un sistema di naming distribuito (Distributed Information Systems) per avere spazio di nomi unificato
 - Il **naming** è una funzione di associazione tra oggetti logici e oggetti fisici
- Idealmente un DFS dovrebbe apparire ai propri client come un normale file system centralizzato
- La molteplicità e la dispersione dei suoi server e dei suoi dispositivi di archiviazione dovrebbero essere **trasparenti**.
- Un **DFS trasparente** facilita la mobilità dei client portando l'ambiente di lavoro del client nel sito in cui il client effettua l'accesso.
- **Trasparenza rispetto alla locazione** → Il nome di un file non rivela alcun indizio sulla sua locazione fisica
- **Indipendenza dalla locazione** → Non si deve modificare il nome di un file se cambia la sua locazione di memoria fisica

Naming

- Le architetture client-server utilizzano un sistema di naming distribuito (Distributed Information Systems) per avere spazio di nomi unificato
 - Il **naming** è una funzione di associazione tra oggetti logici e oggetti fisici
 - Esistono parecchi metodi per il **naming** in un DFS
 - In quello più semplice i file si nominano attraverso una combinazione del loro nome di macchina e del loro nome locale, il che garantisce un nome unico per tutto il sistema
 - Un altro metodo, usato dall'NFS (*network file system* di Sun), fornisce mezzi per unire directory remote a directory locali, dando così l'impressione di un albero di directory coerente
 - Un terzo metodo consiste in una sola struttura globale di nomi che si estende a tutti i file del sistema

Naming

- Le architetture client-server utilizzano un sistema di naming distribuito (Distributed Information Systems) per avere spazio di nomi unificato
 - Possono usare il DNS (Domain Name System) che fornisce servizi di traduzione host-name-to-network-address su Internet
 - Microsoft CIFS (Common Internet File System)
 - Utilizza active directory, versione del protocollo di autenticazione di rete kerberos (Network Authentication Protocol) per fornire servizi di naming e autenticazione tra i computer in una rete
 - Oracle Solaris verso il protocollo LDAP (lightweight directory-access protocol) come meccanismo sicuro per il naming distribuito

Sun Network File System (NFS)

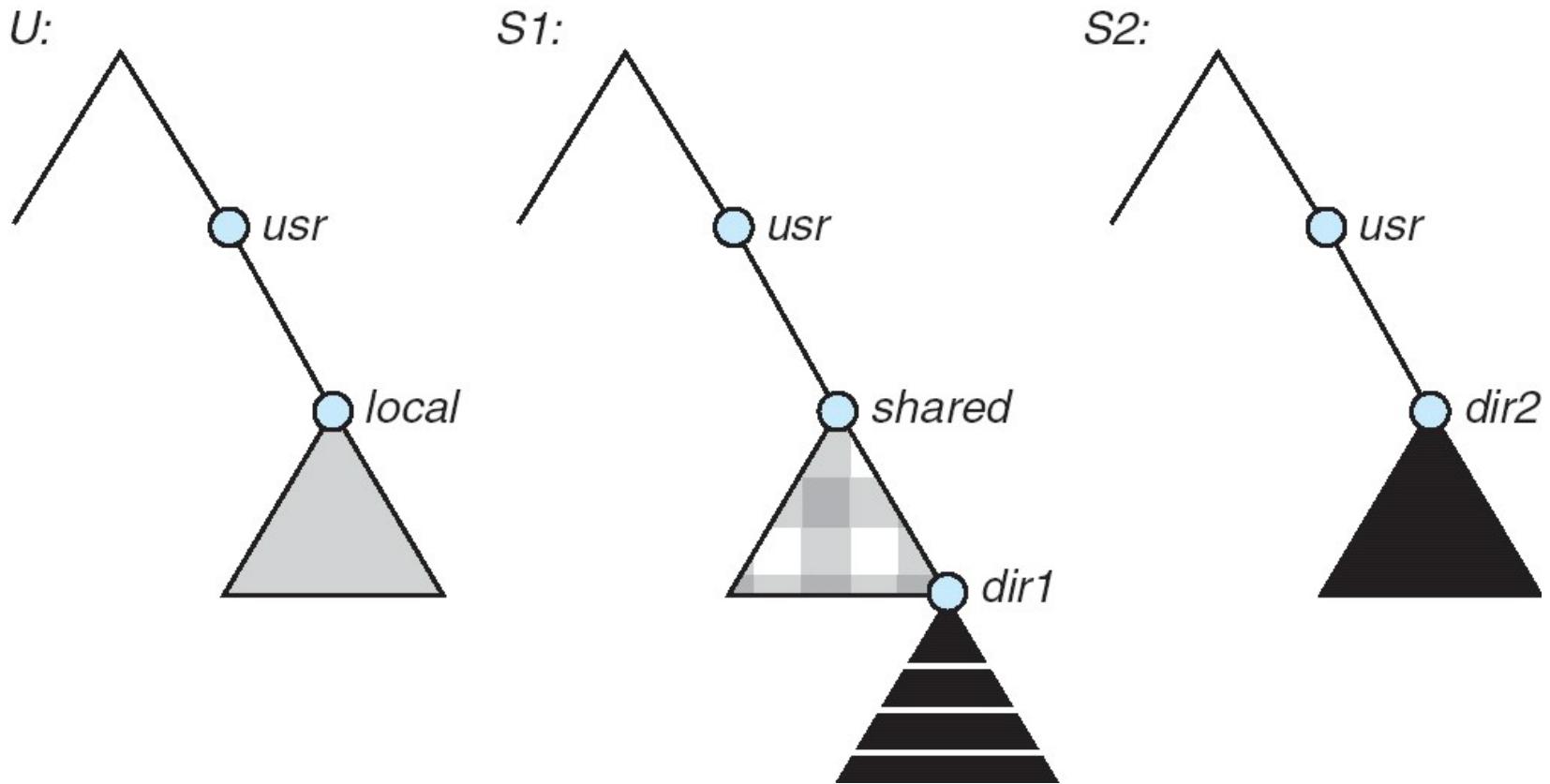
- I Network file systems (NFS) sono comuni, tipicamente integrati nella struttura delle directory
- Ad esempio Sun NFS è una implementazione e una specifica di un sistema software system per l'accesso remoto a file su LANs (o WANs)
- L'implementazione è parte di Solaris e SunOS su Sun workstation sia con un protocollo a datagram (UDP/IP protocol ed Ethernet) sia TCP

NFS

- Workstation interconnesse con file system indipendenti, che permettono condivisioni tra file system in modo trasparente
 - Modalità client-server (ogni macchina può fare sia da client che da server)
 - Una directory remota viene montata su una directory di un file system locale
 - ▶ Affinché una directory remota sia accessibile in modo trasparente da un calcolatore il client deve prima eseguire un'operazione di montaggio
 - ▶ La directory montata assume l'aspetto di un sottoalbero integrato nel file system locale e sostituisce il sottoalbero che discende dalla directory locale (radice directory montata)
- Specifica della directory remota per il montaggio non è fatta in modo trasparente
 - ▶ Serve l'host name della directory remota
 - ▶ Dopo il montaggio i file nella directory remota sono quindi accessibili in modo trasparente
- Soggetta ad accrediti di diritti di accesso, potenzialmente ogni file system (o directory in un file system) può essere montata da remoto su qualunque directory locale

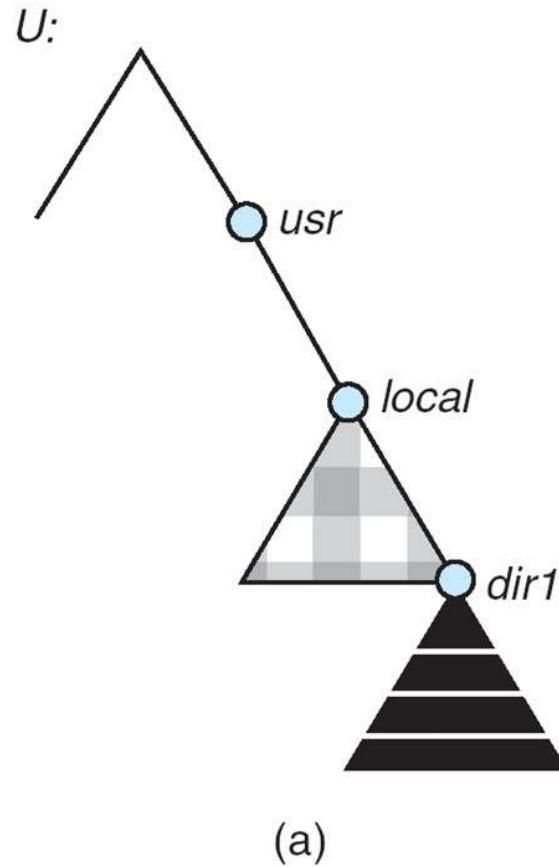
File System Indipendenti

Tre file system indipendenti



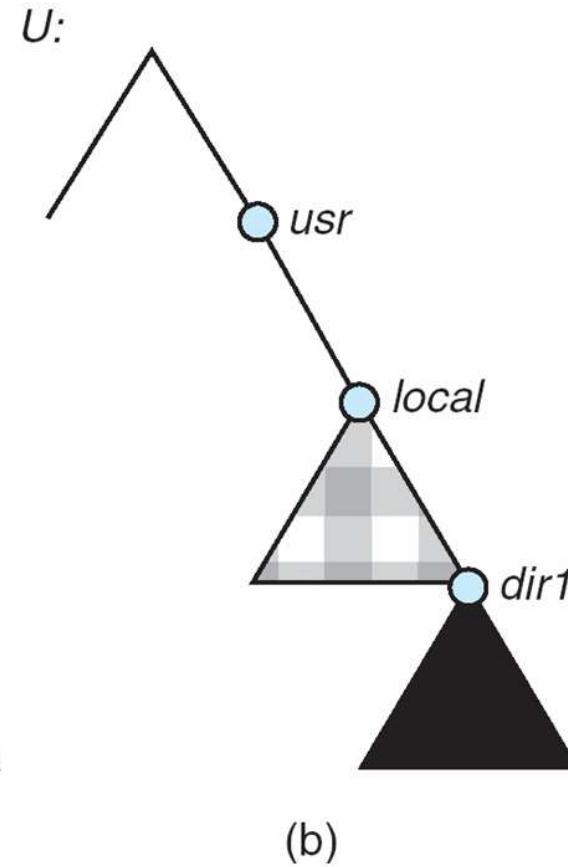
Montaggio in NFS

S1:/usr/shared montata sopra U:/usr/local



Montaggio

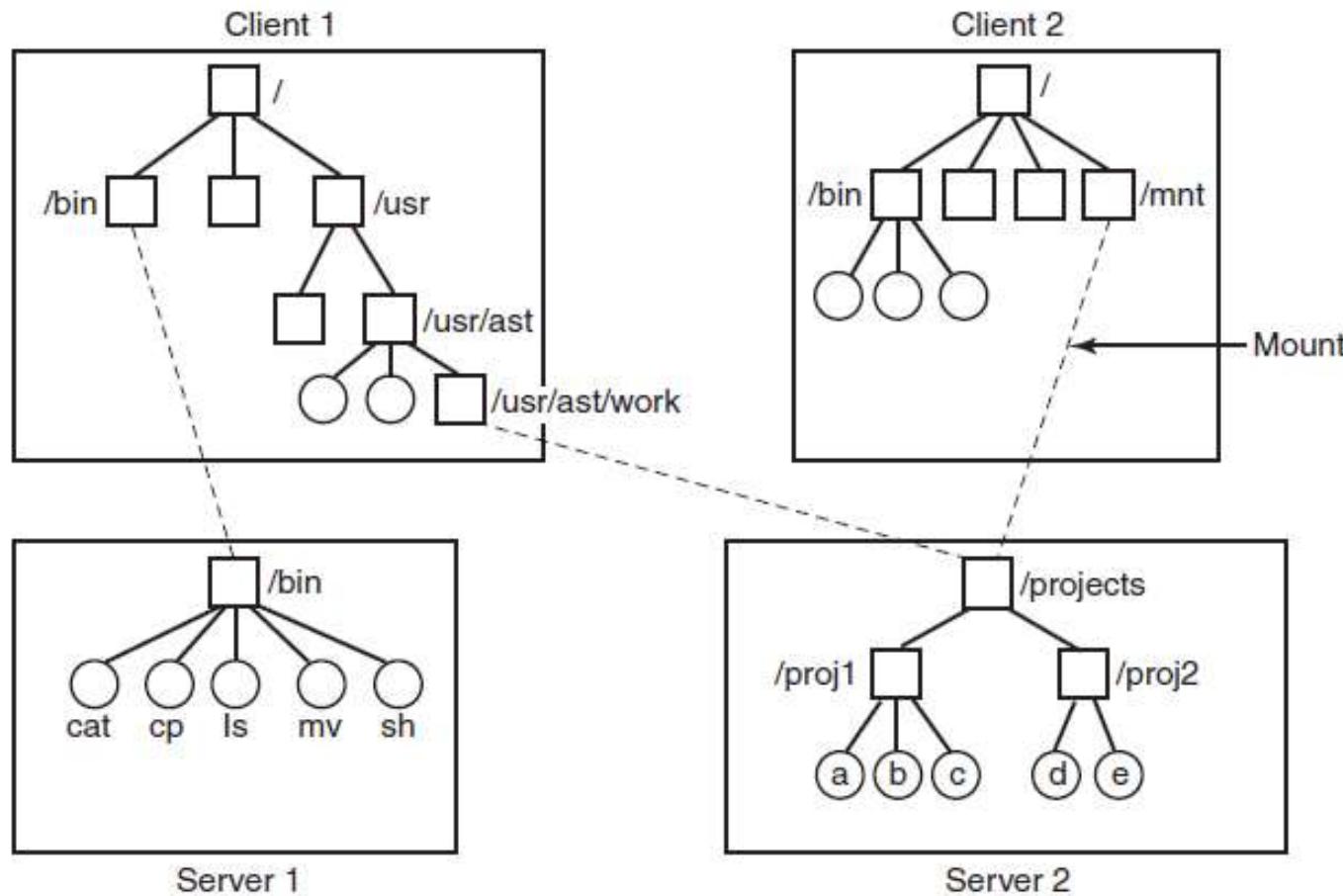
S2:/usr/dir2 sopra U:/usr/local/dir1



Montaggio a cascata

Spostamento su più macchine

Montaggio in NFS



NFS

- NFS progettato per operare ambiente eterogeneo di macchine differenti, sistemi operativi, ed architecture di rete
 - la specifica del NFS è indipendente
- Indipendenza ottenuta usando primitive RPC sopra un protocollo di External Data Representation (XDR)
- La specifica NFS distingue tra i servizi forniti da:
 - montaggio
 - remote-file-access

NFS: Protocollo di Montaggio

- Stabilisce una connessione logica iniziale tra client e server
- L'operazione di montaggio include il nome di una dir remota da montare e il nome di una macchina server che la contiene
 - La richiesta di mount è mappata su una RPC ed inoltrata al mount server che gira sulla macchina server
 - Export list – specifica i file system locali che il server esporta per il montaggio con i nomi delle macchine che possono montarli
- Se la mount request è conforme alla export list, il server ritorna un file handle— una chiave per ulteriori accessi
- File handle – un identificatore del file-system e un inode number per identificare la mounted directory nel file system esportato
- L'operazione di montaggio cambia solo la vista dell'utente e non ha effetto sul server

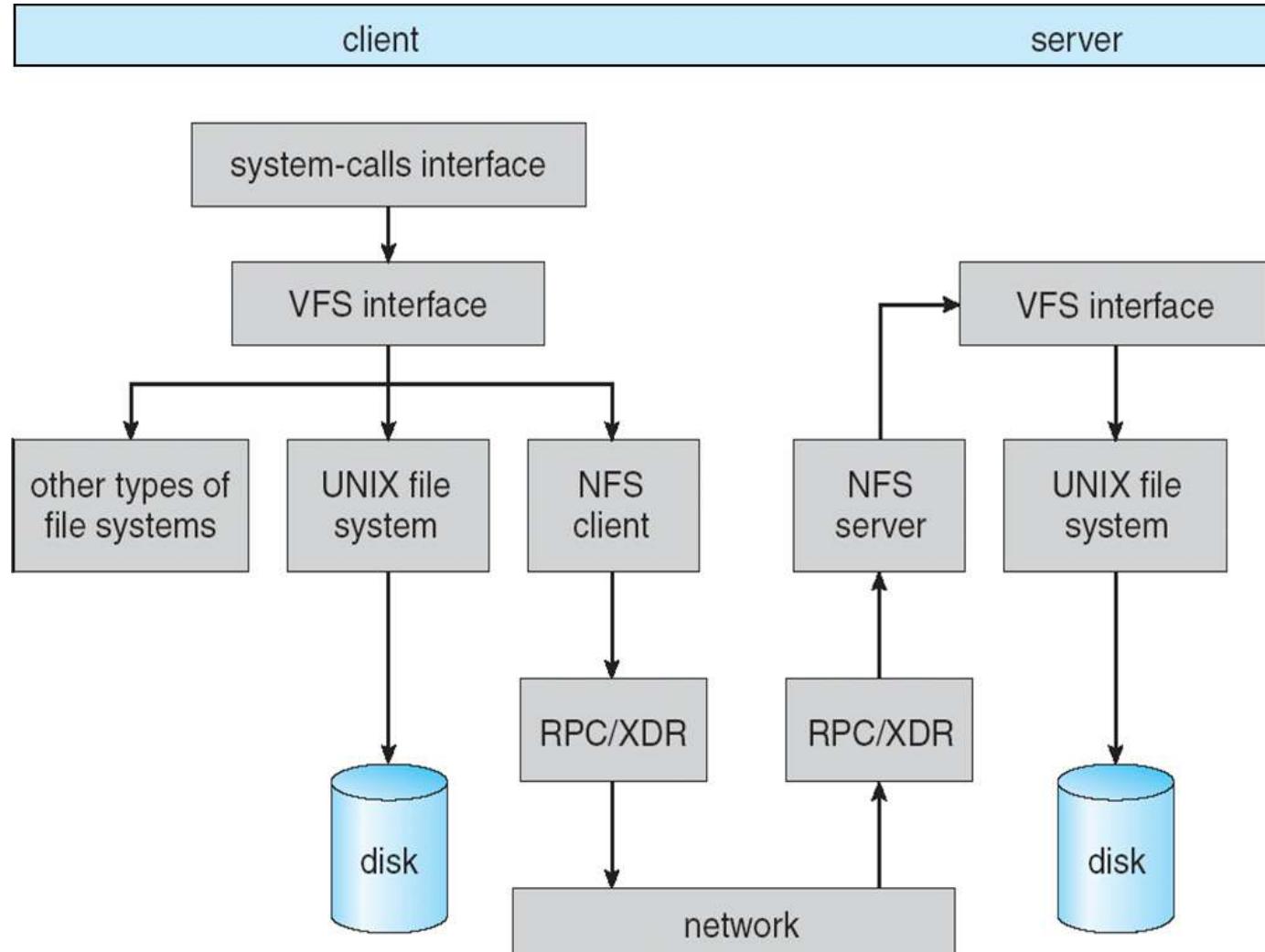
Protocollo NFS

- Insieme di remote procedure calls per le operazioni remote
- Le procedure supportano le operazioni seguenti:
 - ricerca di file in una directory
 - lettura di insiemi di directory entries
 - Manipolazione di link e directory
 - Accesso agli attribute dei file
 - Lettura/scrittura di file
- NFS server sono **stateless**; ogni richiesta deve fornire l'insieme degli argomenti (NFS V4 in arrivo – molto diverso, stateful)
- Dati modificati devono essere committed sul disco del server prima che i risultati vengono ritornati al client (si perde il vantaggio del caching)

Strati di Architettura NFS

- Interfaccia a UNIX file-system (basata su chiamate **open**, **read**, **write**, **close** e **file descriptors**)
- Virtual File System (VFS) layer – distingue file locali da quelli remoti, i file locali ulteriormente distinti per tipo di file-system
 - VFS attiva operazioni file-system-specifiche per gestire richieste locali secondo i tipi di file-system
 - Chiama le procedure del protocollo NFS per richieste remote
- NFS service layer – strato basso dell'architettura
 - Implementa il protocollo NFS

Architettura NFS



File System Implementazione

File System Implementazione

- File-System Struttura
- File-System Operazioni
- Implementazione Directory
- Metodi di Allocazione
- Gestione del Free-Space
- Efficienza e Performance
- Recovery
- Esempio: WAFL File System

Obiettivi

- Descrivere i dettagli dell'implementazione del file system locale e delle strutture delle directory
- Discutere allocazione di blocchi, algoritmi di free-block e trade-off

Realizzazione di un File System

- Una specifica realizzazione del file system risponde alle seguenti domande:
 - Come vengono memorizzati i file e le directory?
 - Come viene gestito lo spazio sul dispositivo di memorizzazione?
 - Quali strutture dati e algoritmi vengono usate dal SO per gestire in modo efficiente e affidabile il file system?

Struttura del File-System

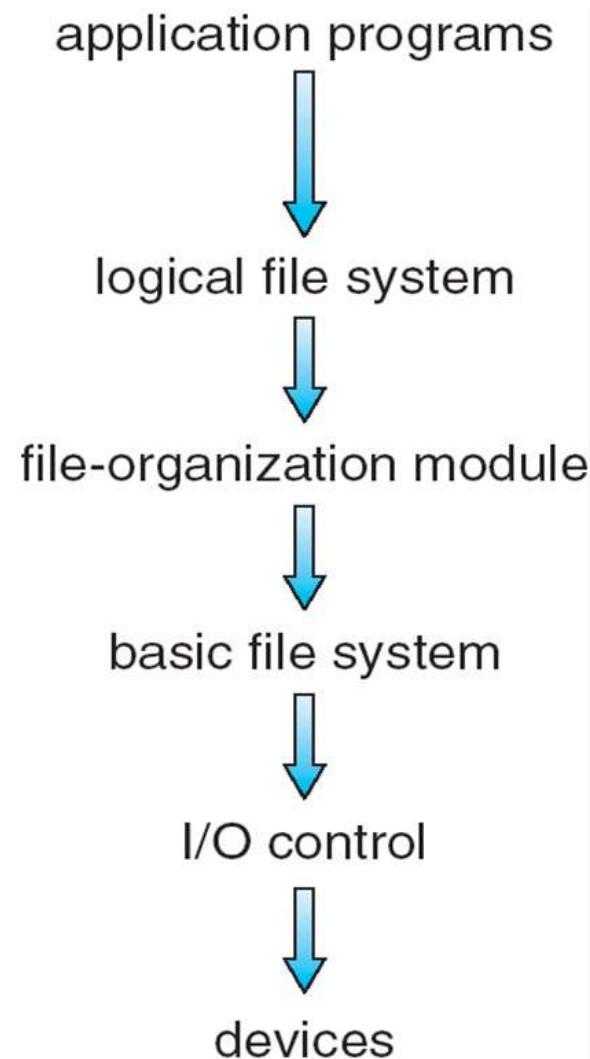
- Il file system risiede permanentemente in memoria di massa
- Tipicamente questa memoria è fornita dai dischi, i quali vengono usati per due principali motivi:
 - Possono essere riscritti localmente (leggo e riscrivo lo stesso blocco)
 - Permettono l'accesso, sia sequenziale che diretto, ai blocchi fisici che memorizzano i diversi file
- La dimensione dei blocchi varia da dispositivo in dispositivo
 - Da 32 a 4096 Byte
 - Solitamente 512 Byte, 4096 per NVM

Struttura del File-System

- Struttura File
 - Unità logica di storage
 - Collezione di informazione correlata
- **File system** residente in memoria secondaria (dischi)
 - Fornisce una user interface per lo storage, mappando memoria logica su memoria fisica (come deve apparire all'utente?)
 - Fornisce un accesso efficiente e conveniente al disco consentendo ai dati di essere memorizzati, localizzati e ripresi facilmente (come mappare?)
- Disco dispositivo per file system
 - Fornisce in-place rewrite (letto/trasformato/riscritto) e accesso random
 - Trasferimenti I/O in **blocchi** di **settori** (es., 512 bytes, 4096 per NVM)
- **File Control Block** – struttura di storage che fornisce informazione su file
- **Device driver** controlla il dispositivo fisico
- File system organizzato in strati

Struttura del File-System

- Il file system sono tipicamente realizzati secondo una struttura a strati
- Il livello superiore si basa sui servizi offerti dal livello inferiore



Strati del File System

- Il file system sono tipicamente realizzati secondo una struttura a strati
- Il livello superiore si basa sui servizi offerti dal livello inferiore
- Livelli di un file system:
 - **Controllo dell'I/O:** driver dei dispositivi e gestore degli interrupt
 - ▶ Si occupa del trasferimento dell'informazione dalla memoria di massa a quella centrale
 - **File System di base:** invia dei comandi di base al controllo dell'I/O per scrivere e leggere blocchi fisici
 - ▶ E.g. Leggi il blocco nell'unità 1, cilindro 12, superficie 4, settore 2
 - **Modulo di organizzazione dei file:** conoscendo il metodo di allocazione traduce un indirizzo di un blocco logico in un indirizzo di un blocco fisico
 - ▶ Comprende anche il gestore dello spazio libero (mantiene la lista dei blocchi liberi)
 - **File System Logico:** gestisce i metadati e la struttura delle directory
 - ▶ Gli attributi dei file vengono salvati in File Control Blocks (FCB)

Strati del File System

- **Device driver** gestiscono i dispositivi I/O al livello di controllo I/O
 - Da comandi come “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” genera comandi hardware specific per il controllore del hardware
- **Basic file system** comandi del tipo “prendi il blocco 123” li traduce in comandi generici per i device driver
 - Comandi con indirizzi logici, schedulazione delle richieste I/O
 - Gestisce anche buffer e cache (allocazione, freeing, replacement)
 - Buffer mantengono data in transito, caches mantengono metadati frequentemente usati
- **File organization module** livello di gestione dei file, indirizzi logici e blocchi fisici
 - Ogni blocco è numerato da 0 a N
 - Traduce un blocco logico # in un blocco fisico #
 - Gestisce lo spazio libero e l’allocazione del disco

Strati del File System

- **Logical File System** gestisce informazione sui metadati
 - Traduce i nomi di file in file number, file handle, locazioni mantenendo i File Control Blocks (**inodes** in UNIX)
 - Gestione delle directory
 - Protezione
- Stratificazione utile per ridurre la complessità e la ridondanza,
 - Il codice è più riutilizzabile
 - Diversi file system possono essere implementati nello stesso SO riusando gli strati di basso livello
 - Ma aggiunge overhead e può ridurre la performance
- La decisione di quanti strati implementare è importante nel progetto del SO

Strati del File System

- Esistono diversi file system, gli SO ne supportano più di uno
 - Ognuno con il suo formato
 - CD-ROM è ISO 9660;
 - Unix ha **UFS**, FFS;
 - Windows ha FAT, FAT32, NTFS come floppy, CD, DVD Blu-ray,
 - Linux ha più di 40 tipi, con **extended file system** ext2, ext3, ext4 principali; più distributed file systems, etc.
 - Nuovi ancora in arrivo – ZFS, GoogleFS, Oracle ASM, FUSE

Implementazione del File-System

- Un file system richiede la definizione di diverse strutture dati
- Alcune risiedono in memoria di massa, altre in memoria centrale
- Strutture dati in memoria di massa:
 - **Boot Control Block**: contiene informazioni necessarie per avviare il SO contenuto nel disco da quel volume (se il volume non contiene SO è vuoto)
 - **Volume Control Block**: contiene dettagli riguardanti il volume
 - ▶ Numero di blocchi fisici e dimensione
 - ▶ Contatore dei blocchi liberi e puntatori ai blocchi
 - ▶ Contatore dei FCB presenti nel volume e puntatori ad essi
 - **Struttura della directory**: organizza i file
 - **File Control Block**: memorizza gli attributi dei file
 - ▶ inode number, permessi, dimensione, date

| |
|--|
| file permissions |
| file dates (create, access, write) |
| file owner, group, ACL |
| file size |
| file data blocks or pointers to file data blocks |

Strutture File System in Memoria

- Info per gestione e per aumentare la performance
 - Dati acquisiti quando il SO è montato, aggiornati e cancellati quando è smontato
- Strutture dati in memoria centrale:
 - **Tabella di montaggio**: contiene informazioni relative ai volumi attualmente montati
 - **Struttura delle directory** (in parte): contiene le informazioni relative alle directory attualmente in uso dai processi (Cache di directory ad accesso recente)
 - **Tabella dei file aperti** (livello SO)
 - **Tabelle dei file aperti** (livello processo)
 - **Blocchi del file system**: durante la loro lettura e scrittura (nei buffer)

Creazione file

- Per creare un nuovo file un processo
 - Il SO crea e alloca un nuovo FCB
 - Carica in memoria centrale la directory che lo dovrà contenere
 - Aggiorna la directory
 - Riscrive la directory in memoria di massa

- Chiamato il Logical File System (che conosce il formato delle directory)
- Il Logical File System alloca un nuovo FCB
- Il Sistema legge la directory nella memoria, aggiorna con nuovo file, riscrive

- Per alcuni SO la directory è un file (UNIX) per altri no (Windows)
- Il Logical File System chiama il File Organization Module che chiama il Basic File System

Apertura e Chiusura File

- Open() e Close()
- Il SO controlla la tabella dei file aperti per vedere se il file è già aperto
 - In caso negativo
 - ▶ Aggiunge il FCB alla tabella del SO
 - In ogni caso
 - ▶ Aggiunge alla tabella del processo chiamante un nuovo elemento che punta alla tabella del SO
 - ▶ Incrementa il contatore delle aperture del file
- Alla chiusura del file il SO decrementa il contatore e rimuove l'elemento di quel file dalla tabella del processo chiamante
 - Se il contatore di aperture vale 0 il FCB viene rimosso dalla tabella del SO

Apertura File

- Più in dettaglio:
- Open() passa il nome del file al Logical File System
- Prima verifica sulla system-wide open-file table se già aperto
 - Se aperto, open-file table per-processo aggiornato con pointer su entry di system-wide table
 - Se non è aperto, si cerca il nome nella struttura della directory
 - ▶ Parte della struttura è in cache per velocizzare
 - Appena trovato viene copiata la FCB in system-wide open-file table (con contatore di file aperti)
 - Poi si inserisce una entry nella open-file table per-processo
 - ▶ Puntatore alla system-wide
 - ▶ Offset di lettura/scrittura
 - ▶ Modalità di accesso
 - Open restituisce un puntatore alla open-file table per-processo
 - ▶ Tutte le operazioni su questo puntatore

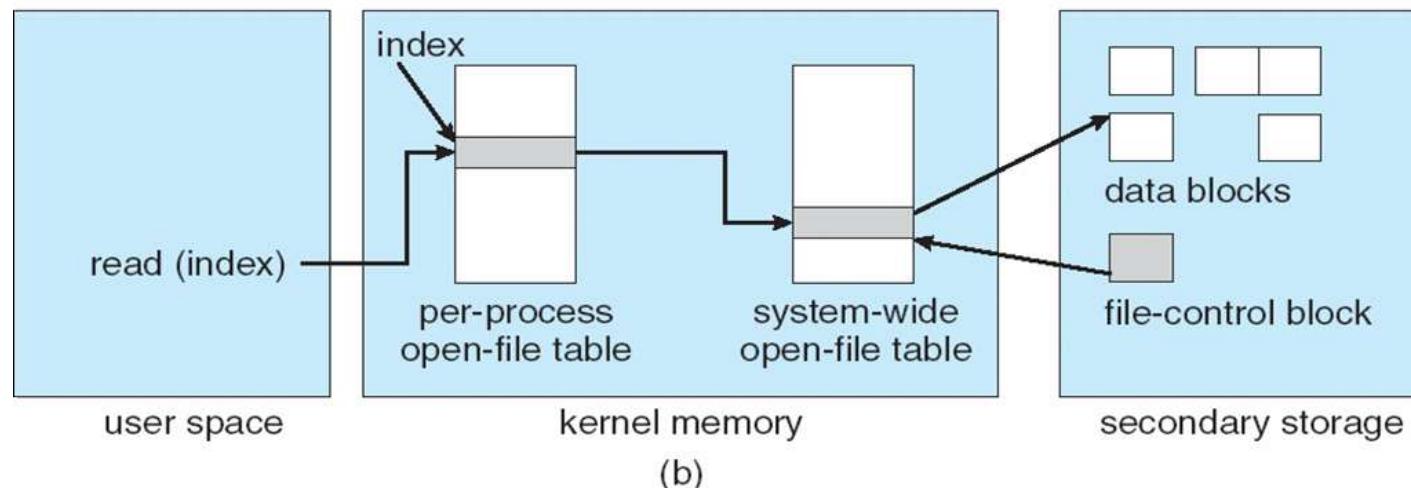
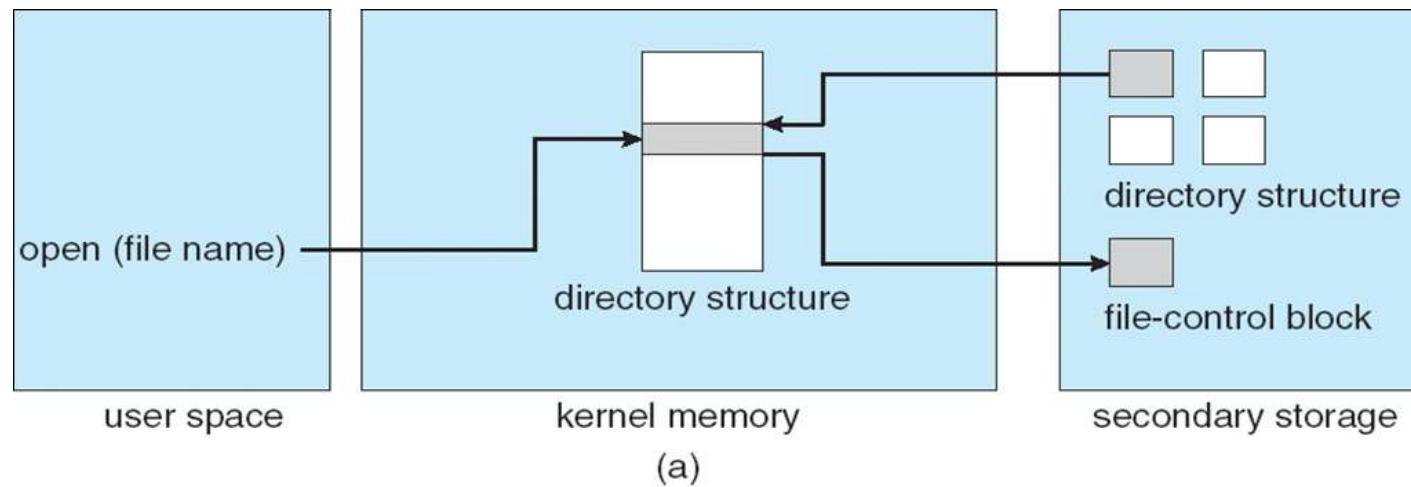
Chiusura File

- Quando un processo chiude un file (Close())
 - Deallocata la entry nel open-file table per-processo
 - Decrementato il contatore sulla system-wide open-file table
 - Quando tutti hanno chiuso il file
 - La memoria secondaria è scritta
 - Si dealloca la entry nella system-wide table
- Molti sistemi tengono l'informazione sugli open-file in memoria principale (tranne che i blocchi riferiti che sono in memoria secondaria)

Strutture File System in-Memory

- Strutture necessarie del file system fornite dal SO

- apertura di un file e lettura del file



Partizioni e Montaggio

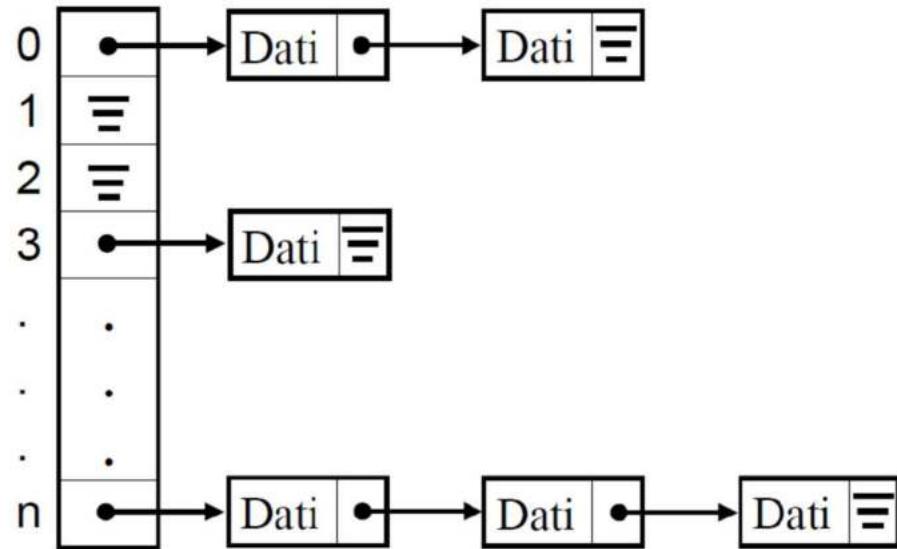
- Tipicamente un unico disco può contenere diverse partizioni
 - ognuna può avere un file system diverso
- Un partizione senza file system è detta raw partition
 - Esempio è la partizione usata per lo swap nei sistemi Unix
- Un'altra partizione priva di file system, ma con un formato proprio, è quella di boot
 - Consiste in un insieme di blocchi che vengono caricati in memoria centrale e contengono le istruzioni per l'avvio del SO
- Una volta scelto il SO la partizione radice viene montata
 - Essa contiene il kernel del SO ed eventuali file di sistema
 - Altre partizioni possono essere montate in modo automatico o su richiesta
 - Il SO inserisce le informazioni della partizioni montate nella tabella di montaggio

Implementazione Directory

- Implementazione delle directory impatta molto sulle prestazioni
- Modo più semplice di implementare le directory è con una **lista lineare** un array contenente identificatore del file e puntatore al FCB
 - Facile da programmare (si cerca il file e si accede)
 - Poco efficiente da eseguire
 - ▶ Tempo di ricerca proporzionale alla dimensione
 - ▶ Si può tenere una lista ordinata per fare ricerca binaria, ma costi di gestione
- **Hash Table** – lista lineare con struttura dati ad hash table
 - Abbassa i tempi di ricerca su directory
 - **Collisioni** – situazioni dove due file name hanno hash sulla stessa locazione location
 - ▶ Metodo chained-overflow

Implementazione Directory

- **Hash Table** – lista lineare con struttura dati ad hash table
 - Abbassa i tempi di ricerca su directory
 - **Collisioni** – situazioni dove due file name hanno hash sulla stessa location
 - Metodo chained-overflow

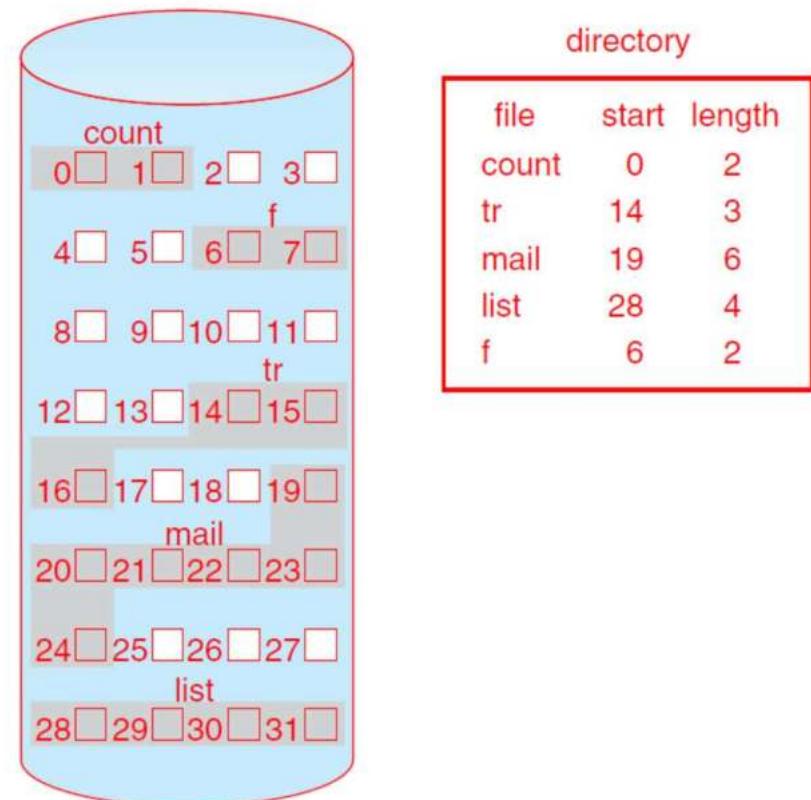


Metodi di Allocazione - Contigua

- Un metodo di allocazione si riferisci al modo in cui i blocchi sono allocati per i file
 - Accesso veloce e spazio utilizzato in modo efficiente
- Tre metodi:
 - Contiguo
 - Linkato
 - Indicizzato

Metodi di Allocazione - Contigua

- **Allocazione Contigua** – ogni file occupa un insieme di blocchi contigui
 - Miglior performance in molti casi
 - Semplice – richiesto solo locazione di partenza (numero del blocco) e lunghezza (numero di blocchi) – dati salvati in FCB
- Pro:
 - ▶ spostamenti della testina richiesti per accedere a tutti i blocchi di un file è trascurabile
 - ▶ Allo stesso mondo è trascurabile il tempo di ricerca
- Problemi:
 - ▶ trovare spazio per il file,
 - ▶ sapere la dimensione del file,
 - ▶ frammentazione esterna,
 - ▶ Necessità di **compattare**
 - ▶ **off-line (downtime)** o **on-line**



Allocazione Contigua

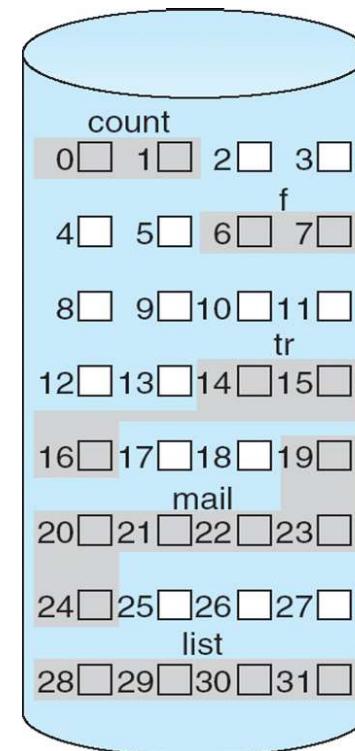
□ Mapping da logica a fisica

LA/512

Q

R

Blocco da accedere = Q +
indirizzo di inizio
Dislocazione nel blocco = R



| directory | | |
|-----------|-------|--------|
| file | start | length |
| count | 0 | 2 |
| tr | 14 | 3 |
| mail | 19 | 6 |
| list | 28 | 4 |
| f | 6 | 2 |

Metodi di Allocazione - Contigua

- Pro: l'accesso ai file può essere eseguito semplicemente sia in modo sequenziale che diretto (meno veloce)
- Contro 1: è necessario trovare lo spazio libero per allocare il file
 - Diversi algoritmi
 - ▶ First-fit: primo buco abbastanza grande
 - ▶ Best-fit: il buco che approssima meglio la dimensione (per eccesso)
 - Si presenta il problema della frammentazione esterna
 - ▶ Può essere risolta tramite la compattazione
 - ▶ Si spostano i dati su un altro disco
 - ▶ Si crea una così una zona contigua di spazi libero
 - ▶ Si riportano i file su disco originale allogandoli in modo contiguo
 - ▶ La compattazione richiede molto tempo e può essere fatta
 - Off line: non è possibile usare il volume
 - On line: peggiora le prestazioni

Metodi di Allocazione - Contigua

- Pro: l'accesso ai file può essere eseguito semplicemente sia in modo sequenziale che diretto (meno veloce)
- Contro 2: è necessario stimare quanto spazio allocare ad un file
 - Quando un file finisce lo spazio disponibile
 - ▶ Il processo può essere interrotto
 - ▶ Il file può essere spostato in una zona di memoria più grande
 - Allocare più spazio di quello inizialmente necessario può portare alla frammentazione interna
 - ▶ Specialmente nel caso di file che crescono lentamente
 - Alcuni sistemi usano l'allocazione contigua estesa
 - ▶ Quando un file necessita di spazio gli viene assegnata una seconda area contigua
 - ▶ Ogni area è caratterizzata da: blocco d'inizio, numero blocchi, area successiva

Sistema basato su Allocazione Estesa

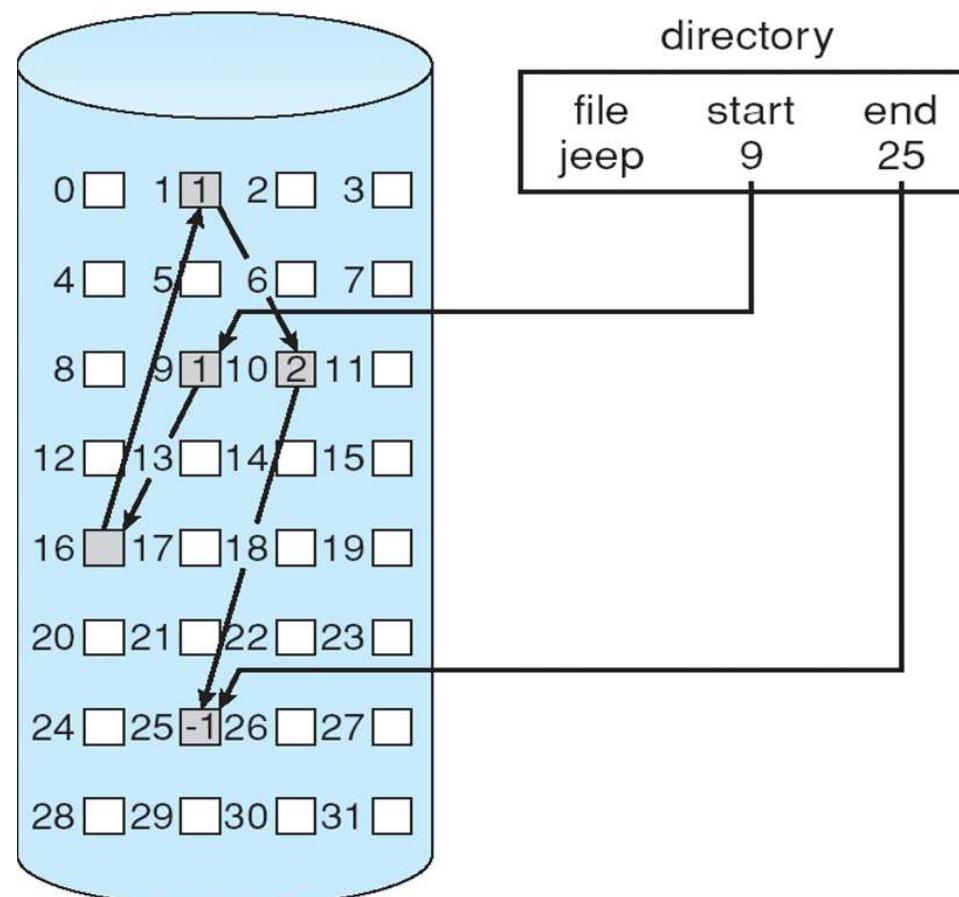
- Nuovi file system (i.e., Veritas File System) usano uno schema di allocazione contiguo modificato
- Extent-based file system allocano blocchi di disco in allocazione estesa
- Un **extent** è un blocco contiguo di memoria
 - Extents sono allocati per allocazione di file
 - Un file consiste di uno o più estensioni
 - ▶ Prima viene allocato un chunk, poi vengono aggiunti altri

Metodi di Allocazione - Concatenata

- **Allocazione Concatenata** – ogni file è una lista linkata di blocchi
 - Ogni blocco contiene un pointer al prossimo blocco
 - File finisce al nil pointer
 - La directory memorizza per ogni file
 - Puntatore al primo e ultimo blocco

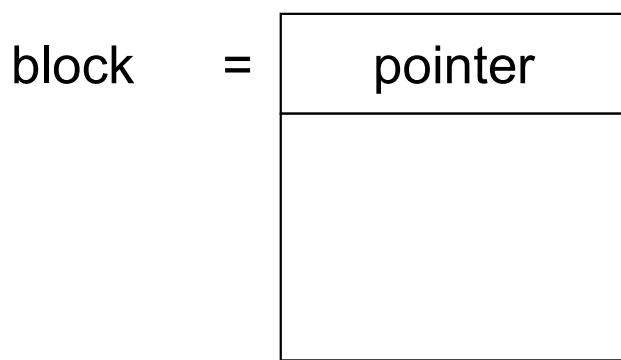
Il puntatore non è visibile all'utente, quindi "prende" spazio:

- Blocco da 512 Byte
- puntatore da 4 Byte
- Spazio utile: 508 Byte



Allocazione Concatenata

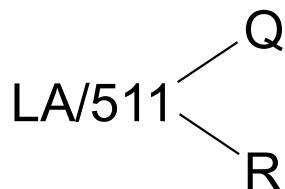
- Ogni file è una lista concatenata di blocchi del disco: i blocchi possono essere sparsi ovunque su disco



Il puntatore non è visibile all'utente, quindi “prende” spazio:

- Blocco da 512 Byte
- puntatore da 4 Byte
- Spazio utile: 508 Byte

Per accesso si fa riferimento al Q-esimo blocco nella lista concatenata che rappresenta il file, quindi si accede riferimento al record nel blocco



Allocazione Concatenata

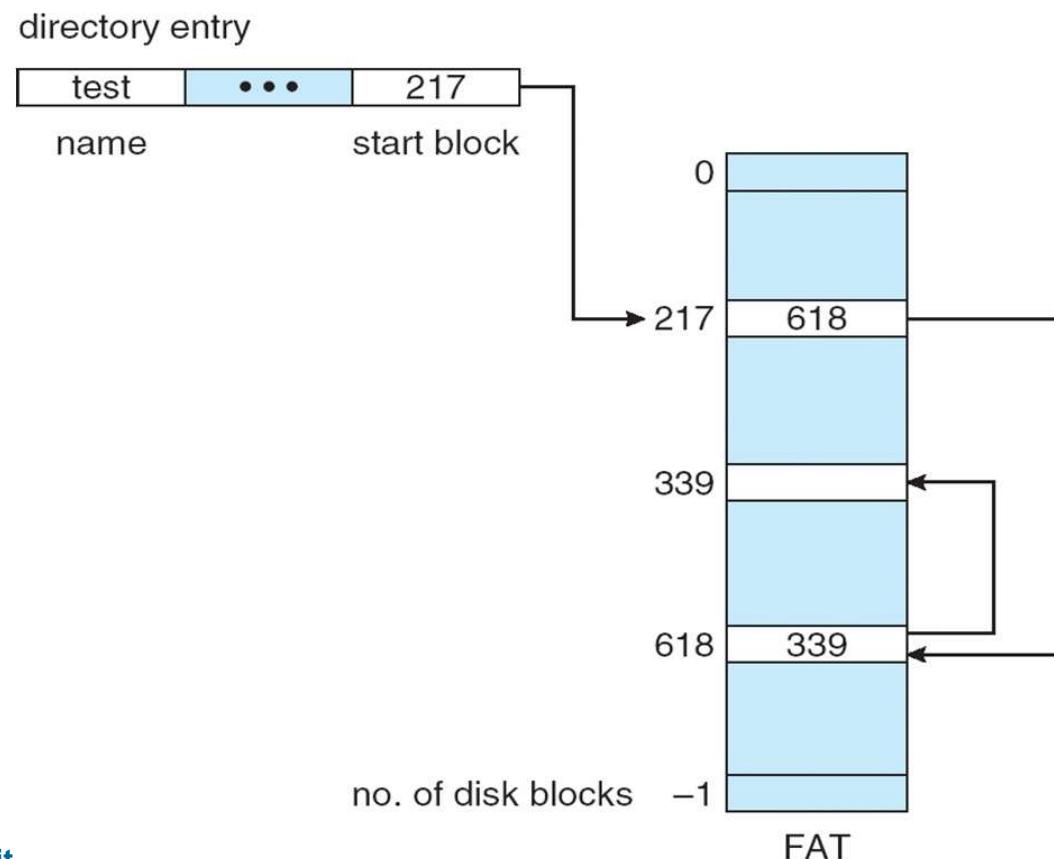
- Creazione di un file:
 - si aggiunge un elemento alla directory con Puntatore al primo blocco settato a null dimensione pari a 0
- Scrittura di un file:
 - Si cerca un blocco libero e si effettua la scrittura dei dati
 - Lo si concatena all'ultimo blocco del file (se presente)
 - Si aggiornano i puntatori della directory all'ultimo e al primo blocco (nel caso il file sia ancora vuoto)
- Lettura di un file:
 - si scorrono in sequenza i blocchi seguendo la concatenazione

Metodi di Allocazione - Concatenata

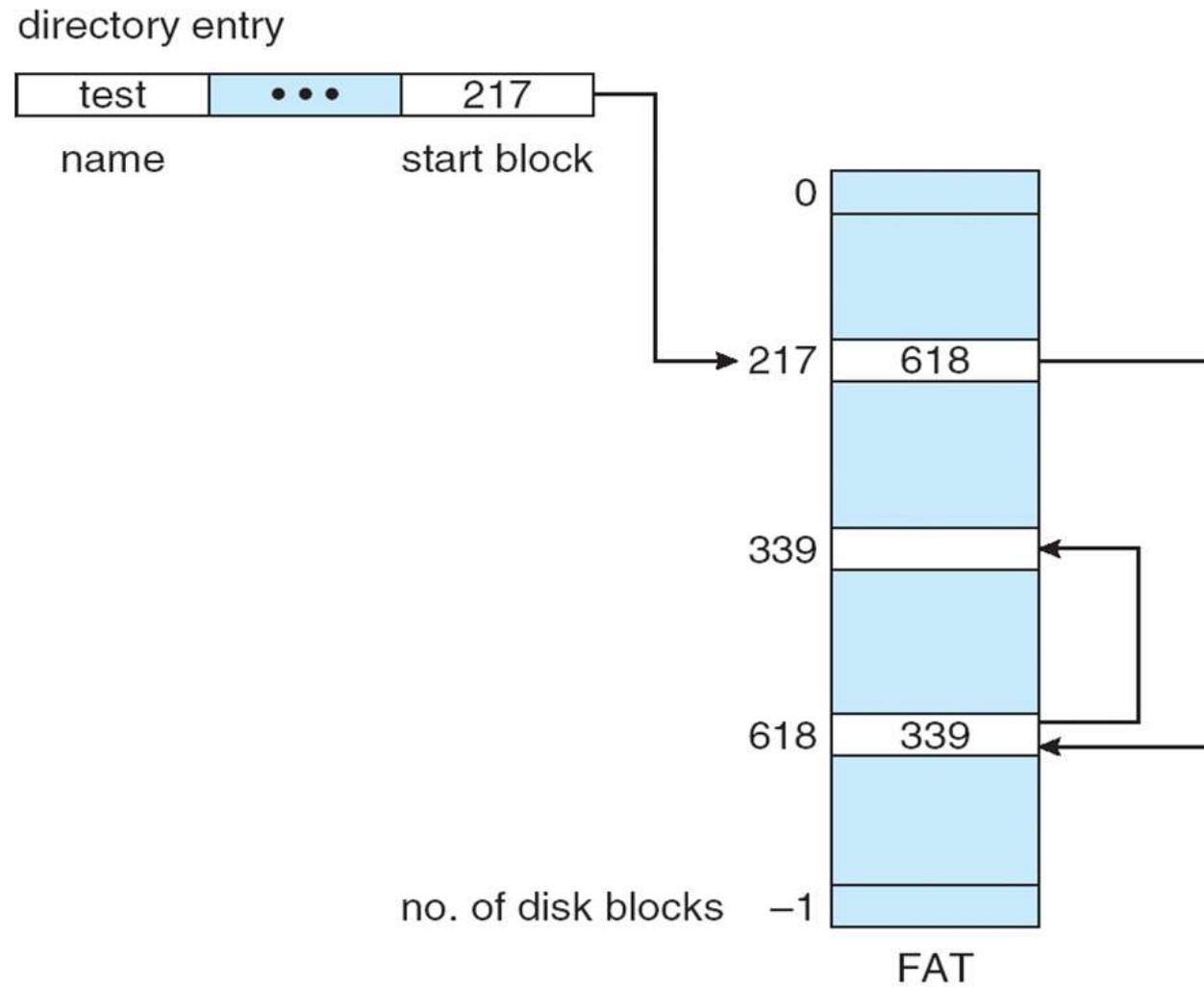
- L'allocazione concatenata risolve i problemi dell'allocazione contigua:
 - Non c'è frammentazione esterna, non serve la compattazione
 - Non è necessario conoscere la dimensione del file al momento della creazione
- Problemi:
 - È efficiente solo per l'accesso sequenziale (a causa dei puntatori)
 - Richiede più spostamenti della testina
 - I puntatori consumano spazio
 - Una soluzione è quella di allocare cluster di blocchi anziché singoli blocchi
 - Meno spazio sprecato per i puntatori, meno spostamenti della testina
 - Aumenta però la frammentazione interna
 - Perdita o danneggiamento di un puntatore
 - Potrebbe puntare ad un blocco vuoto o un blocco di un altro file
 - Possibile soluzione: liste doppiamente concatenate

File-Allocation Table

- Variante importante è FAT (File Allocation Table)
 - Nella prima parte del disco si istanzia un tabella
 - Contenente tanti elementi quanti sono i blocchi
 - Ordinata in base al numero del blocco
 - La directory contiene per ogni file il puntatore al suo primo elemento nella FAT
 - Seguendo i puntatori nella FAT si ricavano gli indici dei blocchi fisici di un file



File-Allocation Table

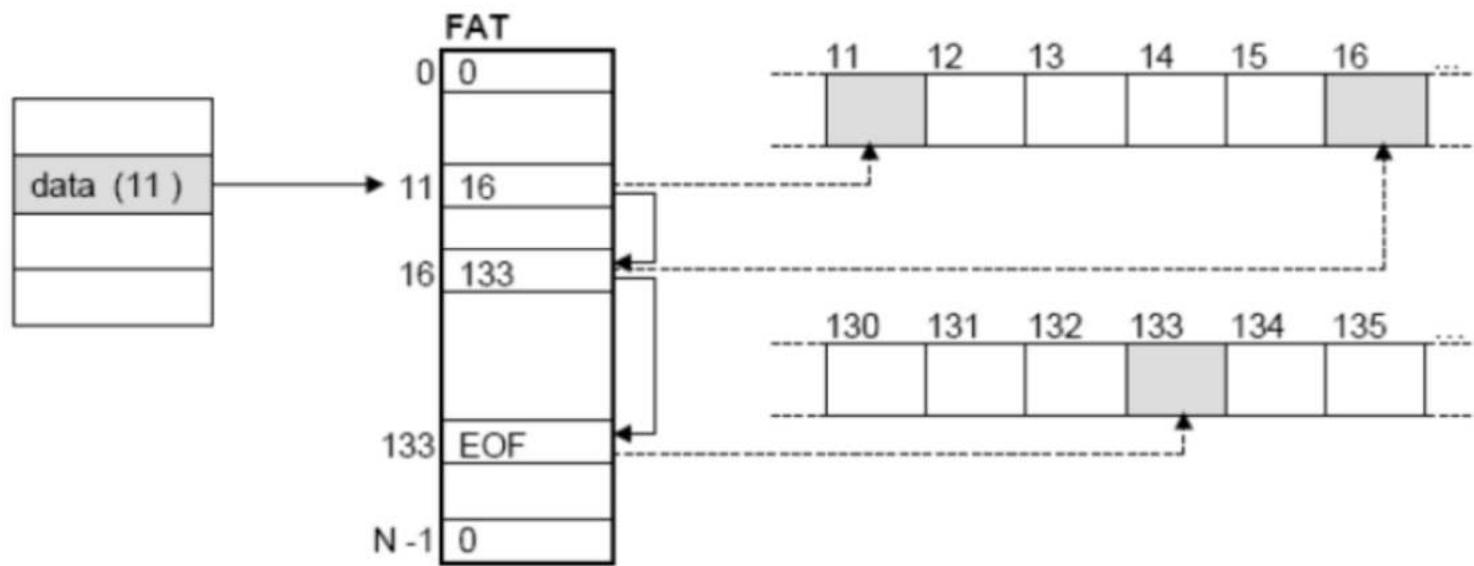


File-Allocation Table

- Variante importante è FAT (File Allocation Table)
 - Nella prima parte del disco si istanzia un tabella
 - Contenente tanti elementi quanti sono i blocchi
 - Ordinata in base al numero del blocco
 - La directory contiene per ogni file il puntatore al suo primo elemento nella FAT
 - Ogni elemento ha un numero di blocco ed è concatenato al blocco successivo nel file
 - L'ultimo elemento contiene un valore end of file
 - L'elemento di un blocco che non è associato a nessun file contiene uno 0
 - Seguendo i puntatori nella FAT si ricavano gli indici dei blocchi fisici di un file

File-Allocation Table

- Variante importante è FAT (File Allocation Table)



File-Allocation Table

- Variante importante è FAT (File Allocation Table)
- Per essere efficiente la FAT richiede l'uso di una cache
 - Altrimenti la testina del disco deve continuamente spostarsi tra l'inizio del disco e la locazione del blocco successivo
- Migliora le prestazioni nel caso di accesso diretto
 - La testina del disco scandisce la FAT e poi si fa uno spostamento grande per accedere al blocco voluto
- La differenza fra FAT12, FAT16 e FAT32 consiste in quanti bit sono allocati per numerare i blocchi del disco
 - Con 12 bit, il file system può indirizzare al massimo $2^{12} = 4096$ blocchi, mentre con 32 bit si possono gestire $2^{32} = 4.294.967.296$ blocchi
 - L'aumento del numero di bit di indirizzo dei blocchi e la dimensione stessa del blocco sono stati resi necessari per gestire unità a disco sempre più grandi e capienti

File-Allocation Table

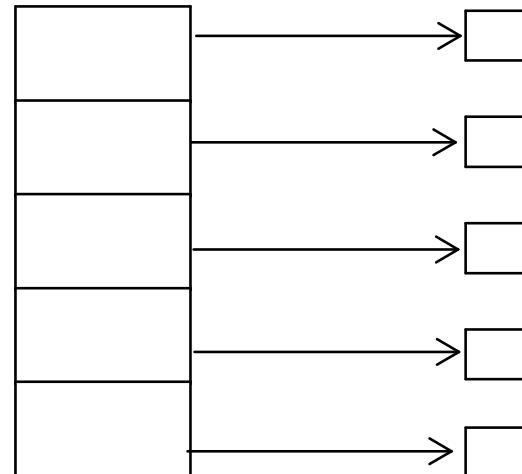
- Variante importante è FAT (File Allocation Table)
- Dimensione del blocco e FAT influenza la dimensione massima della partizione

| Block size | FAT-12 | FAT-16 | FAT-32 |
|-------------------|---------------|---------------|---------------|
| 0.5 KB | 2 MB | | |
| 1 KB | 4 MB | | |
| 2 KB | 8 MB | 128 MB | |
| 4 KB | 16 MB | 256 MB | 1 TB |
| 8 KB | | 512 MB | 2 TB |
| 16 KB | | 1024 MB | 2 TB |
| 32 KB | | 2048 MB | 2 TB |

Metodi di Allocazione - Indicizzata

□ Allocazione indicizzata

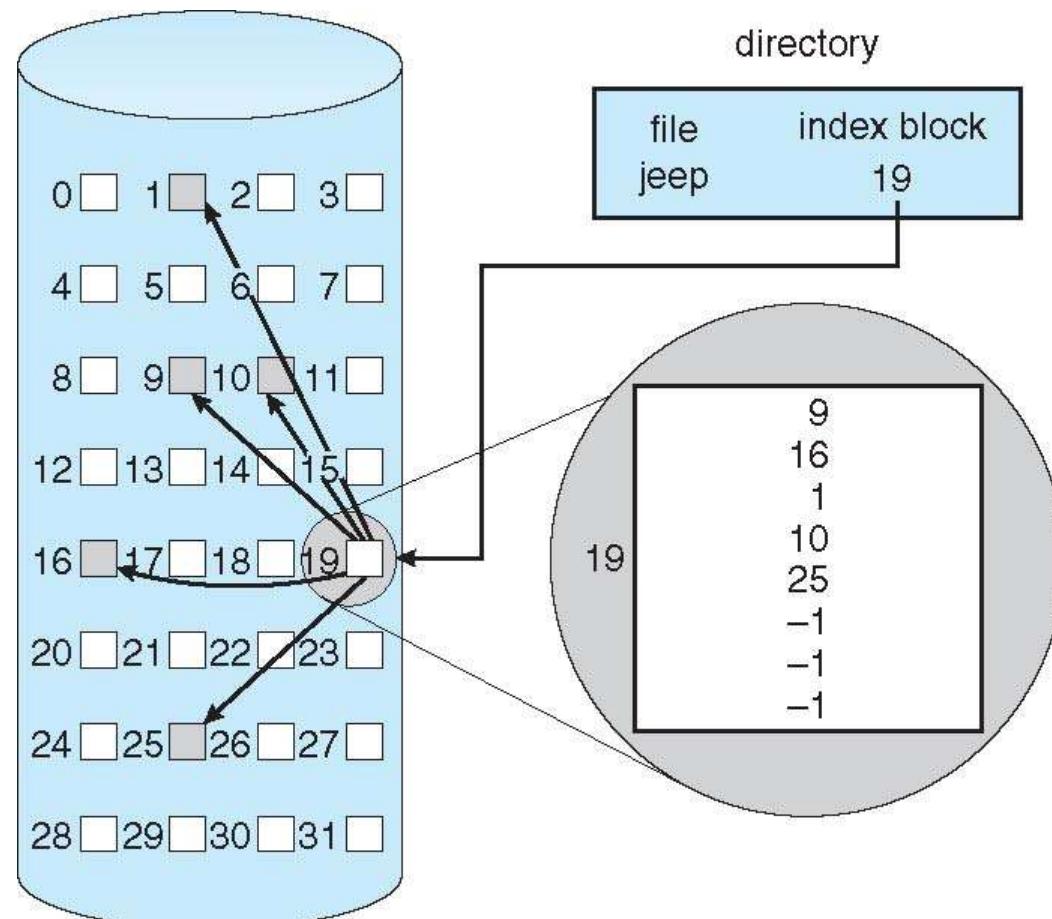
- Ogni file possiede un indice dei blocchi memorizzato in un blocco indice
 - ▶ Questo blocco contiene un array di puntatori agli altri blocchi del file
 - ▶ L'i-simo elemento dell'array punta all'i-simo blocco del file
- La directory memorizza per ogni file un puntatore al suo blocco indice
- Questo risolve il problema dell'accesso diretto presente nell'allocazione concatenata ed evita la frammentazione esterna



index table

Esempio di Allocazione Indicizzata

- Ogni file possiede un indice dei blocchi memorizzato in un blocco indice
 - Questo blocco contiene un array di puntatori agli altri blocchi del file
 - L'i-simo elemento dell'array punta all'i-simo blocco del file
- La directory memorizza per ogni file un puntatore al suo blocco indice



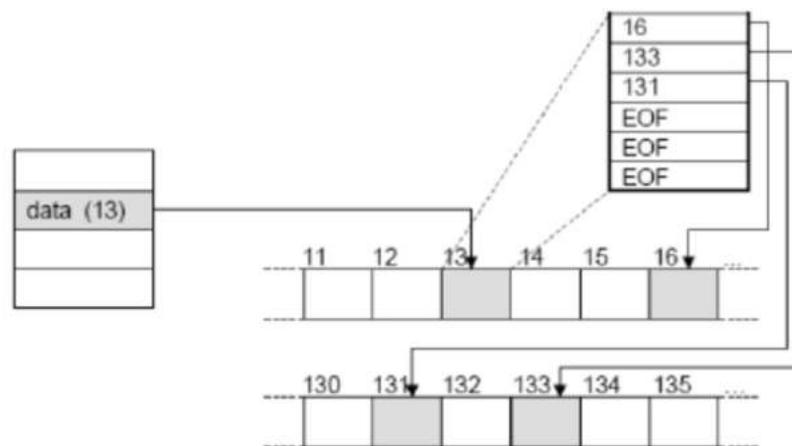
Allocazione Indicizzata

- Creazione di un file:
 - Si crea un blocco indice con tutti gli elementi settati a null
 - Si aggiunge un elemento alla directory con puntatore ad un blocco indice
- Scrittura di un file:
 - Si cerca un blocco libero e si effettua la scrittura dei dati
 - Si aggiorna il blocco indice
- Lettura di un file:
 - si scorrono in sequenza i blocchi seguendo l'array dell'indice

Allocazione Indicizzata

- Problema:

- l'indice richiede l'uso di un intero blocco, quindi nel caso di un file piccolo si ha frammentazione interna

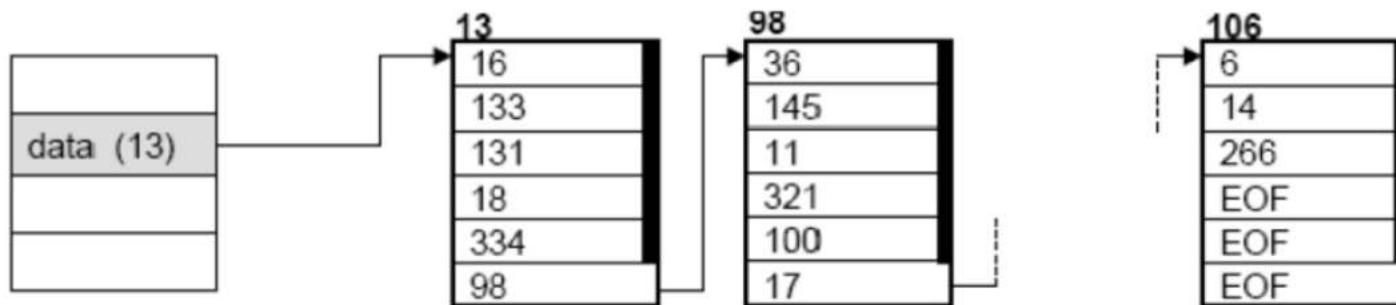


- Si dovrebbero usare blocchi piccoli in modo da ridurre la frammentazione
 - Problemi con i file grandi
 - Diverse soluzioni

Allocazione Indicizzata

□ Soluzioni:

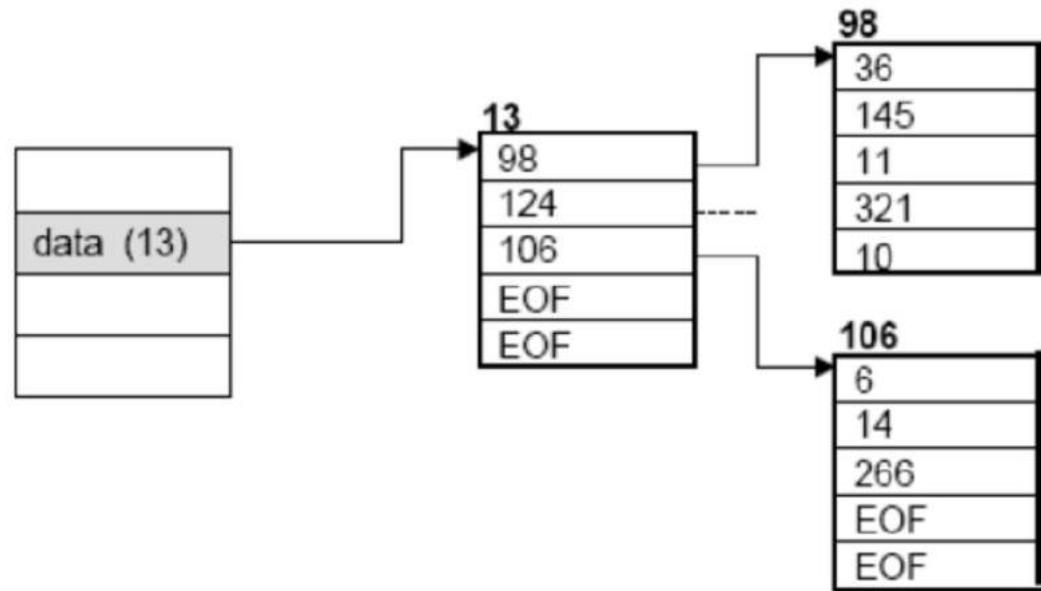
- Schema concatenato: si usano blocchi piccoli e nel caso di file grandi l'indice è composto da più blocchi concatenati



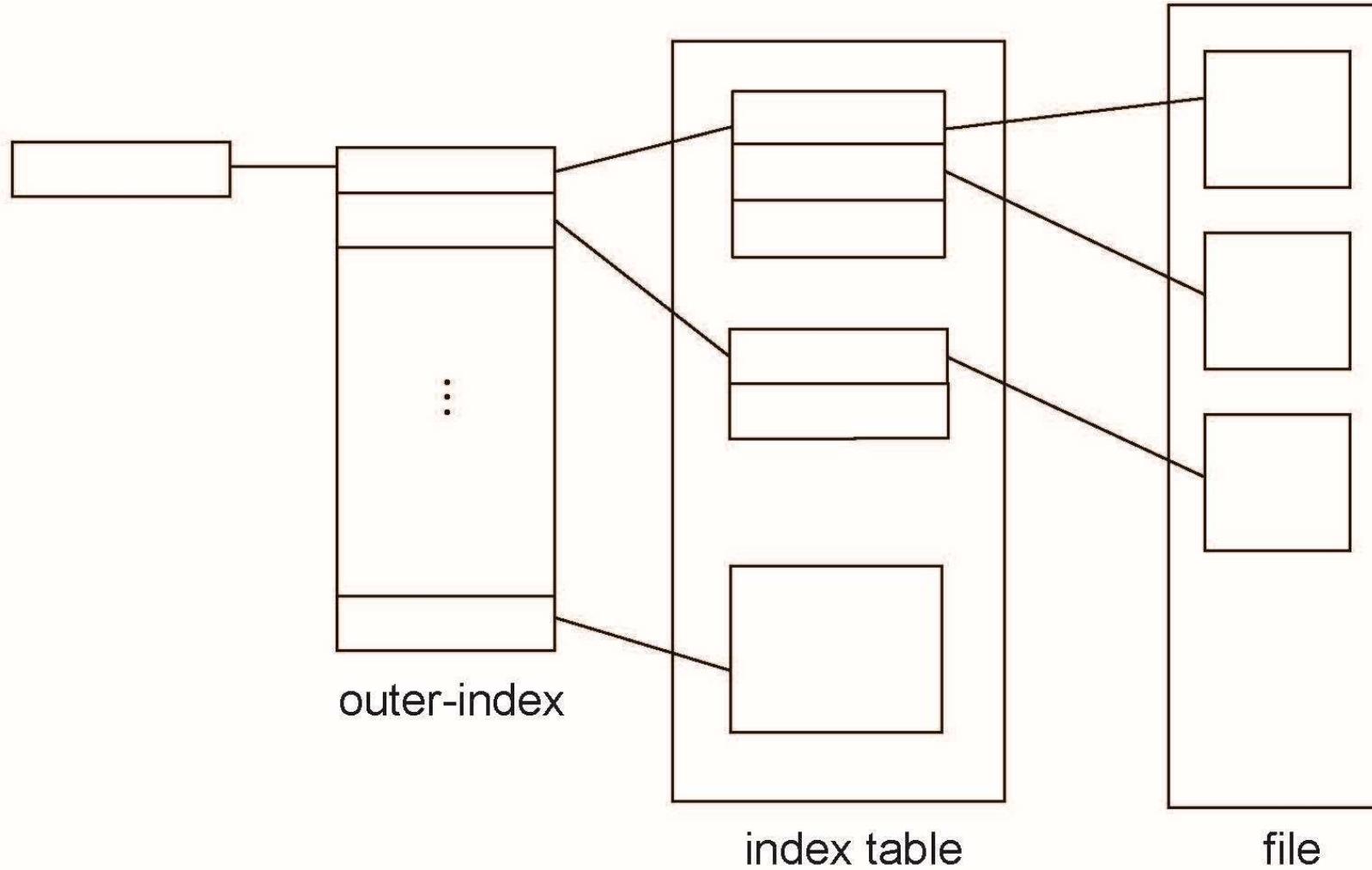
Allocazione Indicizzata

□ Soluzioni:

- Schema a più livelli: si ha un indice di primo livello che punta ad un insieme di indici di secondo livello, i quali puntano ai blocchi del file
- Con blocchi da 4 KByte e puntatori da 4 Byte si possono indicizzare file da 4GB
- È possibile estendere lo schema a 3-4 livelli



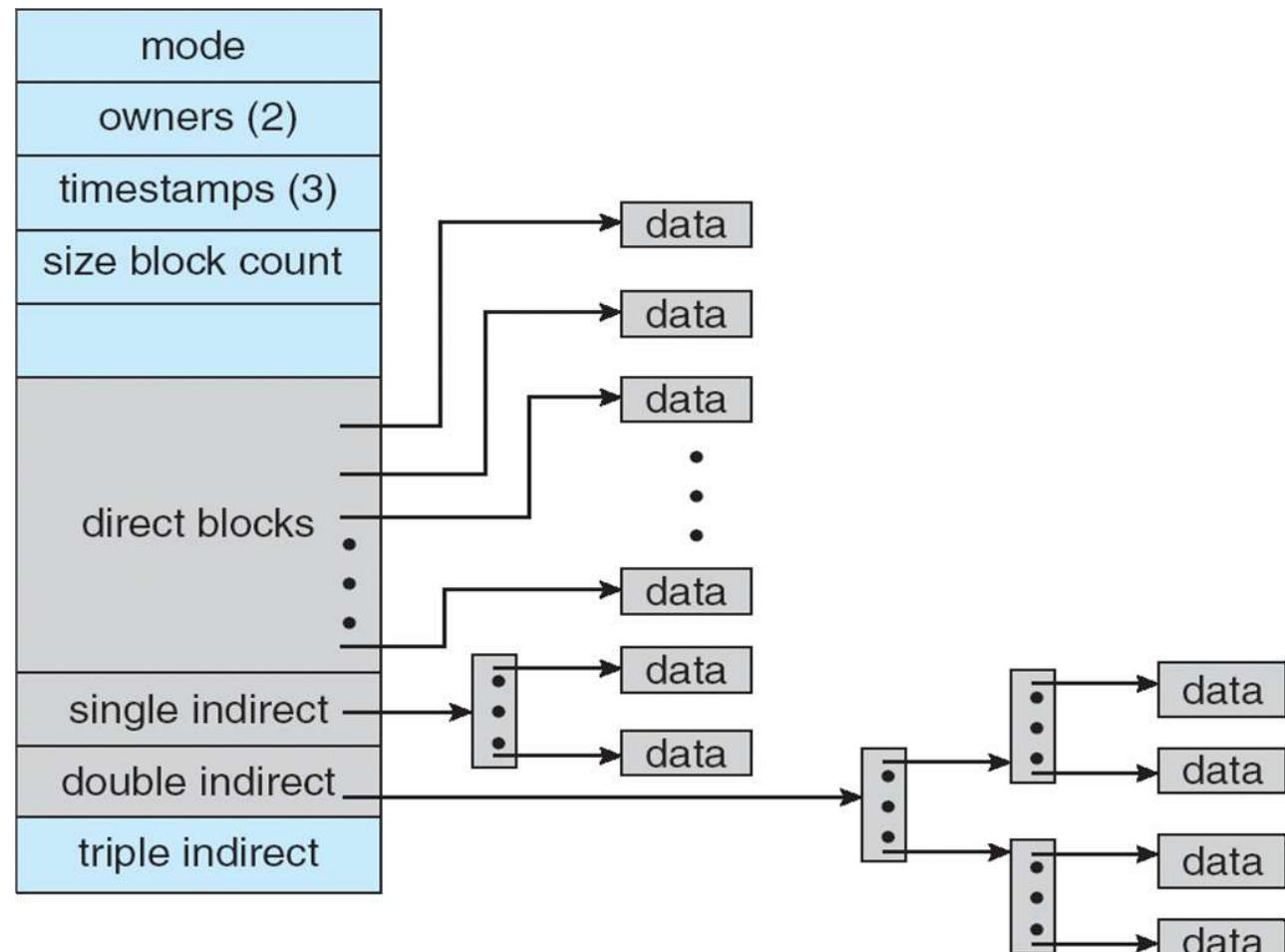
Allocazione indicizzata – Mapping



Schema Combinato: UNIX UFS

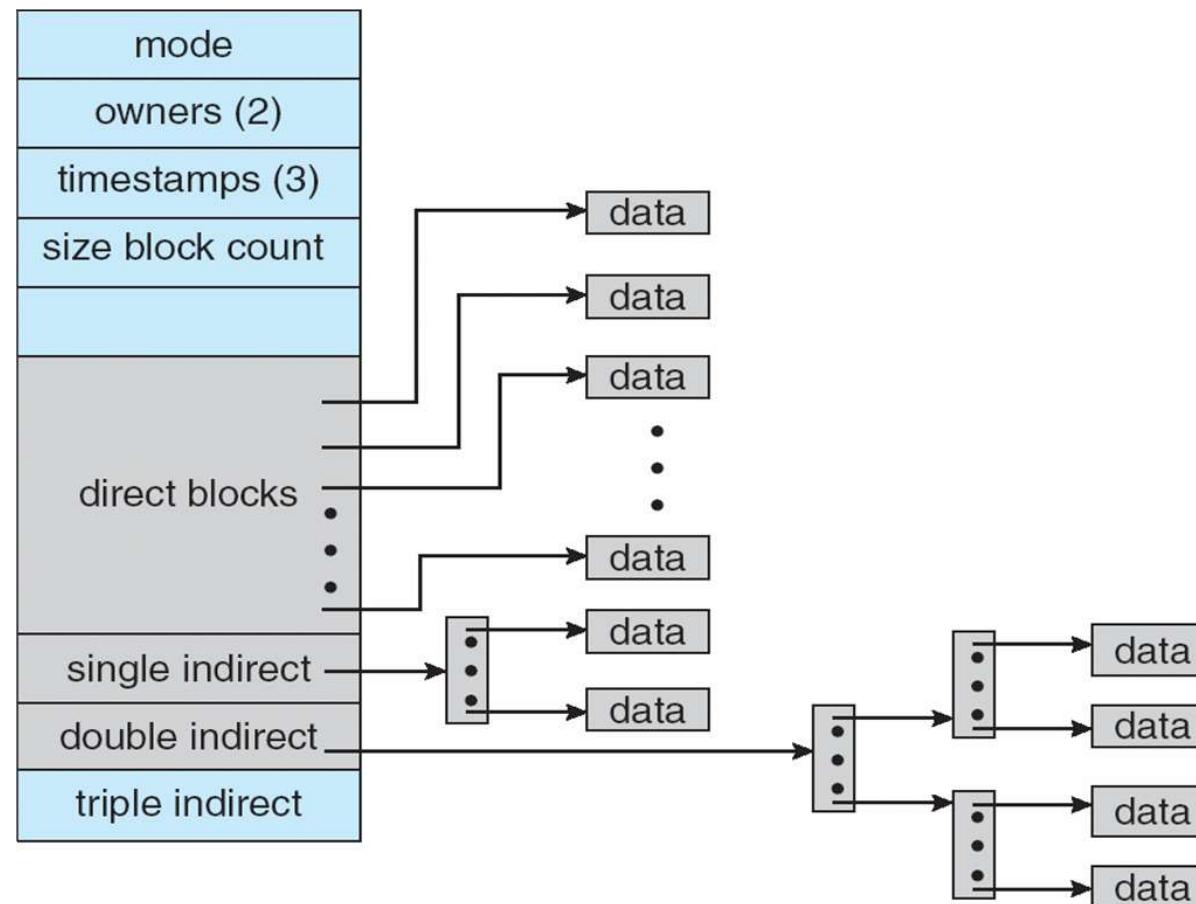
Schema combinato: usato nei sistemi Unix:

- Il FCB contiene 15 elementi per indicizzare i file
- I primi 12 sono puntatore diretti a dei blocchi
- Il 13° punta ad un blocco contenente un indice
- Il 14° punta ad un indice a due livelli
- Il 15° punta ad un indice a tre livelli



Schema Combinato: UNIX UFS

4K bytes per blocco, indirizzi a 32-bit



Più blocchi indicizzati di quanti possono essere indirizzati con puntatori di file a 32-bit

Scelta Metodo Allocazione

- La scelta del metodo di allocazione deve considerare due fattori:
 - Efficienza di memorizzazione
 - Tempo di accesso ai dati
- La scelta è influenzata dalla modalità di accesso al file
 - L'allocazione contigua richiede un solo accesso al disco qualsiasi sia il tipo di accesso (sequenziale o diretto)
 - Anche con l'accesso diretto, noto l'indirizzo del primo blocco fisico e l'indirizzo del blocco logico desiderato, è possibile calcolare l'indirizzo fisico di accesso
 - L'allocazione concatenata invece è efficiente per l'accesso sequenziale, ma non per quello diretto
 - Accesso all'i-esimo blocco porta all'accesso di i blocchi intermedi

Scelta Metodo Allocazione

- Per questo motivo alcuni sistemi gestiscono
 - File ad accesso sequenziale con allocazione concatenata
 - File ad accesso random con l'allocazione contigua
 - È necessario specificare a priori la dimensione del file
 - È sempre possibile convertire il formato del file
 - Le prestazioni dell'allocazione indicizzata invece dipendono
 - Dalla profondità dell'indice (quanti blocchi indice)
 - Dalla dimensione del file (se grande occorre scorrere molti blocchi indice)
- Alcuni sistemi usano
 - Allocazione contigua per file piccoli
 - Allocazione indicizzata per file grandi

Gestione dello Spazio Libero

- Per trovare in modo veloce un insieme di blocchi da allocare ad un file il Sistema Operativo tiene traccia dei blocchi non allocati
 - Lista dello spazio libero
- Creazione di un file:
 - Si cerca nella lista il numero di blocchi necessari
 - Si allocano al file
 - Si rimuovono dalla lista
- Eliminazione di un file
 - I blocchi associati al file vengono deallocati
 - Si aggiungono alla lista

Gestione dello Spazio Libero

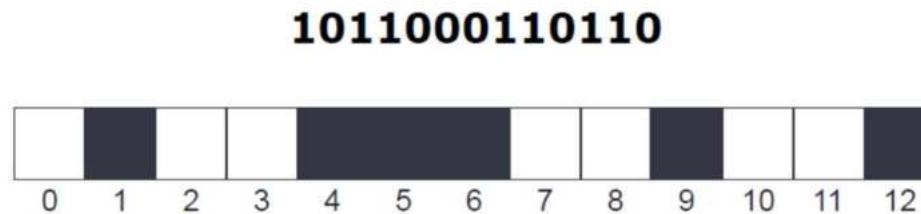
- Diverse possibili implementazioni:
 - Vettore di bit
 - Lista concatenata
 - Raggruppamento
 - Conteggio

Vettore di Bit o BitMap

- Si memorizza un array di lunghezza pari al numero dei blocchi
- Il valore delle celle indica la disponibilità di un blocco
 - 1 libera
 - 0 occupata
- Per creare un file
 - Ricerca del primo bit a 1 (allocazione concatenata o indicizzata)
 - Ricerca di un blocco di bit a 1 sufficientemente grande (allocazione contigua)
- Pro: semplice ed efficiente
- Contro:
 - Per essere efficiente deve però essere mantenuta in memoria centrale
 - Dischi grandi richiedono vettori di bit molto grandi

Vettore di Bit

- I calcolatori forniscono supporto HW
- Istruzioni per trovare lo scostamento del primo bit a 1 in numero di parole
- Parole_0 fatte di zero bit, si devono scorrere
- Se l'array è diviso in parole di n bit:
 - indice_blocco = (n * num_parole_0) + scostamento_primo_bit_1
 - ▶ Num_parole_0 è il numero di parole consecutive con tutti i bit a 0 partendo dalla prima parola

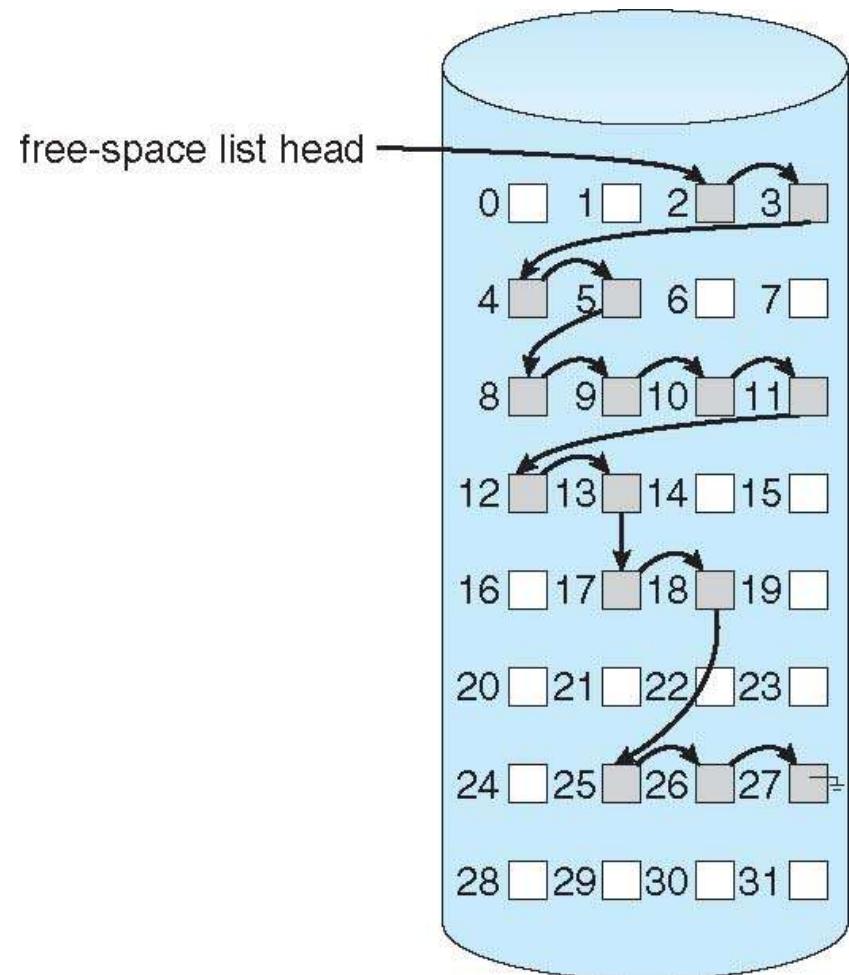


Vettore di Bit

- Bit map richiede spazio extra
 - Esempio:
 - block size = 4KB = 2^{12} bytes
 - disk size = 2^{40} bytes (1 terabyte)
 - $n = 2^{40}/2^{12} = 2^{28}$ bits (o 32MB)
 - se cluster di 4 blocchi -> 8MB of memory
- Dischi aumentano di dimensioni quindi aumenta il problema

Liste Concatenate

- Si memorizza in memoria centrale un puntatore al primo blocco libero
- Ogni blocco libero viene poi concatenato al successivo
- Contro: è poco efficiente in quanto scorrere l'intera lista richiede molti accessi alla memoria
- Fortunatamente lo scorrimento della lista è un evento raro
- Il SO si limita a cercare il primo blocco libero per allocarlo ad un file
- Allocato il blocco si aggiorna il puntatore al primo blocco libero in memoria centrale



Raggruppamento e Conteggio

□ Raggruppamento

- Modifica dell'approccio a free-list
- Il primo blocco contiene gli indirizzi di n blocchi liberi
 - ▶ Al primo accesso subito info sui blocchi liberi
- n-1 sono effettivamente liberi
- L'ultimo contiene a sua volta n indirizzi di blocchi liberi

□ Conteggio

- Sfrutta il fatto che tipicamente si ha più di un blocco contiguo libero
- Si usa un array in cui ogni elemento contiene
 - ▶ Indirizzo del primo blocco libero
 - ▶ Conteggio degli n blocchi contigui liberi

Space Maps

□ Space Maps

- Usato in **ZFS** (Oracle) per gestire grandi quantità di file e directory
- Gestisce meta-data per file system molto grandi
 - ▶ bit maps non potrebbero stare in memoria
 - ▶ migliai di I/Os per allocare e liberare grandi quantità di memoria
- Divide lo spazio del dispositivo in unità **metaslab** e le gestisce
 - ▶ Un volume può contenere centinaia di metaslab
- Ogni metaslab ha associate una space map
 - ▶ Usa algoritmi di counting
- Registra le info su log file piuttosto che nel file system
 - ▶ Log di tutte le attività del blocco, in ordine di tempo, formato di conteggio
- Metaslab attività -> carica la space map in memoria in una struttura ad albero bilanciato
 - ▶ Fa update dell'albero
 - ▶ Compatta blocchi liberi in una singola entry
 - ▶ La free list è ottenuta da log + albero

Efficienza e Prestazioni

- I dischi sono tipicamente conosciuti come il collo di bottiglia di ogni sistema
 - A causa della loro velocità di accesso
 - Anche NVM sono lente rispetto a CPU
- Gli algoritmi di allocazione dei blocchi e gestione delle directory devono essere scelti con cura al fine di ottimizzare
 - Efficienza nell'uso dei dischi
 - Prestazioni

Efficienza

- Diverse scelte influenzano l'efficienza del disco

- Allocazione dei FCB:

- alcuni SO, es. Unix, allocano preventivamente i FCB e li distribuiscono nel file system
 - Anche un disco senza file ha una certa quantità di spazio occupata dai FCB
 - Questa scelta può migliorare le prestazioni del file system

- Scelta degli attributi da memorizzare per ogni file:

- Salvare le date di accesso ai file può essere utile per fornire informazioni all'utente
 - Tuttavia questo richiede due accessi al disco ogni volta che si legge un file
 - Uno in lettura e uno in scrittura per modificare la directory
 - Gli attributi da memorizzare vanno scelti con cura

- Lunghezza dei puntatori:

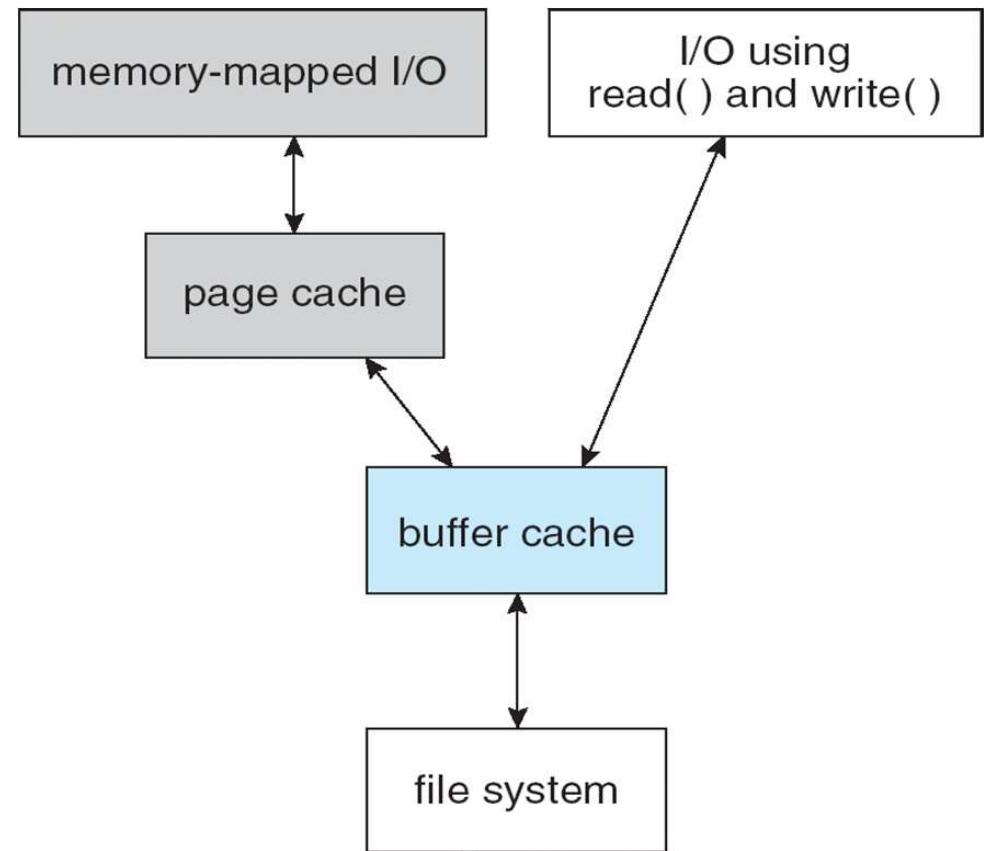
- Più bit si usano per i puntatori più grandi possono essere i file ($32 \text{ è bit } 2^{32} = 4\text{GB}$)
 - Tuttavia puntatori grandi richiedono più spazio per eseguire gli algoritmi di allocazione e gestione dello spazio libero (è spazio non usabile dall'utente)

Prestazioni

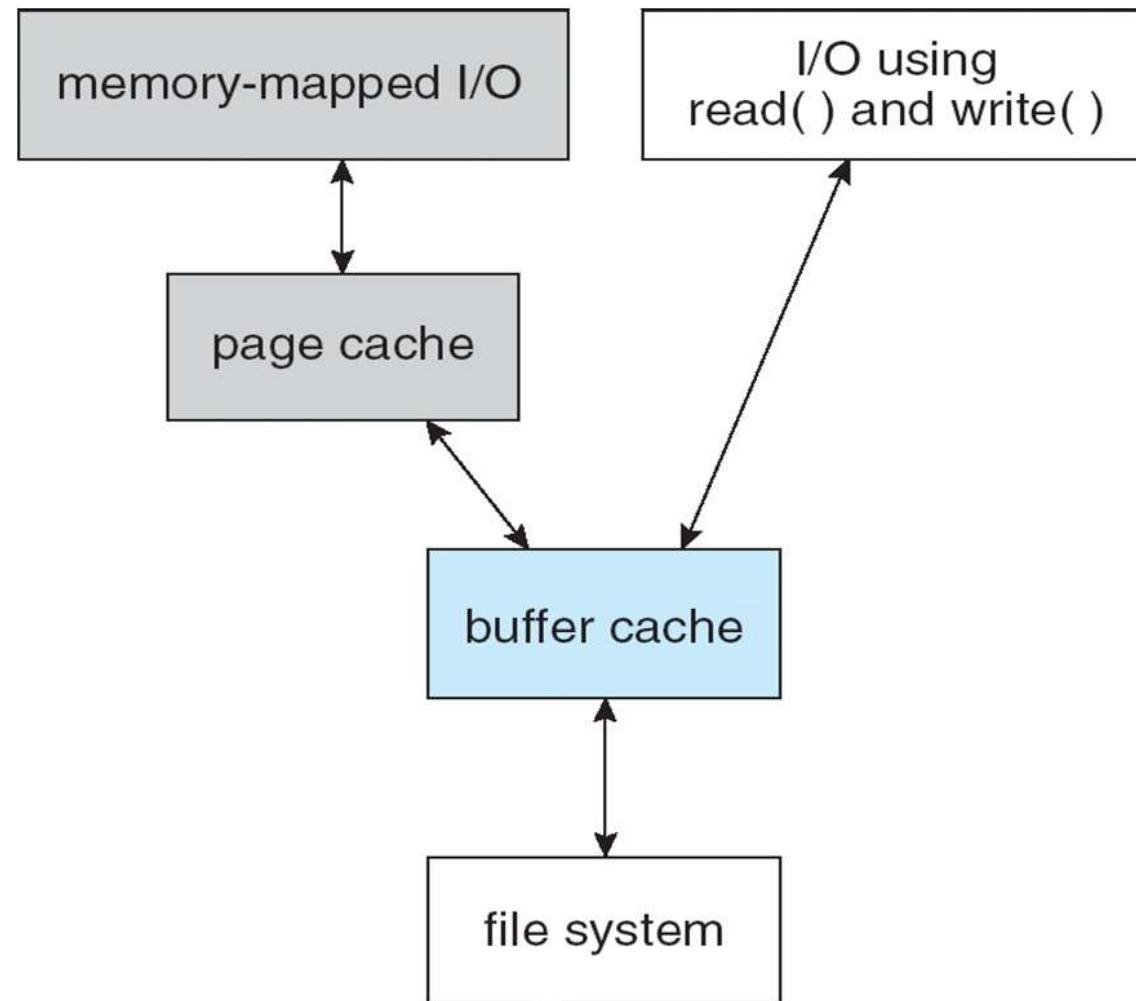
- Una volta scelti gli algoritmi per la gestione del file system si possono adottare diverse tecniche per migliorare le prestazioni
- Cache del disco (buffer cache): area riservata della memoria centrale dove vengono mantenuti i blocchi usati di recente
 - Probabilmente verranno riusati a breve
- Cache delle pagine: si usano tecniche di memoria virtuale non solo per la gestione dei processi ma anche per la gestione dei file
 - Buffer cache unificata: è utile per gestire in modo efficiente l'accesso ai file
 - Sia tramite le chiamate di sistema read() e write()
 - Che tramite la mappatura dei file in memoria

Page Cache

- Se non si usa la cache unificata
- Le chiamate di sistema usano direttamente la cache buffer cache
- Il sistema di memoria virtuale non può interfacciarsi direttamente con la buffer cache
- È necessario copiare nella cache delle pagine il contenuto del file presente nella buffer cache
- Fenomeno del double caching
- Doppio passaggio di cache
- Spreco di memoria, cicli di CPU e di I/O



I/O Without a Unified Buffer Cache

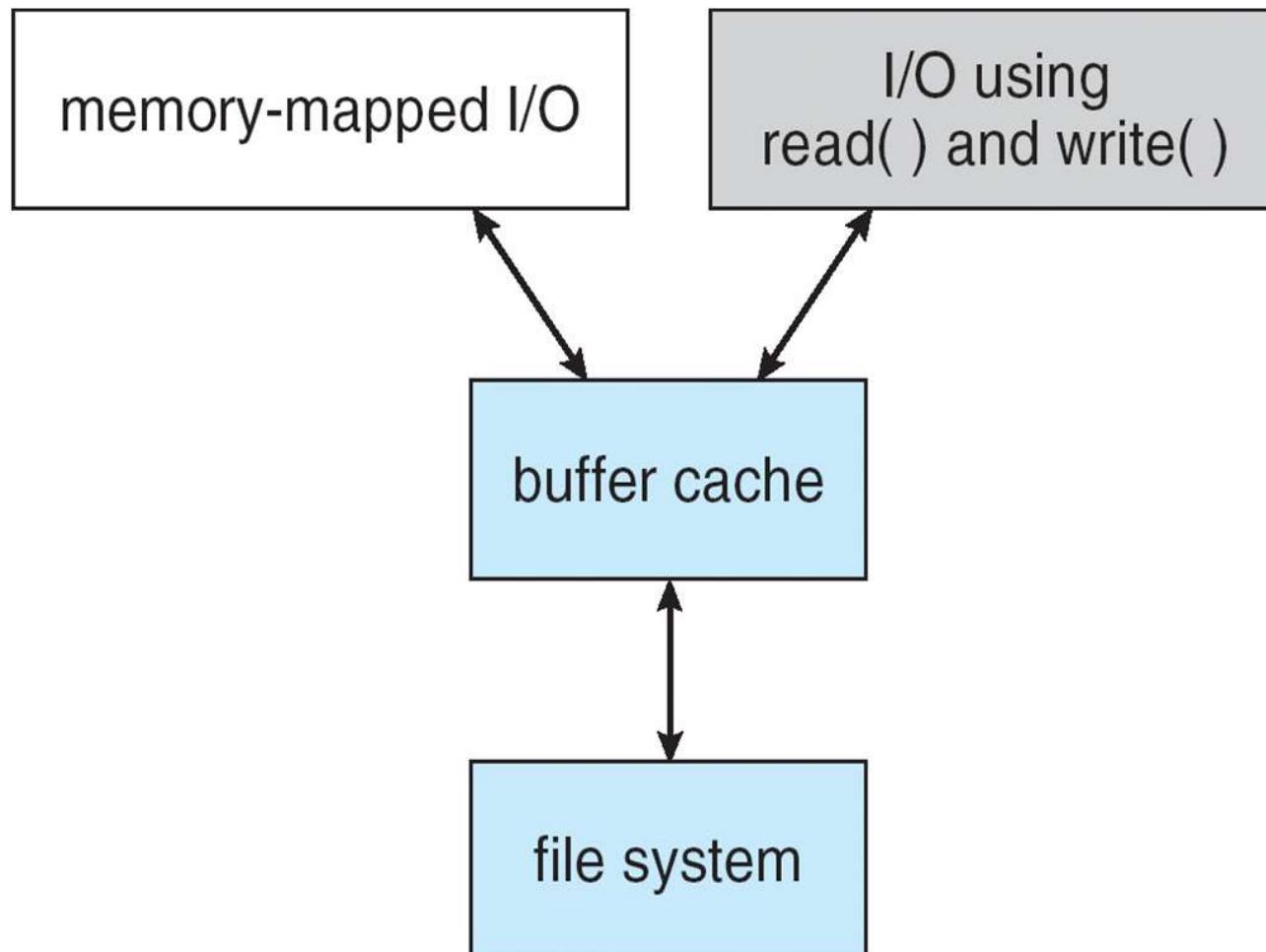


Unified Buffer Cache

- A **unified buffer cache** uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid **double caching**
- But which caches get priority, and what replacement algorithms to use?

Buffer Cache Unificata

- Con la cache unificata il problema del double caching viene risolto
- Il sistema di memoria virtuale può interfacciarsi direttamente con la cache del file system



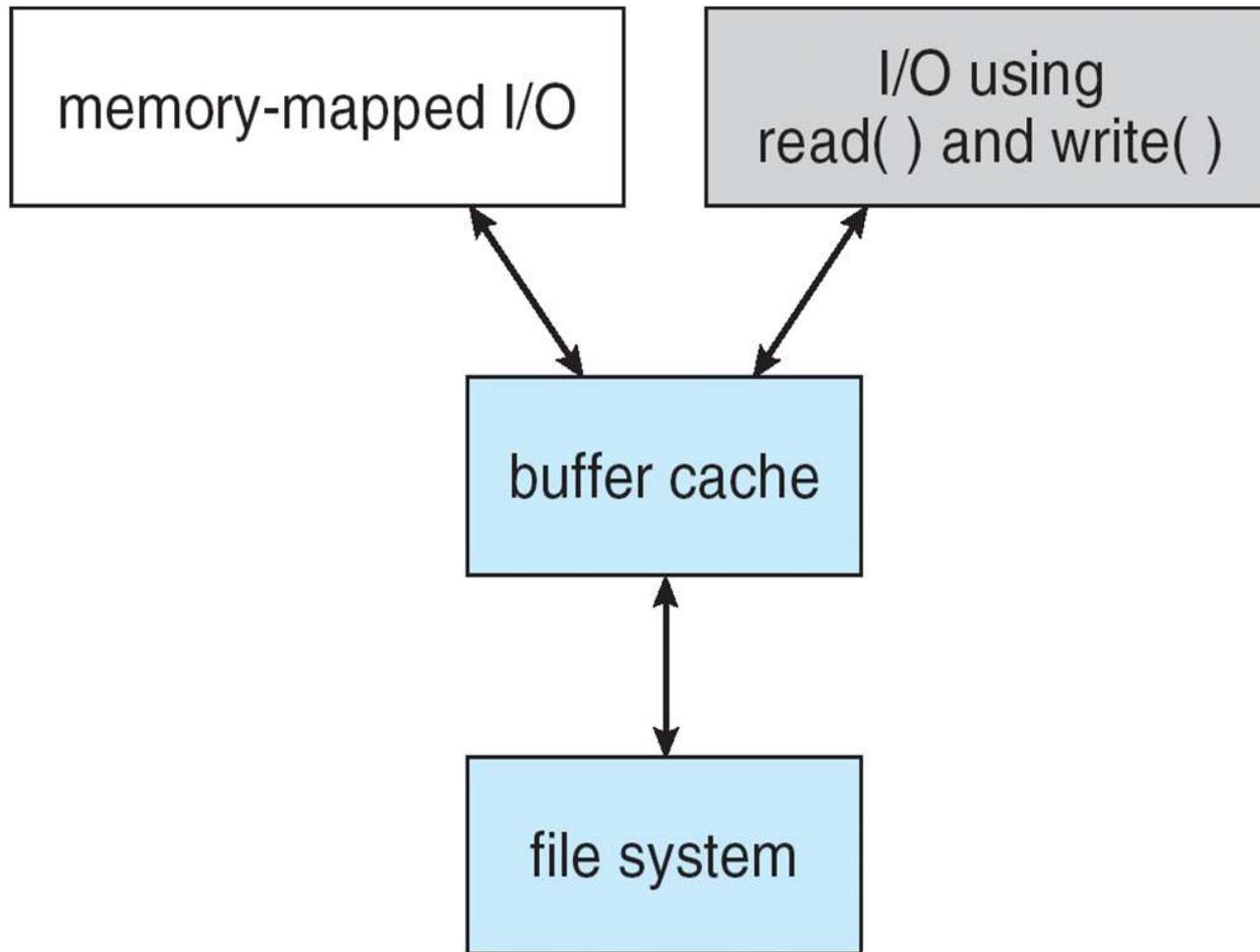
Scritture

- Scritture sincrone e asincrone:
 - Influenzano le prestazioni di I/O
- Scrittura sincrona: le scritture su disco avvengono nell'ordine in cui il driver riceve le richieste
 - Il chiamante è bloccato fino a quando il dato non viene scritto
- Scritture asincrone: le scritture avvengono sulla cache del disco
 - Il driver del dispositivo si occupa poi di trasferire i dati sul disco
 - Il chiamante non resta bloccato
- Le scritture asincrone sono usate più frequentemente
 - Per i metadati le scritture sono spesso sincrone

Sostituzione Pagine

- Algoritmo di sostituzione della pagine:
 - Il metodo di accesso al file influenza la scelta dell'algoritmo di sostituzione delle pagine
 - In caso di accesso sequenziale LRU non è ottimale
 - La pagina usata più recentemente è quella che sarà usata più in la nel tempo
 - Rilascio indietro:
 - Si rimuove la pagina dalla memoria quando viene fatto un riferimento alla pagina successiva
 - Lettura anticipata:
 - Si copiano nella cache la pagina richiesta ed n pagine successive
 - Utilizzando queste tecniche si ha un notevole risparmio di tempo

I/O Using a Unified Buffer Cache



Recupero

- Operazioni eseguite molto frequentemente, come la creazione di un file, comportano molti cambiamenti nelle strutture dati del file system
 - Struttura della directory
 - Lista dello spazio libero
 - FCB
- Se un crollo del sistema avviene prima che tutte queste operazioni siano completate possono crearsi delle incoerenze
- Es., un nuovo FCB potrebbe essere stato allocato ma la struttura delle directory potrebbe non contenere il puntatore ad esso
- Il SO adotta diverse tecniche per risolvere le incoerenze

Controllo di Consistenza

- Per scoprire eventuali errori è necessario analizzare i meta-dati
- Questo richiede molto tempo
 - Prima di modificare i meta-dati il SO mette un bit di modifica a 1
 - Se le modifiche terminano con successo il bit viene rimesso a 0
 - Se resta ad un 1 significa che c'è stato un crollo e al riavvio viene eseguito il verificatore della coerenza
- Confronta i meta-dati con i blocchi del disco per correggere eventuali errori
 - Con l'allocazione concatenata i puntatori permettono di ricostruire un file
 - ▶ Si può ricreare l'elemento della directory
 - Con l'allocazione ad indicizzazione la perdita dell'indice è grave
 - ▶ I blocchi infatti non contengono informazioni sugli altri blocchi del file

Registrazione Modifiche (log)

- La verifica della coerenza comporta alcuni problemi
 - Non sempre le incoerenze sono risolvibili
 - In alcuni casi la risoluzione richiede l'intervento umano
 - Richiede molto tempo
- Si adottano degli algoritmi usati nelle basi di dati
 - Si mantiene un log dove vengono salvate in modo sequenziale le modifiche ai metadati
 - ▶ [log-based transaction-oriented file systems \(journaling\)](#)
 - Quando si opera su un file, l'operazione è considerata committed quando
 - ▶ Tutte le modifiche da apportare ai meta-dati sono state salvate nel log (transazione)
 - La chiamata di sistema restituisce il controllo all'utente
 - SO si occupa di aggiornare in modo asincrono i meta-dati come indicato dal log
 - Quando tutti i metadati sono stati aggiornati le modifiche vengono tolte dal log
 - ▶ Implementato come un buffer circolare (sovrascrive i vecchi valori)
 - Usato NTFS, Veritas, UFS su Solaris, ext3, ext4, ZFS

Registrazione Modifiche (log)

- Se il sistema crolla al momento del riavvio
 - Si analizza il log e si eseguono tutte le transazioni registrate
 - queste modifiche non erano ancora state fatte sui metadati
- Problema quando la transazione non è committed al momento del crash
 - Tutti i cambiamenti fatti al FS annullati per mantenere consistenza
 - ▶ unico ripristino necessario
- Il log è tipicamente salvato in un'area dedicata del disco
 - Accedere in modo sequenziale e sincrono al log per salvare le transazioni
 - ▶ È più veloce che accedere in modo diretto e sincrono ai blocchi contenenti i meta-dati
- Questa tecnica diminuisce il tempo in cui un processo resta bloccato in attesa dell'aggiornamento dei metadati

Altri Metodi

- Un'alternativa consiste nel non sovrascrivere i metadati (WAFL e Solaris ZFS)
- Quando è necessario aggiornarli
 - Si scrivono i nuovi valori in un nuovo blocco del disco
 - Si aggiorna il puntatore ai metadati
 - Si dealloca il blocco contenente i vecchi meta-dati

Backup e Recupero

- Le tecniche viste finora permettono di recuperare eventuali inconsistenze fra metadati e file
- Esistono anche delle tecniche che permettono di recuperare i file in caso di guasti irreparabili ai dischi (backup e ripristino)
- Il backup può essere
 - Assoluto: ogni volta si copia l'intero disco
 - Incrementale: la prima volta si copia l'intero disco, le volte successive solo i file modificati
 - ▶ Si basa sui valori degli attributi
 - ▶ Se `data_ultima_modifica > data_backup` è nuovo backup del file
- Il metodo di ripristino varia in base al tipo di backup

Backup e Recupero

- Giorno 1. Copia in un mezzo di backup tutti i file del file system
 - This is called a full backup.
- Giorno 2. Copia su un altro mezzo tutti i file cambiati dal giorno 1.
 - This is an incremental backup.
- Giorno 3. Copia su altro mezzo tutti i file cambiati dal giorno 2.
 - ...
- Giorno N. Copia su altro mezzo tutti i file cambiati dal giorno N-1.
 - vai al giorno 1.

- Per diminuire il numero di mezzi che devono essere letti per un ripristino:
 - eseguire un backup completo
 - poi ogni giorno eseguire il backup di tutti i file che sono stati modificati dal backup completo.

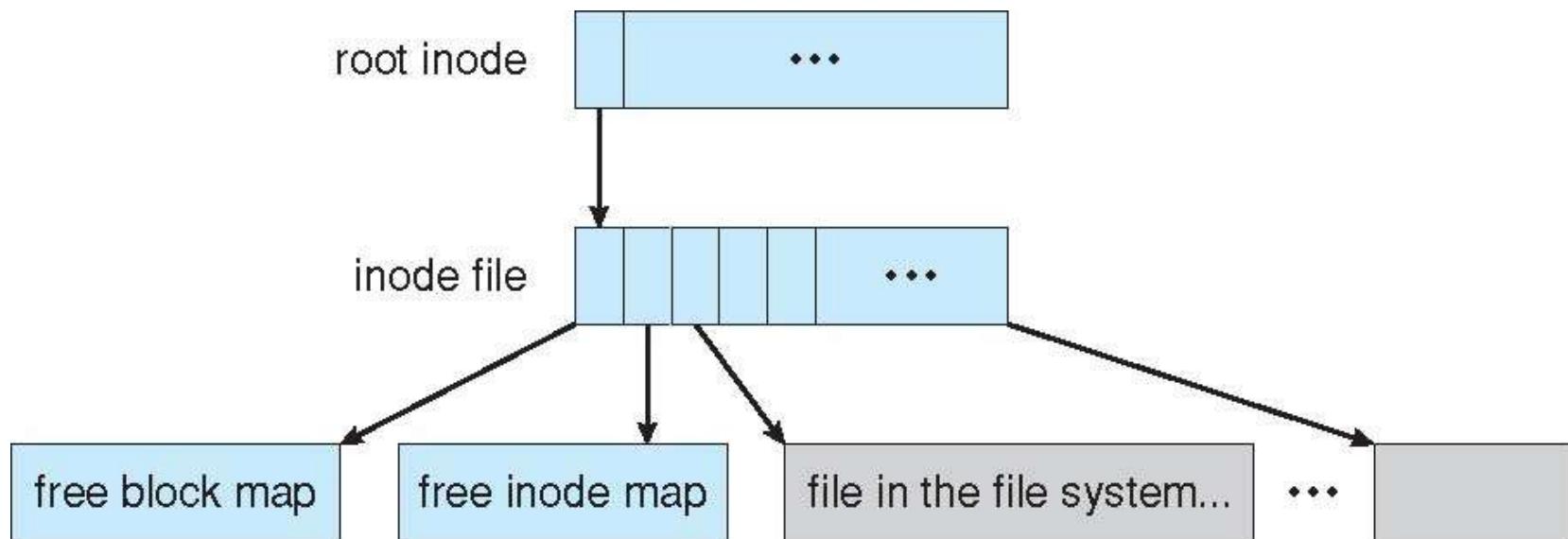
Esempio: WAFL File System

- “Write-anywhere file layout”
- Ottimizzato per random write
- Da utilizzare su file system distribuito, su file server di rete (di NetApp)
- Interagisce con protocolli (NFS, CIFS, http, ftp)
- Se molti client fanno richieste di file gli accessi sono random read/write
- Le letture sono mediate da cache, problema fondamentale è scrittura
- Simile al Berkeley Fast File System, con diverse modifiche
 - È basato su blocchi
 - Usa gli inode per descrivere i file
 - Ogni inode ha 16 puntatori di blocchi
 - Tutti i metadata in un file
 - Tutti gli inodes sono in un file, la free-block map in un altro, e la free-inode map in un terzo

WAFL File Layout

Simile al Berkeley Fast File System, con diverse modifiche

- Usa gli inode per descrivere i file
- Ogni inode ha 16 puntatori di blocchi
- Tutti i metadata in un file
- Tutti gli inodes sono in un file, la free-block map in un altro, e la free-inode map in un terzo

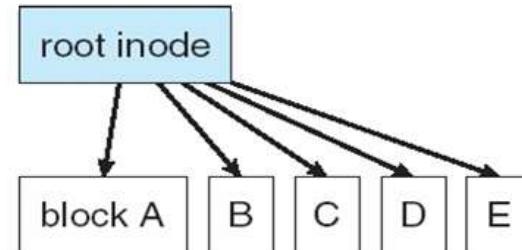


Snapshot in WAFL

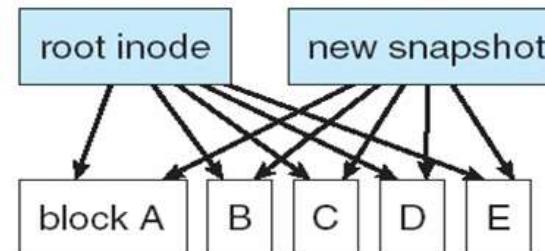
Per fare una istantanea (snapshot) del FS, WAFL crea una copia del root inode. Ogni aggiornamento di file o metadata dopo lo snapshot va in nuovi blocchi e non sovrascrive quelli esistenti

Bitmap con più bit, uno per ogni snapshot che usa il blocco. Quando tutti liberano il blocco, quell blocco è liberato

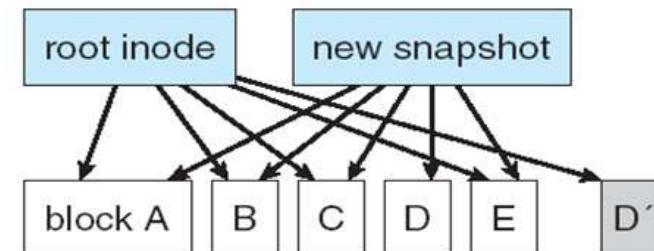
Blocchi mai sovrascritti, la scrittura è veloce perché utilizza il blocco libero più vicino



(a) Before a snapshot.



(b) After a snapshot, before any blocks change.



(c) After block D has changed to D'.

Interfaccia File-System



Interfaccia File-System

- File e directory
- Modalità di accesso
- Disco e Struttura Directory
- File-System
- File Sharing
- Protezioni
- Memory mapped files

Obiettivi

- Spiegare le funzioni del file system
- Descrivere le interfacce al file system
- Discutere la progettazione del file-system, i tradeoff, inclusi metodo accesso, file sharing, file locking, struttura directory
- Esplorare la protezione del file-system

Concetto di File

- Unità logica di immagazzinamento in memoria secondaria
- Vista uniforme sull'informazione memorizzata in mem non volatile
- Spazio contiguo di indirizzi logici
- Diversi tipi di dato:
 - Dati
 - ▶ numerico
 - ▶ caratteri
 - ▶ binari
 - ▶ programmi (sorgente, eseguibile, etc.)
- Contenuti definiti dal creatore del file
 - Diversi tipi
 - ▶ Considera **text file, source file, executable file**

Attributi del File

- Un file è associato ad un insieme di attributi
 - **Name** – denominazione (per utente umano)
 - **Identifier** – tag unico (numero) che identifica il file nel file system
 - **Type** – necessario per sistemi che supportano tipi diversi
 - **Location** – pointer alla locazione del file sul device
 - **Size** – dimensione corrente
 - **Protection** – chi può fare reading, writing, executing
 - **Time, date, and user identification** – dati per protection, security e usage monitoring (creazione, modifica, uso)
- Informazioni su file nella struttura della directory mantenuta su disco
- Diverse variazioni, incluse estensioni dei file attribute

File info Window su Mac OS X



Operazioni File

- File è un **tipo di dato astratto**
- Occorre definire il tipo di operazioni (associate a system call)
 - **Create** - due passi, allocato spazio, creato nella struttura a directory
 - **Open(F_i)** – cerca nelle directory su disco per la entry F_i , controlla i permessi, restituisce handler per le altre operazioni
 - **Close (F_i)** – chiude il file
 - **Write** – per la scrittura mantiene un **write pointer**
 - **Read** – mantiene un **read pointer**
 - **Reposition within file - seek**
 - **Delete** – (solo dopo che tutti gli hard link cancellati)
 - **Truncate** – cancellazione del file mantenendo gli attributi, size diventa zero

Open File

- Per evitare di scorrere ogni volta il file system si apre il file prima di usare
- Con open si mantengono le info sul file aperto
 - **Open-file table**: il Sistema Operativo traccia tutti file aperti, i processi hanno info sui loro file aperti (due tavole, per process e per sistema)
 - Chiusura file rimuove file aperto
 - **File-open count**: contatore del numero di volte che il file è open – per permettere di rimuovere data dalla open-file table quando gli ultimi processi chiudono
 - Create e delete non necessitano file aperti
 - File pointer: per i sistemi che non usano offset, punta all'ultima locatione di read/write (mantenuto per processo che ha il file aperto)
 - Locazione del file su disco: informazione per l'accesso diretto al file, senza dover cercare attraverso le directory
 - **Permessi**: informazione su modalità di accesso per-processo

Open File Locking

- Fornito da alcuni SO e file system
 - Simile a read-write locks
 - **Shared lock** simile a reader lock – molti processi possono acquisire concorrentemente
 - **Exclusive lock** simile a writer lock
- Mandatory o advisory:
 - **Mandatory** – accesso è negato a seconda dei lock mantenuti e richiesti (Win)
 - **Advisory** – processo può leggere lo status dei lock e decidere che fare (UNIX) lasciato al programmatore

File Locking Example – Java API

Esempio Java API. Lock prima metà esclusivo, seconda metà shared

```
import java.io.*;
import java.nio.channels.*;
public class LockingExample {
    public static final boolean EXCLUSIVE = false;
    public static final boolean SHARED = true;
    public static void main(String args[]) throws IOException {
        FileLock sharedLock = null;
        FileLock exclusiveLock = null;
        try {
            RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");
            // get the channel for the file
            FileChannel ch = raf.getChannel();
            // this locks the first half of the file - exclusive
            exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);
            /** Now modify the data . . . */
            // release the lock
            exclusiveLock.release();
        }
    }
}
```

FileLock lock(long begin, long end, boolean shared)

File Locking Example – Java API

```
// this locks the second half of the file - shared  
sharedLock = ch.lock(raf.length()/2+1, raf.length(),  
                     SHARED);  
/** Now read the data . . . */  
// release the lock  
sharedLock.release();  
} catch (java.io.IOException ioe) {  
    System.err.println(ioe);  
}finally {  
    if (exclusiveLock != null)  
        exclusiveLock.release();  
    if (sharedLock != null)  
        sharedLock.release();  
}  
}  
}
```

Tipo di File – Nome, Estensione

| file type | usual extension | function |
|----------------|--------------------------|---|
| executable | exe, com, bin or none | ready-to-run machine-language program |
| object | obj, o | compiled, machine language, not linked |
| source code | c, cc, java, pas, asm, a | source code in various languages |
| batch | bat, sh | commands to the command interpreter |
| text | txt, doc | textual data, documents |
| word processor | wp, tex, rtf, doc | various word-processor formats |
| library | lib, a, so, dll | libraries of routines for programmers |
| print or view | ps, pdf, jpg | ASCII or binary file in a format for printing or viewing |
| archive | arc, zip, tar | related files grouped into one file, sometimes compressed, for archiving or storage |
| multimedia | mpeg, mov, rm, mp3, avi | binary file containing audio or A/V information |

Struttura dei File

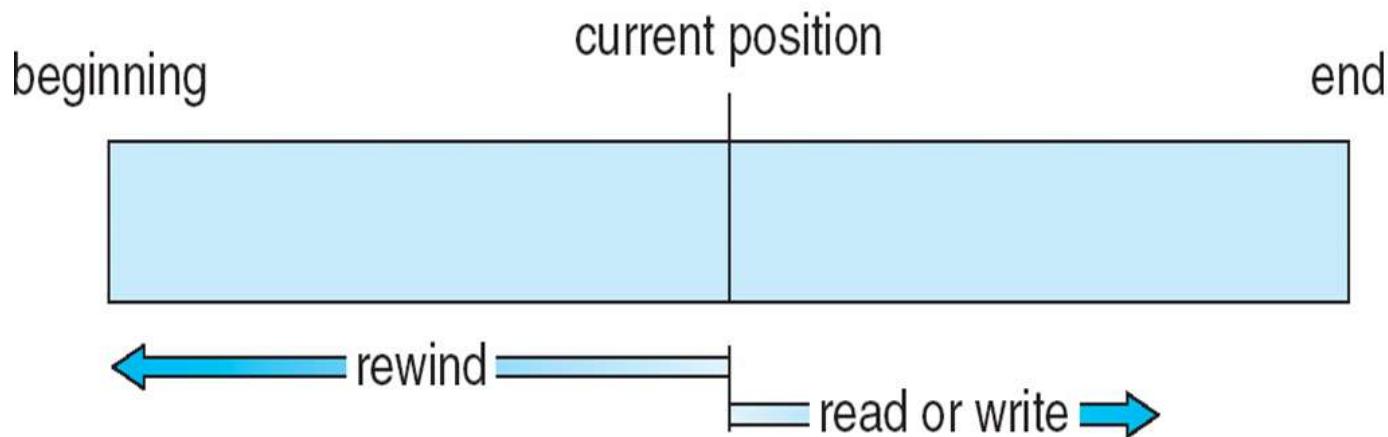
- Se si assumono diverse strutture di file, ogni tipo richiede algoritmi diversi
- Struttura minimale (UNIX, Windows)
- UNIX solo una – sequenza di bytes (programmi gestiscono da soli)
 - Ogni byte del file viene indirizzato con offset,
 - ogni blocco logico è 1 byte che è contenuto su un blocco del disco
- I file si definiscono in blocchi logici, ma si mappano su blocchi del disco
 - Se file di 1949 byte con blocchi di 512 byte su disco, si allocano 4 blocchi, cioè 2048 byte con spreco di 99 (frammentazione interna)

Metodi di Accesso

- Diverse possono essere le modalità di accesso di un file
- **Accesso sequenziale** (modello del nastro)

read next
write next
reset

Funziona sia su dispositivi ad accesso sequenziale, sia random



Metodi di Accesso

- Diverse possono essere le modalità di accesso di un file

- **Accesso sequenziale**

```
read next  
write next  
reset
```

- **Accesso diretto (relativo) – modello disco**

file è assunto di lunghezza fissata composto di **logical records**

```
read n  
write n  
position to n  
read next  
write next  
rewrite n
```

n = numero di blocco relativo

Simulazione di accesso sequenziale su Direct-access File

- Diverse possono essere le modalità di accesso di un file
- **Accesso sequenziale simulato con l'accesso diretto**

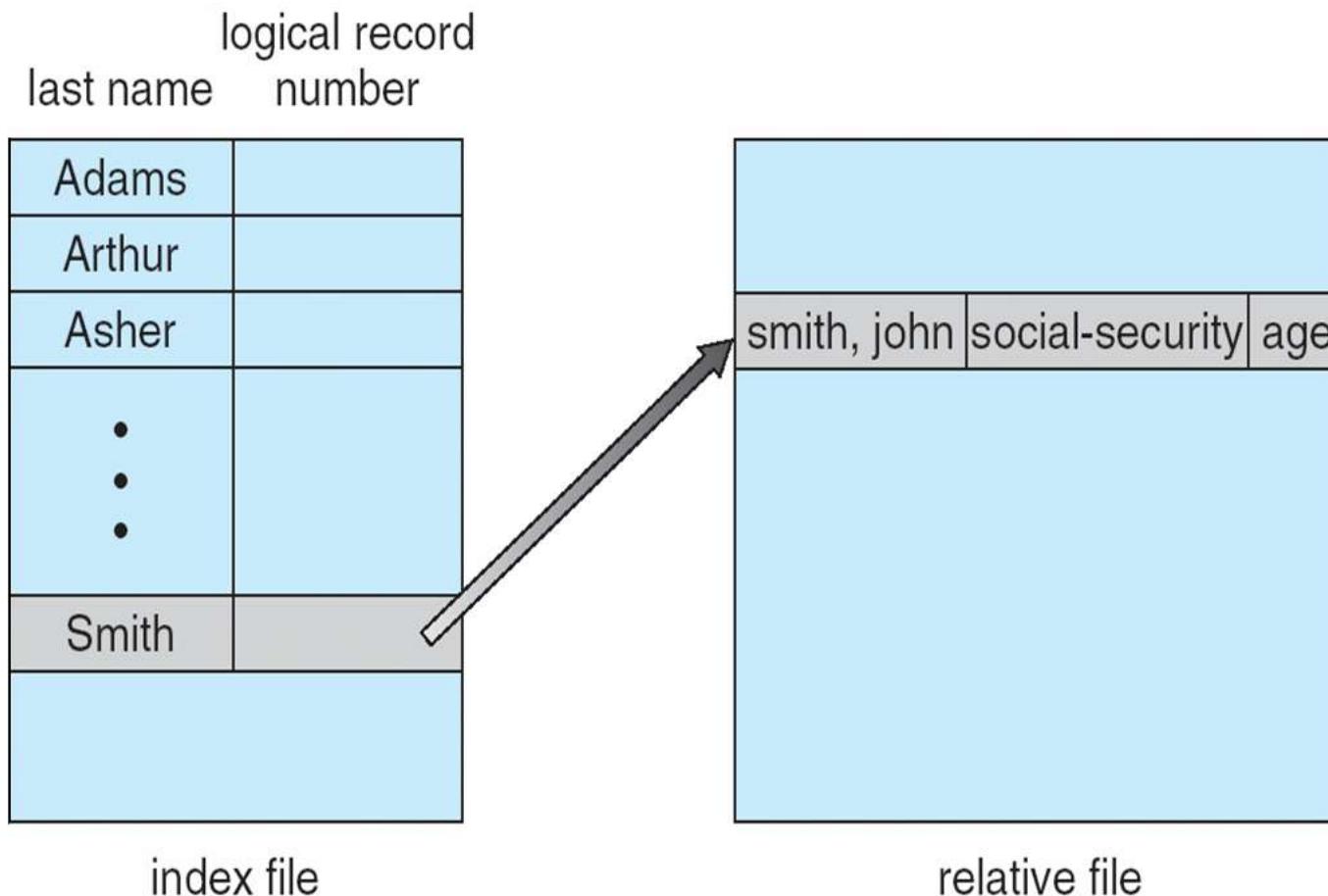
| sequential access | implementation for direct access |
|-------------------|------------------------------------|
| <i>reset</i> | $cp = 0;$ |
| <i>read next</i> | <i>read cp;</i> $cp = cp + 1;$ |
| <i>write next</i> | <i>write cp;</i> $cp = cp + 1;$ |

Altri Metodi di Accesso

- Con accesso diretto si possono definire altri metodi di accesso
- Creazione di **indici** per i file
 - Mantiene indici in memoria per un veloce determinazione della locazione dei blocchi da processare (puntatori ai blocchi)
 - Prima cerca indice, poi blocco, poi record
 - Ricerca veloce per file grandi senza troppi accessi
 - Se troppo grande, indice (in memoria) dell'indice (su disco)
- IBM indicizza con **Indexed Sequential-Access Method (ISAM)**
 - Indici master, pochi, puntano su blocchi del disco con indici secondari, indice secondario punta sui blocchi del file
 - File tenuto ordinato su una chiave per fare una binary search
 - Tutto gestito dal SO

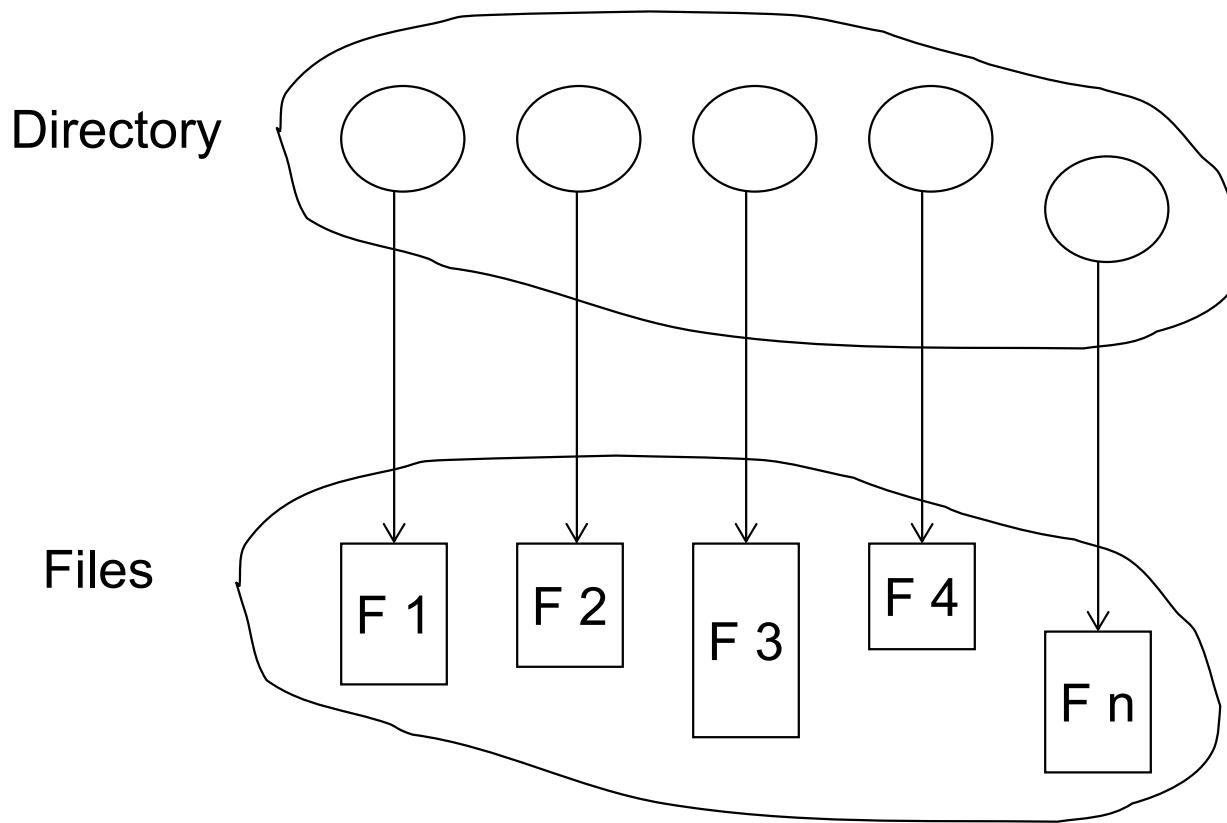
Esempio di Indice e Relative Files

Esempio (OpenVMS) SO fornisce indici e file relativi



Struttura della Directory

- Le directory possono essere viste come tavole dei simboli per tradurre i nomi dei file nei loro file control block
- Una collezione di nodi contenente informazione su tutti i file



La directory structure e il files residente su disco

Operazioni su Directory

- Le directory possono essere viste come tavole dei simboli per tradurre i nomi dei file nei loro file control block
- La directory devono permettere diverse operazioni:
 - Ricerca di un file
 - Creazione di un file
 - Cancellazione di file
 - Lista del contenuto di una directory
 - Rename di un file
 - Spostamenti sul file system

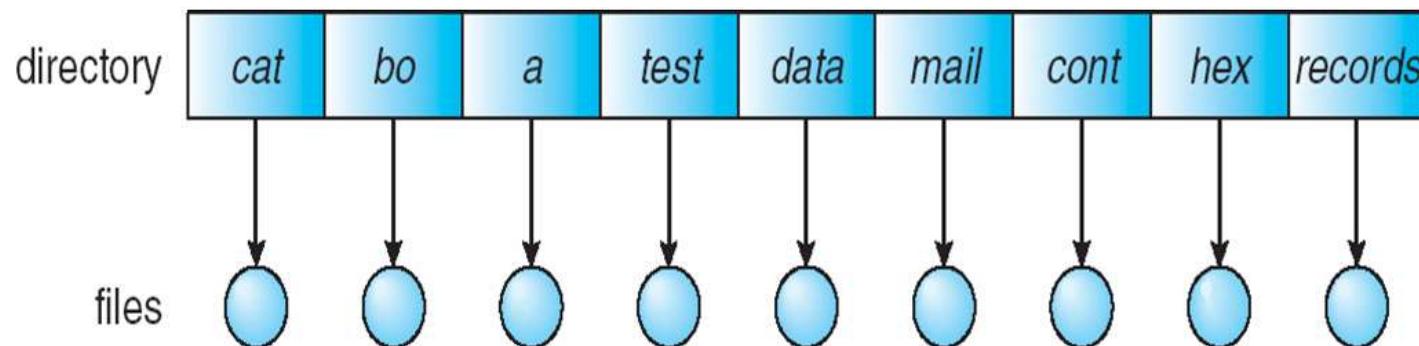
Organizzazione Directory

La struttura delle directory può essere organizzata logicamente per ottenere

- Efficienza – locazione veloce dei file
- Naming – utile per gli utenti
 - Due user possono avere stesso nome per file differenti
 - Lo stesso file con nomi differenti
- Grouping – raggruppamento logico di files per proprietà, (e.g., tutti i programmi Java, tutti i game, etc. ...)

Single-Level Directory

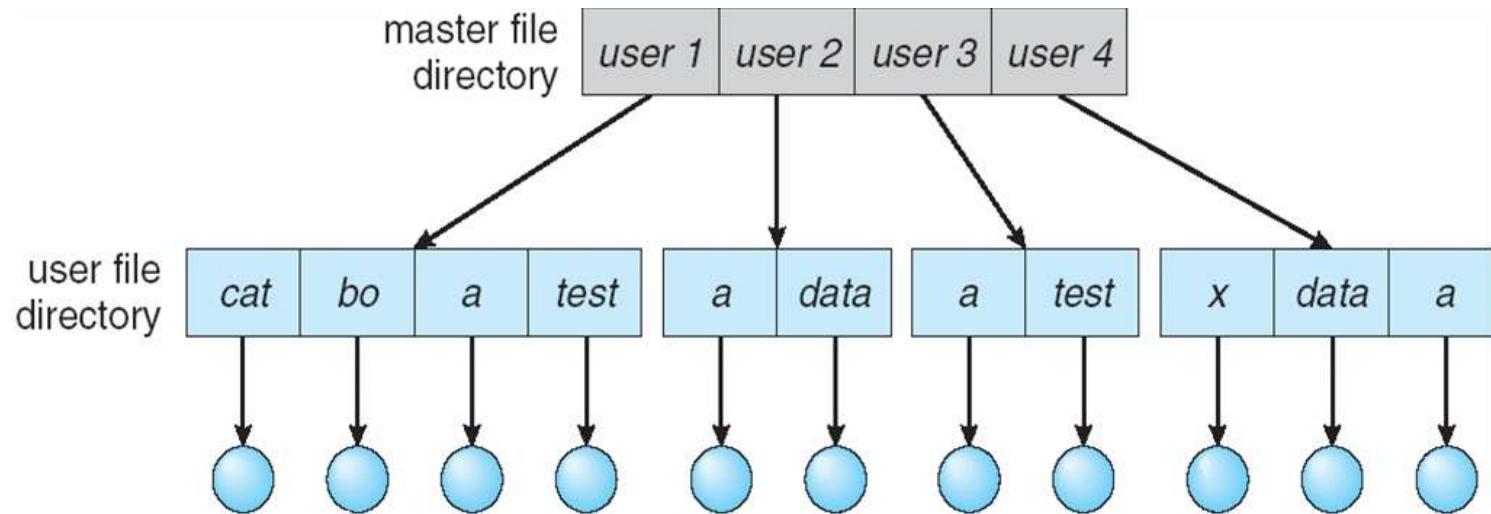
- Una singola directory per tutti gli utenti



- Problema del naming
- Problema del Grouping

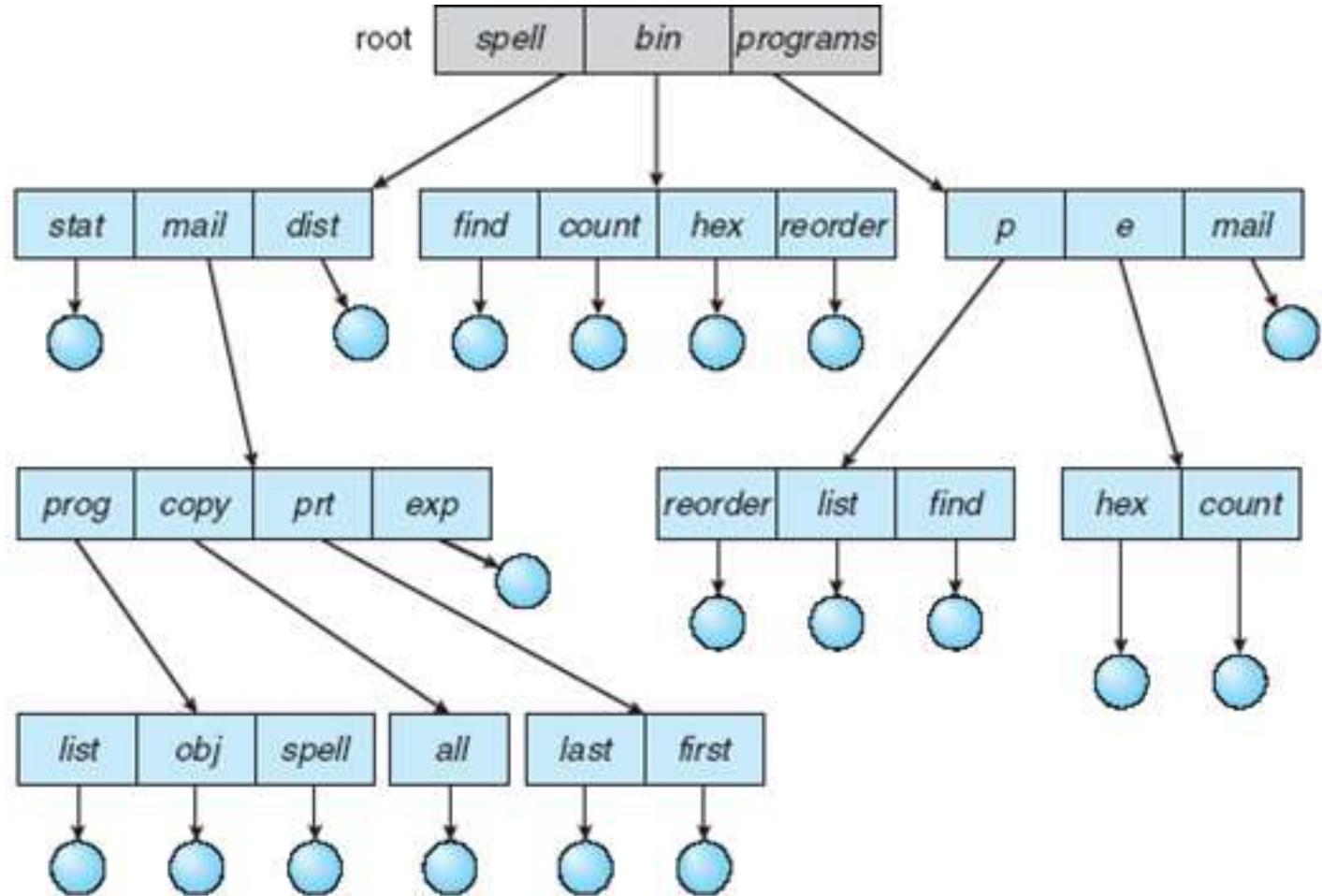
Two-Level Directory

- Directory separate per ogni user



- Path name
- Si può avere lo stesso nome di file name per utenti differenti
- Ricerca efficiente
- Non si può fare grouping

Tree-Structured Directory



Tree-Structured Directory

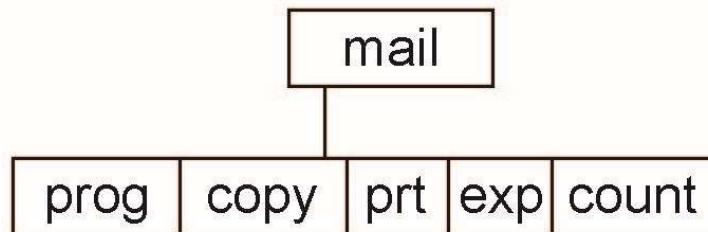
- Ricerca efficiente
- Capacità di grouping
- Directory attuale (working directory)
 - `cd /spell/mail/prog`
 - `type list`

Tree-Structured Directories

- path name **assoluti** o **relativi**
- Creazione di un nuovo file fatta nella directory corrente
- Cancellazione di file
 - `rm <file-name>`
- Creazione di una nuova subdirectory è fatto nella directory corrente
 - `mkdir <dir-name>`

Esempio: se la directory corrente è `/mail`

`mkdir count`

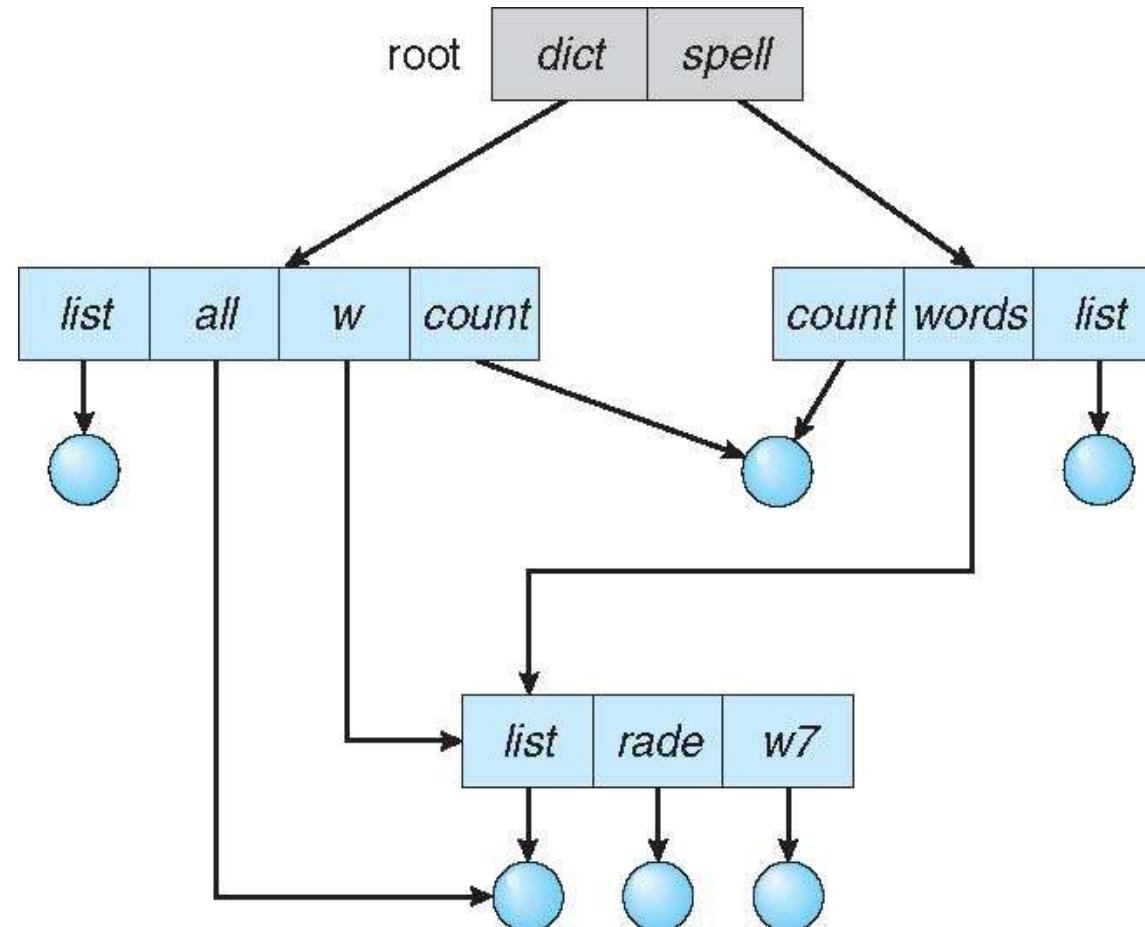


Cancellando “mail” ⇒ si cancella l’intero subtree con radice in “mail”

Accesso di utenti nelle directory di altri consentita

Acyclic-Graph Directory

- Ha subdirectory e file condivisi



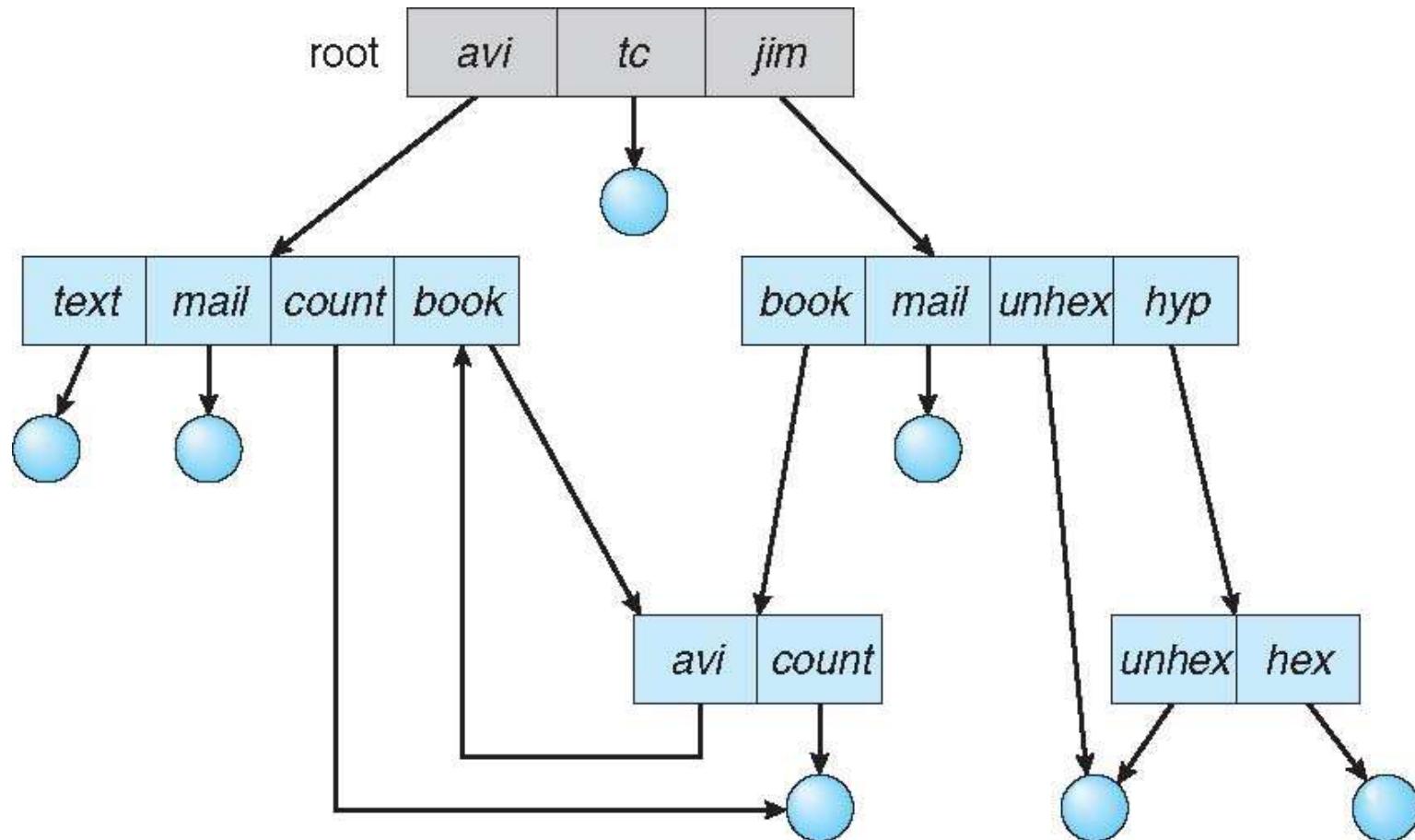
Acyclic-Graph Directory

- Due nomi differenti (aliasing)
- Se si cancellano i file \Rightarrow dangling pointer per i link

Soluzione:

- Backpointers, così si possono cancellare tutti i pointer
Problema dei record a dimensione variabile
 - Backpointers usando un'organizzazione a daisy chain
 - Soluzione entry-hold-count
-
- Nuovo tipo di entry per directory
 - **Link** – un altro nome (pointer) ad un file esistente
 - **Resolve the link** – segue il pointer per localizzare il file

General Graph Directory



General Graph Directory

- Come garantiamo che non si hanno cicli?
 - Permettere solo link a file non a subdirectory
 - **Garbage collection**
 - Ogni volta che un nuovo link è aggiunto usa un algoritmo di cycle detection per determinare se è OK
 - Costoso, si evita il ciclo

File System UNIX: Caratteristiche

- Struttura gerarchica
- Files senza struttura (byte streams)
- Protezione da accessi non autorizzati
- Semplicità di struttura

"On a UNIX system, everything is a file; if something is not a file, it is a process."

File Unix

I tipi principali di File sono:

- **File ordinari**
- **Directory**
- **File Speciali**

Il sistema assegna biunivocamente a ciascun file un identificatore numerico, detto *i-number* ("index-number"), che gli permette di rintracciarlo nel file system.

File Ordinari

- Sono sequenze di byte (byte streams)
- Possono contenere informazioni qualsiasi (dati, programmi sorgente, programmi oggetto,...)
- Il sistema non impone alcuna struttura



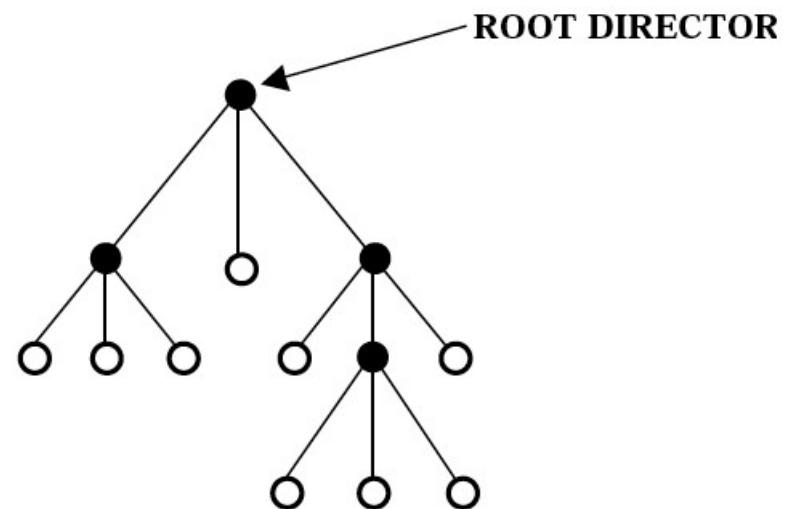
text file



binary file

Organizzazione dei File in UNIX

Per consentire all'utente di rintracciare facilmente i propri files, Unix permette di raggrupparli in cartelle, dette **Directories**, organizzate in una (unica) struttura gerarchica:

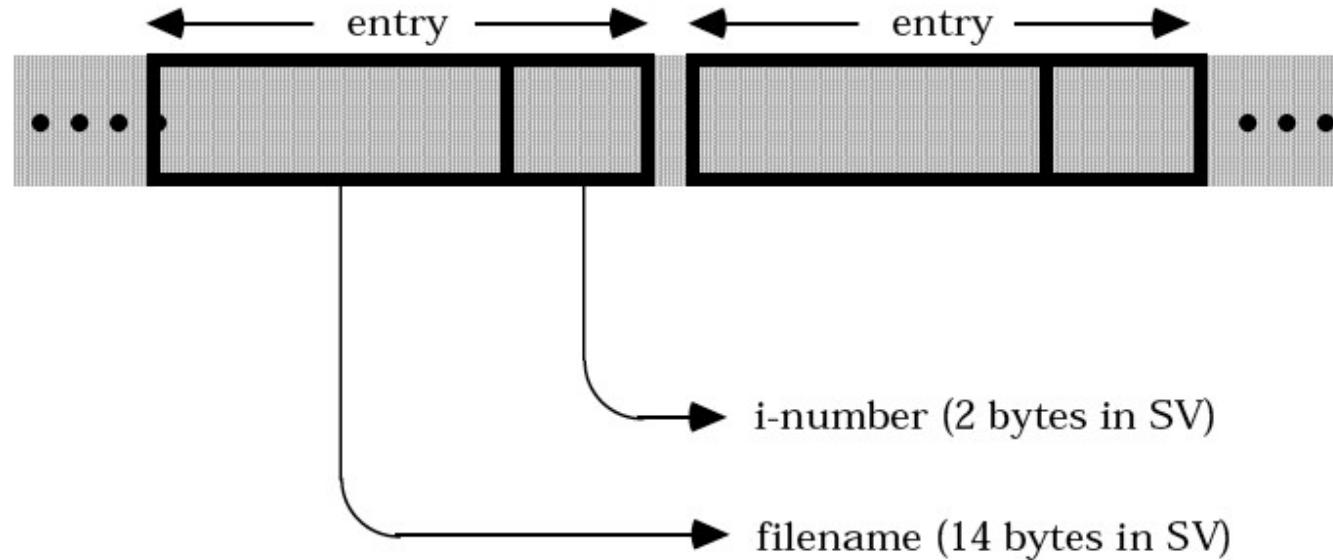


● : directory

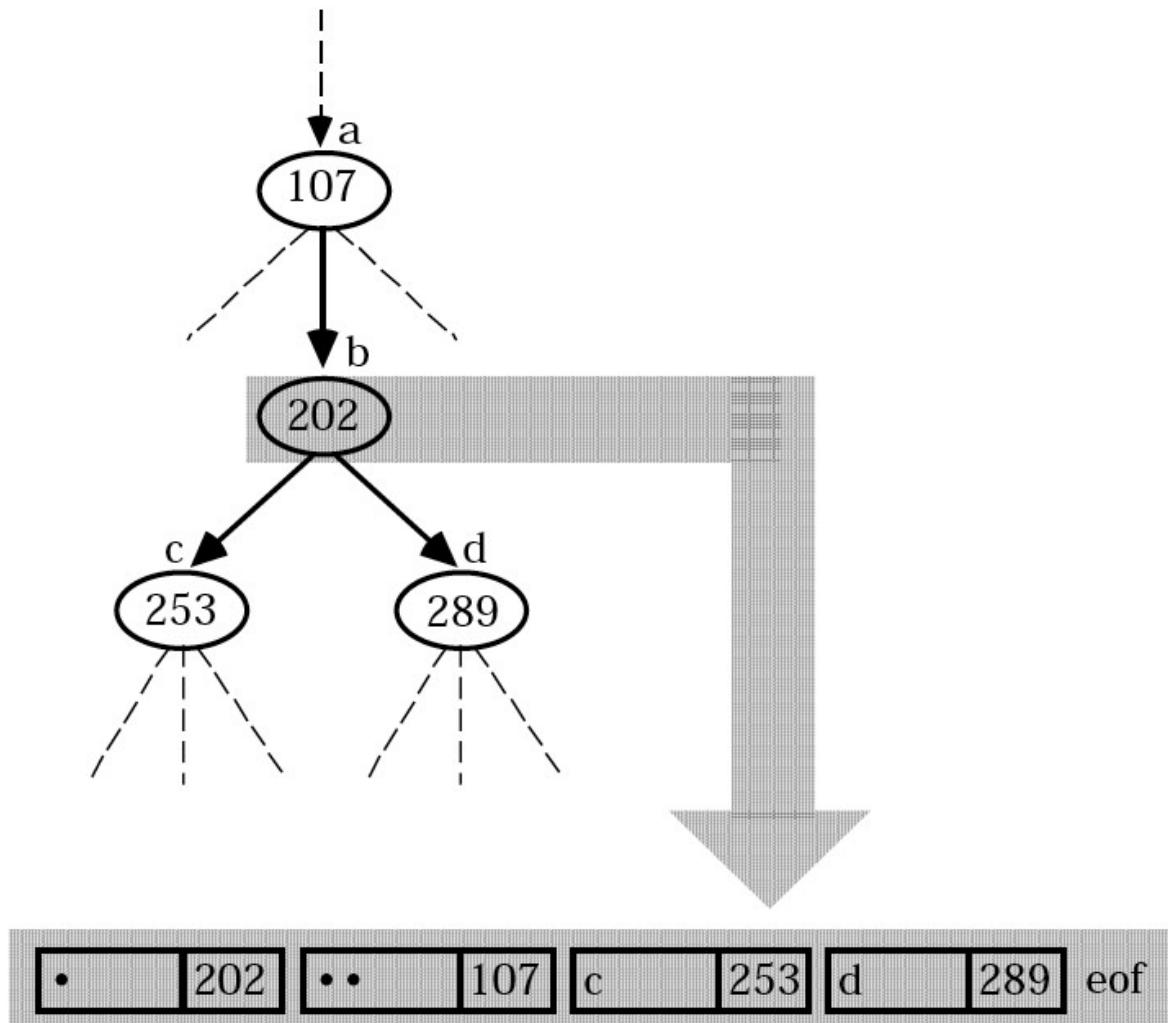
○ : file ordinario
directory (vuota)
file speciale

Directories in Unix

- Sono sequenze di bytes come i file ordinari;
- Differiscono dai file ordinari solo perché non possono essere scritte da programmi ordinari
- Il loro contenuto è una serie di **directory entries**: associazione fra gli i-number (usati dal sistema) e i **filename** mnemonici (usati dall'utente):



Esempio

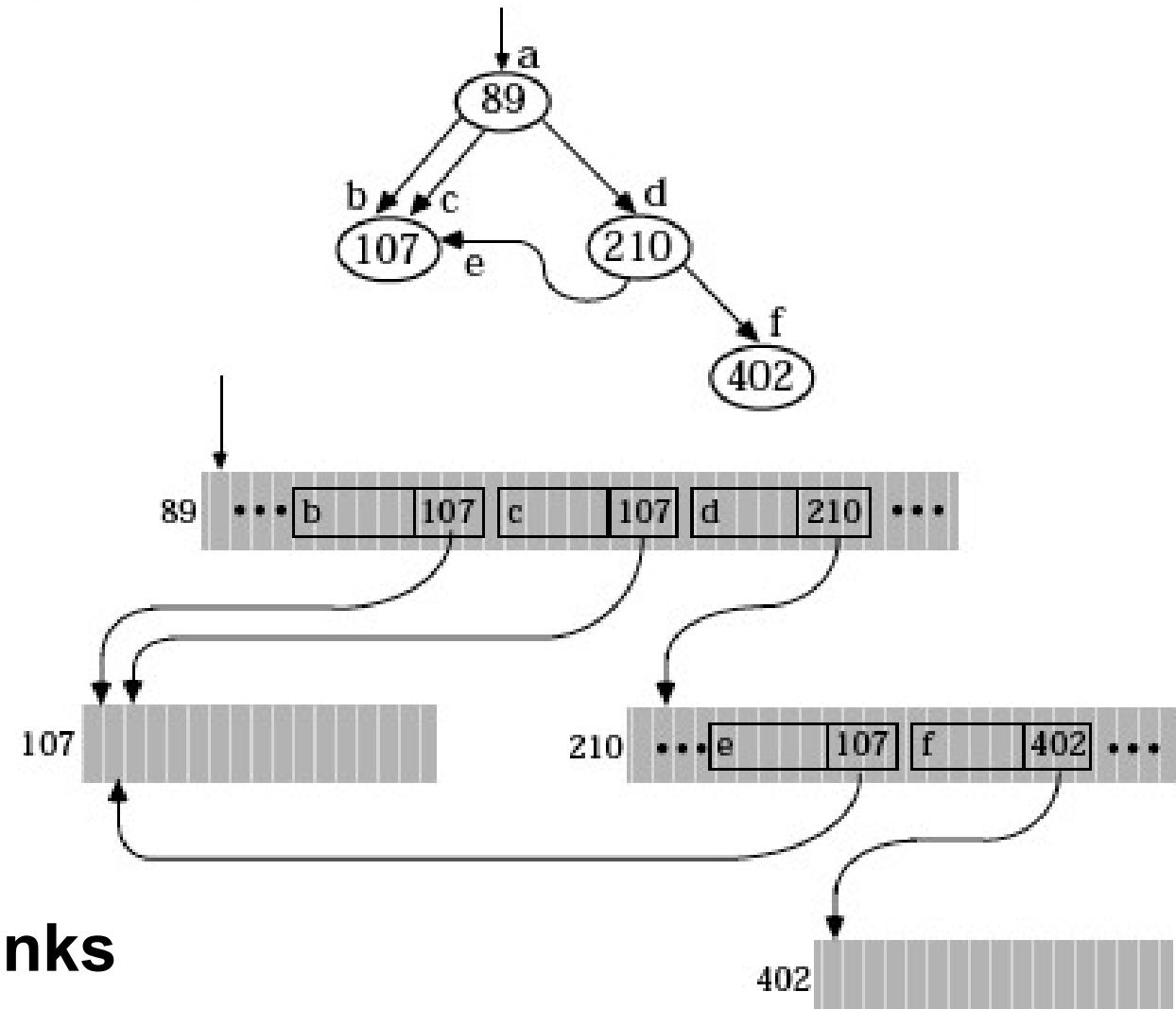


Almeno due entry: la directory stessa “.”, la directory padre “..”

Files Sinonimi in UNIX

Un file può avere più filename (ma sempre un solo i-number)

Esempio:

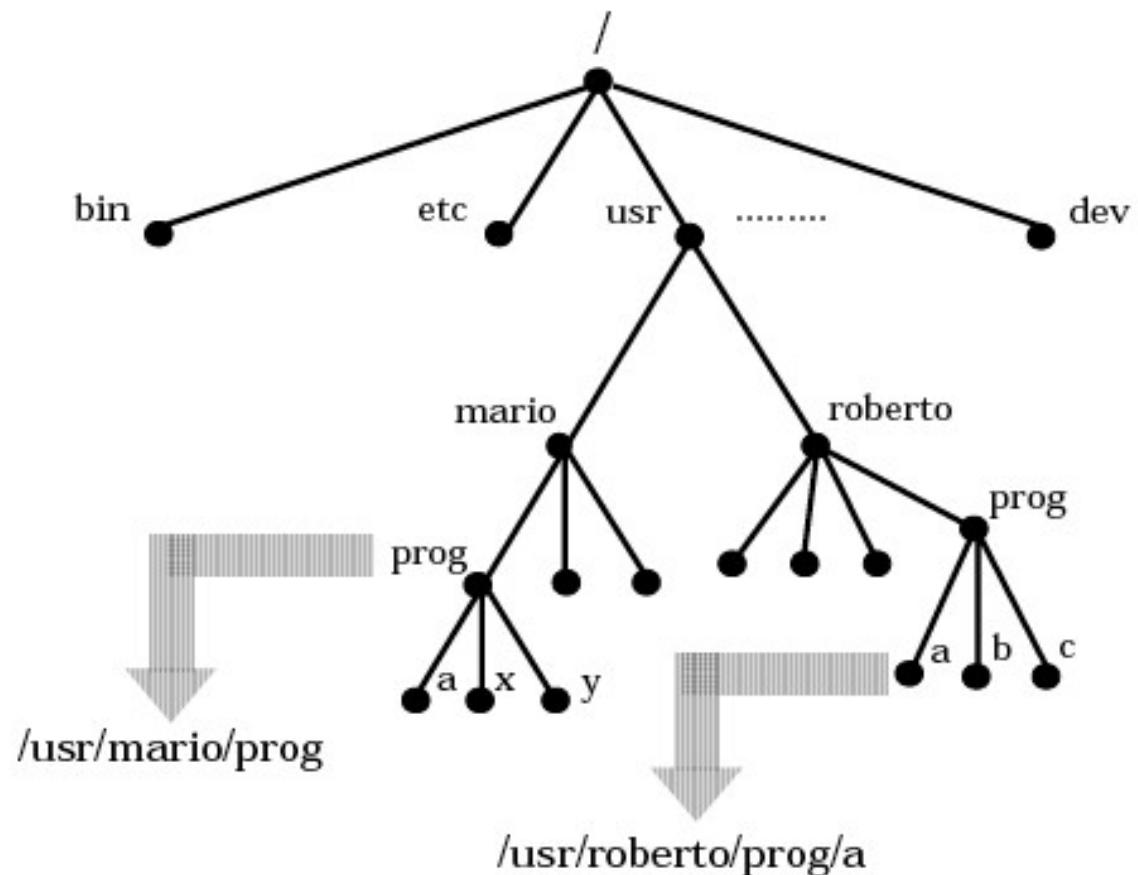


Il file 107 ha 3 links

Pathnames

Ogni file viene identificato univocamente specificando il suo **pathname**, che individua il cammino dalla root-directory al file:

/dir/dir/.... /dir/filename
↑ ↑ ↑ ↑ ↑
root separatori



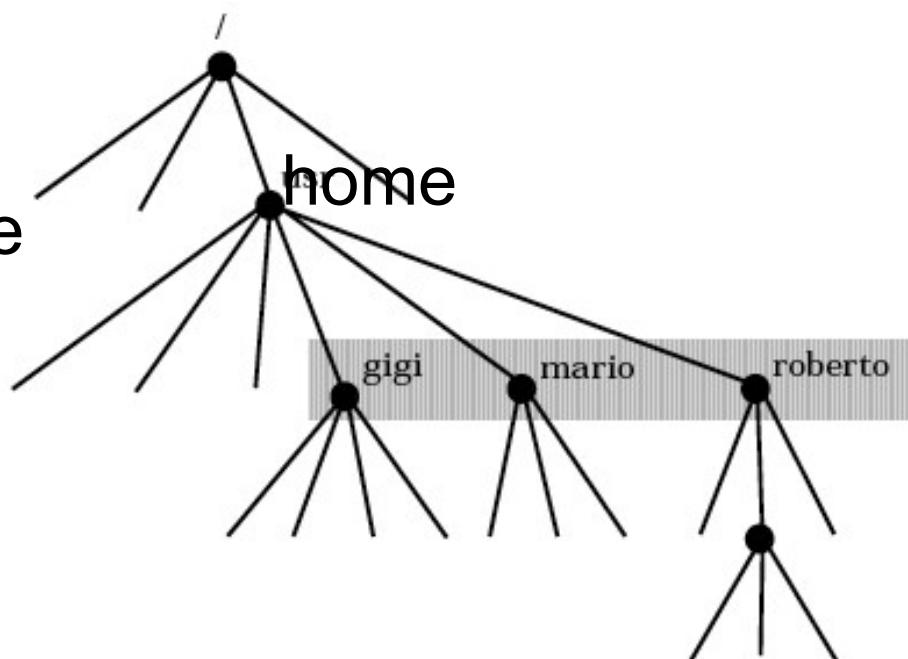
Tipiche Directories in Linux

- /bin comandi eseguibili
- /dev file speciali (I/O devices)
- /etc file per l'amministrazione del sistema, ad esempio:
 /etc/passwd
- /lib librerie di programmi
- /tmp area temporanea usata dal sistema
- /home home directories
- /usr Programmi, librerie, doc. etc. per i programmi user-related.

Home Directory

- Ad ogni utente viene assegnata dal system administrator una directory proprietà (**home directory**) che ha come nome lo username del proprietario;
- Ad essa, l'utente potrà appendere files (o subdir):

Per denotare la propria home directory si può usare l'abbreviazione "~"



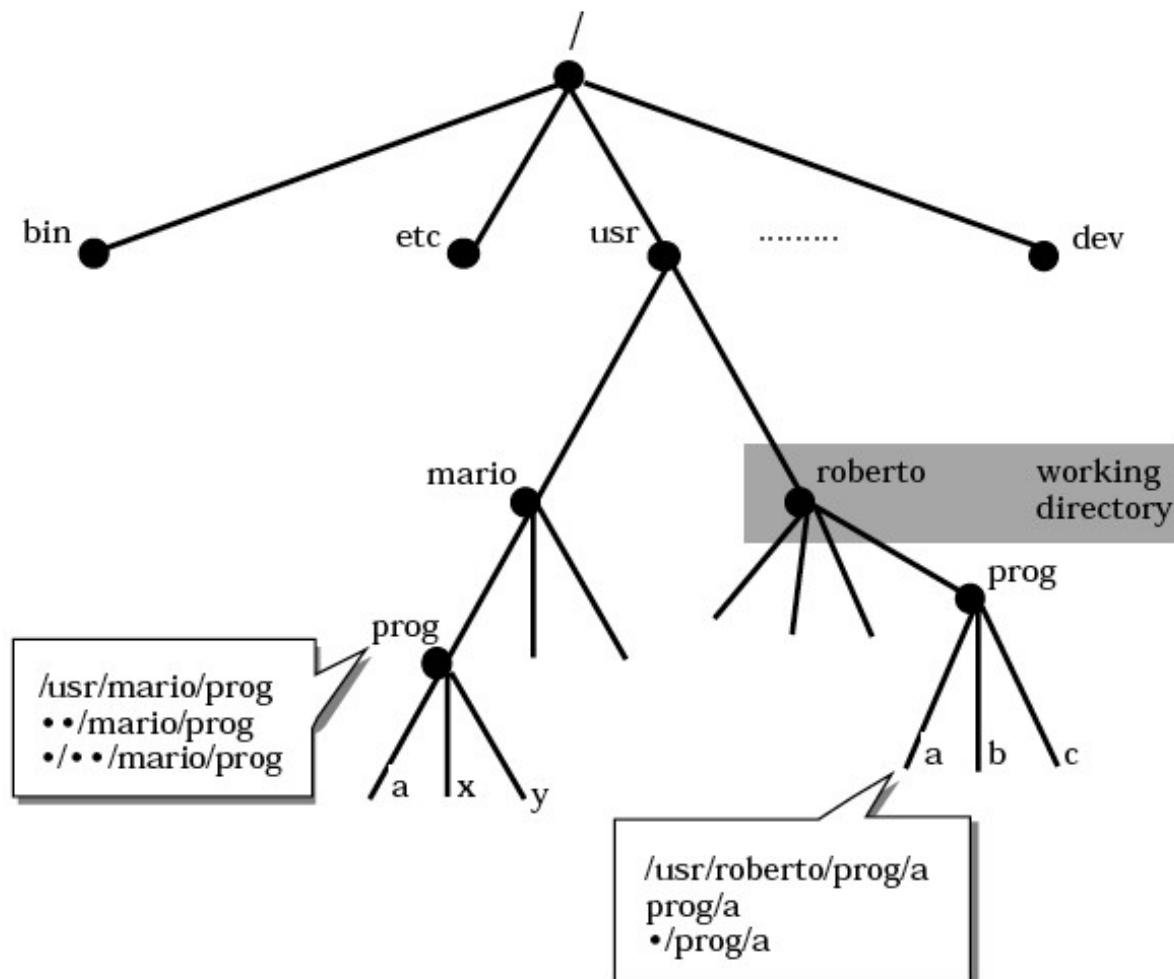
Working Directory

- Ogni utente opera, ad ogni istante, su una directory corrente, o **working directory**
- Subito dopo il login, la working directory è la home directory dell'utente
- L'utente può cambiare la working directory con il comando (**cd change directory**)

Pathnames Relativi

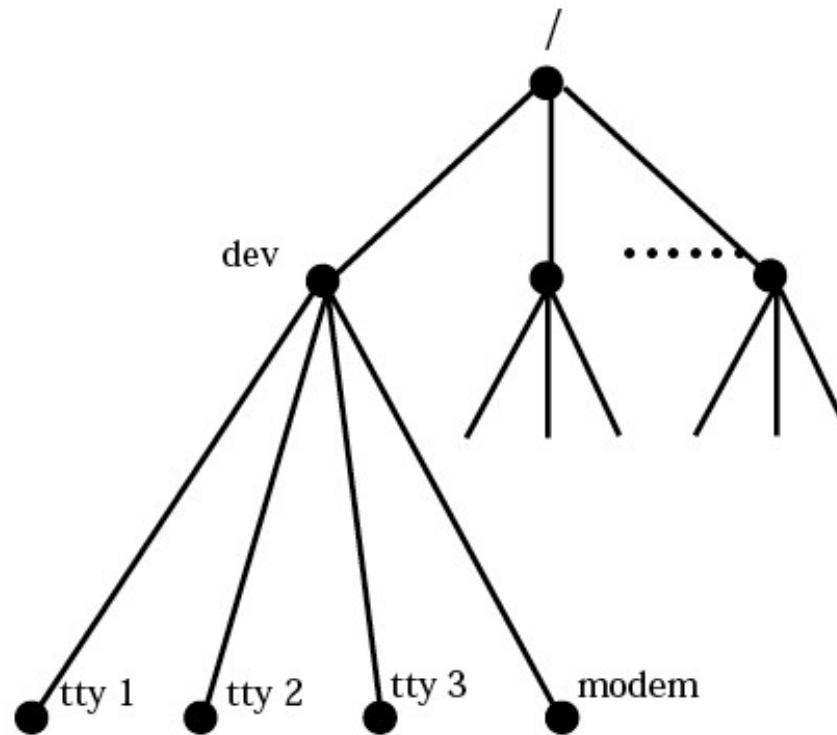
Ogni file può essere identificato univocamente specificando solamente il suo **pathname relativo alla working directory**

Esempio:



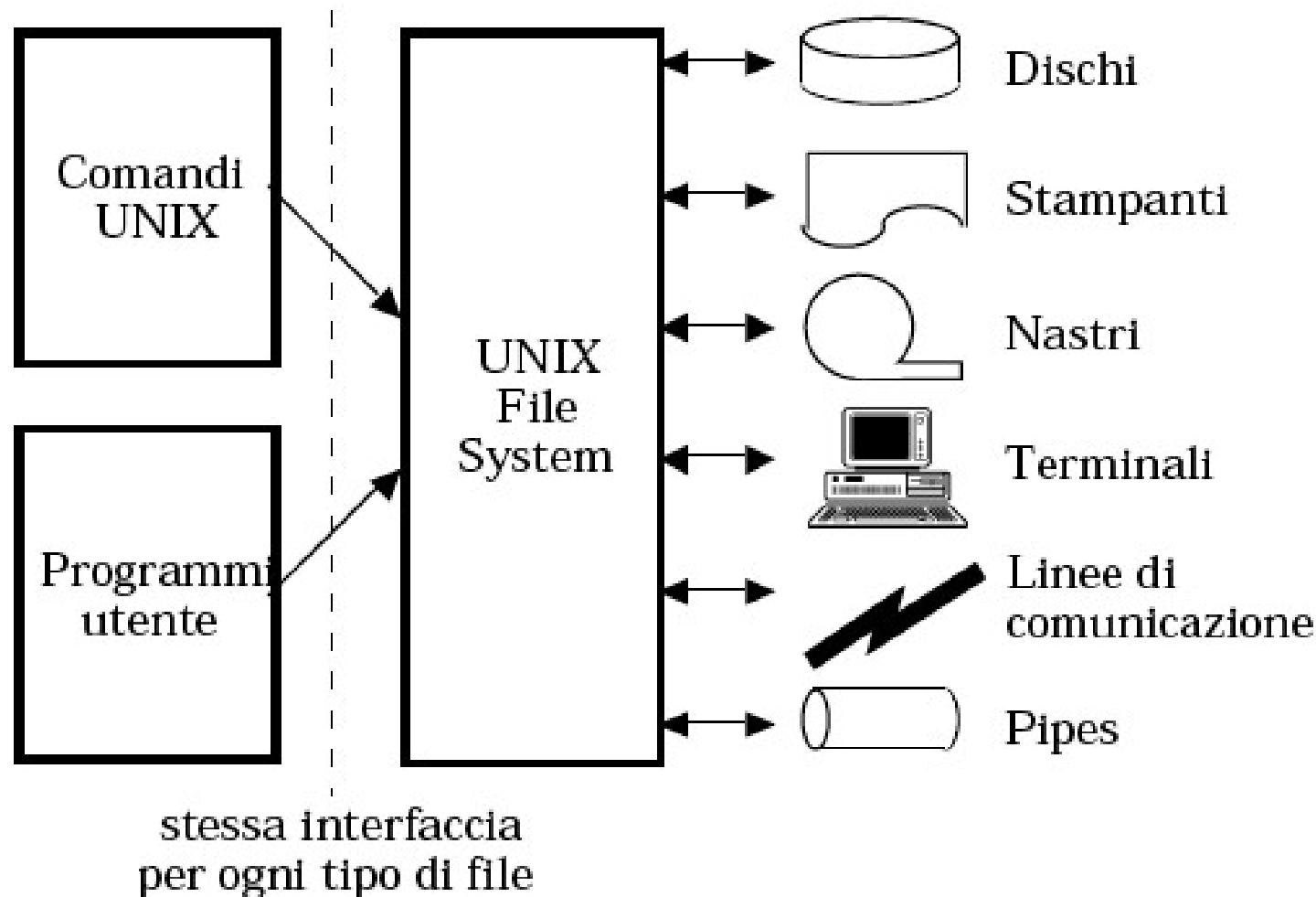
Files Speciali

- Ogni device di I/O viene visto, a tutti gli effetti, come un file (**file speciale**)
- Richieste di lettura/scrittura da/a files speciali causano operazioni di input/output dai/ai devices associati

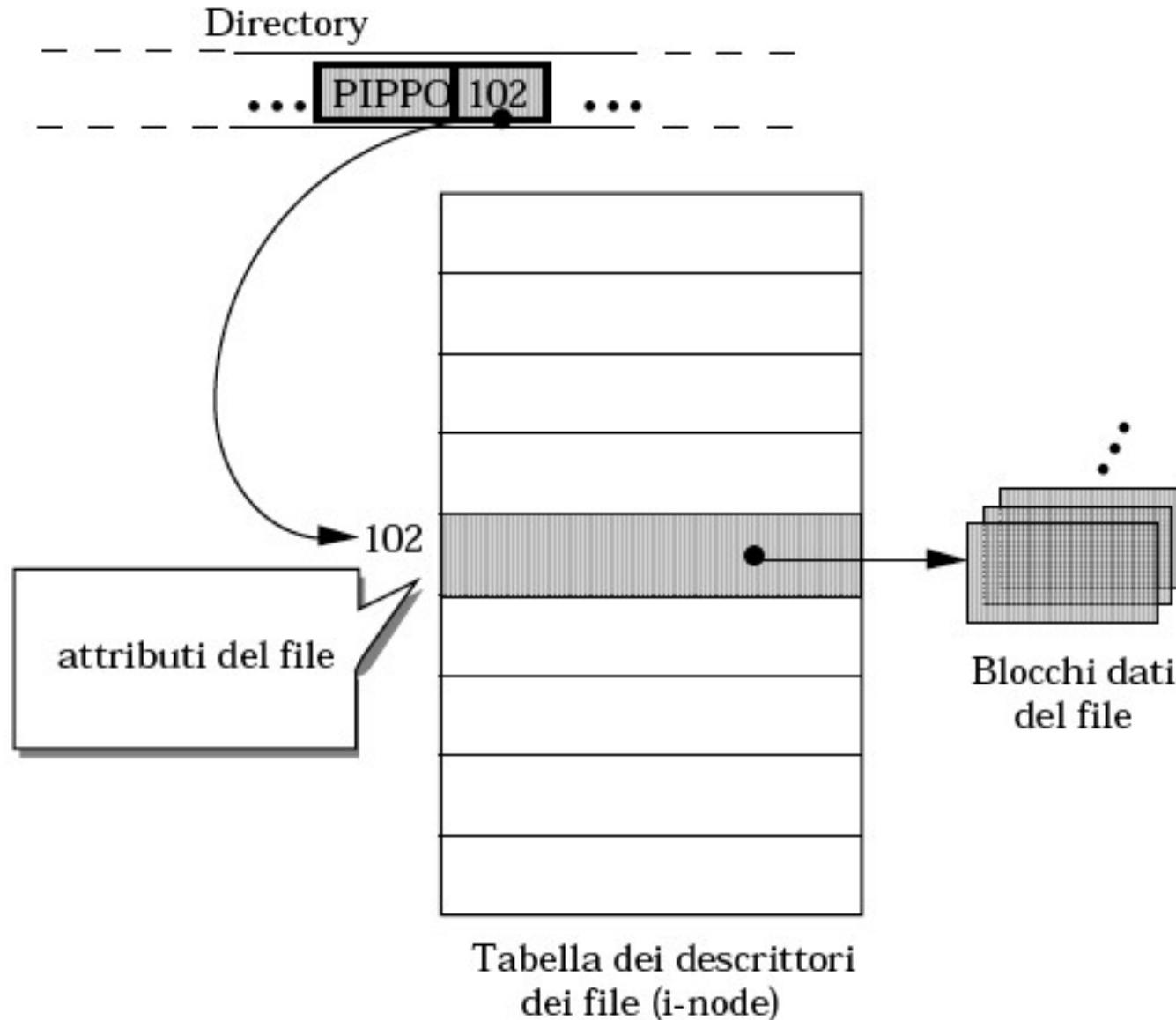


Files Speciali: vantaggi

- Trattamento uniforme di files e devices
- In Unix i programmi non sanno se operano su un file o su un device



Implementazione File



Attributi di un File in UNIX

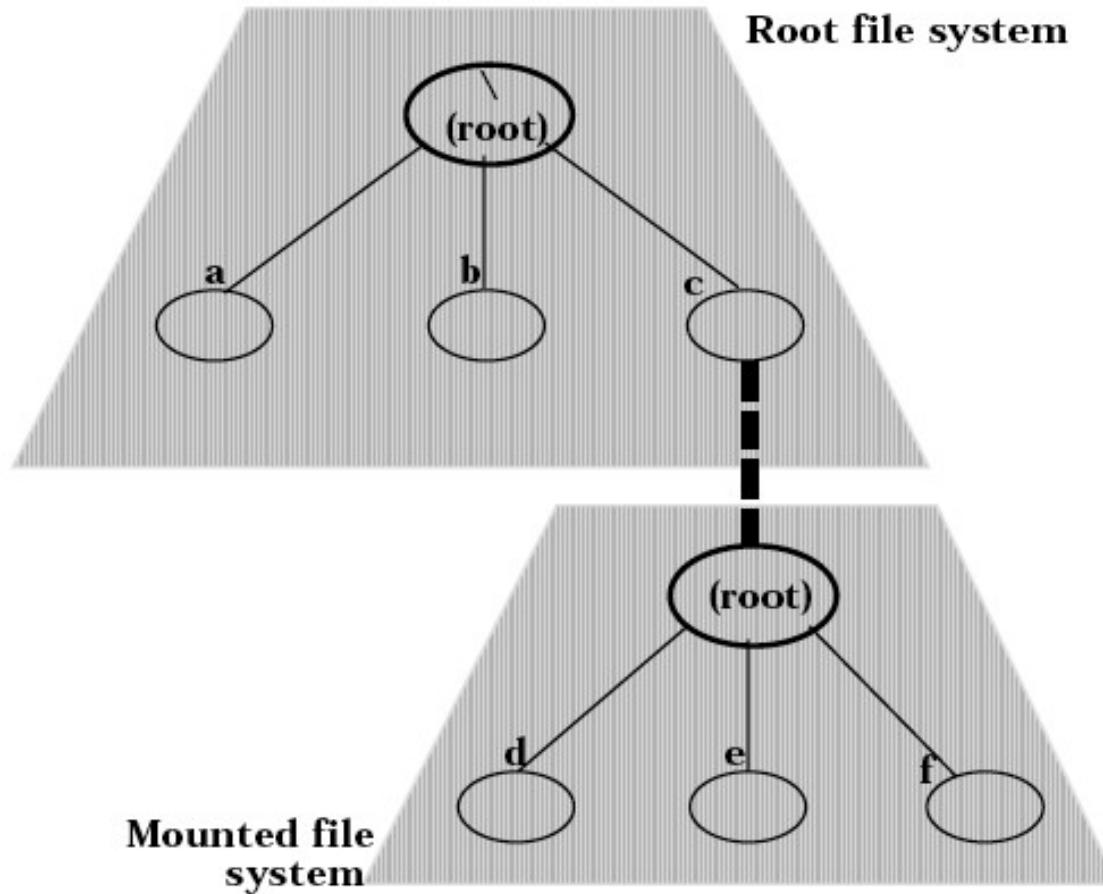
Per ogni file (ordinario, directory, speciale) Unix mantiene le seguenti informazioni nel descrittore del file:

| | |
|------------------------|---|
| Tipo | ordinario, directory, speciale? |
| Posizione | dove si trova? |
| Dimensione | quanto è grande? |
| Numero di links | quanti nomi ha? |
| Proprietario | chi lo possiede? |
| Permessi | chi può usarlo e come? |
| Creazione | quando è stato creato? |
| Modifica | quando è stato modificato più di recente? |
| Accesso | quando è stato l'accesso più recente? |

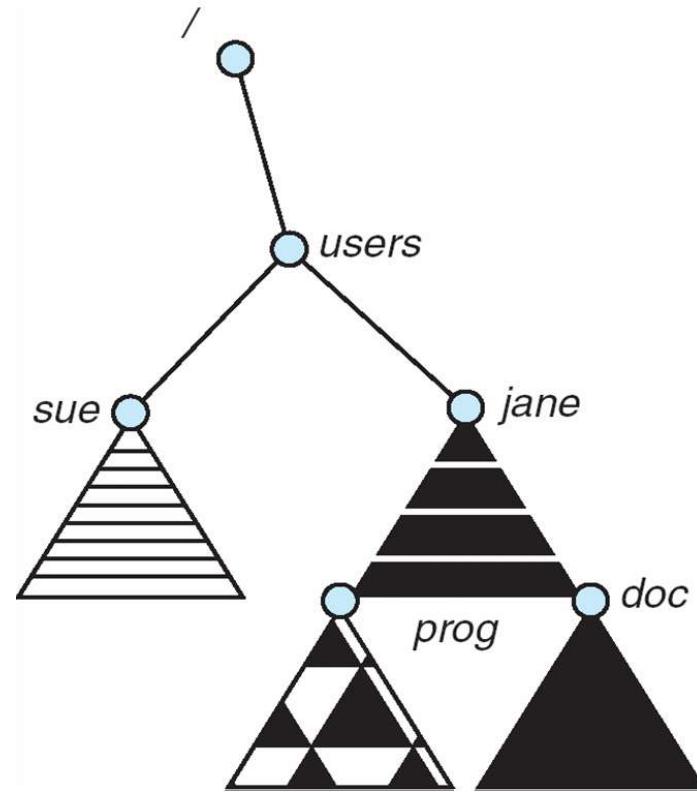
File System Montabile

Un file system Unix è sempre **unico**, ma può avere parti residenti su device rimuovibili:

- "montate" prima di potervi accedere (mount)
- "smontate" prima di rimuovere il supporto (umount)

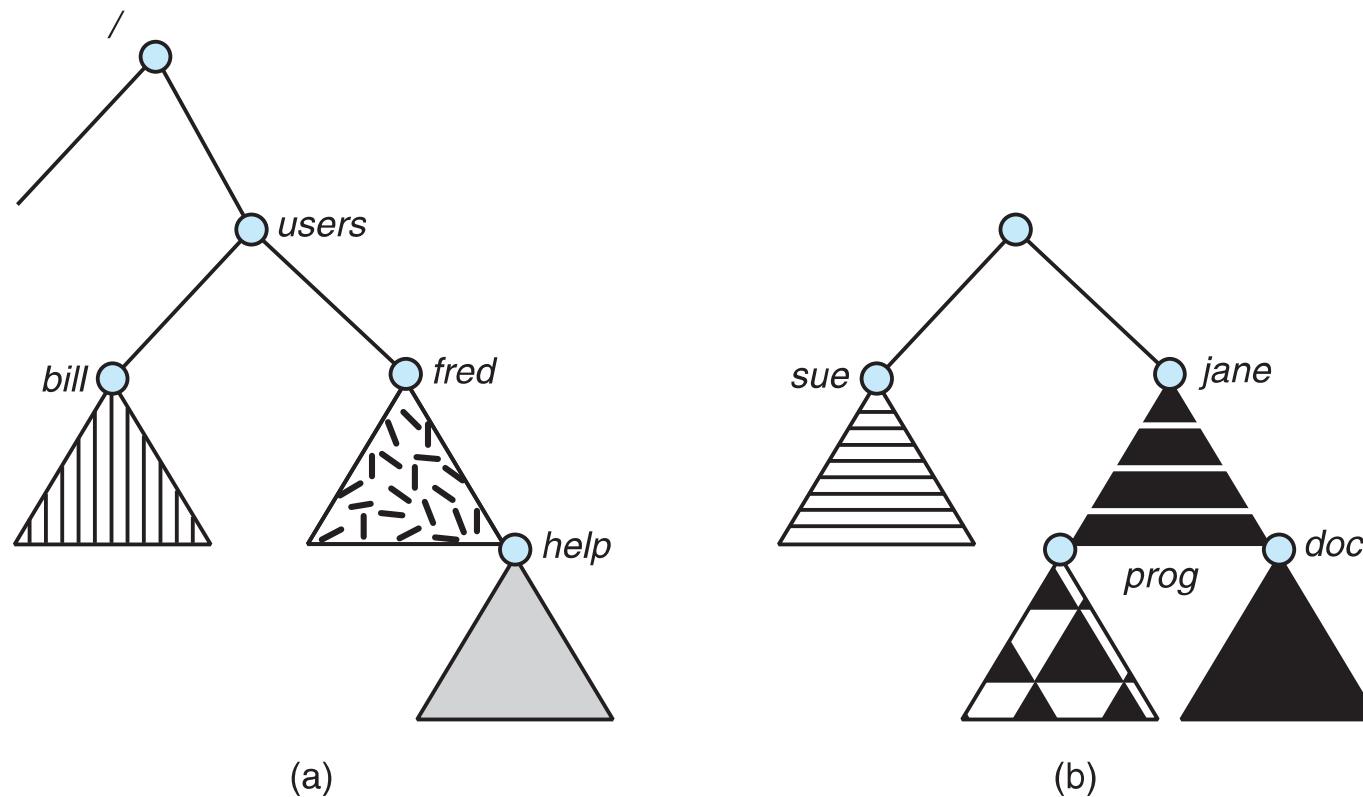


Mount Point



File System Mounting

- Un file system deve essere **montato** prima dell'accesso
- Un file system unmounted file system è montato su un **mount point**



Gestione delle Directories

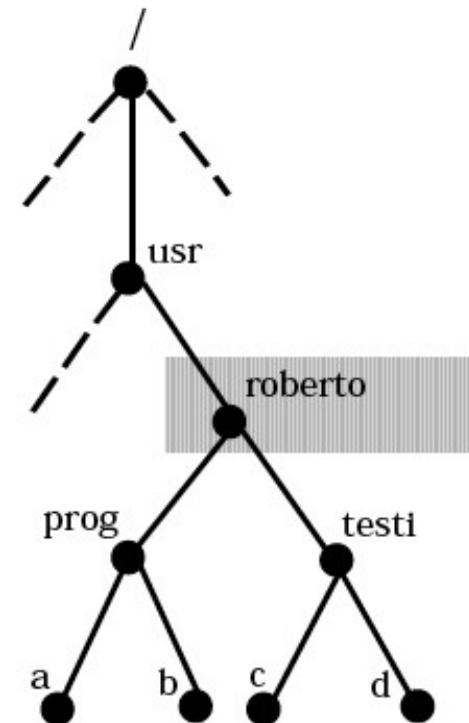
- **pwd** print working directory
- **cd** change directory
- **ls** list directory
- **du** disk usage
- **mkdir** make directory
- **rmdir** remove directory
- **ln** link

pwd (print working directory)

- Stampa pathname directory corrente

Esempio:

```
% pwd  
/usr/roberto  
%
```

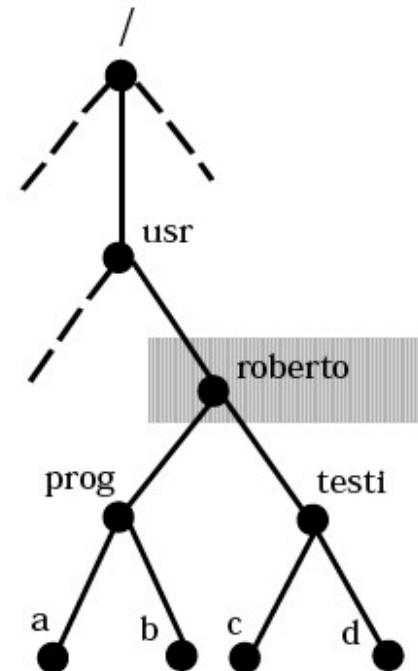


cd change directory

- La directory specificata diviene la working directory
- se nessuna directory specificata, si "ritorna" alla home directory

Esempio:

```
%cd /usr  
%pwd  
/usr  
%cd  
%pwd  
/usr/roberto  
%
```



Il comando ls

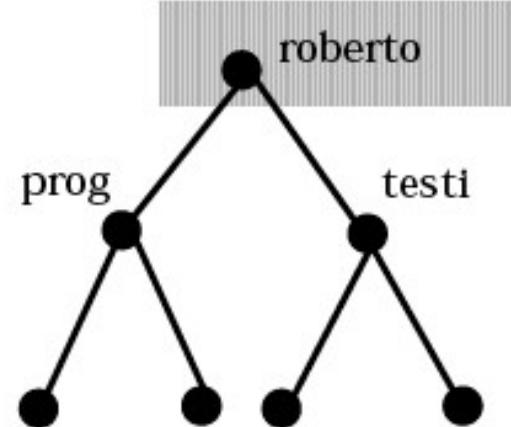
ls [options][directory...]

"list directory"

- lista (in ordine alfabetico) il contenuto della o delle directories indicate
- se nessuna directory indicata, lista il contenuto della working directory
- possiede numerose opzioni

Esempio:

```
% ls  
prog testi  
%
```



Alcune Opzioni

- **-s** fornisce la dimensione in blocchi (**size**)
- **-t** lista nell'ordine di modifica (prima il file modificato per ultimo) (**time**)
- **-1** un nome per ogni riga
- **-F** aggiunge / al nome delle directory e * al nome dei files eseguibili
- **-R** si chiama ricorsivamente su ogni sottodirectory
- **-i** fornisce l'i-number del file
- e molte altre

ls Esempi

```
% ls
dir1 file1
% ls -s
total 4 2 dir1 2 file1
% ls -t
file1 dir1
% ls -1
dir1
file1
% ls -F
dir1/ file1
% ls -R
dir1 file1
./dir1:
file1 file2 file3 file4
% ls -i
199742 dir1 51204 file1
%
```

File Nascosti (dotfiles)

- I files il cui nome inizia con "." vengono listati
- solo specificando l'opzione -a ("all")
- **Esempio:**

```
% ls -a  
.. .cshrc .mailrc dir1  
... .login .sh_history file1  
%
```

ls – campi del formato esteso

Totale dimensione occupata (in blocchi)

Riferimenti al file Dimensione (byte) Nome

```
lso:~>ls -l
total 12
-rw-rw-r-- 1 lso    lso        10 Mar  4 13:29 a
-rw-rw-r-- 1 lso    lso        10 Mar  4 14:12 b
drwxrwxr-x 2 lso    lso      4096 Mar  4 14:29 c
```

↑
Tipo
↓

↑
Permessi
↓

Proprietario

Gruppo primario

Data ultima modifica

(r)ead, (w)rite, e(x)ecute

(d)irectory, (l)ink, (c)haracter special file, (b)lock special file, (-) ordinary file

mkdir e rmdir

- **mkdir** directory ... : Crea la/le directory

Esempio:

```
% mkdir dir1 dir2  
% ls  
dir1 dir2
```

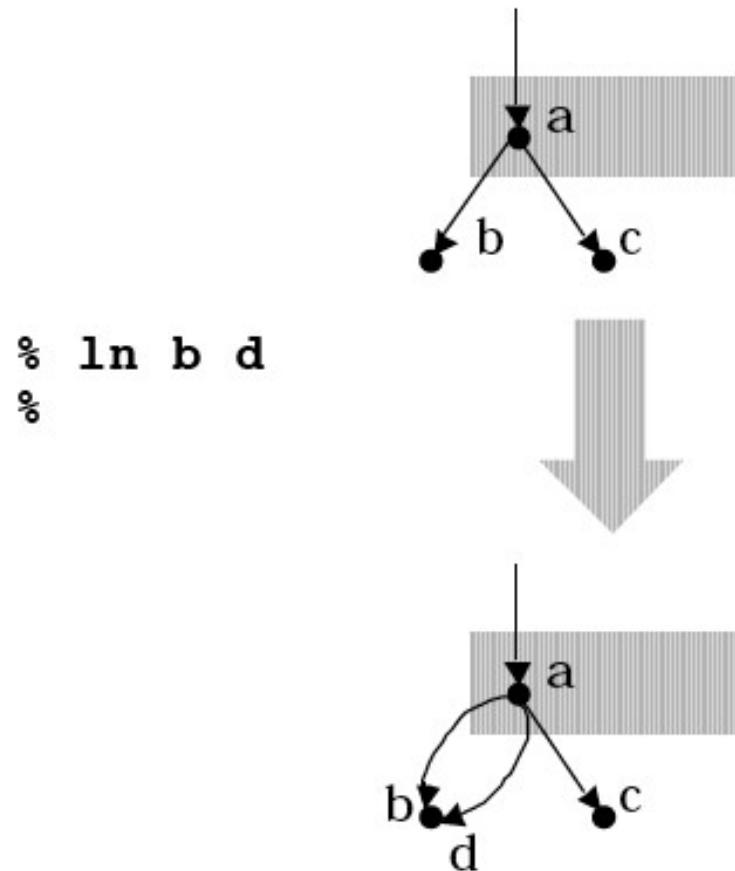
- **rmdir** directory ... : rimuove la/le directory (deve essere vuota)

```
% rmdir dir  
rmdir: dir: Directory not empty  
% ls dir  
a  
% rm dir/a  
% rmdir dir
```

Il comando “link”: ln

ln name1 name2 "link"

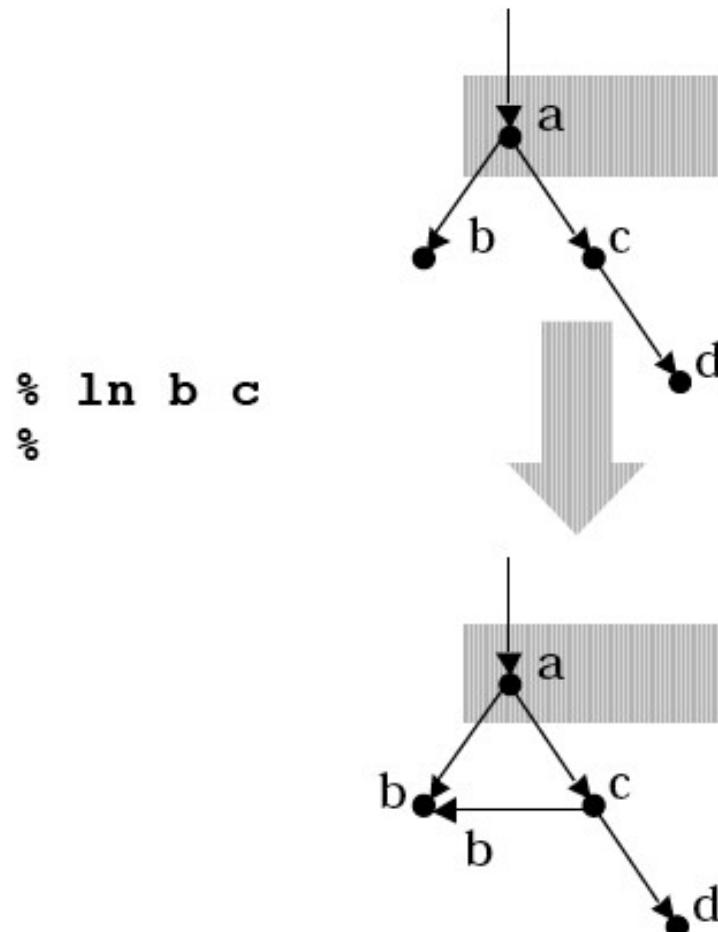
associa il nuovo nome (link) name2 al file (esistente) name1, che non può essere una directory



Il comando “link”: ln

`ln name1 name2` "link"

Se name2 è una directory, il nuovo nome è
name2/name1

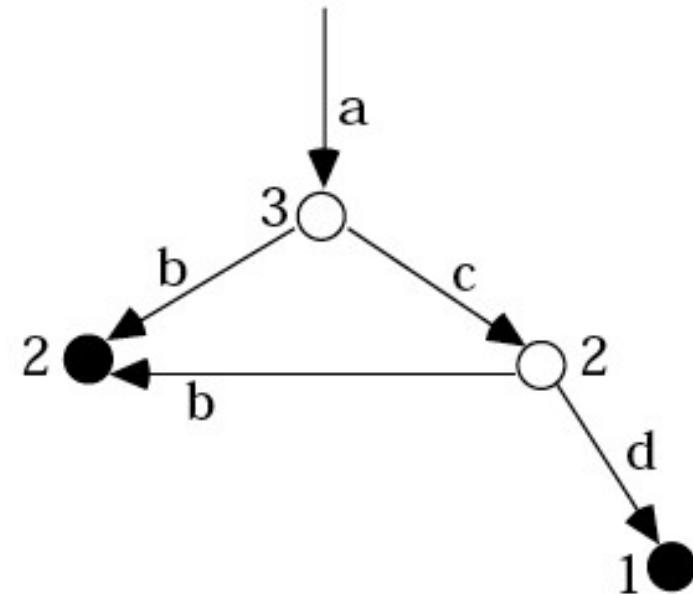


Numero links in UNIX

- Numero links e' un attributo gestito dal sistema

Per vedere:

ls -l



○ directory
● file

Esempio comandi (bash)

```
% mkdir dir  
% touch file  
% ls -l  
total 2  
drwxr-sr-x  2 roberto  usrmail      512  Mar 11 19:40 dir  
-rw-r--r--  1 roberto  usrmail        0  Mar 11 19:40  
file
```

Crea file

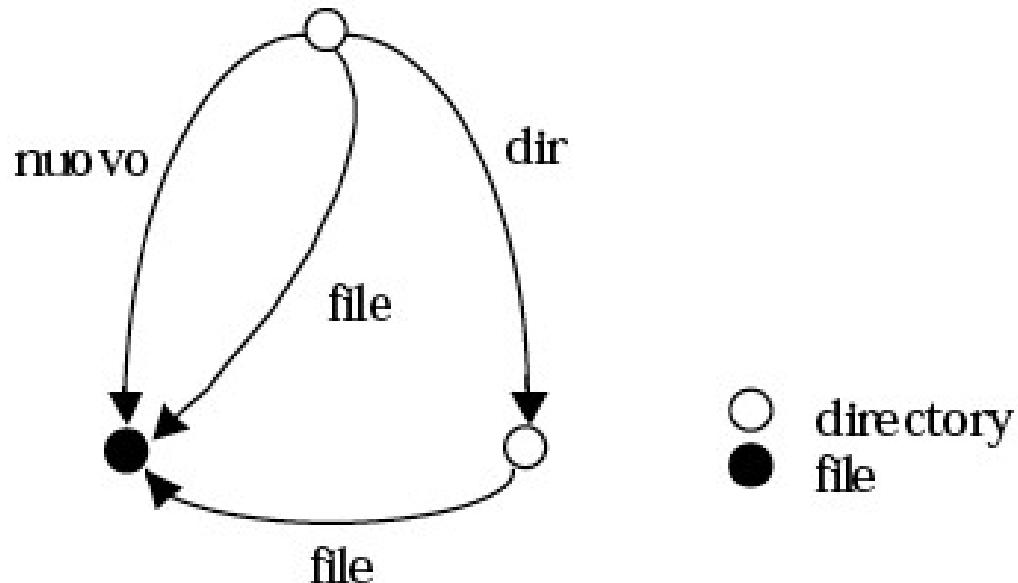
```
% ln file nuovo  
% ls -i  
199742 dir    51204 file    51204 nuovo
```

Crea link a file da nuovo

```
% ls -l  
total 2  
drwxr-sr-x  2 roberto  usrmail      512 Mar 11 19:40 dir  
-rw-r--r--  2 roberto  usrmail        0 Mar 11 19:40 file  
-rw-r--r--  2 roberto  usrmail        0 Mar 11 19:40  
nuovo
```

Esempio (bash)

```
% ln file dir          link a file da directory
% ls -l dir
total 0
-rw-r--r--    3 roberto  usrmail      0 Mar 11 19:40 file
% ln dir nuovissimo
ln: dir is a directory
%
```

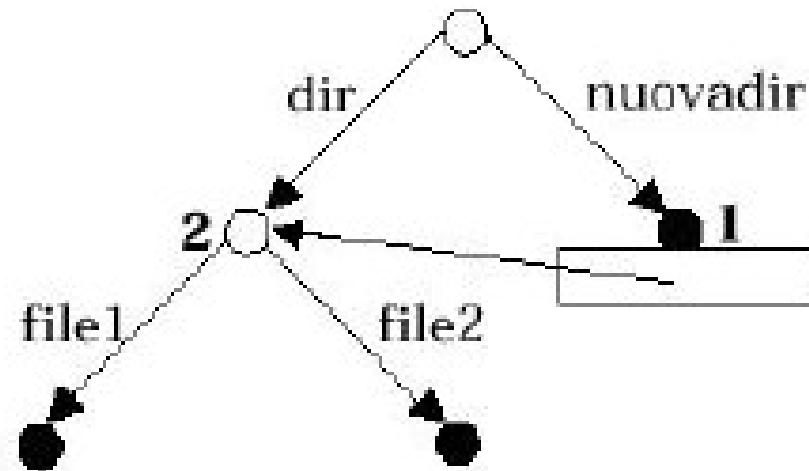


Il comando “link”: ln

- Tutti i link allo stesso file hanno identico status e caratteristiche
- Non è possibile distinguere la entry originaria dai nuovi link
- I link di questo tipo non possono essere fatti
con file che stanno su file system diversi

Link Simbolici

- In -s name1 name2
- Permette di creare link a directory;
- Permette di creare link fra file o directory che stanno su file system diversi;
- Viene creato un file name2 che contiene il link simbolico (i.e. il path di name1)



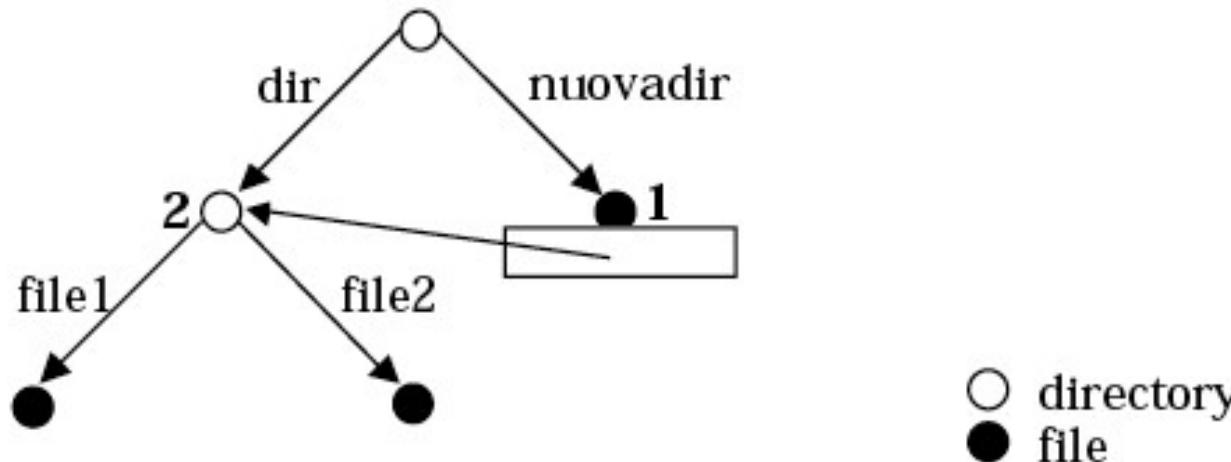
Esempio:

```
% ls  
dir  
% ls dir  
file1  file2  
% ln -s dir nuovadir  
% ls  
dir      nuovadir  
% ls nuovadir  
file1  file2  
% ls -l          ...dir con 2 rif. nuovadir con 1 rif.  
total 4  
drwxr-sr-x  2 roberto  usrmail  512 Mar 11 19:24  
dir  
lrwxrwxrwx  1 roberto  usrmail    3 Mar 11 19:24  
nuovadir -> dir  
%
```

C'e' una directory ...

... contenuto della directory...

...link simbolico a **dir** da **nuovadir**



- directory
- file

Comando mv

- **mv [options] name...target**
1. muove il file o directory name sotto la directory target;
 2. se name e target non sono directories, il contenuto di target viene sostituito dal contenuto di name

Se **target** è una directory:

□ Caso1:

```
% ls  
file1      file2      targetdir  
% mv file1 file2 targetdir  
% ls  
targetdir  
% ls targetdir  
file1      file2  
% mv targetdir/file1 targetdir/file2 .  
% ls  
file1      file2      targetdir
```

□ Caso2:

Se **target** è un file:

```
% ls  
file1      file2      file3      targetfile  
% mv file1 targetfile  
% ls  
file2      file3      targetfile  
% mv file2 file3 targetfile  
mv: Target targetfile must be a directory  
Usage: mv [-f] [-i] f1 f2  
        mv [-f] [-i] f1 ... fn d1  
        mv [-f] [-i] d1 d2
```

- Caso3:

Se target non esiste:

```
% ls  
file1      file2  
% mv file1 file2 target  
mv: target not found  
% mv file1 target  
% cat target  
contenuto di file1  
%
```

Comando cp

- **cp** [options][name...] target
- come **mv**, ma name viene copiato

```
% ls  
file1 file2 targetdir  
% cp file1 file2 targetdir  
% ls . targetdir  
.:  
file1 file2 targetdir  
  
targetdir:  
file1 file2  
  
% ls  
file1 targetfile  
% cp file1 targetfile  
% ls  
file1 targetfile
```

Comando rm

- **rm [-r] name...**
- rimuove i files indicati
- se un file indicato è una directory: messaggio di errore, a meno che non sia specificata l'opzione **-r**
... nel qual caso, rimuove ricorsivamente il contenuto della direttrice

Protezioni

- I possessori/creatori di File devono controllare:
 - Cosa può essere fatti
 - Da chi
- Tipi di accesso
 - **Read**
 - **Write**
 - **Execute**
 - **Append**
 - **Delete**
 - **List**

Accessi a Liste e Gruppi

- Lista di accessi:
 - Per ogni file e directory mantieni una lista di utenti abilitati
 - La tecnica non è efficiente (lunghe liste da mantenere con utenti variabili)
 - Versione condensata con gruppi di utenti
- Modelli di accesso: read, write, execute
- Tre classi di utenti in Unix / Linux

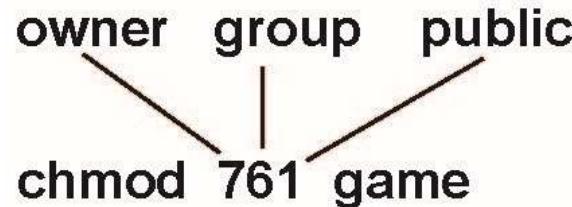
| | | | |
|------------------|---|---|-------|
| | | | RWX |
| a) owner access | 7 | ⇒ | 1 1 1 |
| | | | RWX |
| b) group access | 6 | ⇒ | 1 1 0 |
| | | | RWX |
| c) public access | 1 | ⇒ | 0 0 1 |
- Chiedi al manager di creare un gruppo (nome unico), diciamo G, e aggiungi qualche utente al gruppo
- Per un file particolare (es., *game*) o subdirectory si definisce un accesso appropriato (es., Solaris)

Accessi a Liste e Gruppi

- Modelli di accesso: read, write, execute
- Tre classi di utenti in Unix / Linux

| | | | |
|------------------|---|---|-------|
| | | | RWX |
| a) owner access | 7 | ⇒ | 1 1 1 |
| | | | RWX |
| b) group access | 6 | ⇒ | 1 1 0 |
| | | | RWX |
| c) public access | 1 | ⇒ | 0 0 1 |

- Chiedi al manager di creare un gruppo (nome unico), diciamo G, e aggiungi qualche utente al gruppo.
- Per un file particolare (es., *game*) o subdirectory si definisce un accesso appropriato.

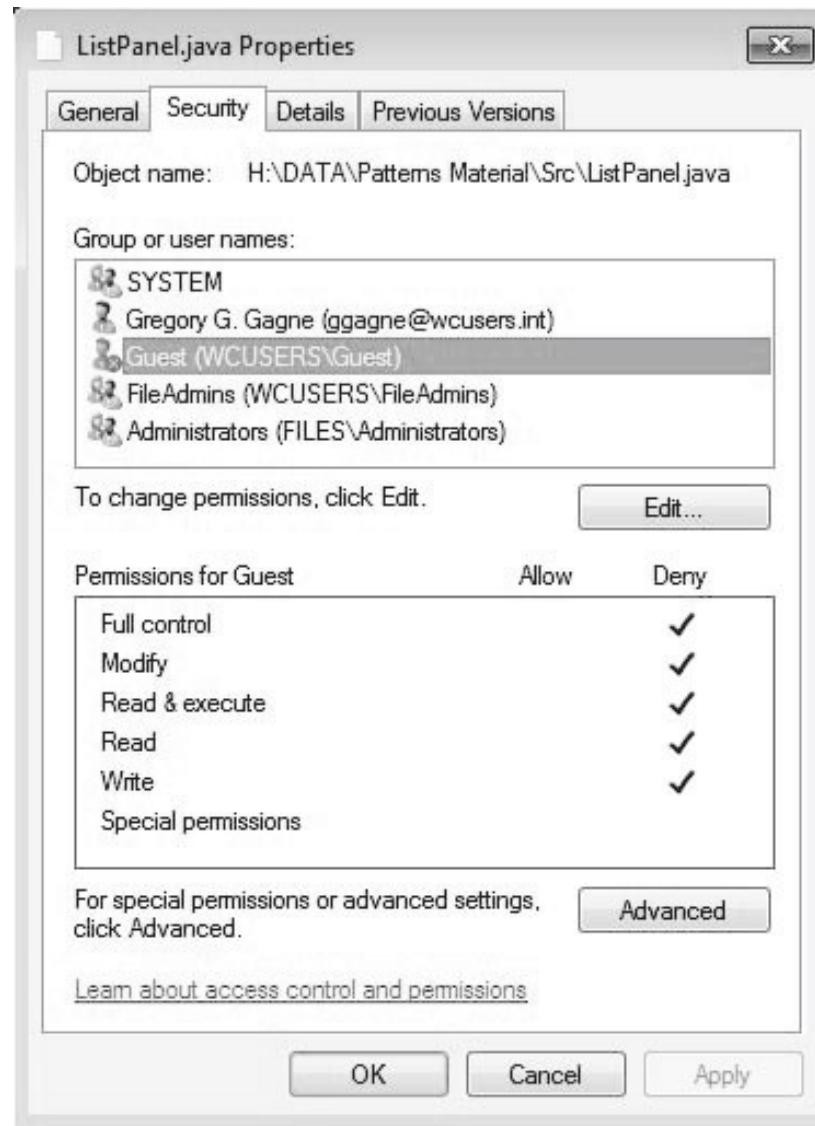


Attacca un gruppo ad un file

`chgrp G game`

Windows 7 Access-Control List Management

Windows users typically manage access-control lists via the GUI. The figure shows a file-permission window on Windows 7 NTFS file system. In this example, user “guest” is specifically denied access to the file ListPanel.java.



A Sample UNIX Directory Listing

| | | | | | | |
|------------|---|-----|---------|-------|--------------|---------------|
| -rw-rw-r-- | 1 | pbg | staff | 31200 | Sep 3 08:30 | intro.ps |
| drwx----- | 5 | pbg | staff | 512 | Jul 8 09:33 | private/ |
| drwxrwxr-x | 2 | pbg | staff | 512 | Jul 8 09:35 | doc/ |
| drwxrwx--- | 2 | pbg | student | 512 | Aug 3 14:13 | student-proj/ |
| -rw-r--r-- | 1 | pbg | staff | 9423 | Feb 24 2003 | program.c |
| -rwxr-xr-x | 1 | pbg | staff | 20471 | Feb 24 2003 | program |
| drwx--x--x | 4 | pbg | faculty | 512 | Jul 31 10:31 | lib/ |
| drwx----- | 3 | pbg | staff | 1024 | Aug 29 06:52 | mail/ |
| drwxrwxrwx | 3 | pbg | staff | 512 | Jul 8 09:35 | test/ |

The first field describes the protection of the file or directory. A d as the first character indicates a subdirectory. Also shown are the number of links to the file, the owner's name, the group's name, the size of the file in bytes, the date of last modification, and finally the file's name (with optional extension).

Protezioni di un File in UNIX

A ciascun file (normale, speciale, directory) sono associati alcuni attributi:

- **Proprietario (owner)**: l'utente che ha creato il file
- **Gruppo (group)**: il gruppo a cui il proprietario appartiene
- **Permessi (permissions)** Il tipo di operazioni che il proprietario, i membri del suo gruppo o gli altri utenti possono compiere sul file

Proprietario, gruppo e permessi iniziali sono assegnati dal sistema al file al momento della sua creazione.

Il proprietario può successivamente modificare tali attributi con appositi comandi (**chown**,**chgrp**, **chmod**)

Identificazione Utenti

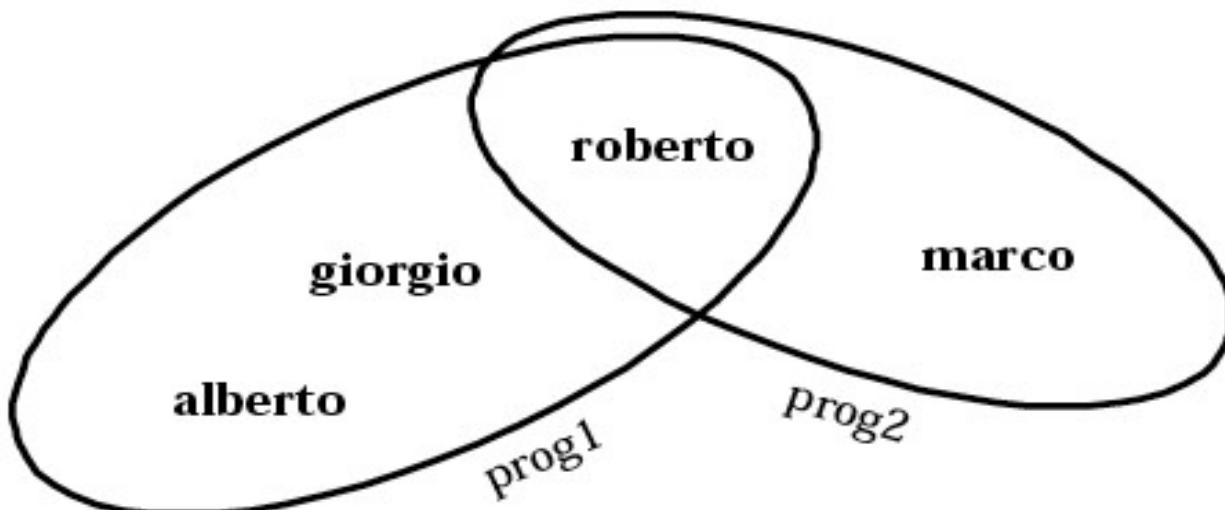
- Ogni utente viene identificato da uno **user name** assegnato dall'amministratore del sistema. Ad esso corrisponde biunivocamente uno **userid** numerico, assegnato dal sistema
- **User name** e user-id sono **pubblici**

GRUPPI

Ogni utente può far parte di uno o più **gruppi**, definiti dall'amministratore del sistema

Ogni gruppo è identificato da un **group name** di al più 8 caratteri, associato biunivocamente a un **group-id** numerico

Esempio:

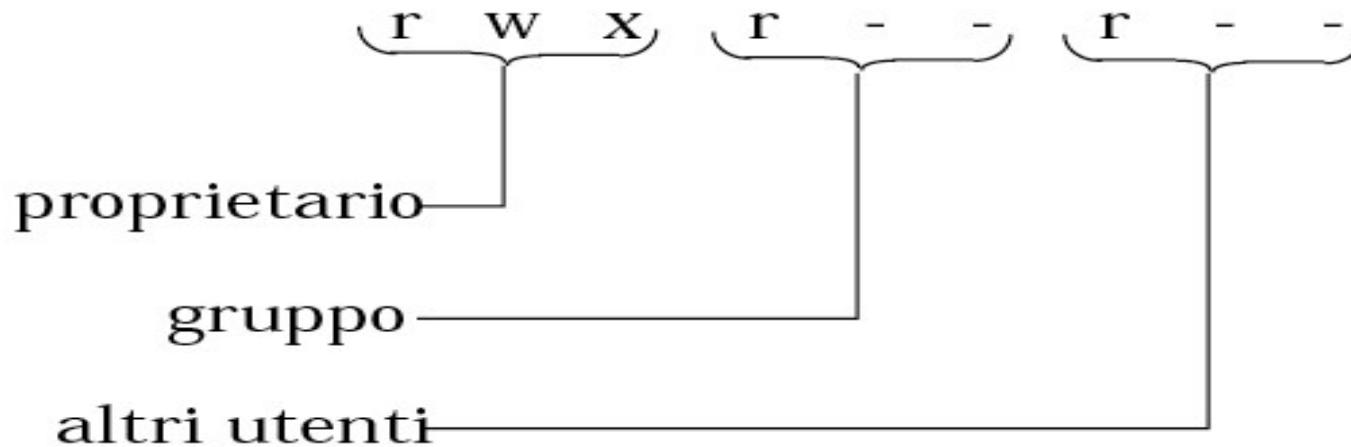


Permessi

Ad un file possono essere attribuiti i seguenti permessi:

| | | |
|----------------|-----|--|
| r : readable | per | proprietario gruppo altri utenti |
| w : writable | | |
| x : executable | | |

Esempio:



In binario: 1 1 1

1 0 0

1 0 0

In ottale: 7

4

4

Permessi iniziali

- Alla creazione di un file, Unix assegna i seguenti permessi:

- Per i *files ordinari non eseguibili*:

rw-rw-rw

110 110 110

6 6 6

- Per i *files ordinari eseguibili* e per directories:

rwx rwx rwx

111 111 111

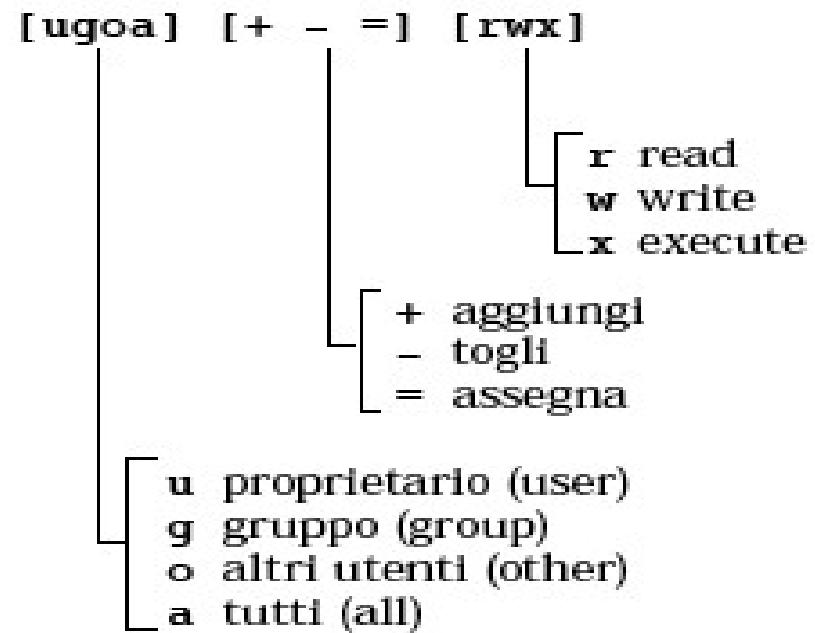
7 7 7

Comando chmod

chmod *permissions filename...*

"change mode"

- attribuisce le *permissions* a *filename*
(solo da parte del proprietario del file!)
- *permissions* può essere espresso in
forma ottale o simbolica



Permessi in forma ottale:

| | | |
|-----|-----|-----|
| 6 | 6 | 4 |
| 110 | 110 | 100 |
| rw- | rw- | r-- |

6

```
% chmod 664 file1 file2
% ls -l
total 4
-rw-rw-r-- 1 roberto  usrmail   35 Mar 11 16:34 file1
-rw-rw-r-- 1 roberto  usrmail   17 Mar 11 16:17 file2
%
```

Permessi in forma simbolica:

```
% ls -l
total 4
-rw-rw-r-- 1 roberto  usrmail   35 Mar 11 16:34 file1
-rw-rw-r-- 1 roberto  usrmail   17 Mar 11 16:17 file2
% chmod ugo+r file1
% chmod o=rwx file2
% ls -l
total 4p
-rwxrwxr-x 1 roberto  usrmail   17 Mar 11 16:34 file1
-rw-rw-rwx 1 roberto  usrmail   17 Mar 11 16:17 file2
```

chown (change owner)

```
chown [options] [user] [:group] file...
```

Cambia proprietario e/o gruppo primario per uno o più file.

Se dopo ":" non segue il nome del gruppo, viene attribuito il gruppo principale cui appartiene user.

Se prima di :group non viene indicato il nome dell'utente, viene cambiato solo il gruppo primario (chgrp)

– IL COMANDO chgrp

chgrp newgroupid file...

"change group"

- *newgroupid* diventa il nuovo gruppo dei *file...*
- il comando può essere eseguito solo dal proprietario (o dal superuser)

Memory Mapped File

- Con il metodo memory mapped una parte dell'indirizzo virtuale viene usato per accedere ai file
- Si evita di fare accesso per ogni operazione (open, read, write, etc.)
- Blocco del disco mappato su una pagina in memoria
 - Al primo accesso page fault, poi funziona come accesso normale
 - La scrittura su disco non è immediata, in alcuni casi quando è chiuso il file
 - Alcuni sistemi solo con specifiche system call fanno memory mapping
 - Altri invece trattano scrittura come memory mapped
 - Solaris, distingue: mmap in user space se esplicito, altrimenti nel Kernel

Memory-Mapped Files

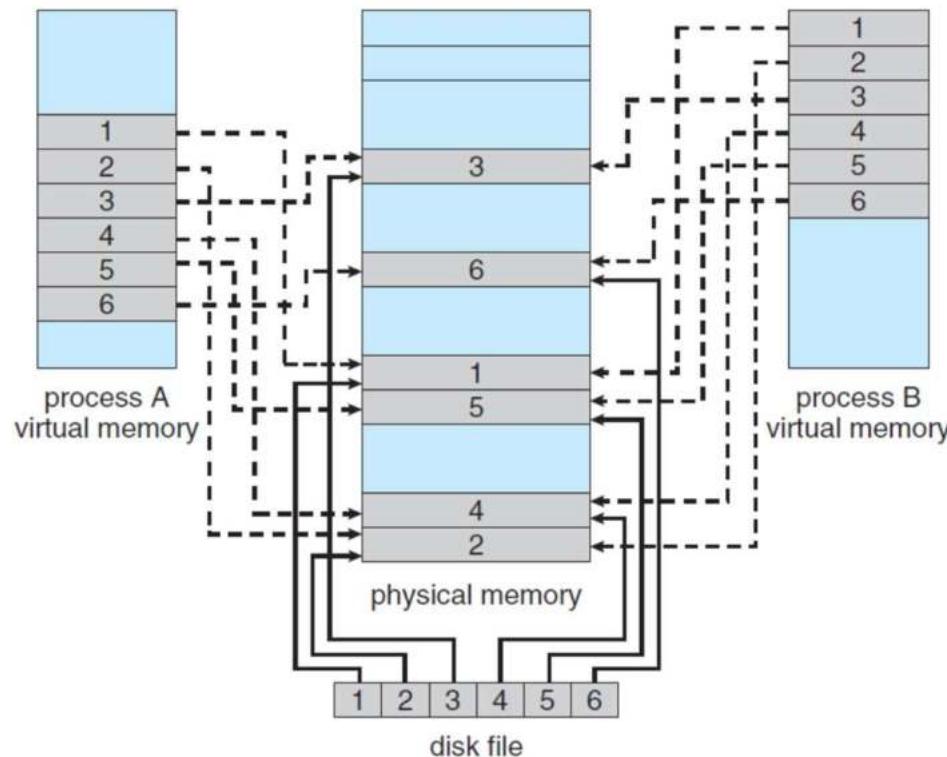
- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging
 - A page-sized portion of the file is read from the file system into a physical page
 - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies and speeds file access by driving file I/O through memory rather than `read()` and `write()` system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- But when does written data make it to disk?
 - Periodically and / or at file `close()` time
 - For example, when the pager scans for dirty pages

Memory-Mapped File Technique for all I/O

- Some OSes uses memory mapped files for standard I/O
- Process can explicitly request memory mapping a file via `mmap()` system call
 - Now file mapped into process address space
- For standard I/O (`open()`, `read()`, `write()`, `close()`), `mmap` anyway
 - But map file into kernel address space
 - Process still does `read()` and `write()`
 - ▶ Copies data to and from kernel space and user space
 - Uses efficient memory management subsystem
 - ▶ Avoids needing separate subsystem
- COW can be used for read/write non-shared pages
- Memory mapped files can be used for shared memory (although again via separate system calls)

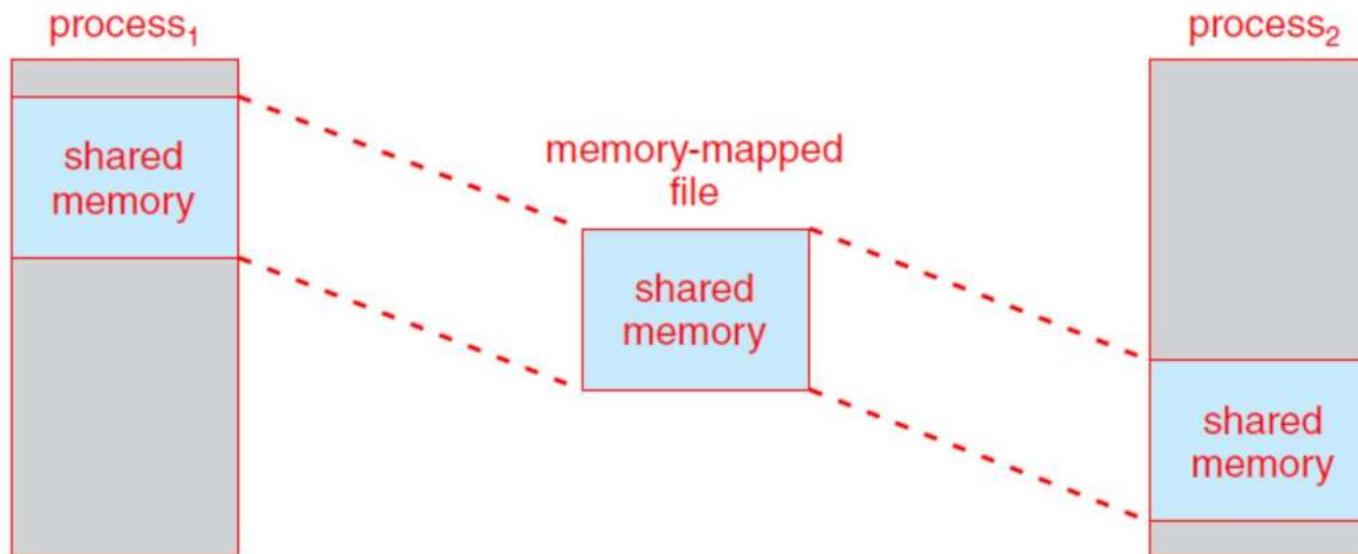
Memory Mapped File

- Con il metodo memory mapped una parte dell'indirizzo virtuale usato per accedere ai file
- Disk block mappato su una pagina in memoria
 - Memory mapping come meccanismo che supporta anche la shared memory
 - Le pagine in memoria virtuale puntano alle stesse pagine in memoria fisica



Memory Mapped File

- Con il metodo memory mapped una parte dell'indirizzo virtuale usato per accedere ai file
- Disk block mappato su una pagina in memoria
 - ▶ Memory mapping file come meccanismo che supporta anche la shared memory
 - ▶ Supporta la comunicazione interprocesso



Shared Memory in Windows API

- First create a **file mapping** for file to be mapped
 - Then establish a view of the mapped file in process's virtual address space
- Consider producer / consumer
 - Producer create shared-memory object using memory mapping features
 - Open file via `CreateFile()`, returning a `HANDLE`
 - Create mapping via `CreateFileMapping()` creating a **named shared-memory object**
 - Create view via `MapViewOfFile()`
- Sample code in Textbook

Memory Mapped File

- Memory Mapped File con Windows API
- Producer

Crea il mapping (named shared memory)

MapViewOfFile definisce una vista del mapped file in virtual memory (tutto o porzione, per porzione offset e size)

Scrittura su shared mem

Toglie il mapping e chiude

```
#include <windows.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    HANDLE hFile, hMapFile;
    LPVOID lpMapAddress;

    hFile = CreateFile("temp.txt", /* file name */
                      GENERIC_READ | GENERIC_WRITE, /* read/write access */
                      0, /* no sharing of the file */
                      NULL, /* default security */
                      OPEN_ALWAYS, /* open new or existing file */
                      FILE_ATTRIBUTE_NORMAL, /* routine file attributes */
                      NULL); /* no file template */

    hMapFile = CreateFileMapping(hFile, /* file handle */
                                NULL, /* default security */
                                PAGE_READWRITE, /* read/write access to mapped pages */
                                0, /* map entire file */
                                0,
                                TEXT("SharedObject")); /* named shared memory object */

    lpMapAddress = MapViewOfFile(hMapFile, /* mapped object handle */
                                FILE_MAP_ALL_ACCESS, /* read/write access */
                                0, /* mapped view of entire file */
                                0,
                                0);

    /* write to shared memory */
    sprintf(lpMapAddress, "Shared memory message");

    UnmapViewOfFile(lpMapAddress);
    CloseHandle(hFile);
    CloseHandle(hMapFile);
}
```

Memory Mapped File

- Memory Mapped File
in Windows

- Consumer

Apre il file

MapViewofFile definisce una
view del mapped file

Lettura da shared mem

Toglie mapping e chiude

```
#include <windows.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    HANDLE hMapFile;
    LPVOID lpMapAddress;

    hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS, /* R/W access */
        FALSE, /* no inheritance */
        TEXT("SharedObject")); /* name of mapped file object */

    lpMapAddress = MapViewOfFile(hMapFile, /* mapped object handle */
        FILE_MAP_ALL_ACCESS, /* read/write access */
        0, /* mapped view of entire file */
        0,
        0);

    /* read from shared memory */
    printf("Read message %s", lpMapAddress);

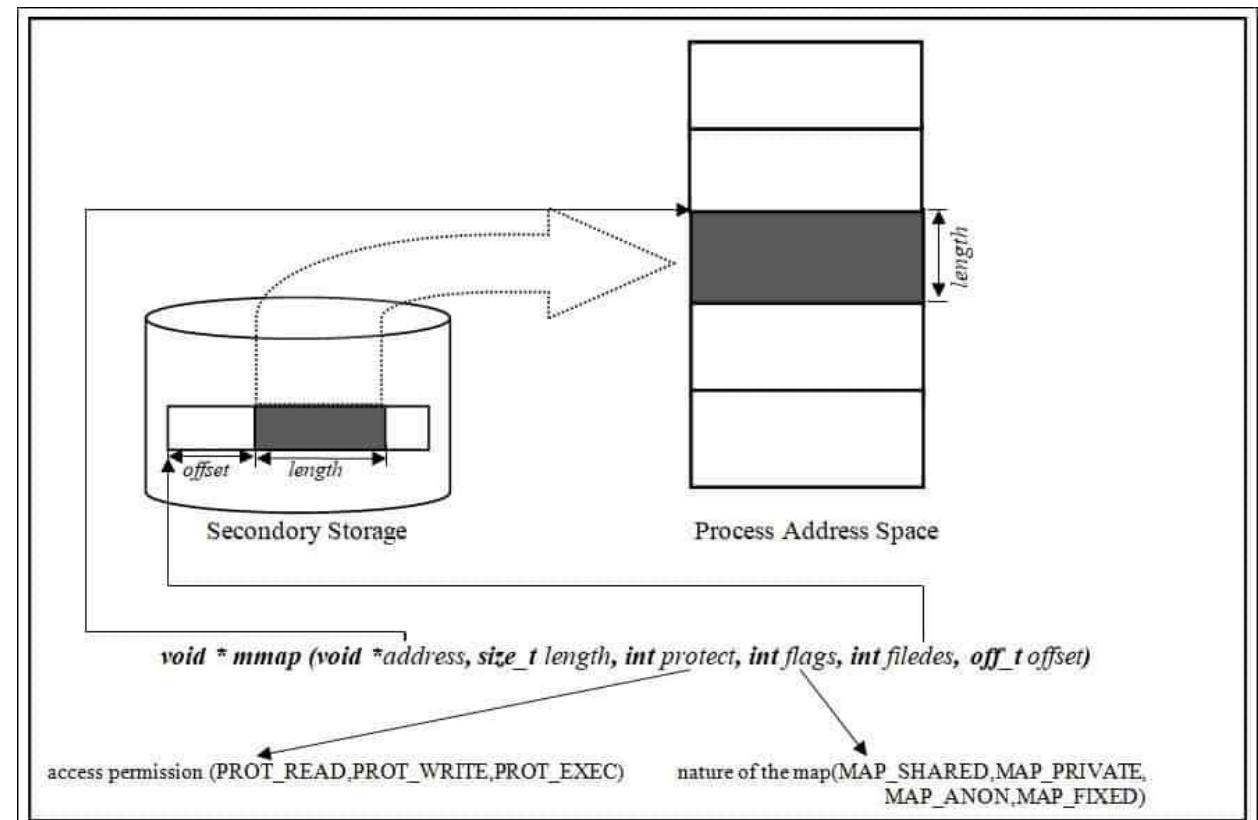
    UnmapViewOfFile(lpMapAddress);
    CloseHandle(hMapFile);
}
```

Memory Mapped File

□ Memory Mapped File in Linux

```
#include <sys/mman.h>
```

```
void * mmap (void *address, size_t length, int protect, int flags, int filedes, off_t offset)
```



Memory Mapped File

□ Memory Mapped File in Linux

```
#include <stdio.h>
#include <sys/mman.h>

int main(){

    int N=5;
    int *ptr = mmap ( NULL, N*sizeof(int), PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, 0, 0 );
    if(ptr == MAP_FAILED){
        printf("Mapping Failed\n");
        return 1;
    }
    for(int i=0; i<N; i++)
        ptr[i] = i*10;

    for(int i=0; i<N; i++)
        printf("[%d] ",ptr[i]);

    printf("\n");
    int err = munmap(ptr, 10*sizeof(int));
    if(err != 0){
        printf("UnMapping Failed\n");
        return 1;
    }

    return 0;
}
```

Memory Mapped File

□ Memory Mapped File in Linux

```
int main(int argc, char *argv[]){
    if(argc < 2){
        printf("File path not mentioned\n");
        exit(0);
    }

    const char *filepath = argv[1];
    int fd = open(filepath, O_RDONLY);
    if(fd < 0){
        printf("\n\"%s \" could not open\n",
               filepath);
        exit(1);
    }

    struct stat statbuf;
    int err = fstat(fd, &statbuf);
    if(err < 0){
        printf("\n\"%s \" could not open\n", filepath);
        exit(2);
    }
}
```

fstat prende lo
stato del file

```
char *ptr = mmap(NULL,statbuf.st_size,
                  PROT_READ|PROT_WRITE,MAP_SHARED, fd,0);
if(ptr == MAP_FAILED){
    printf("Mapping Failed\n");
    return 1;
}
close(fd);

ssize_t n = write(1,ptr,statbuf.st_size);
if(n != statbuf.st_size){
    printf("Write failed");
}

err = munmap(ptr, statbuf.st_size);
if(err != 0){
    printf("UnMapping Failed\n");
    return 1;
}
return 0;
}
```

Memory Mapped File

□ Memory Mapped File in Linux

```
int main(int argc, char *argv[]){
    if(argc < 2){
        printf("File path not mentioned\n");
        exit(0);
    }

    const char *filepath = argv[1];
    int fd = open(filepath, O_RDWR);
    if(fd < 0){
        printf("\n\"%s \" could not open\n",
               filepath);
        exit(1);
    }
    struct stat statbuf;
    int err = fstat(fd, &statbuf);
    if(err < 0){
        printf("\n\"%s \" could not open\n", filepath);
        exit(2);
    }
    char *ptr = mmap(NULL,statbuf.st_size,
                     PROT_READ|PROT_WRITE, MAP_SHARED, fd,0);

    if(ptr == MAP_FAILED){
        printf("Mapping Failed\n");
        return 1;
    }
    close(fd);
    ssize_t n = write(1,ptr,statbuf.st_size);
    if(n != statbuf.st_size){
        printf("Write failed\n");
    }
    // Reverse the file contents
    for(size_t i=0; i < n);
    n = write(1,ptr,statbuf.st_size);
    if(n != statbuf.st_size){
        printf("Write failed\n");
    }
    err = munmap(ptr, statbuf.st_size);
    if(err != 0){
        printf("UnMapping Failed\n");
        return 1;
    }
    return 0;
}
```

Lezione 1: Introduzione

Obiettivi

- Introduzione al concetto di Sistema Operativo
- Storia dei Sistemi Operativi e nozioni principali
- Descrivere l'organizzazione base di un Calcolatore
- Introdurre le principali componenti di un Sistema Operativo

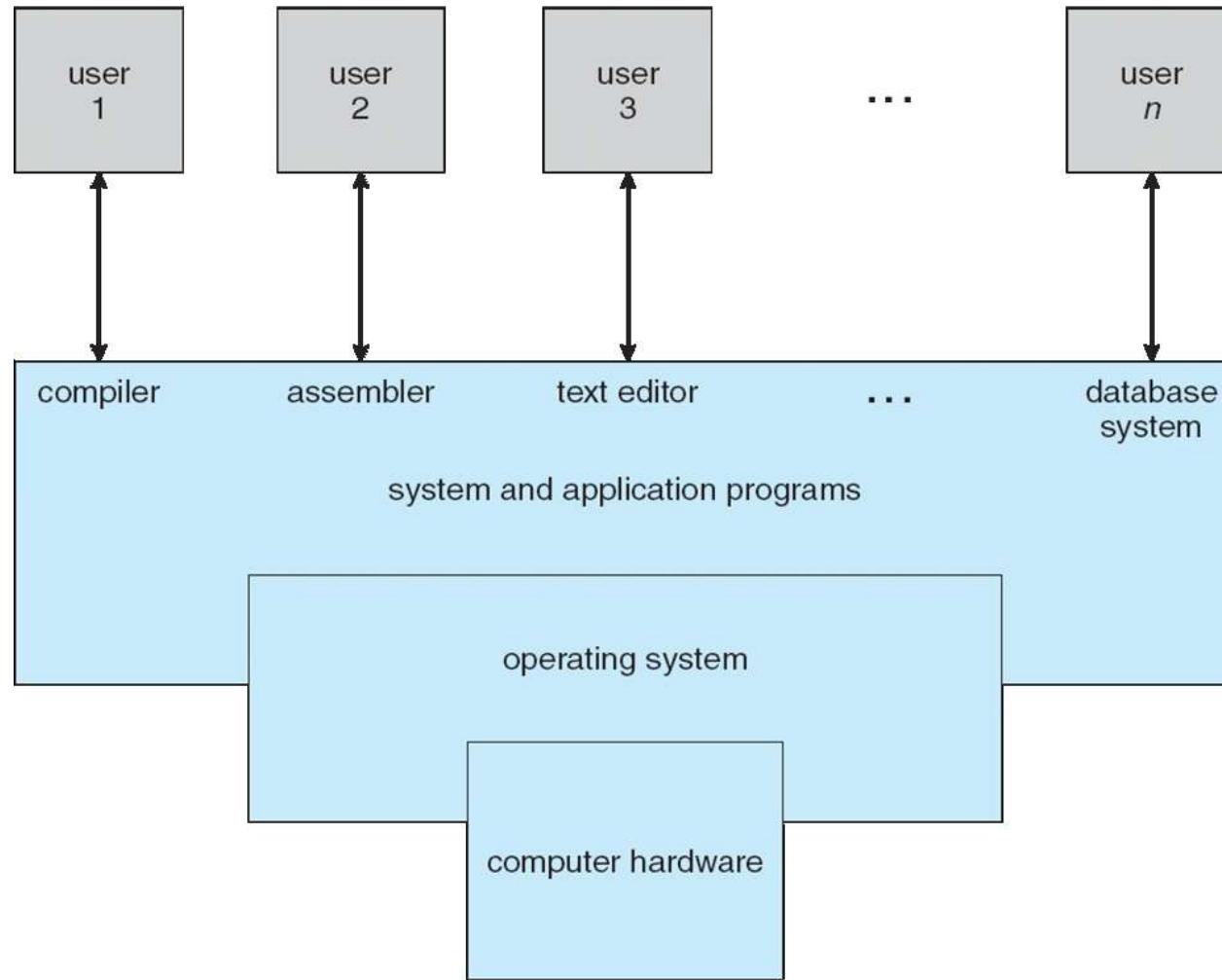
Cos'è un Sistema Operativo?

- Un programma che gestisce le risorse di un calcolatore e fa da intermediario tra l'utente di un calcolatore e l'hardware del calcolatore
- Obiettivi di un Sistema Operativo:
 - Eseguire i programmi utente e semplificare l'interazione con il calcolatore
 - Rendere efficace ed efficiente l'utilizzo del calcolatore
 - Utilizzare l'hardware in modo efficiente

Componenti di un Sistema di Calcolo

- Si possono identificare quattro componenti:
 - Hardware – fornisce le risorse computazionali di base
 - ▶ CPU, memoria, dispositivi I/O
 - Sistema Operativo
 - ▶ Controlla e coordina l'uso dell'hardware tra diverse applicazioni e diversi utenti
 - Programmi applicativi – usano le risorse di sistema per fornire le elaborazioni richieste dagli utenti
 - ▶ Word processors, compilers, web browsers, database systems, video games
 - Utenti
 - ▶ Persone, altri processi, altri calcolatori, etc.

Componenti di un Sistema di Calcolo

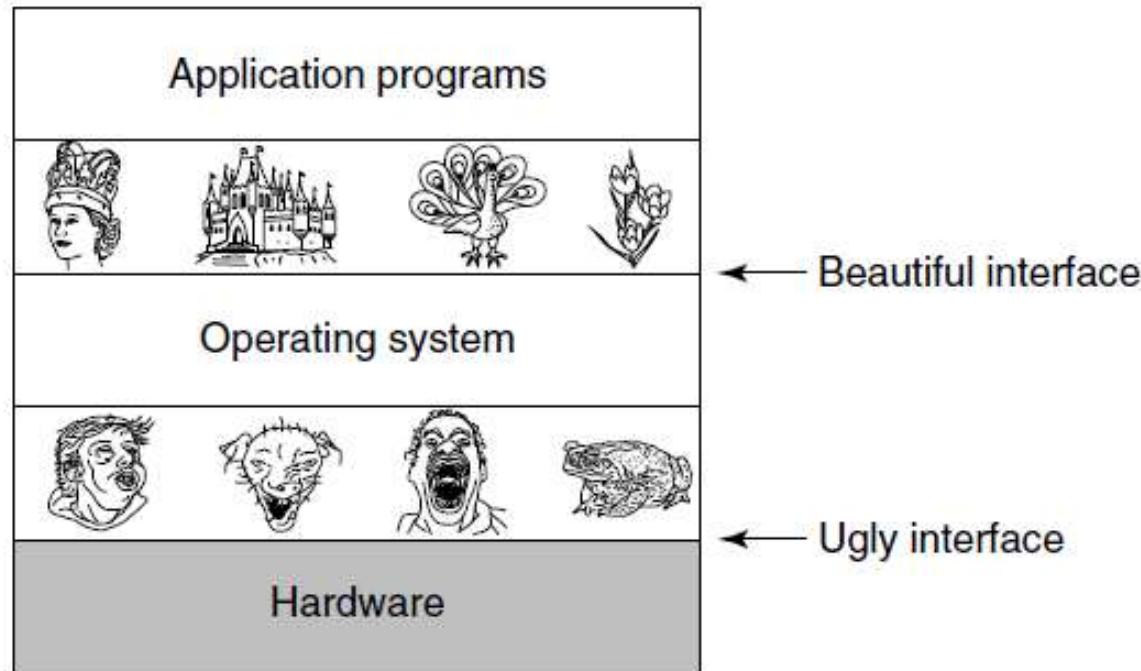


Definizione di un Sistema Operativo

- L'SO è un **allocatore di risorse**
 - Gestisce tutte le risorse del calcolatore
 - Decide tra richieste conflittuali per l'assegnazione corrette ed efficiente delle risorse
- L'SO è un **programma di controllo**
 - Controlla l'esecuzione dei programmi evitando errori e usi impropri del calcolatore
- L'SO come **macchina estesa**
 - Fornisce un'**astrazione** fornendo un ambiente coerente ed uniforme per l'esecuzione dei programmi

SO come Macchina Estesa

- Sistema Operativo trasforma il brutto hardware in belle astrazioni
(Tanenbaum et al. 2013)



Tanenbaum & Bo, Modern Operating Systems:4th ed., (c) 2013 Prentice-Hall, Inc. All rights reserved.

Definizione di Sistema Operativo

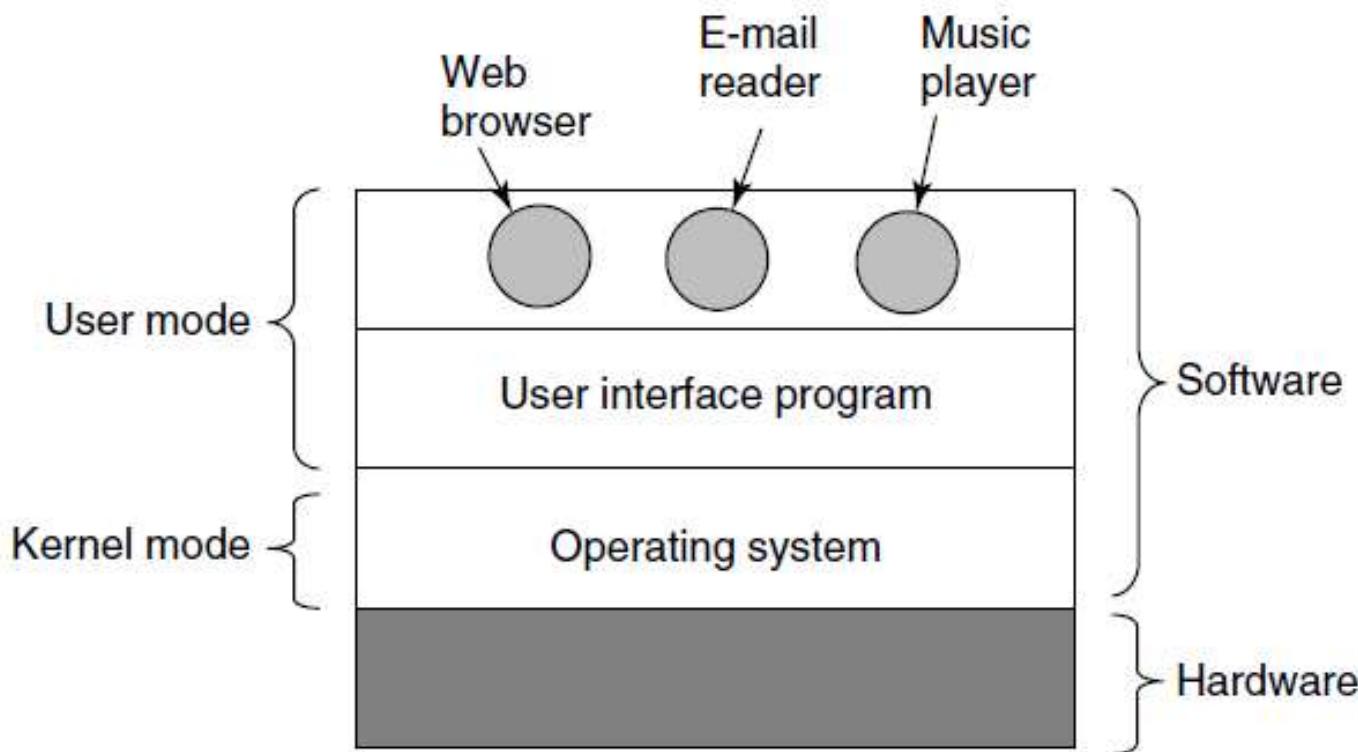
- Non c'è una definizione universalmente accettata
- “Everything a vendor ships when you order an operating system”
(Silberschatz et al.) è una buona approssimazione
- A rigore il Sistema Operativo è il programma continuamente in esecuzione che gestisce le risorse del calcolatore:
 - Chiamato il **nucleo (kernel)**
 - Tutto il resto:
 - ▶ se non è **programma di sistema** (utilità fornite con il SO) ...
 - ▶ ... è un **programma applicativo**
- Gli argomenti del corso vertono soprattutto su **kernel** di SO general-purpose

Nucleo del Sistema Operativo

- Il **nucleo** del sistema (**kernel**) gestisce le risorse essenziali: la CPU, la memoria, le periferiche, etc.
- Tutto il resto, anche l'interazione con l'utente, è ottenuto tramite programmi eseguiti dal kernel, che accedono alle risorse hardware tramite delle richieste a quest'ultimo.

Il kernel è il solo programma ad essere eseguito in modalità privilegiata, con il completo accesso all'hardware, gli altri programmi vengono eseguiti in modalità protetta.

Nucleo del Sistema Operativo



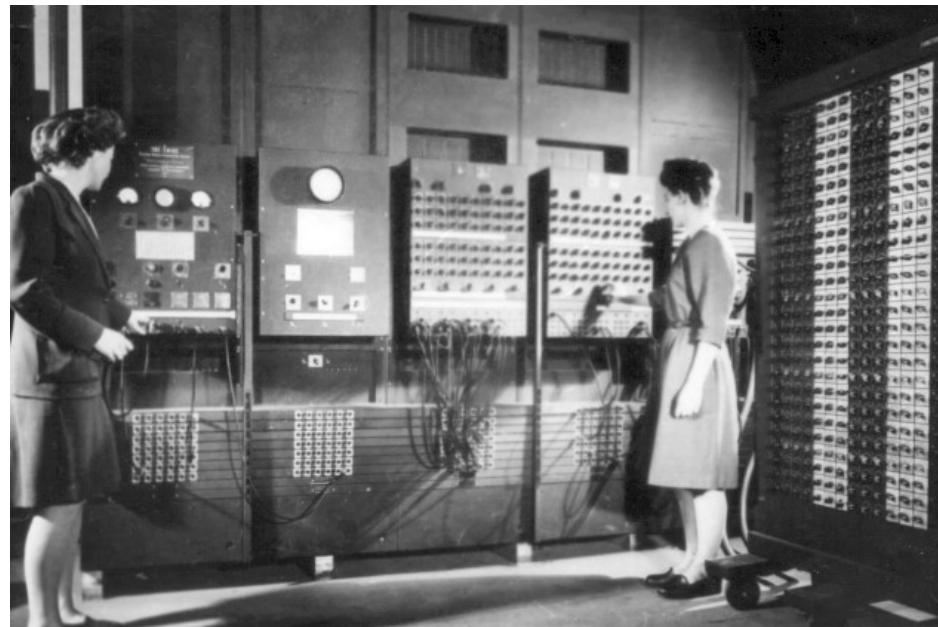
Tanenbaum & Bo, Modern Operating Systems: 4th ed., (c) 2013 Prentice-Hall, Inc. All rights reserved.

Storia Sistemi Operativi

- Prima generazione (1945–55) valvole
- Seconda generazione (1955–65) transistor e sistemi batch
- Terza generazione (1965–1980) Circuiti integrati e multiprogramming
- Quarta generazione (1980–presente) personal computer
- Quinta generazione (1990–presente) mobile computer

Storia Sistemi Operativi

- Prima generazione (1945–55) valvole
 - Non esiste Sistema Operativo, gestione manuale
 - Operatore e programmatore coincidenti
 - Prenotazione della macchina e uso esclusivo
 - Esecuzione programma da console
 - ▶ Programma caricato in memoria un'istruzione alla volta
 - ▶ Controllo errori su spie della console



Storia Sistemi Operativi

- Prima generazione (1945–55) valvole
 - Prima evoluzione
 - ▶ Diffusione di periferiche (lettore/perforatore di schede, nastri, stampanti)
 - ▶ Programmi di interazione con periferiche (device driver)
 - ▶ Sviluppo di primo “software” di supporto
 - Librerie, compilatori, linker, loader

Storia Sistemi Operativi

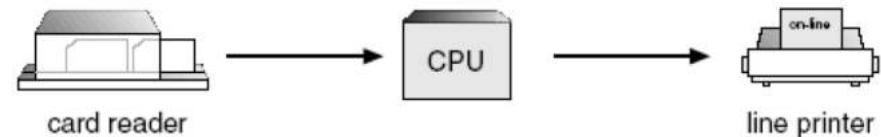
- Seconda generazione (1955–65) transistor e sistemi batch
 - Separazione di operatore e programmatore
 - Eliminazione dello schema a prenotazione
 - Operatore elimina parte dei tempi morti
 - Batching dei lavori: **batch** = lotto
 - Raggruppamento di programmi (**job**) simili nell'esecuzione
 - Monitor residente (primi esempi di Sistemi Operativi)



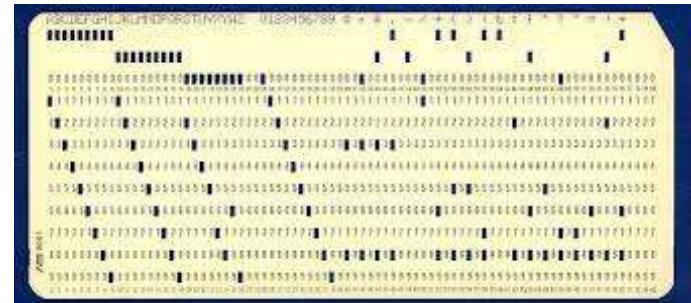
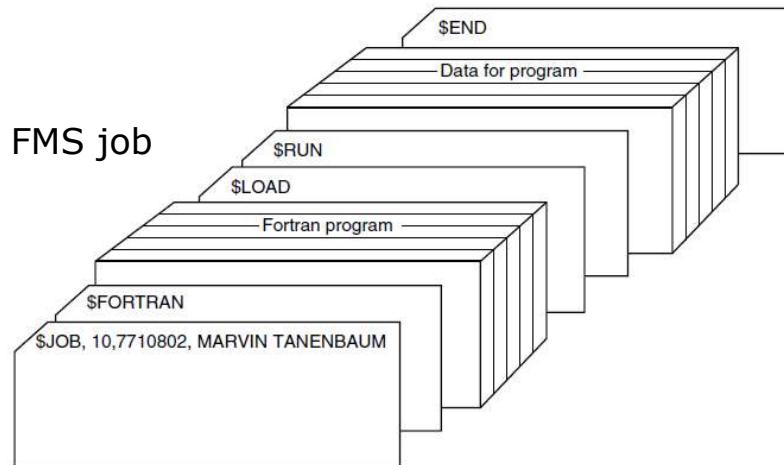
IBM 7094 (1962)

Storia Sistemi Operativi

- Seconda generazione (1955–65) transistor e sistemi batch
 - Raggruppamento in lotti dei lavori
 - Gestione automatica dei job
 - Job scritti su schede perforate
 - Job Control Language
 - Monitor residente (in memoria)
 - ▶ Primi esempi di Sistema Operativo per sequenziamento
 - ▶ Es. Fortran Monitoring System (FMS), IBSYS, etc.



Struttura di un FMS job

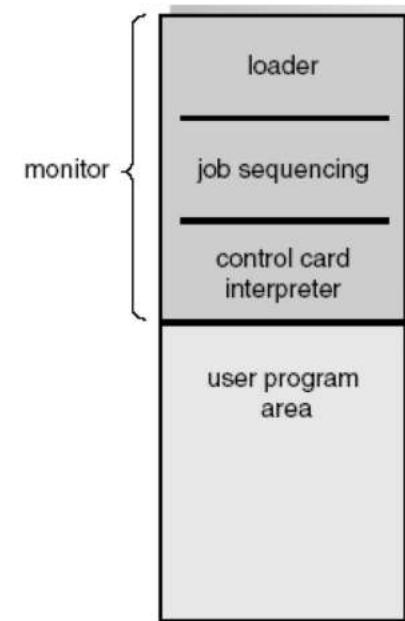


Esempio di scheda

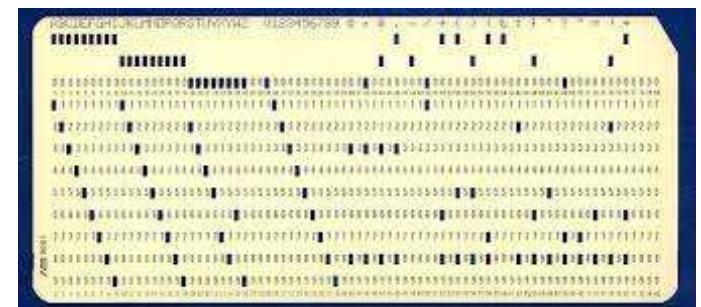
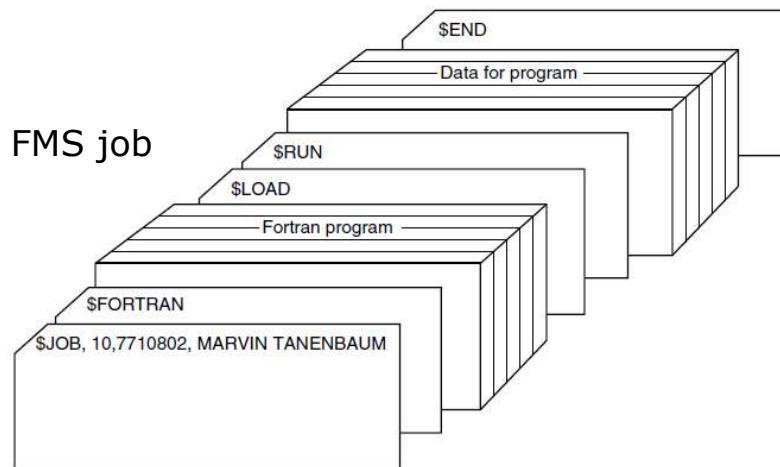
Storia Sistemi Operativi

□ Seconda generazione (1955–65) transistor e sistemi batch

- Raggruppamento in lotti dei job
- Gestione automatica dei job
- Monitor residente in memoria
 - ▶ Primi esempi di Sistema Operativo
 - ▶ Es. Fortran Monitoring System (FMS), IBSYS, etc.
 - ▶ Sequenziatore, interprete schede, caricature
 - ▶ Programmi descritti da istruzione su schede perforate
 - ▶ Job descrivono come eseguire i programmi



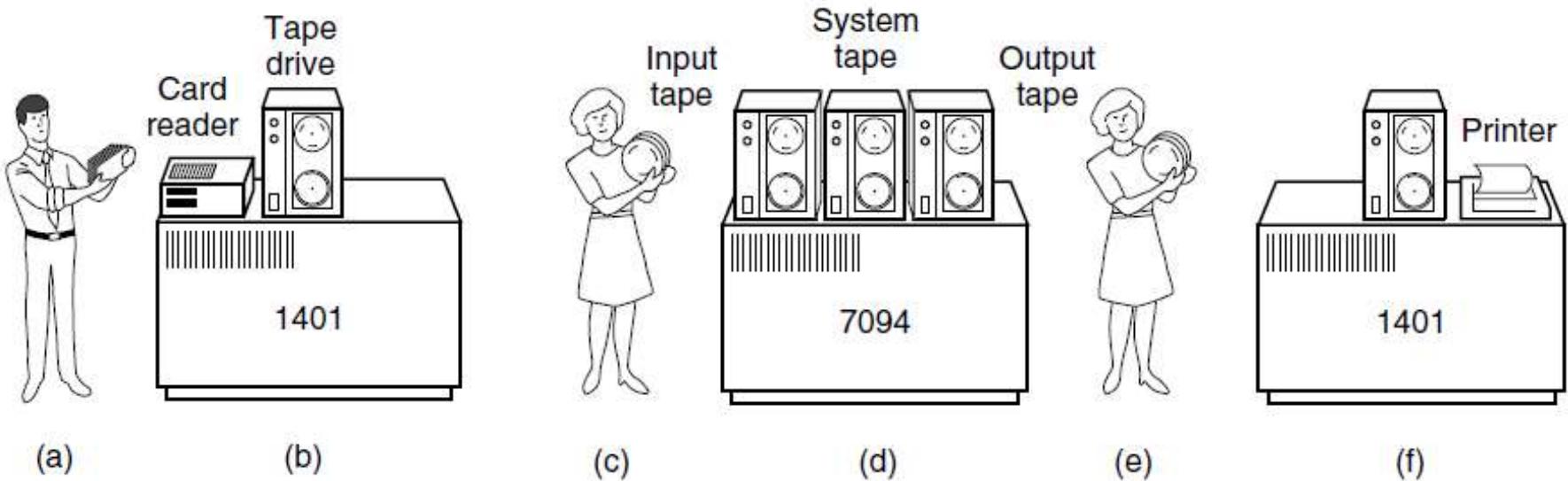
Struttura di un FMS job



Esempio di scheda

Storia Sistemi Operativi

- Seconda generazione (1955–65) transistor e sistemi batch
 - Operazioni I/O lente rispetto a CPU
 - Gestione off-line di operazione I/O



- (a) I programmatori portano le schede al 1401. (b) 1401 legge i lotti (batch) di lavori (jobs) in un nastro
(c) L'operatore porta nastri di input e al 7094. (d) 7094 elabora i lotti e stampa su nastro l'output
(e) l'operatore porta i nastri di output al 1401. (f) 1401 stampa l'output.

Tanenbaum & Bo, Modern Operating Systems:4th ed., (c) 2013 Prentice-Hall, Inc. All rights reserved.

Storia Sistemi Operativi

□ Evoluzione dei sistemi batch

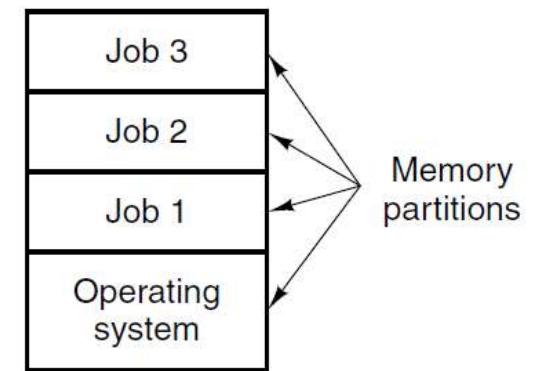
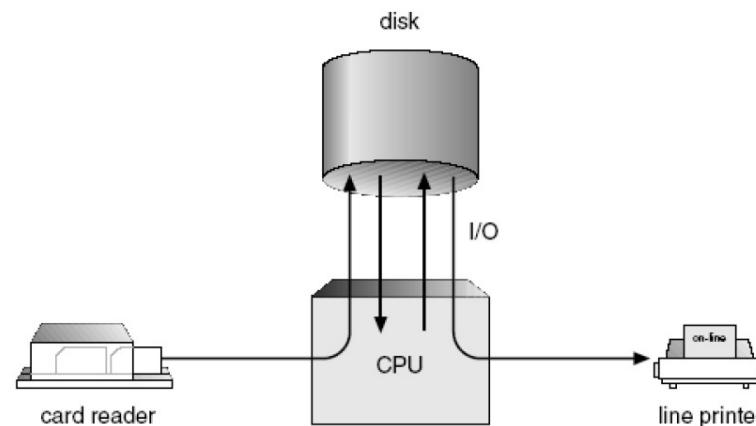
- CPU ed I/O con una sola macchina ma disaccoppiate
- Architettura più complessa per gestione delle periferiche

□ **Buffering**

- Buffer per ogni dispositivo periferico per disaccoppiare letture/scritture

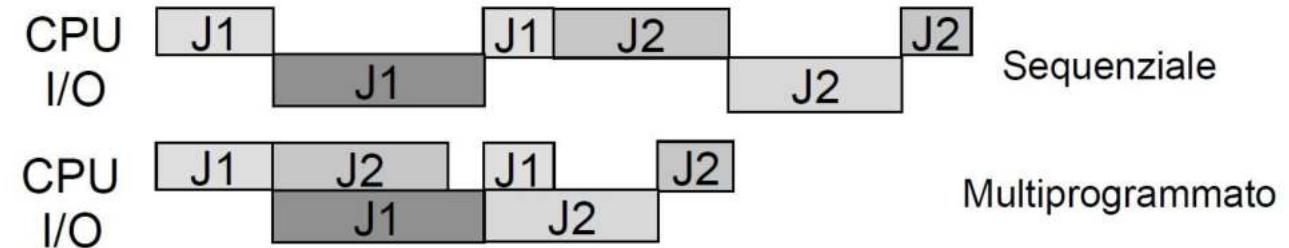
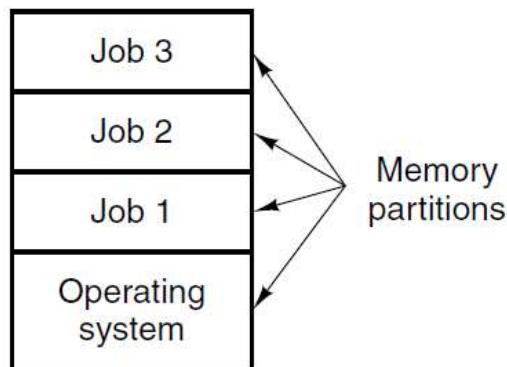
□ **Spooling** (Simultaneous Peripheral Operation On Line)

- Possibile grazie “ad accesso casuale” dei dischi
- Utilizzo del disco come un grande buffer per tutti i job
- Job registrati su unità a disco e S.O. accede in modo diretto tramite una tabella
- Concetto di **pool di job** e **job scheduling**



Storia Sistemi Operativi

- Terza generazione (1965–1980) ICs e multiprogramming
 - Buffering e Spooling
 - Buffering per parallelizzare I/O e calcolo di un job
 - Spooling (Simultaneous Peripheral Operation On Line) parallelizza più job
 - **Multiprogrammazione** (processamento multiplo)
 - Più job contemporaneamente in competizione (pool di job)
 - CPU continuamente impegnata e passa da job a job
 - **Job scheduling** (cosa deve essere caricato per l'esecuzione)
 - **CPU scheduling** (cosa deve essere processato)
 - **Timesharing** (CTSS at M.I.T by Corbató et al. 1962)
 - Gestione esecuzione per utenti multipli (multiutenti)



Storia Sistemi Operativi

- Terza generazione (1965–1980) ICs e multiprogramming
 - Multiprogrammazione e Multiutente
 - Più processi e più utenti contemporaneamente
 - Condivisione delle risorse della macchina
 - **Meccanismi di protezione**
 - Protezione I/O, Memoria, CPU
 - S.O. deve sempre mantenere il controllo
 - **Modalità Operativa Duale** (Dual Mode)
 - Superuser e User
 - Richieste al Sistema
 - Protezione hardware
 - Gestione memoria dedicate ai processi
 - Gestione slot di tempo dedicato ad ogni processo
 - Gestione dei permessi degli utenti
 - Etc.

Storia Sistemi Operativi

- Terza generazione (1965–1980) ICs e multiprogramming
 - **MULTICS** (MULTIplexed Information and Computing Service)
 - ▶ MIT, Bell Labs and General Electric 1964
 - ▶ Prodotto commerciale di GE nel 1967 poi Honeywell
 - ▶ Bell Labs esce dal progetto nel 1969, ma parte l'iniziativa Unix
 - ▶ Multiutente, Multiprogrammazione, File System Multilivello
 - ▶ Molto complesso, limitata diffusione commerciale

Storia Sistemi Operativi

- Terza generazione (1965–1980) ICs e multiprogramming

- **Sistema Unix**

- ▶ Bell Labs esce nel 1969 ma parte l'iniziativa Unix
 - ▶ Progetto ripreso da Ken Thompson e Dennis Ritchie
 - ▶ Semplificazione dei concetti: Unix vs Multics
 - ▶ Sviluppato in linguaggio C
 - ▶ Multiutente, Multiprogrammazione, File System Multilivello, Shell come processo separato
 - ▶ Diversi sviluppi:
 - System V di AT&T, BSD (Berkeley Software Distribution)
 - Minix, Linux

Storia Sistemi Operativi

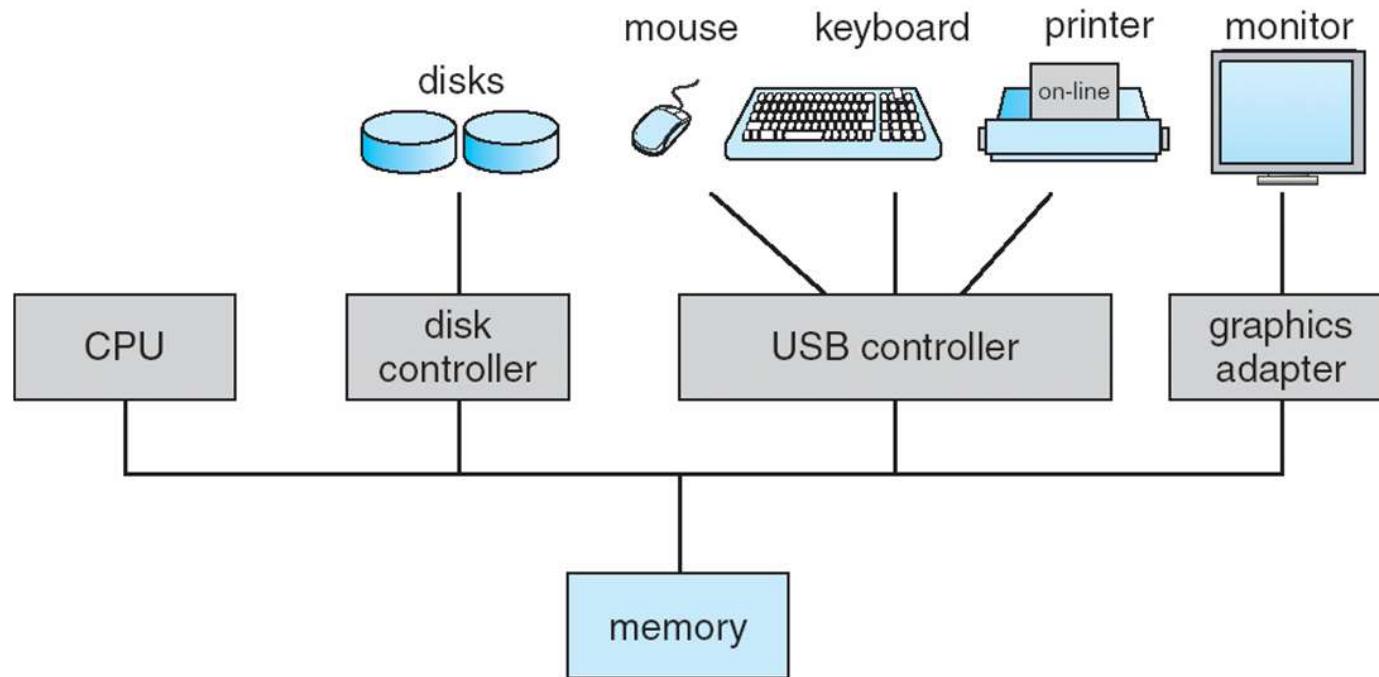
- Quarta generazione (1980–presente) personal computer
 - Circuiti LSI (Large Scale Integration)
 - Processore 8080 Intel
 - ▶ 1977 SO CP/M (Control Program for Microcomputers)
 - 1980 IBM Personal Computer
 - ▶ DOS (Disk Operating System)
 - ▶ MS-DOS (MicroSoft Disk Operating System)
 - GUI
 - ▶ Xerox PARC
 - ▶ Project Lisa and Apple Macintosh

Storia Sistemi Operativi

- Quinta generazione (1990–presente) mobile computer
 - PDA (Personal Digital Assistant), tablet, smartphone, smartwatch, etc.
 - Diversi SO
 - ▶ MS Windows
 - ▶ Android
 - ▶ Apple iOS
 - ▶ PureOS, Ubuntu Touch

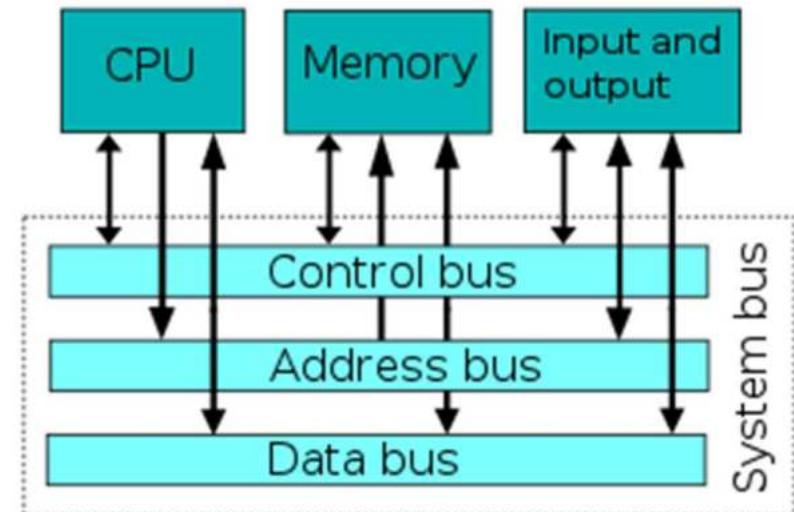
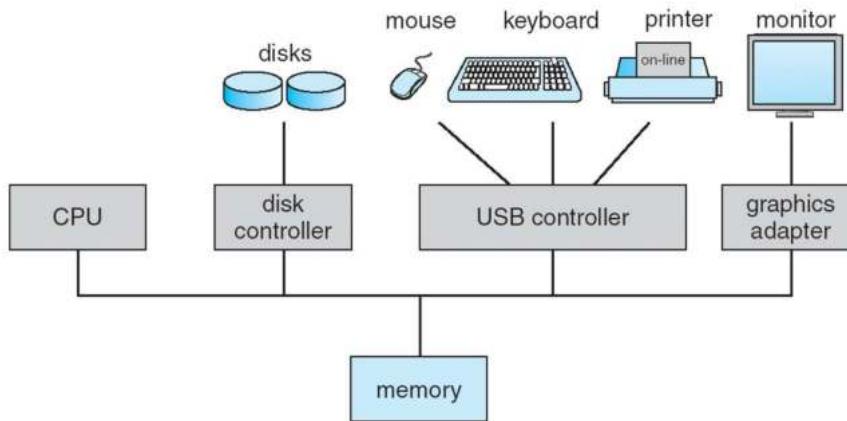
Organizzazione Calcolatore

- Il Sistema Operativo è strettamente legato all'architettura del calcolatore:
 - Una o più CPU ed i controllori di dispositivo (**driver**) sono connessi da un canale (**bus**) che permette l'accesso alla memoria condivisa
 - La CPU e i controllori possono eseguire operazioni in parallelo competendo per l'accesso alla memoria



Organizzazione Calcolatore

- Il Sistema Operativo è strettamente legato all'architettura del calcolatore
 - Una o più CPU ed i controllori di dispositivo (**driver**) sono connessi da un canale (**bus**) che permette l'accesso alla memoria condivisa
 - La CPU e i controllori possono eseguire operazioni in parallelo competendo per l'accesso alla memoria

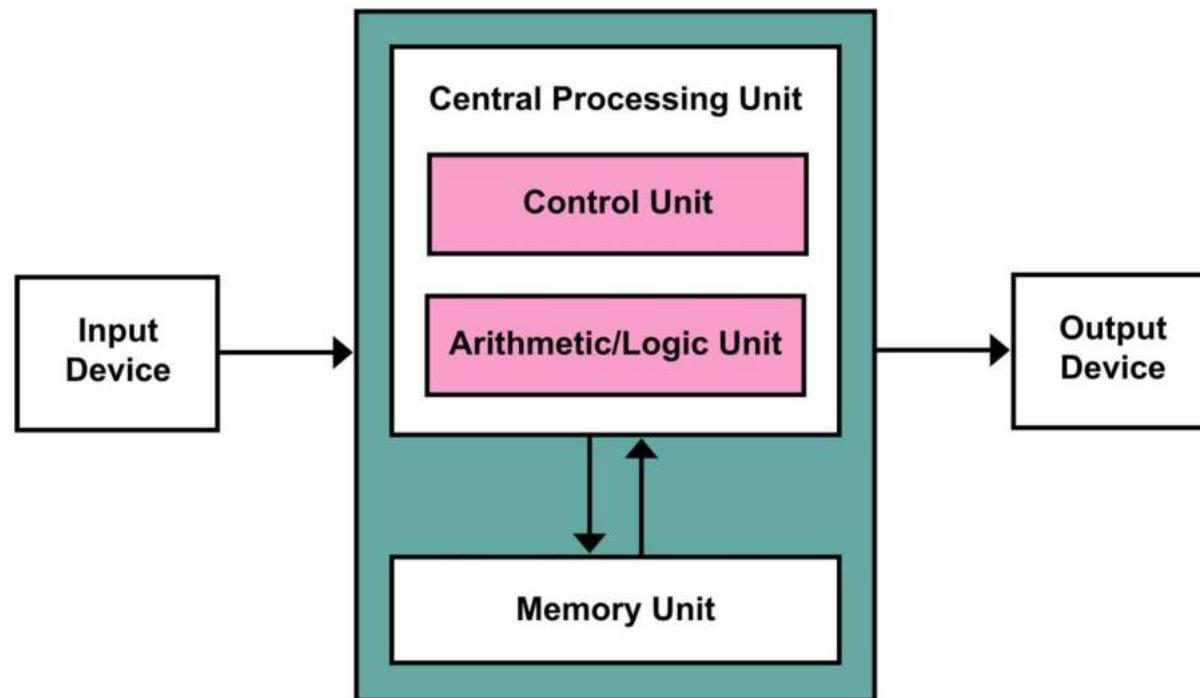


CPU

□ Central Processing Unit

□ Esegue le istruzioni dalla memoria:

- Ogni CPU ha un set di istruzioni che può eseguire
- Ciclo: fetch, decode, execute

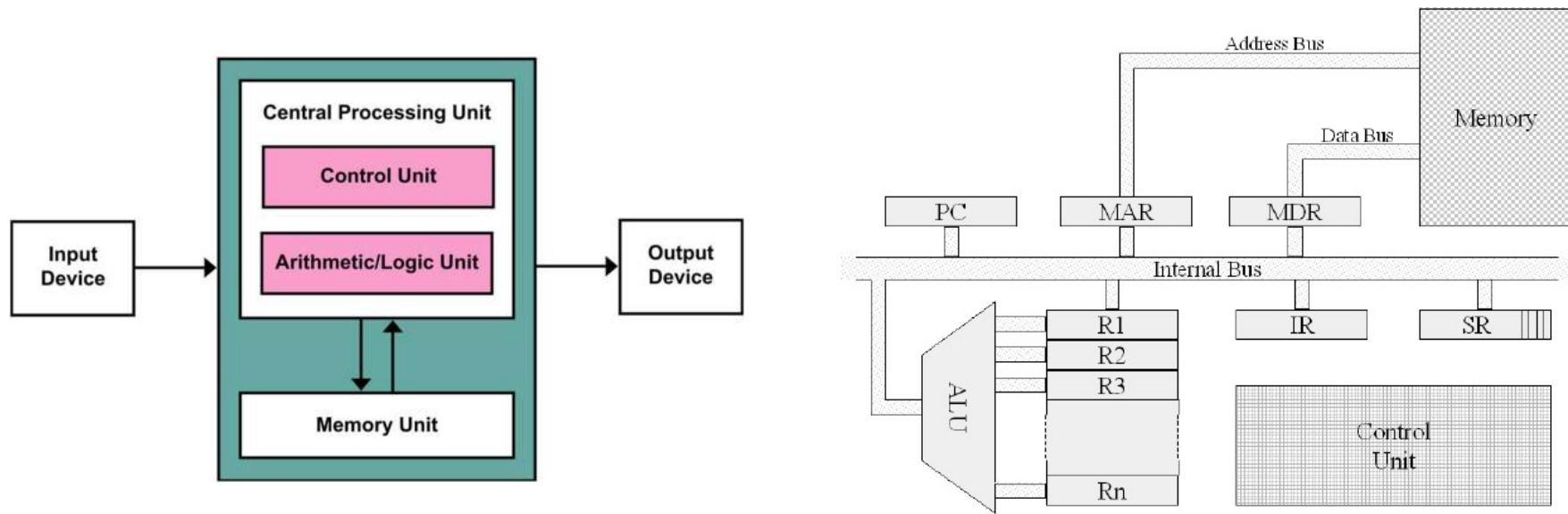


CPU

□ Central Processing Unit

□ Esegue le istruzioni dalla memoria:

- Ogni CPU ha un set di istruzioni che può eseguire
- Ciclo: fetch, decode, execute
- Registri:
 - Program Counter, Stack Pointer, Instruction Register, Status Register
Memory Add. Reg., Memory Data Reg.



CPU

- **CPU**: componente hardware che esegue le istruzioni
- **Processore**: chip che contiene una o più CPU
- **Unità di calcolo (core)**: unità di elaborazione di base della CPU
- **Multicore**: che include più unità di calcolo sulla stessa CPU
- **Multiprocessore**: che include più processori

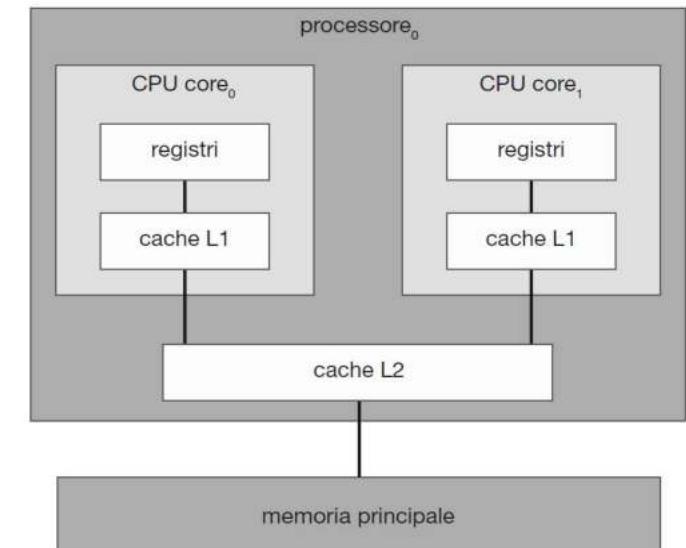
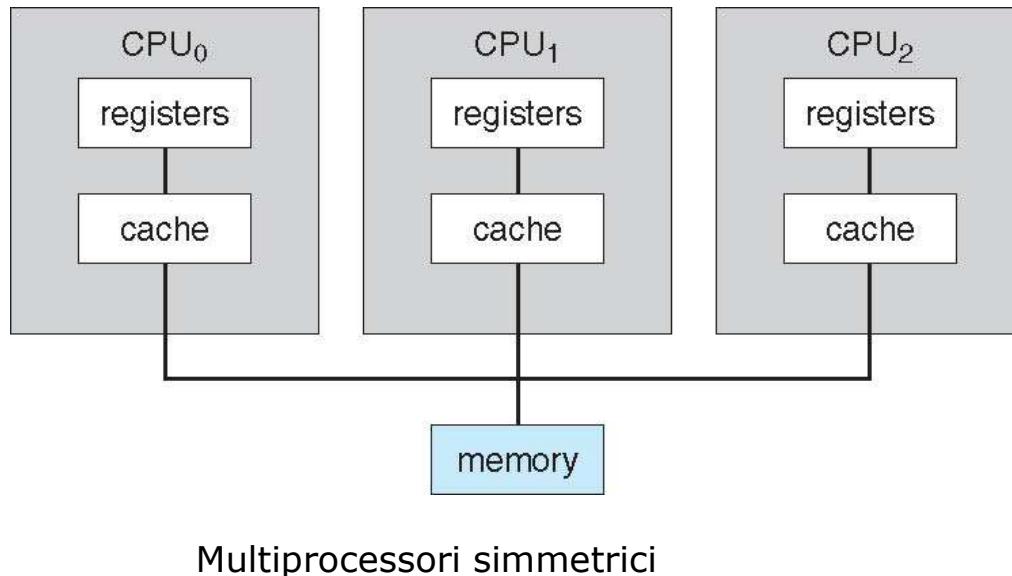
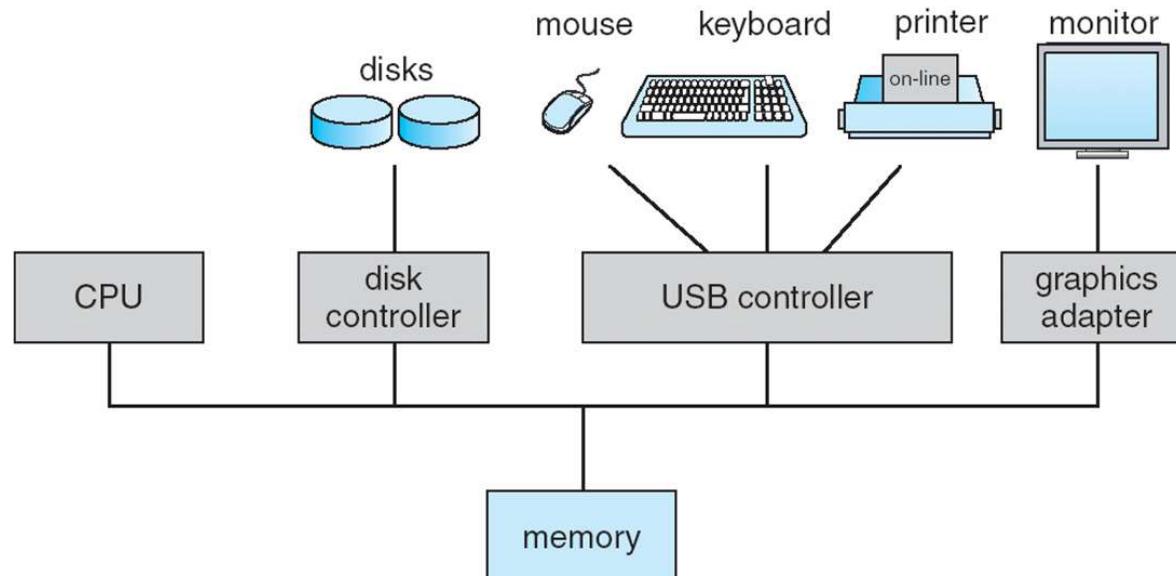


Figura 1.9 Architettura dual-core, con due unità sullo stesso chip.

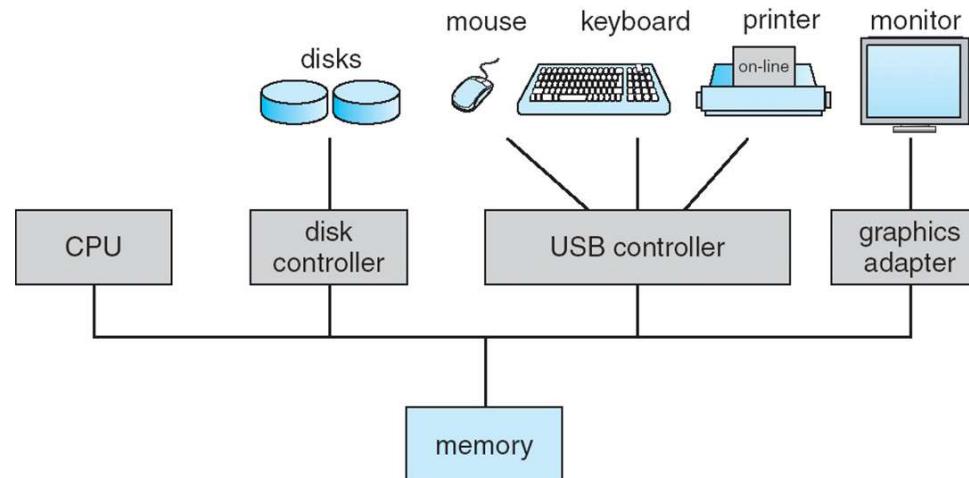
Sistemi con Interrupt

- I dispositivi I/O e la CPU sono in esecuzione concorrente
- Ogni **controllore di dispositivo** è responsabile di un tipo di dispositivo
- Ogni controllore ha un **buffer locale**
- CPU porta dati dalla/alla memoria ai/dai buffers locali
- I/O dal dispositivo ai buffer locali del controller
- Ogni controllore informa la CPU che ha finito le sue operazioni attraverso il meccanismo delle interruzioni (**interrupt**)



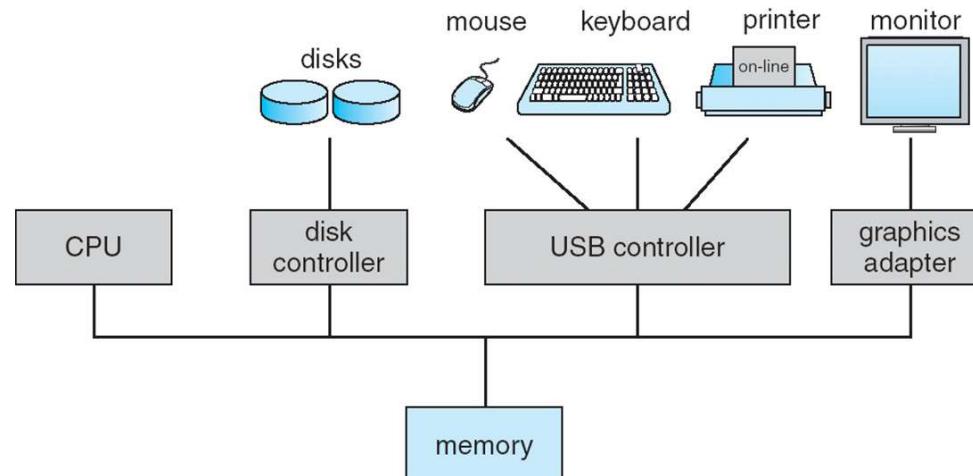
Interruzioni

- Le interruzioni permettono di gestire le operazioni concorrenti
 - Sovrapporre CPU e operazioni I/O
 - Evitare **busy waiting**
- Un **Sistema Operativo moderno è guidato dalle interruzioni**
 - Le interruzioni trasferiscono il controllo ad una procedura di servizio dell'interruzione
 - Una volta eseguita la routine di servizio il controllo ritorna all'operazione interrotta



Polling vs Interrupt

- Interrupt e polling sono due modalità con cui gli eventi generati dai dispositivi I/O possono essere gestiti dalla CPU
 - Con il polling, la CPU tiene traccia delle comunicazioni dei dispositivi di I/O interrogandoli ad intervalli regolari
 - Con interrupt, il dispositivo di I/O interrompe la CPU comunicando ad essa che ha bisogno di andare in esecuzione



Interrupt Timeline

- Per avviare I/O la CPU carica opportuni registri del controllore I/O
- La CPU continua il processamento ...
- intanto il controllore I/O avvia le operazioni sul proprio buffer ...
- ... quando ha terminato invia l'**interrupt**

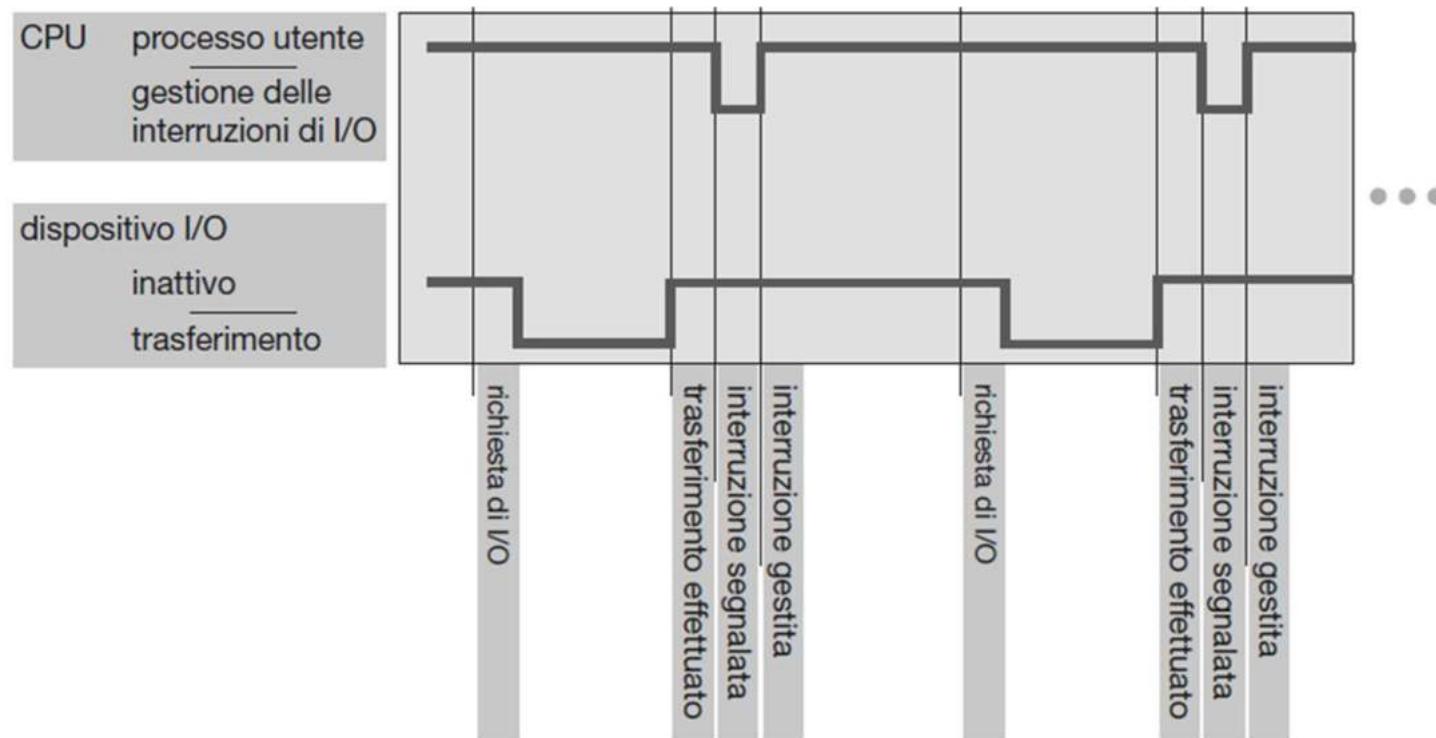


Figura 1.3 Diagramma temporale delle interruzioni per un singolo programma che invia dati in output.

Interrupt Timeline

- ... la CPU controlla la interruzione periodicamente
- Determina il tipo di interruzione e gestisce l'interruzione
 - **Gestore esecuzione** invoca la routine di servizio
- Gestita l'interruzione la CPU riprende l'elaborazione interrotta

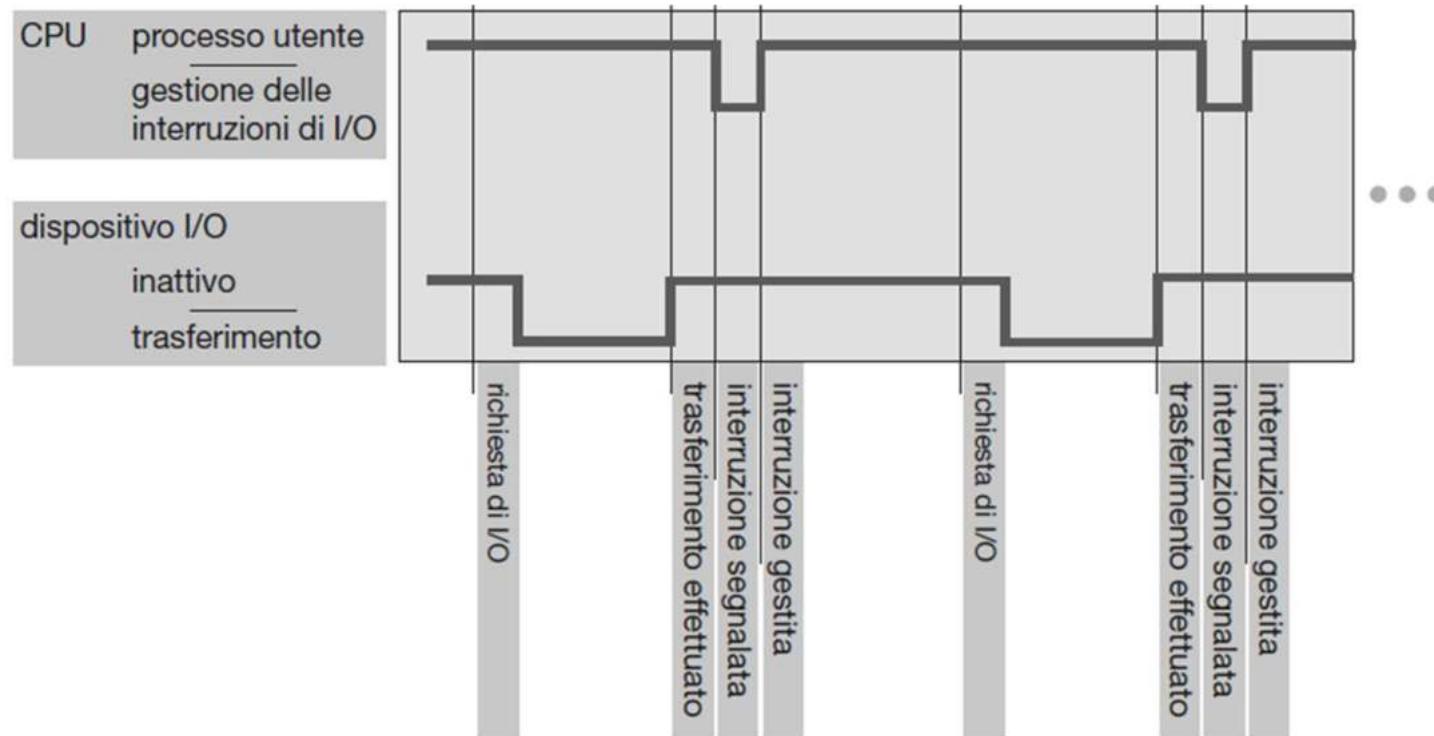


Figura 1.3 Diagramma temporale delle interruzioni per un singolo programma che invia dati in output.

Gestione delle Interruzioni

- Il dispositivo I/O invia un segnale su Interrupt Request Line (IRQ)
- La CPU controlla la IRQ per ogni istruzione ...
- Due tipi di interruzione (**nonmaskable/maskable**)
 - Non mascherabile (errori irreversibili)
 - Masherabile (utilizzata dai controllori di dispositivo)
- Per servire la richiesta la CPU:
 - Determina il tipo di interruzione
 - Passa il controllo alla procedura di servizio fornendone l'indirizzo iniziale
 - **vettore delle interruzioni:** fornisce direttamente inidirizzo proc. servizio
 - Il **gestore dell'interruzione** salva le informazioni di stato dell'operazione interrotta (registri e program counter)
 - La procedura di servizio viene eseguita, determina la causa dell'interruzione, processa, quindi il controllo torna all'operazione interrotta (recuperando lo stato esecutivo)

Gestione delle Interruzioni

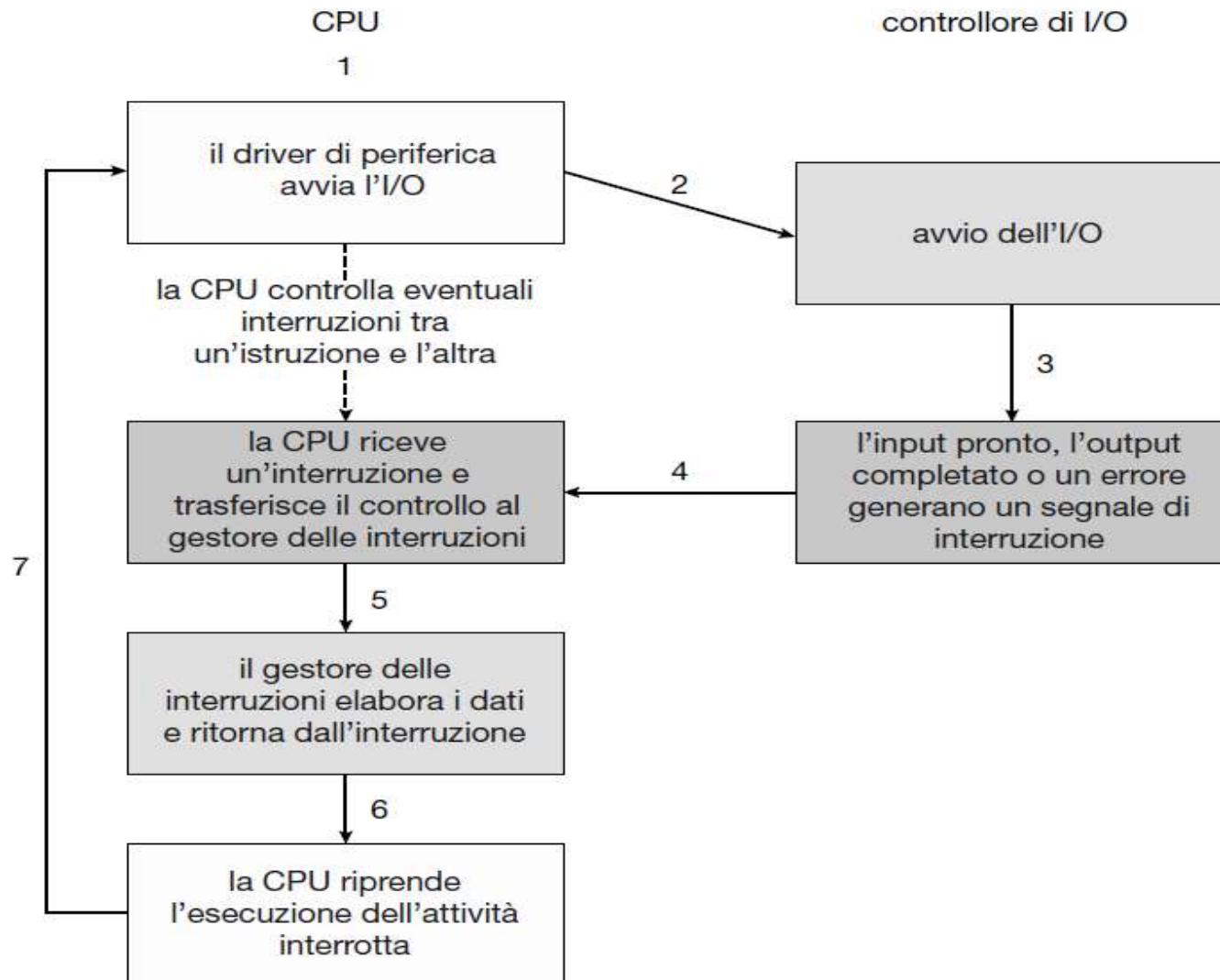


Figura 1.4 Ciclo di I/O guidato dalle interruzioni.

Gestione delle Interruzioni

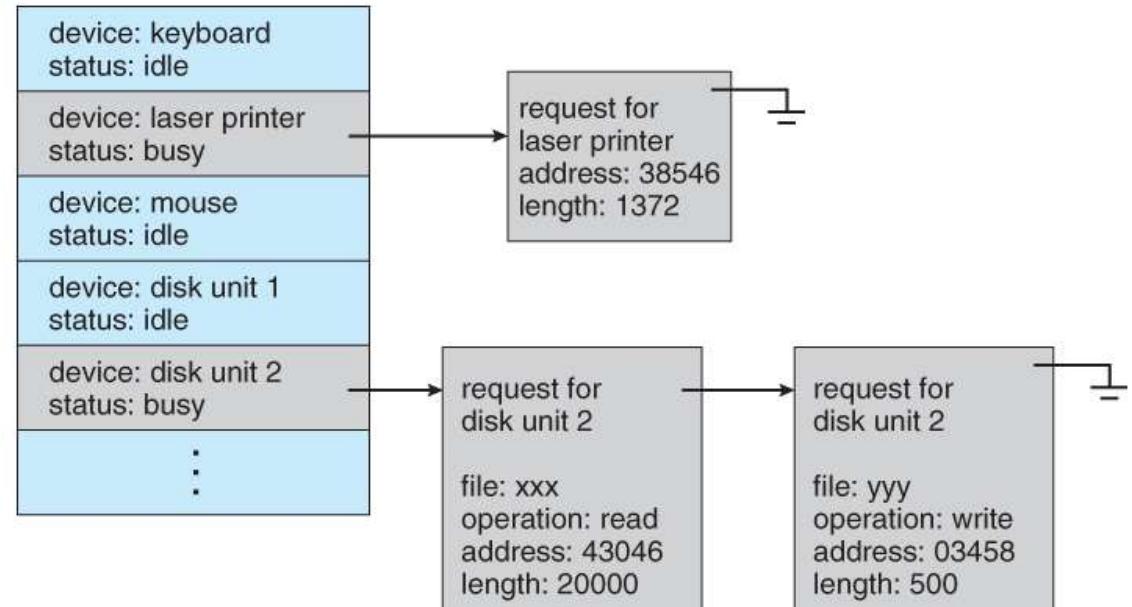
- Vettore delle interruzioni dei processori Intel, da 0 a 31 non mascherabili, da 32 a 255 mascherabili

| numero di vettore | descrizione |
|-------------------|---|
| 0 | errore di divisione |
| 1 | eccezione di debug |
| 2 | interruzione null |
| 3 | breakpoint |
| 4 | eccezione di overflow |
| 5 | eccezione di range exceeded |
| 6 | codice operativo non valido |
| 7 | dispositivo non disponibile |
| 8 | doppio errore |
| 9 | overrun del segmento coprocessore (riservato) |
| 10 | task state segment (tss) non valido |
| 11 | segmento non presente |
| 12 | errore di stack |
| 13 | protezione generale |
| 14 | errore di pagina |
| 15 | (riservato Intel, non utilizzare) |
| 16 | errore in virgola mobile |
| 17 | controllo dell'allineamento |
| 18 | controllo della macchina |
| 19–31 | (riservato Intel, non utilizzare) |
| 32–255 | interruzioni mascherabili |

Figura 1.5 Tabella degli eventi di un processore Intel.

Ciclo di I/O

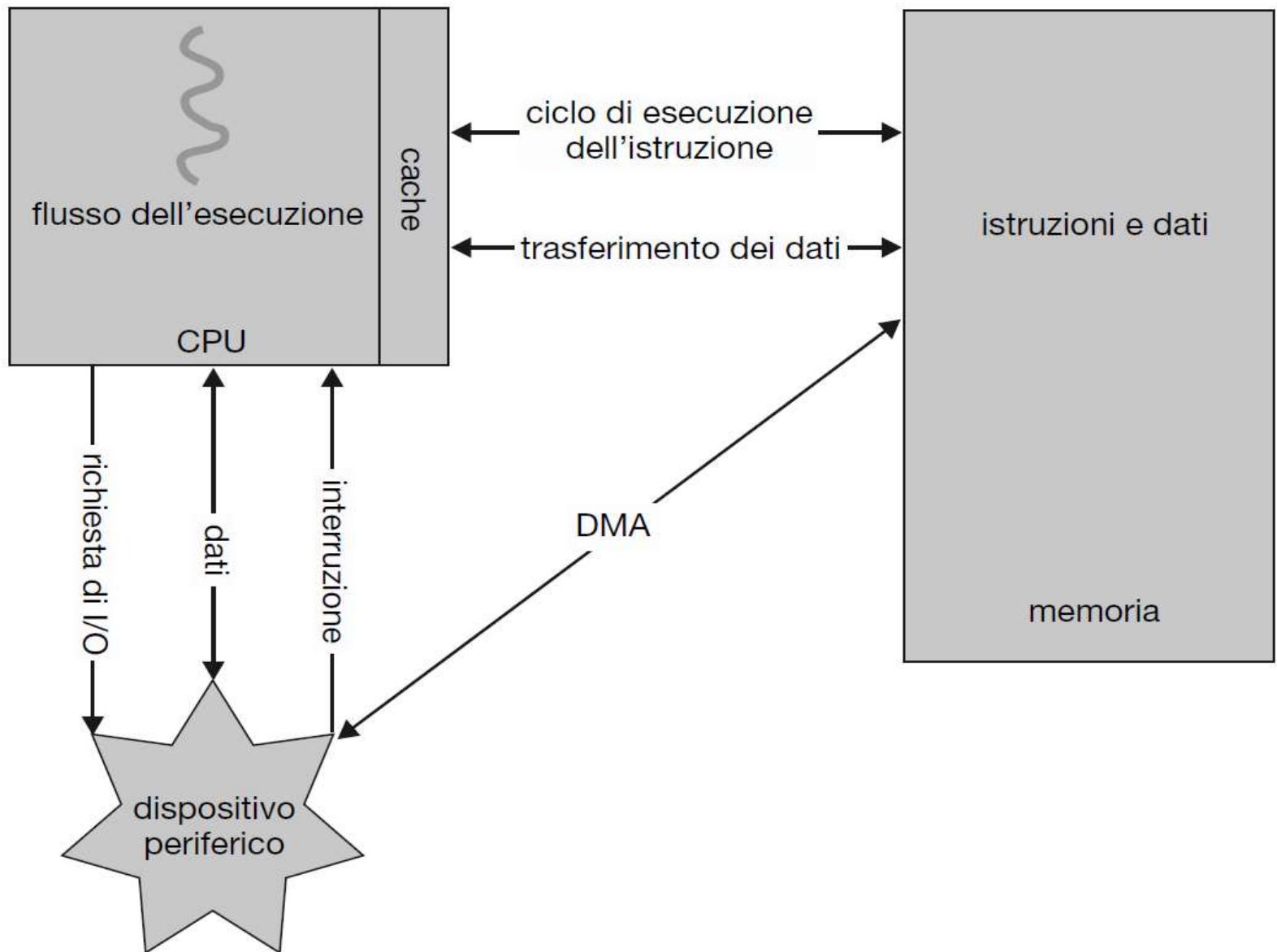
- Due modalità di gestione dopo l'avvio di operazioni I/O
 - Il controllo torna al prog. utente **dopo il completamento I/O**
 - Istruzione wait mette in idle la CPU fino al prossimo interrupt
 - Ciclo di wait attivo (compete per l'accesso in memoria)
 - Una richiesta pendente alla volta (vantaggio: si conosce il dispositivo che ha richiesto)
 - Il controllo torna al prog. utente **senza aspettare il completamento**
 - Gestione richieste contemporanee
 - **Tavola dello stato dei dispositivi** entry per ogni I/O che indica tipo, indirizzo e stato
 - S.O. usa la tavola dei dispositivi I/O per determinare il dispositivo che ha richiesto l'interruzione, verifica lo stato del dispositivo e modificare la entry per indicare che ha servito l'interruzione



Accesso Diretto in Memoria

- Usato per dispositivi I/O ad alta velocità
 - capaci di trasmettere informazione a velocità vicina a quella della memoria
- **Il controller del dispositivo trasferisce interi blocchi di dati dal buffer alla memoria centrale senza intervento della CPU**
- Il trasferimento richiede un solo un interrupt per block invece che uno per ogni byte
- DMA (Direct Memory Access)

Struttura I/O



Tipica architettura con DMA

Strutture di Memoria

- Memoria Principale che la CPU può accedere direttamente
 - **Random access memory (RAM)**
 - Tipicamente **volatile**
- Memoria di sola lettura
 - Es. ROM, PROM, EPROM, EEPROM (Elettronicamente canc. e progr.)
- Memoria Secondaria
 - Estensione della memoria principale che fornisce capacità di memorizzazione **non-volatile**
 - Hard-disk drive (hdd)
 - Disco logicamente diviso in **tracce** suddivise in **settori**
 - Disco a stato solido
 - più veloce, accesso casuale uniforme, tempi di accesso brevi

Gerarchia di Memorie

- Memorie organizzate in gerarchie
 - Velocità
 - Costo
 - Volatilità
- **Caching** – copia temporaneamente dati in sistemi più veloci; la memoria principale può essere vista come cache per la secondaria
- **Device Driver** per ogni controllore di dispositivo che gestisce I/O
 - Fornisce un'interfaccia uniforme tra controller e kernel

Gerarchia di Memorie

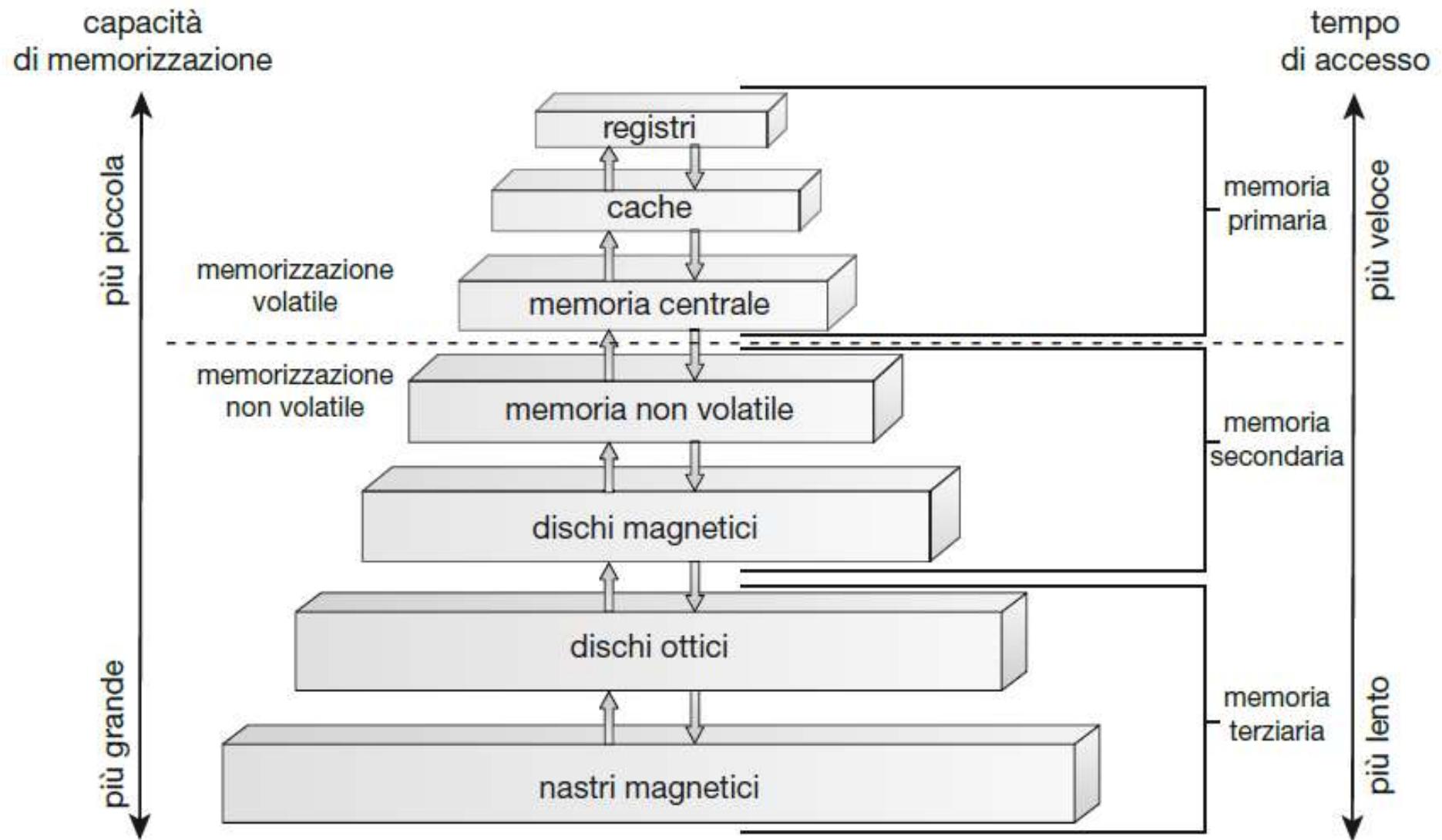


Figura 1.6 Scala gerarchica dei sistemi di memorizzazione.

Caching

- Principio valido a diversi livelli (hardware, SO, software)
- Informazione temporaneamente copiata da memoria più lenta a più veloce
- Memoria più veloce (cache) considerate per prima cercando un dato
 - Se il dato c'è, viene immediatamente utilizzato (veloce)
 - Se non c'è, viene copiato in cache ed usato
- Cache più piccola della memoria cached
 - Gestione della cache
 - Dimensione della cache e politica di replicazione

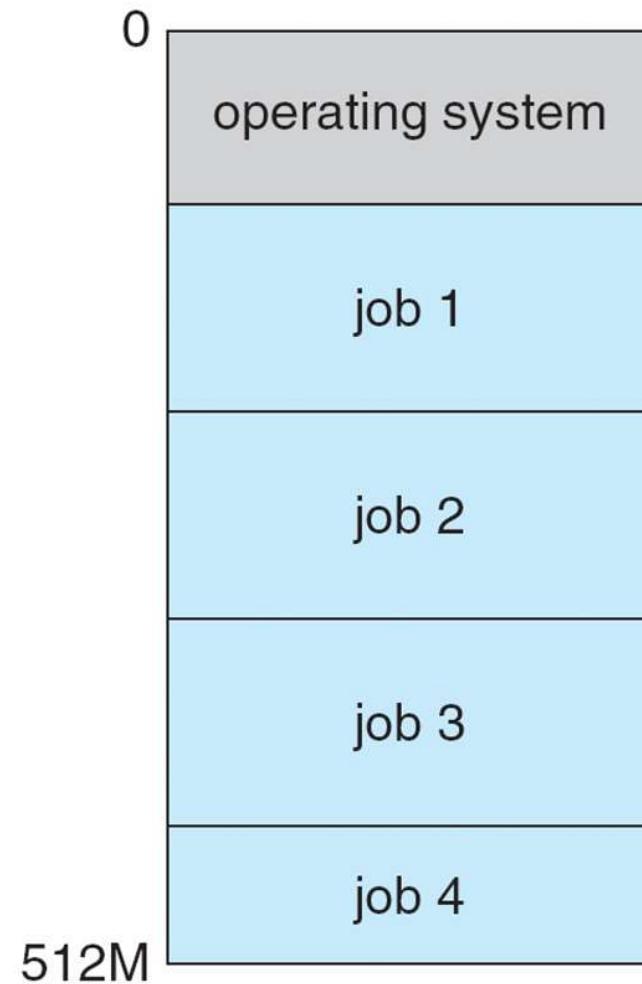
Caratteristiche Sistema Operativo

- **Multiprogrammazione** per efficienza
 - Un singolo utente non riesce ad utilizzare al meglio CPU e dispositivi I/O
 - Multiprogrammazione organizza i jobs (codice e dati) per mantenere sempre attiva la CPU
 - Un sottinsieme dei job totali del sistema è mantenuto in memoria
 - Un job viene selezionato ed eseguito dal **job scheduler**
 - Nel caso di attesa (operazioni I/O per esempio), OS cambia il job
- **Timesharing (multitasking)** la CPU che cambia job molto frequentemente permettendo agli utenti di interagire (**interactive computing**)
 - **Tempo di risposta** breve
 - Ogni utente programmi in esecuzione ⇒ **processi**
 - Se più jobs sono pronti per essere eseguiti ⇒ **CPU scheduling**
 - Se i processi non entrano in memoria, lo **swapping** li carica o scarica dalla memoria per eseguirli
 - **La memoria virtuale memory** permette di eseguire processi non completamente in memoria

Attività del Sistema Operativo

- 1. Programma di avviamento (*bootstrap program*).**
- 2. Il kernel** inizia a offrire servizi al sistema e agli utenti.
- 3. Interruzioni ed eccezioni (*traps o exceptions*).**
- 4. Chiamata di sistema o system call.**
- 5. Multiprogrammazione → *multitasking*.**

Memoria per Sistema a Multiprogrammazione



Modalità Operativa Duale

- Sistemi multiprogrammati e multiutente richiedono **meccanismi di protezione**
- Le operazioni in dual-mode permettono all'OS di proteggersi e proteggere altri componenti
 - **User mode** e **kernel mode** (modo utente e supervisore)
 - Un **bit di modalità** (hardware)
 - Permette di distinguere quando il sistema esegue codice utente o kernel
 - Alcune istruzioni sono **privilegiate** quindi solo eseguibili in modalità kernel
 - Con interrupt/trap si va alla modalità kernel, quando viene servita l'interruzione il sistema ripristina la modalità user
 - MS-DOS per Intel 8088 non aveva bit di modo

Protezione CPU

- Programmi utenti non devono tenere il controllo indefinito
- Timer per dare a SO controllo periodico della CPU
 - Timer invia segnale d'interruzione alla CPU dopo una latenza fissata
 - ▶ Contatore decrementato dal clock fisico
 - SO setta il contatore (modalità privilegiata) a zero genera un interrupt
 - Permette ad SO di riprendere il controllo o terminare processi

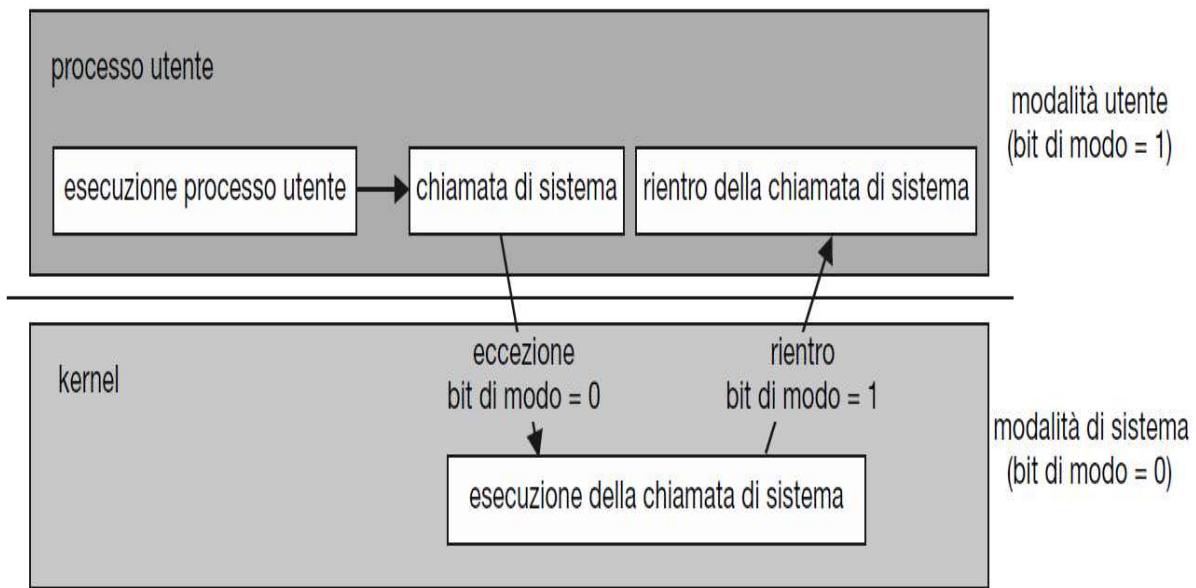


Figura 1.13 Transizione da modalità utente a modalità di sistema.

Protezione Memoria

- Occorre proteggere la modalità di esecuzione privilegiata
- Sovrascrivendo il vettore delle interruzioni e le operazioni di servizio si potrebbe trasferire la modalità privilegiata all'utente
- Occorrono meccanismi di protezione hardware della memoria

- Per stabilire gli indirizzi di memoria a cui un programma può accedere si possono utilizzare due registri:
 - **Registro base:** più piccolo indirizzo accessibile
 - **Registro limite:** dimensione del range di indirizzi

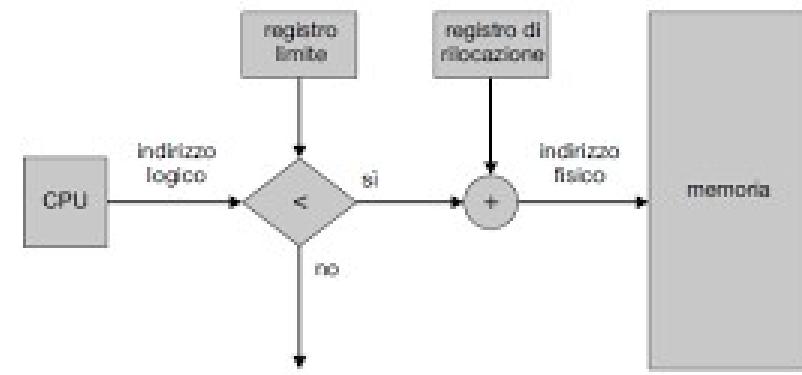


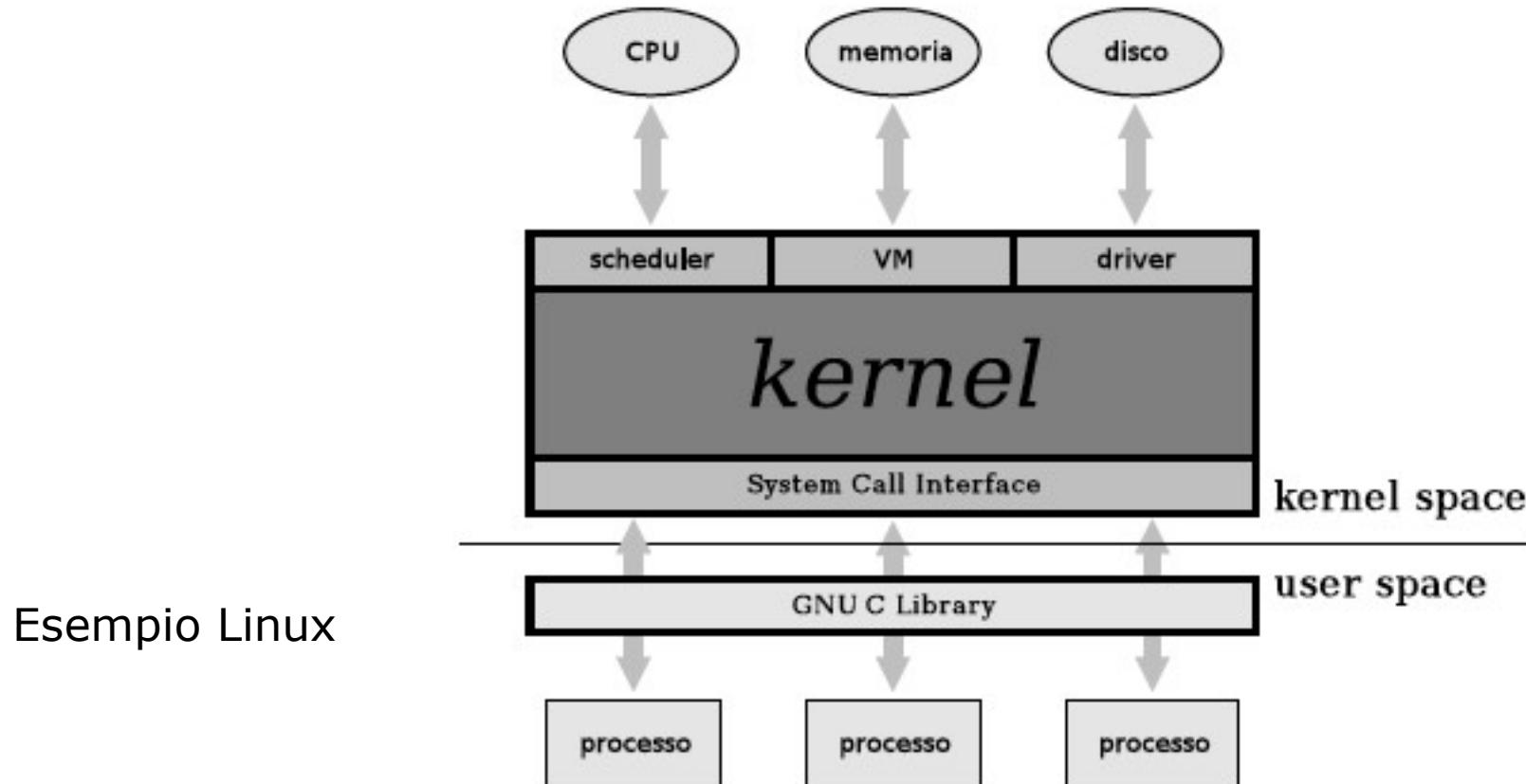
Figura 9.6 Registri di rilocazione e limite.

Operazioni di un Sistema Operativo

- **Interrupt driven** (hardware e software)
 - Hardware interruzione da un dispositivo
 - Software interrupt (**exception** o **trap**):
 - Errori Software (e.g., division by zero)
 - Richiesta per un servizio del sistema operativo
 - Altri problemi di processo includono infinite loop, processi che si modificano reprocamente, etc.

Chiamate di Sistema

- Le interfacce con cui i programmi possono accedere all'hardware vanno sotto il nome di chiamate al sistema (system call):
 - insieme di funzioni che un programma può chiamare (viene generata un'interruzione del processo passando il controllo dal programma al kernel)



Gestione Risorse

Gestione dei processi

Gestione della memoria

Gestione dei file

Gestione della memoria di massa

Gestione della cache

Gestione dell'I/O

Gestione Processi

- Un processo è un programma in esecuzione. Il programma è una **entità passiva**, il processo è una **entità attiva**.
- Un processo ha bisogno di risorse per svolgere un task
 - CPU, memoria, I/O, files
 - Dati di inizializzazione
- Alla terminazione di un processo occorre il rilascio e recupero delle risorse
- Processi **single-threaded** hanno un **program counter** (locazione prossima istruzione da eseguire)
 - Istruzioni sequenziali, una alla volta, fino a completamento
- Processi **multi-threaded** hanno un program counter per thread
- Tipicamente i sistemi hanno in esecuzione molti processi, per più utenti, in esecuzione concorrente su una o più CPU

Gestione Processi

Il Sistema Operativo fornisce diversi meccanismi per la gestione dei processi, in particolare per:

- Creazione e cancellazione di utenti e processi di sistema
- Sospensione e resume di processi
- Sincronizzazione di processi
- Comunicazione tra processi
- Gestione del deadlock

Gestione Memoria

- Per eseguire un programma istruzioni e dati devono essere caricati in memoria
- Attività di gestione della memoria:
 - Tenere traccia di quali parti della memoria sono attualmente utilizzate e da chi
 - Decidere quali processi (o parti di essi) e dati spostare dentro e fuori alla/dalla memoria
 - Allocazione e deallocazione dello spazio di memoria secondo necessità

Gestione Archiviazione

- Il sistema operativo fornisce una rappresentazione logica e uniforme dell'archiviazione delle informazioni
 - Astrazione di proprietà fisiche
 - Unità logica di archiviazione è il **file**
- Gestione del File-System
 - Files tipicamente organizzati gerarchicamente in directory
 - Controllo dell'accesso per determinare chi può accedere a cosa
 - Il SO gestisce
 - ▶ Creazione e cancellazione file e directory
 - ▶ Primitive per modifica di file e directorie
 - ▶ Mapping di file nella memoria secondaria
 - ▶ Backup di file su memoria non-volatile

Varie Forme di Archiviazione dei Dati

| Livello | 1 | 2 | 3 | 4 | 5 |
|---------------------------|--|--------------------------------|-------------------|----------------------|-------------------|
| Nome | registri | cache | memoria centrale | disco a stato solido | disco magnetico |
| Dimensione tipica | < 1 KB | < 16 MB | < 64 GB | < 1 TB | < 10 TB |
| Tecnologia | memoria dedicata con porte multiple (CMOS) | CMOS SRAM (on-chip o off-chip) | CMOS DRAM | memoria flash | disco magnetico |
| Tempo d'accesso (ns) | 0,25 – 0,5 | 0,5 – 25 | 80 – 250 | 25.000-50.000 | 5.000,000 |
| Aampiezza di banda (MB/s) | 20.000 – 100.000 | 5000 – 10.000 | 1000 – 5000 | 500 | 20 – 150 |
| Gestito da | compilatore | hardware | sistema operativo | sistema operativo | sistema operativo |
| Supportato da | cache | memoria centrale | disco | disco | disco o nastro |

Figura 1.14 Caratteristiche di varie forme di archiviazione dei dati.

Gestione Cache

- Occorre utilizzare il dato più recente, indipendentemente da dove è archiviato nella gerarchia di archiviazione

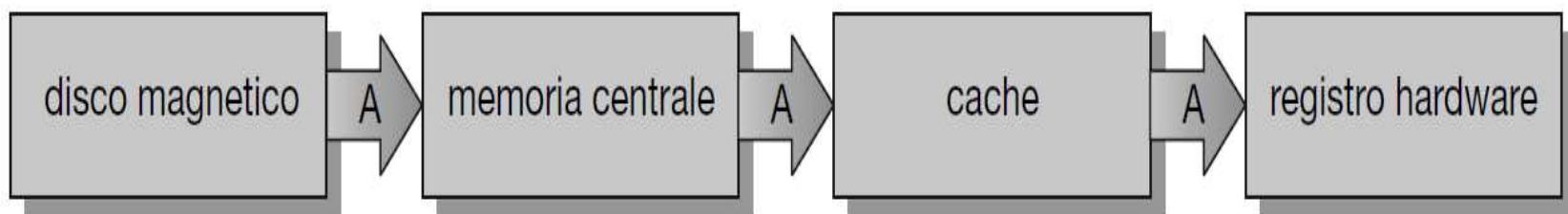


Figura 1.15 Migrazione di un intero A da un disco a un registro.

- In ambiente multiprocessore occorre la coerenza della cache perché tutte le CPU abbiano il dato più recente nella loro cache
- In ambiente distribuito ancora più complessa (più copie su più macchine)

Sistemi I/O

- L'OS deve nascondere all'utente le specificità hardware dei dispositivi fornendo un'interfaccia uniforme
- Sottosistemi I/O responsabili di
 - Gestione della memoria di I/O incluso buffering (memorizzazione temporanea dei dati durante il trasferimento), memorizzazione nella cache, spooling (sovraposizione dell'output di un job con l'input di altri job)
 - Interfaccia generica per i dispositivi
 - Driver per dispositivi hardware specifici

Protezione e Sicurezza

- **Protezione** – qualunque meccanismo di controllo di accesso alle risorse per processi e utenti
- **Sicurezza** – difesa del sistema da attacchi interni ed esterni
 - Inclusi denial-of-service, worms, viruses, identity theft, theft of service
- I sistemi generalmente prima distinguono tra gli utenti, per determinare chi può fare cosa:
 - Identità utenti (**user IDs**, security IDs)
 - User ID associato a file e processi per il controllo di accesso
 - Identificativi di Gruppo (**group ID**) definiscono insiemi di utenti con permessi di accesso
 - **Privilege escalation** consentono agli utenti di ottenere maggiori permessi di accesso

Strutture dati del Kernel

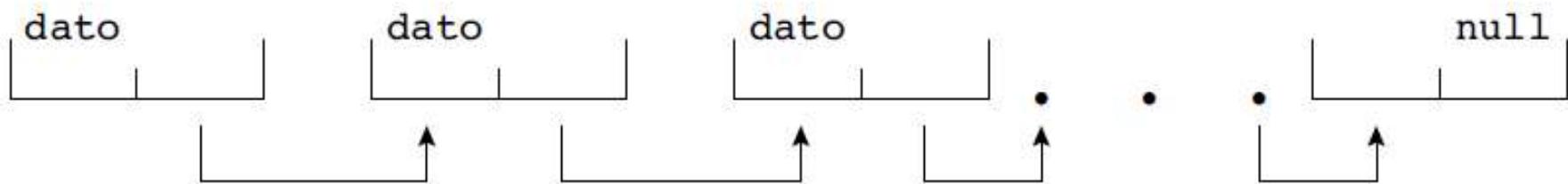


Figura 1.17 Lista semplicemente concatenata.

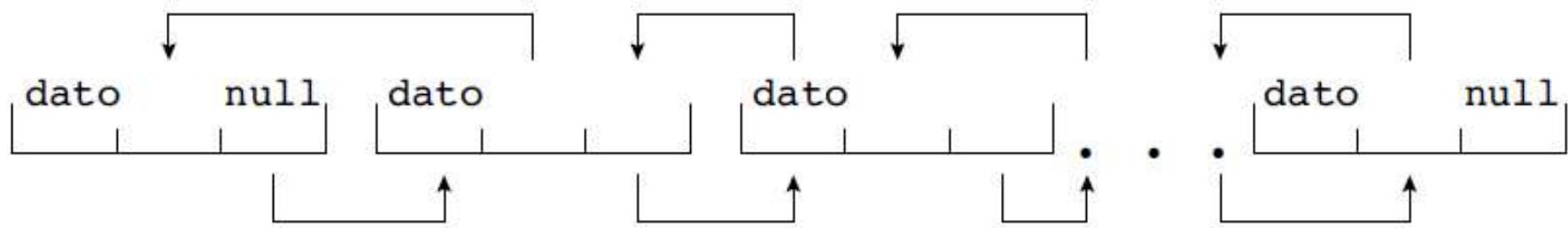


Figura 1.18 Lista doppiamente concatenata.

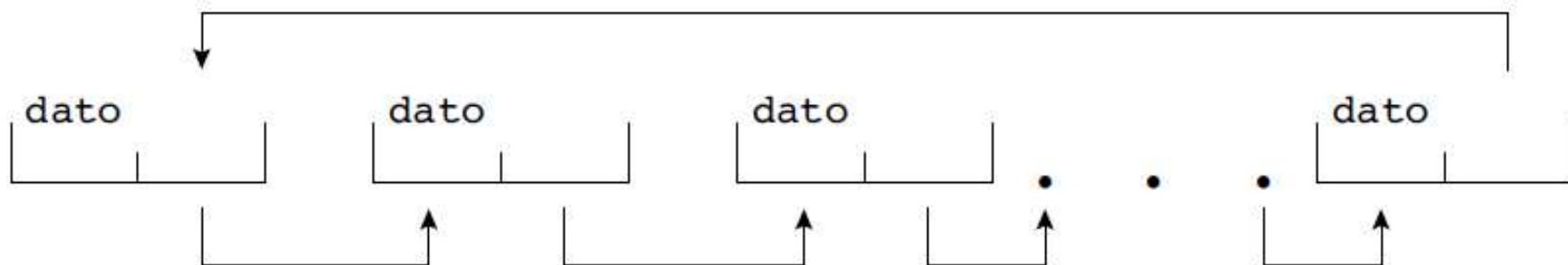


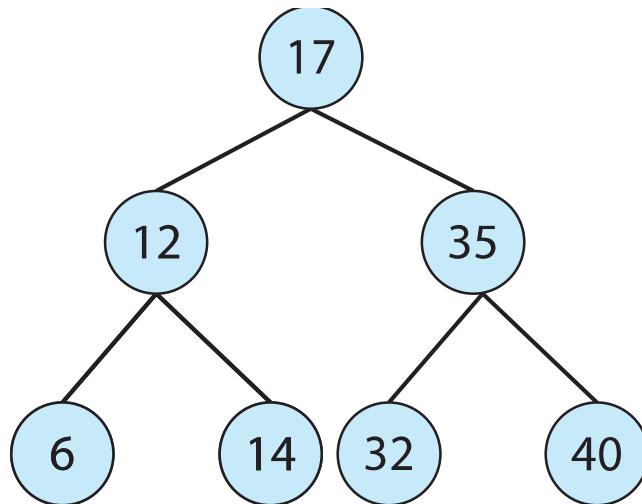
Figura 1.19 Lista circolare.

Strutture dati del Kernel

- **Alberti binari di ricerca**

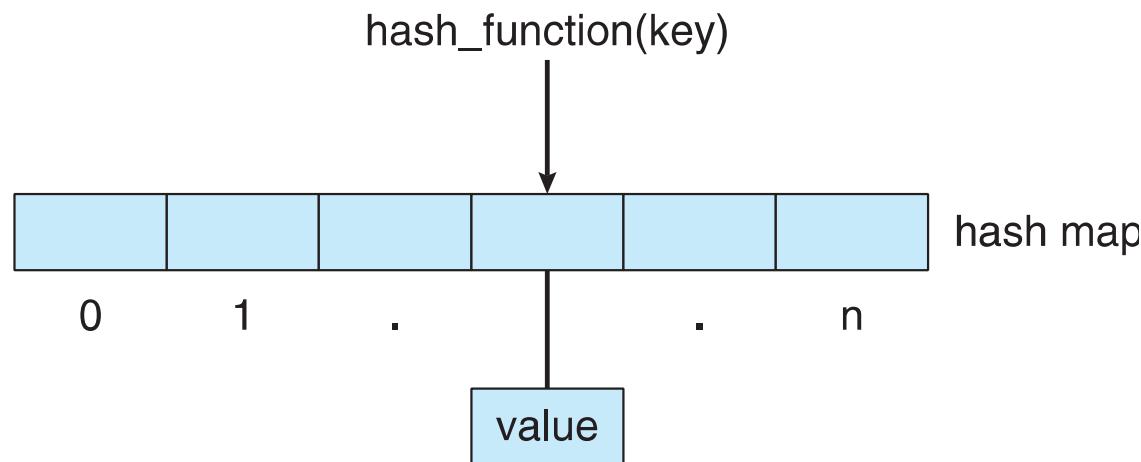
left \leq right

- Complessità $O(n)$
- **Albero di ricerca Bilanciato** $O(\lg n)$



Strutture dati del Kernel

□ Funzione e mappe Hash



- **Funzione hash** – mappa chiave in valori (indici di tabella)
- Funzione utilizzata per creare una **tabella hash** (o mappa hash) che associa coppie [chiave:valore]

Sistemi Operativi Open-Source

- Sistemi Operativi disponibili open-source disponibili in formato sorgente invece che codice binario compilato
 - Software libero anche dotato di licenza che consente uso, ridistribuzione e la modifica senza costi
-
- Open-source iniziato con **Free Software Foundation (FSF)** con la **GNU Public License (GPL)**
 - Esempi sono **GNU/Linux**, **BSD UNIX** (incluso il core di **Mac OS X**) **Solaris** ed altri
 - Microsoft Windows è proprietario
-
- Si possono usare VMM come VMware Player (Free on Windows) o Virtualbox (open source and free on many platforms - <http://www.virtualbox.com>)

Memoria di Massa



Memoria di Massa

- Overview della struttura di un Sistema di Memoria di Massa
- Struttura del Disco
- Schedulazione del Disco
- Gestione del disco
- Swap-Space Gestione
- Struttura RAID
- Implementazione Stable-Storage

Obiettivi

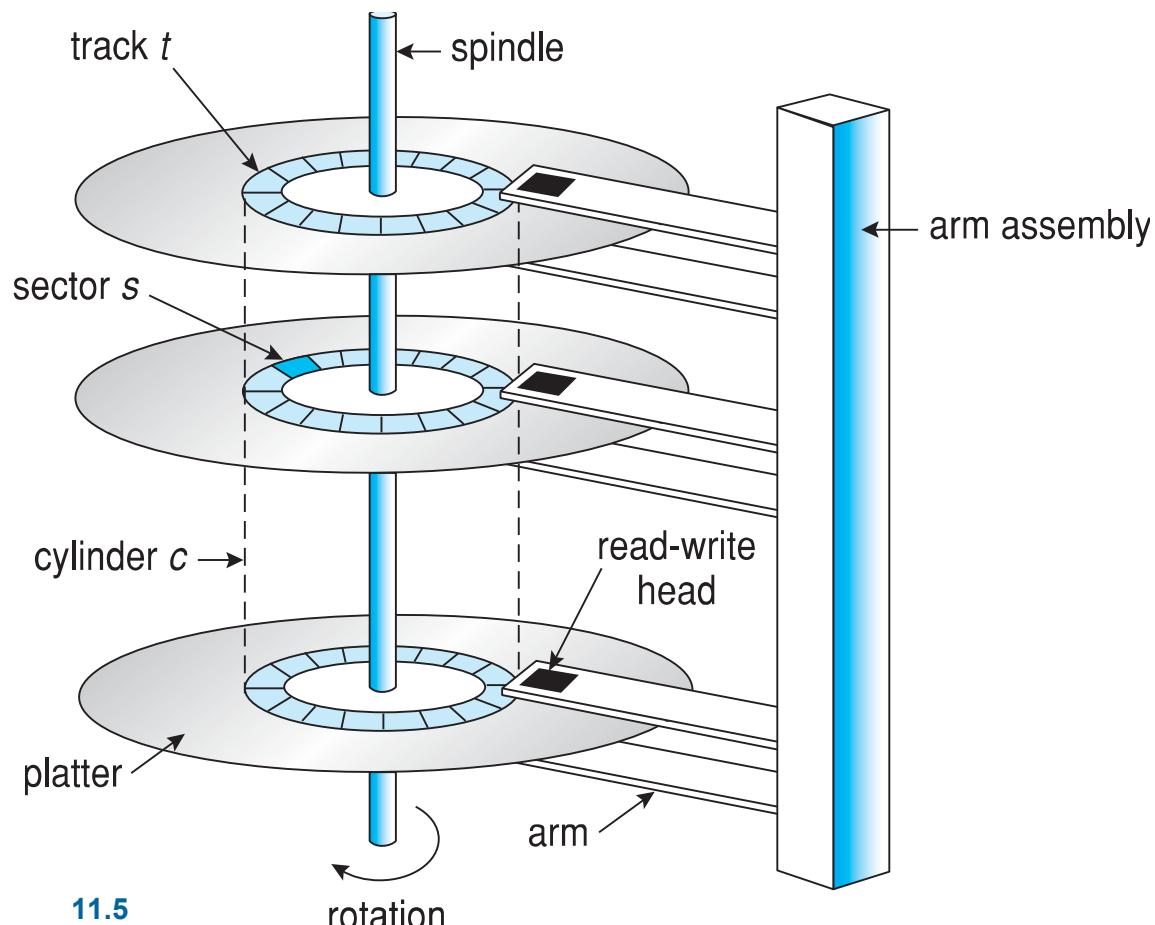
- Descrivere la struttura fisica di un dispositivo di memoria secondario e i suoi effetti sull'uso del dispositivo
- Spiegare le performance dei dispositivi di memoria di massa
- Valutare gli algoritmi di disk scheduling
- Discutere i servizi del SO per la memorizzazione di massa, inclusi i RAID

Overview della Struttura di Memoria di Massa

- Consideriamo due tipi di memorie secondarie
 - **Hard Disk Drives (HDDs) e nonvolatile memory (MVM)**
 - Coprono la maggior parte dei dispositivi per la memoria secondaria dei moderni computer
 - Si descrivono le caratteristiche di questi dispositivi

Hard Disk

- Strutturati su dischi
- Dischi piatti come CD hanno dimensioni da .85" a 14" (storicamente)
 - Comunemente 3.5, 2.5 e 1.8 pollici
 - Superfici coperte da materiale magnetico
- Una testina read-write si move sui piatti
- Braccio muove le testine
- Piatti divisi in trace
- Tracce divise in settori
- Testine si muovono su cilindri (insieme di tracce)
- Migliaia di cilindri, per ogni traccia centinaia di settori
- Prima settori 512 bytes ora fino a 4 KB



Hard Disk

- Capacità da GB a TB
- Motore disco
 - da 60 a 250 rotazioni per secondo,
 - rotation per minute (RPM) da 5400 a 15000
- Performance
 - Transfer Rate
 - da disco a computer
 - teorico – 6 Gb/sec
 - Effective Transfer Rate – real – 1Gb/sec
 - Tempo di posizionamento
 - Seek time e rotational latency
 - Seek time da 3ms a 12ms – 9ms comune per desktop drives
 - Latenza dipende da spindle speed
 - $1 / (\text{RPM} / 60) = 60 / \text{RPM}$
 - Average latency = $\frac{1}{2}$ latency

| Spindle [rpm] | Average latency [ms] |
|---------------|----------------------|
| 4200 | 7.14 |
| 5400 | 5.56 |
| 7200 | 4.17 |
| 10000 | 3 |
| 15000 | 2 |



Hard Disk Performance

- **Access Latency = Average access time = average seek time + average latency**
 - Per dischi più veloci $3\text{ms} + 2\text{ms} = 5\text{ms}$
 - Per dischi lenti $9\text{ms} + 5.56\text{ms} = 14.56\text{ms}$
- Average I/O time = average access time + (quantità da trasferire / velocità di trasferimento) + overhead del controllore
- Per esempio per trasferire un blocco di 4KB su un disco da 7200 RPM con un 5ms average seek time, 1Gb/sec di transfer rate con un .1ms overhead del controllore =
 - $5\text{ms} + 4.17\text{ms} + 0.1\text{ms} + \text{transfer time} =$
 - Transfer time = $4\text{KB} / 1\text{Gb/s} = 0.031\text{ ms}$
 - Average I/O time for 4KB block = $9.27\text{ms} + .031\text{ms} = 9.301\text{ms}$

Il Primo Disk Drive Commerciale



1956

IBM RAMDAC computer
included the IBM Model
350 disk storage system

5M (7 bit) characters

50 x 24" platters

Access time = < 1 second

Dispositivi NVM

I dispositivi **NVM** sempre più rilevanti e diffusi. Sono elettrici e non meccanici. Tipicamente composti basati su memoria flash (chip NAND flash) vengono spesso inseriti in contenitori simili a unità disco, e per questa ragione sono chiamati **dischi a stato solido**, o **SSD**

Un dispositivo NVM può anche assumere la forma di un'**unità USB**

In tutte le sue forme possiamo trattarlo in modo uniforme



Figura 11.3 Scheda di un SSD da 3,5 pollici.

Solid-State Disks (SSDs)

- Memoria nonvolatile usata come un hard drive
 - Non meccanica
 - Implementata con diverse tecnologie
- Può essere più affidabile di HDDs
- Più costosa per MB
- Forse life span più breve
- In generale ha meno capacità di HDD ma più veloce
- Capacità cresce rapidamente ed il prezzo cala, quindi si stanno affermando
- I bus di connessione possono essere troppo lenti
 - connessi direttamente al bus di sistema
- Non parti mobili, non parti meccaniche
 - **non seek time e rotational latency**

Algoritmi del controllore delle NAND Flash

Implementate con **semiconduttori NAND** che lanciano nuove sfide su capacità di memorizzazione e affidabilità

I **semiconduttori NAND** *non possono essere sovrascritti direttamente*, devono prima essere cancellati. Organizzati in pagine e presentano *pagine che contengono dati non validi*.

La cancellazione avviene in blocco e prende tempo ($>$ del tempo di scrittura $>$ tempo lettura). Le operazioni possono avvenire in parallelo. Deterioramento dopo ogni cancellazione. Durata misurata in Drive Writes per Day: quanto volte può essere scritta per giorno prima del fallimento (es. 1 TB può avere 5 DWPD nel periodo garantito)

Algoritmi gestiti dal controller, non dal sistema Operativo

| | | | |
|-------------------|---------------|-------------------|-------------------|
| pagina valida | pagina valida | pagina non valida | pagina non valida |
| pagina non valida | pagina valida | pagina non valida | pagina valida |

Figura 11.4 Blocco NAND con pagine valide e non valide.

Controllore delle NAND Flash

I semiconduttori NAND non possono essere sovrascritti direttamente, devono prima essere cancellati. Sono di solito presenti *pagine* che contengono dati *non validi*.

Il controller contiene una tabella: Flash Transaltion Layer (FTL) indica quali pagine contengono dati validi

Per gestire i dati occorrono: algoritmi di garbage collection per liberare blocchi, pagine sempre disponibili per fare il write (**overprovisioning**), meccanismi per distribuire le cancellazioni, meccanismi di correzione del dato (se errori continui su una pagina, questa è marcata come bad)

| | | | |
|-------------------|---------------|-------------------|-------------------|
| pagina valida | pagina valida | pagina non valida | pagina non valida |
| pagina non valida | pagina valida | pagina non valida | pagina valida |

Figura 11.4 Blocco NAND con pagine valide e non valide.

Memoria Volatile

- Memoria volatile (DRAM) può essere usata per fare storage
- Si parla di RAM drivers o **RAM disk**

- Creati dai device driver che presentano porzioni della DRAM come se fosse memoria di massa (temporaneo)
- Accessibile ad utenti e programmatori
- In Linux /dev/ram e /tmp
- Memorizzazione temporanea, ma accesso molto rapido

Dischi Magnetici

- Era il primo mezzo di memorizzazione secondaria
 - Evoluto da bobina a cartucce
- Relativamente permanente, contiene larghe quantità di dati
- Tempo di accesso molto lento
- Random access ~1000 volte più lento del disco, 100000 più lento di SSD
- Usate soprattutto per backup, stoccaggio di dati non usati frequentemente, trasferimento tra sistemi
- Mantenuto in bobina e avvolto o riavvolto per la testina di lettura-scrittura
- Una volta che i dati sotto testina, velocità di trasferimento simile al disco
 - 140MB/sec e maggiore
- TB di storage

Metodi di Connessione Memoria Secondaria

- Drive connesso al computer via **I/O bus** o il **bus di sistema**
 - Bus variano, es. **EIDE**, **ATA**, **SATA**, **USB**, **Fibre Channel**, **SCSI**, **SAS**, **Firewire**
 - Per HDDs più comune è il SATA
 - Per NVM interfacce veloci NVM express che collega direttamente al Peripheral Component Interconnect (PCI) bus
- Trasferimento dati sul bus gestiti da **controller**
- **Host controller** gestisce il trasferimento dati lato computer che usa il bus per parlare con il **disk controller**

Mapping degli Indirizzi

- Le unità disco sono indirizzati come grandi array 1-dimensionali di **blocchi logici** dove il logical block è la più piccola unità di trasferimento
 - Formattazione di basso livello creano **blocchi logici** su mezzi fisici
 - Ogni blocco logico mappato su un settore fisico o su pagina di un dispositivo NVM
- Su Disco blocchi logici mappati sequenzialmente in settori
 - Settore 0 è il primo settore della prima traccia sul cilindro più esterno
 - Il mapping procede in ordine, da quella traccia al resto delle tracce su quel cilindro e poi attraverso il resto dei cilindri (dal più esterno al più interno)
- Per NVM mapping da una tupla (chip, blocco, pagina) ad un blocco logico
 - Indirizzamento basato su Logical Block Address (LBA) più semplice di settore, cilindro, testina o chip, blocco, pagine

Mapping degli Indirizzi

- Le unità disco sono indirizzati come grandi array 1-dimensionali di **blocchi logici** dove il logical block è la più piccola unità di trasferimento
 - Formattazione di basso livello creano **blocchi logici** su mezzi fisici
 - Ogni blocco logico mappato su un settore fisico o su pagina di un dispositivo NVM
- Su Disco blocchi logici mappati sequenzialmente in settori
- Per NVM mapping da una tupla (chip, blocco, pagina) ad un blocco logico
- Il passaggio da indirizzi logici a fisici dovrebbe essere facile ma ...
 - Settori possono essere danneggiati danneggiati
 - Non-constante numero di settori per traccia
 - Gestione interna del mapping tra LBA e settori fisici
 - Si tende a mantenere comunque allineati indirizzi logici e fisici (salgono insieme)

Struttura Disco

- L'array 1-dimensionale di blocchi logici è mappato in settori del disco sequenzialmente
 - Il numero di settori per traccia varia, più nelle tracce esterne che interne
 - 40% in più di settori nelle tracce esterne
 - Nei CD velocità aumenta verso l'interno per mantenere la lettura dei settori costante
 - Constant Linear Velocity (CLV)
 - In altri casi la densità dei bit varia per mantenere costante il flusso dei dati
 - Constant Angular Velocity (CAV)
 - Settori per traccia diverse centinaia, decine di migliaia di cilindri

Scheduling del Disco

- Il SO è responsabile dell'uso efficiente dell'hardware
 - Per le unità disco significa veloce accesso e alto trasferimento dati
- Per le unità a disco
 - minimizzare il seek time e il latency time
 - Seek time \approx seek distance
 - Latency time tempo per arrivare al settore con rotazione
- La disk **bandwidth** è il numero totale di bytes transferiti diviso per il tempo totale tra la prima richiesta di servizio ed il completamento dell'ultimo transferimento
- Si può aumentare sia l'accesso che il trasferimento gestendo l'ordine con cui vengono servite le diverse richieste di I/O

Scheduling del Disco

- Ci sono tipi di richieste di I/O da disco
 - SO
 - Processi di sistema
 - Processi utente
- Le richieste I/O riguardano input o output, file da aprire, l'indirizzo di memoria e su disco, numeri di settori da trasferire
- SO mantiene una coda di richieste, per disco o dispositivo
- Un disco disponibile in idle può subito servire la richiesta di I/O, il disco occupato gestisce una coda di richieste
 - Gli algoritmi di ottimizzazione hanno un ruolo nel caso di code
 - Nel passato molto sforzo per definire interfacce ai dischi per ottimizzare
 - Oggi molto è gestito direttamente dai controllori del disco
 - Non è accessibile la locazione esatta della testina di lettura dei dischi
 - Però si può assumere prossimità tra indirizzi fisici e logici

Scheduling del Disco

- Nota che i controllori di unità hanno piccoli buffer e possono gestire code di richieste di I/O
 - Molti algoritmi proposti per schedulare il modo in cui sono servite le richieste di I/O (basati su riduzione del seek time)
 - L'analisi è valida per uno o più piatti
-
- Illustriamo gli algoritmi di scheduling con una coda di richieste a blocchi su cilindri (0-199)

98, 183, 37, 122, 14, 124, 65, 67

Testina che punta a 53

FCFS

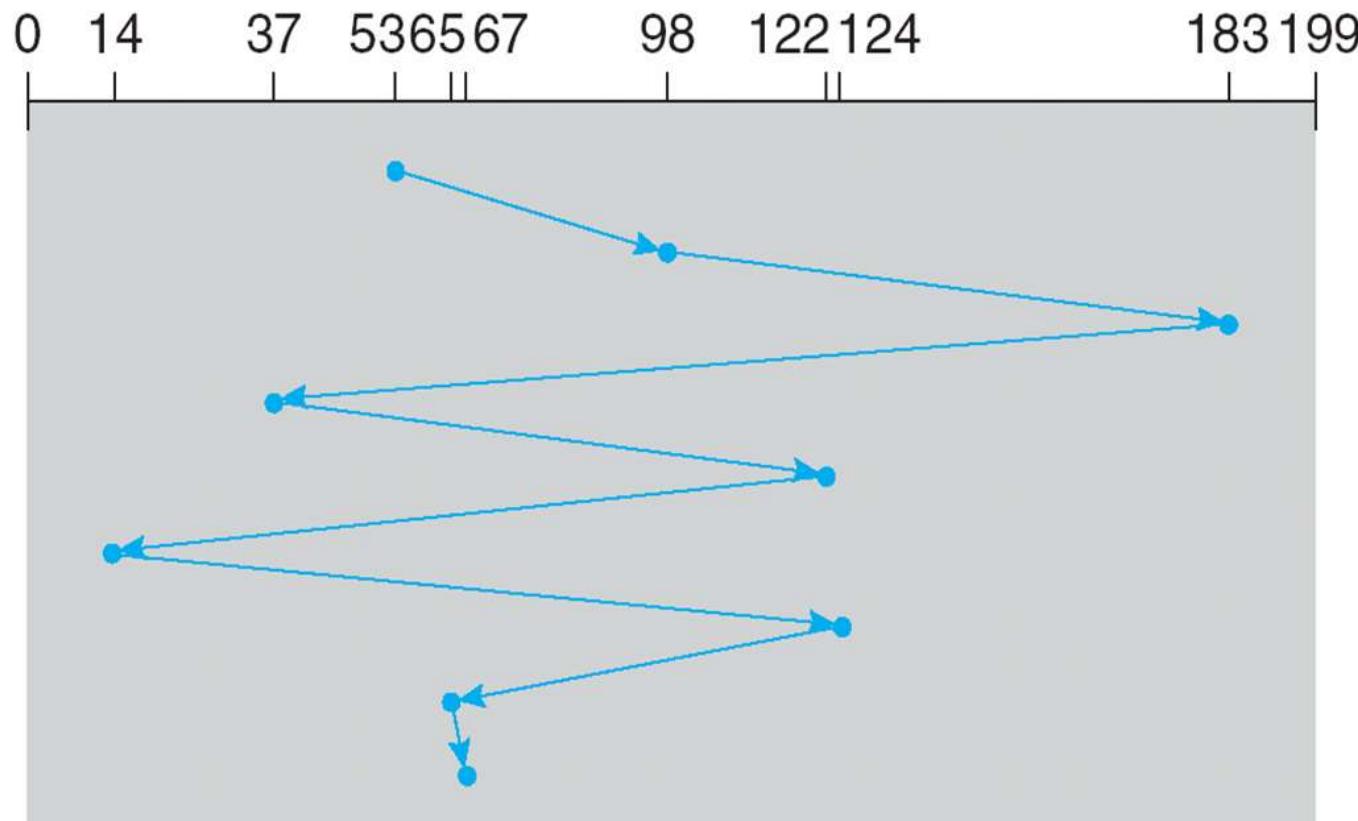
Algoritmo più semplice

La figura mostra un movimento di testina attraverso 640 cilindri

Si potrebbe migliorare con 37 e 14 serviti vicini, come 122 e 124

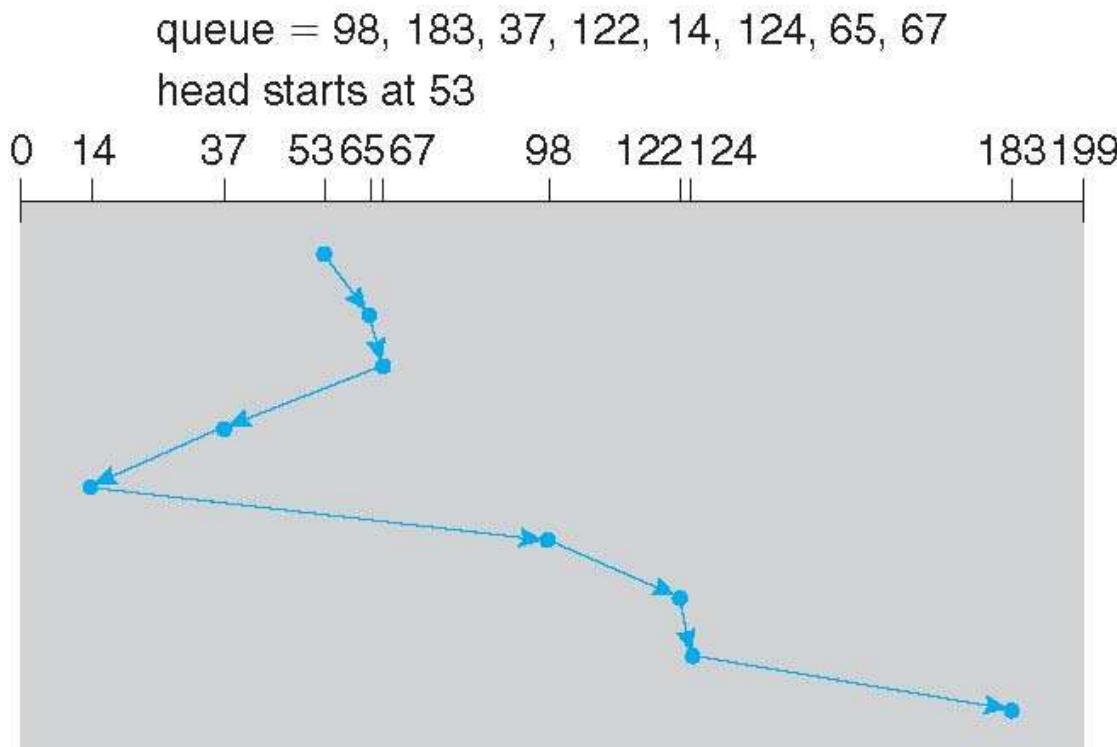
queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



SSTF

- Shortest Seek Time First, seleziona la richiesta con il minimo seek time dalla corrente posizione della testina
- SSTF scheduling è una forma di SJF scheduling; può causare starvation di qualcuna delle richieste
- La figura mostra un numero totale di movimenti di testa di 236 cilindri



SCAN

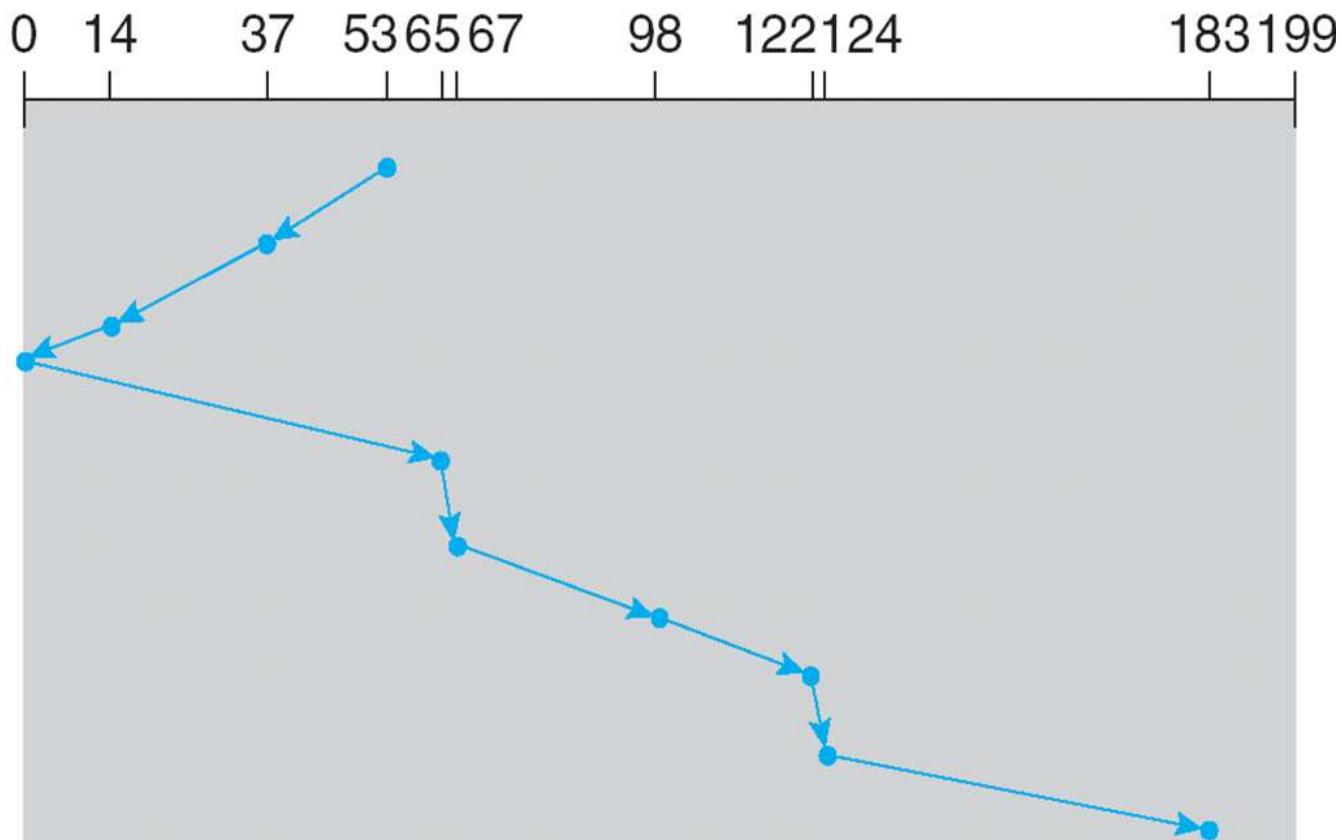
- La testina inizia ad un estremo del disco e move verso l'altro estremo servendo le richieste finché non arriva all'altro estremo dove inverte il percorso
- **SCAN algorithm** chiamato anche **elevator algorithm**
- Nota che se le richieste non sono uniformemente dense con densità accumulate all'altro estremo del disco, queste possono aspettare molto

SCAN

- La figura mostra un movimento totale di 236 cilindri
- Nota che se le richieste non sono uniformemente dense con densità accumulate all'altro estremo del disco, queste aspettano molto

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



C-SCAN

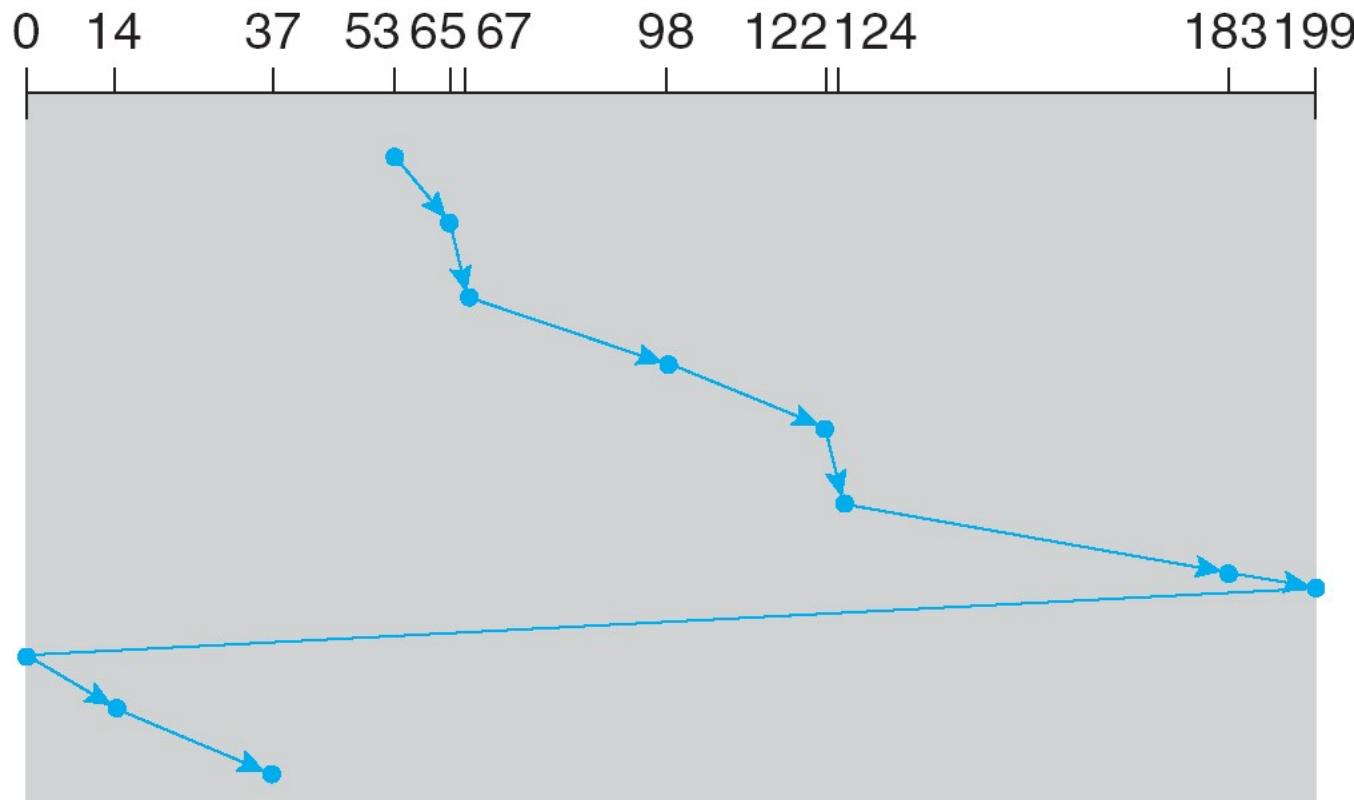
- Tempi di attesa più uniformi rispetto a SCAN
- La testina si muove da un estremo del disco all'altro servendo le richieste nel mentre
 - Quando raggiunge l'altro estremo, comunque, immediatamente torna all'inizio del disco, senza servire le richieste nel viaggio di ritorno
- Mette i cilindri in una lista circolare dove l'ultimo cilindro porta al primo

C-SCAN

- Si assume il movimento sempre verso destra

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

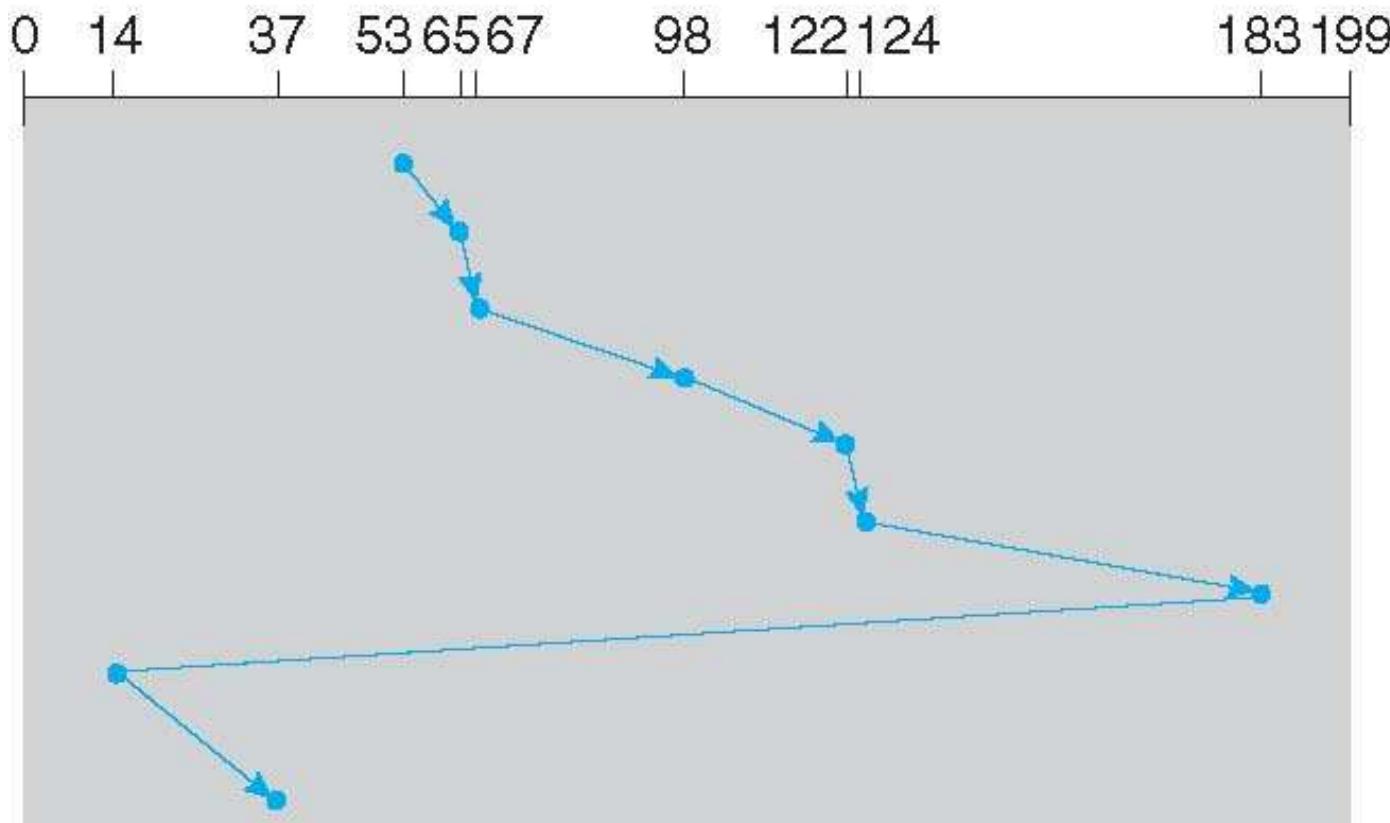


C-LOOK

- LOOK è un tipo di SCAN, C-LOOK è una versione di C-SCAN
- Il braccio avanza fino all'ultima richiesta in ogni direzione, poi cambia direzione

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

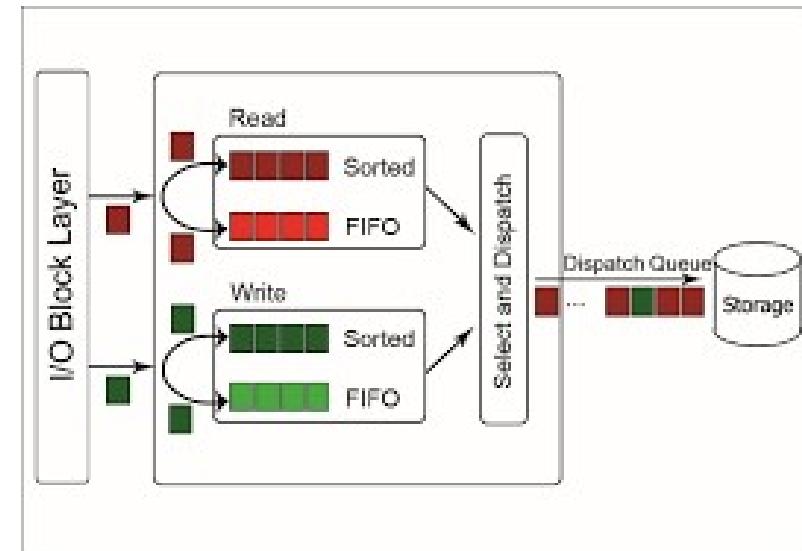


Selezionare un Algoritmo di Disk-Scheduling

- SSTF (Shortest Seek Time First) è tipico e naturale
- SCAN e C-SCAN funzionano meglio per sistemi con grosso carico su disco
 - Meno starvation
- Performance dipende dal numero e dai tipi di richiesta
- Le richieste di servizio possono essere influenzate dai metodi di file-allocation
 - File allocato in modo contiguo meno movimenti di testina
 - File likato sparso più movimenti di testina
 - Allocazione delle directory (file aperti richiedono ricercar nella struttura)
- Algoritmi di disk-scheduling implementati come moduli separati del SO per consentire un aggiornamento se necessario
- SSTF che LOOK sono una ragionevole algoritmo di default

Selezionare un Algoritmo di Disk-Scheduling

- In alcuni sistemi più code con priorità per evitare starvation
- Linux – deadline scheduler con code di read e write con priorità e ordine LBA (Logical Block Addressing) implementate in modo simile a C-SCAN
 - read maggiore priorità perché può bloccare
- Quattro code per gestire le deadline
 - 2 per read e 2 per write, una gestita con LBA l'altra con FCFS
 - Le richieste gestite in batch, dopo ogni batch si controlla se ci sono richieste in FCFS vecchie (più di 500 ms per read e 5 sec per write) e si sceglie quello come prossimo batch LBA
 - Quando scade il timer di deadline, Deadline comincia a servire le richieste dalla coda FIFO. Si serve per ordine di arrivo. Si contrasta la starvation delle richieste (con preferenza alle richieste in lettura)



Selezionare un Algoritmo di Disk-Scheduling

- Lo scheduler NOOP (NO OPeration) è il più semplice fornito da Linux.
 - echo noop > /sys/block/sda/queue/scheduler
 - Funzionamento. Una coda di richieste, gestita in modalità FIFO con merging delle richieste contigue
-
- **Vantaggi.** Scheduler semplicissimo e molto efficiente in termini di esecuzione. Non ha nessuna logica di riduzione dei seek (a parte il merging) ideale per dispositivi non rotazionali → (SSD).
 - **Svantaggi.** Non ha nessuna logica di riduzione dei seek (a parte il merging) disastroso sui dispositivi rotazionali. → Starvation delle richieste. Incapacità di differenziare in base alla importanza del processo.

NVM scheduling

- Lo scheduling HDD deve minimizzare i movimenti meccanici
- Nel caso di NVM non ci sono movimenti meccanici e si usa FCFS
 - Es. In Linux NOOP usa FCFS modificato per servire richieste adiacenti
- Le richieste read sono servite in modo uniforme, ma write in modo non uniforme
 - Alcuni SO servono le read con FCFS, e accorpando solo richieste di write
- I/O possono avvenire in modo random o sequenziale
 - Per HDD meglio sequenziale
 - Per NVM random è ok
 - input/output operations per second (IOPS) diversi per HDD e NVM
 - Nel caso di accesso random centinaia vs centinaia di migliaia
 - Nel caso di accesso sequenziale più simile
 - Le performance di NVM degradano nel tempo e la scrittura è più lenta della lettura

Gestione del Disco

- Il Sistema Operativo è responsabile di diverse operazioni
 - Inizializzazione, bootstrap, bad-block recovery
- Il dispositivo deve essere strutturato
 - HDD in tracce e settori, NMV con pagine e FTL (Flash Transalition Layer)
 - **Formattazione di basso livello** o **fisca**, disco in settori che il disk controller può leggere e scrivere (produttore fornisce e fa testing)
 - Ogni settore può mantenere un header, dati, più un trailer
 - Il trailer contiene un error correction code (**ECC**)
 - Di solito 512 bytes di dati ma si può selezionare
- Il SO deve anche avere una strutturazione del disco
 - **Partizione** del disco in uno o più gruppi di cilindri, ognuno trattato come un disco logico separato

Error Detection and Correction

- Occorrono tecniche per trovare e correggere errori
- **Parity Bit**
 - Ogni byte ha associato un bit che controlla se il numero di 1 è pari (parity = 0) oppure è dispari (parity = 1)
 - Se uno dei bit è danneggiato allora il parity non coincide più con quello precalcolato (incluso il danno dello stesso parity)
 - Tutti gli errori di un un bit su un byte sono trovati (due bit non visti)
 - Il parity si calcola rapidamente con lo XOR ($1 \text{ xor } 1 = 0$)
 - Parity è un esempio di metodo checksums che utilizza metodi aritmetici per controllare dati di lunghezza fissata
- **Error Correction Code (ECC)** fa anche la correzione
 - Disk drive usano un ECC per settore, NVM per pagina
 - Quando si scrive un settore/pagina ECC è scritto
 - Quando si legge ECC è controllato, se non coincide allora il settore/pagina è bad
 - Se soft error può essere corretto (pochi bit) altrimenti hard error

Gestione dei Bad Block

- Blocchi del disco possono essere corrotti
 - Nei vecchi sistemi la ricerca dei bad blocchi richiedeva una scansione lanciata dall'utente
 - Una gestione più sofisticata è fornita dai controllori del disco più recenti
-
- Il controllore del disco mantiene una lista dei bad block del disco e ha a disposizione dei settori di ricambio non visti dal SO
 - Quando prova ad accedere ad un settore e lo trova corrotto (ECC)
 - Riporta l'errore all'SO e marca il settore come bad e lo sostituisce con un settore di richiamo
 - Quando è richiesta il medesimo blocco logico, questo viene tradotto nel nuovo settore
-
- La sostituzione può interferire con lo scheduling del disco se le riparazioni sono "lontane", per questo i settori di ricambio si cercano nello stesso cilindro
 - Altro metodo è il sector slipping
 - Per la NVM la gestione è più semplice perché non c'è seek time da gestire

Gestione del Disco

- Il SO deve anche avere una strutturazione del disco

suddivide il dispositivo in uno o più gruppi di blocchi o pagine, dette **partizioni**



crea e gestisce il **volume**



formattazione logica, cioè crea un file system

Gestione del Disco

- Il SO deve anche avere una strutturazione del disco
 - **Partizione** del disco in uno o più gruppi di cilindri, ognuno trattato come un disco logico separato (info su partizioni è su locazione del disco)
 - In Linux fdisk per avere info sul dispositivo di storage
 - Il Sistema Operativo riconosce un dispositivo e legge la sua info su partizione
 - Il Sistema crea una entry per quella partizione (in Linux /dev)
 - Montare una partizione significa rendere il file system disponibile
- Il secondo step è la creazione e gestione di un volume (drive logico)
 - Creazione del volume può essere implicita o esplicita
 - Linux volume manager lvm2
- Il terzo step è la **formattazione logica** o la creazione di un file system
 - Per incrementare l'efficienza i file system raggruppano i blocchi in **clusters**
 - Disk I/O fatto in blocchi
 - File I/O fatto in clusters
 - Partizione indica anche se la partizione consente il bootstrap

Gestione del Disco

- Esempio del tool di disk management di Windows 7
 - Illustrati tre volumi C: E: F:, E: F: in partizioni del disco 1, c'è spazio non allocato per più partizioni

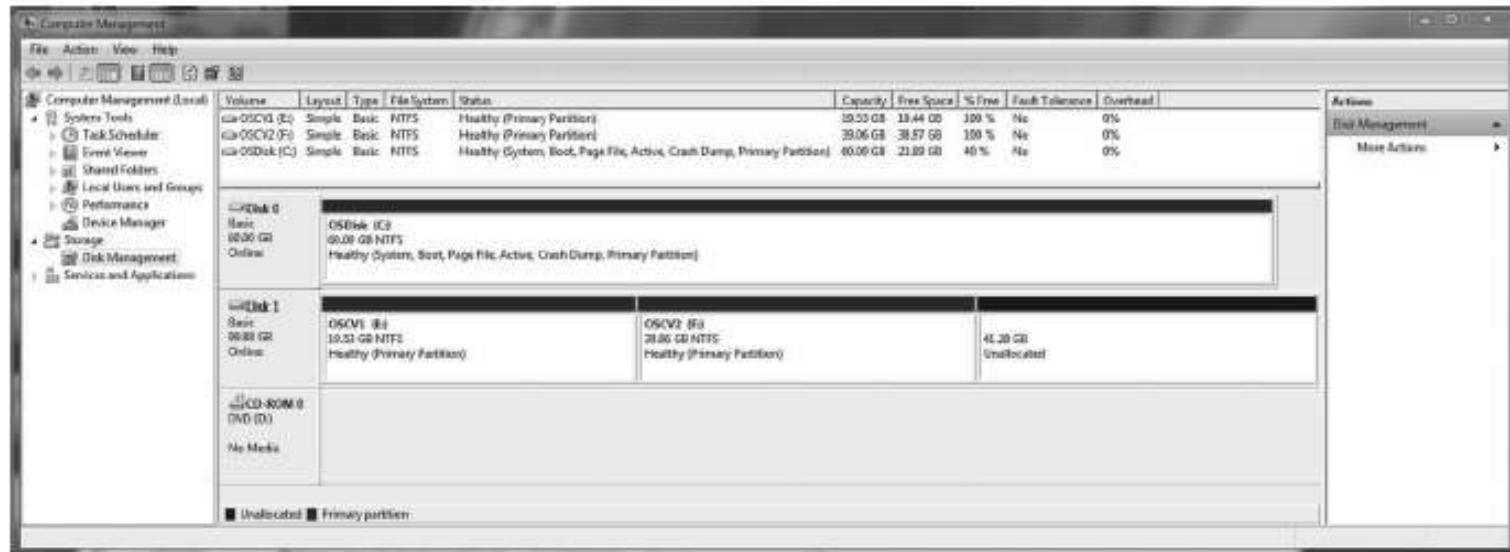


Figura 11.9 Lo strumento gestione dischi di Windows 7 mostra i dispositivi, le partizioni, i volumi e i file system.

Gestione del Disco

- Accesso al *raw disk* per applicazioni che richiedono una gestione dei blocchi dedicata, esclude il SO (database per esempio)
- Boot block inizializza il sistema
 - Il bootstrap è in ROM
 - **Bootstrap loader** è un programma nei boot block della partizione di boot
 - La partizione di boot fornisce la root del file system
 - Il file system di un computer consiste di tutti i volume montati
 - ▶ In windows C:, D:, E:, etc.

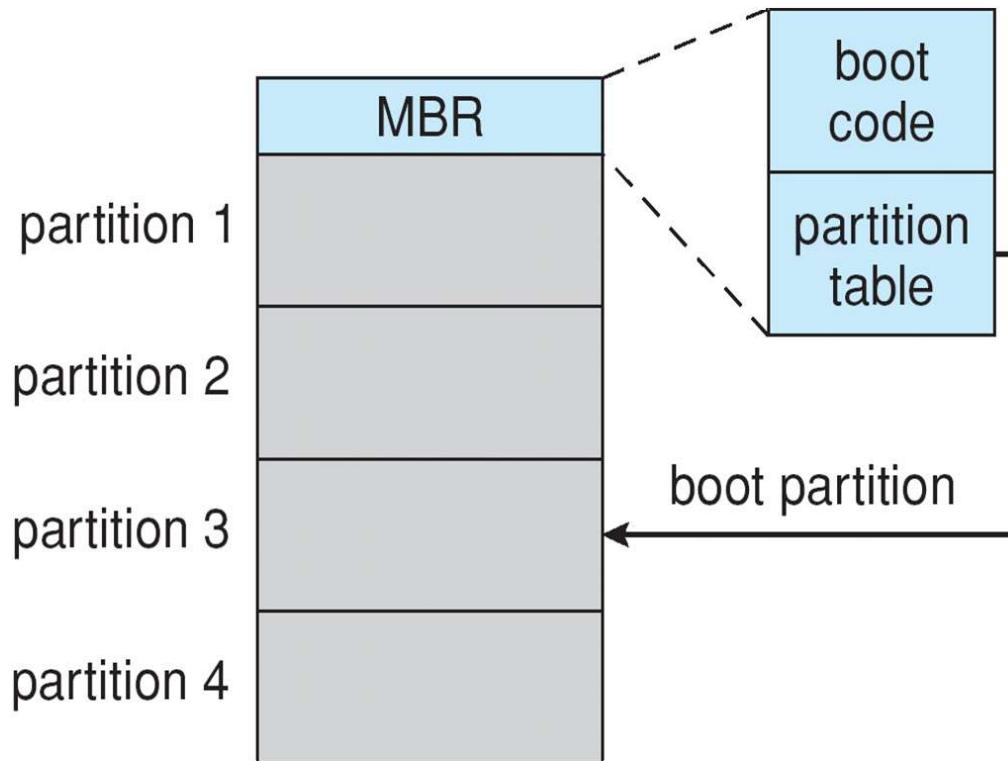
Gestione del Disco

- Boot block inizializza il sistema
 - I primi passi del bootstrap sono in firmware
 - **Bootstrap loader** è un programma che inizializza il SO e porta in memoria il bootstrap program dalla memoria secondaria
 - Il **bootstrap program** è memorizzato nel “boot block” in una locazione predefinita
 - Un dispositivo che ha il boot block è un boot disk o system disk
 - Il bootstrap program permette allo storage controller di caricare il boot blocks e di iniziare l'esecuzione del codice
 - Il **bootstrap program** più complesso del **bootstrap loader**
 - ▶ può caricare tutto il Sistema Operativo da una locazione non prestabilita
- Esempio Windows
 - ▶ Partizione di boot ha SO e driver dei dispositivi
 - ▶ Boot code nel primo blocco logico di HDD o prima pagina di NVM - Master Boot Record (MBR)
 - ▶ Boot inizia lanciando il programma nel firmwere
 - ▶ Il Bootstrap loader carica il codice di boot dal MBR (lancia il sistema)
 - ▶ MBR ha boot code e tabella delle partizioni, da queste si risale alle partizioni di boot

Booting da disco in Windows

Esempio Windows

- Boot code nel Master Boot Record (MBR)
- Il Bootstrap loader carica il codice di boot dal MBR (lancia il sistema)
- MBR ha boot code e tabella delle partizioni, da queste si risale alle partizioni di boot



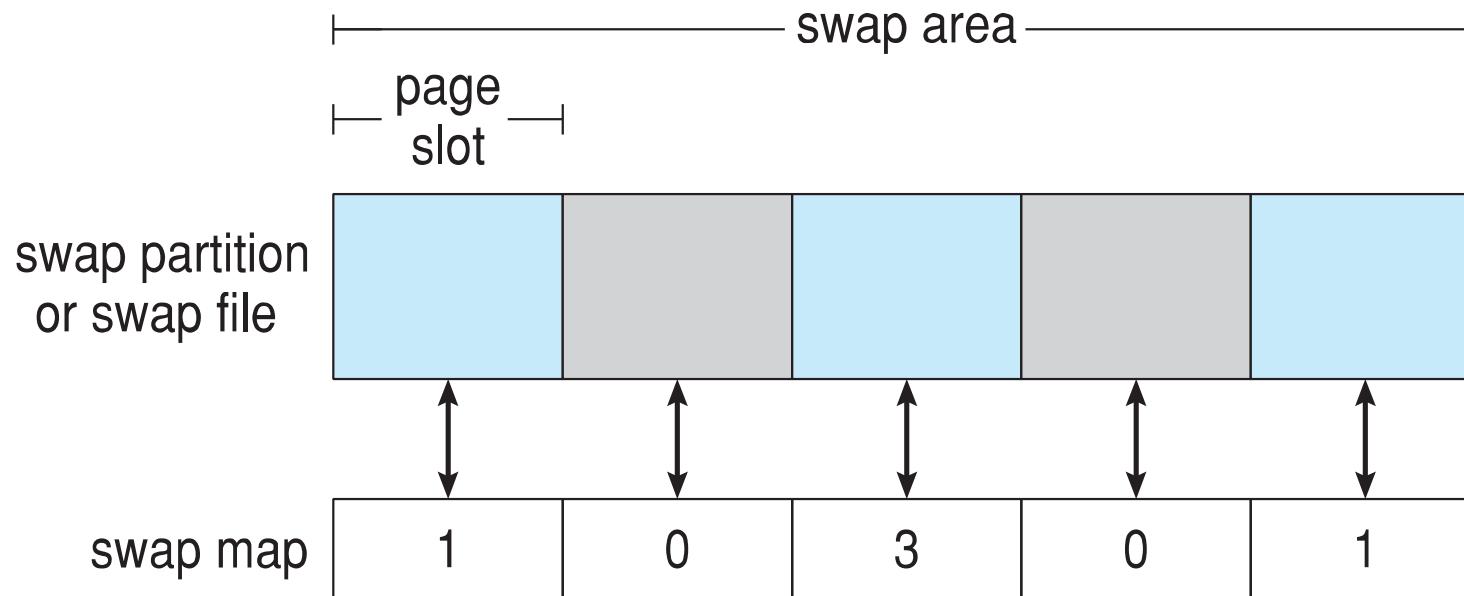
Gestione dello Swap-Space

- Swap-space — Virtual memory usa spazio su disco come estensione della memoria principale
- Lo swap space può variare da pochi megabytes a gigabytes a seconda della dimensione della memoria fisica, di quella virtuale e del modo in cui si usa la VM
 - Solitamente meglio sovrastimare che sottostimare perché se finisce i processi possono essere abortiti o arrivare al crash di Sistema
 - Es. Solaris suggerisce tanto quanto $SS = VM - FM$, Linux prima $SS = FM$ ora meno
 - In Linux multipli swap space su file system o su diverse partizioni
- Swap-space può essere ottenuto dal normale file system, ma più comunemente da una partizione del disco separata (raw)
 - Le partizioni raw permettono accesso più veloce ma è meno efficiente lo stoccaggio (frammentazione interna), cmq dati in swap a breve termine
 - Linux permette swap sia su file system che su partizione raw

Gestione dello Swap-Space

- Gestione dello Swap-space – Esempio nei Sistemi UNIX
 - Nei primi sistemi il Kernel faceva swapping dell'intero processo in locazioni contigue del disco, poi si è passata alla gestione con paging
 - In Solaris 1 si rileggono le pagine di text direttamente dal file system, memoria anonima su swap space (stack, heap, memoria non inizializzata)
 - Solaris 2 alloca swap space solo quando una dirty page viene mandata fuori dalla memoria fisica (non quando è creata la pagina della virtual memory)
 - In Linux, come in Solaris, swap space solo per memoria anonima
 - 4.3BSD alloca swap space quando un processo parte
 - Mantiene più aree swap (sia file system sia raw) – area composta di pagine di 4 KB
 - Il Kernel usa **mappe di swap** per tracciare l'uso dello swap-space – array di contatori interi che contano quanti processi si riferiscono a quella pagina (es. 0 pagina libera, 3 indica tre processi che condividono la stessa pagina)

Strutture Dati per lo Swapping in Linux



Connessione Dispositivi di Memorizzazione

- Il computer può accedere ai dispositivi di memorizzazione in tre modi:

un dispositivo
collegato alla
macchina

un dispositivo
connesso
alla rete

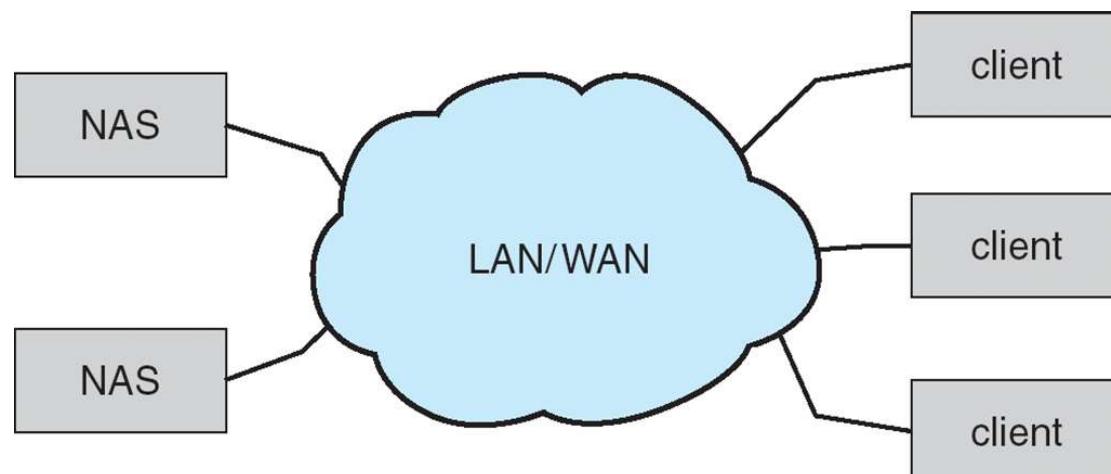
un dispositivo
cloud

Collegamento diretto alla Macchina

- Il computer accede al dispositivo direttamente (no rete)
 - Direct Access Storage (DAS) tramite porte I/O
 - Connesso diverse tecnologie
 - ▶ ATA, SATA (serial advanced technology attachment), eSATA, SCSI, SAS (serial attached SCASI), FC (Fiber Channel)
 - ▶ La più diffusa è SATA, architetture più sofisticate usano FC
 - Dispositivi connessi
 - ▶ HDDs, NVMs, CD, DVD, etc.

Network-Attached Storage

- Network-attached storage (**NAS**) storage a disposizione su una rete invece che tramite connessione locale (come un bus)
 - Accede da remoto al file systems
- NFS (Network file system) e CIFS (Common Internet File System) sono protocolli tipici
- Implementato con *remote procedure calls* (RPCs) tra host e dispositivo di storage su reti (tipicamente TCP o UDP on IP)
- Il protocollo **iSCSI** usa IP network per implementare il protocollo SCSI
 - Si collega remotamente ai dispositivi

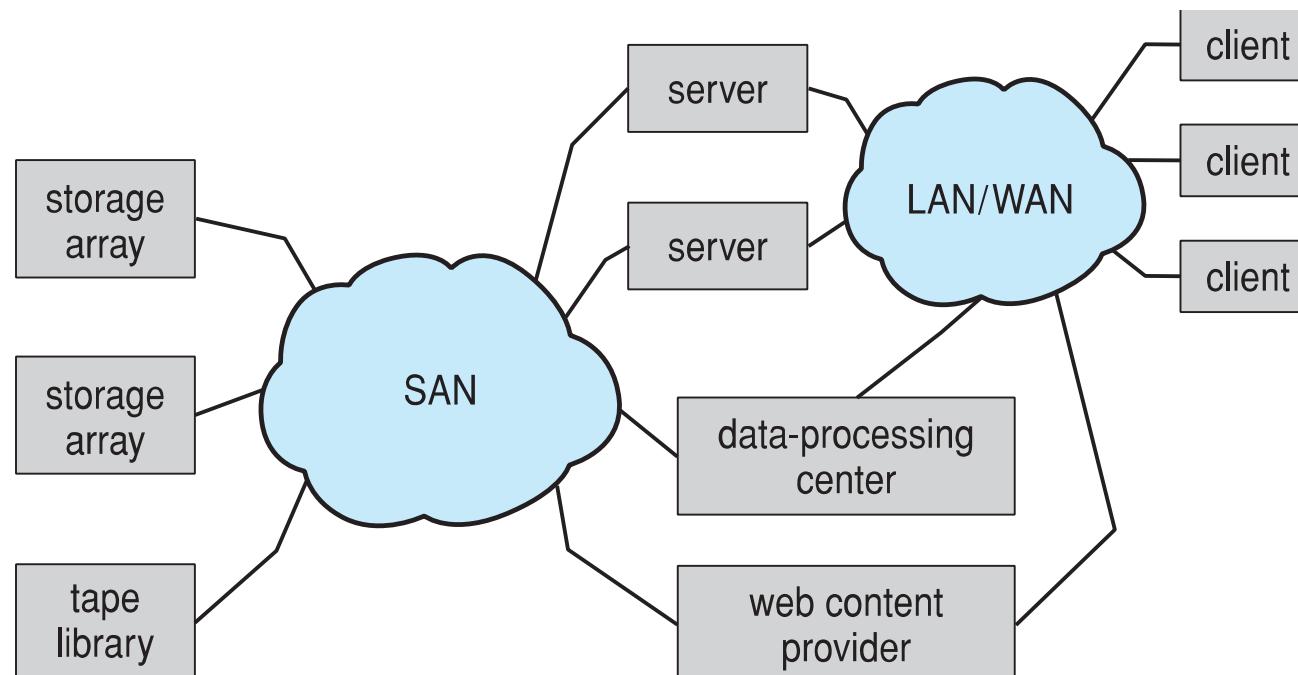


Cloud Storage

- Come Network-attached storage (NAS) permette l'accesso tramite rete
 - Accesso tramite Internet ad un centro remoto di storage
 - Non accede tramite file system
 - Accesso tramite API, più robusto rispetto a perdita di connettività
 - NAS accede assumendo LAN affidabile con protocolli come NFS e CIFS, se la LAN connection fallisce possono rimanere in hang

Storage Area Network

- Problema con NAS: comunicazione occupa banda e aumenta la latenza
- Comunicazione tra client e server contende banda a quella tra server e dispositivi di memoria
- Tipica in grandi ambienti di storage
- Rete privata utilizzando i protocolli di storage invece che di rete
- Host multipli attaccati a storage array multipli - flessibile



Storage Area Network

- SAN è una rete privata che collega server con dispositivi di memoria
 - Connessi più host e array di memorie
 - Host anche attaccati a switch
 - Storage disponibile via **LUN Masking** (logical unit num masking)da specifici array a specifici server
- Facile aggiungere o rimuovere storage, aggiungere un nuovo host e allocare il suo storage
- Più veloci di NAS, ma meno dispositivi collegati



Struttura RAID

- RAID – Redundant Array of Inexpensive Disks
 - Multiple unità disco forniscono affidabilità e performance via **ridondanza**
 - Prima Inexpensive ora Independent
- La probabilità di fallimento di N dischi è più bassa
- Consideriamo il **mean time between failure (MTBF)**
 - Se 100.000 ore in un array di 100 dischi la probabilità di rottura di un disco è 1000 ore, cioè 41,66 giorni ...
 - Senza replicazione dei dati è un problema
- Modo più semplice di replicare è il **mirroring** duplicazione dei dischi
 - Ogni disco logico corrisponde a più dischi fisici (es. scrittura doppia)
 - perdita di dato solo se anche il secondo disco fallisce prima della riparazione del primo

Struttura RAID

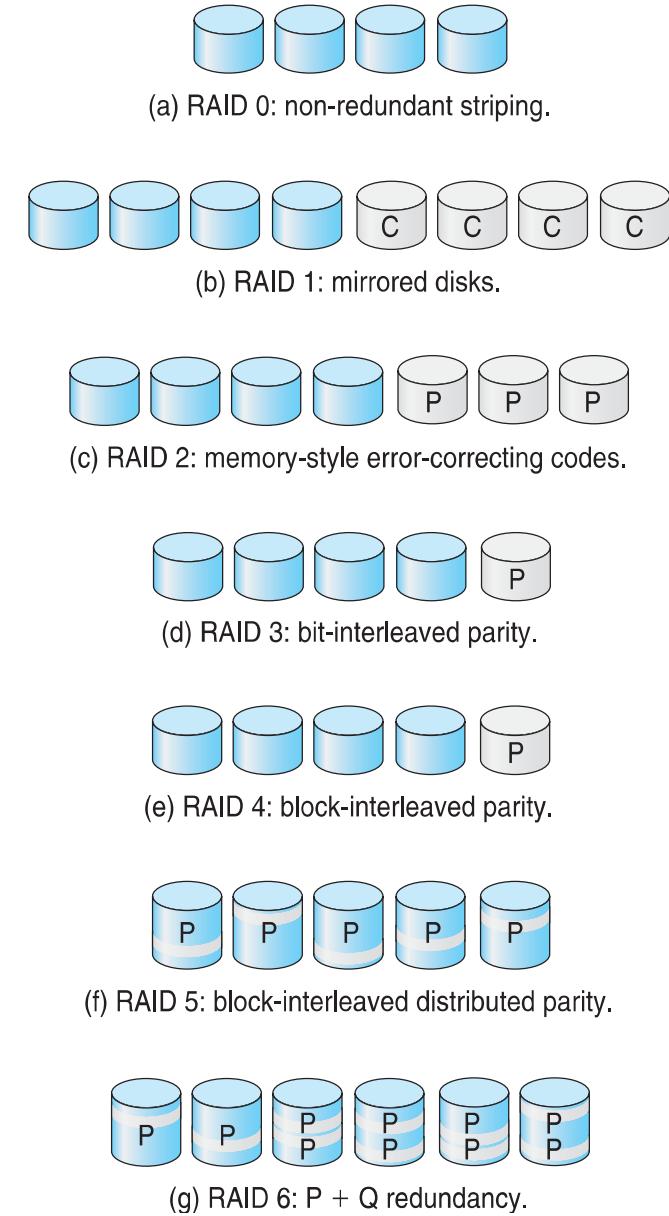
- Mirrored Disk
 - Oltre al **mean time between failures**
 - **Mean time to repair** – tempo di esposizione quando un altro fallimento potrebbe causare perdita di dati
 - **Mean time to data loss** tempo medio di perdita dei dati dovuti a fallimento
 - Se i mirrored disks fallissero independentemente, preso un disco con 100000 ore di mean time between failures e 10 ore di mean time to repair
 - Il mean time to data loss è $100000^2 / (2 * 10) = 500 * 10^6$ ore cioè 57000 anni!
- Però i fallimenti non sono indipendenti
 - power failure è particolarmente problematica durante la scrittura
 - Scrivere in momenti diversi, oppure usare NVM
- Molti miglioramenti nelle tecniche di utilizzo del disco riguardano l'uso di dischi multipli che cooperano

RAID

- Nel mirroring duplicate le scritture, però le richieste di lettura ad una delle unità, questo non migliora il trasferimento
- Disk **striping** usa un gruppo di dischi come un'unica unità di storage
 - Dati distribuiti (strisciati) su più dischi
 - **Bit-level striping**, ogni i-esimo bit sul drive i-esimo
 - **Block-level striping**, blocco i-esimo su $(i \bmod n) + 1$ drive
 - ▶ Tipicamente questo è il metodo
- Due obiettivi principali
 - Incrementare il throughput di piccoli accessi bilanciando il carico
 - Ridurre il tempo di risposta di un accesso grande

RAID

- Mirroring porta ridondanza, ma è costoso
- Striping aumenta il trasferimento di dati, ma non l'affidabilità
- Alcuni schemi combinano striping con parity bit
- Organizzazioni a livelli RAID
- Es. 4 dischi per i dati, gli altri per ridondanza
 - Livello 0 non ridondanza, molto fragile
 - No fault tolerance: se un disco fallisce dati persi
 - Veloce, parallelismo, no parity control
 - Minimo numero di dischi 2
 - Livello 1 mirrored (C copia)
 - Dati duplicati, non overhead in velocità di scrittura
 - Velocizza la lettura (dati da più dischi)
 - Metà della capacità del disco
 - Minimo numero di dischi 2



RAID 2, 3, 4

□ RAID 2 poco utilizzato

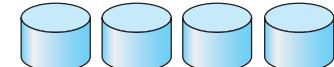
- Utilizza striping al livello di bit
- Utilizza hamming code per error detection/correction
- Minimo numero di dischi 3

□ RAID 3 poco utilizzato

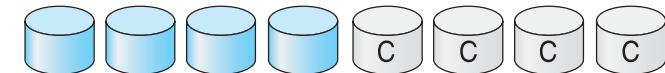
- Utilizza byte level striping
- Con un parity disk
- Minimo numero dischi 3

□ RAID 4

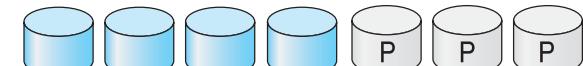
- Utilizza block level striping
- Con parity disk
- Scrittura lenta, scrittura del parity su unico disco
- Minimo numero dischi 3



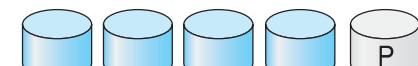
(a) RAID 0: non-redundant striping.



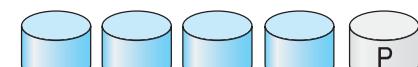
(b) RAID 1: mirrored disks.



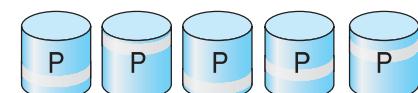
(c) RAID 2: memory-style error-correcting codes.



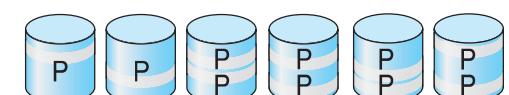
(d) RAID 3: bit-interleaved parity.



(e) RAID 4: block-interleaved parity.



(f) RAID 5: block-interleaved distributed parity.



(g) RAID 6: P + Q redundancy.

RAID 5, 6

- RAID 5 distribuisce i parity su tutti i dischi, evita di usare troppo il disco parity

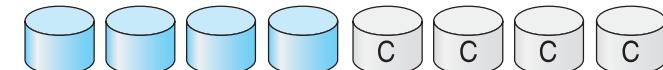
- Minimo numero di dischi 3
- Block level striping + ECC distribuito
- ECC usato con stripping
 - Il primo blocco in drive 1, scondo in drive 2, N in drive N, error in N + 1
- Permette anche la correzione dei dati
- Più veloce del livello 1 nell'accesso
- Scrittura migliore di livello 4

- RAID 6 simile

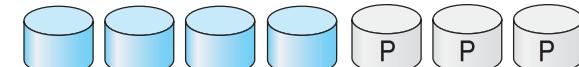
- Minimo numero di dischi 4
- aggiunge ridondanza per permettere recovery da fallimenti multipli
- 2 blocchi di parity distribuiti nei dischi
- Tollera due fallimenti di dischi
- Penalizzazione in scrittura



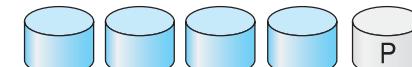
(a) RAID 0: non-redundant striping.



(b) RAID 1: mirrored disks.



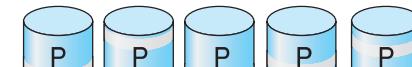
(c) RAID 2: memory-style error-correcting codes.



(d) RAID 3: bit-interleaved parity.



(e) RAID 4: block-interleaved parity.



(f) RAID 5: block-interleaved distributed parity.



(g) RAID 6: P + Q redundancy.

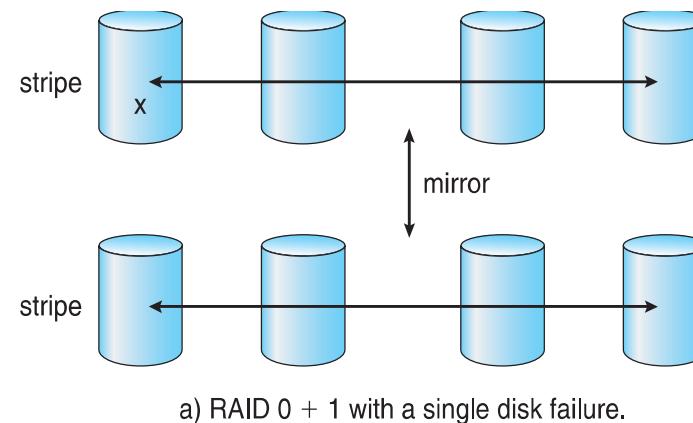
RAID (0 + 1) e (1 + 0)

Nested RAID o Hybrid RAID combinano i livelli standard

Striped mirrors (**RAID 1+0**) o mirrored stripes (**RAID 0+1**) forniscono alta performance e alta affidabilità: RAID 0 performance, RAID 1 affidabilità

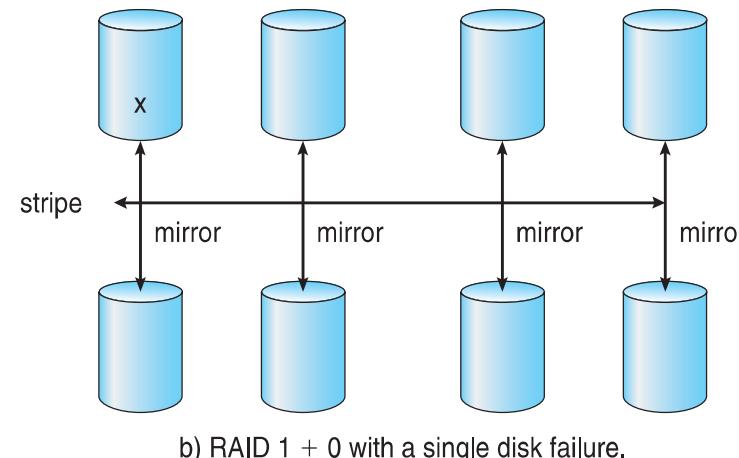
RAID 01 (0 + 1)

Dati striped e duplicate



RAID 10 (1 + 0)

Prima dati duplicati
poi striped sui dischi
(almeno 4)
Vicino a RAID 0 per
performance (usato
per sistemi I/O intense)



RAID

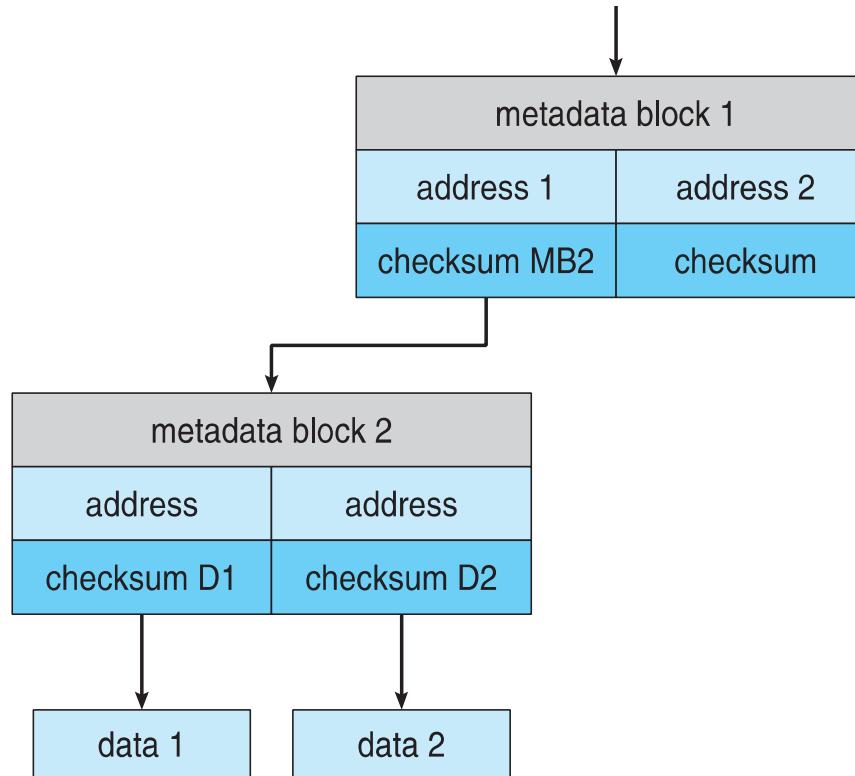
- Mirroring porta ridondanza, ma costoso
 - Striping aumenta il trasferimento di dati, ma non l'affidabilità
 - Alcuni schemi combinano striping con parity bit
-
- RAID è organizzato in sei livelli differenti
 - Gli schemi RAID aumentano le performance e l'affidabilità dello storage memorizzando dati ridondanti
 - **Mirroring** o **shadowing** (**RAID 1**) mantiene un duplicato di ogni disco
 - Striped mirrors (**RAID 1+0**) o mirrored stripes (**RAID 0+1**) fornisce alta performance e alta affidabilità
 - **Block interleaved parity** (**RAID 4, 5, 6**) usa molta meno ridondanza
-
- RAID in uno storage array può sempre fallire se fallisce l'array, quindi la **replicazione** automatica dei dati tra array è tipica
 - Frequentemente, un piccolo numero di dischi **hot-spare** sono lasciati non allocati, replicando automaticamente un disco danneggiato con dati ricostruiti

Altre Caratteristiche

- Indipendentemente da come il RAID è implementato si possono aggiungere altre caratteristiche
- **Snapshot** è una vista del file system prima che occorra un insieme di cambiamento (i.e., ad un certo istante di tempo)
- La replicazione è una duplicazione automatica delle scritture tra siti separati
 - Per ridondanza e disaster recovery
 - Può essere sincrona o asincrona
- Il disco hot spare (di ricambio) non è usato, solo in caso di fallimento per replicare il disco fallito e ricostruire il RAID set se possibile
 - Diminuisce il mean time to repair

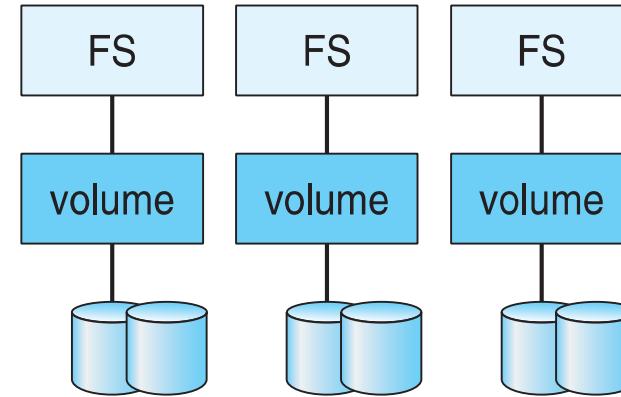
Estensioni

- RAID da solo non previene o trova tutti gli errori, solo errori di disk failures
- Solaris ZFS ha approccio innovativo
 - Aggiunge un **checksum** ad ogni dato e metadato
 - ▶ Checksum del blocco mantenuto in blocco separato: nel puntatore al blocco
 - Se checksum non corrisponde può trovare e correggere errori

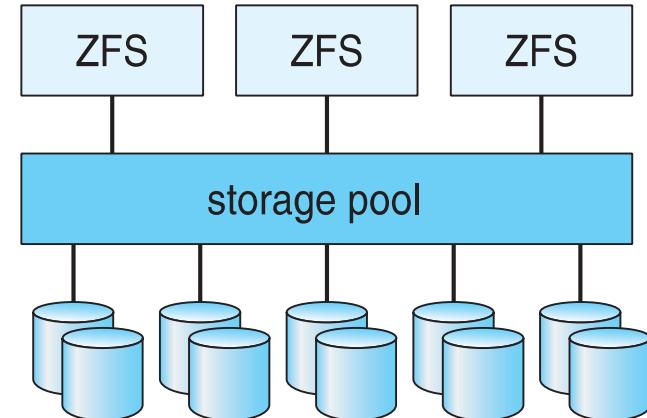


Estensioni

- Solaris ZFS gestisce in modo flessibile anche il file system su volumi e partizioni
 - Dischi allocate in **pool**
 - Filesystem allocati su un pool condivide quel pool dove lo spazio si usa e rilascia come con le chiamate `malloc()` e `free()`
 - Non c'è bisogno di fare resize o riallocazione del file system sui volumi



(a) Traditional volumes and file systems.



(b) ZFS and pooled storage.

Implementazione di uno Stable-Storage

- Scritture su disco hanno 3 possibili esiti
 1. **Successful completion** - dati scritti correttamente
 2. **Partial failure** - faillimento avvenuto durante il trasferimento, solo alcuni settori scritti e i settori scritti durante il faillimento possono essere corrotti
 3. **Total failure** - fallimento prima che la scrittura su disco parta quindi i dati precedenti su disco sono intatti
- Stable storage significa che i dati non sono persi durante la scrittura (failure, etc)
 - Scrittura atomica
- Per implementare uno stable storage:
 - Replica l'informazione su più di un dispositivo nonvolatile con modalità di fallimento indipendente
 - Aggiorna l'informazione in modalità controllata per assicurare che si possa recuperare i dati stabili dopo ogni failure durante il trasferimento dati o il recovery

Implementazione Stable-Storage

- Se il fallimento occorre durante il block write, le procedure di recovery block ristabiliscono uno stato consistente
 - Vengono mantenuti 2 blocchi fisici per blocco logico e procede come segue:
 1. Scrive sul primo bocco fisico
 2. Se ha successo scrive sul secondo blocco fisico
 3. Dichiara completamento solo dopo che il secondo write completa con successo

Sistemi talvolta usano NVRAM come un supporto fisico per velocizzare la scrittura

Memoria Principale



Obiettivo

- Fornire una descrizione dettagliata dei vari modi di organizzare l'hardware di memoria
- Discutere varie tecniche di gestione della memoria, inclusi il paging e la segmentazione

Background

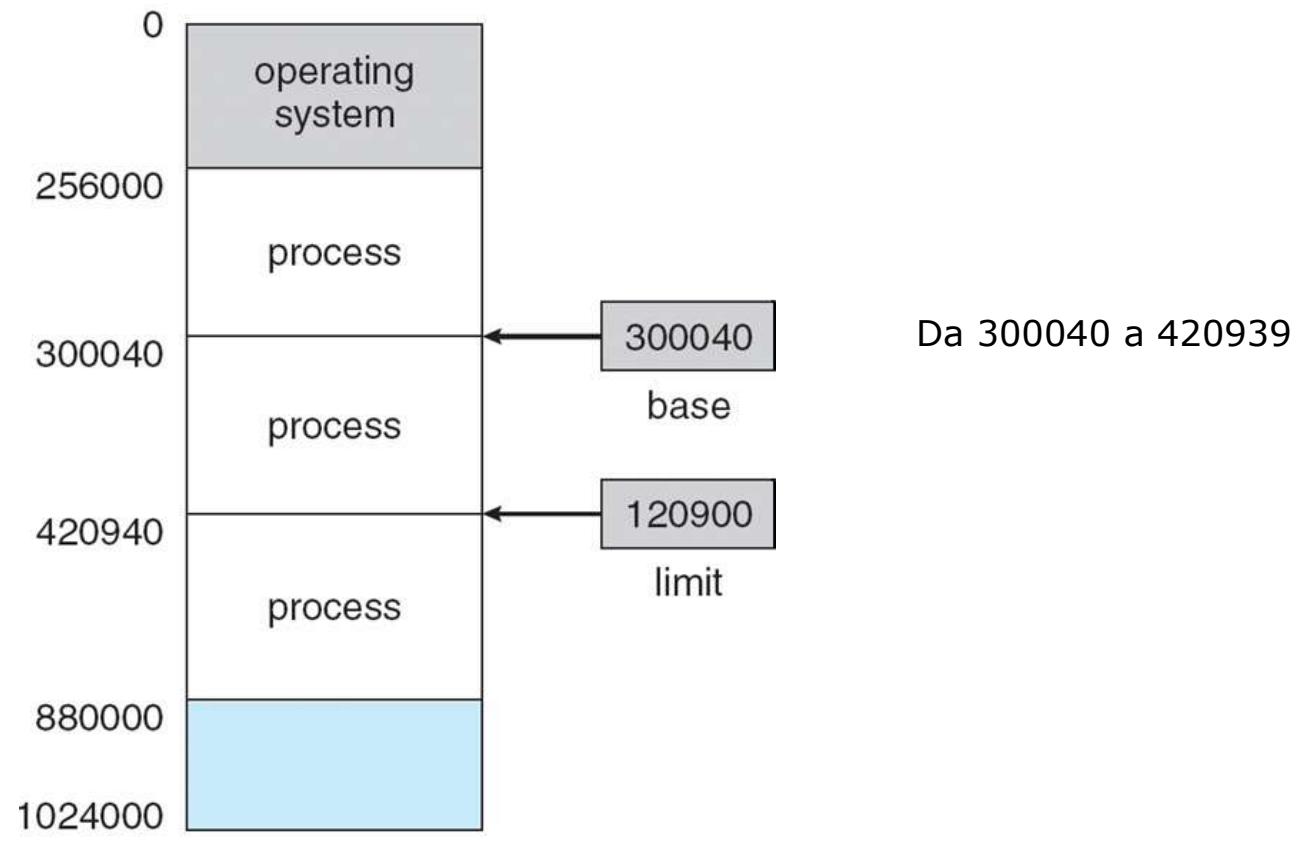
- Un sistema di elaborazione esegue programmi
 - Per essere eseguito un programma deve essere portato (dal disco) in memoria e inserito all'interno di un processo
 - La memoria principale ed i registri sono la sola memoria a cui la CPU accede direttamente interpretando le istruzioni dei programmi
 - Tanti processi devono essere caricati (almeno parzialmente) in memoria contemporaneamente
-
- La CPU carica istruzioni indicate dal PC, decodifica ed esegue
 - La memoria principale può essere vista come un grande array di bytes con indirizzi
 - L'unità di memoria vede solo un flusso di indirizzi + richieste di lettura o indirizzo + dati e richieste di scrittura

Hardware

- La memoria principale ed i registri sono la sola memoria a cui la CPU accede direttamente interpretando le istruzioni
- Supporto hardware per velocizzare l'accesso:
 - L'accesso ai registri in un clock della CPU (o anche meno)
 - La memoria principale può richiedere molti cicli (memory bus), causando uno stallo in attesa del dato da elaborare
 - Accesso frequente, troppi stalli
 - Cache tra la memoria principale e i registri della CPU per accelerare
- Protezioni hardware:
 - Necessaria protezione hardware della memoria per garantirne il corretto funzionamento (proteggere sistema e utenti)
 - Il Sistema non interviene tra CPU e Memoria per non rallentare

Protezione: Registri Base e Limite

- In primo luogo occorre separare i processi in memoria
- Una coppia di registri **base** e **limite** definisce lo spazio degli indirizzi logici per un processo
 - Base il primo indirizzo fisico, limite il range
- La CPU deve controllare ogni accesso in memoria in user mode per verificare che sia tra gli indirizzi base e limite per quel processo

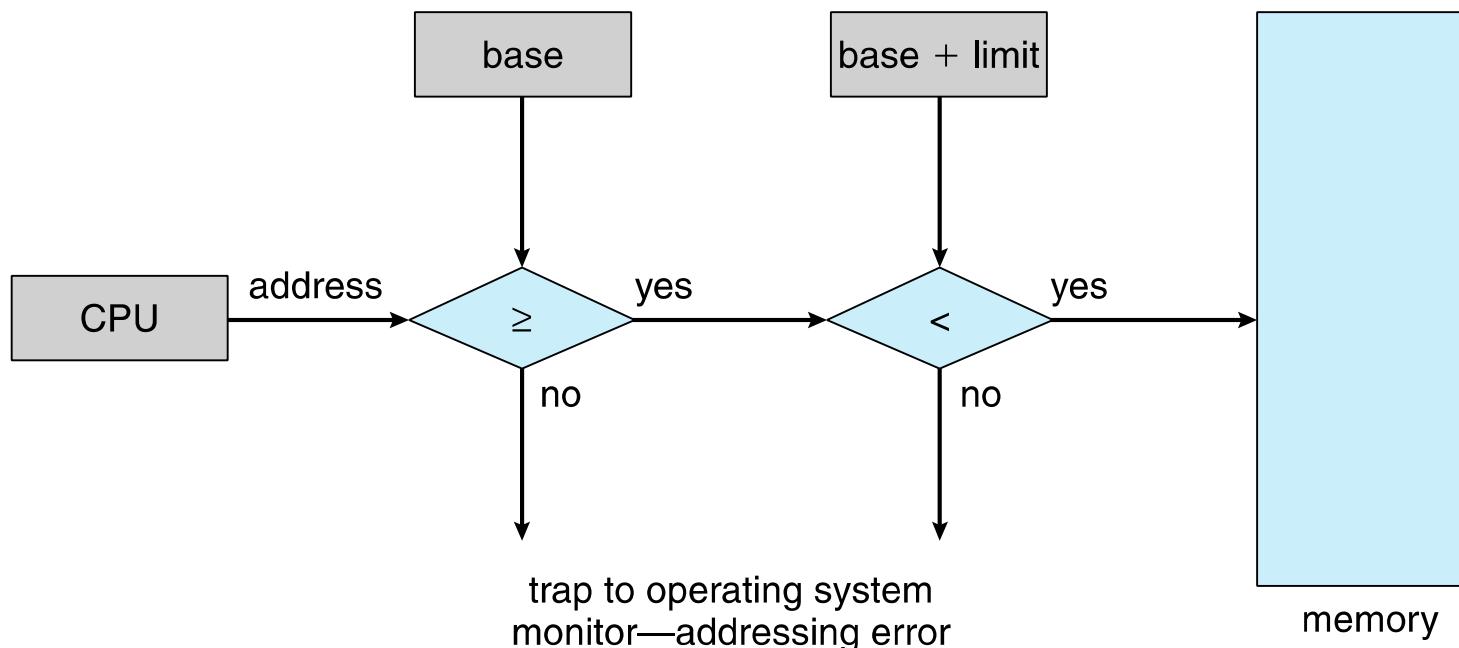


Protezione hardware degli indirizzi

Il controllo di accesso è hardware, se violato l'accesso si genera un trap che il Sistema Operativo considera come fatal error

I registri possono essere caricati solo dal SO in modalità privilegiata, quindi solo in kernel mode

In kernel mode il SO ha accesso non ristretto sia alla memoria di SO sia a quella degli utenti



Binding degli Indirizzi

- I programmi su disco sono file eseguibili pronti per essere caricati ed eseguiti
- I processi in attesa di essere caricati in memoria formano una coda di input
- Una volta caricati la CPU può accedere a dati ed istruzioni
 - In linea di principio il processo può risiedere in ogni parte della memoria
- Quando il processo termina lo spazio di memoria viene reso disponibile

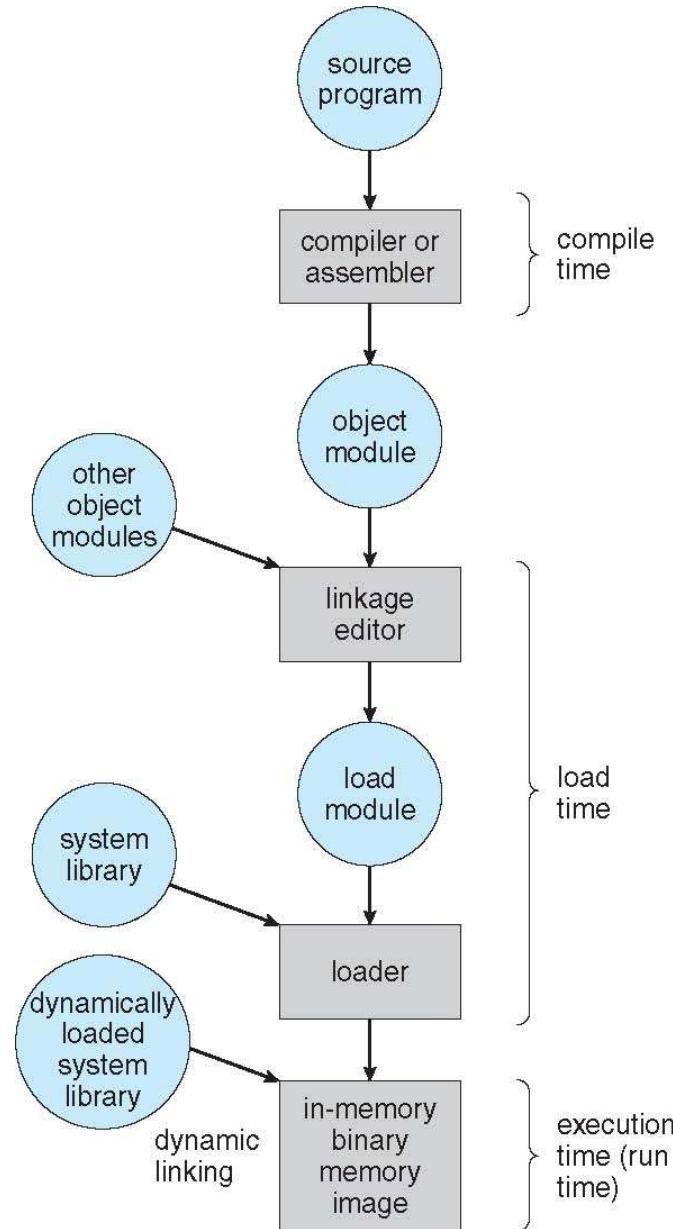
- Gli indirizzi sono rappresentati in modi diversi nei momenti diversi del ciclo di vita di un programma:
 - Nel codice sorgente sono simbolici (e.g., variabile count)
 - Nel codice compilato gli indirizzi sono associati (**bind**) ad indirizzi rilocabili
 - ▶ es. “14 bytes dall’inzio di questo modulo”
 - Il loader legherà gli indirizzi rilocabili agli indirizzi assoluti
 - ▶ es. 74014
- Il binding rappresenta un collegamento che mappa uno spazio di indirizzi in un altro

Binding di Istruzioni e Dati alla Memoria

- Il binding (degli indirizzi di istruzioni e dati) agli indirizzi di memoria può avvenire in diverse fasi
 - **Tempo di Compilazione:** se al momento della compilazione è già noto dove il processo risiederà in memoria si può generare **codice assoluto**, il codice va quindi ricompilato se la locazione di partenza viene modificate
 - **Tempo di Caricamento:** si genera **codice rilocabile** se la locazione di memoria non è nota a tempo di compilazione; il collegamento finale viene ritardata finché non c'è il caricamento. Se la locazione iniziale per il processo cambia occorre ricaricare il codice.
 - **Tempo di Esecuzione:** se durante l'esecuzione il processo può essere spostato da un segmento di memoria all'altro allora il binding è ritardato fino al run-time
 - ▶ Occorre supporto hardware speciale per mappare gli indirizzi

Processamento Multistep di Programma Utente

Un programma utente, prima di essere eseguito, deve passare attraverso *varie fasi*, alcune delle quali possono essere facoltative



Indirizzi logici e fisici

- La separazione tra **indirizzi logici** ed **indirizzi fisici** è centrale per la gestione della memoria
 - **Indirizzo logico**
 - generato dalla CPU; anche chiamato **indirizzo virtuale**
 - **Indirizzo fisico**
 - indirizzo visto dall'unità di memoria
 - caricato registro di indirizzi di memoria
- Il binding degli indirizzi a tempo di compilazione o caricamento rende identici gli indirizzi logici e fisici
- Gli indirizzi logici (virtuali) e fisici differiscono quando il binding è a tempo di esecuzione
- Indirizzi logici e fisici:
 - **Spazio degli indirizzi logici:** insieme di tutti gli indirizzi logici generate da un programma
 - **Spazio degli indirizzi fisici:** insieme di tutti gli indirizzi fisici generate da un programma

Memory-Management Unit (MMU)

- MMU è dispositivo hardware che a run-time mappa gli indirizzi virtuali in indirizzi fisici
- Diversi schemi di mapping possono essere introdotti
 - Consideriamo uno schema basato su registri base e limite
 - In questo caso il registro di base è chiamato registro di rilocazione

Unità di gestione della memoria (memory management unit, MMU) svolge l'associazione nella fase d'esecuzione dagli indirizzi virtuali agli indirizzi fisici.

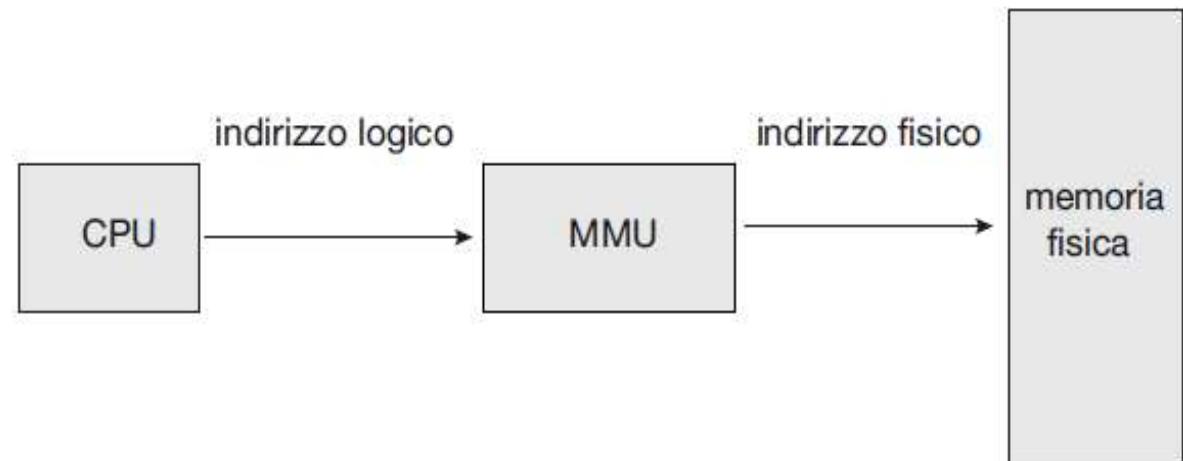
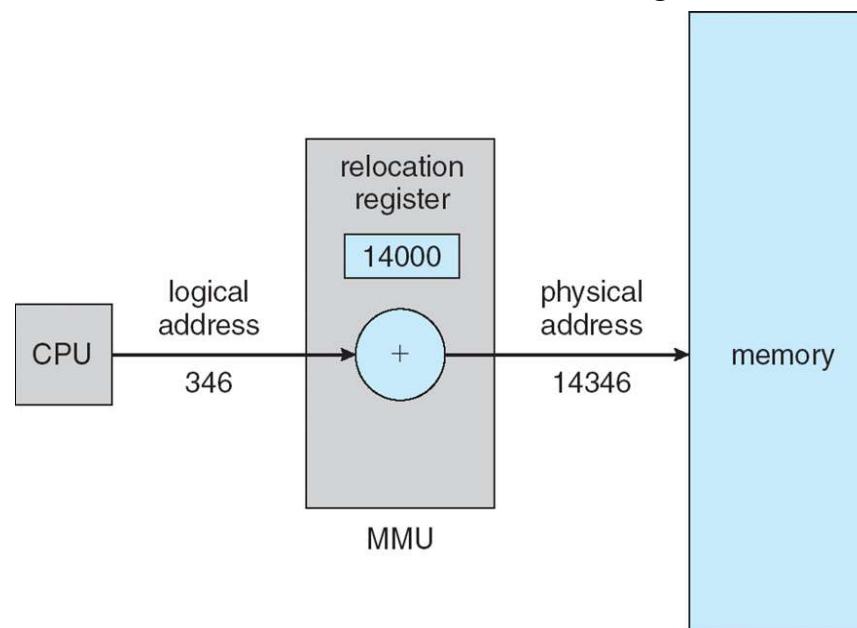


Figura 9.4 Unità di gestione della memoria (MMU).

Memory-Management Unit (MMU)

- MMU è dispositivo hardware che a run time mappa gli indirizzi virtuali in indirizzi fisici
- Diversi metodi sono possibili per fare il mapping
 - Consideriamo uno schema semplice dove il valore nel registro di relocazione è addizionato all'indirizzo generato dal processo utente
 - Registro base chiamato ora **relocation register**
 - Es. CPU considera 346 (logico) che viene mappato su 14346
 - MS-DOS su Intel 80x86 usava 4 relocation registers

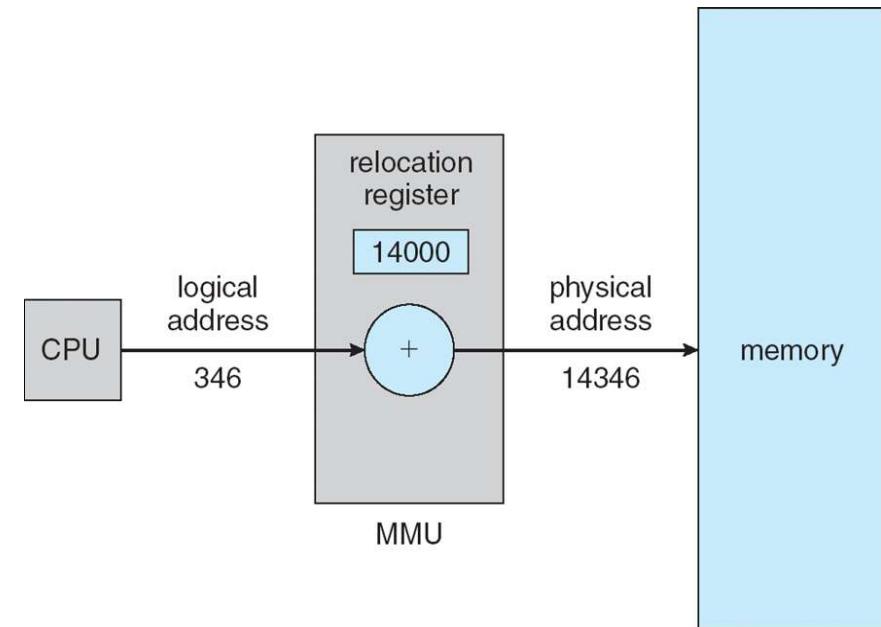


Memory-Management Unit (MMU)

- MMU è dispositivo hardware che a run time mappa gli indirizzi virtuali in indirizzi fisici
- Diversi metodi sono possibili per fare il mapping
 - Consideriamo uno schema semplice dove il valore nel registro di relocation è addizionato all'indirizzo generato dal processo utente
 - Registro base chiamato ora **relocation register**
 - MS-DOS su Intel 80x86 usava 4 relocation registers
- Il processo utente vede solo gli indirizzi logici, non vede mai gli indirizzi fisici reali
 - Es. Può gestire un puntatore, manipolarlo etc. rimanendo nello spazio logico
 - Il binding a tempo di esecuzione occorre solo quando si accede ad una locazione di memoria fisica. In questo caso l'hardware per il mapping in memoria converte l'indirizzo virtuale in indirizzo fisico
 - Il processo utente lavora nello spazio di indirizzi logici (es. intervallo [0, max]) che in memoria saranno mappati in indirizzi fisici (es. [R, R + max])

Caricamento Dinamico

- Non tutto il programma deve essere caricato per eseguire
- Miglior utilizzo dello spazio di memoria
 - Una routine non è caricata finché non è chiamata
 - routine non usate mai caricate
 - Tutte le routine tenute su disco in formato rilocabile
- Utile quando una grande quantità di codice necessaria per gestire casi non frequenti



Linking Dinamico

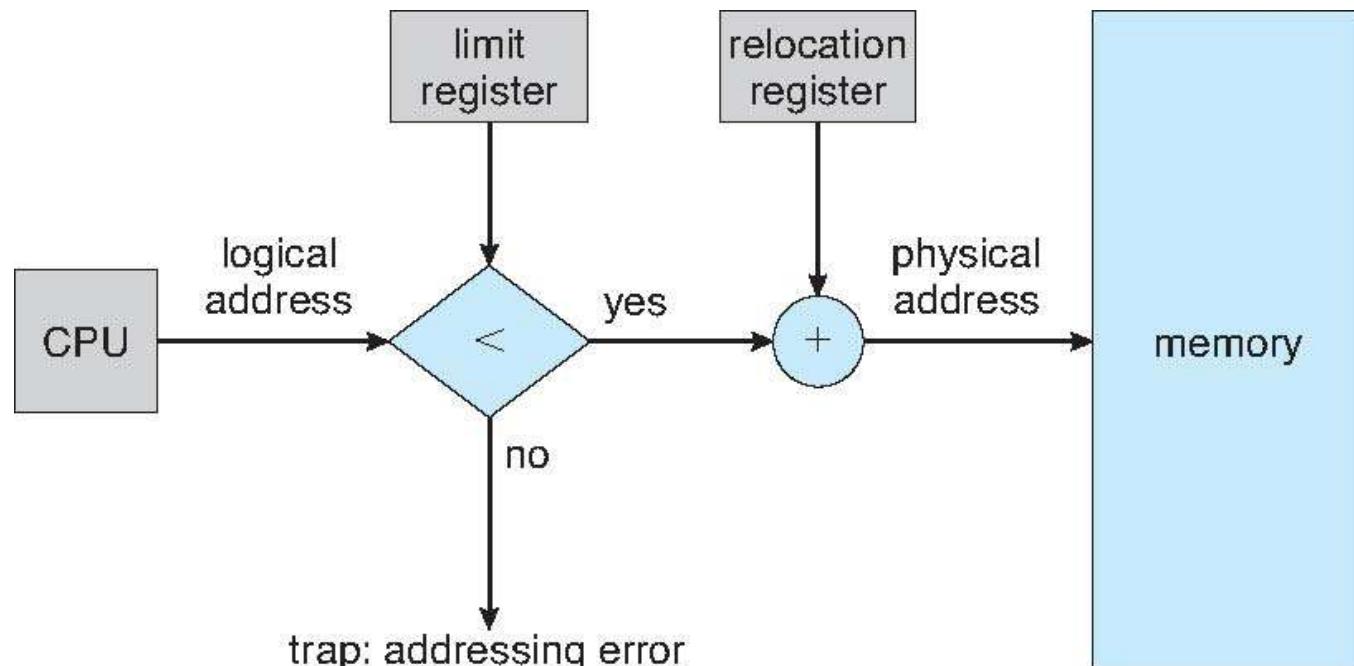
- **Linking statico** – librerie e codice del programma collegate dal loader nell'immagine binaria del programma in formato eseguibile
- **Linking dinamico** – linking è postposto fino al tempo di esecuzione
 - Simile al caricamento dinamico, invece del caricamento posticipato il link
 - Utilizzato per librerie di sistema (es. C standard library)
 - Dynamic Linking Libraries (DLLs)
 - ▶ evitano il caricamento completo per ogni programma
 - ▶ permettono la condivisione (**shared libraries**) tra processi multipli (usate in Win, Linux, etc.)
 - ▶ semplificano la gestione degli aggiornamenti e delle versioni
- Per ogni riferimento alla libreria inserito uno **stub** che indica come localizzare la routine in libreria
 - ▶ Lo stub sostituisce sè stesso con l'indirizzo della routine ed esegue la routine, al prossimo riferimento la routine verrà eseguita direttamente
- Richiesto supporto da parte del Sistema Operativo

Allocazione Contigua di Memoria

- La memoria principale deve supportare sia i processi di sistema che di utente
- Memoria principale di solito divisa in due **partizioni**:
 - Sistema Operativo residente tenuto in alta o bassa memoria
 - Linux e Win in memoria alta
 - Processi utente in bassa o alta memoria
 - Assumiamo bassa
- Più processi devono essere portati in memoria contemporaneamente
- Allocazione contigua è tra i primi metodi impiegati
 - Processi successivi caricati in sezioni contigue di memoria
 - Ogni processo contenuto in una singola e contigua sezione della memoria
 - Necessari meccanismi di protezione della memoria dei processi

Protezione

- Il registro di rilocazione usato per proteggere i processi utente l'uno dall'altro e per impedire il cambiamento del codice e i dati di SO
 - Registro base contiene il valore dell'indirizzo fisico più basso
 - Registro limite contiene il range di indirizzi logici – ogni indirizzo logico deve essere minore del limite
 - MMU mappa dinamicamente gli indirizzi logici
 - Il dispatcher carica il processo insieme ai registri limite e relocation
 - Anche la dimensione del SO può variare dinamicamente

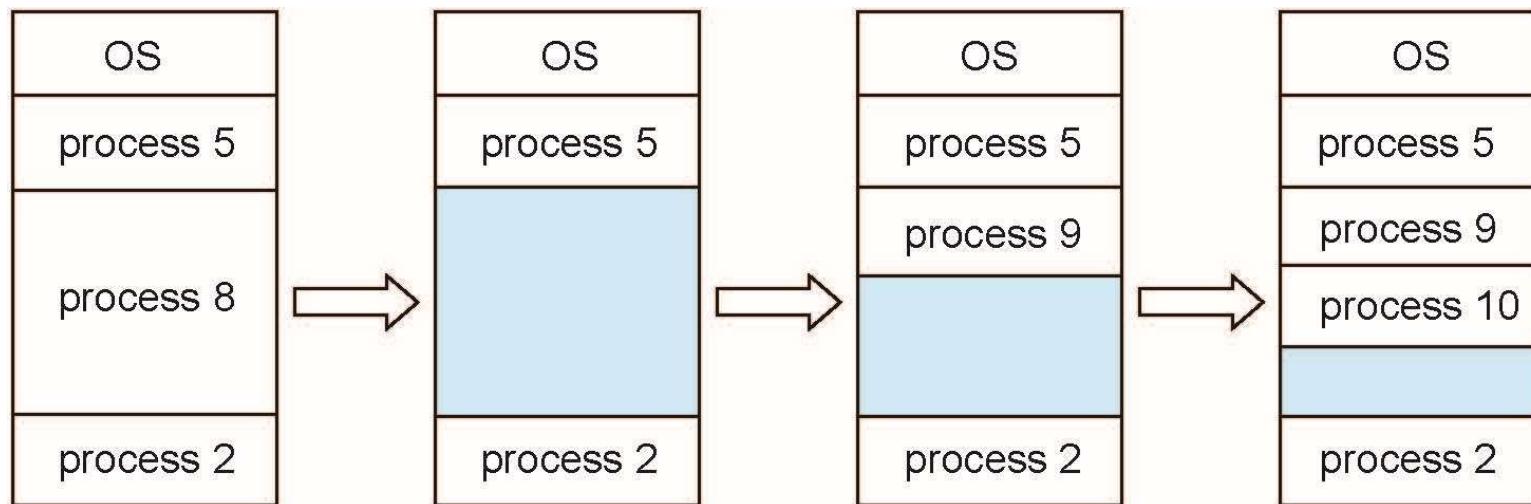


Allocazione con Partizioni Multiple

- Partizioni multiple di dimensione variabile
 - Suddividere la stessa in partizioni di **dimensione variabile** dove ciascuna partizione può contenere esattamente un processo
 - Tra i metodi più semplici per l'allocazione della memoria
 - Il SO mantiene traccia delle partizioni occupate
 - **Hole** – blocco di memoria disponibile
 - ▶ buchi di dimensioni varie sparpagliati per la memoria
 - Inizialmente tutta la memoria è disponibile (unico hole)
 - Quando un processo arriva, è allocato in un hole largo abbastanza per gestirlo
 - ▶ Dimensione **variable** della partizione (dimensionate sui bisogni del processo)
 - I processi uscenti liberano le partizioni e le partizioni libere adiacenti sono combinate in una sola
 - SO mantiene informazione su:
 - a) partizioni allocate b) partizioni libere (hole)
 - Grado di multiprogrammazione limitato dal numero di partizioni

Allocazione con Partizioni Multiple

- Partizioni multiple di dimensione variabile
 - Allocati gli hole dimensionati sui bisogni del processo



- Se non c'è spazio sufficiente il processo può essere respinto o messo in attesa
- Se il buco è grande viene diviso allocandone una parte e rilasciandone una parte libera
- Ma come scegliere gli hole da allocare?

Problema di Allocazione con Storage Dinamico

Come soddisfare una richiesta di dimensione n da una lista di hole liberi?

- **First-fit:** Alloca il **primo** hole libero che è grande abbastanza
 - Non deve scandire tutta la lista
 - Può scandire dall'inizio o dall'ultima ricerca
- **Best-fit:** Alloca il **più piccolo** hole che è grande abbastanza
 - Deve cercare l'intera lista se non ordinata con la dimensione
 - Produce il più piccolo residuo di hole
- **Worst-fit:** Alloca il **più largo** hole
 - Deve cercare l'intera lista, se non ordinata
 - Produce il più largo residuo di hole

Dalle simulazione si vede che first-fit e best-fit superiori a worst-fit in termini di velocità e utilizzo dello storage

First-fit più veloce, non chiaro invece per quanto riguarda lo storage

Frammentazione

- Problemi di **frammentazione esterna** sia per best-fit che per first-fit
- **Frammentazione esterna** – esiste memoria totale per soddisfare una richiesta, ma non è contigua
 - Tanti buchi piccolo non contigui
 - Con first-fit dati N blocchi allocati, $0.5 N$ blocchi sono persi per frammentazione
 - ▶ 1/3 può essere non usabile -> **50-percent rule**
- **Frammentazione interna** – la memoria allocata può essere più grande di quella richiesta
 - La differenza di dimensione è interna ad una partizione, ma non è usata
 - Esempio:
 - ▶ Processo richiede 18462 byte ma buco di 18464, troppo piccolo il resiuduo per tenerne traccia
 - ▶ Per evitare il problema si divide la memoria in blocchi di dimensione fissa, ma allora frammentazione interna

Frammentazione

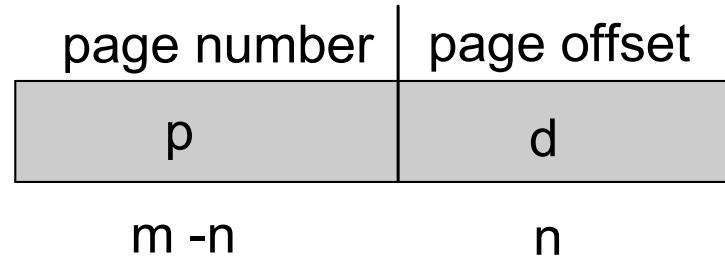
- La frammentazione esterna si riduce con la **compattazione**
 - Riassegna i contenuti della memoria per mettere insieme tutta la memoria libera in un blocco di grandi dimensioni
 - La compattazione è possibile se la rilocazione è dinamica ed è fatta a tempo di esecuzione
- Un'altra strategia è la **paginazione** che permette l'allocazione in locazioni non contigue:
 - Il paging è la strategia più usata

Paging

- Lo spazio dell'indirizzi fisici di un processo può essere non contiguo
 - Ad un processo è allocata memoria fisica quando è disponibile
 - Si evita la frammentazione esterna
 - Si evita il problema di frammenti di memoria con dimensioni variabili
- Si divide la memoria fisica in blocchi di dimensione fissa detti **frame**
 - Dimensione potenza di 2, tra 4 KB e 2 GB
- Si divide la memoria logica in blocchi della stessa dimensione detti **page**
- Si tiene traccia di tutti i frame liberi
- Per lanciare un programma di dimensione N page, si devono trovare N frame liberi e caricare il programma
- Occorre una **page table** per tradurre gli indirizzi logici in fisici
- Anche il backing store è suddiviso in pagine
- Ancora problemi di frammentazione interna

Schema di Traduzione degli Indirizzi

- L'indirizzo generato dalla CPU è diviso in due parti:
 - **Page number (p)** – usato come indice nella **page table** che contiene l'indirizzo base di ogni pagina nella memoria fisica
 - **Page offset (d)** – che combinato con l'indirizzo base definisce l'indirizzo di memoria fisica



- Con spazio degli indirizzi logici di dim 2^m e pagine di dim 2^n
- Spazio degli indirizzi logici sparato da quello degli indirizzi fisici

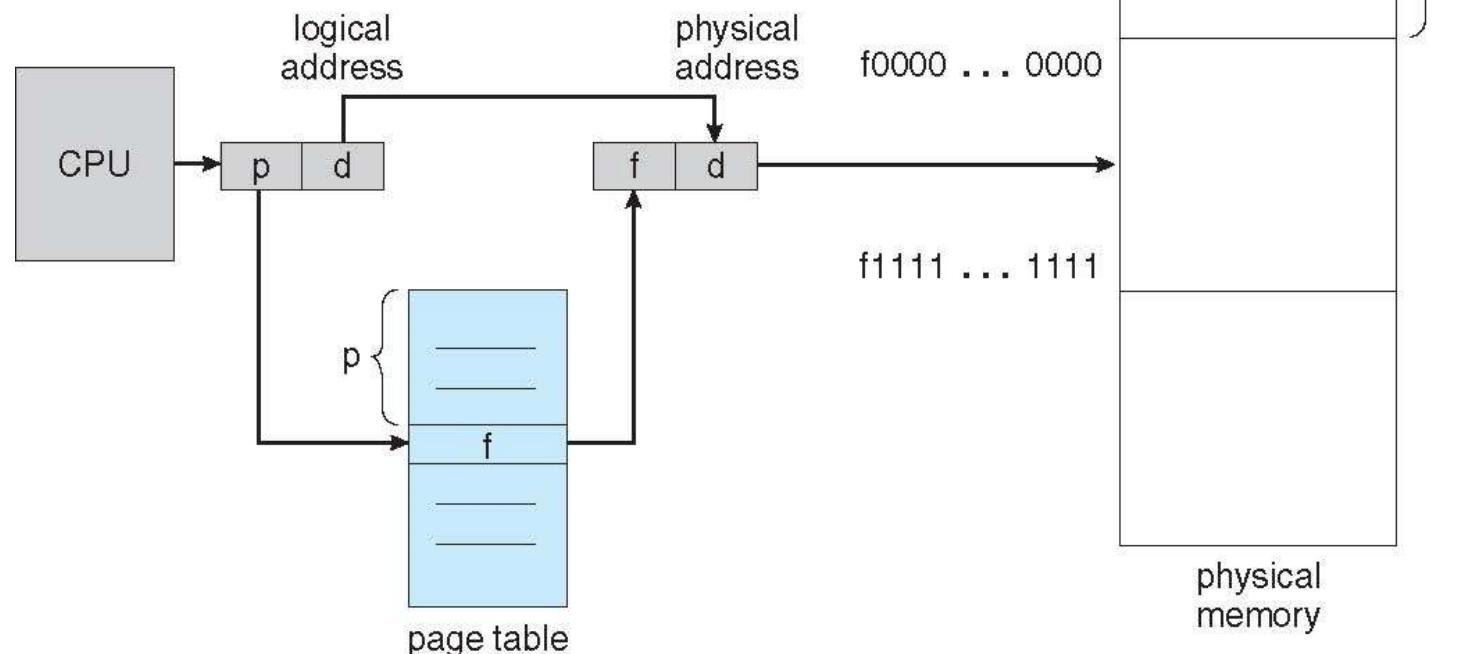
Hardware per il Paging

La MMU esegue i seguenti passi:

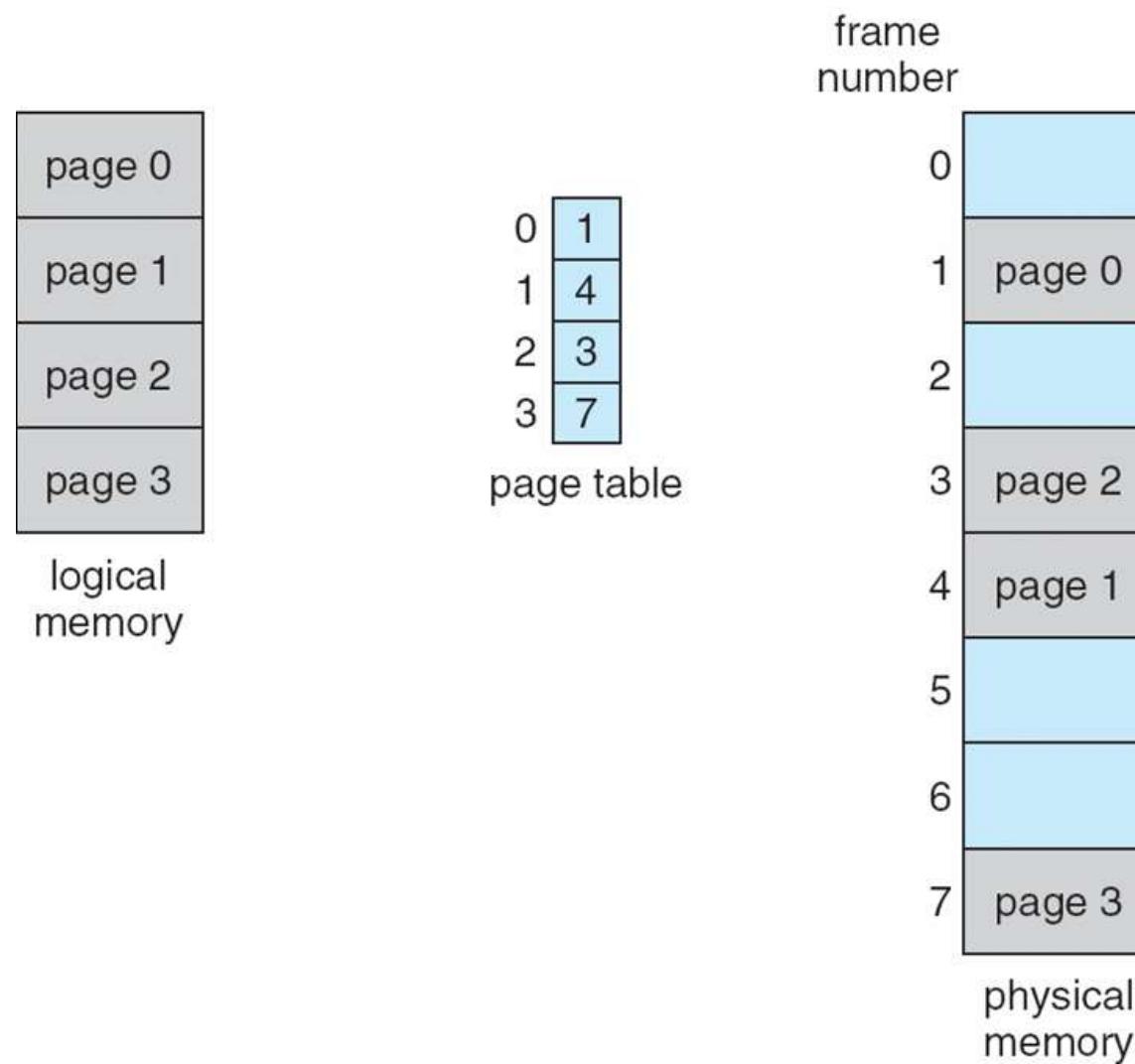
- Estraie in numero p di pagina usandolo come indice
- Estraie il frame number f dalla tabella
- Compone l'indirizzo sostituendo f a p

Le pagine con potenza di 2 semplificano l'operazione:

- Se 2^m indirizzi logici ed frame di dim 2^n allora
- $m - n$ bit per il page number n per l'offset



Modello di Paging di Memoria Logica e Fisica



Esempio Paging

Indirizzo logico 0 = pagina 0, offset 0

La pagina 0 è nel frame 5,
quindi 0 logico si mappa in 20 [= $(5 \times 4) + 0$] fisico

Indirizzo logico 3 (pagina 0, offset 3) mappa
nell'indirizzo fisico 23 [= $(5 \times 4) + 3$]

Indirizzo logico 4 è in page 1, offset 0;
page 1 si mappa nel frame 6.
Quindi 4 logico si mappa nel fisico 24 [= $(6 \times 4) + 0$]

| | |
|----|---|
| 0 | a |
| 1 | b |
| 2 | c |
| 3 | d |
| 4 | e |
| 5 | f |
| 6 | g |
| 7 | h |
| 8 | i |
| 9 | j |
| 10 | k |
| 11 | l |
| 12 | m |
| 13 | n |
| 14 | o |
| 15 | p |

logical memory

| | |
|---|---|
| 0 | 5 |
| 1 | 6 |
| 2 | 1 |
| 3 | 2 |

page table

| | |
|----|---|
| 0 | |
| 4 | i |
| | j |
| | k |
| | l |
| 8 | m |
| | n |
| | o |
| | p |
| 12 | |
| 16 | |
| 20 | a |
| | b |
| | c |
| | d |
| 24 | e |
| | f |
| | g |
| | h |
| 28 | |

physical memory

$n=2$ e $m=4$ memoria 32-byte e pagine 4-byte

Paging

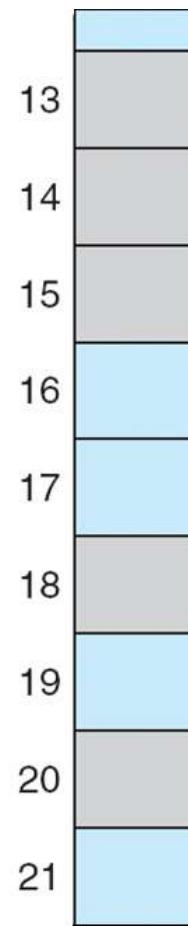
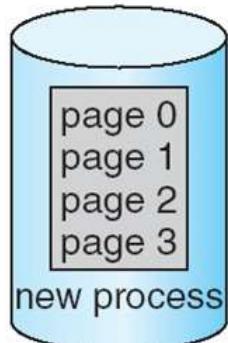
- Con paging solo frammentazione interna
- Calcolo della frammentazione interna
 - Esempio:
 - ▶ Page size = 2,048 bytes
 - ▶ Process size = 72,766 bytes
 - ▶ 35 pages + 1,086 bytes
 - ▶ Frammentazione interna di $2,048 - 1,086 = 962$ bytes
 - Frammentazione worst case = 1 frame – 1 byte
 - In media frammentazione = $1 / 2$ frame size
 - Quindi piccoli frame desiderabili?
 - ... ma ogni entry della page table impegna memoria per tracciarla
 - Le dimensioni delle pagine crescono nel tempo
 - ▶ Solaris supporta due dimensioni – 8 KB e 4 MB
- La vista del process è molto differente dalla memoria fisica
- Un processo può accedere solo alla sua memoria

Frame Liberi

Un processo da eseguire viene valutato in pagine e se richiede n pagine occorrono n frame liberi in memoria. Le pagine sono via via allocate e associate ai frame liberi

free-frame list

14
13
18
20
15

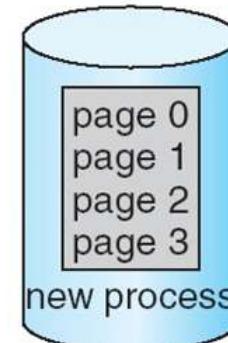


(a)

Before allocation

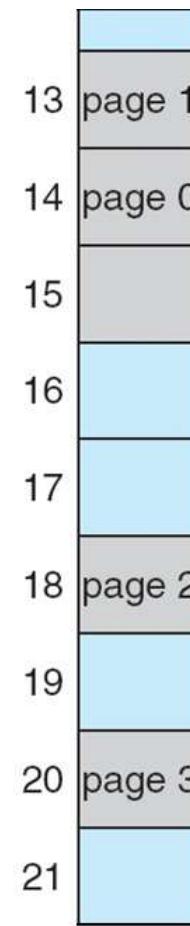
free-frame list

15



| | |
|---|----|
| 0 | 14 |
| 1 | 13 |
| 2 | 18 |
| 3 | 20 |

new-process page table



(b)

After allocation

Implementazione della Page Table

- Struttura per-processo con puntatore nel PCB
- Quando un processo va in esecuzione devono essere ricaricati i registri e ricaricati i valori per la gestione della page table
- La page table può essere implementata in vari modi
 - Registri dedicati caricati dal dispatcher della CPU, accesso del SO in modalità privilegiata, richiede gestione context switch, piccola tabella (es. 256 entry)
 - Se più grande (es. 2^{20} entry) la page table è mantenuta in memoria principale
 - **Page-table base register (PTBR)** punta alla page table
 - **Page-table length register (PTLR)** indica la dim della page table
 - Veloce context switch (cambia il puntatore)
 - Lento accesso a memoria. Ogni accesso a dati/instruzioni richiede due accessi in memoria: uno per la page table ed uno per i dati/instruzioni
 - I due accessi in memoria possono essere risolti utilizzando una cache speciale detta **registri associativi** o **translation look-aside buffers (TLBs)**
 - **TLB** è formata da registri chiave-valore, quando è presentato un elemento,

Memoria Associativa

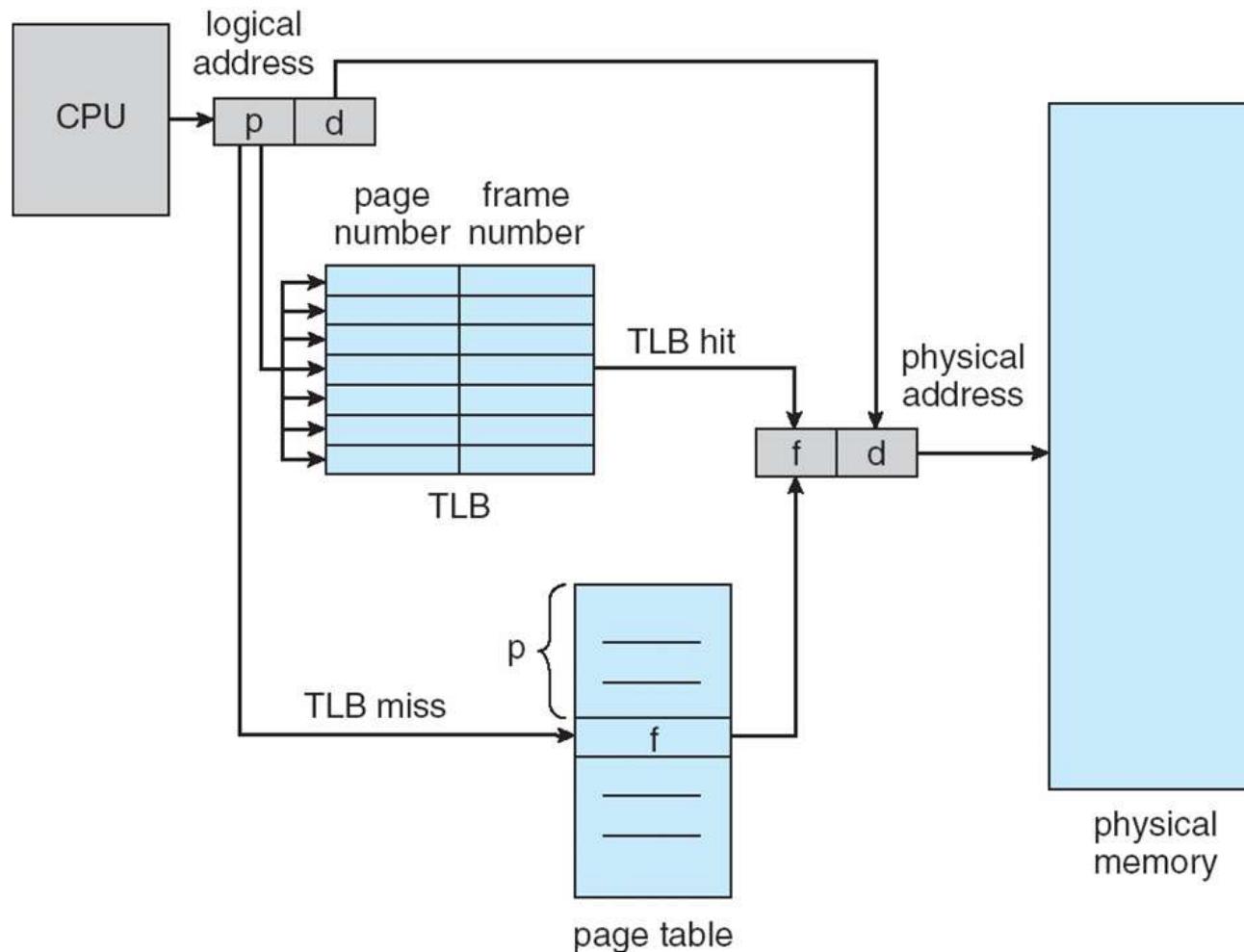
- Per la traduzione di un indirizzo si cerca prima in memoria associative e poi in memoria principale
- Memoria associativa – ricerca parallela

| Page # | Frame # |
|--------|---------|
| | |
| | |
| | |
| | |

- Traduzione di un indirizzo (p, d)
 - Se p è in un registro associativo, genera il frame # in output
 - Altrimenti (TLB miss) prendi il frame # dalla page table in memoria
 - ▶ La nuova coppia trovata può essere aggiunta nei registri

Hardware per Paging con TLB

Hardware necessario per implementare il paging con **Translation Look-aside Buffers (TLBs)**



Implementazione della Page Table

- Ogni volta che non si trova il valore in TLB (TLB miss), il valore è caricato sulla TLB per un accesso veloce la volta successiva
 - Se piena occorrono politiche di rimpiazzo
 - ▶ Least recently used (LRU), round robin, random
 - Alcune entry possono essere **cablate** per consentire l'accesso permanente
 - ▶ Es. Codice del kernel rilevante
- Alcune TLB hanno **address-space identifiers (ASIDs)** per ogni entry
 - Identifica univocamente il processo
 - Permette di mantenere info su diversi processi contemporaneamente
 - ▶ Altrimenti dovrebbe fare il flush per ogni context switch per evitare di far accedere il processo alle associazioni della page table di un processo precedente
 - Fornisce una protezione del suo spazio di indirizzi
 - ▶ Quando risolve l'indirizzo verifica se il processo corrente corrisponde a quello indicato, se non corrisponde si considera una TLB miss

Tempo di Accesso Effettivo

- Assumiamo ε unità di tempo la ricerca in TLB
 - Può essere < 10% del tempo di accesso in memoria
- Assumiamo la hit ratio = α
 - Hit ratio – percentuale di volte che si trova la pagina in TLB (dipende dal numero di registri)
 - ▶ Es. 80% significa 80% delle volte si trova la pagina
- Considiamo $\alpha = 80\%$, $\varepsilon = 20\text{ns}$ per una ricerca in TLB e 100ns per un accesso in memoria
- **Effective Access Time (EAT)**
$$\begin{aligned} \text{EAT} &= (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha) \\ &= 2 + \varepsilon - \alpha \end{aligned}$$
 - Se $\alpha = 80\%$, $\varepsilon = 20\text{ns}$ per la ricerca in TLB search, 100ns per accesso in memoria ... $\text{EAT} = 0.80 \times 100 + 0.20 \times 200 = 120\text{ns}$
 - Con un più realistico hit ratio -> $\alpha = 99\%$, $\varepsilon = 20\text{ns}$ per la ricerca in TLB, 100ns per accesso in memoria ... $\text{EAT} = 0.99 \times 100 + 0.01 \times 200 = 101\text{ns}$

Tempo di Accesso Effettivo

- Nei sistemi moderni più complicato il calcolo perché più livelli di TLB
 - Es. La CPU Intel Core i7 ha una TLB L1 da 128-entry per le istruzioni ed una TLB L1 dati da 64-entry
 - Se c'è il miss ad L1 occorrono 6 cicli di CPU per verificare la entry nella L2 512-entry TLB.
 - Se miss in L2 allora la CPU o cerca le entry in memoria, con costo di centinaia di clicli, oppure può interrompere per delegare al SO il compito ...
- TLB sono elementi hardware a supporto del paging, gli SO devono essere progettati tenendo presente le caratteristiche di questi elementi che possono variare a seconda della piattaforma

Protezione di Memoria

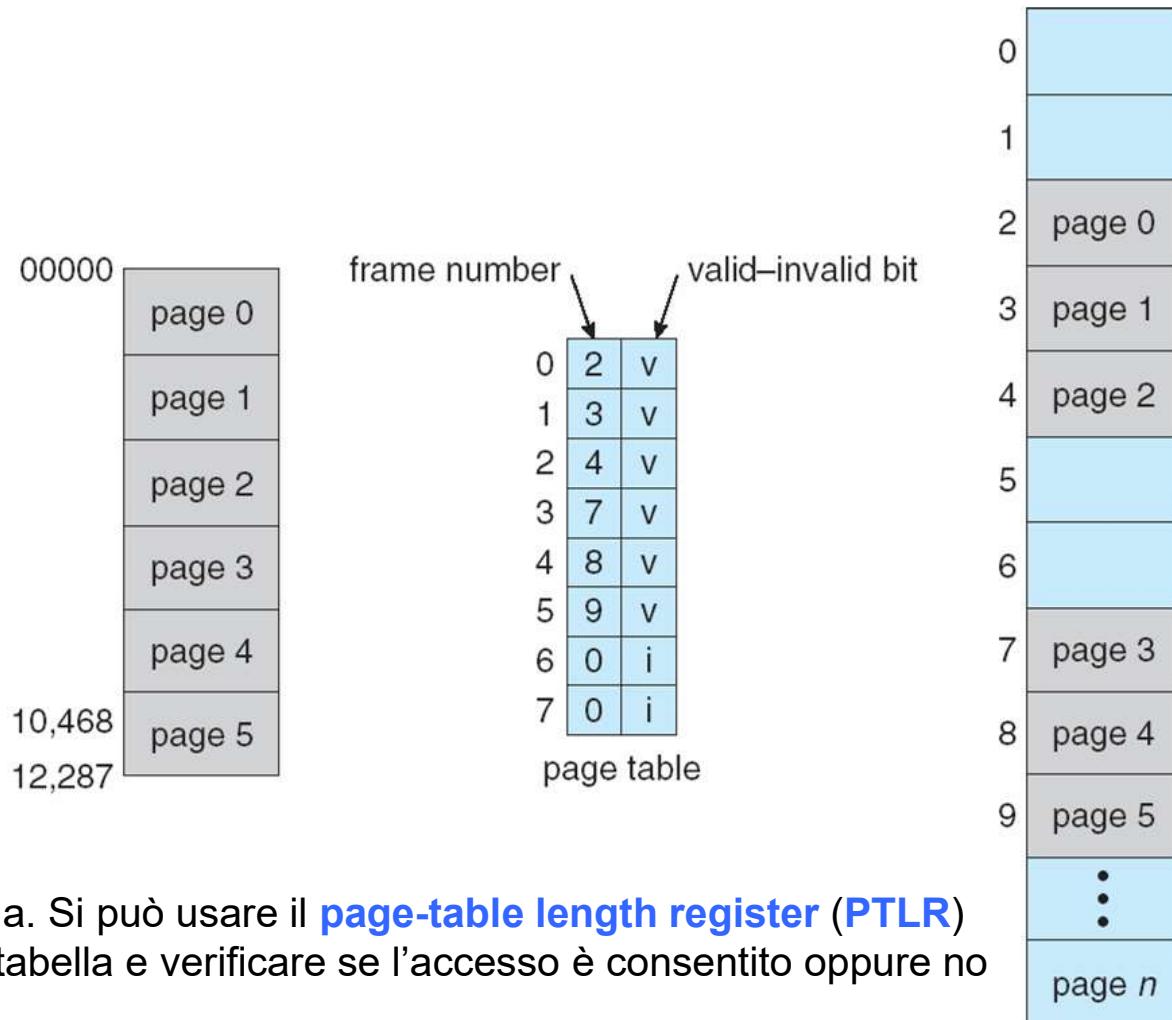
- La protezione della memoria è implementata associando dei bit di protezione per ogni frame per indicare i permessi (read-only, read-write)
 - Si possono aggiungere più bit per indicare execute-only, etc.
 - Mentre si cerca il frame si può controllare l'accesso
 - Una violazione dei permessi provoca un trap hardware
- Alle entry della page table sono associate anche bit **valid-invalid** :
 - “valid” indicata che la pagina nello spazio logico del processo è valida, quindi legale
 - “invalid” indica che la pagina non è nello spazio logico del processo
 - Ogni violazione provoca un trap al kernel
 - Il SO usa questi bit per permettere o vietare l'accesso alle pagine

Valid (v) or Invalid (i) Bit In A Page Table

Es. 14-bit address space [0, 16383], processo solo in [0, 10468]

Pagine di 2Kb, quindi le pagine 6, 7 non sono valide

Però pagina 5 parzialmente accessibile (frammentazione interna)



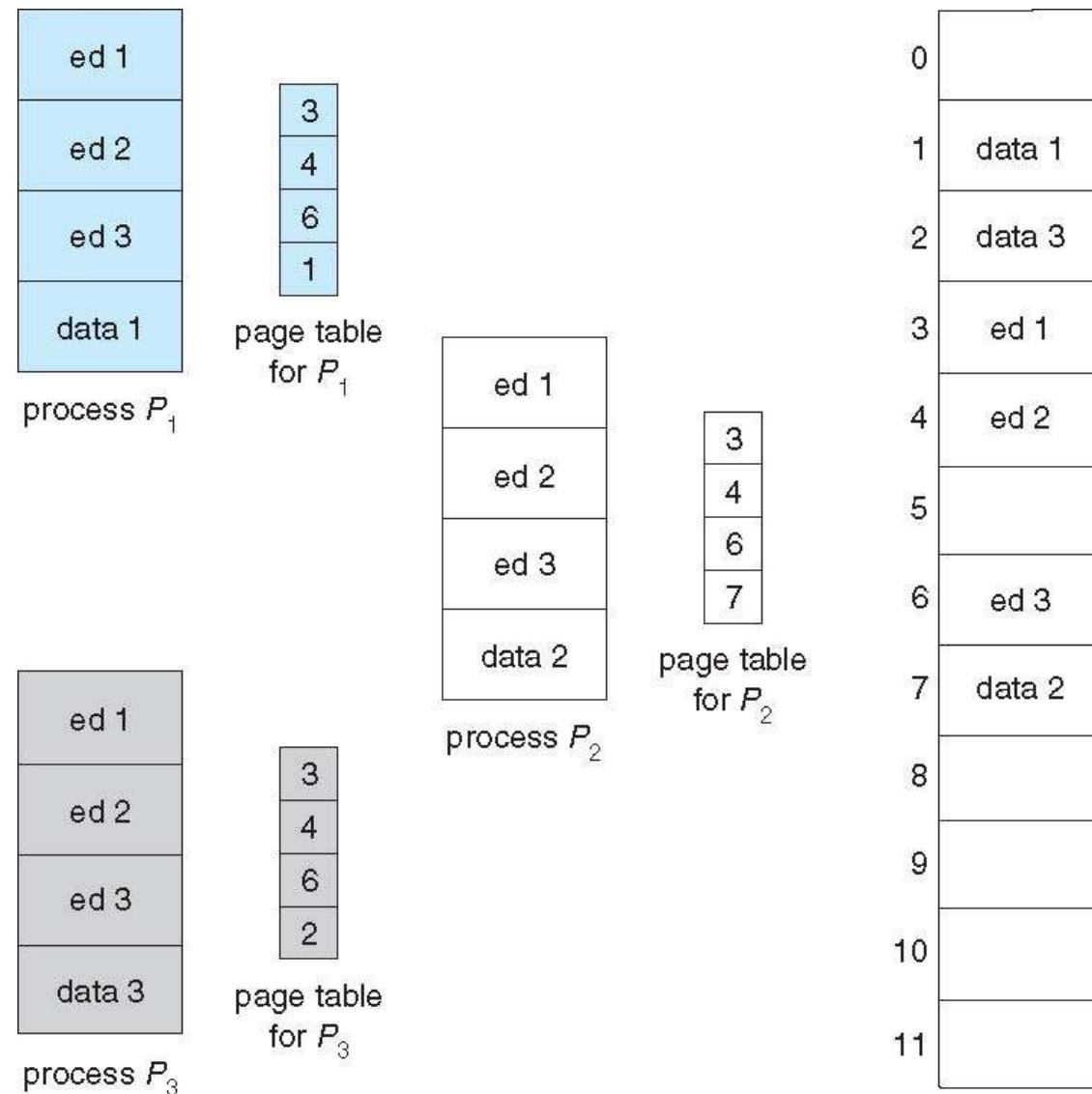
La granularità è al livello di pagina. Si può usare il **page-table length register (PTLR)** per indicare la dimensione della tabella e verificare se l'accesso è consentito oppure no

Pagine Condivise

- Vantaggio del paging è la possibilità di condividere pagine fra processi
 - Particolarmente vantaggioso in architetture multiutente
- **Codice condiviso**
 - Una copia di codice read-only (**rientrante**) condivisa tra processi (i.e., text editor, compilatori, librerie) evitando duplicazioni
 - Due o più processi possono condividere lo stesso codice
 - Es. Standard C library condivisa tra processi (se 2Mb per 40 utenti, 2Mb vs 80 Mb)
 - Simile a thread multipli che condividono lo stesso spazio di processo
 - Utile anche per interprocess communication se possibile condividere pagine read-write
- **Codice e dati privati**
 - Ogni processo mantiene copia separata del codice e dei dati
 - Le pagine per codice privato e dati può apparire ovunque nello spazio degli indirizzi logici

Esempio Pagine Condivise

- Esempio codice editor condiviso da tre processi



Struttura della Tabella delle Pagine

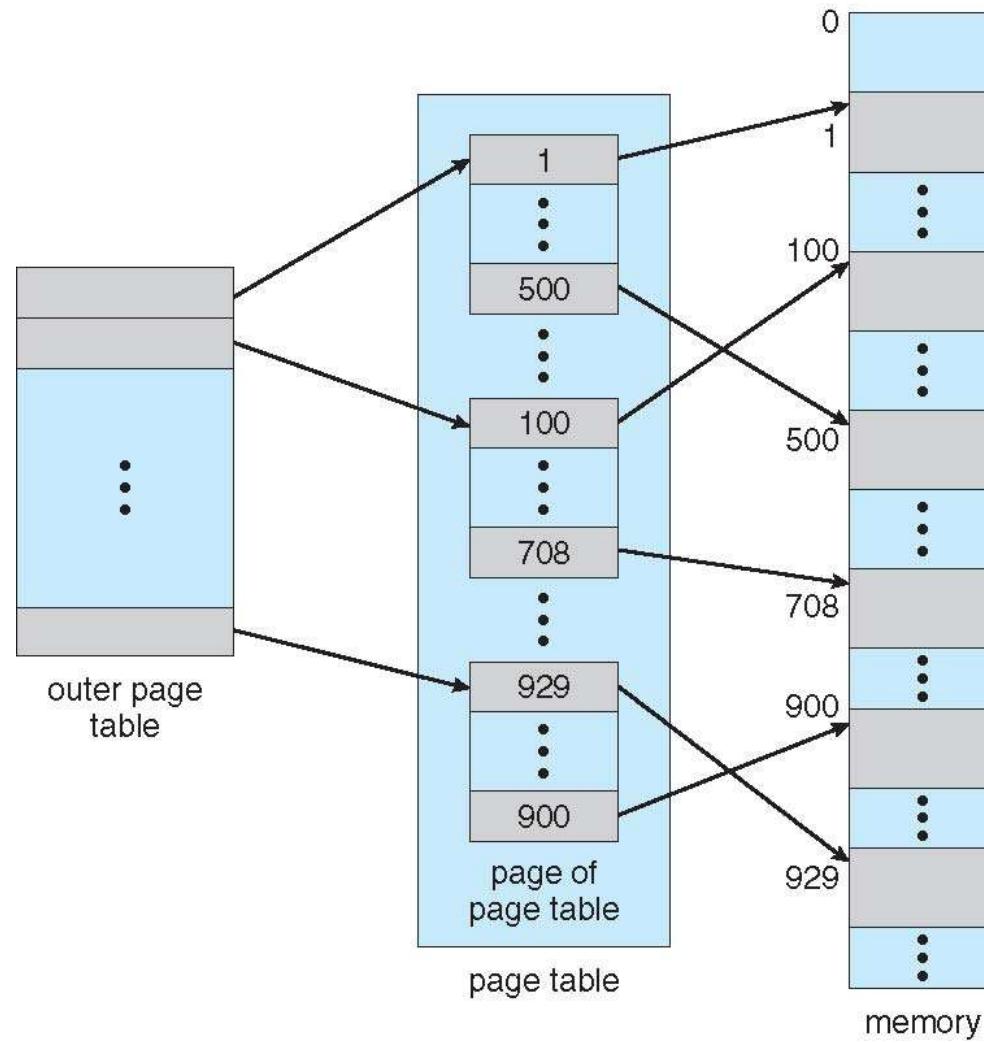
- Moderni elaboratori supportano spazi di indirizzi logici molto estesi (da 2^{32} a 2^{64})
- Struttura per la paginazione può diventare enorme
 - Se indirizzi logici di 32-bit e dimensione delle pagine di 4 KB (2^{12})
 - La page table avrebbe più di un milione di entry ($2^{32} / 2^{12}$)
 - Se ogni entry è di 4 byte -> 4 MB di spazio di indirizzi fisici solo per la tabella delle pagine
 - Alto costo di memorizzazione
 - Non si vuole allocare in modo contiguo in memoria principale
- Metodi
 - Hierarchical Paging
 - Hashed Page Tables
 - Inverted Page Tables

Tabella delle Pagine Gerarchiche

- Suddividere lo spazio degli indirizzi logici su più tabelle delle pagine
- Una tecnica semplice è la tabella delle pagine su più livelli
 - Si pagina la tabella delle pagine
 - Il page number è a sua volta diviso in page number e offset
 - Es. con spazio degli indirizzi logici a 2^{32} e pagine di 4KB

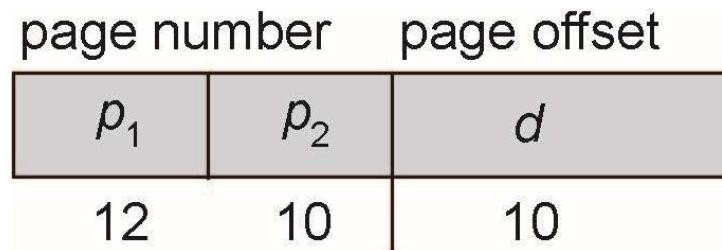
| page number | | page offset |
|-------------|-------|-------------|
| p_1 | p_2 | d |
| 10 | 10 | |
| | | 12 |

Tabella delle Pagine a Due Livelli



Esempio Paginazione a due Livelli

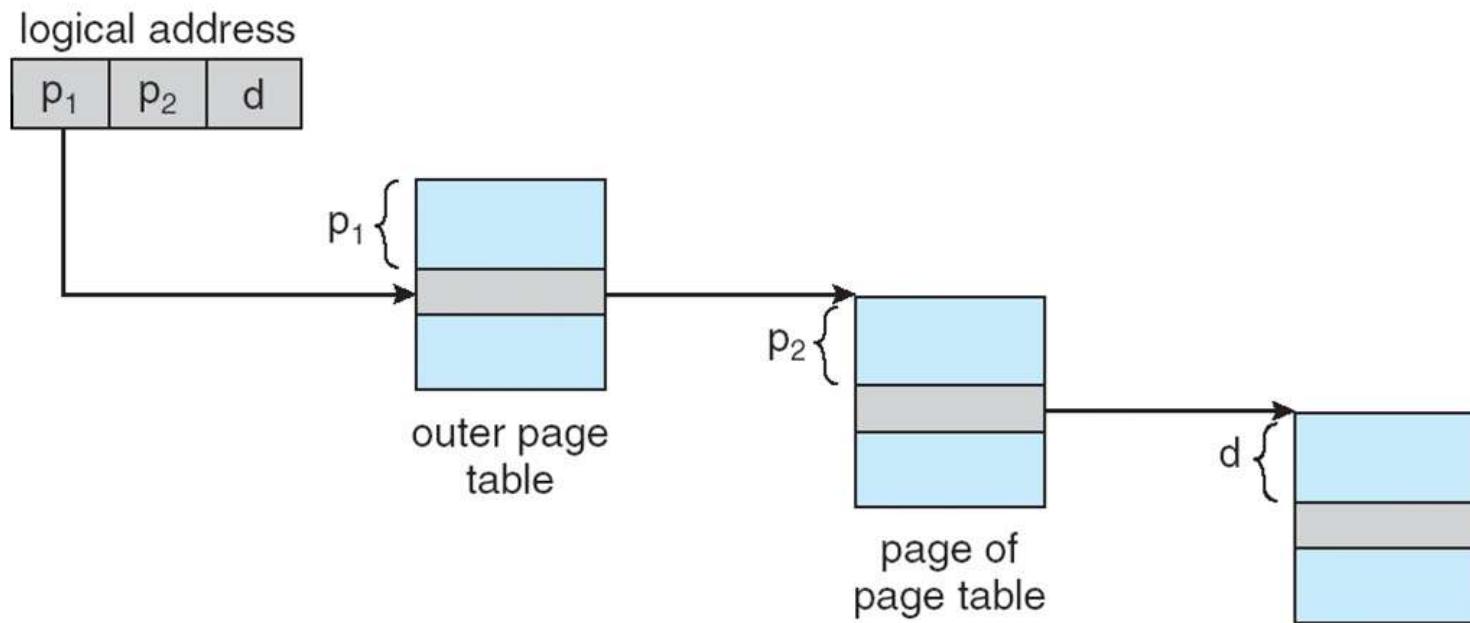
- Un indirizzo logico (su macchina a 32-bit con pagine da 1KB) diviso in:
 - page number di 22 bit
 - page offset di 10 bit
- La tabella delle pagine è paginate e il page number è ancora suddiviso in:
 - 12-bit page number
 - 10-bit page offset
- L'indirizzo logico è quindi:



- dove p_1 è indice della tabella esterna, p_2 è l'offset nella pagina della tabella delle pagine esterna
- Metodo chiamato **forward-mapped page table**

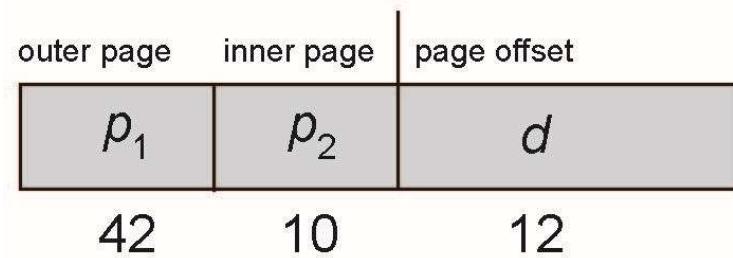
Schema di Address-Translation

- Metodo chiamato **forward-mapped page table**



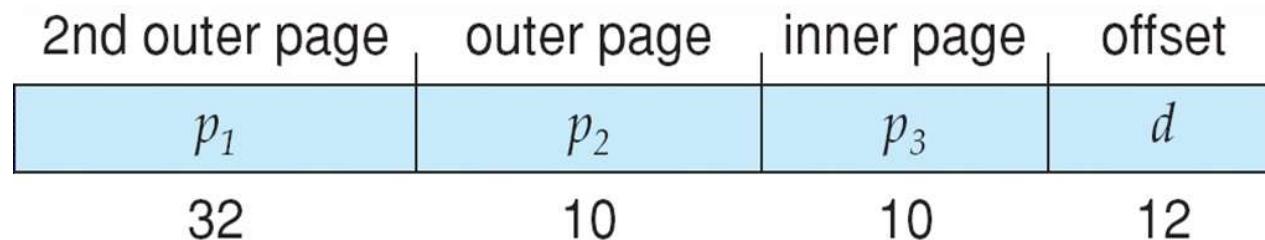
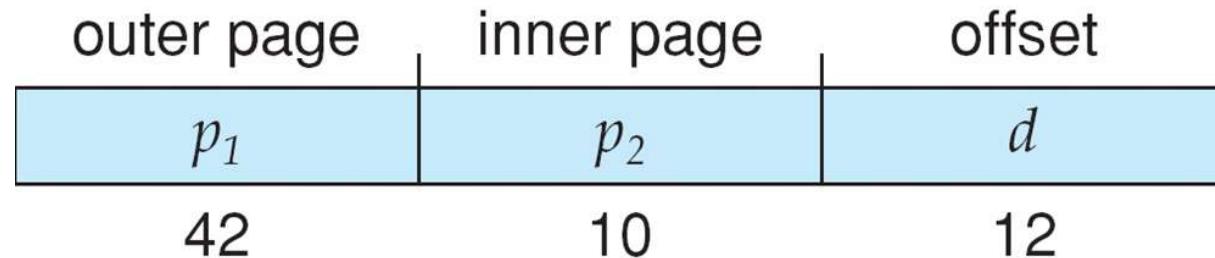
64-bit Logical Address Space

- Ma anche lo schema di paging a due livelli non è sufficiente
- Se la dimensione delle pagine fosse 4 KB (2^{12})
 - La page table avrebbe 2^{52} entry
 - Con lo schema a due livelli e pagine interne di 2^{10} con entry di 4-byte
 - L'indirizzo sarebbe



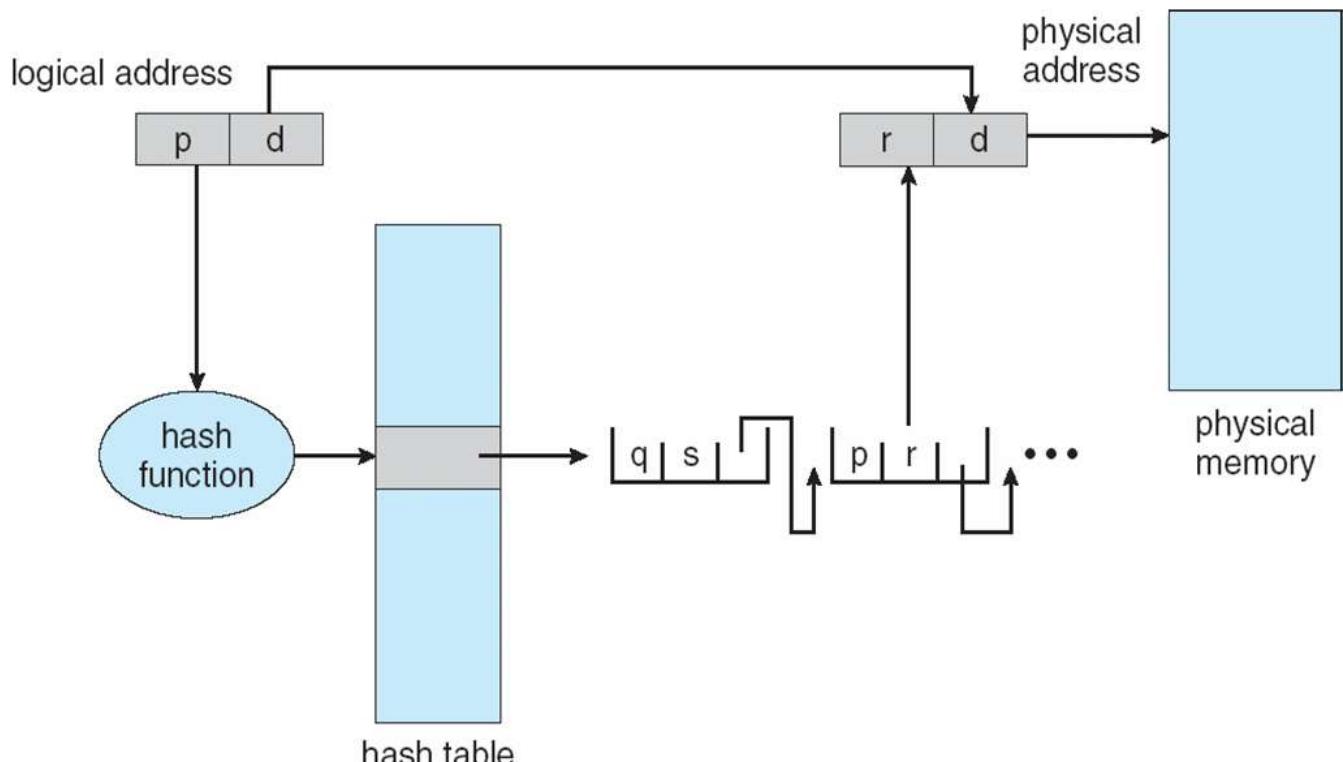
- La tabella esterna avrebbe 2^{42} entry
- Una soluzione prevede l'aggiunta di una seconda tabella esterna
- Ad esempio una seconda tabella esterna di 2^{34} byte
 - ▶ Può portare fino a 4 accessi in memoria per un singolo accesso

Schema di Paging a Tre Livelli



Hashed Page Table

- Tipico con spazio di indirizzi > 32 bits
- Il numero di pagina virtuale viene messo in una hash page table
 - La page table contiene una catena di elementi in hash sulla stessa locazione
 - Ogni elemento contiene (1) il numero di pagina virtuale (2) il valore del page frame mappato (3) un puntatore all'elemento successivo
 - I numeri di pagina virtuale sono confrontati nella catena in cerca di un match
 - Se un match è trovato viene estratto il frame fisico corrispondente



Hashed Page Table

- Tipico con spazio di indirizzi > 32 bits
- Il numero di pagina virtuale viene messo in una hash page table
 - La page table contiene una catena di elementi in hash sulla stessa locazione
 - Ogni elemento contiene (1) il numero di pagina virtuale (2) il valore del page frame mappato (3) un puntatore all'elemento successivo
 - I numeri di pagina virtuale sono confrontati nella catena in cerca di un match
 - Se un match è trovato viene estratto il frame fisico corrispondente
- Una variazione per indirizzi a 64-bit sono le **clustered page tables**
 - Simili alle hashed ma ogni entry nella hash si riferisce a molte pagine (16) invece di una sola
 - Molto utile per spazi di indirizzi **sparsi** (dove i riferimenti in memoria sono non contigui e sparpagliati)

Tabella delle Pagine Invertite

- Invece di avere ogni processo con una page table e tener traccia di tutte le pagine logiche si tracciano le pagine fisiche
 - Una entry per ogni pagina reale di memoria
 - Le entry consistono dell'indirizzo virtuale della pagina contenuta in memoria reale con informazione sul processo che possiede quella pagina
- $\langle \text{process-id}, \text{page-number}, \text{offset} \rangle$
- Minore memoria per le tabelle delle pagine, ma aumenta il tempo necessario per cercare la tabella quando c'è un riferimento ad una pagina
 - Usata da 64-bit UltraSPARC e PowerPC

Architettura delle Tabella delle Pagine invertite

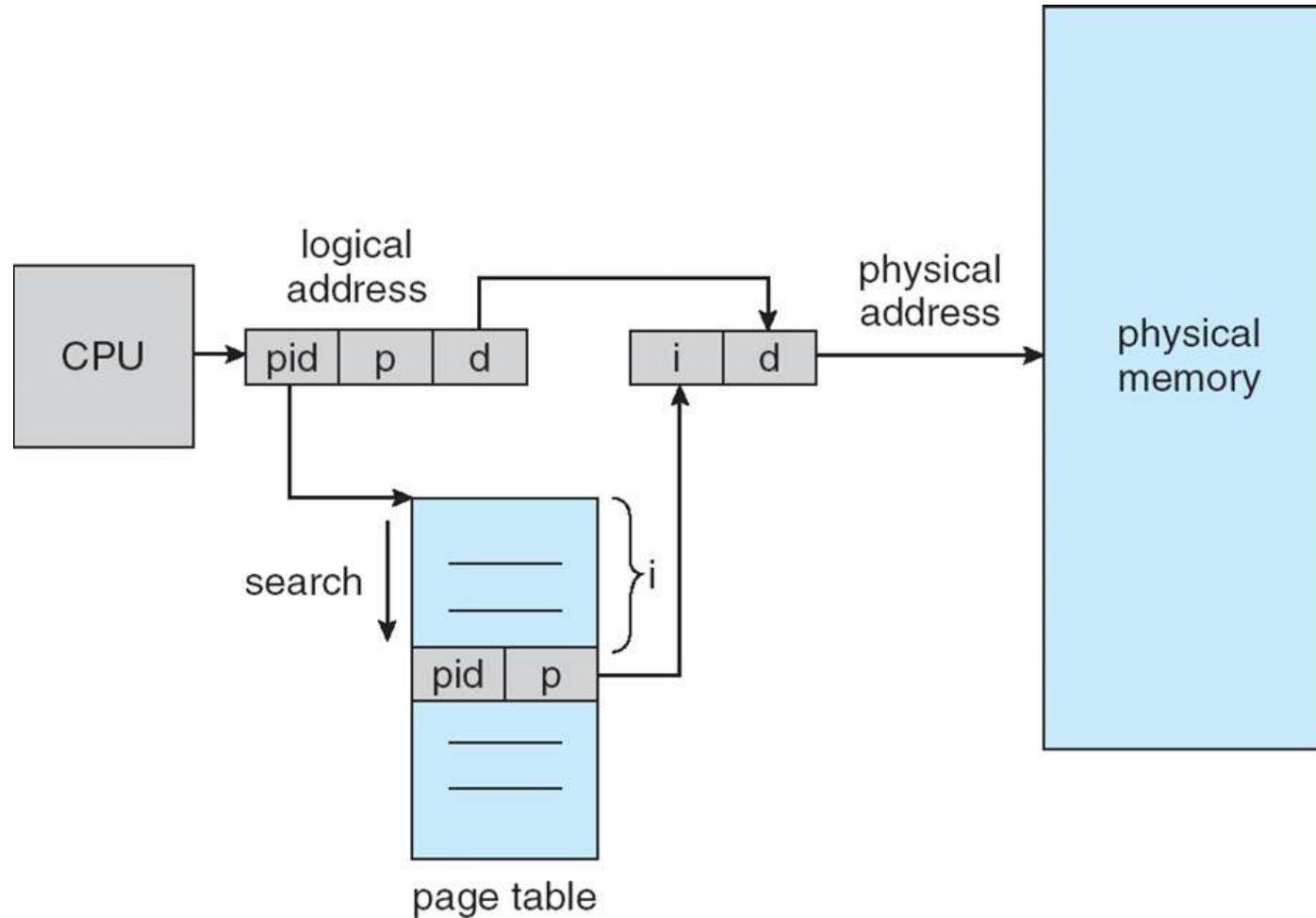


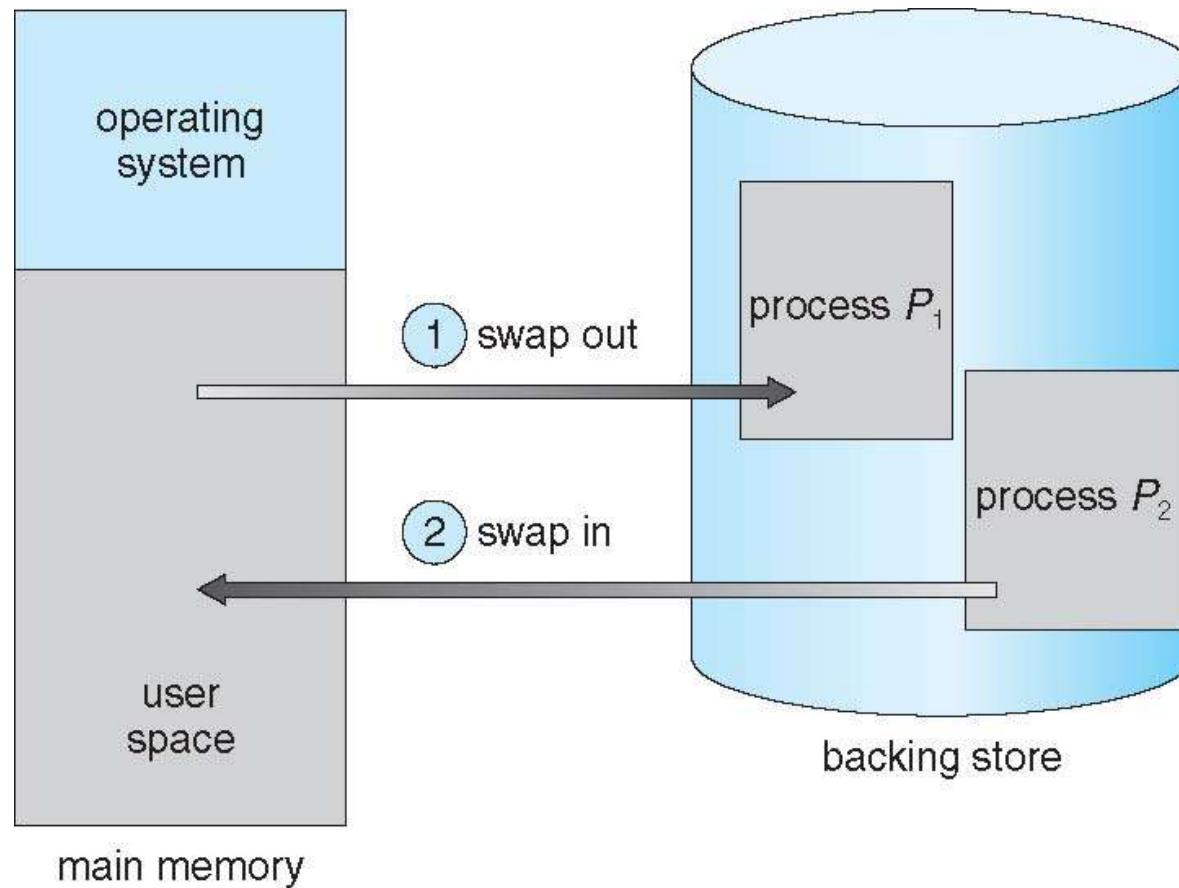
Tabella delle Pagine Invertite

- Minore memoria per le tabelle delle pagine, ma aumenta il tempo necessario per cercare la tabella quando c'è un riferimento ad una pagina
 - <process-id, page-number, offset>
- Hash table per limitare la ricerca ad una - o poche - entry della tabella
 - Due accessi in memoria (uno per accedere ad hash table)
 - TLB può accelerare l'accesso (prima di accedere ad hash table)
- Più complesso implementare la memoria condivisa
 - Un mapping solo di un indirizzo virtuale all'indirizzo fisico condiviso
 - Un riferimento da parte di altro processo genera il fallimento ed eventuale rimpiazzo del processo che insiste sul segmento

Swapping

- Un processo può essere scambiato (**swapped**) temporaneamente con memoria ausiliaria (backing store) e poi portato ancora nella memoria principale per continuare l'esecuzione
 - La memoria fisica totale occupata dai processi può eccedere la memoria fisica disponibile
- **Backing store** – memoria secondaria veloce grande abbastanza per accomodare i processi che devono essere stoccati e repringati
 - Deve fornire un accesso diretto a queste immagini in memoria
- **Swap out, swap in** – swapping di processi
 - Processi inattivi possono essere selezionati
 - La maggior parte del tempo di swap è di tempo di trasferimento; il tempo di trasferimento è direttamente proporzionale alla quantità di memoria trasferita (swapped)
 - Il sistema mantiene una **ready queue** di processi pronti per l'esecuzione che hanno immagine di memoria su disco
- **Roll out, roll in** – swapping con priorità:
 - processi di bassa priorità sono portate fuori (swapped out) così i processi a più alta priorità possono essere caricati ed eseguiti

Schema dello Swapping



Swapping

- Il processo portato fuori deve rientrare nello stesso indirizzo fisico?
 - Dipende dal metodo di binding
 - ▶ se binding a tempo di esecuzione anche indirizzo diverso
 - Considerando anche gli I/O pendenti da/a processi nello spazio di memoria
- Versioni modificate dello swapping nei diversi sistemi (es., UNIX, Linux, Windows)
 - Swapping di solito disabilitato
 - Si abilita se raggiunta una soglia di allocazione di memoria
 - Disabilitato ancora quando la richiesta di memoria va sotto soglia

Tempo di Context Switch incluso lo Swapping

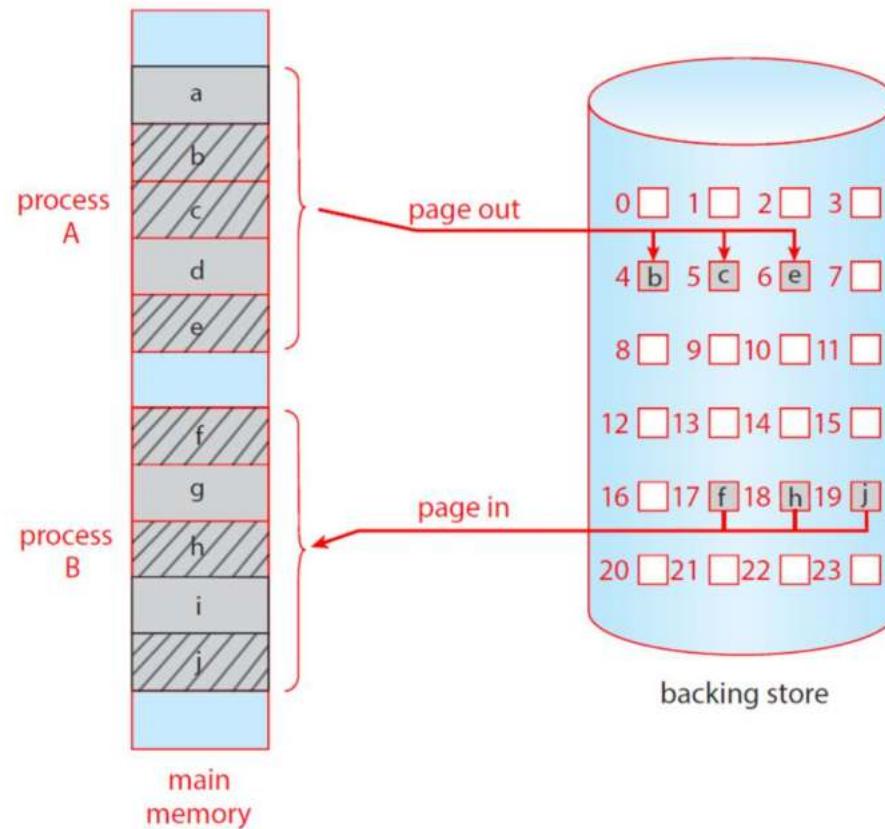
- Se il prossimo processo da mandare alla CPU non è in memoria si deve fare lo swap-out di un processo e lo swap-in del processo target
- Il tempo di context switch può essere molto alto
 - Un processo di 100MB che fa swapping su hard disk con un transfer rate di 50MB/sec
 - tempo di swap out di 2000 ms
 - ... più swap di un processo della stessa dimensione
 - tempo totale di context switch swapping è 4000ms (4 secondi)
- Si può ridurre se si riduce la dimensione della memoria swapped
 - Sapendo quanta memoria viene usata esattamente da un processo si può ridurre
 - ▶ System call per informare il SO dell'uso di memoria
 - ▶ `request_memory()` `release_memory()`

Tempo di Context Switch incluso lo Swapping

- Altri vincoli sullo swapping
 - I/O pendenti – non può fare swap out perché l'I/O sarebbe assegnato ai processi sbagliati
 - O sempre transferire I/O al kernel space, poi ai dispositivi I/O
 - ▶ Noto come **double buffering**, aggiunge overhead
- Lo standard swapping non è usato nei moderni SO
 - Una versione modificata è tipica
 - ▶ Swap solo quando la memoria libera è estremamente bassa
 - Viene fatto swapping con paging
 - ▶ Sottoinsieme di pagine page-in page-out

Tempo di Context Switch incluso lo Swapping

- Lo standard swapping non è usato nei moderni SO
 - Una versione modificata è tipica modified
 - Swap solo quando la memoria libera è estremamente bassa
 - Viene fatto swapping con paging
 - Sottoinsieme di pagine page-in page-out



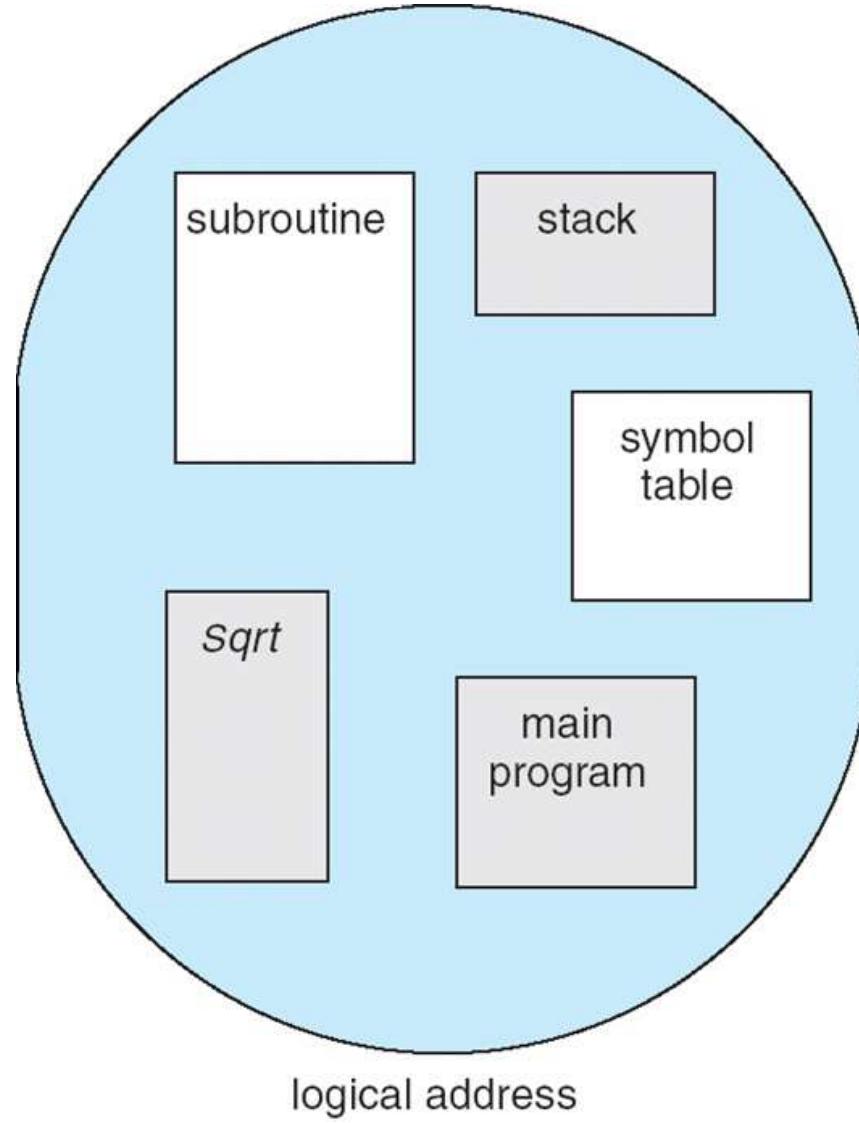
Swapping su Sistemi Mobile

- Tipicamente non supportata
 - Basta su flash memory
 - ▶ poco spazio
 - ▶ limitato numero of di cicli di scrittura
 - ▶ dialogo carente tra la flash memory e la CPU su piattaforma mobile
- Si usano altri metodi per liberare memoria se poca
 - iOS chiede alle app di lasciare volontariamente memoria
 - ▶ Dati read-only rilasciati e ricaricati dalla flash se necessario
 - ▶ Fallimento nel rilascio può portare alla terminazione
 - Android termina le app se poca memoria, prima salva lo stato sulla flash per un restart veloce
 - Entrambi i sistemi supportano il paging

Segmentazione

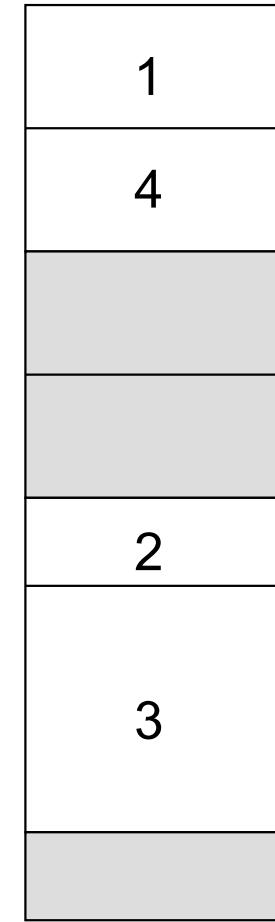
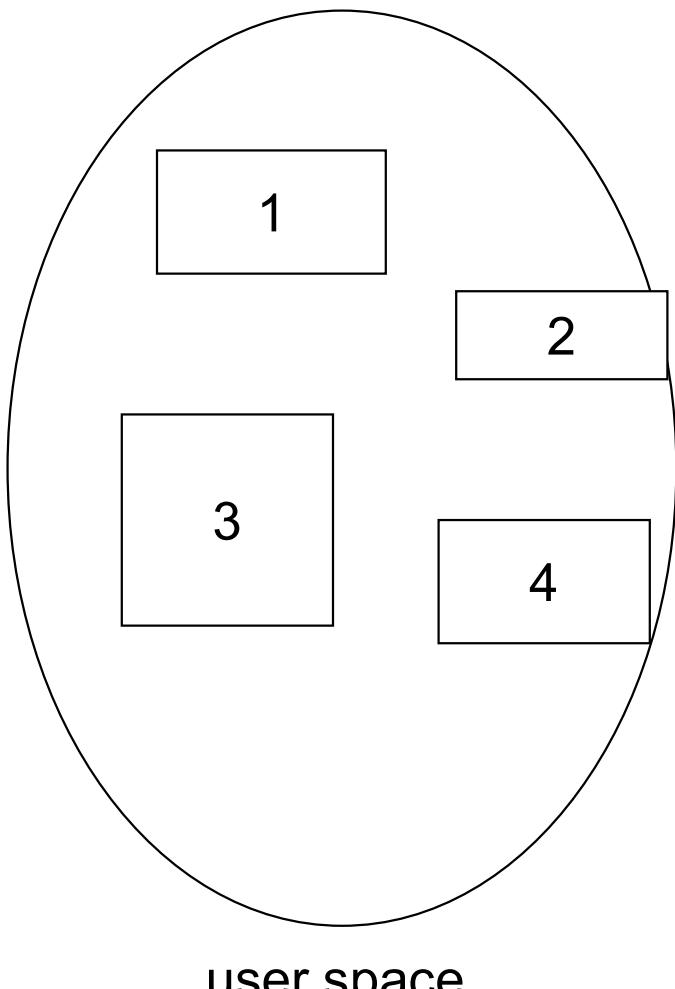
- La paginazione separa memoria logica e memoria fisica
- Segmentazione è uno schema di gestione della memoria che supporta la prospettiva utente sulla memoria
- Un programma è visto come una collezioni di segmenti
 - Un segmento è un'unità logica come:
 - main program
 - procedure
 - function
 - method
 - object
 - local variables, global variables
 - common block
 - stack
 - symbol table
 - arrays

Visione utente di un Programma



Visione Logica della Segmentazione

I segmenti sono numerati

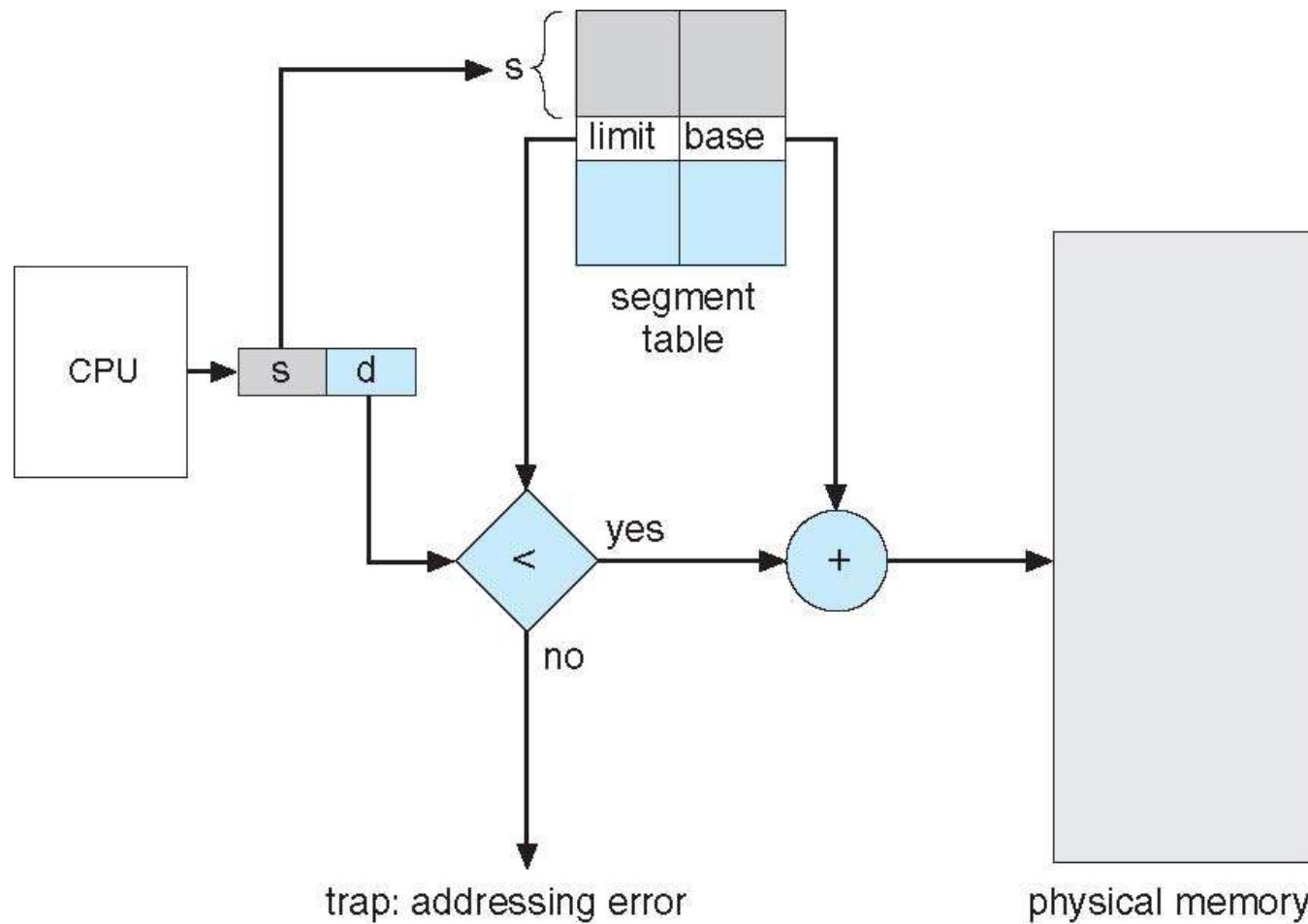


physical memory space

Architettura di Segmentazione

- Indirizzi logici sono definiti dalla tuple bidimensionale:
 $\langle \text{segment-number}, \text{offset} \rangle$,
- Serve una struttura che mappa le tuple in indirizzi fisici
- **Tabella dei Segmenti** – mappa indirizzi bidimensionali in fisici;
 ogni table entry ha:
 - **base** – contiene l'indirizzo fisico di partenza del segmento
 - **limite** – specifica la lunghezza del segmento

Hardware per Segmentazione

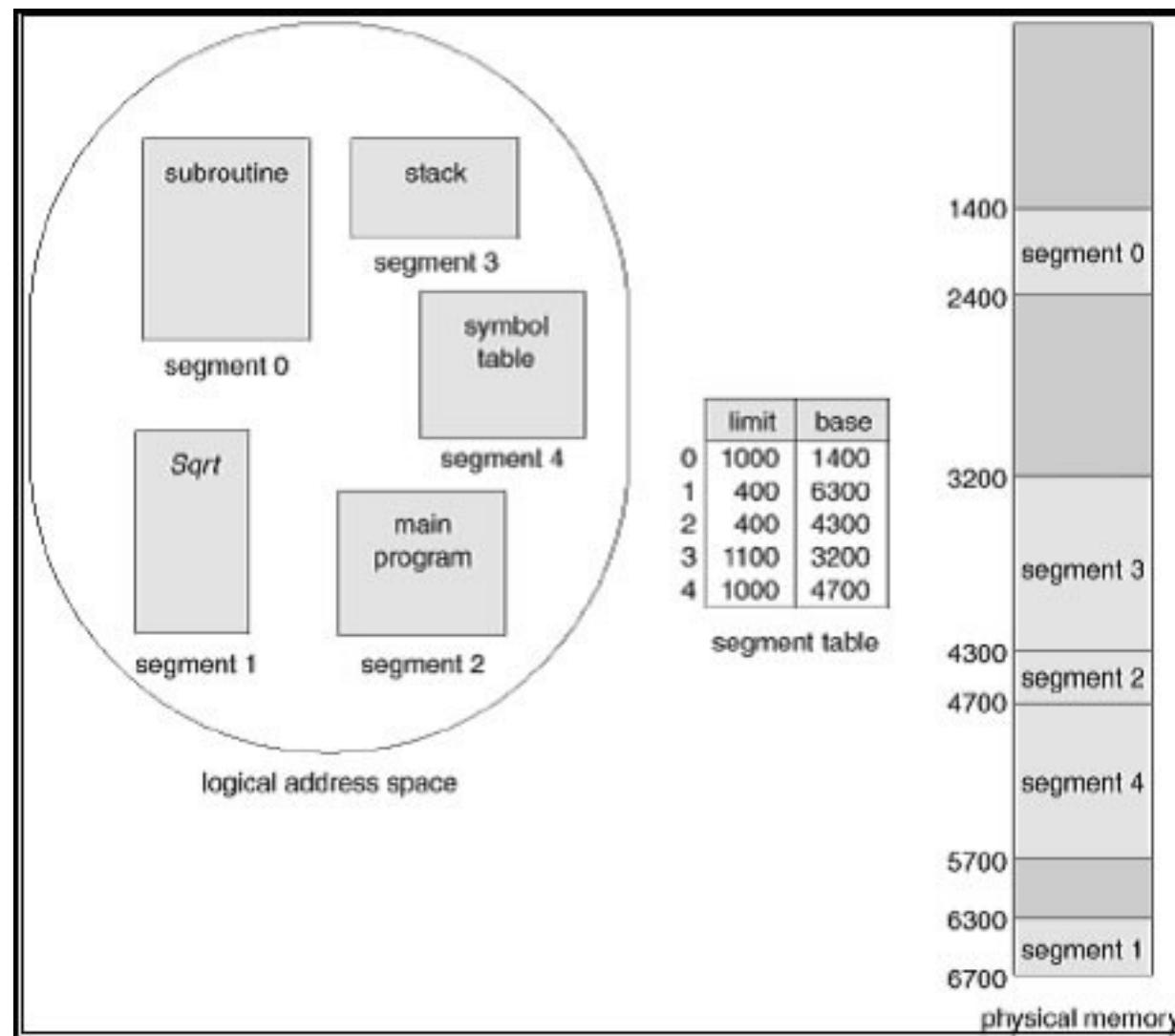


Architettura di Segmentazione

- Indirizzi logici sono definiti dalla tuple bidimensionale:
 $\langle \text{segment-number}, \text{offset} \rangle,$
- Serve una struttura che mappa le tuple in indirizzi fisici
- **Tabella dei Segmenti** – mappa indirizzi bidimensionali in fisici; ogni table entry ha:
 - **base** – contiene l'indirizzo fisico di partenza del segmento
 - **limite** – specifica la lunghezza del segmento
- La tabella dei segmenti non può essere tenuta in registri, quindi in memoria
 - **Segment-table base register (STBR)** indica la locazione di memoria della tabella dei segmenti
 - **Segment-table length register (STLR)** indica il numero dei segmenti usati da un program:
 - ▶ Dato indirizzo logico (s,d) , il numero s è legale se $s < \text{STLR}$
 - Registri associativi per limitare i due accessi in memoria

Architettura di Segmentazione

□ Esempio



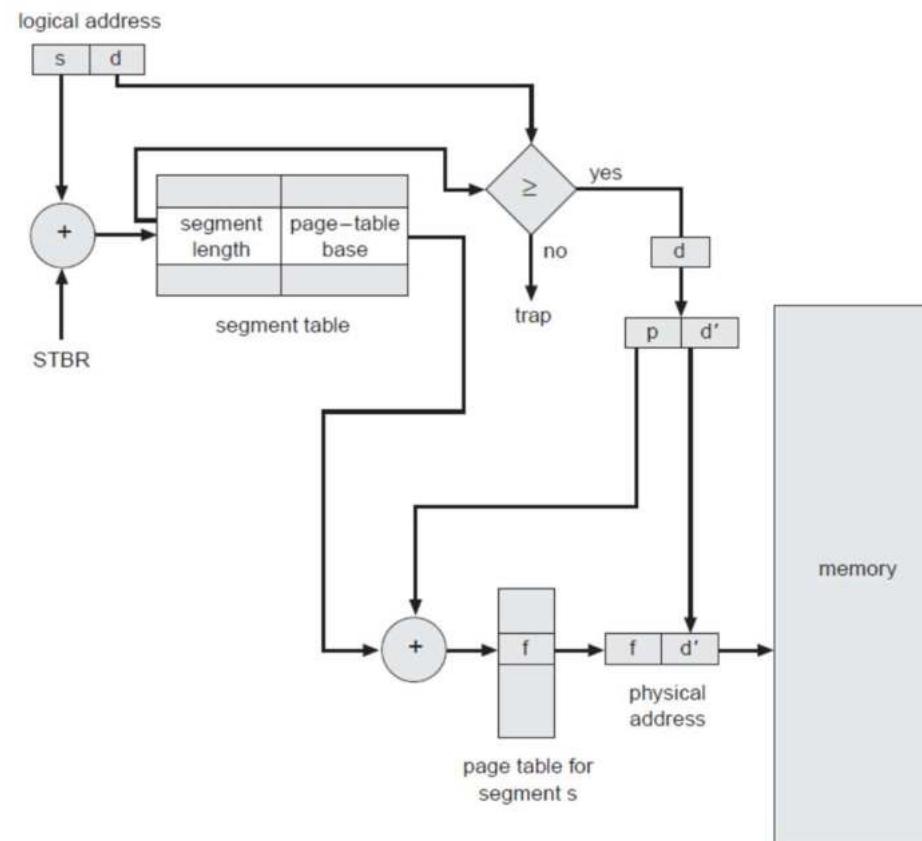
Architettura per Segmentazione

- Protezione (stesso segmento, stessa protezione)
 - Ogni entry nella tabella dei segmenti è associata a:
 - ▶ validation bit = 0 \Rightarrow segmento illegale
 - ▶ privilegi read/write/execute
 - Bit di protezione associati ai segmenti
- Possono essere condivisi i segmenti tra processi
 - La condivisione del codice avviene al livello di segmento
 - Come per il paging una copia di un programma condiviso (es. editor di testo)
 - Più programmi possono dover fare riferimento allo stesso numero di frammento
 - ▶ Porzioni di codice con riferimento diretto a sé stesse
- Problema del doppio accesso come per paginazione
- I segmenti di un processo, come per pagine, non necessariamente contigui
 - segmenti di dimensione diversa quindi problemi di frammentazione
 - si possono fondere i metodi di paginazione e segmentazione

Segmentazione Paginata

- Segmentazione paginate in GE 645 (MULTICS)

- Tabella dei segmenti:
 - lunghezza segmento e base della tabella
 - Tabella delle pagine per il segment s
 - $\text{base tab} + p = \text{base pagina}$

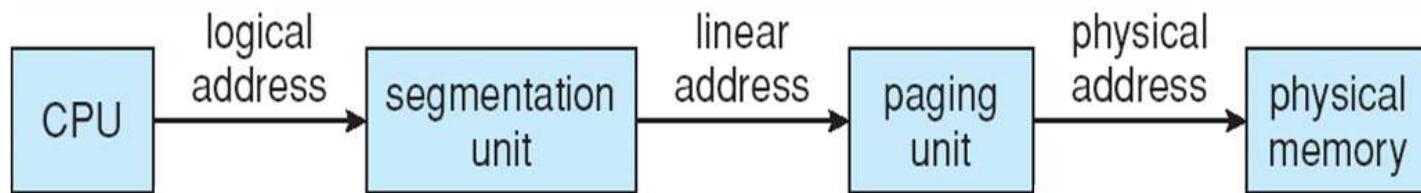


Esempi: Architetture Intel 32 e 64-bit

- Intel ha dominato l'industria per decadi
 - Negli anni 70 8086 a 16-bit seguita da 8088 (usato nel PC IBM)
 - Poi passata ad architetture a 32-bit (Architetture IA-32 – Intel Architecture)
 - Le CPU Pentium sono esempi di 32-bit
 - Ora le CPU Intel sono a 64-bit e sono chiamata architetture IA-64
- Molte varianti nei chip, si consideriamo solo i concetti principali

Esempio: l'Architettura Intel IA-32

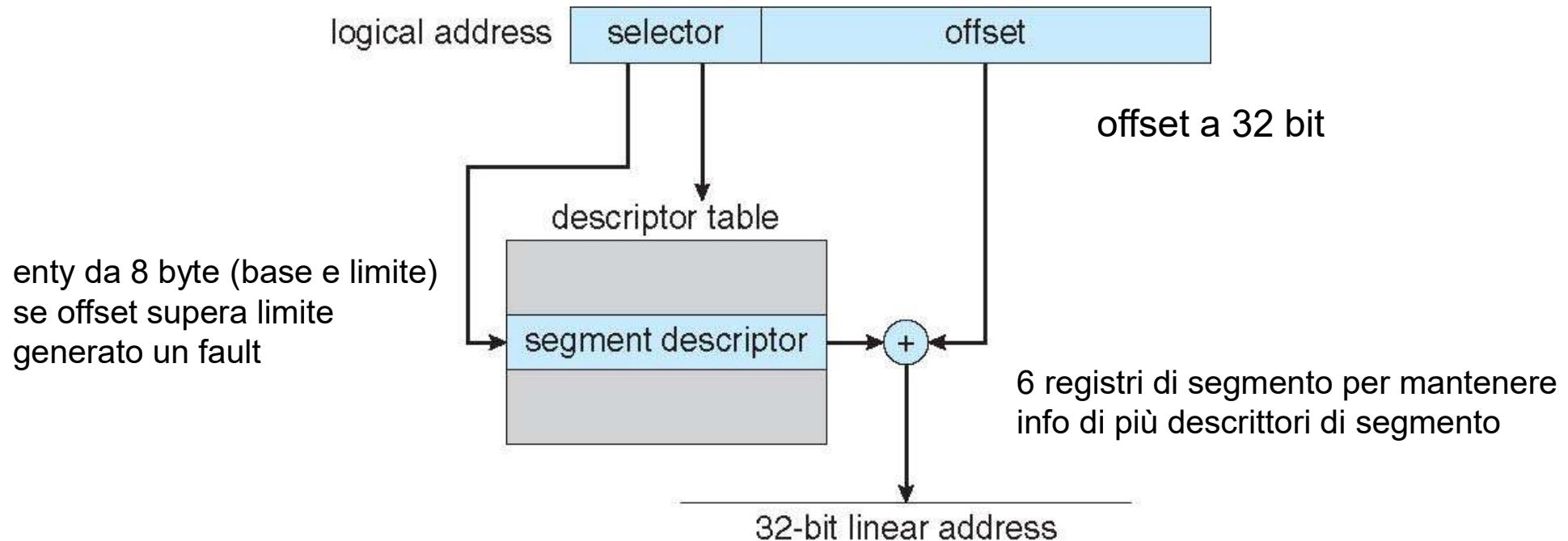
- Supporta sia segmentazione che segmentazione con paging
- La CPU genera indirizzi logici passati all'unità di segmentazione che produce **indirizzi lineari** poi passati all'unità di paging che produce l'indirizzo fisico (segmentation unit + page unit = MMU)



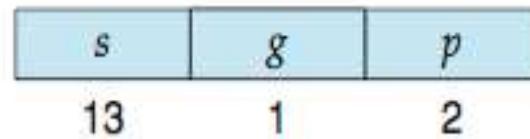
- Ogni segmento può essere di 4 GB
- Fino a 16 K segmenti per processo
 - Divisi in due partizioni
 - Prima partizione fino ad 8 K segmenti sono privati per il processo (descritti in una **local descriptor table (LDT)**)
 - Seconda partizione fino a 8K segmenti condivisi tra tutti i processi (descritti in una **global descriptor table (GDT)**)

Esempio: Architettura Intel IA-32

- La CPU genera indirizzi logici per l'unità di segmentazione



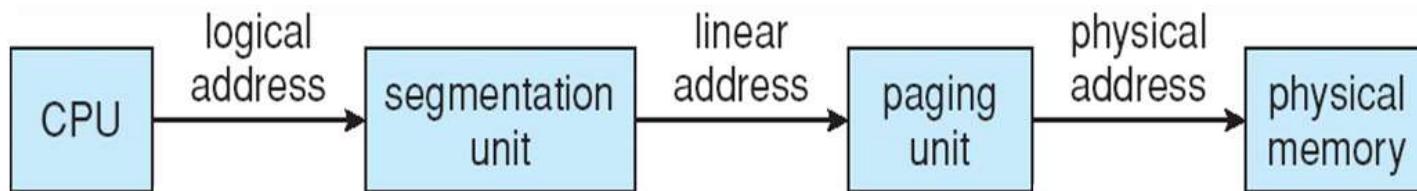
- Il selettore è a 16-bit



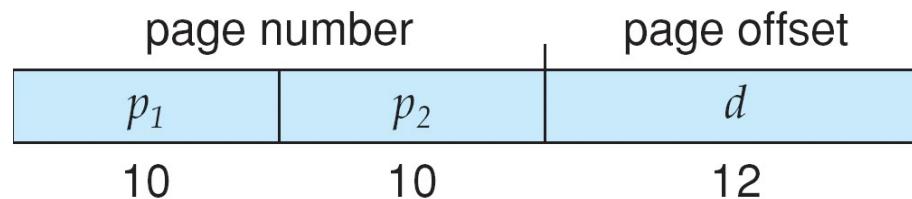
- s per il numero di segmento, g il tipo (global/local), p la protezione

Esempio: Architettura Intel IA-32

- Traduzione da indirizzo logico a fisico in IA-32

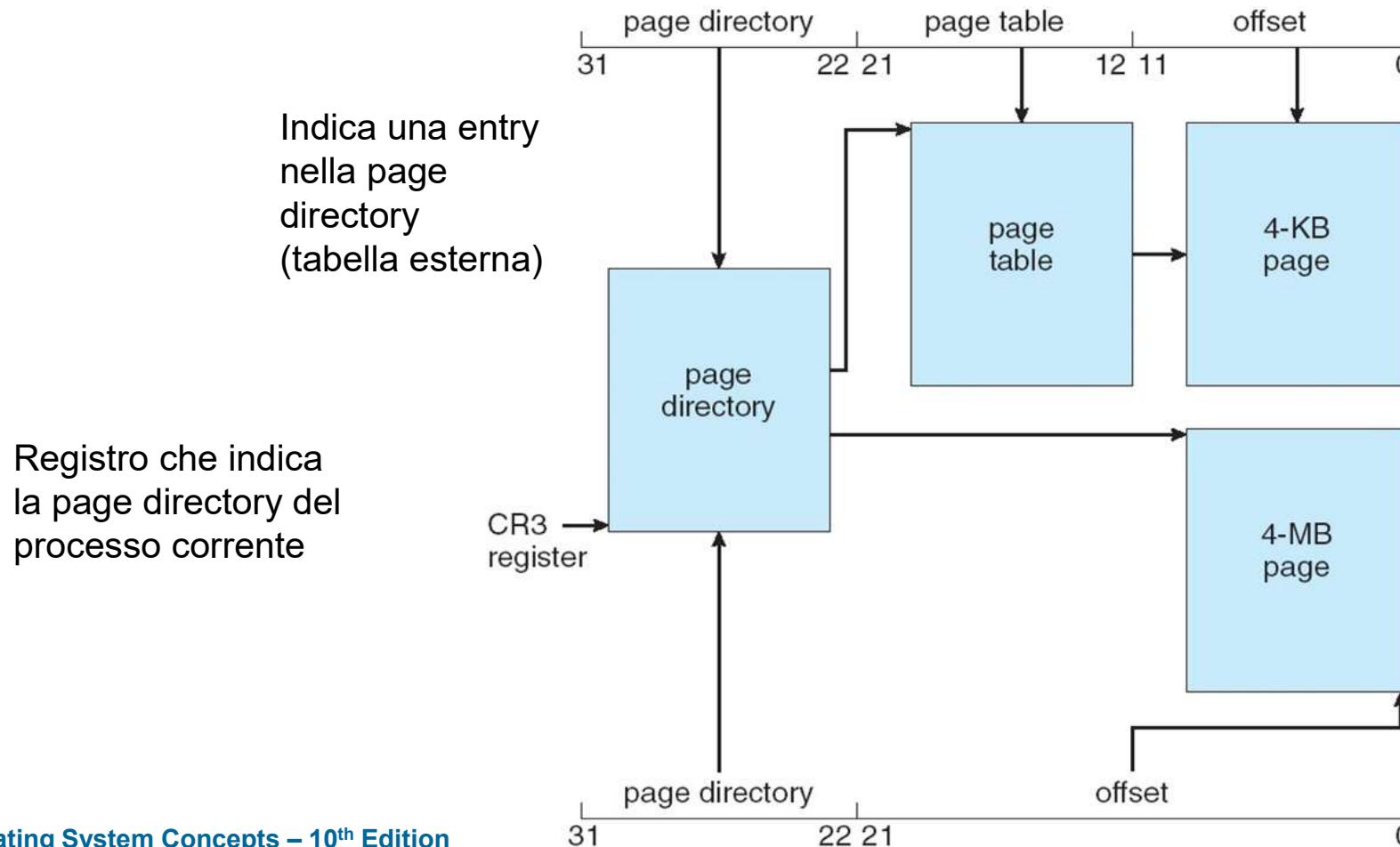


- Unità di paging di IA-32
- Dimensioni delle pagine 4 KB o 4 MB
 - Per 4 KB schema di paging a due livelli
 - Per 4 MB schema ad un livello
 - Indirizzo lineare 32 bit per paging a 2 livelli



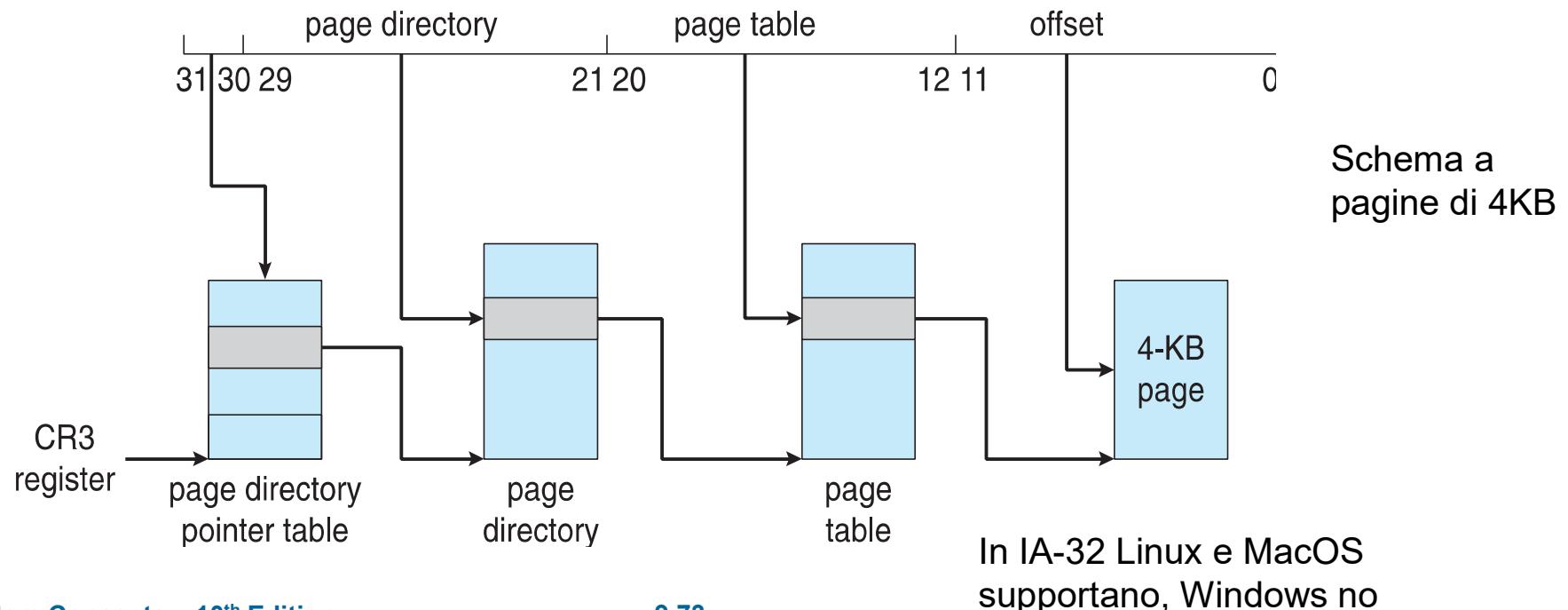
Esempio: Architettura Intel IA-32

- Unità di paging di IA-32
- Dimensioni delle pagine 4 KB o 4 MB
 - Per 4 KB schema di paging a due livelli
 - Per 4 MB schema ad un livello



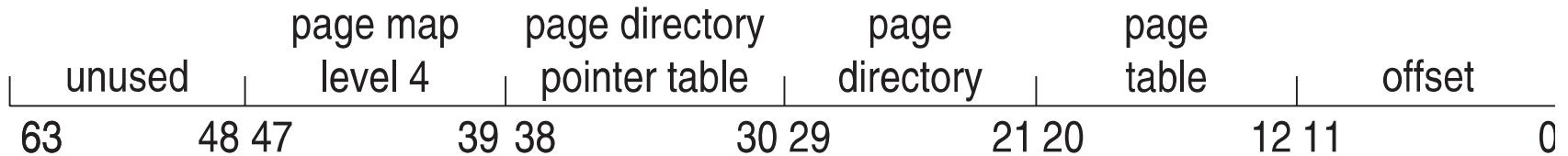
Intel IA-32 Estensioni Indirizzo Pagina

- Limiti di accesso di indirizzi 32-bit ha portato Intel a creare **page address extension (PAE)** permettendo ad app di 32-bit l'accesso a memorie di 4GB
 - Paging con schema a 3-livelli (per pagine a 4 KB)
 - Primi 2 bit si riferiscono ad una **page directory pointer table**
 - Le entry di page-directory e page-table aumentate a 64-bit
 - ▶ Base address da 20 a 24 bit + 12 bit per offset
 - ▶ indirizzi 36 bit – fino a 64GB di memoria fisica



Intel x86-64

- Generazione corrente è architettura x86-64
- x86-64 proposto da AMD e adottato da Intel (AMD64 e Intel64)
- 64 bit consentirebbero dimensioni enormi ($2^{64} > 16$ exabyte)
- In pratica è implementato un indirizzamento a 48 bit
 - Dimensione di pagina 4 KB, 2 MB, 1 GB
 - Quattro livelli di gerarchia di paging



Anche se 48-bit virtuali con page address extension fino a 52-bit indirizzi fisici

Esempio: Architettura ARM

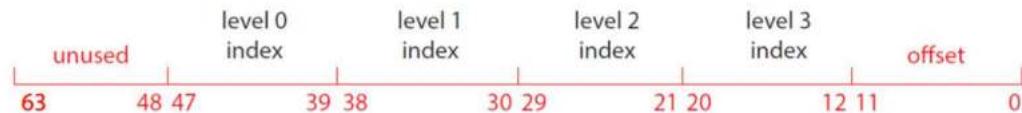
- Dominante per i chip su piattaforma mobile (es. Apple iOS e Google Android, ma anche sistemi embedded real-time)
- CPU moderna ed efficiente energeticamente
- ARMv8 è a 64-bit

Esempio: Architettura ARMv8

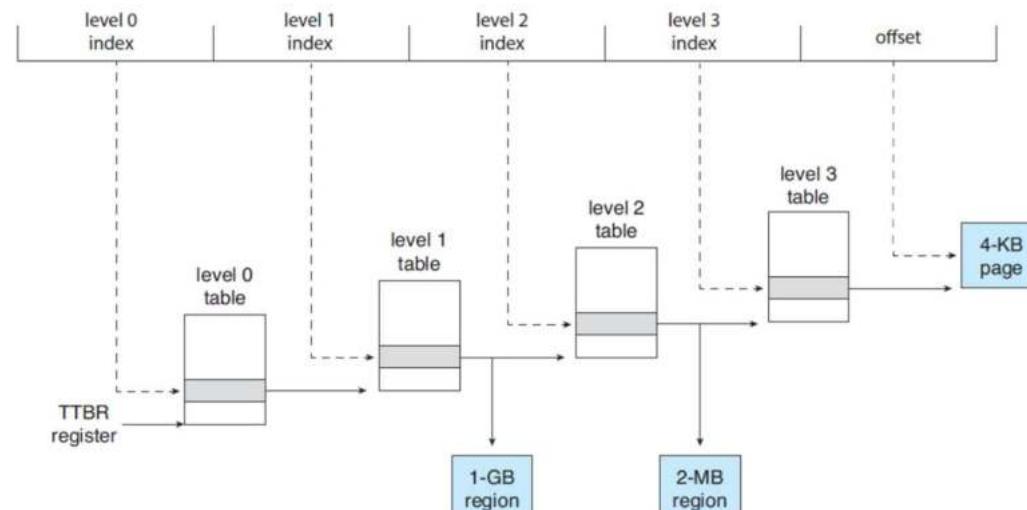
- ARMv8 è a 64-bit
- Tre granularità di traduzione 4 KB, 16 KB, 64 KB associate a dimensione di pagina e sezioni di memoria contigua (regions)

| Translation Granule Size | Page Size | Region Size |
|--------------------------|-----------|-------------|
| 4 KB | 4 KB | 2 MB, 1 GB |
| 16 KB | 16 KB | 32 MB |
| 64 KB | 64 KB | 512 MB |

- Usati fino a 48 bit



- Per 4 KB e 16 KB fino a 4 livelli di paging, fino a 3 per 64 KB

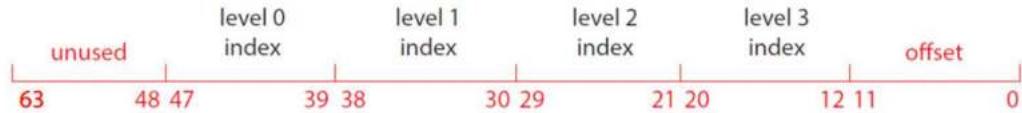


Esempio: Architettura ARMv8

- ARMv8 è a 64-bit
- Tre granularità di traduzione 4 KB, 16 KB, 64 KB associate a dimensione di pagina e sezioni di memoria contigua (regions)

| Translation Granule Size | Page Size | Region Size |
|--------------------------|-----------|-------------|
| 4 KB | 4 KB | 2 MB, 1 GB |
| 16 KB | 16 KB | 32 MB |
| 64 KB | 64 KB | 512 MB |

- Usati fino a 48 bit



- Per 4 KB e 16 KB fino a 4 livelli di paging, fino a 3 per 64 KB
- I livelli 1 e 2 possono essere anche usati per indirizzare regioni da 1-GB (livello 1) e 2-MB (livello 2)
 - Se livello 1 – 30 bit di offset, se livello 2 – 20 bit di offset
 - Due livelli di TLBs
 - ▶ Livelli esterni hanno due micro TLBs (una dati, una istruzioni)
 - ▶ La più interna è una singola TLB
 - ▶ Prima si cerca su interna, sul miss si cerca su esterne, sul miss la CPU fa il page table walk

Memoria Virtuale

Obiettivi

- Descrivere i benefici di un sistema a memoria virtuale
- Spiegare i concetti di demand paging, sostituzione di pagina e l'allocazione di page frames
- Discutere il principio del modello working-set
- Esaminare la relazione tra shared memory e memory-mapped files
- Esaminare come è gestita la memoria kernel

Background

- Il codice deve essere in memoria per essere eseguito, ma non tutto è sempre necessario
 - Codice di errore, routine non usate, strutture dati grandi
- Non necessario tutto il codice del programma nello stesso tempo
- Se possibile eseguire programmi parzialmente caricati ...
 - Programmi non più vincolati dalla memoria fisica
 - Ogni programma occupa meno memoria a run-time -> più programmi eseguiti allo stesso tempo
 - ▶ Miglior utilizzo di CPU e maggior throughput senza incremento in tempi di risposta o tempo di turnaround
 - Meno I/O necessari per caricare o swappare programmi in memoria -> ogni programma utente più veloce

Background

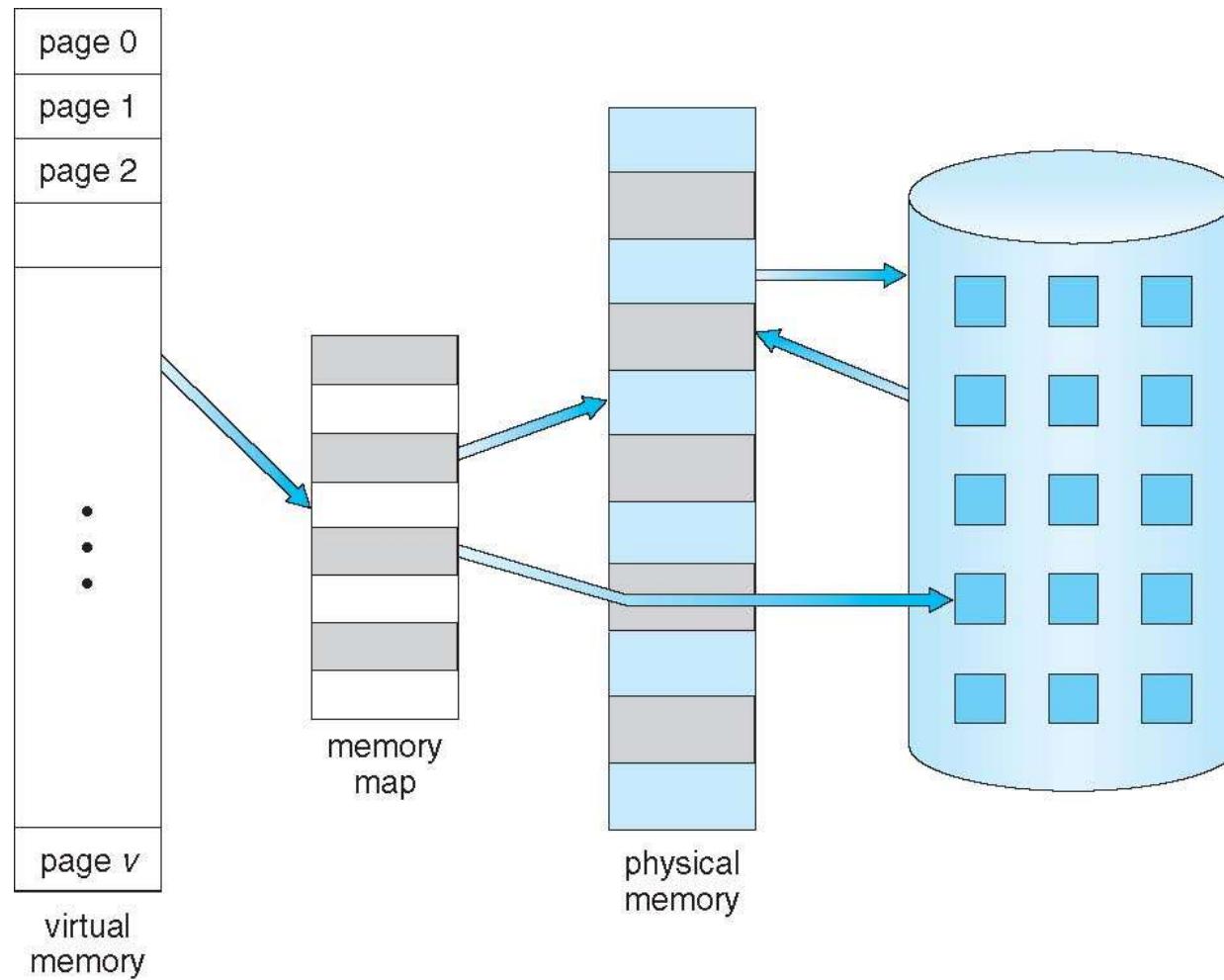
- **Memoria Virtuale** – separazione della memoria logica dalla memoria fisica
 - Solo una parte del programma in memoria per l'esecuzione
 - Spazio di indirizzi logici può essere più grande dello spazio di indirizzi fisici
 - Spazio indirizzi condiviso da molti processi
 - Creazione di processi più efficiente
 - Più programmi eseguiti in concorrenza
 - Meno I/O necessario per caricare o swappare processi

Background

- **Spazio degli indirizzi virtuali** – vista logica su come il processo è contenuto in memoria
 - Spazio inizia da un indirizzo (es. indirizzo 0)
 - Indirizzi contigui fino alla fine dello spazio
 - Memoria fisica organizzata in page frame
 - MMU deve mappare indirizzo logico su fisico
- Memoria Virtuale implementata con:
 - Demand paging
 - Demand segmentation

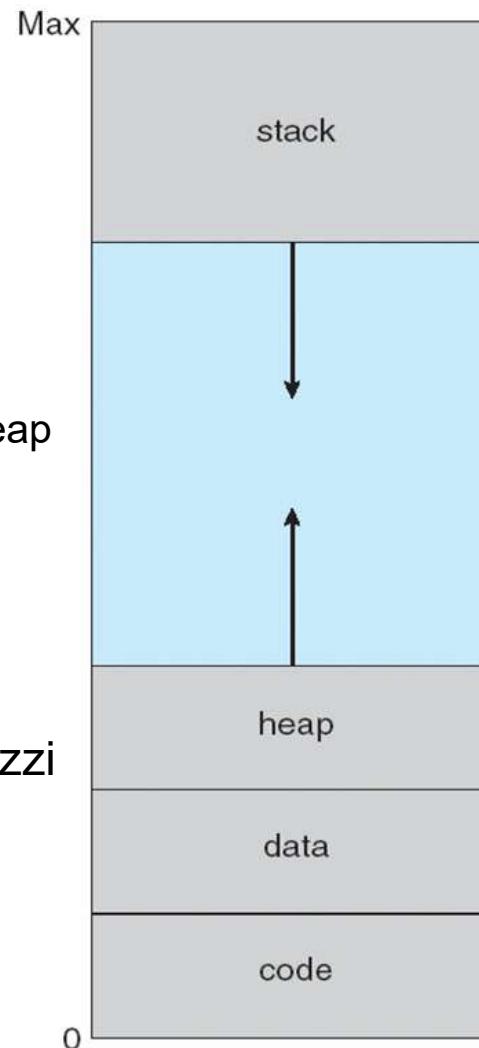
Memoria Virtuale più grande della memoria fisica

Schema che mostra memoria virtuale più grande di quella fisica



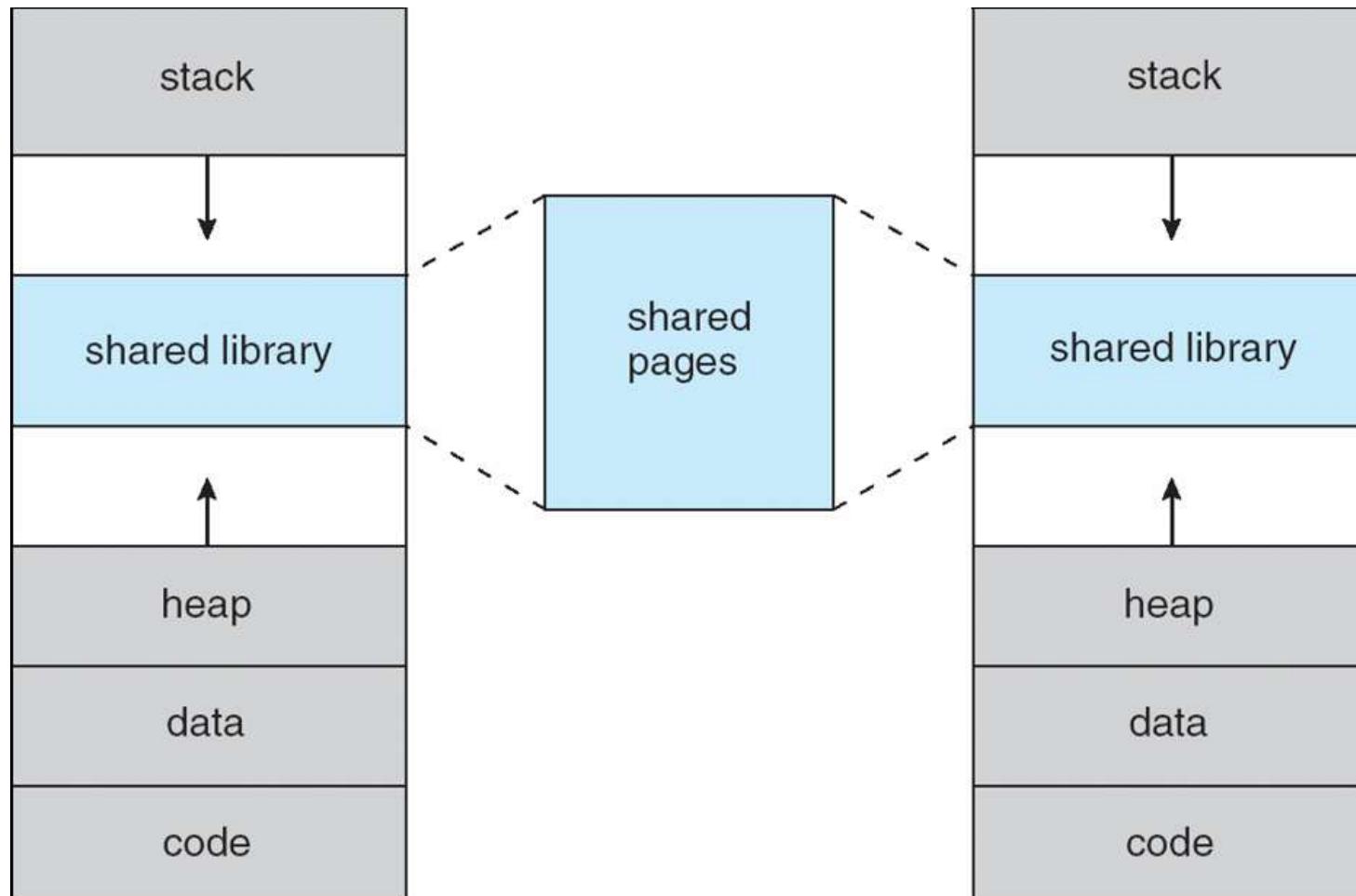
Spazio degli indirizzi virtuali

- Di solito spazio di indirizzi logici con:
 - Stack inizia da indirizzo logico massimo e cresce “in giù”
 - Heap cresce “in su”
 - Massimizza l’uso dello spazio di indirizzi
 - Spazio di indirizzi non usato è un buco
 - memoria fisica non necessaria finchè lo heap o stack non cresce fino a nuova pagina
- Spazio di indirizzi **sparso** con buchi lasciati per crescita, dynamically linked libraries, etc
- Librerie di sistema condivise nello spazio di indirizzi virtuali
- Memoria condivisa mappando pagine read-write nello spazio degli indirizzi virtuali
- Le pagine possono essere condivise durante il `fork()` per accelerare la creazione del processo



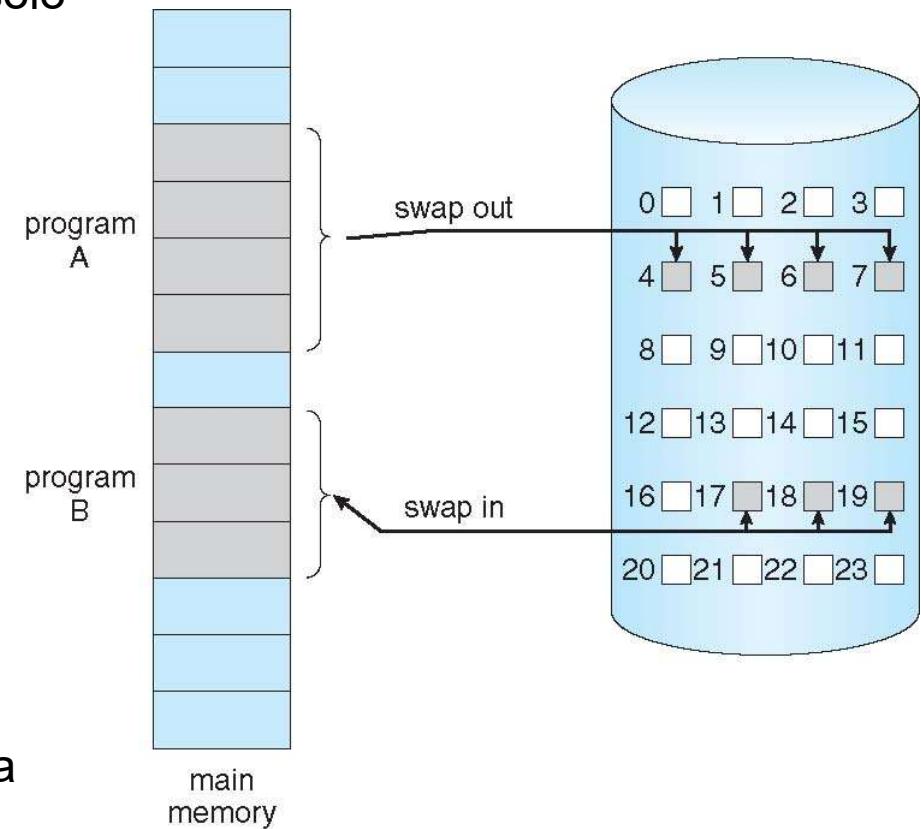
Shared Library usando Memoria Virtuale

- Librerie di sistema condivise nello spazio di indirizzi virtuali
- Shared memory mappando pagine read-write nel virtual address space
- Pagine possono essere condivise durante il fork() per accelerare la creazione del processo



Demand Paging

- Si può portare l'intero processo in memoria a tempo di caricamento
- Oppure portare una pagina in memoria solo quando è necessario:
 - Meno I/O, elimina il non necessario
 - Minor memoria richiesta
 - Risposta più rapida
 - Più utenti contemporaneamente
- Simile al paging con swapping
- Pagina richiesta \Rightarrow riferimento ad essa
 - Riferimento non valido \Rightarrow abort
 - Non-in-memoria \Rightarrow portala in memoria
- **Lazy swapper** – non swappa una pagina in memoria se non richiesta
 - Swapper che carica singole pagine è un **pager**



Concetti di Base

- Il pager anticipa quali pagine verranno usate prima di essere di nuovo portate fuori dalla memoria fisica
- Non tutto il processo caricato, il pager porta in memoria solo le pagine richieste
- Come determinare questo insieme di pagine?
 - Occorre una nuova funzionalità MMU per implementare il demand paging
- Se la pagina richiesta è già **residenti in memoria**
 - Non c'è differenza con normale esecuzione
- Se la pagina richiesta non è residente in memoria
 - Si genera un trap di page fault
 - Si deve trovare e caricare la pagina in memoria dallo storage
 - senza cambiare il comportamento del programma
 - senza che il programmatore debba cambiare il codice

Valid-Invalid Bit

- Ogni entry della page table ha associato un bit valid–invalid (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Inizialmente il valid–invalid bit è settato a **i** su tutte le entry
- Es. di uno stato di page table:

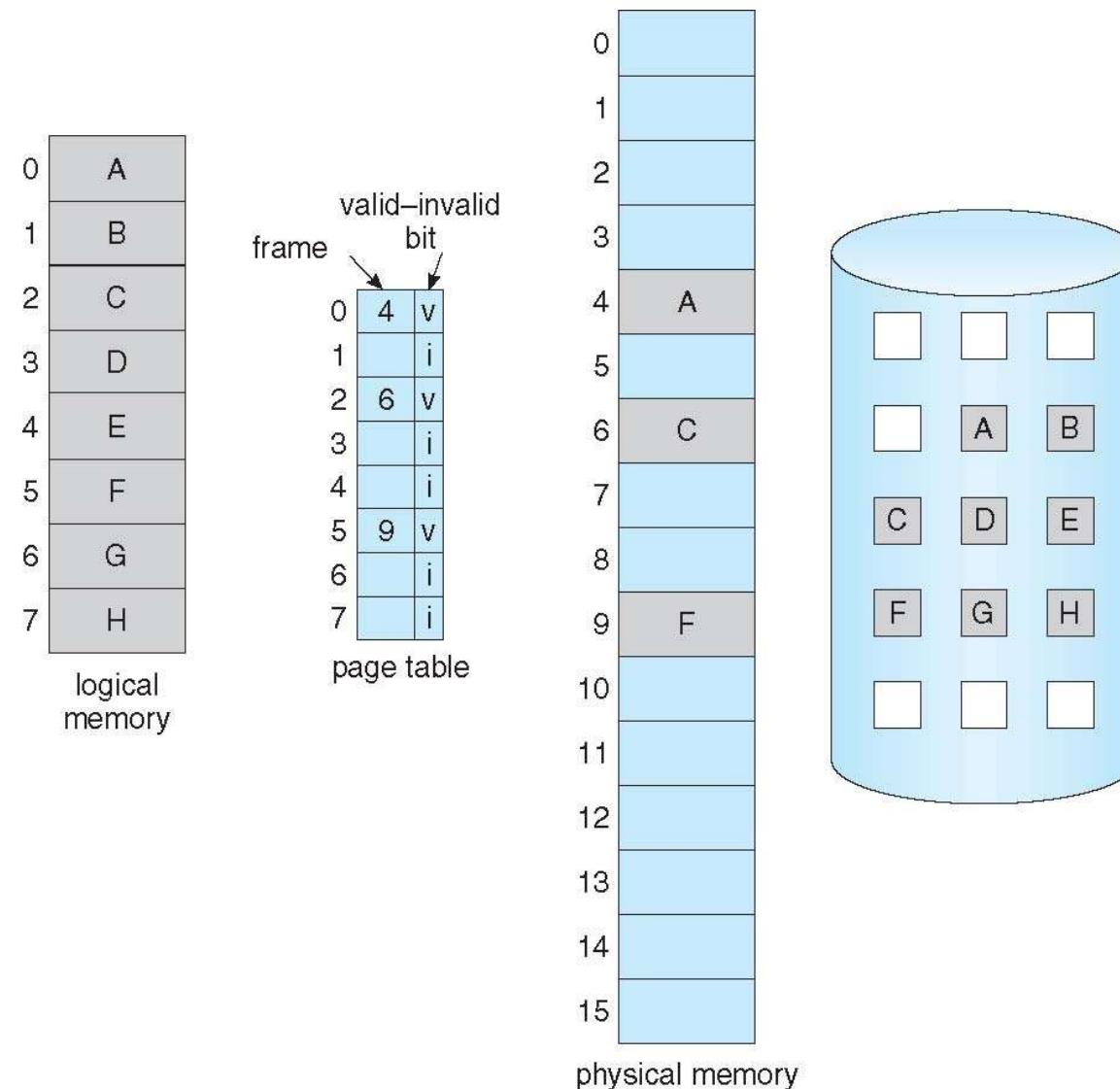
| Frame # | valid-invalid bit |
|---------|-------------------|
| | v |
| | v |
| | v |
| | i |
| ... | |
| | i |
| | i |

page table

- Se invalid o il frame non è in memoria o accesso vietato
- Durante la traduzione della MMU se il valid–invalid bit nella page table entry è **i** ⇒ page fault

Page Table quando alcune pagine non sono in Memoria

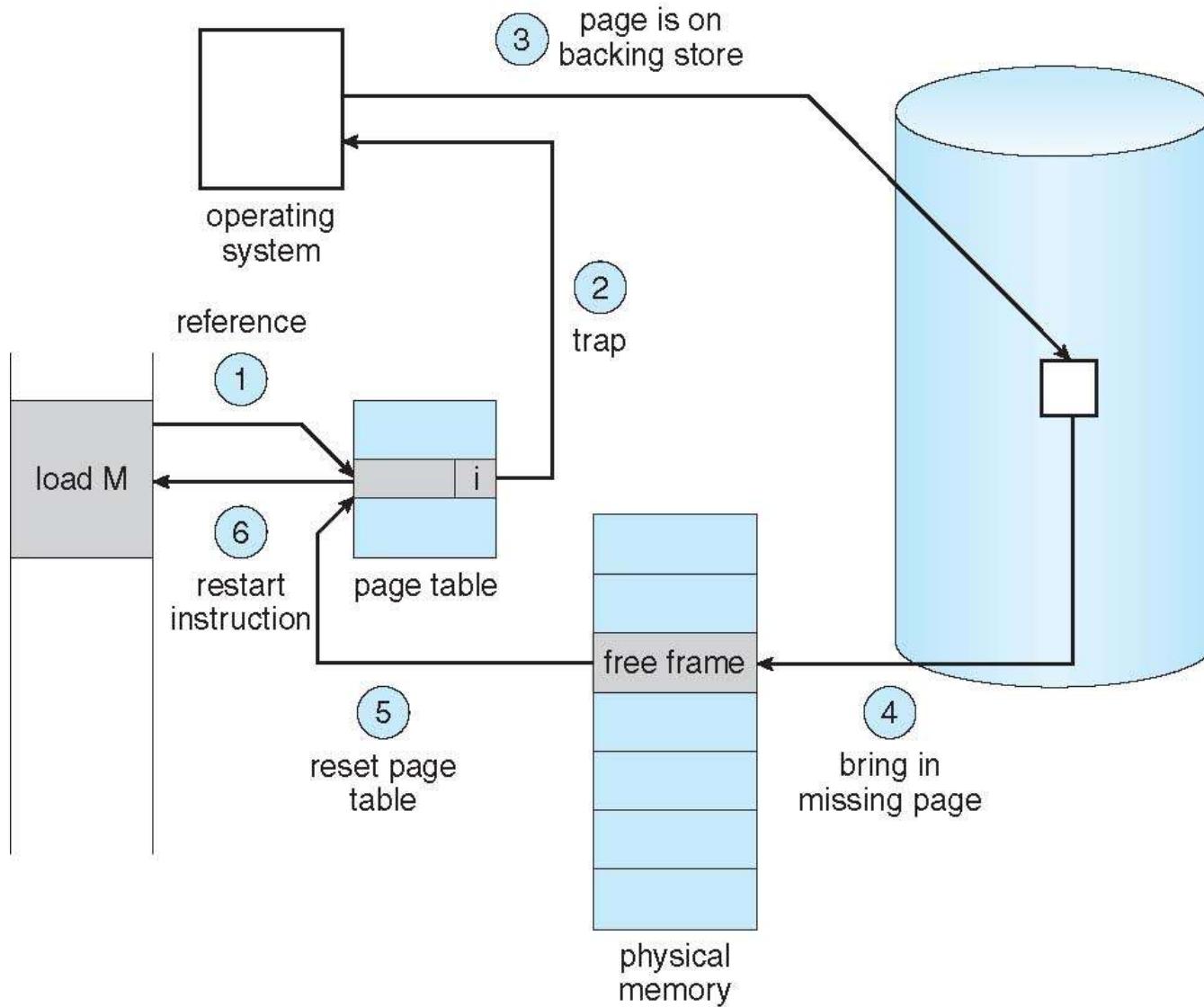
Es. 0 è valido, 1 non è valido, etc.



Page Fault

- Il primo riferimento ad una pagina non in memoria genera un trap all'OS:
 - Trap di page fault
- Page fault gestito come segue:
 1. OS cerca su altra tabella (in PCB) per verificare se:
 - Riferimento non valido \Rightarrow abort
 - Semplicemente non in memoria
 2. Se non valido termina, altrimenti procedi con l'inserimento della pagina
 3. Trova un frame libero (cercando su lista dei frame liberi)
 4. Schedula operazione su disco per lettura della pagina e scrittura nel frame libero
 5. A lettura completata resetta le tabelle per indicare che la pagina è in memoria; setta il validation bit = **V**
 6. Riavvia l'instruzione che ha causato il page fault perché a questo punto il processo può accedere alla pagina in memoria

Passi per Gestire il Page Fault

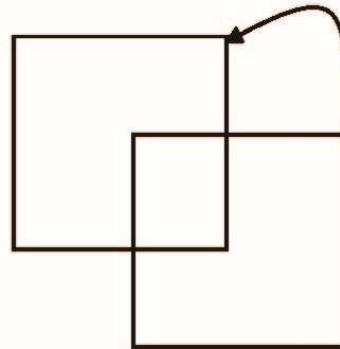


Aspetti del Demand Paging

- Caso estremo – il processo inizia senza pagine in memoria
 - SO setta l'instruction pointer sulla prima istruzione e la memoria non è residente -> page fault
 - ... lo stesso per ogni altro processo al primo accesso
 - ▶ **Pure demand paging**
- In realtà un'istruzione può accedere più pagine (una per l'istruzione e tante per i dati) -> multipli page fault
 - Es. fetch e decode di istruzione di somma di 2 numeri da memoria che registra il risultato in memoria
 - Problema mitigato dalla **località del riferimento**
- Supporto hardware per demand paging
 - Page table con valid / invalid bit
 - Memoria secondaria (dispositivo di swap con **swap space**)
 - Restart dell'istruzione

Instruction Restart

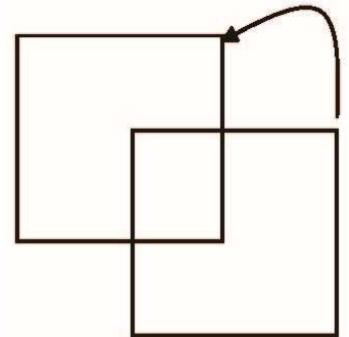
- È necessario il restart delle istruzioni dopo il page fault
- Fault può avvenire in qualunque momento:
 - fetch, decode, exec
 - Se interrotta durante il prelievo di operando, va rifatta dal fetch
- Considera un'istruzione che può accedere a molte locazioni diverse
 - Si muove tutto un blocco da una locazione ad un'altra



- auto increment/decrement della locazione
- Restart dell'intera operazione?
 - ▶ Che succede se sorgente e destinazione si sovrappongono?

Instruction Restart

- È necessario il restart delle istruzioni dopo il page fault
- Fault può avvenire in qualunque momento:
 - fetch, decode, exec
 - Se interrotta durante il prelievo di operando, va rifatta dal fetch
 - Si consideri operazione ADD A, B con risultato in C
 - ▶ Prelievo e decodifica di ADD
 - ▶ Prelievo di A
 - ▶ Prelievo di B
 - ▶ Addizione A e B
 - ▶ Memorizzazione somma in C
 - Se interrotta su C, va ripetuta l'operazione dal prelievo
 - Istruzioni ancora più complesse:
 - ▶ IBM System 360/370 MVC può spostare 256 byte da locazione ad un'altra anche sovrapposte
 - ▶ se il blocco di dati è tra due pagine e page fault?
 - ▶ Blocco origine modificato, non si può fare restart, due soluzioni:
 - Si controlla prima dell'operazione caricando tutte le pagine
 - Si mantiene in registri lo stato precedente e si ripristina



Instruction Restart

- È necessario il restart delle istruzioni dopo il page fault
- Fault può avvenire in qualunque momento:
 - fetch, decode, exec
 - Se interrotta durante il prelievo di operando, va rifatta dal fetch
 - Si consideri operazione ADD A, B con risultato in C
 - Se interrotta su C, va ripetuta l'operazione dal prelievo
 - Istruzioni ancora più complesse:
 - ▶ IBM System 360/370 MVC può spostare 256 byte da locazione ad un'altra anche sovrapposte
 - ▶ se il blocco di dati è tra due pagine e page fault?
 - ▶ Blocco origine modificato, non si può fare restart, due soluzioni:
 - Si controlla prima dell'operazione caricando tutte le pagine
 - Si mantiene in registri lo stato precedente e si ripristina
 - Autoincremento/ autodecremento
 - ▶ Registri autoincrementati/decrementati prima/dopo utilizzo...
 - ▶ ... anche in questo caso, se interrotta l'operazione occorre il ripristino
 - Il paging con demand page richiede un'architettura adatta

Lista dei frame liberi

□ Fame Free List

- Quando c'è un page fault occorre caricare la pagina da memoria secondaria in un frame libero
- Il Sistema mantiene una lista di frame liberi



- I frame sono azzerati prima dell'allocazione (zero-fill-on-demand)

Performance del Demand Paging

- Passi del Demand Paging (caso peggiore)
 1. Trap al sistema operativo
 2. Salva gli user register e il process state
 3. Determina se l'interrupt era un page fault
 4. Verifica che il riferimento alla pagina era legale e determina la locazione della pagina su disco
 5. Schedula una lettura dal disco ad un frame libero:
 1. Attendi in una coda per il dispositivo finché la richiesta di read è servita
 2. Attendi la ricerca e/o il tempo di latenza
 3. Inizia il trasferimento della pagina al frame libero
 6. Mentre attendi alloca la CPU a qualche altro user
 7. Ricevi un interrupt dal I/O (I/O completato)
 8. Salva i registeri e lo stato del processo dell'altro user
 9. Determina che l'interrupt era dal disco
 10. Correggi la page table e le altre tabelle per mostrare la pagina in memoria
 11. Attendi che la CPU venga di nuovo allocata a questo processo
 12. Ripristina gli user register, il process state e la nuova page table, quindi riprendi l'istruzione interrotta

Performance del Demand Paging

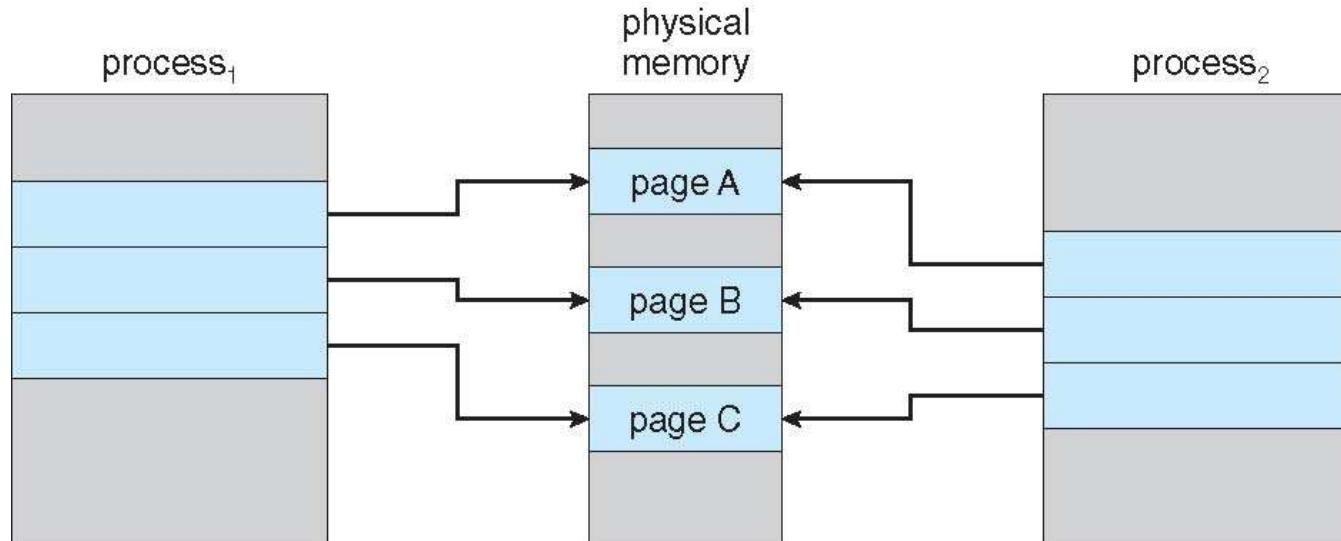
- Non sempre tutte le operazioni precedenti sono richieste
- In ogni caso sono richieste tre principali attività:
 - Servi l'interrupt – ben codificato, necessarie centinaia di istruzioni
 - Leggi la pagina – molto tempo
 - Restart del processo – poco tempo
- **Page Fault Rate** $0 \leq p \leq 1$
 - se $p = 0$ non page fault
 - se $p = 1$, ogni riferimento è un fault
- **Effective Access Time (EAT)**
$$\text{EAT} = (1 - p) \times \text{memory access}$$
$$+ p \text{ (page fault time)}$$

Esempio Demand Paging

- Tempo accesso memoria = 200 nanosecondi
- Tempo medio di servizio del page-fault = 8 millisecondi
- $$\begin{aligned} \text{EAT} &= (1 - p) \times 200 + p (8 \text{ millisecondi}) \\ &= (1 - p) \times 200 + p \times 8,000,000 \\ &= 200 + p \times 7,999,800 \end{aligned}$$
- Se un accesso su 1000 causa un page fault allora
 $\text{EAT} = 8.2 \text{ microsecondi}$
È un slowdown di un fattore di 40!
- Se si vuole una performance degradation < 10 percent
 - $$\begin{aligned} 220 &> 200 + 7,999,800 \times p \\ 20 &> 7,999,800 \times p \end{aligned}$$
 - $p < .0000025$
 - < una page fault per ogni 400,000 accessi in memoria

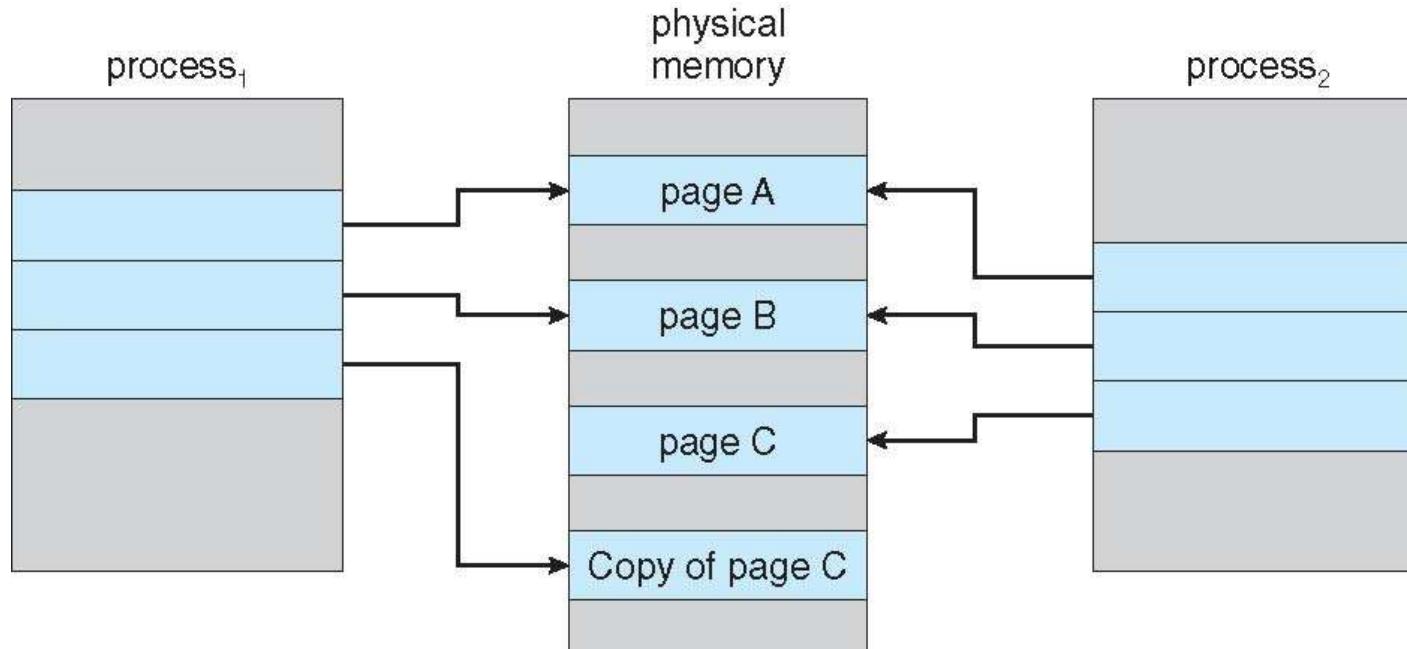
Copy-on-Write

- La call `fork()` crea duplicato del padre (se il figlio fa subito `exec()` non è necessario il duplicato di memoria)
- **Copy-on-Write** (COW) permette ai processi padre e figlio di **condividere** inizialmente le stesse pagine in memoria
 - Le pagine condivise sono marcate come pagine copy-on-write
 - Se uno dei processi modifica una pagina condivisa allora questa viene copiata
 - COW permette una creazione di processo più efficiente dal momento che solo le pagine modificate sono copiate
 - Tecnica usata su Linux, Windows e macOS



Copy-on-Write

- La call `fork()` crea duplicato del padre, se il figlio fa subito `exec()` non è necessario il duplicato di memoria
- **Copy-on-Write** (COW) permette ai processi padre e figlio di **condividere** inizialmente le stesse pagine in memoria
 - Le pagine condivise sono marcate come pagine copy-on-write
 - Se uno dei processi modifica una pagina condivisa allora questa viene copiata
 - COW permette una creazione di processo più efficiente dal momento che solo le pagine modificate sono copiate
 - Tecnica usata su Linux, Windows e macOS

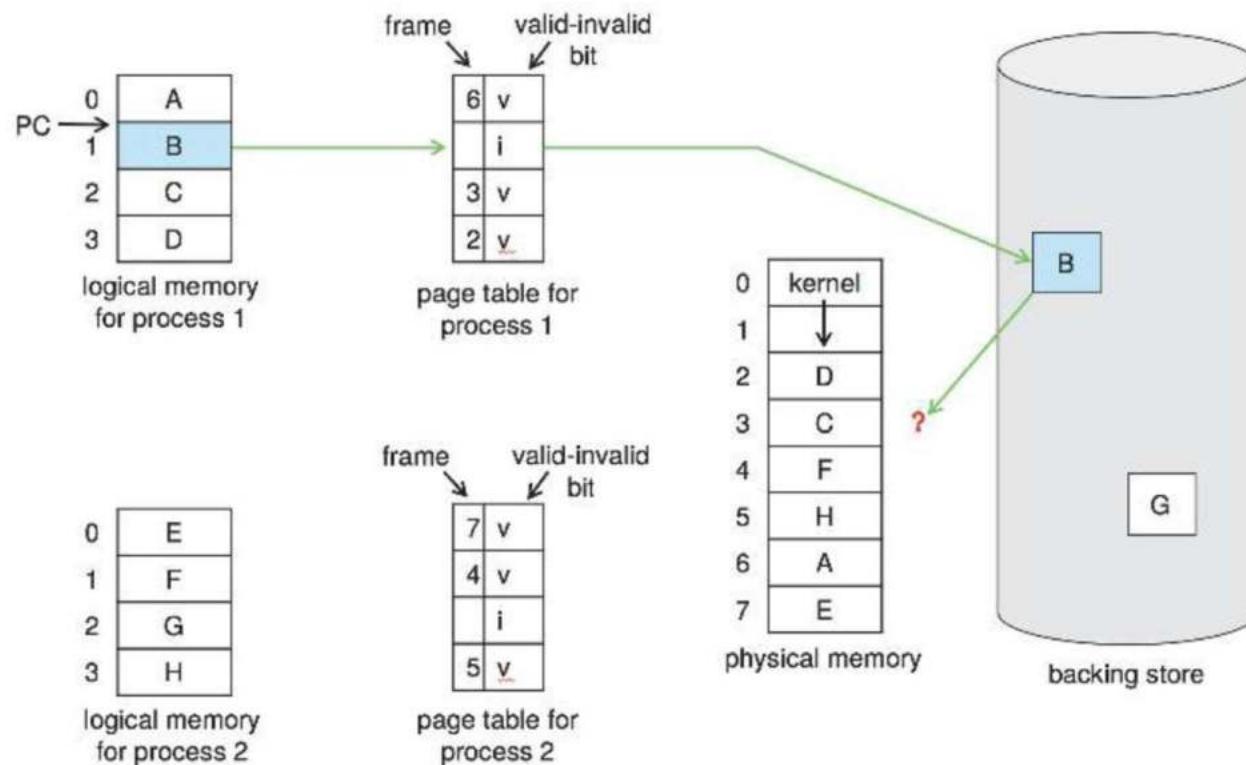


Copy-on-Write

- La call `fork()` crea duplicato del padre, se il figlio fa subito `exec()` non necessario duplicato di memoria
- **Copy-on-Write** (COW) permette ai processi padre e figlio di **condividere** inizialmente le stesse pagine in memoria
 - Le pagine sono marcate come pagine copy-on-write
 - Se uno dei processi modifica una pagina condivisa allora questa è copiata
 - COW permette una creazione di processo più efficiente dal momento che solo le pagine modificate sono copiate
 - Tecnica usata su Linux, Windows e macOS
- `vfork()` (virtual memory fork) è variante di `fork()`
 - Sospende il padre e usa l'address space del padre
 - Non usa copy-on-write quindi può scrivere sulla memoria del padre
 - Progettato per avere subito la call `exec()` per il figlio
 - Molto efficiente

Sostituzione delle Pagine

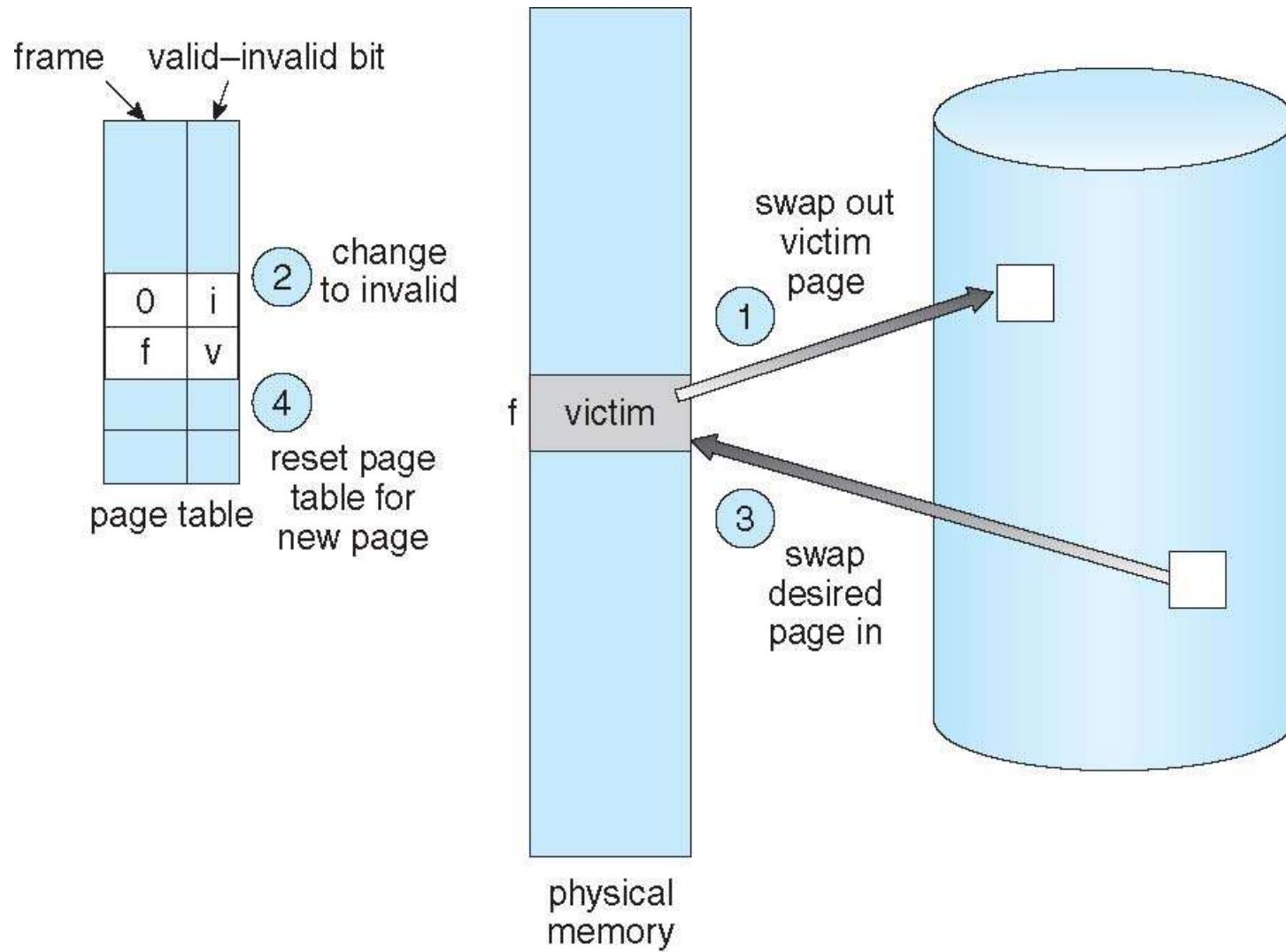
- Se un processo di 10 pagine ne usa la metà allora risparmiati 5 accessi
- Si può aumentare il grado di multiprogrammazione del doppio
- Si sta **sovrallocando** memoria
- ... ma se non c'è un frame libero dopo il page-fault?



Sostituzione delle Pagine

- Se un processo di 10 pagine e ne usa la metà allora risparmiati 5 accessi
 - Si può aumentare il grado di multiprogrammazione del doppio
 - Si sta **sovrallocando** memoria
-
- Se non c'è un frame libero dopo il page-fault?
 - SO può terminare il processo (non è la scelta migliore, utente non dovrebbe accorgersi della gestione)
 - SO può fare uno swapping di un intero processo riducendo il livello di multiprogrammazione (ma non si usa perché troppo costoso)
 - SO può fare **page replacement** combinando swapping e paging

Sostituzione di Pagine



Sostituzione delle Pagine

- La sostituzione di pagine è fondamentale per il demand paging
 - completa la separazione tra la memoria logica e fisica – una larga memoria virtuale può essere ottenuta da una memoria fisica più piccola
- Previene la **sovrallocazione** di memoria modificando il servizio di page-fault per includere la sostituzione di pagine
- Usa **bit di modifica (dirty bit)** per ridurre l'overhead del trasferimento di pagine – solo le pagine modificate sono scritte su disco

Sostituzione di Pagine

1. Trova la locazione della pagina desiderata su disco
2. Trova un frame libero:
 - se c'è un frame libero usalo
 - se non c'è usa algoritmo di page replacement per selezionare un **frame vittima**
 - scrivi il frame vittima su disco se il dirty bit è impostato
3. Porta la pagina desiderata nel nuovo frame libero; aggiorna la tabella di pagina e dei frame
4. Continua il processo riavviando l'istruzione che ha causato il trap

Nota: potenzialmente 2 transferimenti di pagina per page fault

- Incrementato effective access time (EAT)
- Utilizzando il bit di modifica si può ridurre un accesso (2 solo se modifica)

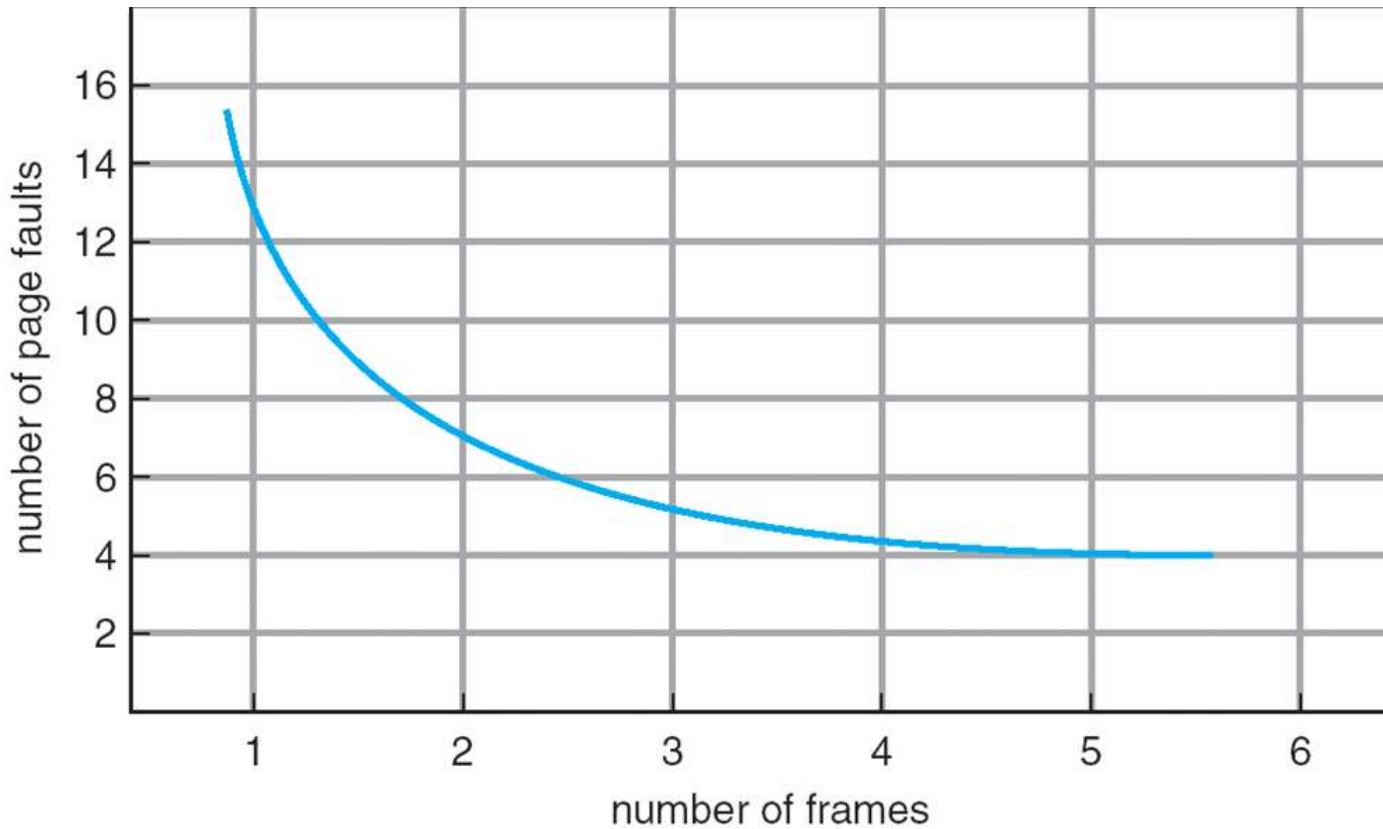
Algoritmi di sostituzione di pagina e di frame

- Per implementare il demand paging occorre:
 - **Algoritmo di allocazione frame**
 - Se più processi, quanti frame assegnare ad ogni processo?
 - **Algoritmo di sostituzione pagina**
 - Se sostuzione pagina quali frame sostituire?
 - Si vuole mantenere un basso tasso di page-fault sia per il primo accesso che per il riaccesso (accesso I/O costoso anche il minimo miglioramento è importante)

Algoritmi di sostituzione di pagina e di frame

- Per implementare il demand paging occorre:
 - **Algoritmo di allocazione frame**
 - **Algoritmo di sostituzione pagina**
- Algoritmo valutato su una particolare **stringa dei riferimenti** in memoria (reference string) calcolando il numero di page fault generati
 - Riferimenti generati sinteticamente o registrando accessi
 - Stringa indica solo numeri di pagina, non indirizzo completo
 - ▶ Accessi ripetuti su stessa pagina non causa un page fault
 - ▶ Esempio, per la sequenza che segue, assumendo 100 bytes per pagina
0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105
 - ▶ Si ottiene
1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1
 - ▶ Risultati dipendono dal numero di frame disponibili (se un solo frame allora tutti fault)

Grafo dei Page Fault vs Numero di Frame



Grafo di page fault per numero di frame

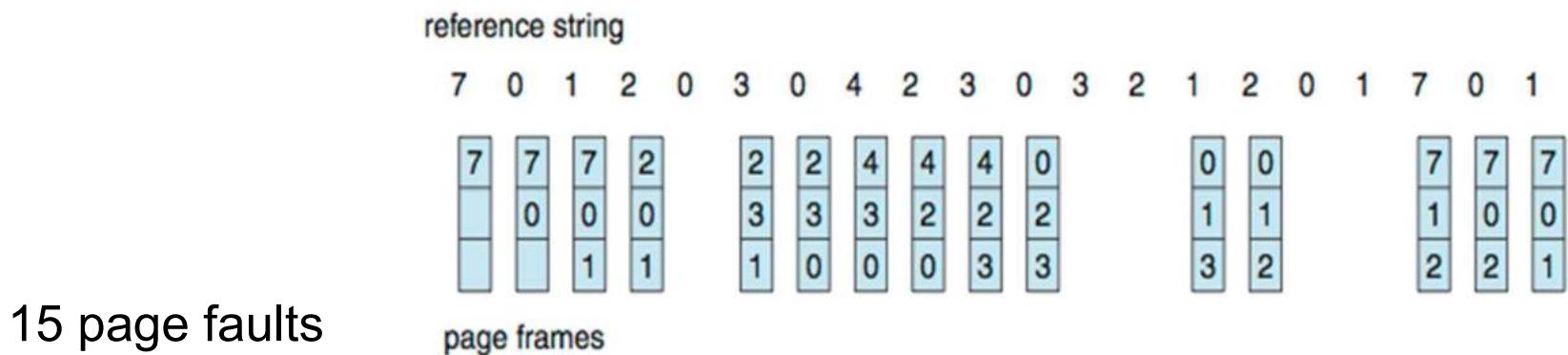
- all'aumentare dei frame diminuiscono i page-fault

In tutti gli esempi che seguono la **stringa dei riferimenti** sarà

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Algoritmo First-In-First-Out (FIFO)

- Sostituzione della pagina più vecchia
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frame (3 pagine in memoria nello stesso momento per processo)



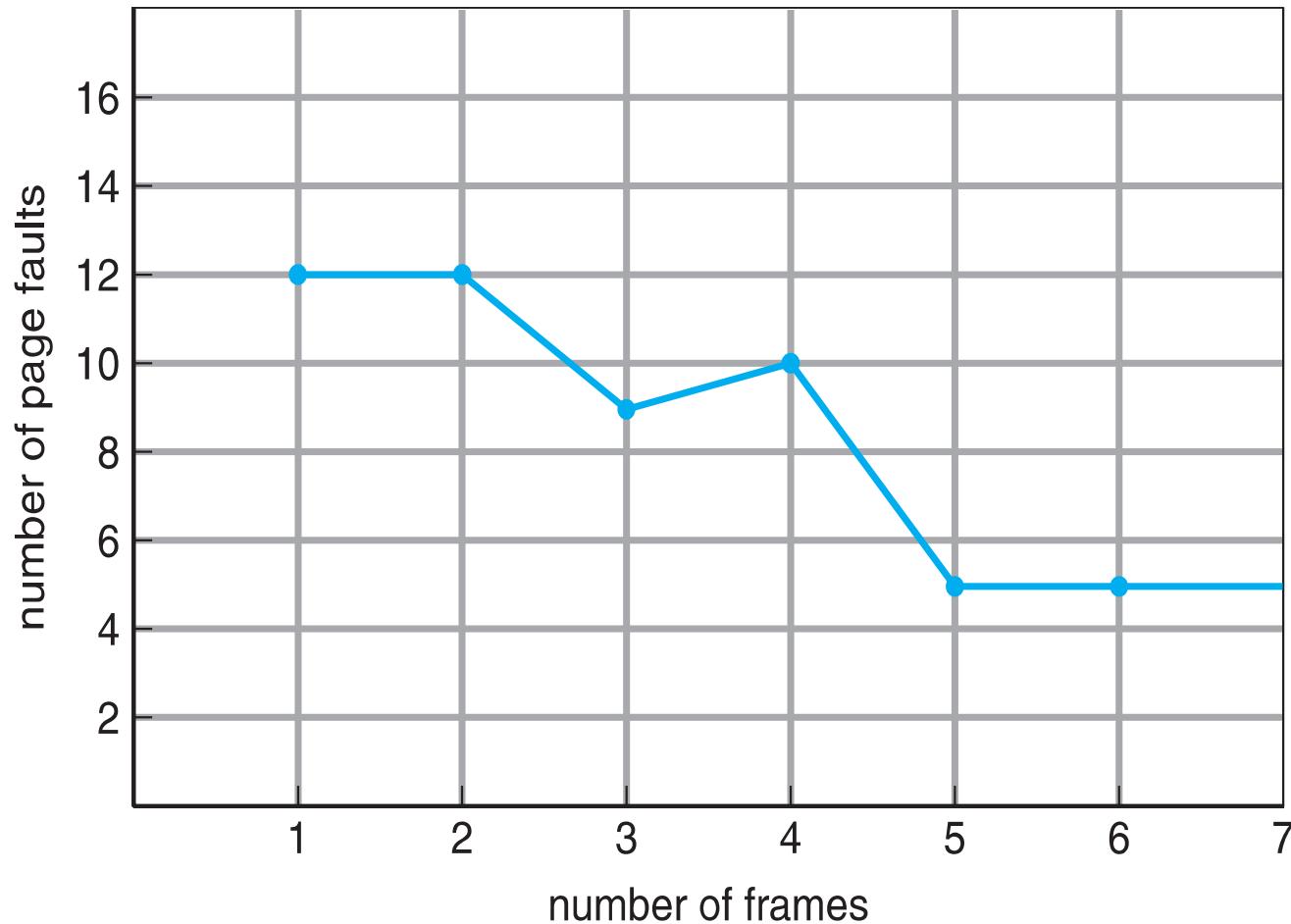
- Le pagine più recenti entrano in coda le più vecchie escono
 - criterio arbitrario (es. vecchie pagine possono contenere dichiarazioni)
- Facile da implementare, ma performance varia con la reference string:
 - Si consideri 1,2,3,4,1,2,5,1,2,3,4,5
 - ... Aggiungendo più frames può causare più page fault
 - ▶ **Anomalia di Belady**

FIFO e anomalia di Belady

Si consideri 1,2,3,4,1,2,5,1,2,3,4,5

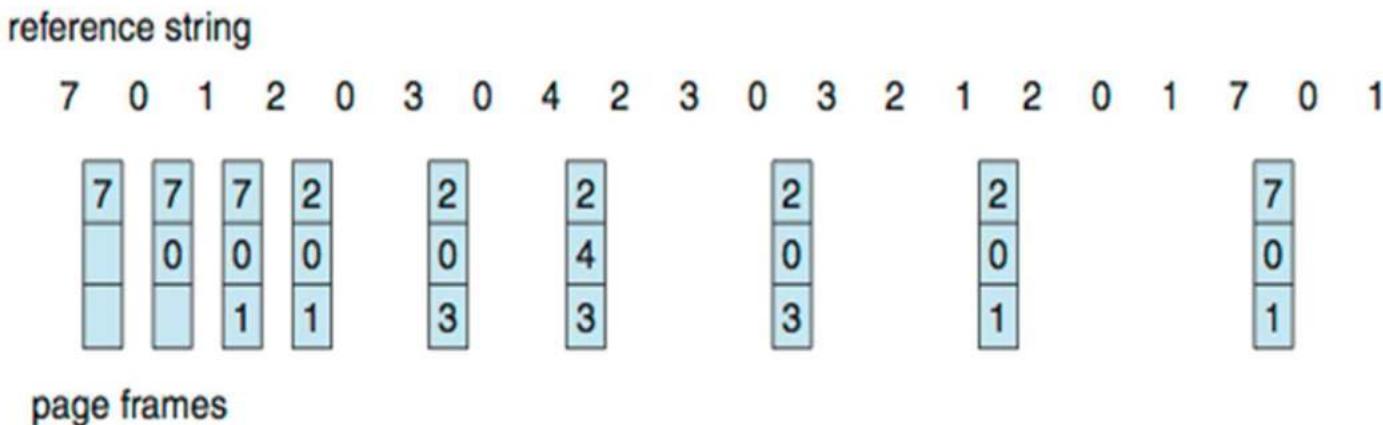
... aggiungere più frames può causare più page fault

Anomalia di Belady (non desiderata)



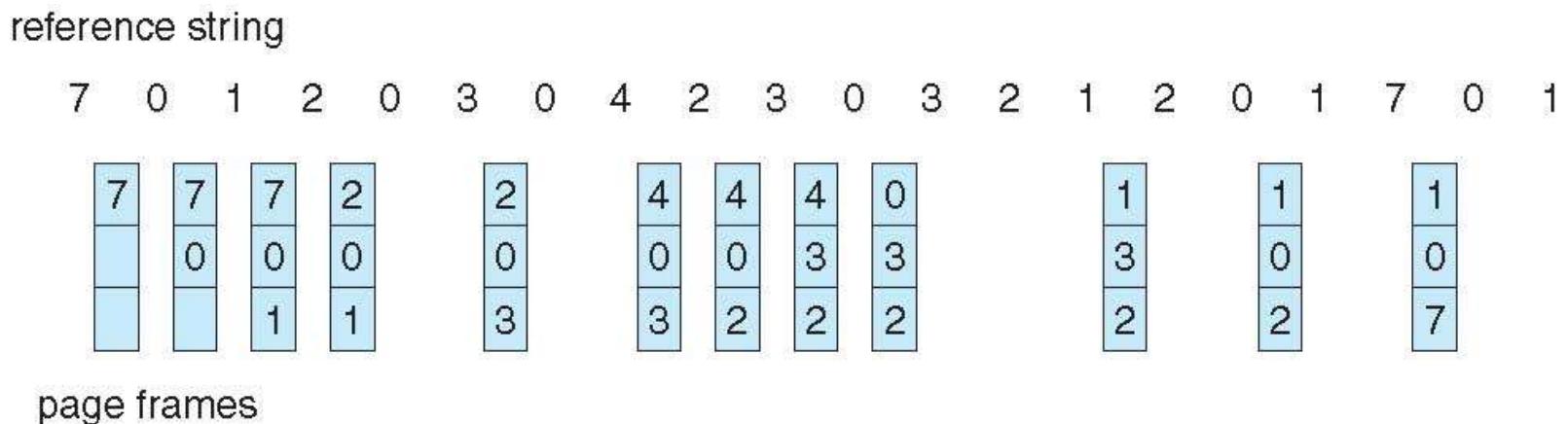
Algoritmo Ottimale

- Sostituisce la pagina che non verrà utilizzata per il periodo più lungo
 - Nell'esempio produce 9 page-fault
- Come si può sapere?
 - Non si può leggere nel futuro
- Usato per valutare le performance degli algoritmi



Algoritmo Least Recently Used (LRU)

- Usa la conoscenza del passato invece che quella del futuro
- Sostituisci le pagine che per più tempo non sono state usate
- Associa ad ogni pagina il tempo dell'ultimo uso



- 12 fault – meglio di FIFO ma peggio di OPT
 - Con riferimento a 4 sostituisce 2 anche se subito dopo si usa
- Solitamente un buon algoritmo, frequentemente usato
 - Ottimale per l'inverso della stringa dei riferimenti
- Il problema è come implementarlo in modo efficiente ...
 - assistenza hardware può essere richiesta

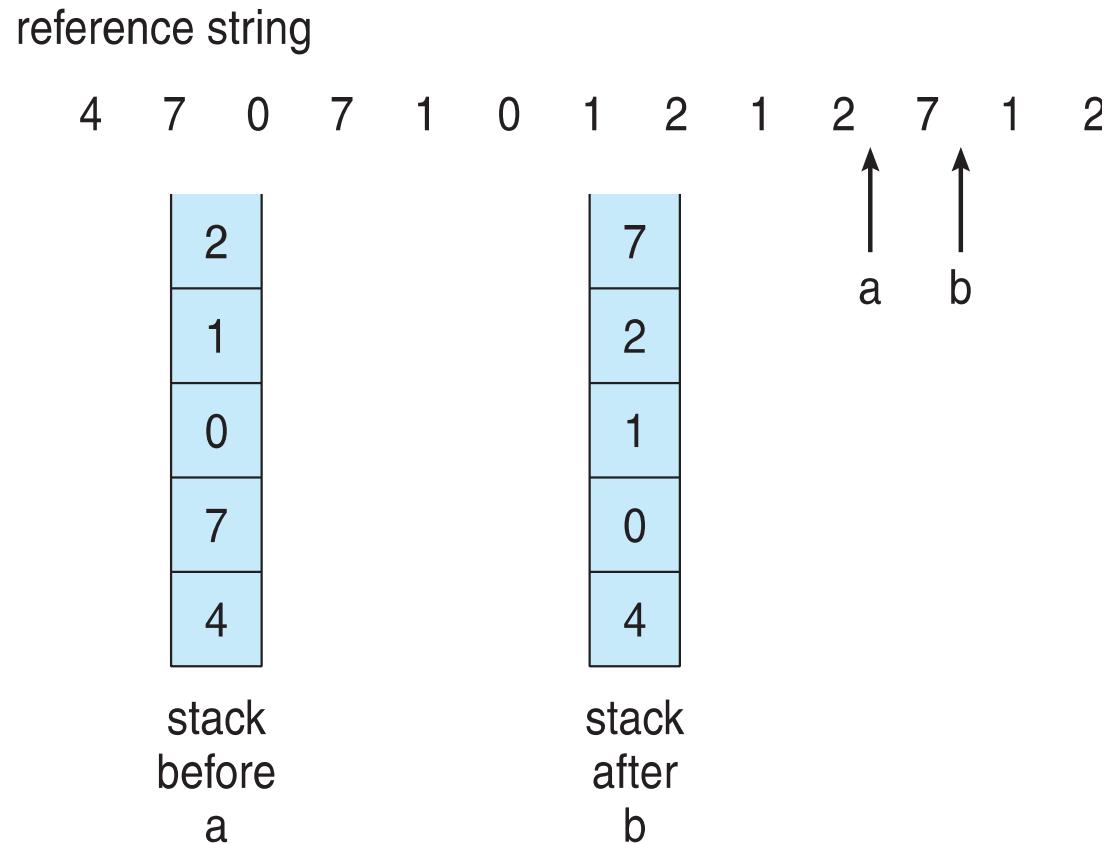
Algoritmo LRU

- Due possibili implementazioni
- Implementazione con **Counter**
 - Ogni elemento della tabella delle pagine ha un registro counter (time-of-use)
 - CPU ha un clock logico/contatore incrementato per riferimenti a memoria
 - Per ogni riferimento ad una pagina il clock copiato nel counter
 - Quando una pagina deve essere cambiata, cerca il valore minore del counter
 - Richiesta: ricerca attraverso la tabella e scrittura del counter per ogni accesso
- Implementazione con **Stack**
 - Mantiene uno stack di numeri di pagina
 - Al riferimento di pagina si mette in cima allo stack
 - La più recente è in cima, la meno recente sul fondo
 - Con lista doppiamente linkata facile accedere al fondo e al top
 - ... non viene fatta alcuna ricerca per la sostituzione
 - Approccio appropriato per realizzazione (o microcodice) software
- LRU e OPT sono casi di **stack algorithm** senza l'anomalia di Belady
 - Insieme pagine in memoria per n frame sottoinsieme di n+1
 - Per LRU le n più recenti rimangono tali anche con n+1 frame

Uso dello stack per registrare il più recente riferimento di pagina

Implementazione con **Stack**

- Mantiene uno stack di numeri di pagina
- Al riferimento di pagina si mette in cima allo stack



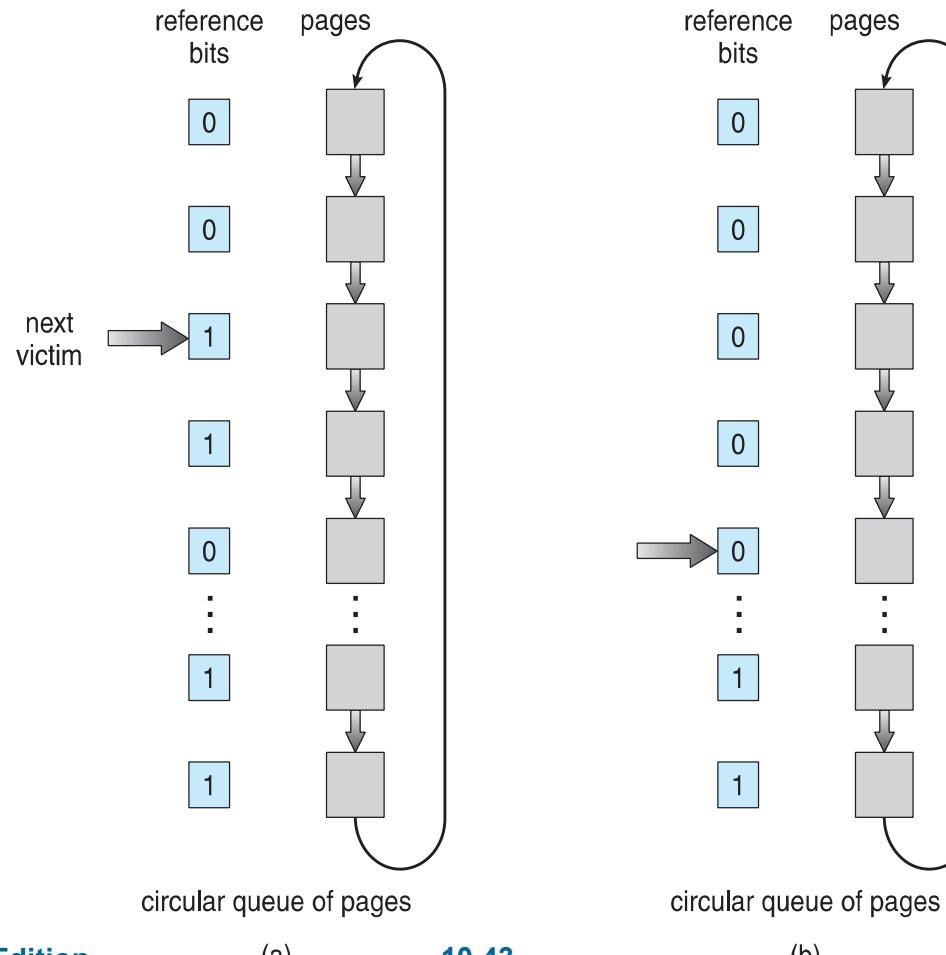
Algoritmi di Approssimazione a LRU

- LRU ha bisogno hardware speciale che non tutti i sistemi forniscono
- In alcuni casi il supporto è un bit di riferimento (**reference bit**)
 - Ogni pagina associata al bit di riferimento, inizialmente = 0
 - Quando si fa riferimento alla pagina (lettura o scrittura) bit settato ad 1
 - Si può sostituire qualunque pagina con bit = 0 (se esiste)
 - ▶ Non si conosce l'ordine di utilizzo ...
- **Algoritmo second-chance**
 - È un FIFO con un reference bit
 - ▶ Dopo la selezione della pagina si controlla il bit, se 1 seconda chance
 - Aggiorna il **clock**
 - Se la pagina da sostituire ha
 - ▶ Reference bit = 0 -> sostituisci
 - ▶ reference bit = 1 allora:
 - setta reference bit 0, lascia la pagina in memoria
 - passa alla prossima pagina con stesse regole
 - Implementato con coda circolare

Second-Chance (clock) Page-Replacement Algorithm

Algoritmo second-chance

- Implementato con coda circolare
- Puntatore indica la pagina da sostituire
- Quando serve un frame avanza finché non trova uno zero
- Pagina sostituita in quella posizione
- Se tutti i bit sono 1 diventa FIFO



Algoritmo Enhanced Second-Chance

- Migliora l'algoritmo usando insieme il reference bit ed il bit di modifica (se disponibile)
- Prendi coppie ordinate (reference, modify)
 1. (0, 0) né usata e né modificata – miglior pagina da sostituire
 2. (0, 1) non usata ma modificata – non così buona, bisogna scrivere prima della sostituzione
 3. (1, 0) usata ma non scritta – potrebbe presto essere usata di nuovo
 4. (1, 1) usata e modificata – può essere riusata e deve essere salvata prima della sostituzione
- Quando occorre una sostituzione di pagina usa lo schema clock (second chance) ma cerca una pagina della classe più bassa non vuota
 - Si potrebbe dover scorrere la lista più volte
 - Preferenza per le pagine che consentono minor I/O

Algoritmi Contatori

- Altri algoritmi per approssimare LRU
- Mantieni un contatore del numero di riferimenti per ogni pagina
 - Non molto usati ... costosi e non approssimano bene l'ottimo
- **Least Frequently Used (LFU) Algorithm:** sostituisci pagine con il contatore più basso
 - Pagine intensamente usate hanno contatori alti
 - Però alcune pagine intensamente usate solo inizialmente
 - ▶ Si può decrementare progressivamente il contatore
- **Most Frequently Used (MFU) Algorithm:** sostituisci pagine con contatore più alto
 - assumendo che le pagine con il contatore più basso siano le più recenti e debbano essere ancora usate

Algoritmi Page-Buffering

- Altri metodi possono essere utilizzati insieme a quelli page-replacement
- I sistemi mantengono un pool di frame liberi
 - Il frame è a disposizione quando occorre, non tovato a tempo di page-fault
 - Leggi la pagina in un frame libero e seleziona una vittima da aggiungere al pool
 - Quando è opportuno porta fuori la vittima
- Variazione: si mantiene lista di pagine modificate
 - Quando il sistema di paging è idle copia pagine modificate in backing store e setta il bit di modifica a non-dirty, si evitano gli accessi in memoria per copiare le pagine modificate
- Si può mantenere intatto il contenuto del frame libero
 - Se è referenziato prima del riuso non occorre ricaricare il contenuto dal disco
 - Alcune versioni di UNIX lo utilizzano con il second chance
 - Utile per ridurre la penalità se viene selezionata una vittima sbagliata

Applicazioni e Sostituzione di Pagina

- In tutti gli algoritmi presentati il SO cerca di anticipare il futuro accesso in memoria
- Alcune applicazioni gestiscono meglio – i.e., databases
- Applicazioni che usano intensamente la memoria possono fare buffering, quindi si può avere un doppio buffering
 - SO mantiene copia di pagine in memoria come un I/O buffer
 - L'applicazione mantiene la pagina in memoria per il suo funzionamento
- SO può dare diretto accesso a queste applicazioni
 - **Raw disk** mode e raw I/O
 - Con questa modalità vengono bypassati diversi servizi, es. buffering, file locking, file name, directory, etc.

Allocazione di Frame

- Come si allocano i frame per processo?
- Caso di un Sistema con 128 frame
 - SO ne prende 35 e 93 per i processi utente
 - Se pure demand, tutti i 93 nella lista dei frame liberi
 - Il processo in esecuzione richiede i frame
 - ▶ fino a 93 page fault prende pagine libere, poi sostituzione
 - ▶ alla fine rilascio (tutti i 93 nella free frame list)
- Molte varianti di questo approccio
 - ▶ Si possono sempre lasciare liberi frame sulla frame list e postporre la scrittura nello swap space della vittima per sostituire
 - ▶ Però il metodo alloca tutti i frame per un processo

Allocazione di Frame

- La strategia di allocazione deve tenere conto di un **minimo numero** di frame richiesti da un processo
 - Prestazioni (più frame, meno page fault)
 - Alcune istruzioni richiedono più frame, se page fault restarted
 - ▶ Dipende da architettura
- Esempio: IBM 370 – 6 pagine per l'istruzione SS MOVE:
 - istruzione di 6 byte, può richiedere 2 pagine
 - 2 pagine per gestire *from*
 - 2 pagine per gestire *to*
- Il **massimo** numero dipende dai frame disponibili nel sistema

Allocazione di Frame

- Due principali schemi di allocazione
 - **Fixed allocation**
 - **Priority allocation**
- Molte varianti

Fixed Allocation

□ Equal allocation

- Se m frame per n processi allora alloca ad ognuno m/n
- Per esempio, se 100 frame (tolti quelli per SO) e 5 processi, per ogni processo 20 frame
- Mantieni un free frame buffer pool

□ Proportional allocation

- Però processi diversi (processo studente vs database)
- Alloca proporzionalmente alla dimensione del processo

— s_i = size of process p_i

— $S = \sum s_i$

— m = total number of frames

— a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 62 \approx 4$$

$$a_2 = \frac{127}{137} \times 62 \approx 57$$

- Esempio: 64 frame, 2 processi ...

Fixed Allocation

□ Equal and Proportional allocation

- Se aumenta il numero dei processi diminuisce la memoria allocata per processo
- Problema: stessa allocazione per priorità differenti
 - ▶ Si può includere la priorità nel computo dei frame allocate

□ Priority allocation

- Usa uno schema di proportional allocation usando le priorità invece della dimensione

Rimpiazzamento Locale vs Globale

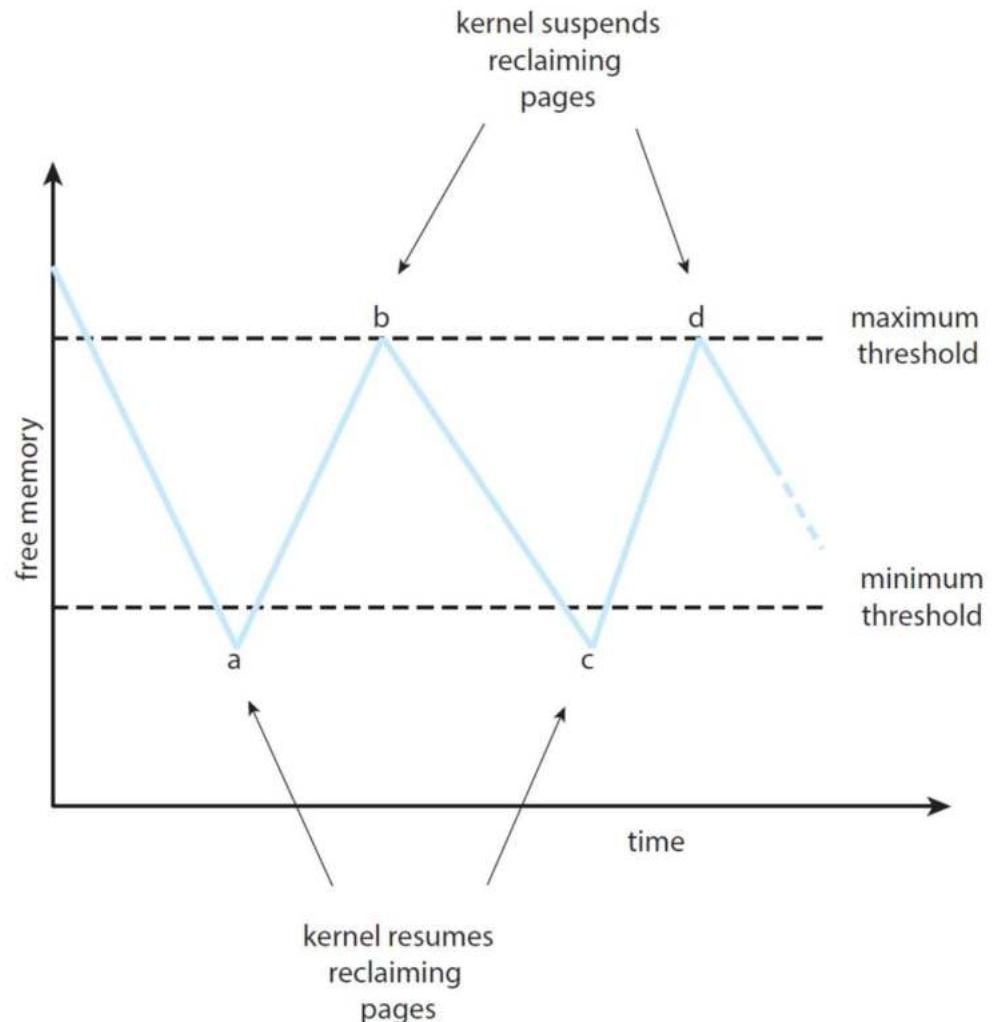
- Se il processo P_i genera un page fault,
 - seleziona per rimpiazzare uno dei suoi frame (locale)
 - selezione per rimpiazzare un frame di un processo con priorità più bassa (globale)

Rimpiazzamento Locale vs Globale

- **Global replacement** – un processo seleziona un frame da rimpiazzare dall'insieme di tutti i frame
 - un processo può prendere un frame di un altro processo
 - ... ma tempi di esecuzione di un processo possono variare molto
 - ... ma più alto il throughput (quindi scelta più comune)
- **Local replacement** – ogni processo seleziona solo dal suo insieme di frame allocati
 - Prestazioni più stabili e consistenti
 - ... ma memoria sottoutilizzata

Allocazione Globale

- **Global replacement** – un processo seleziona un frame da rimpiazzare dall'insieme di tutti i frame
- Strategia per global page-replacement
 - Non si attende che la free-list si esaurisca
 - Page replacement quando sotto una soglia (tipicamente LRU approx)
 - Sotto soglia minima il kernel inizia a recuperare frame ...
 - finché non va sopra la soglia massima
 - Strategie chiamate **reapers**

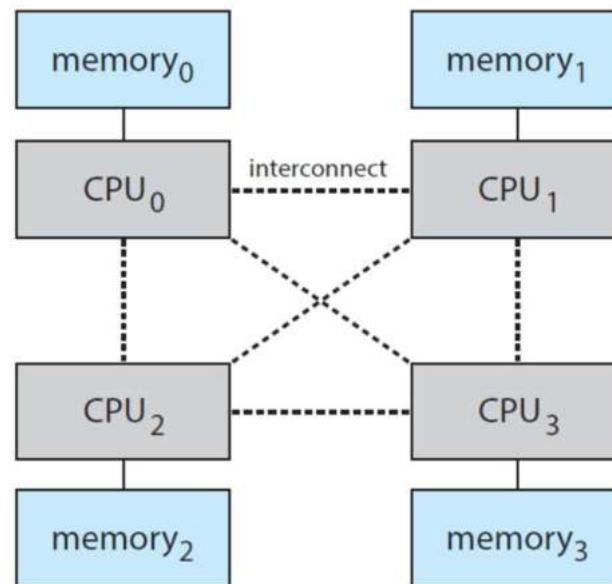


Allocazione Globale

- **Global replacement** – un processo seleziona un frame da rimpiazzare dall'insieme di tutti i frame
- Strategia per global page-replacement può diventare più aggressiva
 - Se non riesce a mantenere i frame liberi può passare da LRU approx a FIFO
 - In Linux se livelli troppo bassi out-of-memory (OOM) killer termina un processo
 - Ogni processo ha un OOM score, più è alto più rischia
 - OOM dipende dalla memoria usata, più percentuale più alto il numero
 - Si può vedere in /proc, es., con PID 2500 /proc/2500/oom_score
- Anche le soglie per attivare le **reaper routine** si possono configurare

Non-Uniform Memory Access

- Fino ad ora si è assunto un accesso uniforme alla memoria virtuale
- Non uniforme con **NUMA** – velocità di accesso a memoria varia
 - Considera schede con più CPU e memorie connesse con bus
 - Ogni CPU con sua memoria locale ad accesso più veloce
 - Accesso in memoria meno veloce ma maggiore parallelismo
 - Se trattato come uniforme l'accesso può essere molto rallentato
 - I frame di memoria allocati più vicino possibile alla CPU di riferimento



Non-Uniform Memory Access

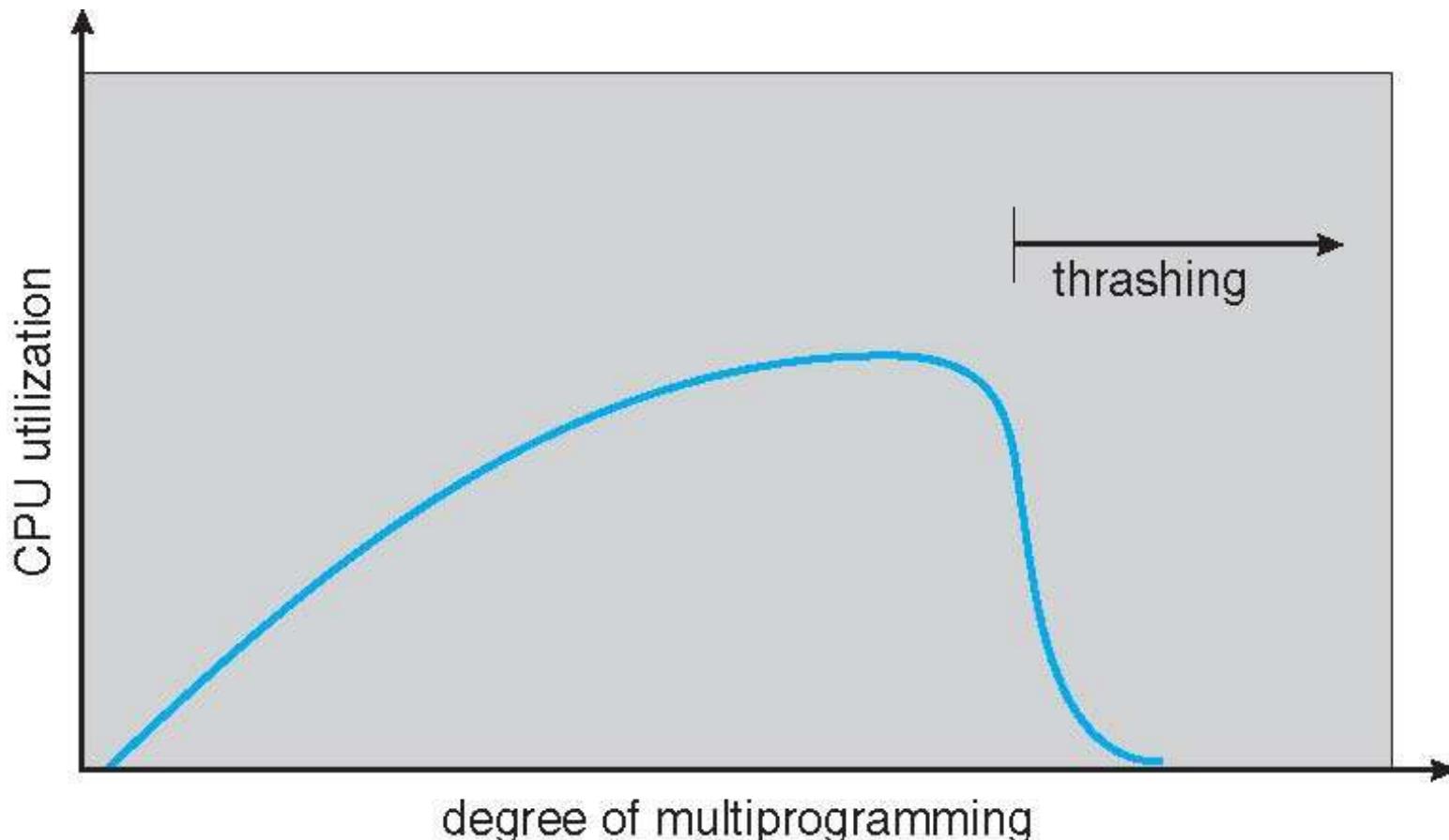
- Fino ad ora si è assunto un accesso uniforme alla memoria virtuale
- Non uniforme con **NUMA** – velocità di accesso a memoria varia
 - allocare memoria “vicina” alla CPU su cui il thread è schedulato
 - se page-fault una memoria virtuale NUMA-aware alloca il nuovo frame vicino
 - Scheduler deve tracciare la CPU su cui gira il processo
 - allocare stessa CPU e frame vicini, con thread ancora più complicato
- Linux utilizza gerarchia di domini di scheduling
 - Scheduler limita la migrazione tra domini
 - Frame-list separate per ogni nodo NUMA
- Solaris utilizza degli **Igroups** (gruppi di località)
 - Ogni CPU nel gruppo può accedere a memoria nel gruppo con una latenza
 - Gerarchia di gruppi
 - Cerca di schedulare i thread di un processo e la memoria per quel processo all'interno del Igroup, altrimenti passa ai gruppi vicini

Thrashing

- Se un processo non ha “abbastanza” frame, il page-fault rate è molto alto
 - Page fault per avere una pagina
 - Rimpiazza un frame esistente
 - ... ma rapidamente il frame deve essere ripreso
 - Rimpiazzamento continuo di pagine ...
 - Questo porta a:
 - ▶ Basso utilizzo di CPU
 - ▶ SO può voler incrementare il grado di multiprogrammazione
 - ▶ Altro processo aggiunto, etc.
- **Thrashing** ≡ un processo è occupato nello swapping di pagine in e out

Thrashing

- SO monitora utilizzo di CPU, se basso aumenta il livello di multiprogrammazione
- Algoritmo globale di page-replacement, i processi iniziano a sottrarre frame ad altri processi, a loro volta vanno in page-fault, serve il paging device per swap in e out, si accodano i processi e si svuota la coda ready ... aumenta la multiprogrammazione
- L'utilizzo della CPU aumenta con la multiprogrammazione, poi va in thrashing



Demand Paging e Thrashing

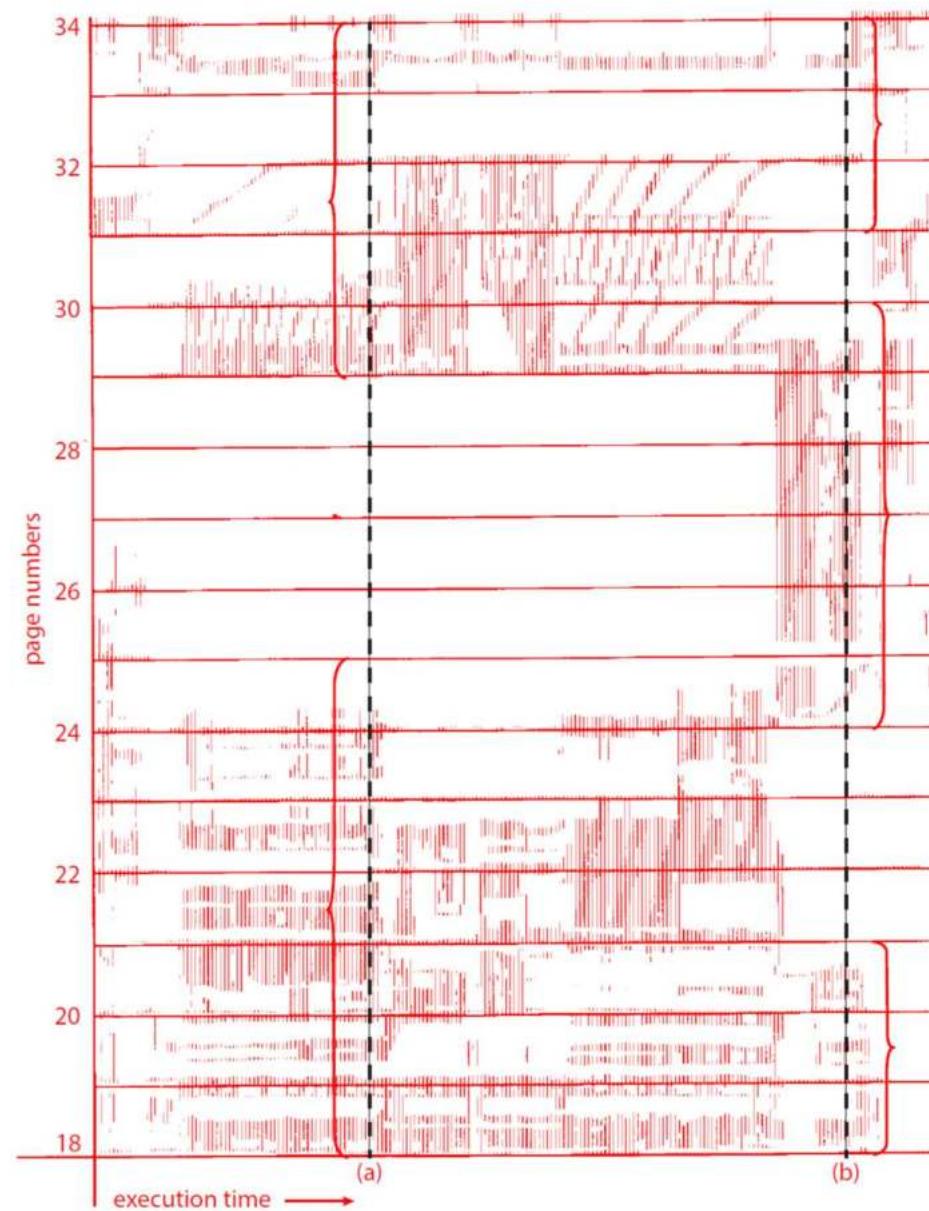
- Limitare l'effetto del thrashing usando un algoritmo di local replacement
 - Frame rimpiazzati solo localmente, non possono essere “rubati”
 - ... un processo in thrashing non trascina gli altri ma ...
 - ... non risolto il problema, un processo in thrashing occupa il dispositivo di paging e rallenta il page fault di tutti i processi
- Per prevenire un thrashing bisogna allocare per un processo tanti frame quanti ne necessita, come si fa a sapere?
 - Guardando a quanti ne sta attualmente usando ...
 - Si definisce un modello di località (**locality model**) del processo
- Locality Model:
 - Un processo passa da locality a locality durante l'esecuzione
 - Le località sono insiemi di pagine che sono usate insieme
 - Le località possono sovrapporsi e nuove funzioni corrispondono a nuove località
 - Località di una funzione: istruzioni, variabili locali, sottoinsieme delle globali

Località in una sequenza di riferimenti in memoria

Località al tempo (a)
 $\{18-24, 29-33\}$

Località al tempo (b)
 $\{18-20, 24-29, 31-33\}$

18, 19, 20 si
sovrappongono



Località

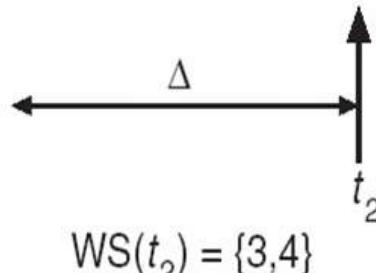
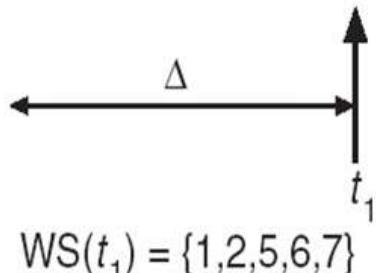
- Le località sono definite dalla struttura del programma e dalle sue struttura dati
- Notare che il modello di località è anche il principio che giustifica l'utilizzo delle cache
- Se si riesce ad allocare i frame per le località di un processo non genererà page fault finché non cambia la località
- Se invece non si allocano frame sufficienti per la località corrente andrà verso il thrashing

Modello Working Set

- Il modello del Working Set è basato sull'assunzione di località
- Utilizza parametro Δ per definire una working-set window
 - I più recenti Δ riferimenti in memoria
 - ▶ Es. 10000 istruzioni
- L'insieme delle Δ pagine più recenti è il **Working Set**
 - Se una pagina è in uso attivo è nel WS, se non è più utilizzata esce dal WS dopo Δ unità di tempo dall'ultimo suo riferimento
- Il WS approssima la località di un programma
- Esempio con $\Delta = 10$ (dimensione di WS varia)

page reference table

... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...



Modello Working-Set

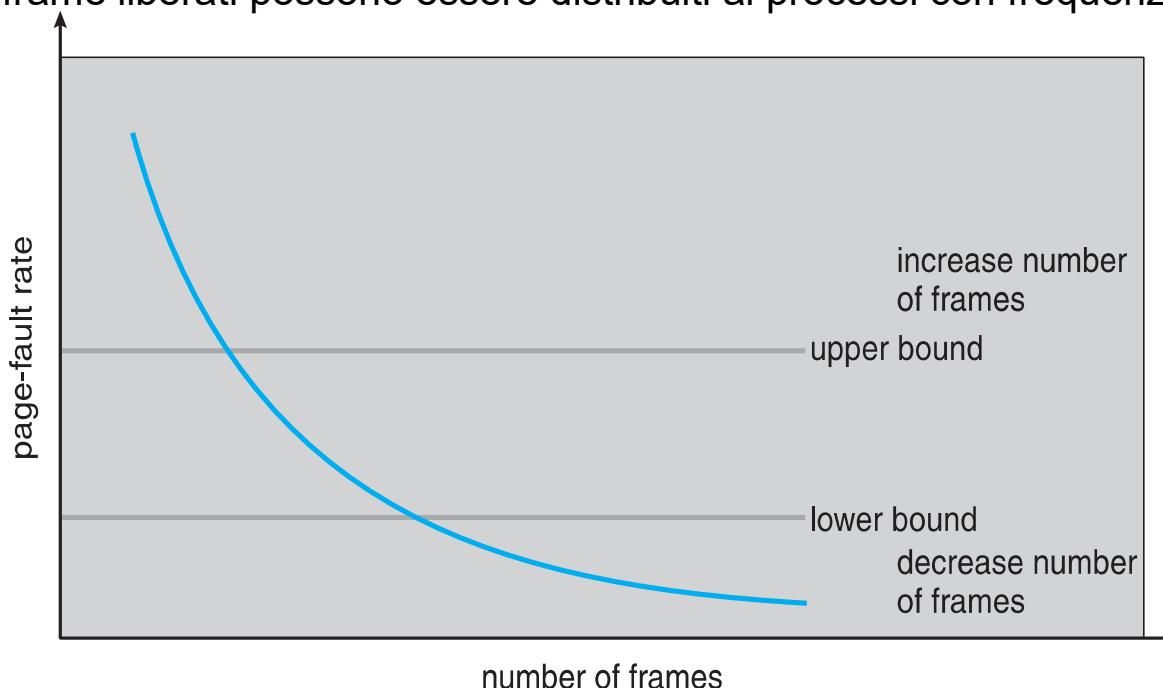
- L'accuratezza del WS dipende da Δ
- Dim del Working Set WSS_i (working set size del processo P_i)
 - numero totale di pagine riferite nel più recente Δ (variabile nel tempo)
 - se Δ troppo piccolo non contiene l'intera località
 - se Δ troppo grande può sovrapporre più località
 - se $\Delta = \infty \Rightarrow$ contiene tutto il programma
- WSS_i è la caratteristica curciale da monitorare
 - $D = \sum WSS_i$ è la richiesta totale di frame
 - Approssima la località
- Se $D > m \Rightarrow$ si verifica il thrashing
- Il SO monitora il Working Set di ogni processo
 - Alloca frame secondo il WSS_i ,
 - Se ci sono frame extra alloca un nuovo processo
 - Se $D > m$ allora suspende o fa swap out di uno dei processi

Tracciare il Working Set

- La finestra del Working Set è mobile
 - Ad ogni riferimento in memoria un elemento entra, l'altro esce
- Si può approssimare con un timer ad intervalli fissi + un bit di riferimento
- Esempio: per $\Delta = 10000$
 - Timer interrupt dopo ogni 5000 unità di tempo
 - Mantiene in memoria 2 bit per ogni pagina
 - Quando il timer interrompe copia e setta i valori di tutti i reference bit a 0
 - Se bit attuale o uno di 2 bit in memoria = 1 \Rightarrow la pagina è nel working set
 - Non accurato, non sappiamo dove è avvenuto il riferimento in 5000 unità
 - Miglioramento: 10 bit di storia e interrupt ogni 1000 unità di tempo

Frequenza dei Page-Fault

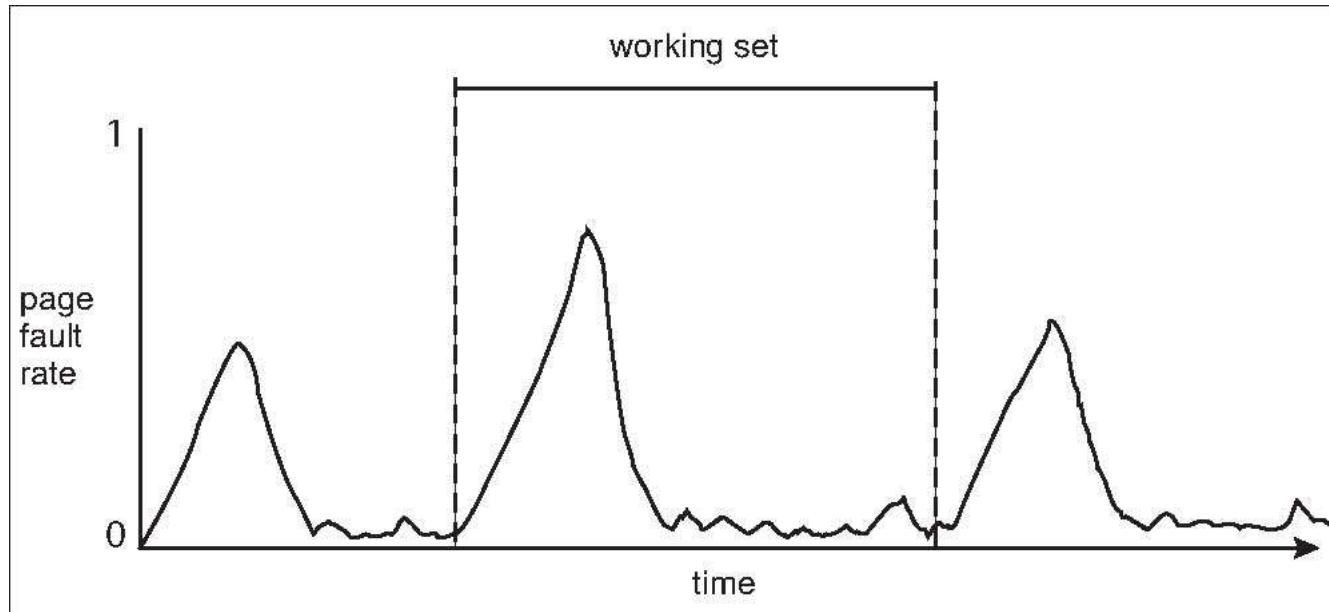
- Approccio più diretto del Working Set Size
- Thrashing aumenta la frequenza di page fault
- Stabilire un tasso “accettabile” di **page-fault frequency (PFF)** per utilizzare una politica di rimpiazzo locale
 - Se il tasso attuale è troppo basso il processo perde frame
 - Se il tasso attuale è troppo alto il processo guadagna frame
- Può essere richiesta la sospensione di un processo
 - i frame liberati possono essere distribuiti ai processi con frequenze più alte



Working Sets e Page Fault Rates

Relazione diretta tra working set di un processo e la frequenza di page-fault

- Working set cambia nel tempo
- Picchi e valli di frequenze di page fault indicano i cambiamenti di località
- L'intervallo tra gli inizi dei picchi definiscono il Working Set



Pratica Corrente

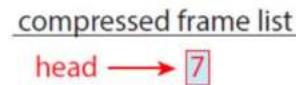
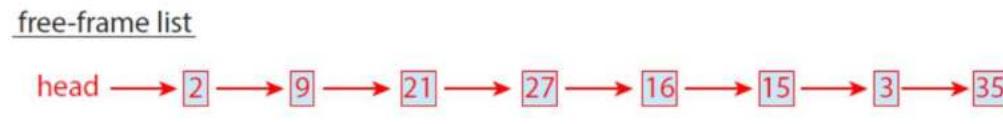
- Thrashing e swapping hanno un importante impatto sulle prestazioni
- Il trend attuale prevede l'utilizzo di memoria fisica sufficiente per evitare sia thrashing che swapping
- Mantenere tutti i working set in memoria tranne che in casi estremi

Compressione di Memoria

- Si utilizza la compressione di memoria per ridurre l'utilizzo di memoria
- Esempio:
 - Free frame list sotto soglia e selezionati frame 15, 3, 35, 26 per liberare memoria



- Invece di scrivere direttamente in swap space fa la compressione di alcuni frame e li mette nei compressed frame (es. 15, 3, 35 compressi in 7 e liberati)



- Quando si fa riferimento alle pagine in 7 si decomprime e rialloca

Compressione di Memoria

- Si utilizza la compressione di memoria per ridurre l'utilizzo di memoria
- Usato nei sistemi mobile (Android, iOS), Windows 10 e macOS
- Velocità di compressione vs riduzione di memoria (compression ratio)
 - Maggiore è la velocità minore la compressione
 - Compressione in parallelo su architetture multicore
 - Microsoft Xpress e Apple WKdm sono veloci e garantiscono buone compressioni

Allocare la Memoria per il Kernel

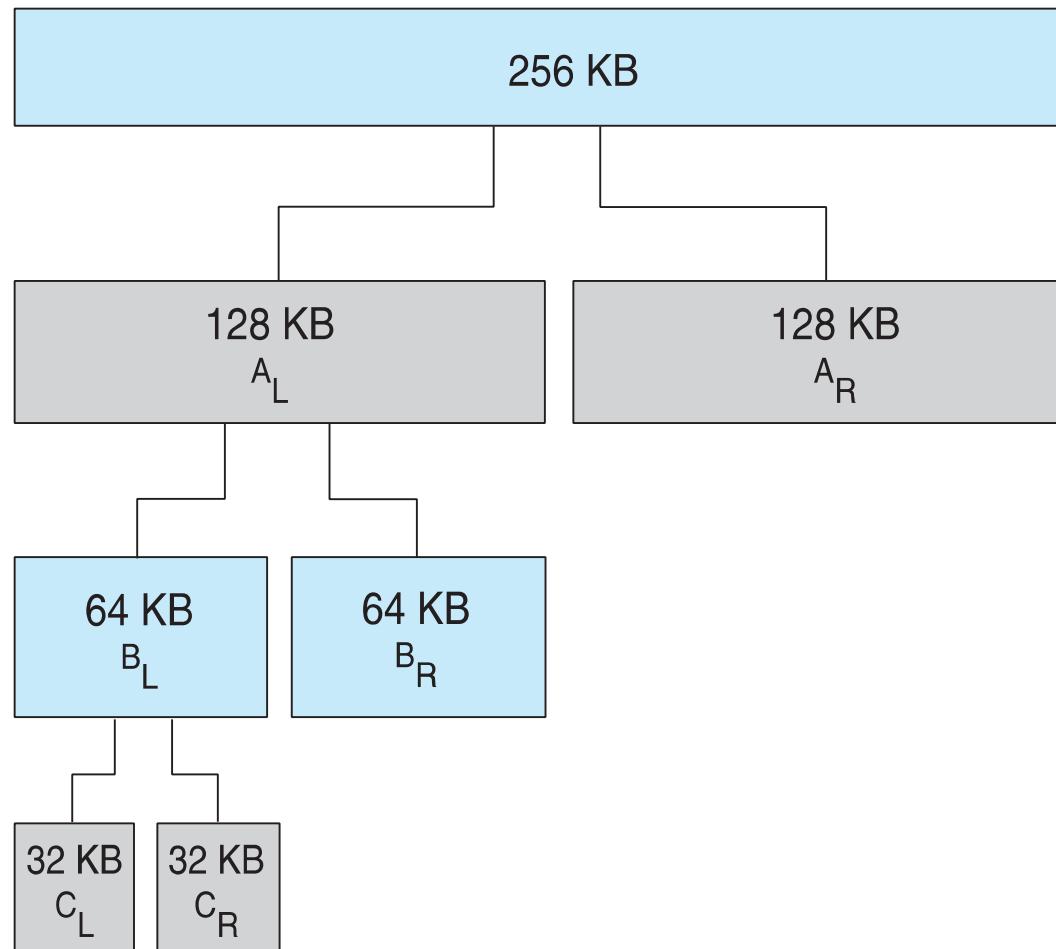
- Trattata in modo differente dalla memoria per processi utente
- Spesso allocate memoria da un pool di free-memory differente
- Due sono le ragioni principali:
 - Kernel richiede memoria per strutture di varie dimensioni, anche sotto la dimensione della pagina, deve usare la memoria in modo più conservativo
 - La memoria del Kernel in alcuni casi necessita di essere contigua
 - ▶ I.e., dispositivi I/O che dialogano direttamente in memoria fisica, in questi casi la memoria deve essere contigua
- Di seguito si presentano due metodi per l'allocazione di memoria ai processi kernel:
 - Buddy System
 - Allocazione slab

Sistema Buddy

- Alloca memoria da segmenti di dimensioni fissate che consistono di pagine fisicamente contigue
 - Un segmento largo e contiguo suddiviso in porzioni più piccolo; più segmenti combinati rapidamente per ottenere frammento contiguo
- Memoria allocata usando **allocatore di potenza-di-2**
 - Soddisfa richieste in unità di dimensione potenza di 2
 - Richieste approssimate alla più alta Potenza di 2
 - Quando occorre un'allocazione più piccolo della disponibile, il chunk corrente diviso in due buddies della prossima potenza di 2 più bassa
 - ▶ Continua finché un chunk appropriato non è disponibile
- Per esempio, assume chunk di 256KB disponibile, il kernel richiede 21KB
 - Diviso in A_L and A_R di 128KB ognuno
 - ▶ Uno ulteriormente diviso in B_L e B_R di 64KB
 - Uno ulteriormente in C_L e C_R di 32KB ognuno – uno usato per soddisfare la richiesta
- Vantaggio – velocemente **compatta** chunk non usati in chunk più grandi
- Svantaggio – frammentazione interna

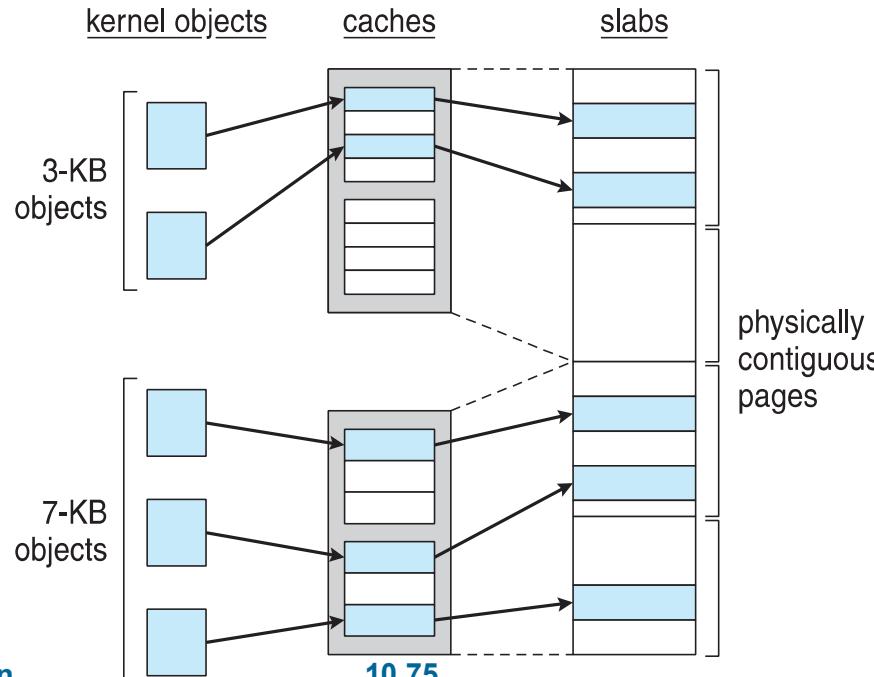
Buddy System Allocator

physically contiguous pages



Slab Allocator

- Strategia alternativa che elimina frammentazione interna
 - Oggetti di un certo tipo preallocati
 - Riuso di oggetti dello stesso tipo
- **Slab** fatto da una o più pagine fisicamente contigue
 - È il contenitore di un oggetto di un certo tipo
- Una **cache** consiste di uno o più slab
 - Una cache per ogni data structure del kernel
 - Ogni cache popolata di **oggetti** – istanze delle strutture dati



Slab Allocator

- Strategia alternativa che elimina frammentazione interna
 - Oggetti di un certo tipo preallocati
 - Riuso di oggetti dello stesso tipo
- **Slab** fatto da una o più pagine fisicamente contigue
 - È il contenitore di un oggetto di un certo tipo
- Una **cache** consiste di uno o più slab
 - Una cache per ogni data structure del kernel
 - Ogni cache popolata di **oggetti** – istanze delle strutture dati
- Quando è creata la cache riempita di oggetti marcati come **free**
- Quando le strutture sono memorizzate oggetti marcati come **used**

- Benefici: non frammentazione, veloce soddisfazione di richieste di memoria

Slab Allocator in Linux

- Esempio: per process descriptor è di tipo `struct task_struct`
- Appross. 1.7KB di memoria
- Nuovo task -> alloca new struct dalla cache
 - Userà `free struct task_struct`
- Slab può essere in tre possibili stati
 1. Full – tutto usato
 2. Empty – tutto libero
 3. Partial – mix di free e used
- Su richiesta, slab allocator
 1. Usa `free struct` in partial slab
 2. Se non c'è, prende uno degli empty slab
 3. Se non c'è empty slab, crea un nuovo empty slab

Slab Allocator in Linux

- Slab iniziato in Solaris, ora diffuso sia per Kernel mode e memoria user in molti SO, Linux da 2.2 prima Buddy
- Linux 2.2 aveva SLAB, ora ha allocatori SLOB e SLUB
 - SLOB per sistemi con limitata memoria
 - Simple List Of Blocks – mantiene 3 liste di oggetti per piccoli, medi e grandi
 - Oggetti 256 byte, 1024 byte, più di una pagina
 - First fit-policy per l'allocazione
 - SLUB è uno SLAB ottimizzato,
 - Introdotto da 2.6.24 per sostituire SLAB
 - usa metadati memorizzati in una page structure, etc.

Altre Considerazioni -- Prepaging

- Prepaging
 - Pure demand paging crea molti page fault allo start di un processo
 - Occorre ridurre il numero di page fault allo start-up del processo
 - Preallocare alcune delle pagine che un processo userà prima che siano riferite
 - Se si utilizza il modello Working Set, si può usare WS per ripristinare le pagine quando il processo è richiamato in memoria
- Costo/beneficio del prepaginaing
 - Se pagine prepaged non usate, I/O e memoria “sprecate”
 - Assumi s pagine sono prepaged con percentuale α delle pagine usata
 - ▶ costo di $s * \alpha$ pages fault evitati > o < del costo del prepaginaing
 - ▶ $s * (1 - \alpha)$ pagine non necessarie?
 - ▶ α vicino a zero \Rightarrow prepaginaing perde
 - Più facile fare prepaginaing di un file che di un programma eseguibile

Altre Considerazioni – Page Size

- I progettisti di SO raramente possono decidere la dimensione delle pagine
- Dimensione definite in fase di progetto
 - Sempre potenza di 2, di solito nel range 2^{12} (4,096 bytes) fino a 2^{22} (4,194,304 bytes)
- Selezione della dimensione delle pagine deve tenere in considerazione:
 - Frammentazione
 - Dimensione della page table
 - Per VM di 4MB (2^{22}) si hanno 4096 pagine di 1024 byte e 512 da 8192 byte
 - Risoluzione
 - Si accede alla porzione di memoria necessaria
 - Minore I/O overhead
 - Maggiore numero di page faults
 - Pagine piccole: località, meno frammentazione, meno I/O overhead
 - Pagine grandi: meno page fault, page table più piccolo

Altre Considerazioni – TLB Reach

- Hit ratio della TLB – quantità di indirizzi risolti in TLB
 - Per aumentare si possono introdurre TLB con più registry, ma costoso
- **TLB Reach** - la quantità di memoria accessibile dalla TLB
 - $\text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size})$
- Idealmente, il working set di ogni processo va immagazzinato nella TLB
 - altrimenti c'è alto livello di page fault ...
 - Per aumentare il reach incrementa la dimensione di pagina (es. da 4 a 16 KB)
 - ... ma potrebbe portare ad un incremento della frammentazione interna dal momento che non tutte le applicazioni richiedono dimensioni grandi
 - ... oppure fornisci dimensioni di pagina multiple
 - Permette alle applicazioni di usare pagine grandi senza incorrere in problemi di frammentazione
- Es. Linux ha pagine di 4KB, ma anche più grandi, e.g., 2M
- Es. ARMv8 più tipi di pagine,
 - ▶ TLB entry hanno contiguous bit (riferiscono blocchi di memoria contigua)
 - ▶ es., da 64KB (16x4KB), 1GB (32x32MB), 2MB (32x64KB)

Altre Considerazioni – Tab Pagine inverse

- Utilizza una sola pagina con coppie <process id, numero di pagina>
- Però solo info su pagine in memoria principale
- Se page fault?

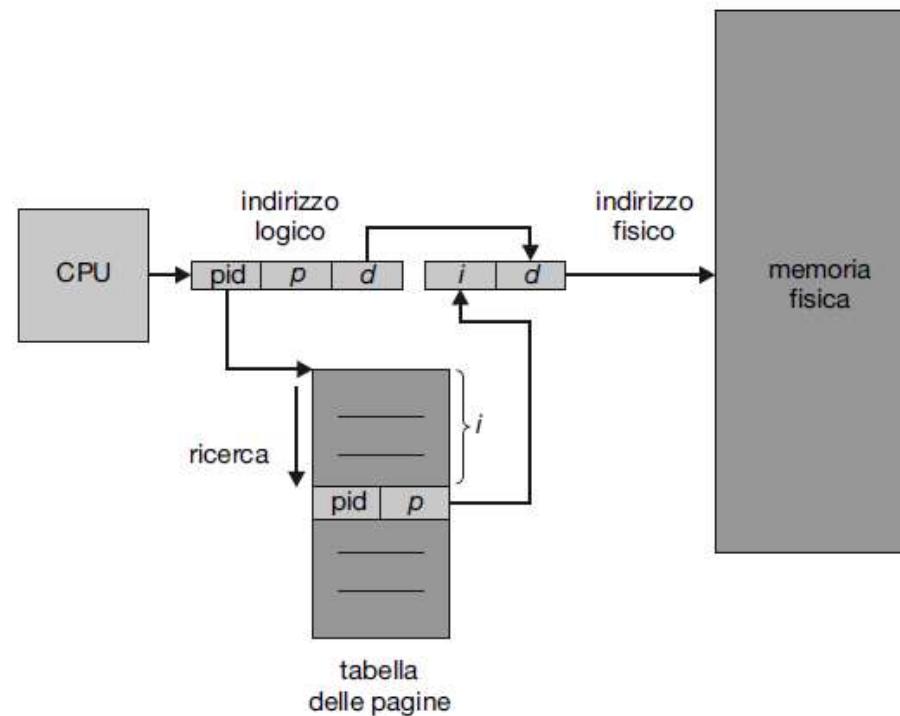


Figura 9.18 Tabella delle pagine invertita.

Altre Considerazioni – Tab Pagine inverse

- Utilizza una sola pagina con coppie <process id, numero di pagina>
- Però solo info su pagine in memoria principale
- Se page fault?
 - Non informazione completa su indirizzo logico della pagina referenziata
 - Demand paging richiede queste informazioni
 - Occorre leggerlo in una tabella anche questa paginata ...
 - Ogni processo ha una tabella delle pagine, ma serve solo sul page fault
 - ... ma anche la tabella può essere paginata e può fare page fault
 - Occorre una gestione attenta da parte del kernel

Altre Considerazioni – Struttura del Programma

- A volte il programmatore può migliorare la performance se conosce il modo in cui è gestita paginazione e demand paging
- Program structure
 - int [128,128] data;
 - Se ogni riga memorizzata in una pagina da 128 parole
 - ▶ Array memorizzato per riga, se il sistema alloca meno di 128 frame, page fault
 - Programma 1

```
for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i, j] = 0;
```

128 x 128 = 16,384 page faults

- Programma 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i, j] = 0;
```

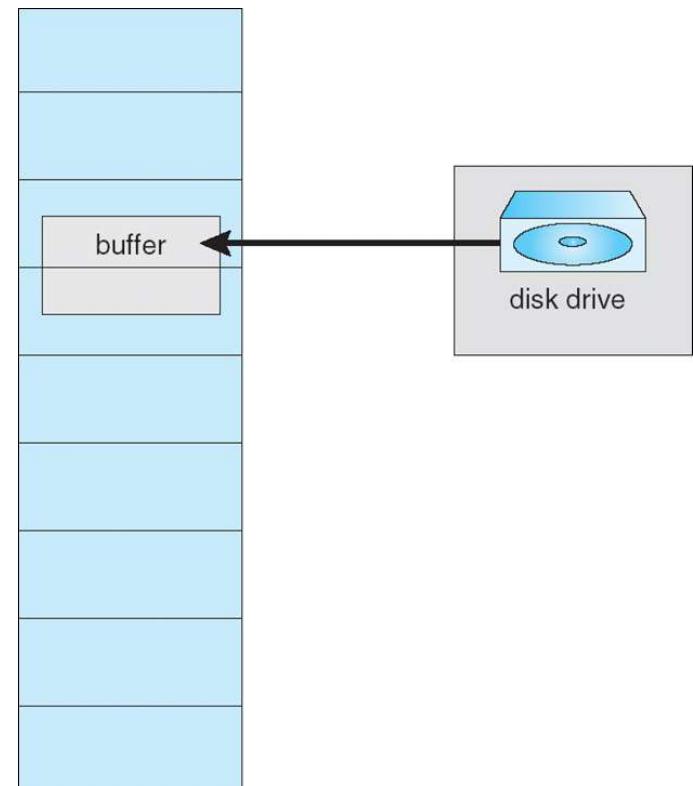
128 page faults

Altre Considerazioni – Struttura del Programma

- A volte il programmatore può gestire
- La scelta delle strutture dati può influire sul numero di page fault
 - Stack solitamente buona località perché accesso su top
 - Hash table invece solitamente località meno buona perché accesso sparso
- Compiler e loader può avere un impatto sul paging
 - Separazione tra codice e dati, uso di codice rientrante (pagine read-only), etc.
 - Loader carica routine non a cavallo di pagine, routine che si chiamano vicendevolmente vicine (possibilmente nella stessa pagina), etc.

Altre Considerazioni – I/O interlock

- **I/O Interlock** – le pagine qualche volta devono essere “locked” in memoria
- Nel caso di I/O - pagine usate per copiare un file da un dispositivo (es. USB) devono essere locked per evitare che siano selezionate da un algoritmo di page replacement
 - Es. Processo richiede I/O, altro processo schedulato, fa page fault e sottrae pagina in global replacement
- **Pinning** (fissare) pagine da tenere in memoria
- Lock bit per impedire il trasferimento di pagina
- Può essere usato anche durante il paging, es. per impedire che proc. ad alta priorità prenda pagine appena caricate a proc. a bassa priorità



Esempi di Sistemi Operativi

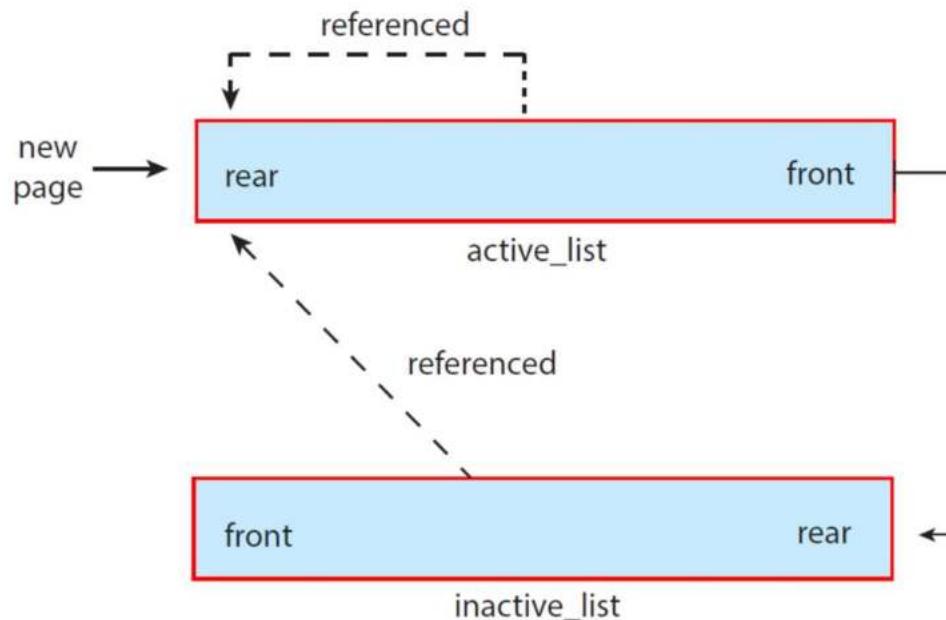
- Linux
- Windows
- Solaris

Linux

- Per memoria virtuale Linux usa demand paging con allocazione da lista di free frame
- Global page replacement con clock approx LRU
- Due liste di pagine **active_list** e **inactive_list**, le inactive sono elegibili per essere riusate

Linux

- Due liste di pagine **active_list** e **inactive_list**, le inactive sono elegibili per essere riusate
 - Un **accessed** bit per stabilire chi ha avuto un riferimento
 - Nuova pagina nel retro della active list (come ogni pagina con accesso in active)
 - Il bit è periodicamente resettato
 - Last Recently Used emerge sul fronte, sul retro retrocede all'inactive
 - Un demone kswapd periodicamente verifica se occorre liberare memoria, se la memoria libera va sotto soglia scorre la **inactive_list** e libera frame



Windows

- Windows 10 supporta 32 o 64 bit (IA-32, IA-64, ARM)
 - Windows 11 solo 64 bit
-
- Su 32-bit memoria virtuale per processo è 2GB (estesa a 3GB), memoria fisica fino a 4GB
 - Con 64-bit spazio indirizzi virtuale 128 TB e 24 TB di memoria fisica
 - Supporta demand paging, copy-on-write, paging, memory compression

Windows

- Usa un demand paging con **clustering**.
 - Il clustering porta in memoria pagine “vicine” alla pagina che ha generato il page fault
 - Nel caso di data page il cluster è di 3 (la pagina richiesta, quella prima e dopo), altrimenti il cluster è 7
- Elemento cruciale è la gestione del working set

Windows

□ Gestione del working set

- Ai processi sono assegnati **working set minimum** e **working set maximum**
- Working set minimum è il numero minimo di pagine che il processo è garantito avere in memoria (50 pagine)
- Al processo possono essere assegnate tante pagine quanto è il suo working set maximum (345 pagine)
- I valori possono essere ignorati a meno che non è settato **hard working set limit**
- Win utilizza un LRU approx. clock con global e local replacement
- Quando la memoria è sufficiente procede allocando free memory per servire il page fault
- Quando la quantità di free memory nel sistema va sotto una soglia **automatic working set trimming** è eseguito per recuperare la quantità di free memory
- Working set trimming rimuove pagine dai processi che hanno pagine in eccesso rispetto al loro working set minimum
- Preferiti processi grandi e non attivi rispetto a quelli piccolo e attivi
- Continua finché non si raggiunge la soglia (anche sottraendo sotto il min ws)
- Trimming sia su processi utente che di sistema

Solaris

- Mantiene una lista di free pages da assegnare ai processi richiedenti
- **Lotsfree** – parametro soglia (quantità di free memory) per iniziare il paging (1/64 della dimensione della memoria fisica)
- **Desfree** – parametro soglia di free memory per incrementare il paging
- Scansione 4 volte al secondo per verificare la soglia,
- Se sotto soglia **Lotsfree** inizia il processo di **pageout**

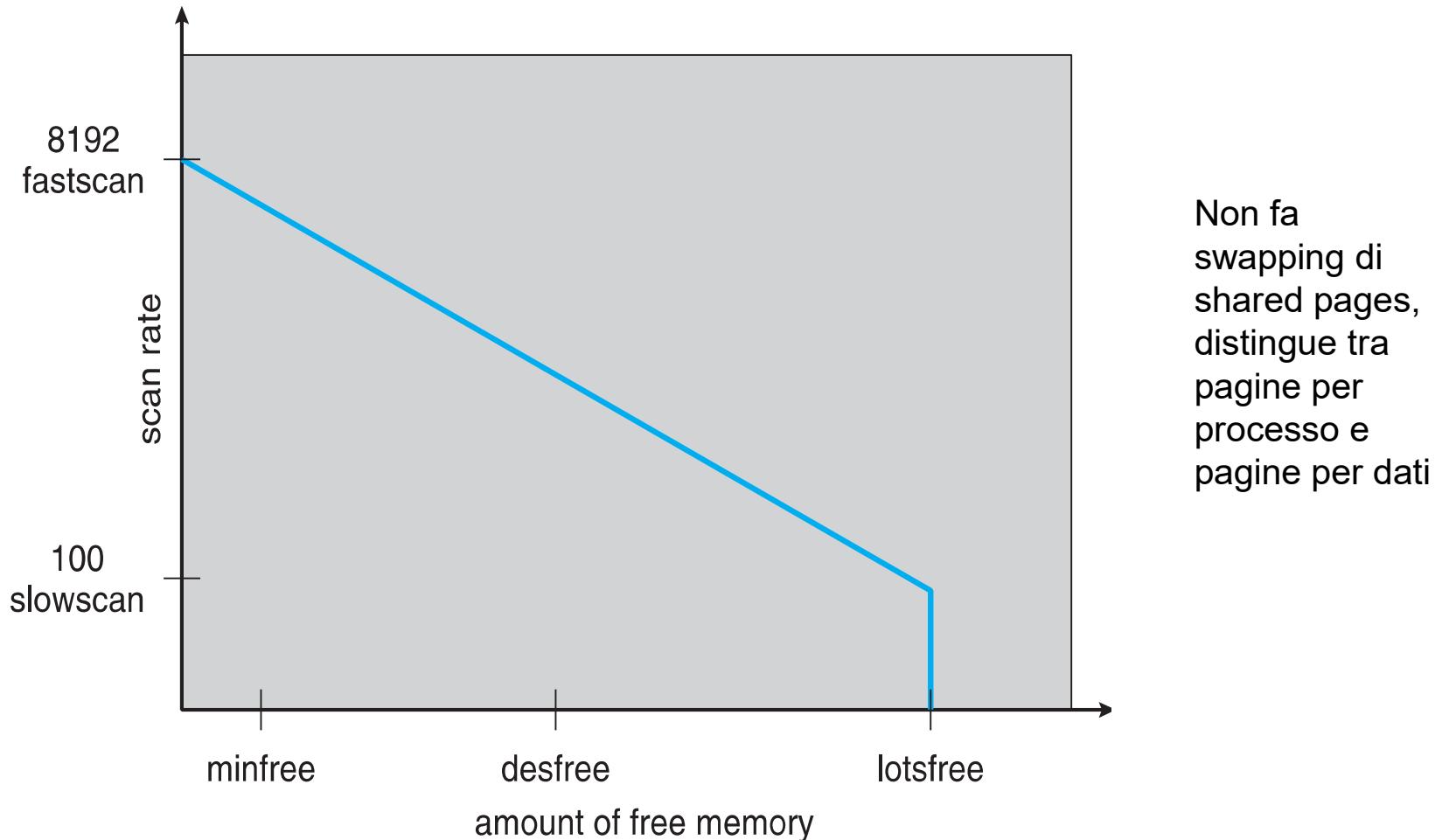
- **Pageout** scansiona le pagine usando un algoritmo second chance modificato
 - **Front hand**: scansione delle pagine azzerando il reference bit
 - **Back hand**: assegna le pagine non referenziate alla free list e la copia in backing store
 - La distanza tra front e back è di **handspear**
 - Una pagina non riassegnata ma con riaccesso può essere recuperate

Solaris

- **Scanrate** è la frequenza di scansione delle pagine.
 - Varia da **slowscan** a **fastscan**
 - ... da 100 pagine per secondo a $\text{mem-fisica}/2$ per secondo
- **Pageout** è chiamata più frequentemente in dipendenza dall'ammontare della free memory disponibile
- Il tempo di scansione dipende da **scanrate** e **handspeard** ...
 - Se **scanrate** è 100 pagine al secondo e **handspeard** è 1024 allora possono passare 10 secondi tra front hand e back hand ...
 - Nei sistemi reali il tempo tra front e back hand è di pochi secondi

Solaris

- **Pageout** fa il check di memoria 4 volte al secondo, se **desfree** è sotto soglia incrementa a 100 volte al secondo
- Se il SO non riesce a tenere **desfree** (per 30 sec) allora inizia lo swapping
- Sotto **minfree** fa **pageout** per ogni richiesta di nuova pagina



Sistemi Operativi I

Corso di Laurea in Informatica

A.A. 2021-2022

Alberto Finzi

Informazioni Generali

- Crediti: 9 CFU
- Orario:
 - Lunedì: 16:30-18:30 aula G3
 - Martedì: 11:00-13:00 aula A6
 - Giovedì: 08:30-10:30 aula A6
- Gruppo 1:
 - Studenti aventi il cognome con iniziali tra **A e G**

Informazioni Generali

- **Propedeuticità ingresso:**
 - Programmazione I, Architettura degli Elaboratori
- **Propedeuticità uscita:**
 - Laboratorio di Sistemi Operativi, Reti di calcolatori I, Operating Systems for Mobile, Cloud and IoT

Docente

- Docente: Alberto Finzi
- Studio: via Claudio 21, 80125 Napoli
- Ricevimento:
 - Mercoledì 16:30-18:30 (Finzi), Teams
- Email: alberto.finzi@unina.it
 - Specificare SEMPRE nel subject “SO1”
- Sito web del corso
 - <http://wpage.unina.it/alberto.finzi/didattica/>

Obiettivi del Corso

Struttura e funzioni dei moderni **Sistemi Operativi**:

- Principi, componenti fondamentali, metodologie di progettazione e di sviluppo, algoritmi e strumenti di base
- Abilità di base nell'uso di una piattaforma a livello utente ed amministratore, principi di scripting e programmazione di sistema
- Particolare riferimento ai sistemi Unix e Linux

Modalità di Esame

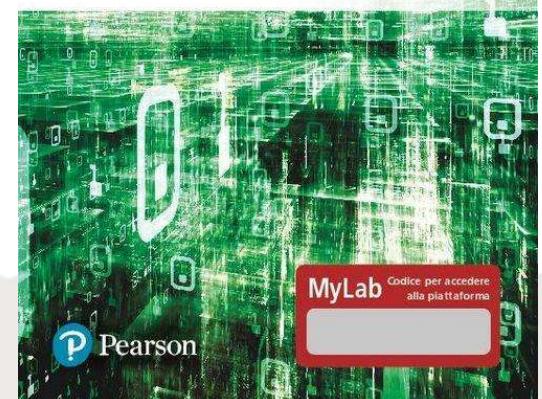
- Prova scritta
 - Domande aperte ed esercizi
- Discussione orale
 - Esercizi
 - Discussione della prova scritta

Libri di Testo

- *Silberschatz, Galvin, Gagne: Sistemi operativi. Concetti ed esempi, decima edizione, Pearson Education Italia, 2019*

oppure

- *Tanenbaum, Bos: I moderni sistemi operativi, quarta edizione, Pearson Education Italia, 2019*
- Anche edizioni precedenti e/o in inglese
- Documenti segnalati a lezione e sul sito web.



Programma di Massima

- Introduzione ai Sistemi Operativi
 - Evoluzione, strutture, architetture, componenti
- Processi
 - Il concetto di processo, stati dei processi, funzioni del kernel, algoritmi di schedulazione, sincronizzazione dei processi e deadlock
- Memoria
 - Gestione memoria principale, swapping, partizione, segmentazione e paginazione, memoria virtuale
- Sistemi I/O
 - Architetture e dispositivi di I/O, interfacce, sottosistema per l'I/O del nucleo, etc.
- Dati permanenti
 - File, metodi di allocazione, directory e metodi di accesso, file system, etc.
- Sistemi distribuiti

Lezione 3: Processi

Obiettivi

- Nozione di processo
- Caratteristiche dei processi (schedulazione, creazione, terminazione, comunicazione, etc.)
- Comunicazione interprocesso
- Comunicazione nei sistemi client-server

Definizione di Processo

- Un SO esegue programmi di varia natura:
 - Sistemi batch – **job**
 - Sistemi time-shared – **programmi utente** o **task**
 - I libri di testo usano **job** e **processo** in modo quasi intercambiabile
- **Processo** – è un programma in esecuzione
 - Un programma è un'entità *passiva* archiviata su disco (**file eseguibile**), un processo è *attivo (in esecuzione)*
 - Unità di attività computazionale coerente di un moderno SO
 - Esecuzione sequenziale
 - Un processo richiede risorse (CPU, memoria, file, dispositivi I/O) per l'esecuzione di un compito (**task**)
 - Un programma può chiamare molti processi
 - I sistemi sono il risultato dell'esecuzione di più processi concorrenti
 - ▶ processi di sistema e processi utente

Definizione di Processo

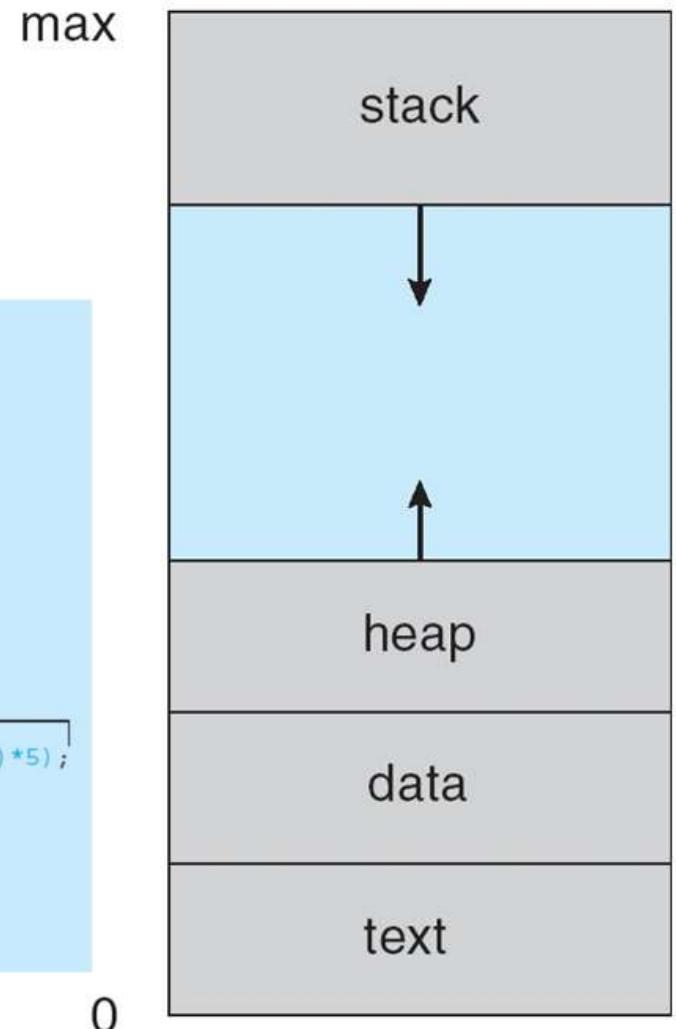
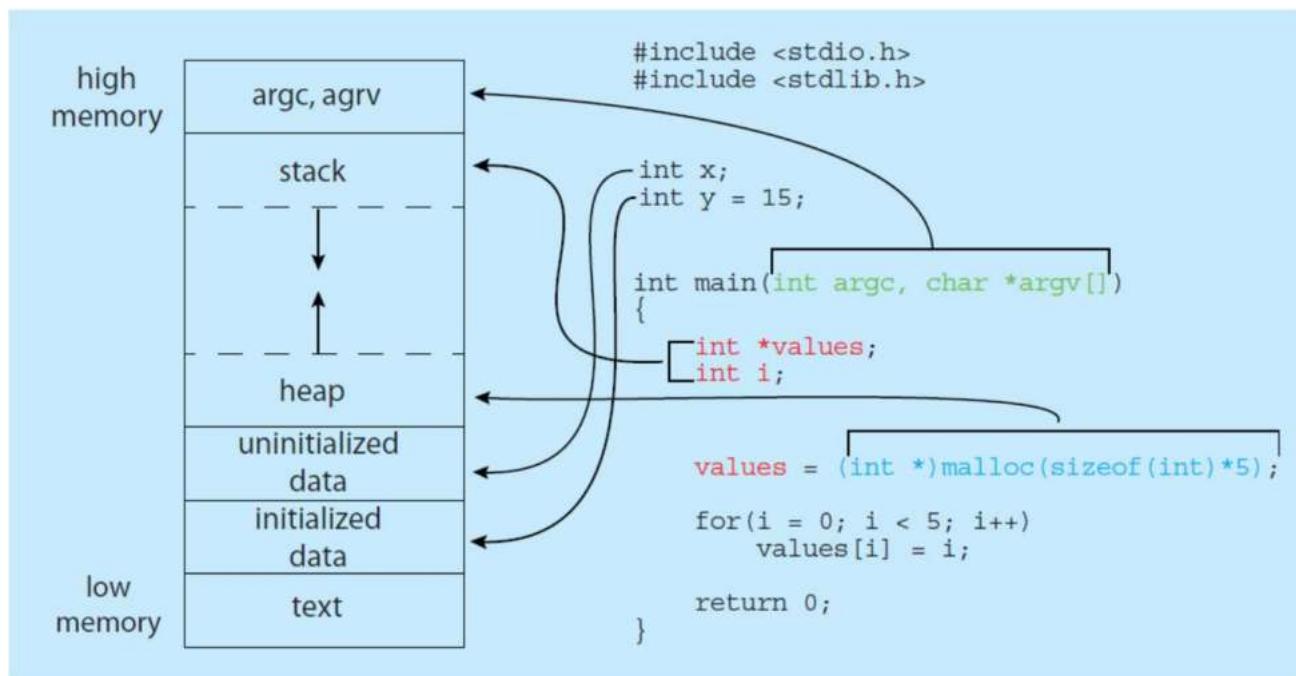
- **Processo** – è un programma in esecuzione:
 - Stato corrente **program counter** e **registri del processore**
 - Layout di memoria
- Layout di memoria:
 - Il codice del programma, chiamata **sezione testo**
 - **Sezione Dati** contenente le variabili globali
 - **Stack** contenente i dati temporanei
 - Parameteri di funzione, indirizzi di return, variabili locali
 - **Heap** contenente memoria dinamicamente allocata a run time durante l'esecuzione di un task

Processo Allocato in Memoria

- **Processo** – è un programma in esecuzione:

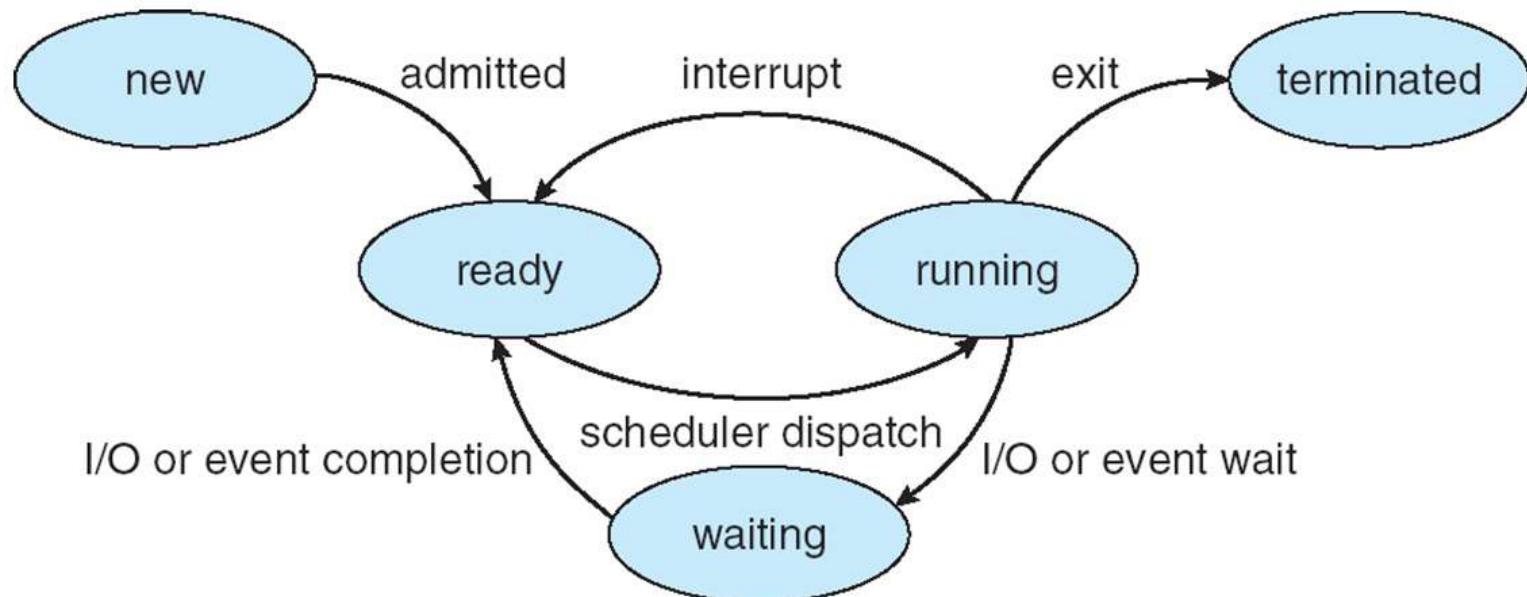
- Layout di memoria:

- Text e Data dimensione fissata
- Stack e Heap variabili a run time



Stato di un Processo

- Durante l'esecuzione un processo può cambiare il suo **stato**
 - **new**: il processo è stato creato
 - **running**: esecuzione delle istruzioni del processo
 - **waiting**: il processo è in attesa di un evento per proseguire (es. I/O)
 - **ready**: il processo è in attesa di essere assegnato a processore
 - **terminated**: il processo ha finite l'esecuzione



Descrittore di Processo

- Il Sistema Operativo mantiene una **Tabella dei processi**
- Per ogni processo una entry (**task control block**)
 - Stato del processo - running, waiting, etc
 - IDs del processo - identificativi di processo
 - Program counter - locazione prossima istruzione
 - Registri CPU - contenuto registri CPU
 - Scheduling di CPU - priorità, parametri di scheduling
 - Memoria - memoria allocata per il processo
 - Contabilità - CPU usata, tempo clock dallo start, etc.
 - Stato I/O - dispositivi I/O allocati, lista di file aperti, etc.

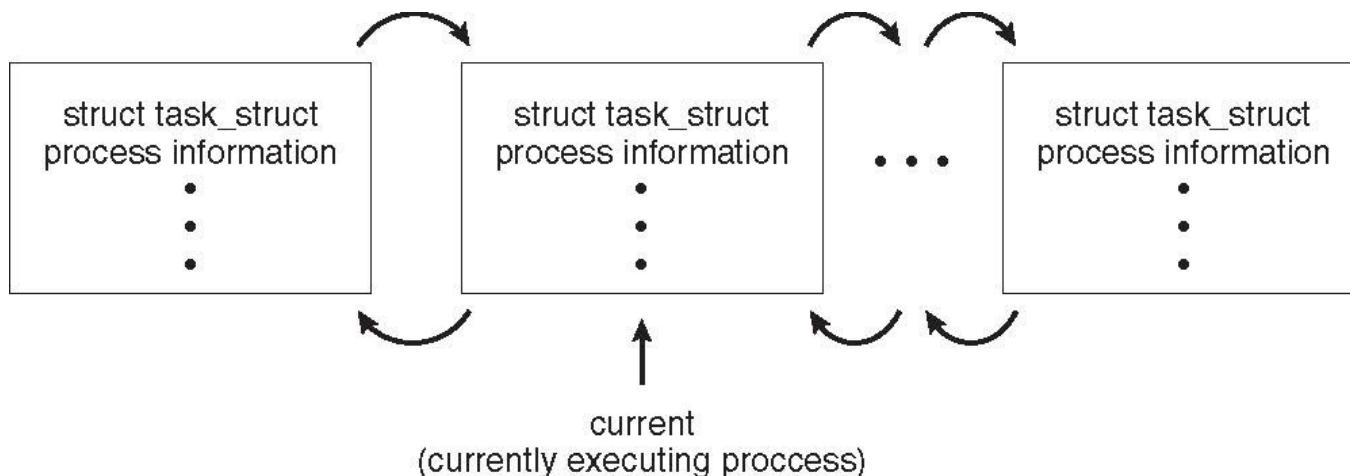
| |
|--------------------|
| process state |
| process number |
| program counter |
| registers |
| memory limits |
| list of open files |
| • • • |

Tabella Processi in Linux

In Linux la PCB è rappresentata in C da una `task_struct` specificata nel codice sorgente del kernel (in `/linux/sched.h`)

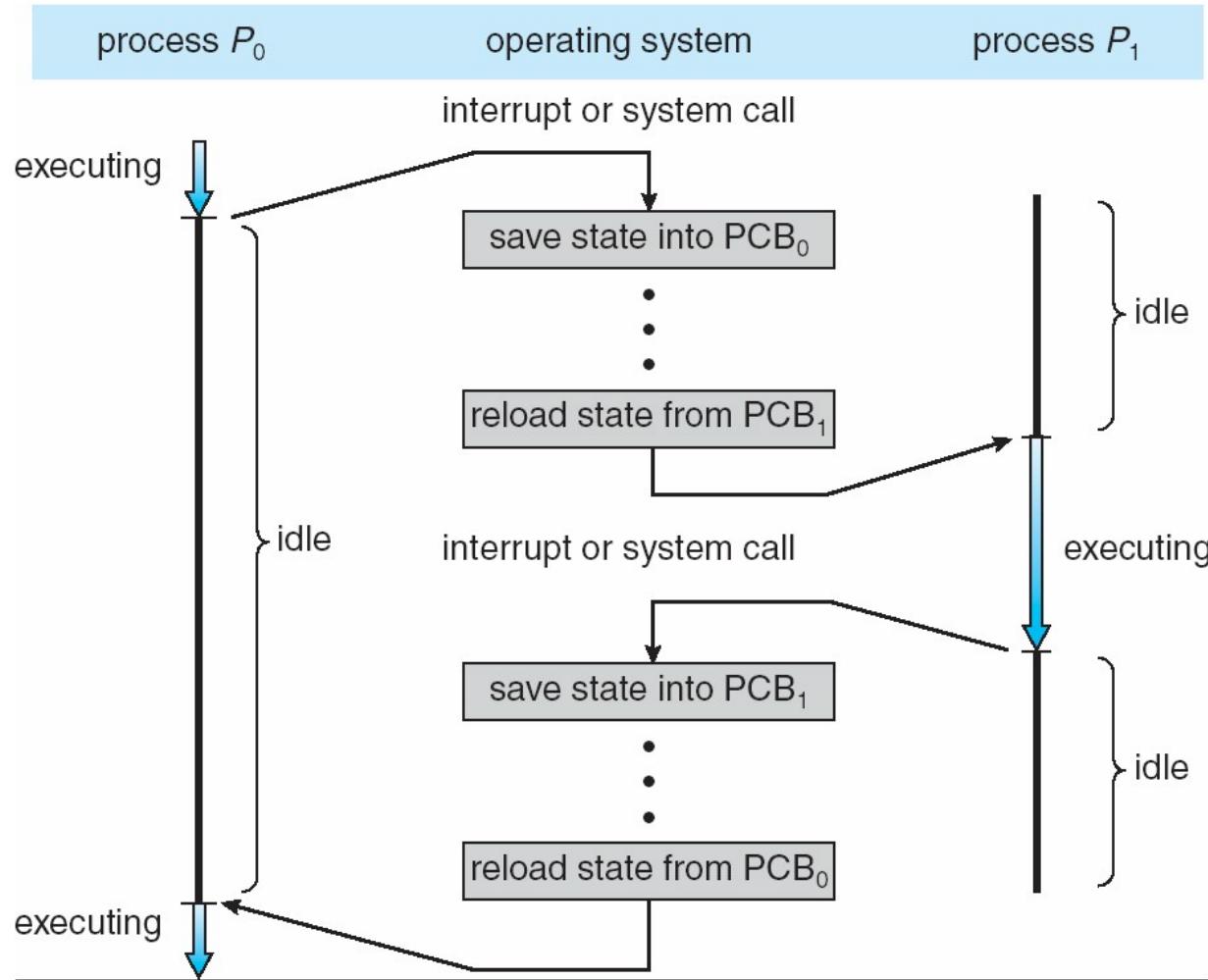
Alcuni campi seguono:

```
pid t_pid; /* process identifier */  
long state; /* state of the process */  
unsigned int time_slice /* scheduling information */  
struct task_struct *parent; /* this process's parent */  
struct list_head children; /* this process's children */  
struct files_struct *files; /* list of open files */  
struct mm_struct *mm; /* address space of this process */
```



Commutazione tra Processi

- La tabella dei processi supporta la commutazione tra processi
 - Lo stato dei processi salvato e ripristinato durante gli switch

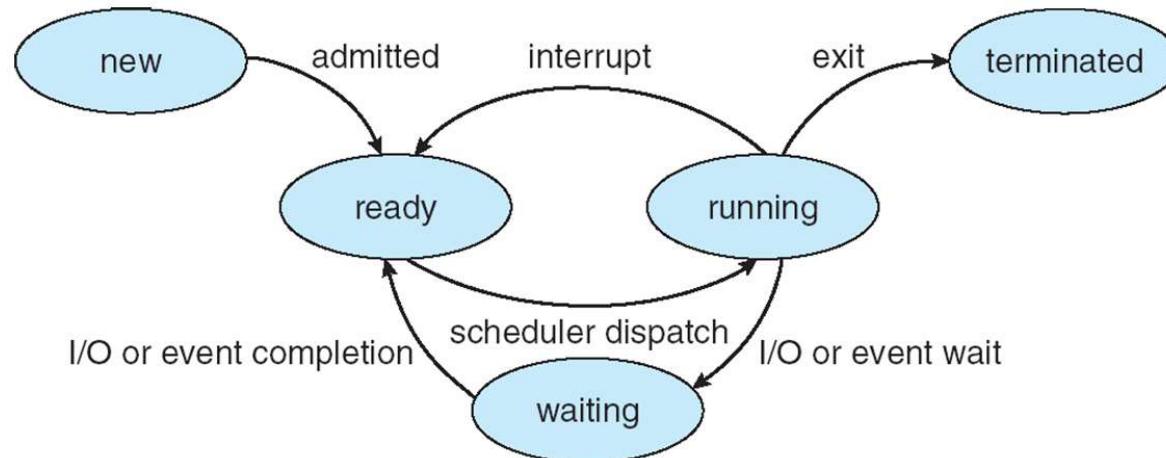


Processi e Thread

- I moderni Sistemi Operativi consentono ai processi di gestire l'esecuzione di più flussi di controllo (**thread**)
- I processi possono diramarsi in più sottounità di esecuzione
 - Più program counter per un processo
 - Multipli flussi di controllo -> threads
 - ▶ Condividono identità e risorse di processo ma hanno diverso stato esecutivo
 - Archiviazioni di ulteriori informazione per thread in PCB

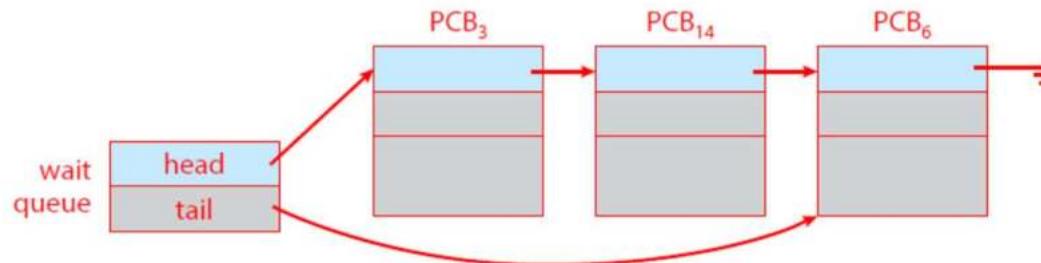
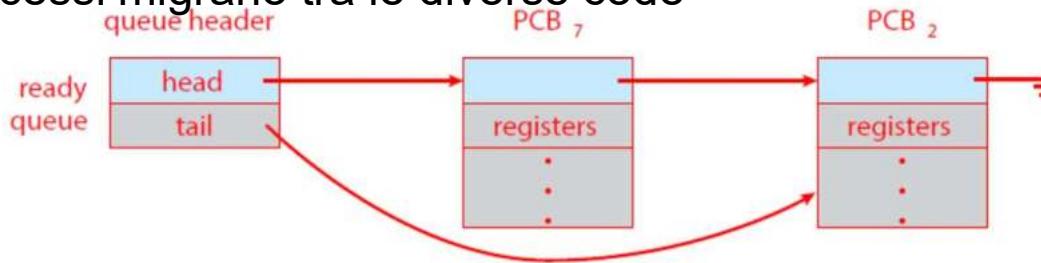
Schedulazione di Processi

- Massimizza l'uso della CPU, fa velocemente lo switch di processi nella CPU per il time sharing
- **Schedulatore dei processi** seleziona tra processi disponibili per la prossima esecuzione su CPU
- Mantiene la **coda di scheduling** dei processi
 - **Job queue** – insieme di tutti i processi nel sistema
 - **Ready queue** – insieme di tutti i processi in memoria principale, pronti (ready)
 - **Wait queue** – insieme dei processi in attesa (waiting)
 - **Device queues** – insieme di processi in attesa (waiting) di un dispositivo di I/O
 - I processi migrano tra le diverse code



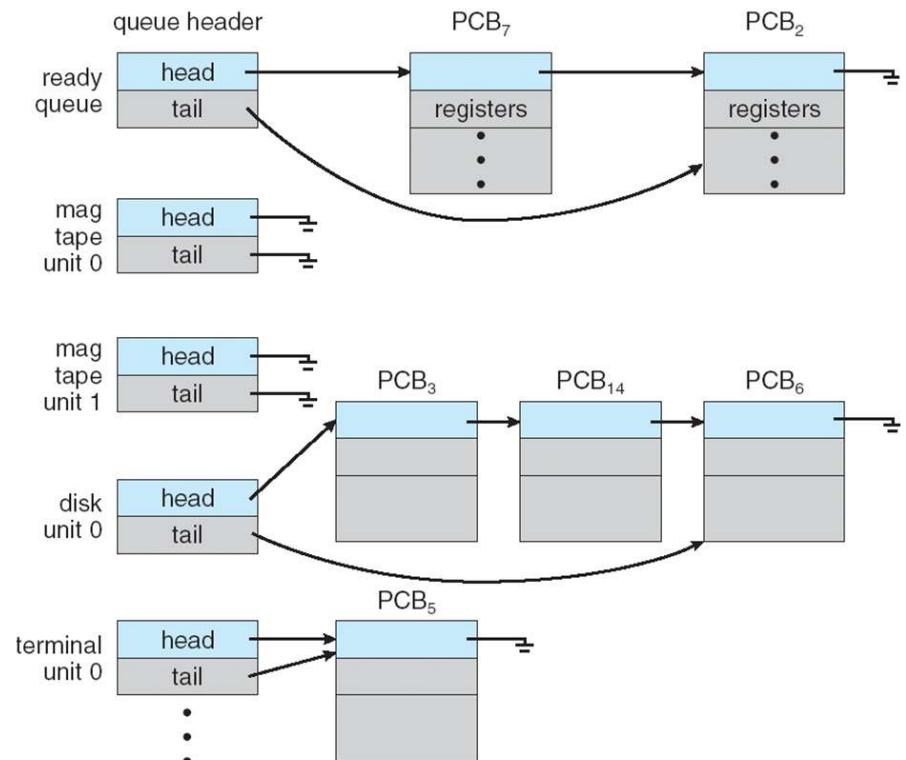
Code Processi Ready e Waiting

- Mantiene la **coda di scheduling** dei processi
 - **Job queue** – insieme di tutti i processi nel sistema
 - **Ready queue** – insieme di tutti i processi in memoria principale, pronti (ready) per l'esecuzione
 - **Wait queue** – insieme dei processi in attesa (waiting)
 - **Device queues** – insieme di processi in attesa (waiting) di un dispositivo di I/O
 - I processi migrano tra le diverse code



Code Processi Ready e Waiting

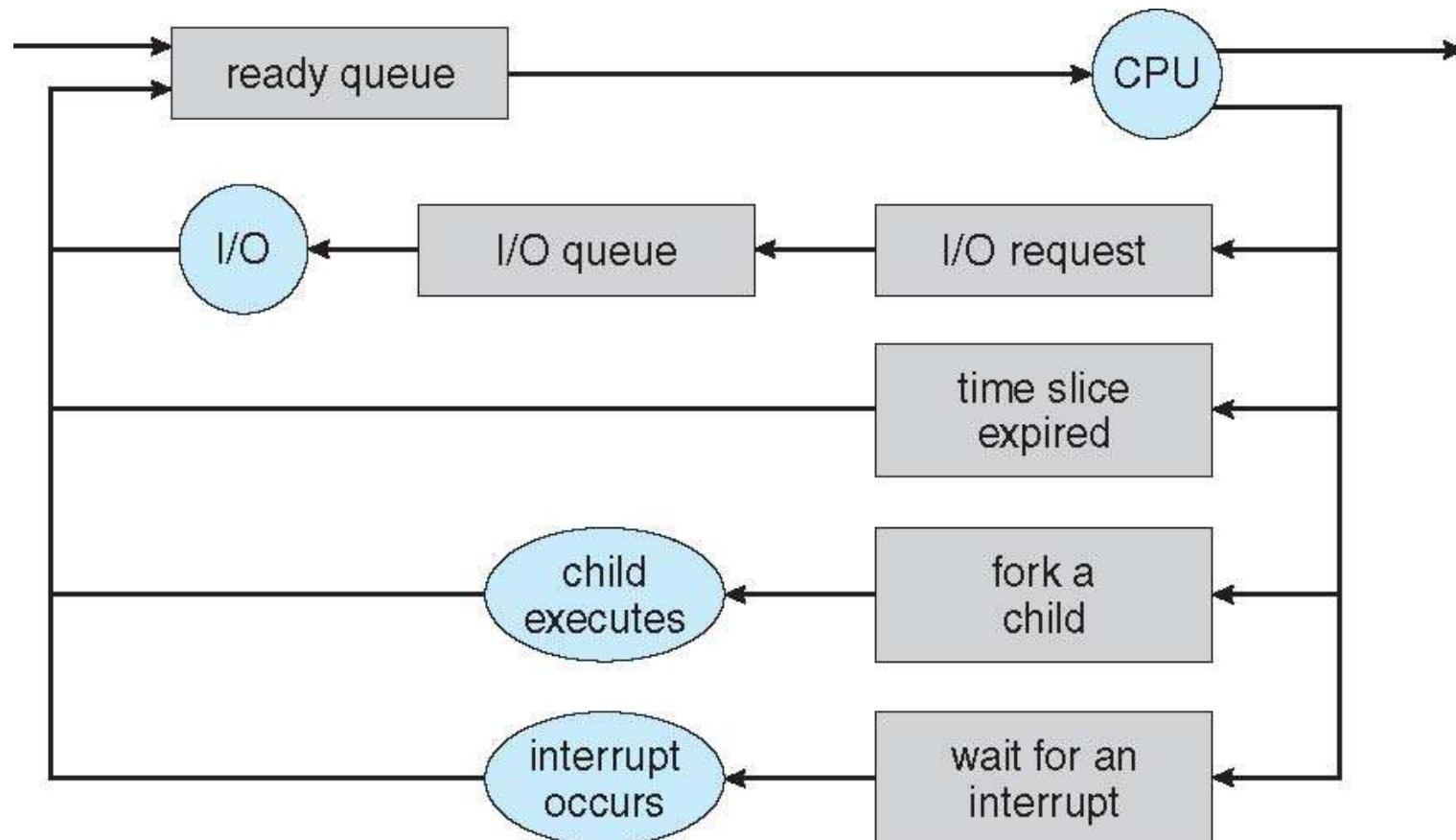
- Mantiene la **coda di scheduling** dei processi
 - **Job queue** – insieme di tutti i processi nel sistema
 - **Ready queue** – insieme di tutti i processi in memoria principale, pronti (ready) per l'esecuzione
 - **Wait queue** – insieme dei processi in attesa (waiting)
 - **Device queues** – insieme di processi in attesa (waiting) di un dispositivo di I/O
 - I processi migrano tra le diverse code



Rappresentazione Scheduling dei Processi

□ Diagramma di accodamento

- Un nuovo processo in coda ready attende la CPU
- Una volta allocata diversi eventi possono avvenire (I/O, fine tempo, fork, wait)

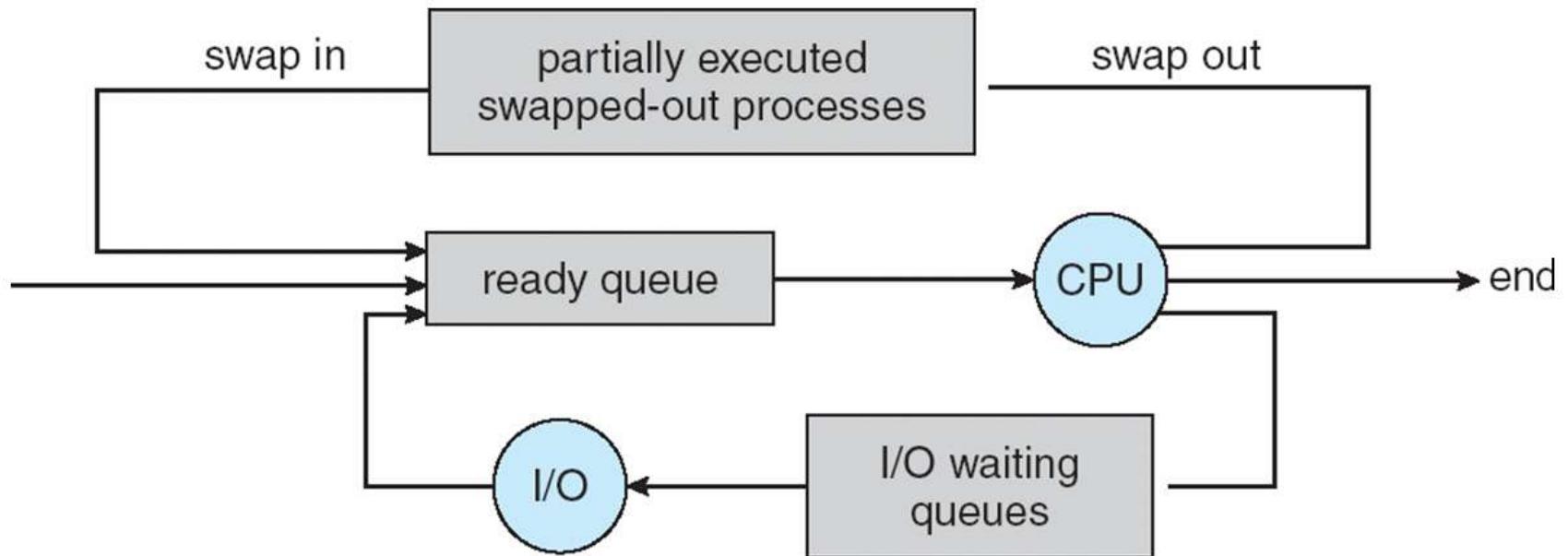


Scheduler

- **Short-term scheduler** (o **CPU scheduler**) – seleziona quale processo deve essere eseguito dalla CPU
 - A volte l'unico scheduler nel sistema
 - Lo short-term scheduler è invocato frequentemente (millisecondi) \Rightarrow (deve essere veloce)
- **Long-term scheduler** (o **job scheduler**) – seleziona quale processo deve essere portato nella coda ready
 - Long-term scheduler non frequente (secondi, minuti) \Rightarrow (può essere lento)
 - Il long-term scheduler controlla il **grado di multiprogrammazione**
 - I processi possono essere descritti come:
 - ▶ **I/O-bound process** – spende più tempo in operazioni I/O che in computazione, piccoli e brevi accessi in CPU
 - ▶ **CPU-bound process** – spende più tempo in computazione; pochi e lunghi accessi in CPU
 - ▶ Long-term scheduler cerca di trovare una buona combinazione di processi

Scheduling di Medio Termine

- **Medium-term scheduler** può essere aggiunto se il grado di multiprogrammazione deve decrescere (minor contesa per CPU)
 - Toglie processi dalla memoria, archivia su disco, riporta in memoria da disco per continuare l'esecuzione: **swapping**



Context Switch

- Al passaggio della CPU ad un altro processo il sistema deve **salvare lo stato** del vecchio processo e caricare lo **stato salvato** del nuovo processo attraverso un **context switch**
- **Context** (contesto) di processo rappresentato in PCB
- Il tempo di context-switch è un **overhead**
 - il sistema non fa lavoro “utile” durante lo switching
 - Più è complesso il sistema e la PCB più è lungo il context switch
- Il tempo dipende dal supporto hardware
 - Alcuni hardware forniscono registri multipli per CPU consentendo caricamenti di più context in una volta

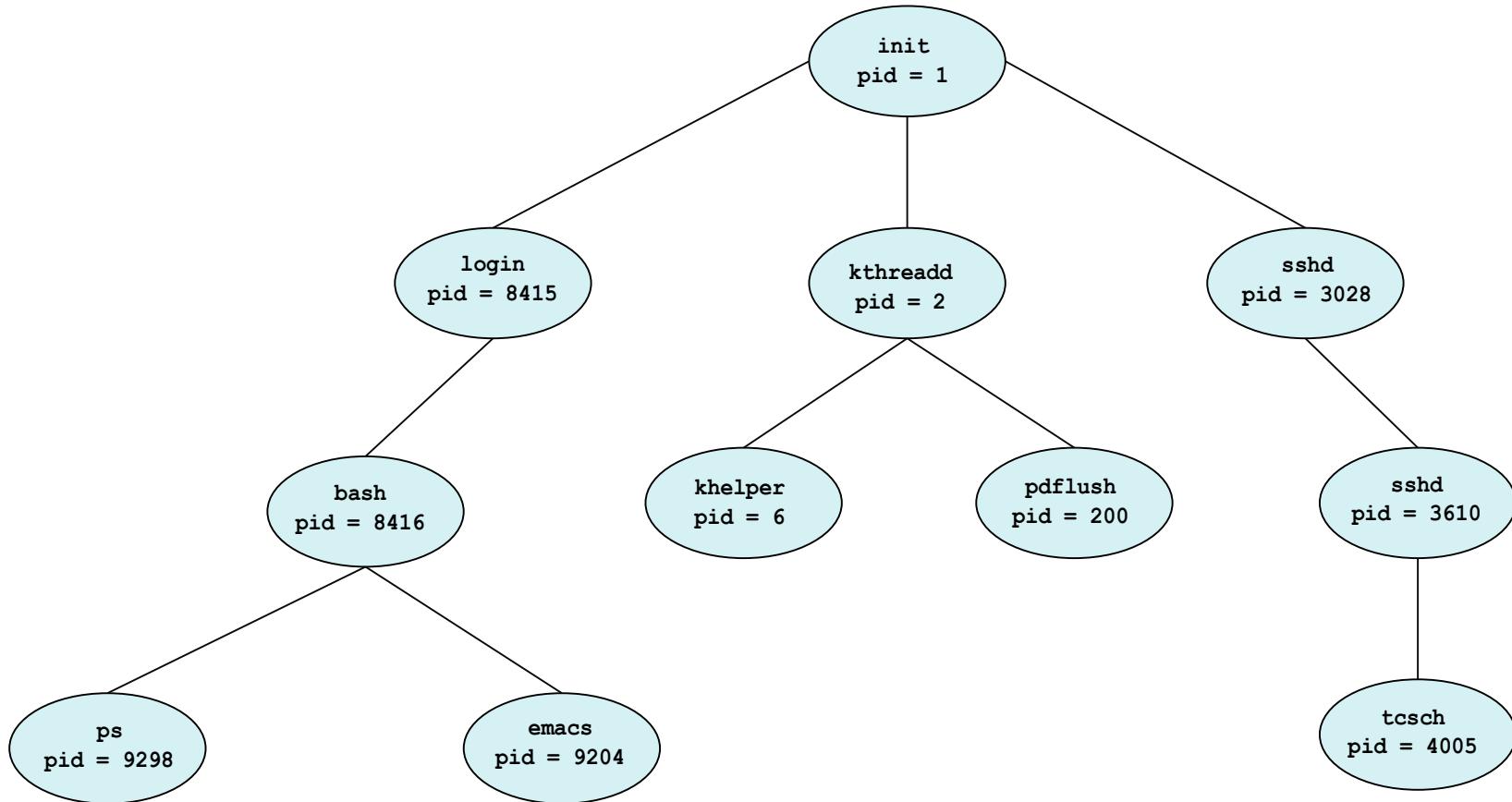
Operazioni su Processi

- Il Sistema deve fornire meccanismi per:
 - Creazione di processi,
 - Terminazione di processi,
 - Comunicazione tra processi
 - Sincronizzazione tra processi
 - Altri meccanismi di gestione di processi (dettagliato dopo)

Creazione di Processi

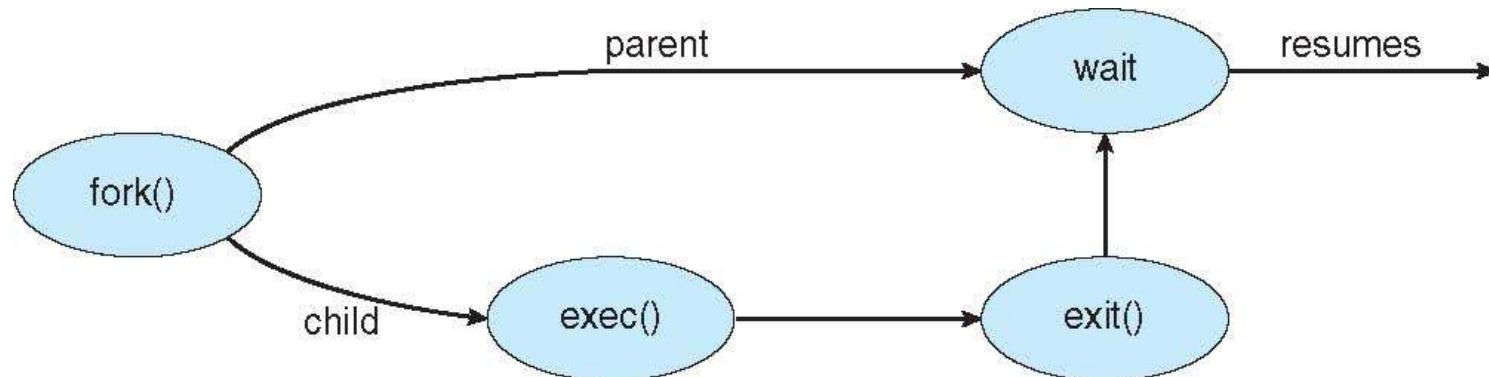
- Processo **padre** crea un processo **figlio**, che a sua volta crea altri processi formando un **albero** di processi
- Processi identificati e gestiti con **identificatori di processo (pid)**
- Condivisione di risorse (opzioni)
 - Padre e figlio condividono tutte le risorse
 - Figlio condivide un sottoinsieme delle risorse del padre
 - Padre e figlio non condividono alcuna risorsa
- Opzioni di esecuzione
 - Padre e figlio vengono eseguiti in concorrenza
 - Padre aspetta che il figlio finisca

Albero di Processi in Linux



Creazione di Processi

- Spazio memoria
 - Figlio duplica quello del padre
 - Figlio ha un programma caricato
- Esempi UNIX
 - `fork()` system call crea nuovo processo
 - `exec()` system call usata dopo la `fork()` per sostituire lo spazio di memoria con un nuovo programma



Programma C con fork ed exec

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```

Creare Processo con Windows API

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

/* allocate memory */
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
ZeroMemory(&pi, sizeof(pi));

/* create child process */
if (!CreateProcess(NULL, /* use command line */
"C:\\WINDOWS\\system32\\mspaint.exe", /* command */
NULL, /* don't inherit process handle */
NULL, /* don't inherit thread handle */
FALSE, /* disable handle inheritance */
0, /* no creation flags */
NULL, /* use parent's environment block */
NULL, /* use parent's existing directory */
&si,
&pi))
{
    fprintf(stderr, "Create Process Failed");
    return -1;
}
/* parent will wait for the child to complete */
WaitForSingleObject(pi.hProcess, INFINITE);
printf("Child Complete");

/* close handles */
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
}
```

Programma C con fork

```
int main(void) {  
    int i;  
  
    for (i=0; i<2 ;i++)  
        if (fork()>0) {  
            printf("Padre! %d\n", i);  
        } else {  
            printf("Figlio! %d\n", i);  
        }  
  
    sleep(10);  
    return 0;  
}
```

- Qual è l'output di questo programma?
- Quanti processi vengono creati?
- Di chi è figlio ciascun processo creato?

Programma C con fork

./a.out

padre 0

padre 1

padre 0

figlio! 0

figlio! 0

padre1 figlio! 1

padre1 figlio! 1

padre 1

figlio! 1

figlio! 1

Terminazione Processo

- I processi eseguono le ultime istruzioni e poi chiedono all'SO di uscire con la call di sistema `exit()`
 - Ritorna i dati di status data dal figlio al padre (via `wait()`)
 - Le risorse sono deallocate dall'SO
- Padri possono terminare le esecuzioni dei processi figli con la call `abort()` ad esempio perché:
 - Il figlio ha allocato risorse in eccesso
 - Il task del figlio non è più richiesto
 - Il genitore esce e non consente a un figlio di continuare

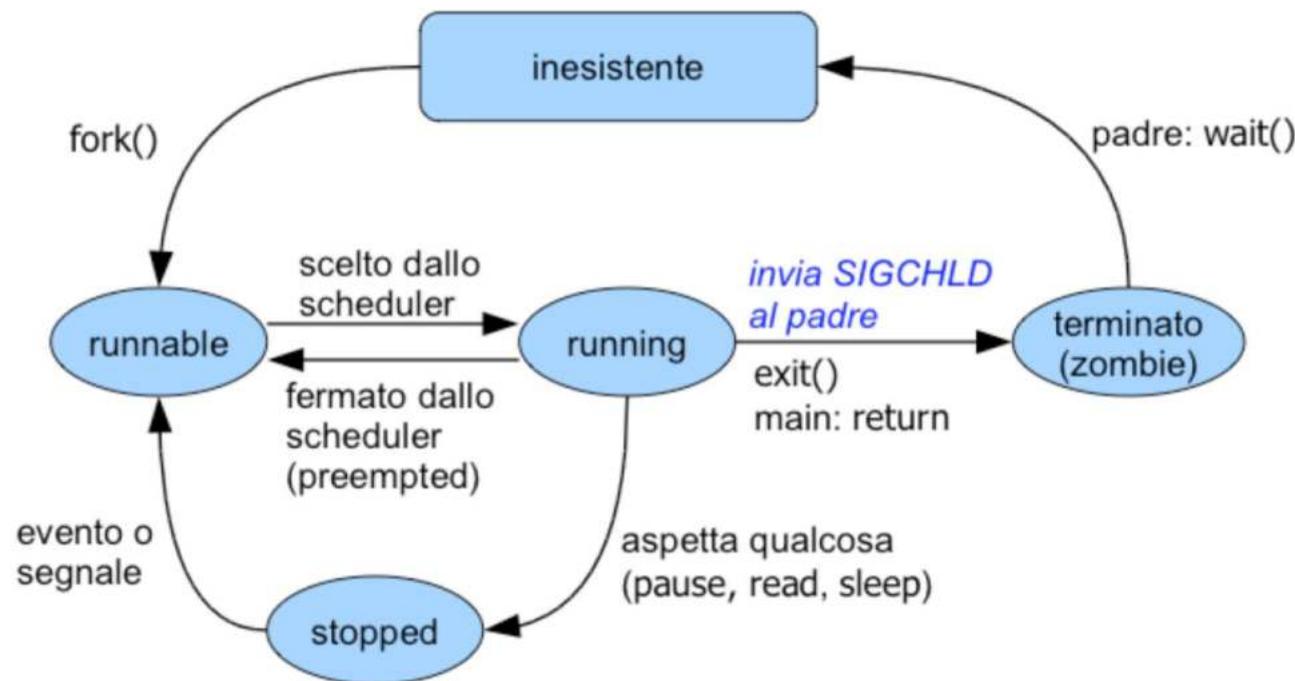
Terminazione Processo

- Alcuni SO non permettono ai figli di esistere se i padri hanno terminato
 - Se un processo termina tutti i figli terminano
 - **cascading termination** (tutta la gerarchia di figli termina)
 - la terminazione è gestita dal SO
- Il padre può aspettare la terminazione di un figlio con la call **wait()**
 - riceve informazioni di stato e il pid del processo terminato

```
pid = wait(&status);
```
- Se non c'è padre in waiting il figlio diventa **zombie**
- Se padre termina prima del figlio senza invocare **wait** il figlio è **orphan**
 - In UNIX adottato da init che chiama continuamente wait per accogliere lo status (in linux init è oggi systemd – System Daemon)

Terminazione Processo

□ In UNIX/Linux



Terminazione Processo

□ In UNIX/Linux

/* [status.c](#): Fornisce un esempio dei diversi stati di uscita
di un processo figlio */

```
#include<sys/types.h> /* per il tipo pid_t */
#include <unistd.h>    /* per la funzione fork */
#include<sys/wait.h>   /* per la funzione waitpid */
#include <stdio.h>      /* per le funz. printf, fgets e perror */
#include <stdlib.h>     /* per la funzione exit */
#include <string.h>     /* per la funzione strlen */
#include <errno.h>      /* per la messaggistica di errore */

int main(void)
{
    pid_t pid;
    int status;

    if ( (pid = fork()) < 0)
        perror("fork"), exit(1);
    else if (pid == 0) /* primo figlio...*/
        exit(7);       /* che termina in modo normale */

    if (wait(&status) != pid) /* il genitore aspetta il figlio... */
        perror("wait"), exit(1);
    print_exit(status); /* e ne mostra lo stato di terminazione*/
}
```

```
if ( (pid = fork()) < 0)
    perror("fork"), exit(1);
else if (pid == 0) /* secondo figlio... */
    abort();          /* che genera SIGABRT */

if (wait(&status) != pid) /* il genitore aspetta il 2ndo figlio... */
    perror("fork"), exit(1);
print_exit(status); /* e ne mostra lo stato di terminazione */

if ( (pid = fork()) < 0)
    perror("wait"), exit(1);
else if (pid == 0) /* terzo figlio.... */
    status /= 0;      /* che divide per 0 e genera SIGFPE */

if (wait(&status) != pid) /* il genitore aspetta il terzo figlio... */
    perror("wait"), exit(1);
print_exit(status); /* e ne mostra lo stato di terminazione */

exit(0);
}
```

Terminazione Processo

□ In UNIX/Linux

□ Stato di uscita

```
int status;
```

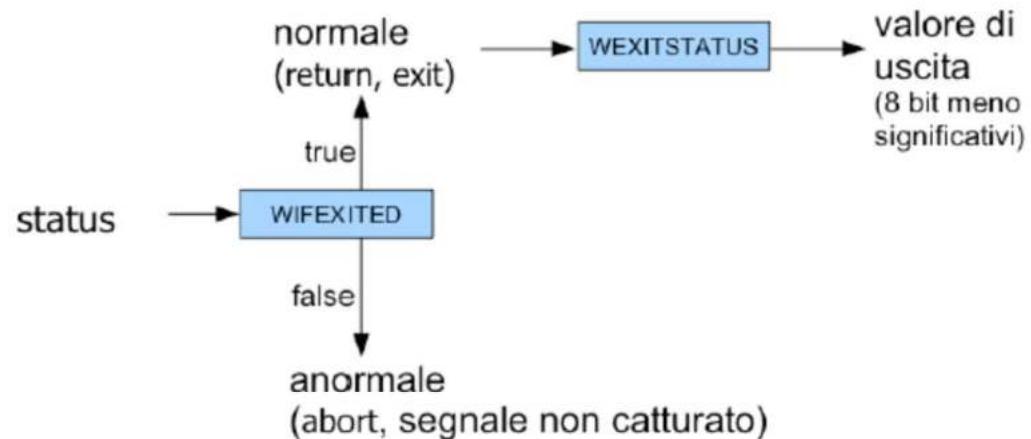
```
wait(&status);
```

```
if ( WIFEXITED(status) )
```

```
    printf("valore di uscita: %d\n", WEXITSTATUS(status));
```

```
else
```

```
    printf("terminazione anomala\n");
```



Architetture Multiprocesso – Chrome Browser

- Molti web browser giravano come un singolo processo (alcuni ancora)
 - Se un web site causa problem il browser può crashare o stallare
- Google Chrome Browser è multiprocesso con 3 differenti tipi di processo:
 - **Browser** che gestisce l'interfaccia utente, il disco e la rete I/O
 - **Renderer** che rende graficamente le web pages, gestisce HTML, Javascript. Per ogni sito web aperto è lanciato un nuovo renderer
 - **Plug-in** per ogni tipo di plug-in

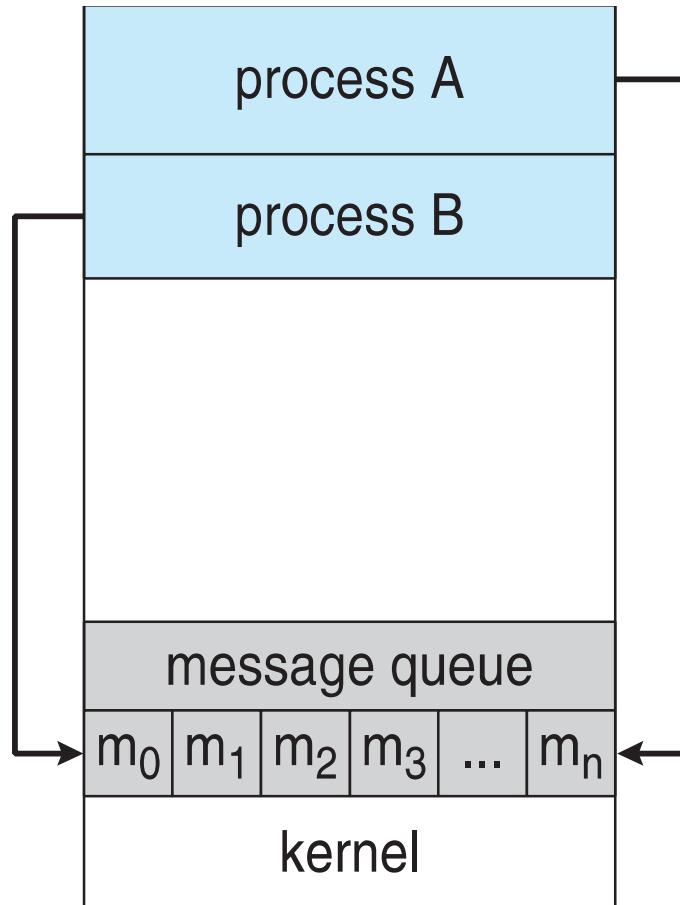


Comunicazione Interprocesso

- Processi possono essere *independenti* o *cooperativi*
- Processi cooperanti si influenzano scambiandosi dati
 - Ragioni per cooperazione:
 - Condivisione di informazione
 - Speed-up computazionale
 - Modularità
- Processi cooperanti hanno bisogno di **interprocess communication (IPC)**
 - Due modelli principali di IPC
 - Shared memory
 - Message passing

Modelli di Comunicazione

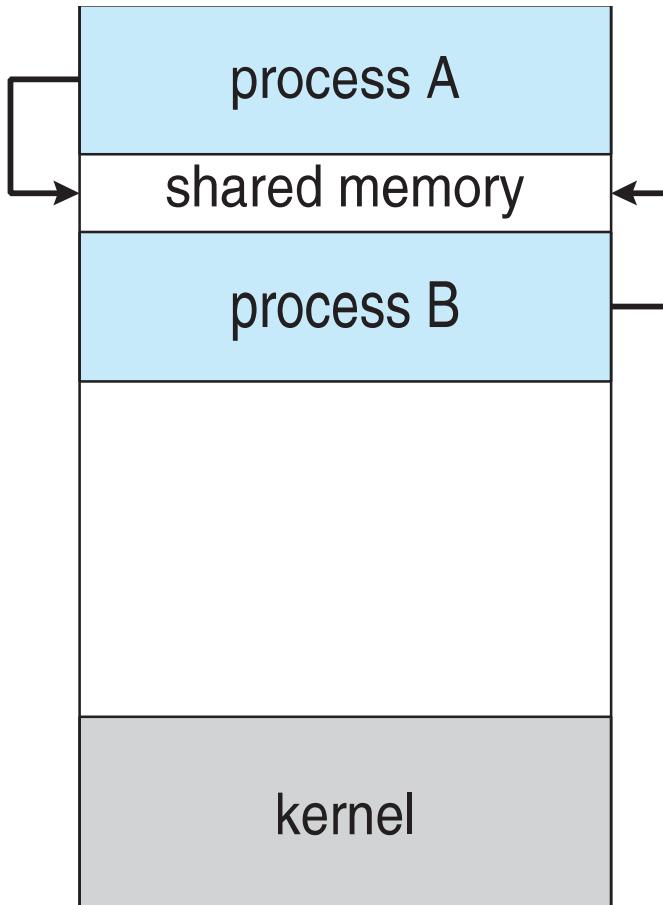
(a) Message passing.



(a)

Scambio di poca informazione,
richieste al Kernel

(b) shared memory.



(b)

Più veloce, più semplice accesso,
problema coordinazione

Memoria Condivisa

- Comunicazione su regione di shared memory
- Tipicamente creata nello spazio di indirizzamento del processo che la crea
- Gli altri processi devono accedere a questo spazio che generalmente è proibito (va rilassata la restrizione)
- I processi leggono e scrivono su questo spazio che è gestito dai processi
 - La coordinazione di lettura e scrittura è sotto la responsabilità dei processi coinvolti, il SO lascia fare

Problema Produttore-Consumatore

- Paradigma tipico per processi cooperanti
 - processo *producer* produce informazione che è consumata dal processo *consumer*
 - Uso di buffer per produrre e consumare
 - Il produttore riempie il buffer
 - Il consumatore svuota il buffer
 - Due tipi di buffer
 - **unbounded-buffer** no limite sulla dimensione del buffer
 - **bounded-buffer** dimensione fissa del buffer

Bounded-Buffer – Shared-Memory

- Un buffer limitato è condiviso tra processi in memoria condivisa

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Buffer circolare con indici `in` (prossima free) e `out` (prima posizione full)
- Soluzione corretta, ma limite su `BUFFER_SIZE-1`
 - Buffer pieno $((in + 1) \% BUFFER_SIZE) == out$
 - Buffer vuoto `in == out`

Bounded-Buffer – Produttore

```
item next_produced;

while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

Bounded Buffer – Consumatore

```
item next_consumed;

while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next consumed */
}
```

Interprocess Communication – Shared Memory

- Area di memoria condivisa tra i processi che desiderano comunicare
 - Comunicazione è sotto il controllo dei processi degli utenti non del SO
 - Occorrono meccanismi che consentano ai processi utente di sincronizzarsi quando accedono alla memoria condivisa
 - La sincronizzazione è discussa in dettaglio in seguito

Interprocess Communication – Message Passing

- Meccanismi che i processi usano per comunicare e per sincronizzarsi
- Messaggi – processi comunicano senza ricorrere a variabili condivise, ma si scambiano messaggi
 - La dimensione del message può essere fissa o variabile
- IPC fornisce solitamente due operazioni:
 - `send(message)`
 - `receive(message)`

Message Passing

- Se i processi P e Q vogliono comunicare devono:
 - Stabilire un canale (communication link) tra loro
 - Scambiare messaggi via send/receive
- Implementazione:
 - Come stabilire il canale?
 - Può il canale coinvolgere più di due processi?
 - Quanti canali di comunicazione tra coppie di processi comunicanti?
 - Capacità di un canale?
 - La dimensione del messaggio scambiato sul canale è fisso o variable?
 - Il canale è unidirezionale o bi-direzionale?

Message Passing

- Implementazione di un canale di comunicazione:
 - Non tanto il canale fisico ...
 - ▶ Memoria condivisa
 - ▶ Bus hardware
 - ▶ Rete
 - ... quanto il canale logico:
 - ▶ Comunicazione diretta o indiretta
 - ▶ Comunicazione sincrona o asincrona
 - ▶ Buffering dei messaggi automatico o esplicito

Comunicazione Diretta

- Processi devono nominarsi esplicitamente (naming):
 - **send** (*P, message*) – invia message al processo P
 - **receive**(*Q, message*) – riceve message dal processo Q
- Proprietà del canale di comunicazione:
 - Il canale è stabilito automaticamente tra coppie di processi comunicanti (che conoscono l'identità reciproca)
 - Il canale è stabilito tra esattamente due processi
 - Per ogni coppia c'è esattamente un canale
 - Il canale può essere unidirezionale o bi-direzionale
- Variante - comunicazione indiretta:
 - **send** (*P, message*) – invia message al processo P
 - **receive**(*id, message*) – riceve message da qualunque processo (*id* ricevuto in ingresso)
- Problema: modularità, modificando l'identità di un processo richiede revisione delle comunicazione

Comunicazione Indiretta

- Messaggi mandati/ricevuti su **mailboxes** (o **porte**):
 - Ogni mailbox ha un unico id
 - Processi possono comunicare solo se condividono una mailbox
- Proprietà del canale:
 - Canale stabilito solo se i processi condividono un mailbox
 - Canale può essere associato a più processi
 - Coppie di processi possono condividere più canali
 - Canali sia unidirezionali che bi-direzionali

Comunicazione Indiretta

- Operazioni

- Creare una nuova mailbox (porta)
- Messaggi send e receive via la mailbox
- distruggere una mailbox

- Primitive:

`send(A, message)` – invia *message* alla mailbox A

`receive(A, message)` – ricevi *message* dalla mailbox A

- Proprietà del canale:

- Canale stabilito solo se i processi condividono un mailbox
- Canale può essere associato a più processi
- Coppie di processi possono condividere più canali
- Canali sia unidirezionali che bi-direzionali

Comunicazione Indiretta

- Condivisione del mailbox
 - Si supponga che P_1 , P_2 , e P_3 condividono la mailbox A
 - Se P_1 , invia M ad A e P_2 e P_3 ricevono da A
 - Chi prende il messaggio M?
- Soluzione dipendono dal metodo scelto:
 - Permettere una connessione tra solo due processi alla volta
 - Permettere ad un processo alla volta di eseguire *receive()*
 - Lasciare al sistema decidere il ricevente (al mittente verrà notificata la decisione)

Sincronizzazione

- La comunicazione tra processi avviene con chiamate *send-receive*
- Il message passing può essere *bloccante* o *non bloccante*
- **Bloccante** considerato **sincrono**
 - **Blocking send** -- mittente è bloccato finché il messaggio non viene ricevuto
 - **Blocking receive** -- destinatario bloccato finché il messaggio non è disponibile
- **Non bloccante** considerato **asincrono**
 - **Non-blocking send** -- mittente invia il messaggio e continua ad eseguire
 - **Non-blocking receive** -- destinatario riceve:
 - messaggio valido, oppure
 - messaggio nullo

Diverse combinazioni di politiche

- Se bloccati sia il mittente che il destinario abbiamo un **rendezvous**

Sincronizzazione

- Con sincronizzazione bloccante il problema del produttore consumatore diventa triviale:
 - Il produttore invia e attende il consumo
 - Il consumatore attende e consuma

```
message next_produced;  
while (true) {  
    /* produce an item in next produced */  
    send(next_produced);  
}  
  
message next_consumed;  
while (true) {  
    receive(next_consumed);  
  
    /* consume the item in next consumed */  
}
```

Buffering

- Le code di messaggi richiedono un canale che accoda i messaggi
- Canale implementato in uno dei tre modi:
 - **Capacità zero** – zero messaggi in coda, il mittente aspetta che il destinatario riceva (rendezvous)
 - **Capacità limitata** – capacità di n messaggi in coda, il mittente aspetta se il canale è pieno
 - **Capacità illimitata** – la coda non ha lunghezza predefinita, il mittente non aspetta mai
- Capacità zero è detto senza buffering
- Capacità non zero è detto con buffering automatico

Esempio di IPC Systems - POSIX

□ POSIX Shared Memory

- Un processo crea un segmento di shared memory con la call
`shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);`

- nome, modalità di apertura/creazione, permessi
 - restituisce un intero (descrittore di file)
 - anche usata per aprire un segmento esistente per condividerlo

- Creato l'oggetto si setta la dimensione

```
ftruncate(shm_fd, 4096);
```

- setta 4096 byte

- Memory-mapped file

```
mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);
```

- `MAP_SHARED` specifica che i cambiamenti sono condivisi

- È quindi possibile scrivere su shared memory

```
sprintf(shared_memory, "Writing to shared memory");
```

IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* strings written to shared memory */
    const char *message_0 = "Hello";
    const char *message_1 = "World!";

    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory object */
    void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

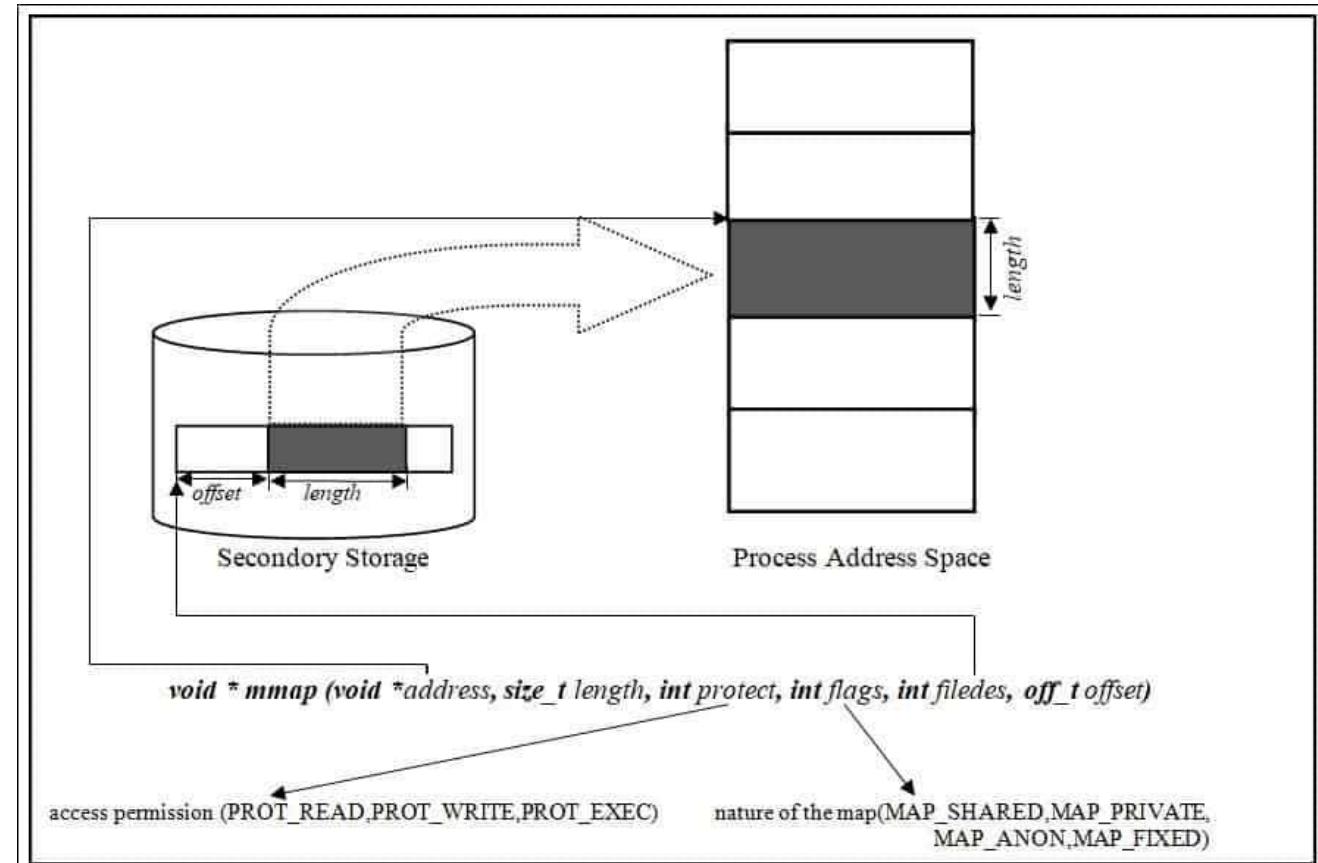
    return 0;
}
```

POSIX mmap

- La funzione **mmap**: mappa in process address space un file o un device
 - Una volta mappato accessibile in modo diretto

```
#include <sys/mman.h>
void * mmap (void *address, size_t length, int protect, int flags, int filedes, off_t offset)
```

- **address**: indirizzo di partenza richiesto
- **size_t**: bytes richiesti
- **protect**:
PROT_READ | PROT_WRITE |
PROT_EXEC | PROT_NONE
- **flags**:
MAP_SHARED, MAP_PRIVATE
MAP_ANON, MAP_FIXED
- **fd**: descrittore di file
- **offset**: offset del file



IPC POSIX Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory obect */
    void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s", (char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

Esempi di Sistemi IPC - Mach

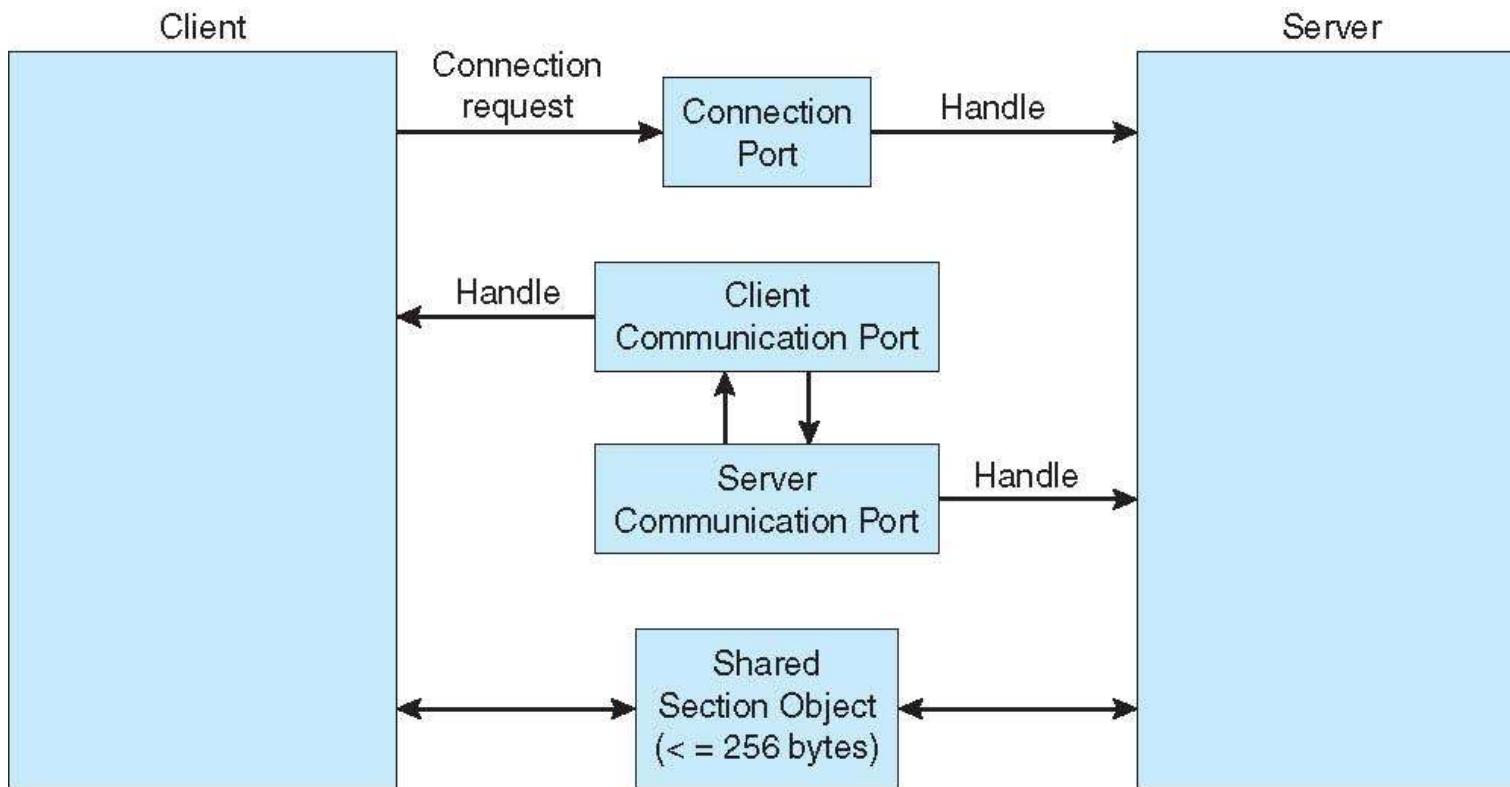
- Mach communication is message based
 - Even system calls are messages
 - Each task gets two mailboxes at creation- Kernel and Notify
 - Only three system calls needed for message transfer
`msg_send()`, `msg_receive()`, `msg_rpc()`
 - Mailboxes needed for communication, created via
`port_allocate()`
 - Send and receive are flexible, for example four options if mailbox full:
 - ▶ Wait indefinitely
 - ▶ Wait at most n milliseconds
 - ▶ Return immediately
 - ▶ Temporarily cache a message

Esempi di Sistemi IPC – Windows

- Windows SO è progettato in modo modulare
 - Supporto per ambienti operativi multipli (subsystems)
 - Programmi comunicano con sottosistemi mediante message-passing
 - Applicazioni Client e sottosistemi sono i Server
- Message-passing con **advanced local procedure call (ALPC)**
 - Solo tra processi sullo stesso sistema
 - Porte (come mailboxes) per stabilire e mantenere canali di comunicazione
 - Due tipi di porte: **connection port** e **communication port**
 - La comunicazione funziona come segue:
 - I processi server pubblicano **connection port** visibili per tutti i processi
 - Il client apre un handle per una **connection port** e manda una richiesta di connessione
 - Il server crea un canale e restituisce un handle al client
 - Il canale fornisce una coppia di **communication port** private nelle due direzioni
 - Client e Server usano gli handle delle porte per mandare messaggi
 - Meccanismi di callback per accettare richieste mentre aspettano replicate

Local Procedure Calls in Windows

- La ALPC non è parte delle Windows API e non è visibile dai processi applicativi, che invece invocano remote procedure calls (RPC)
- Molti servizi del Kernel usano ALPC per comunicare con processi client

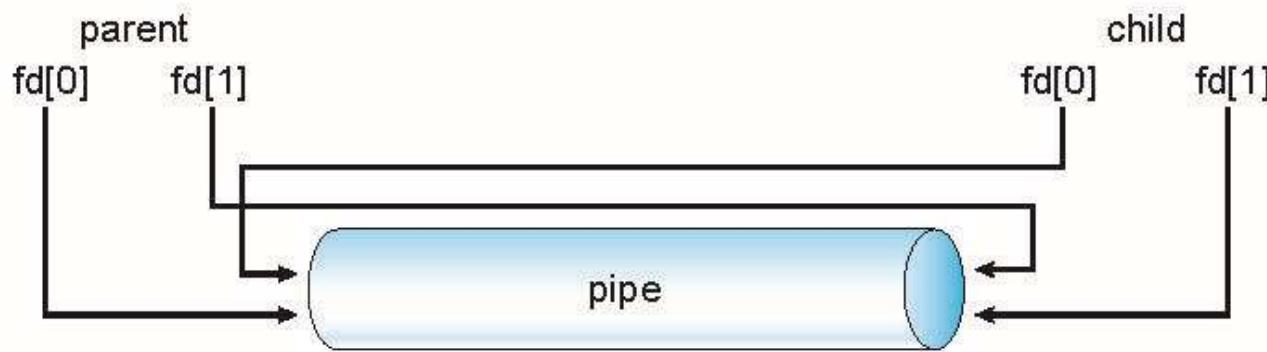


Pipe

- Flusso di dati che permette a due processi di comunicare
- Uno dei primi meccanismi di comunicazione in UNIX
- Problematiche:
 - Comunicazione unidirezionale o bidirezionale?
 - Se two-way communication, è half o full-duplex?
 - Ci deve essere una relazione (i.e., **parent-child**) tra i processi comunicanti?
 - Si possono usare le pipes in una rete?
- Pipe ordinarie
 - Non può essere usata da processi che non hanno creato
 - Processo padre crea la pipe e la usa per comunicare con un processo figlio
- Named pipes
 - Possono essere usate senza una relazione parentale tra processi

Pipe Ordinarie

- Pipe ordinarie comunicano in modalità produttore-consumatore
- Produttore scrive su un'estremità (**write-end** della pipe)
- Consumatore legge dall'altra estremità (**read-end** della pipe)
- Le pipe ordinarie sono unidirezionali
- Richiedono una relazione parentale tra i processi che comunicano



- Windows le chiama **anonymous pipes**

Pipe Ordinarie

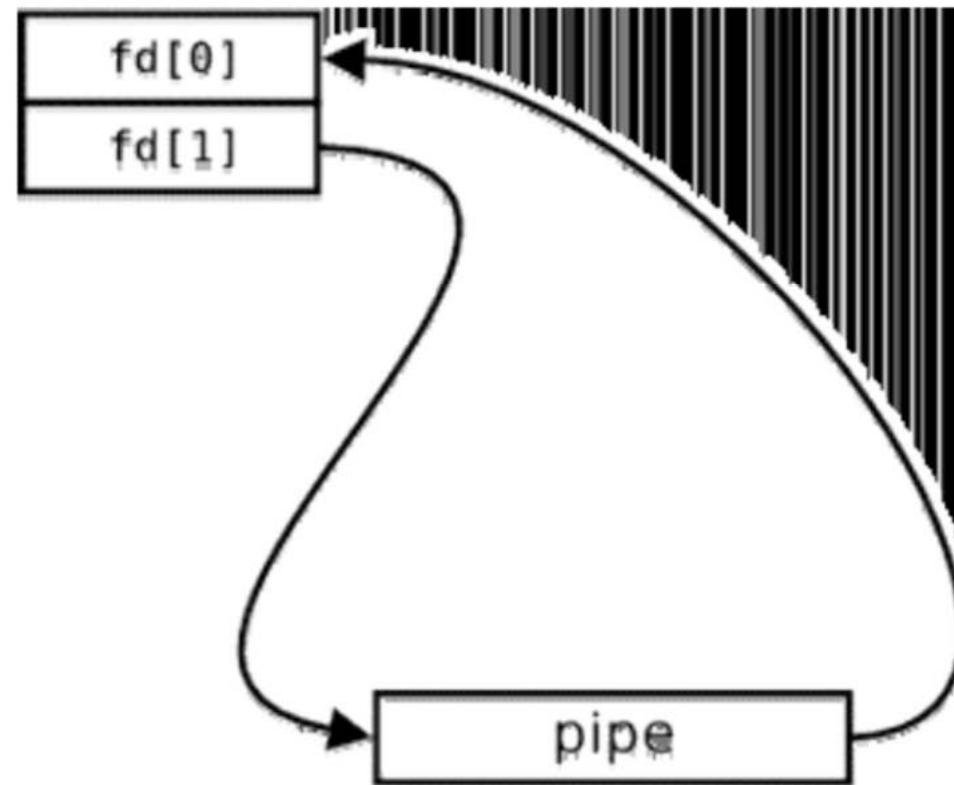
■ La funzione pipe in Unix

```
#include <unistd.h>  
  
int pipe ( int filedes[2] );
```

- l'argomento *filedes* è costituito da due descrittori di file:
 - *filedes[0]* è aperto in lettura e rappresenta il lato in lettura della pipe ;
 - *filedes[1]* è aperto in scrittura e rappresenta il lato in scrittura della pipe;
 - l'output di *filedes[1]* è l'input per *filedes[0]*;
- restituisce 0 in caso di successo, -1 altrimenti.

Pipe Ordinarie

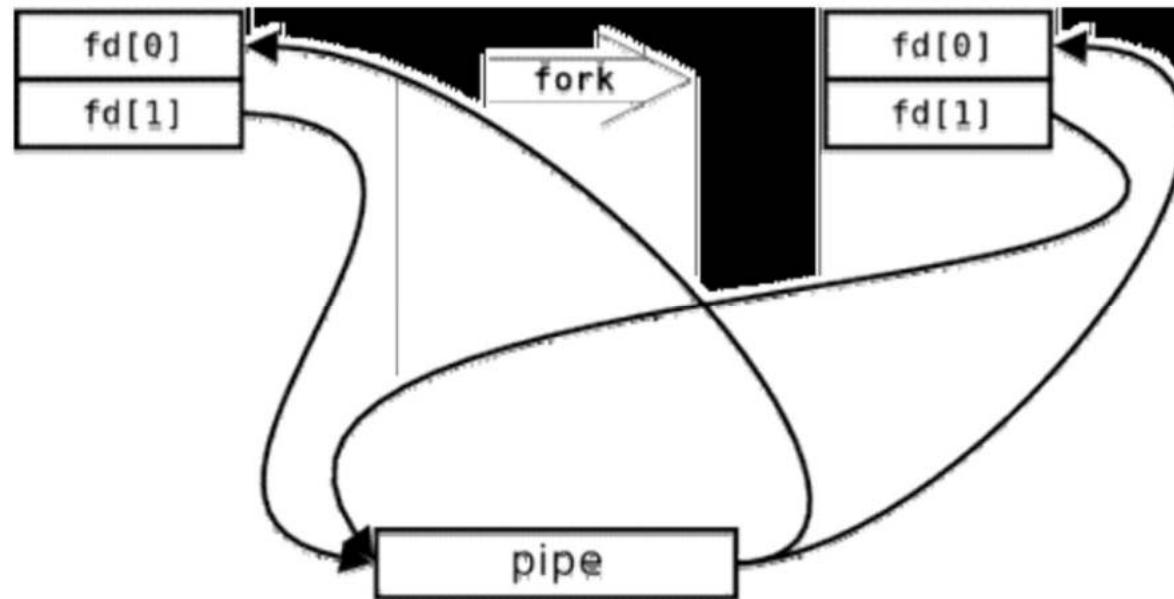
- La funzione pipe in Unix



Generata da un singolo processo ha poca utilità ...

Pipe Ordinarie

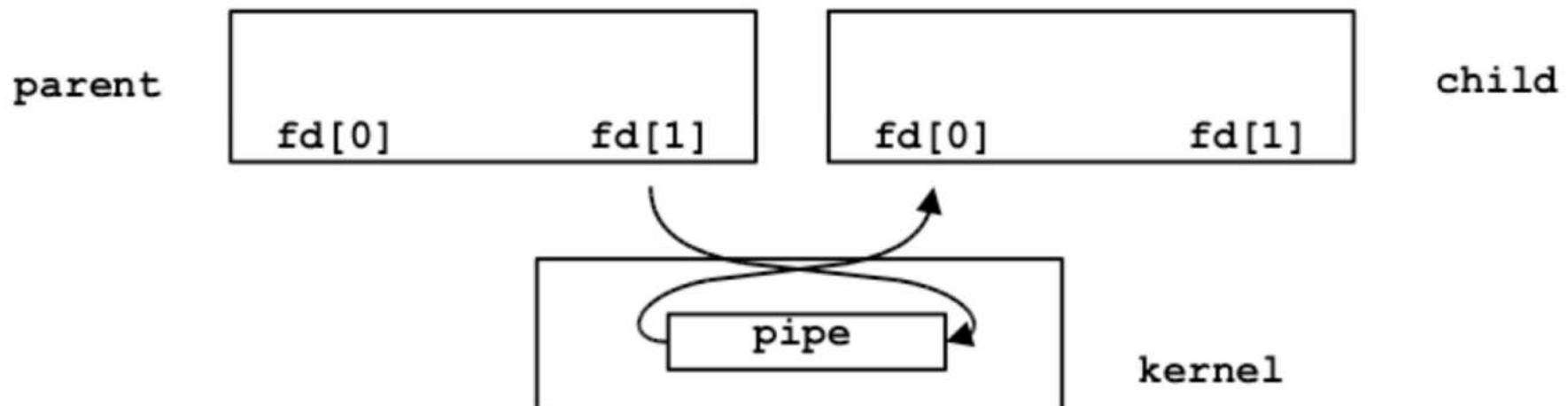
- La funzione pipe in Unix
 - Tipicamente, un processo crea una pipe e poi chiama fork



Pipe Ordinarie

■ La funzione pipe in Unix

- **Come utilizzare i pipe?**
 - Cosa succede dopo la `fork` dipende dalla direzione dei dati
 - I canali non utilizzati vanno chiusi
- **Esempio: parent → child**
 - Il parent chiude l'estremo di read (`close(fd[0])`)
 - Il child chiude l'estremo di write (`close(fd[1])`)



Pipe Ordinarie

■ La funzione pipe in Unix

```
int fd[2];

if (pipe(fd) < 0)
    perror("pipe"), exit(1);
if ( (pid=fork()) < 0 )
    perror("fork"), exit(1);
else if (pid>0) { // padre
    close(fd[0]);
    write(fd[1], "ciao!", 5);
} else {           // figlio
    close(fd[1]);
    n = read(fd[0], buf, sizeof(buf));
    write(STDOUT_FILENO, buf, n);
}
```

Pipe Ordinarie

■ La funzione pipe in Unix

```
/* pipel: invio di dati da un genitore ad un figlio */

#include <stdio.h>
#include <unistd.h>
#define MAXLINE 64

int main(void)
{
    int n, fd[2];
    pid_t pid;
    char line[MAXLINE];

    if (pipe(fd) < 0) perror("pipe"), exit(1);

    if ( (pid = fork()) < 0) perror("fork"), exit(1);

    else if (pid > 0) { /* genitore */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    }
    else { /* figlio */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

Pipe Ordinarie

■ La funzione pipe in Unix

- All'inizio una pipe è vuota
- write aggiunge dati alla pipe
- read legge e *rimuove* dati dalla pipe
 - non si possono leggere piu' volte gli stessi dati da una pipe
 - non si puo' chiamare lseek su una pipe
 - i dati si ottengono in ordine First In First Out
- una pipe con una estremità chiusa si dice rotta (broken)

Pipe Ordinarie

- La funzione pipe in Unix
 - Scrivere: write aggiunge i suoi dati alla pipe
 - se la pipe e' rotta, viene generato il segnale SIGPIPE e write restituisce un errore
 - Leggere: read(fd[0], buf, 100)
 - meno di 100 bytes nella pipe: read legge l'intero contenuto della pipe
 - piu' di 100 bytes nella pipe: read legge i primi 100 bytes
 - pipe vuota: read si blocca in attesa di dati
 - pipe vuota e rotta: read restituisce 0

Pipe Ordinarie

■ La funzione pipe in Windows (pipe anonime)

■ Come in Unix, processi imparentati

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

#define BUFFER_SIZE 25

int main(VOID)
{
    HANDLE ReadHandle, WriteHandle;
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    char message[BUFFER_SIZE] = "Greetings";
    DWORD written;

    /* set up security attributes allowing pipes to be inherited */
    SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES),NULL,TRUE}; Per far ereditare gli handle
    /* allocate memory */
    ZeroMemory(&pi, sizeof(pi));

    /* create the pipe */
    if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) {
        fprintf(stderr, "Create Pipe Failed");
        return 1;
    }
```

Pipe Ordinarie

② La funzione pipe in Windows (pipe anonime)

③ Come in Unix, processi imparentati

```
/* establish the START_INFO structure for the child process */
GetStartupInfo(&si);
si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

/* redirect standard input to the read end of the pipe */
si.hStdInput = ReadHandle;
si.dwFlags = STARTF_USESTDHANDLES;                                Ridirezione dello standard input per
                                                                    leggere da pipe

/* don't allow the child to inherit the write end of pipe */
SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);

/* create the child process */
CreateProcess(NULL, "child.exe", NULL, NULL,
    TRUE, /* inherit handles */                                         Flag a TRUE per far ereditare gli handle
    0, NULL, NULL, &si, &pi);

/* close the unused end of the pipe */
CloseHandle(ReadHandle);

/* the parent writes to the pipe */
if (!WriteFile(WriteHandle, message,BUFFER_SIZE,&written,NULL))
    fprintf(stderr, "Error writing to pipe.");

/* close the write end of the pipe */
CloseHandle(WriteHandle);

/* wait for the child to exit */
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
return 0;
```

Pipe Ordinarie

- La funzione pipe in Windows (pipe anonime)
 - Come in Unix, processi imparentati

```
#include <stdio.h>
#include <windows.h>

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE Readhandle;
CHAR buffer[BUFFER_SIZE];
DWORD read;

/* get the read handle of the pipe */
ReadHandle = GetStdHandle(STD_INPUT_HANDLE);

/* the child reads from the pipe */
if (ReadFile(ReadHandle, buffer, BUFFER_SIZE, &read, NULL))
    printf("child read %s",buffer);
else
    fprintf(stderr, "Error reading from pipe");

return 0;
}
```

Pipe con Nome

- Named Pipes sono più potenti delle pipe ordinarie
- La comunicazione è bidirezionale
- Non è necessaria una relazione parentale tra processi comunicanti
- Molti processi possono usare le named pipe per comunicare
- Sia su UNIX che su Windows

Pipe con Nome in Unix

I file speciali FIFO (pipe con nome) consentono di superare alcune delle limitazioni delle pipe. Essi infatti, rispetto a queste ultime, offrono i seguenti vantaggi:

- una volta creati, esistono nel file system fintanto che non vengono esplicitamente cancellati;
- possono essere usati da processi che non hanno un comune antenato.

I file FIFO possono essere creati in due modi:

- attraverso la shell, con il comando mkfifo;
- all'interno di un programma, con la chiamata alla funzione mkfifo.

Una volta creato un file FIFO, su di esso si possono effettuare le operazioni usuali di IO su file (open, read, write, close, ...)

Pipe con Nome in Unix

```
#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define MAX_BUF_SIZE 1000

int main(int argc, char *argv[]){
    int fd, ret_val, count, numread;
    char buf[MAX_BUF_SIZE];

    /* Create the named - pipe */
    ret_val = mkfifo("miafifo", 0666);
    if ((ret_val == -1) && (errno != EEXIST)) {
        perror("Error creating the named pipe");
        exit (1);
    }

    /* Open the pipe for reading */
    fd = open("miafifo", O_RDONLY);

    /* Read from the pipe */
    numread = read(fd, buf, MAX_BUF_SIZE);
    buf[numread] = '\0';
    printf("Server : Read From the pipe : %s\n", buf);

    /* Convert the string to upper case */
    count = 0;
    while (count < numread) {
        buf[count] = toupper(buf[count]);
        count++;
    }
    printf("Server : Converted String : %s\n", buf);
}
```

Codice Server

Pipe con Nome in Unix

```
#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
    int fd;

    /* Check if an argument was specified. */

    if (argc != 2) {
        printf("Usage : %s <string to be sent to the server>\n", argv[0]);
        exit (1);
    }

    /* Open the pipe for writing */
    fd = open("miafifo", O_WRONLY);

    /* Write to the pipe */
    write(fd, argv[1], strlen(argv[1]));
}
```

Codice Client

Esempio esecuzione:

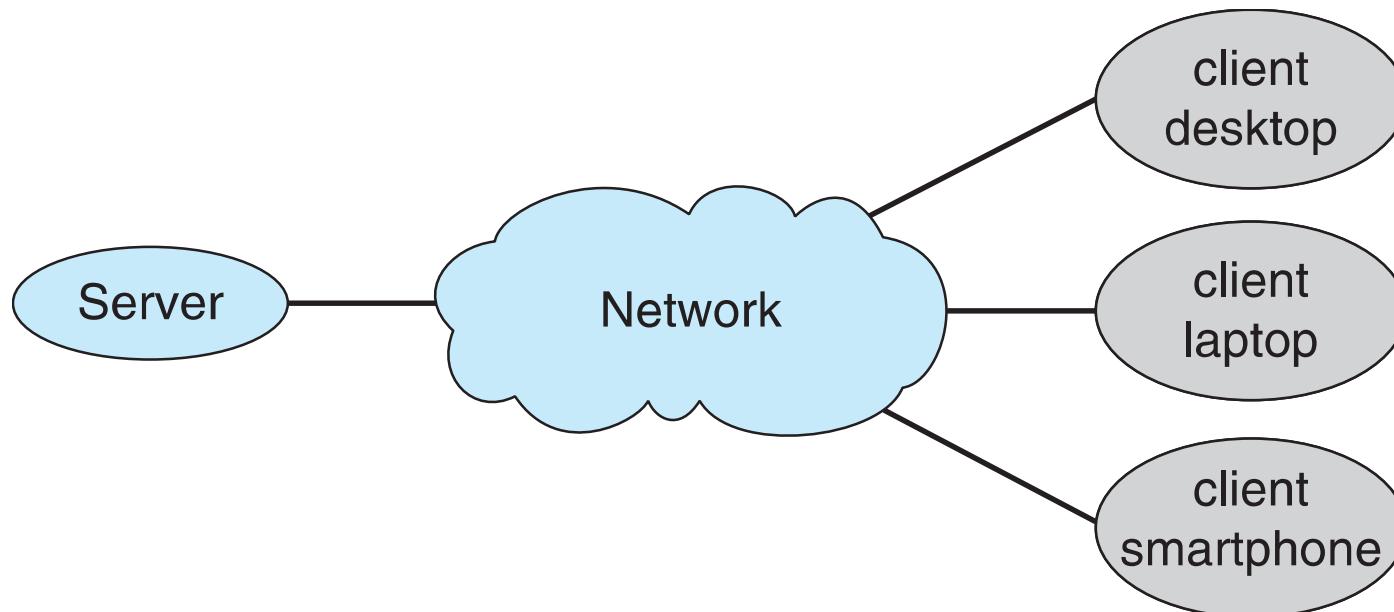
- \$./servFifo &
- \$./clientFifo prova
- Server : Read From the pipe : prova
- Server : Converted String : PROVA

Comunicazione in Sistemi Client-Server

- Per la comunicazione client-server vediamo due metodi
 - Sockets
 - Remote Procedure Call

Sistemi Client-Server

- Sistemi Client-Server
 - Tipica architettura di rete dove Sistemi **Server** rispondono alle richieste dei Sistemi **client**



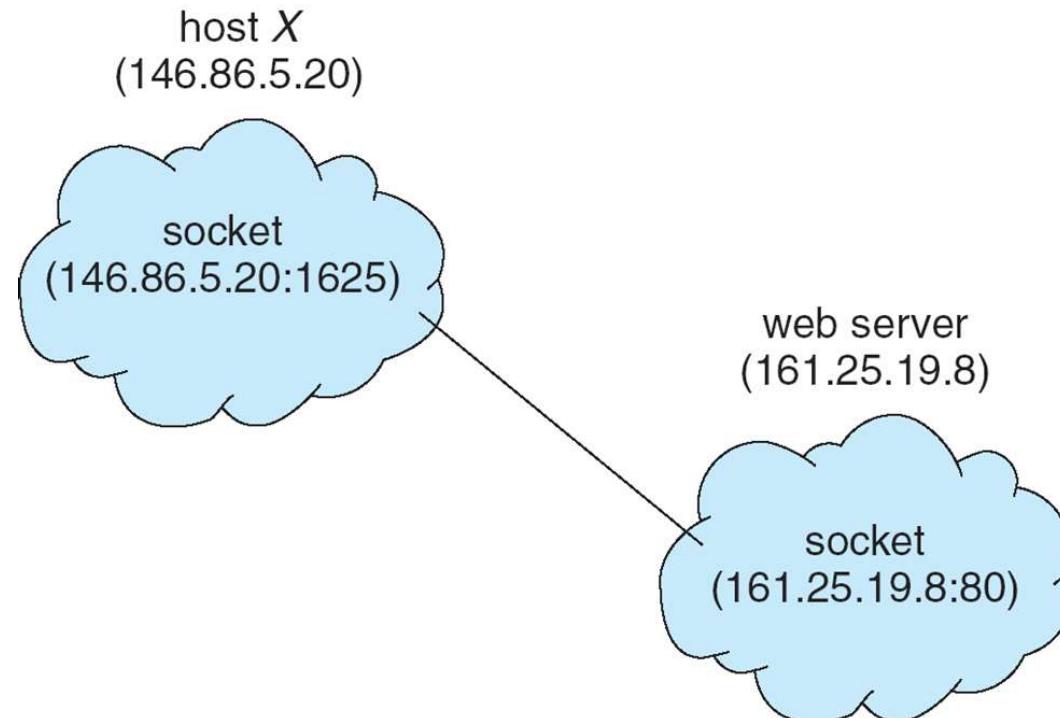
Socket

- Una **socket** è un punto finale (endpoint) di comunicazione
- Introdotti nel 1983 in Berkeley Software Distribution (BSD)
 - Berkeley socket API
- La comunicazione avviene tra coppie di socket
 - Su stessa macchina (IPC locali) o diverse (IPC di rete)
- Una socket internet è identificata da un indirizzo e una porta:
161.25.19.8:1625 indica la porta **1625** sull'host **161.25.19.8**
- La **porta** è un numero che differenzia diversi servizi su un host
- Tutte le porte sotto 1024 sono usate per servizi standard
 - Es. FTP 21, SSH 22, 25 smtp, 80 HTTP, etc. (vedi www.iana.org)
- Un IP address speciale 127.0.0.1 (**loopback**) si riferisce al sistema in cui il processo gira

Comunicazione Socket

□ Comunicazione Client-Server

- Server in ascolto su una porta e Client richiede servizio
- Processo Client stabilisce una comunicazione tramite una porta
 - ▶ Connessione unica per processo (altro Client altra porta)
- Si possono definire diversi tipi di comunicazione
 - ▶ Connection oriented (TCP) o connectionless (UDP)



Comunicazione Socket Unix

- Comunicazione Client-Server Unix

- Server in ascolto

```
int fd1, fd2;
struct sockaddr_in mio_indirizzo;

mio_indirizzo.sin_family      = AF_INET;
mio_indirizzo.sin_port        = htons(5200);
mio_indirizzo.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
fd1 = socket(PF_INET, SOCK_STREAM, 0);
bind(fd1, (struct sockaddr *)&mio_indirizzo, sizeof(mio_indirizzo));

listen(fd1, 5);
fd2 = accept(fd1, NULL, NULL);
...
close(fd2);
close(fd1);
```

Comunicazione Socket Unix

- Comunicazione Client-Server Unix
 - Client connesso

```
int fd;
struct sockaddr_in mio_indirizzo;

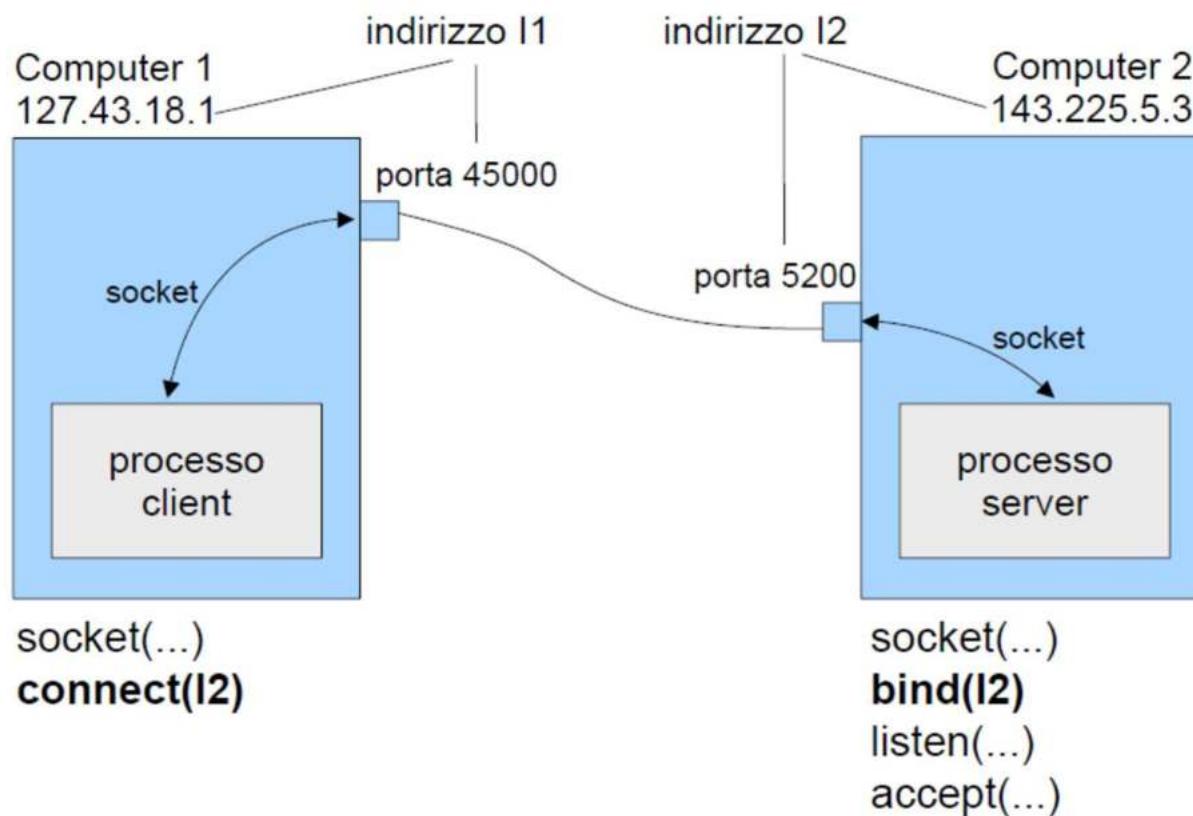
mio_indirizzo.sin_family      = AF_INET;
mio_indirizzo.sin_port        = htons(5200);
inet_aton("143.225.5.3", &mio_indirizzo.sin_addr);

fd = socket(PF_INET, SOCK_STREAM, 0);
connect(fd, (struct sockaddr *) &mio_indirizzo,
sizeof(mio_indirizzo));

...
close(fd);
```

Comunicazione Socket Unix

- Comunicazione Client-Server Unix
 - Schema di connessione



Comunicazione Socket Unix

- Comunicazione Client-Server Unix
 - Schema di un Server concorrente

```
socket(...);
bind(...);
listen(...);

while (1) {
    fd2 = accept(fd1, (struct sockaddr *)NULL, NULL);

    if ( (pid = fork()) < 0 ) {
        perror("fork");
        exit(1);
    } else if (pid == 0) {      // processo figlio
        close(fd1); // al figlio non serve fd1
        // gestisce la connessione usando fd2
        ...
        exit(0); // poi il figlio termina
    }
    // il processo padre chiude fd2 e ripete il ciclo
    close(fd2);
}
```

Socket in Java

- Esempio in Java
- “Date” server comunica la data ai client
- Ascolta sulla porta 6013
- Attende su accept()
- Crea socket per servire un client
- PrintWriter permette di scrivere al client
- Scrive la data
- Chiude la comunicazione con il client
- Si rimette in ascolto

```
import java.net.*;
import java.io.*;

public class DateServer
{
    public static void main(String[] args) {
        try {
            ServerSocket sock = new ServerSocket(6013);

            /* now listen for connections */
            while (true) {
                Socket client = sock.accept();

                PrintWriter pout = new
                    PrintWriter(client.getOutputStream(), true);

                /* write the Date to the socket */
                pout.println(new java.util.Date().toString());

                /* close the socket and resume */
                /* listening for connections */
                client.close();
            }
        } catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

Socket in Java

- Esempio in Java
- “Date” Client chiede la data al Server
- Chiede la connessione al Server
- Stabilita la connessione stabilisce uno I/O stream sulla Socket
- Legge la data
- La scrive in standard output
- Chiude la comunicazione con il Server

```
import java.net.*;
import java.io.*;

public class DateClient
{
    public static void main(String[] args) {
        try {
            /* make connection to server socket */
            Socket sock = new Socket("127.0.0.1", 6013);

            InputStream in = sock.getInputStream();
            BufferedReader bin = new
                BufferedReader(new InputStreamReader(in));

            /* read the date from the socket */
            String line;
            while ( (line = bin.readLine()) != null)
                System.out.println(line);

            /* close the socket connection*/
            sock.close();
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

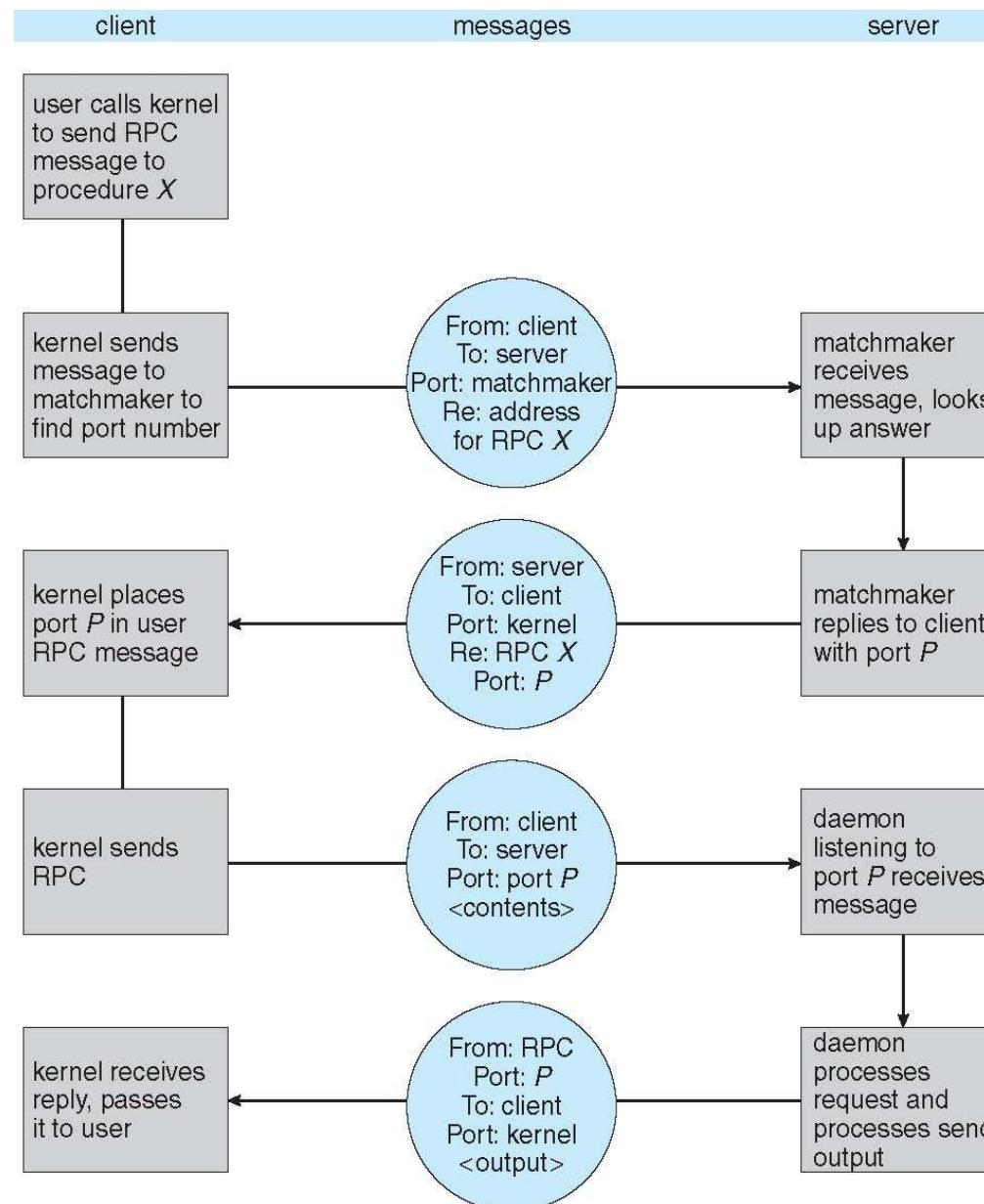
Remote Procedure Call

- Le socket permettono comunicazione di “basso livello”
 - Flusso di dati non strutturati
- Le Remote procedure call (RPC) sono astrazioni di chiamata di procedure tra processi su sistemi in rete
 - Usano porte per differenziare i servizi
- **Stub** – proxy client-side per le effettive procedure sul server
- Lo stub client-side trova il server e prepara i parameteri (**marshalling**)
- Lo stub server-side riceve il messaggio, spacchetta i parameteri e esegue le procedure sul server
- Su Windows lo stub code compila dalla specifica scritta in **Microsoft Interface Definition Language (MIDL)**

Remote Procedure Call

- Rappresentazione dati gestite con il formato **External Data Representation (XDL)** per considerare le diverse architetture
 - **Big-endian e little-endian**
- Remote communication ha più scenari di fallimento rispetto alle locali
 - Messaggi possono essere mandati esattamente una volta invece che al massimo una volta
- SO tipicamente usano meccanismi rendezvous (o **matchmaker**) per connettere client e server
 - Il client manda un messaggio al demone del matchmaker per richiedere la port per la comunicazione

Esecuzione di RPC



Lezione 7: Scheduling CPU

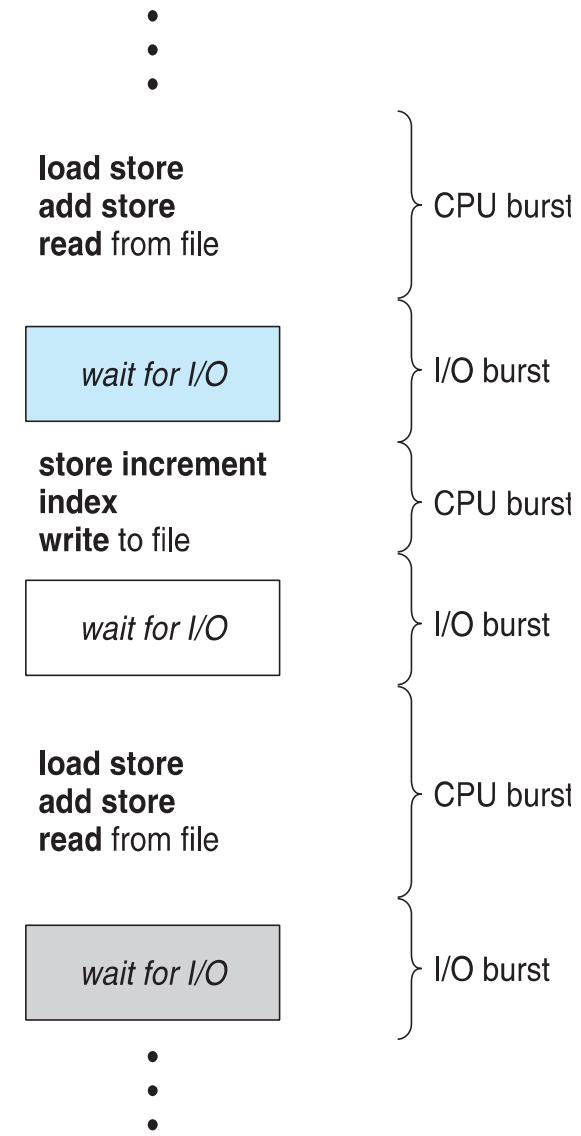


Obiettivi

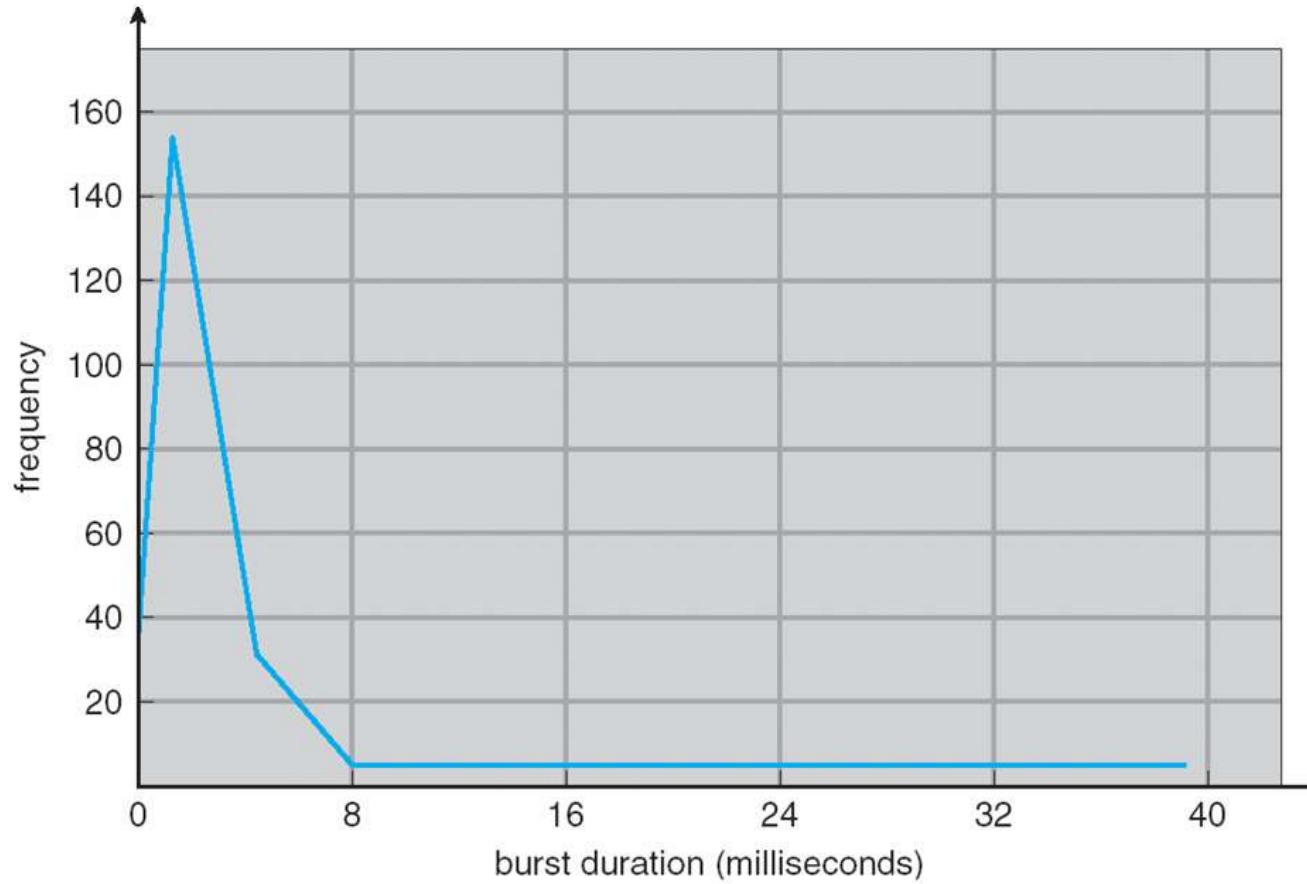
- Introdurre il concetto dello scheduling CPU
- Descrivere algoritmi di scheduling di CPU
- Discutere criteri di valutazione degli algoritmi di scheduling
- Esaminare algoritmi di scheduling algorithms di vari SO

Concetti di Base

- Massimo utilizzo di CPU con multiprogramming
- Cicli CPU–I/O Burst – L'esecuzione dei processi consiste di **cicli** di esecuzione CPU e attese I/O
- **CPU burst** seguiti da **I/O burst**
- Distribuzione di CPU burst è una problematica importante



Istogramma dei tempi di CPU-burst

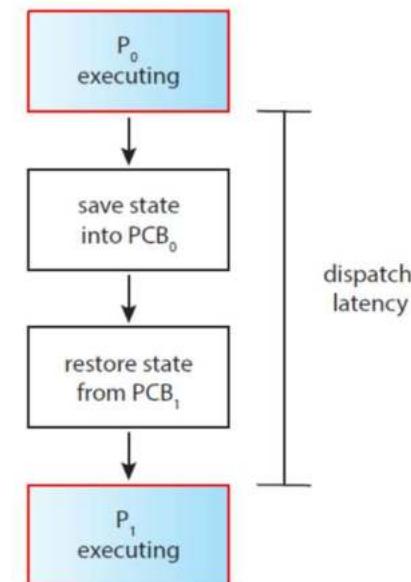


CPU Scheduler

- **Short-term scheduler** seleziona tra processi nella coda ready e alloca la CPU a uno di loro
 - Le code possono essere ordinate in modi diversi
- Le decisioni di CPU scheduling occorrono quando i processi:
 1. Passano dallo stato running a waiting
 2. Passano dallo stato running a ready
 3. Passano da waiting a ready
 4. Terminano
- Scheduling con 1 e 4 non prevede prelazione (**nonpreemptive**)
- Ogni altro scheduling prevede prelazione (**preemptive**)
 - Problema accesso a dati condivisi
 - Problema prelazione in modalità kernel
 - Problema interruzioni durante attività curciali del SO

Dispatcher

- Dispatcher dà il controllo della CPU ai processi selezionati dallo short-term scheduler e richiede:
 - switching del contesto
 - switching allo user mode
 - salto alla corretta locazione del programma utente per ripartire con quel programma
- **Latenza di dispatch** – tempo necessario al dispatcher per fermare un processo e mandare un altro in running
 - Più breve possibile



Criteri di Scheduling

- **Utilizzo CPU** – percentuale di CPU utilizzata; occorre mantenere la CPU occupata.
- **Throughput** – numero di processi che completano l'esecuzione per unità di tempo
- **Turnaround time** – tempo di completamento di processo
 - tempo totale per eseguire un processo
 - accesso memoria + coda ready + CPU + I/O
- **Waiting time** – tempo di attesa per un processo nella coda ready
- **Response time** – tempo che intercorre tra una richiesta e la prima risposta
 - Importante per sistemi interattivi, quando l'elaborazione continua dopo un output
 - Alternativa al tempo di turnaround che prevede l'esecuzione completa

Scheduling: Criteri di Ottimizzazione

- Massimo utilizzo CPU
 - Massimo throughput
 - Minimo tempo di turnaround
 - Minimo tempo di waiting
 - Minimo tempo di risposta
-
- In molti sistemi (desktop, laptop) è più importante minimizzare la varianza dei tempi di risposta che il tempo medio

First- Come, First-Served (FCFS) Scheduling

| <u>Process</u> | <u>Burst Time</u> |
|----------------|-------------------|
| P_1 | 24 |
| P_2 | 3 |
| P_3 | 3 |

- Assumendo che i processi nell'ordine: P_1, P_2, P_3
Il Gantt Chart dello schedule è:



- Tempi di attesa per $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Media del tempo di attesa: $(0 + 24 + 27)/3 = 17$

FCFS Scheduling

Assumendo i processi nell'ordine:

$$P_2, P_3, P_1$$

- Il Gantt diventa:



- Tempi di attesa $P_1 = 6; P_2 = 0, P_3 = 3$
- Tempo di attesa medio: $(6 + 0 + 3)/3 = 3$
- Molto meglio di prima
- **Effetto Convoy** - processi brevi dietro i processi lunghi
 - Considera un processo CPU-bound e tanti I/O-bound
- Sbrigare i processi brevi per migliorare i tempi di risposta

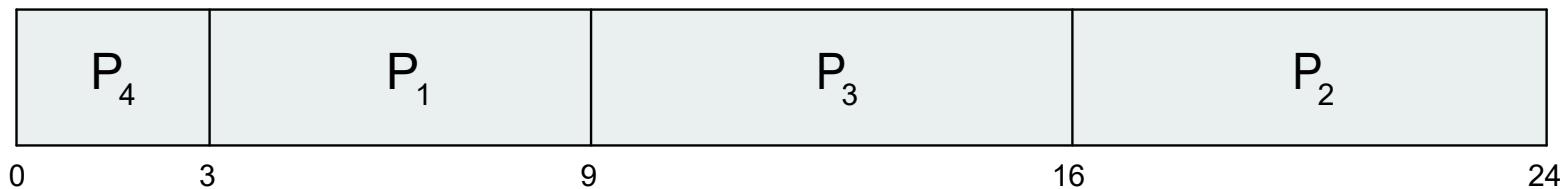
Shortest-Job-First (SJF) Scheduling

- Associa ad ogni processo la lunghezza del prossimo CPU burst
 - Lunghezza usata per schedulare il processo con brust più breve
- SJF è ottimale – minima attesa media per un insieme di processi
 - Difficile conoscere la lunghezza della prossima richiesta di CPU
 - Soluzione: uso di stime basate sul comportamento passato ed esecuzione del processo con il minor tempo di esecuzione stimato
- Algoritmo particolarmente indicato per l'esecuzione batch dei job per i quali i tempi di esecuzione sono conosciuti a priori
- Lo scheduler dovrebbe utilizzare questo algoritmo quando nella coda di input risiedono job di uguale importanza

Esempio di SJF

| <u>Process</u> | <u>Burst Time</u> |
|----------------|-------------------|
| P_1 | 6 |
| P_2 | 8 |
| P_3 | 7 |
| P_4 | 3 |

- SJF scheduling chart

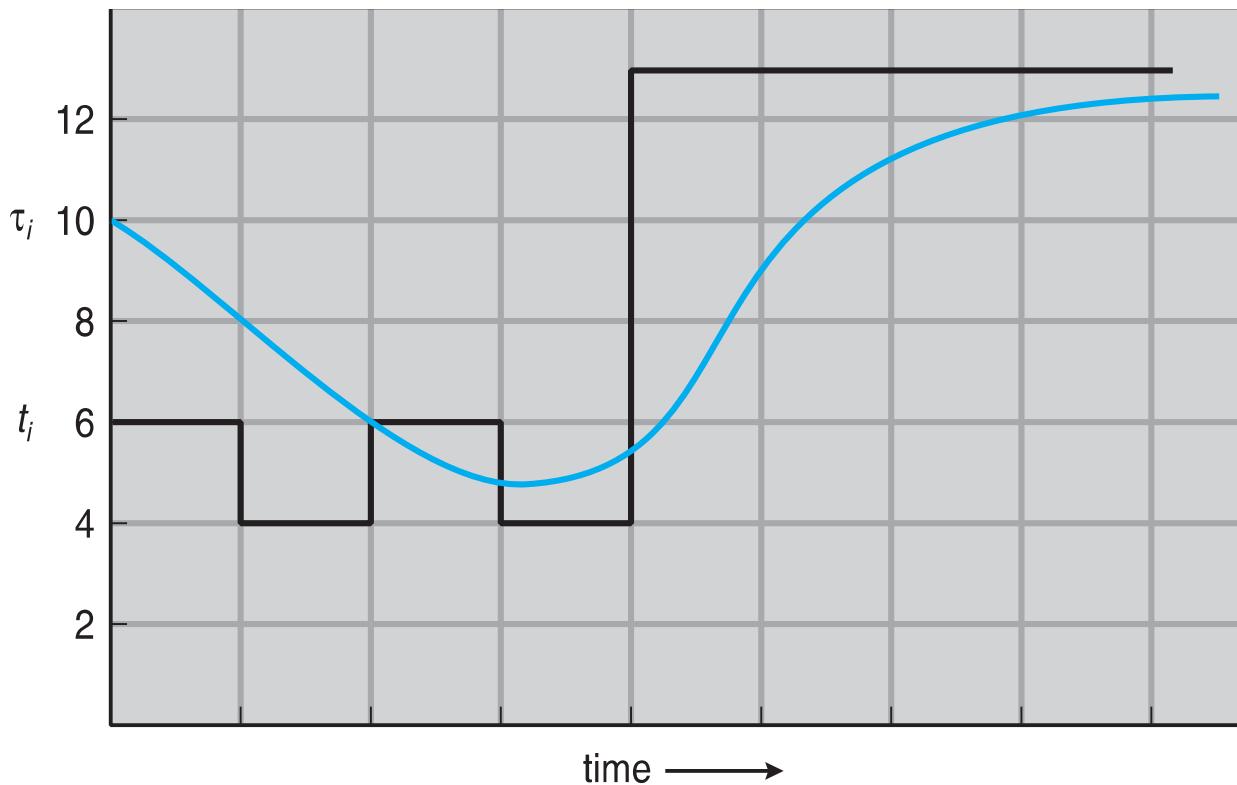


- Tempo di attesa medio = $(3 + 16 + 9 + 0) / 4 = 7$

Lunghezza del prossimo CPU Burst

- Si può solo stimare – simile alla precedente
 - Prendi il processo con il CPU burst stimato come il più breve
- Utilizzare la lunghezza del CPU burst precedente, usando la media esponenziale delle lunghezze precedenti
 1. t_n = actual length of n^{th} CPU burst
 2. τ_{n+1} = predicted value for the next CPU burst
 3. $\alpha, 0 \leq \alpha \leq 1$
 4. Define: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$.
- Solitamente α settato a $\frac{1}{2}$
- Versione preemptive chiamata **shortest-remaining-time-first**

Lunghezza del prossimo CPU Burst



CPU burst (t_i)

6

4

6

4

13

13

13

...

"guess" (τ_i)

10

8

6

6

5

9

11

12

...

Esempi di Exponential Averaging

- $\alpha = 0$

- $\tau_{n+1} = \tau_n$

- Storia recente non conta

- $\alpha = 1$

- $\tau_{n+1} = \alpha t_n$

- Conta solo l'ultimo CPU burst

- Espandendo la formula si ottiene:

$$\begin{aligned}\tau_{n+1} = & \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \dots \\ & +(1 - \alpha)^j \alpha t_{n-j} + \dots \\ & +(1 - \alpha)^{n+1} \tau_0\end{aligned}$$

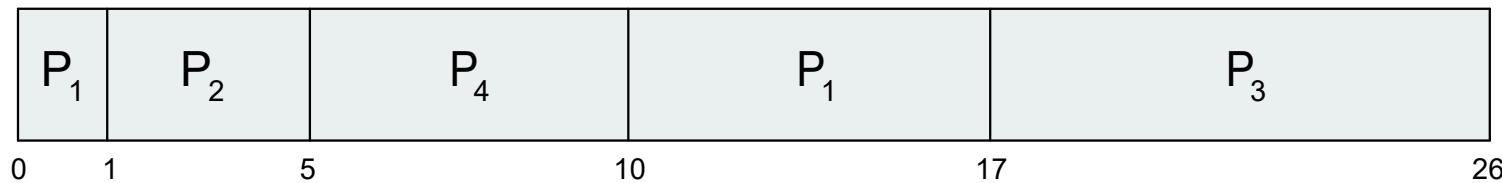
- Sia α che $(1 - \alpha)$ sono minori o uguali ad 1, ogni termine successivo ha minor peso del predecessore

Esempio di Shortest-remaining-time-first

- Si considerano diversi tempi di arrivo e si introduce la prelazione

| <u>Process</u> | <u>Arrival Time</u> | <u>Burst Time</u> |
|----------------|---------------------|-------------------|
| P_1 | 0 | 8 |
| P_2 | 1 | 4 |
| P_3 | 2 | 9 |
| P_4 | 3 | 5 |

- Preemptive SJF Gantt Chart



- Tempo attesa medio = $[(10-1)+(1-1)+(17-2)+5-3]/4 = 26/4 = 6.5$ msec
- Senza prelazione 8.75 msec

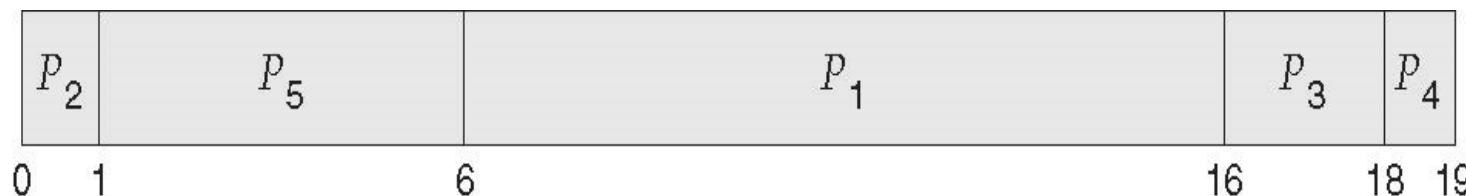
Scheduling con Priorità

- Numero di priorità (intero) associato ad ogni processo
- CPU allocata al processo con più alta priorità (minor intero \equiv maggiore priorità)
 - Preemptive
 - Nonpreemptive
- SJF è un priority scheduling con priorità inversamente proporzionale alla predizione del prossimo CPU
- Problema \equiv **Starvation** – processi a bassa priorità mai eseguiti
- Soluzione \equiv **Aging** – al passare del tempo aumenta la priorità del processo

Esempio di Priority Scheduling

| <u>Process</u> | <u>Burst Time</u> | <u>Priority</u> |
|----------------|-------------------|-----------------|
| P_1 | 10 | 3 |
| P_2 | 1 | 1 |
| P_3 | 2 | 4 |
| P_4 | 1 | 5 |
| P_5 | 5 | 2 |

- Priority scheduling Gantt Chart



- Tempo di attesa medio = 8.2 msec

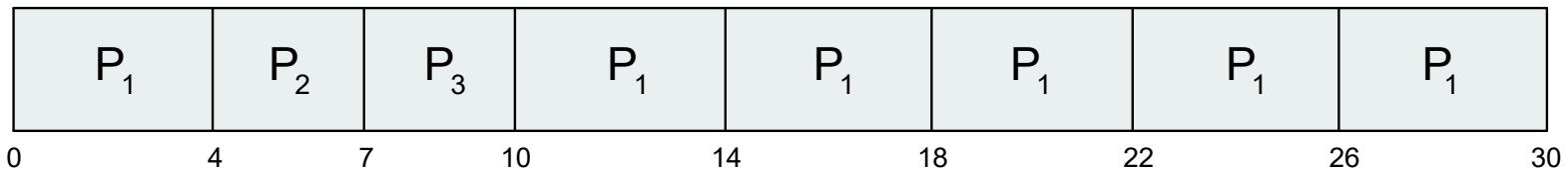
Round Robin (RR)

- Ogni processo riceve una piccola unità di CPU (**time quantum** q), di solito 10-100 msec. A tempo scaduto il processo viene prelazionato e messo alla fine della coda ready
 - Con n processi in coda ready e time quantum q , ogni processo riceve $1/n$ di CPU time in chunk di q unità di tempo
 - Tempi di attesa inferiori a $(n-1)q$ *unità di tempo*.
 - Un timer interrompe ad ogni quanto per schedulare il processo successivo
-
- Performance
 - q largo \Rightarrow FIFO
 - q piccolo $\Rightarrow q$ grande rispetto al context switch altrimenti l'overhead è troppo alto

Esempio di RR con Quanto = 4

| <u>Process</u> | <u>Burst Time</u> |
|----------------|-------------------|
| P_1 | 24 |
| P_2 | 3 |
| P_3 | 3 |

- Gantt chart:



- Tipicamente maggiore turnaround medio di SJF, ma migliore **risposta**
- q deve essere grande rispetto al tempo di context switch
- q di solito tra 10ms e 100ms, context switch < 10 microsec

Quanto di Tempo e Tempo di Context Switch

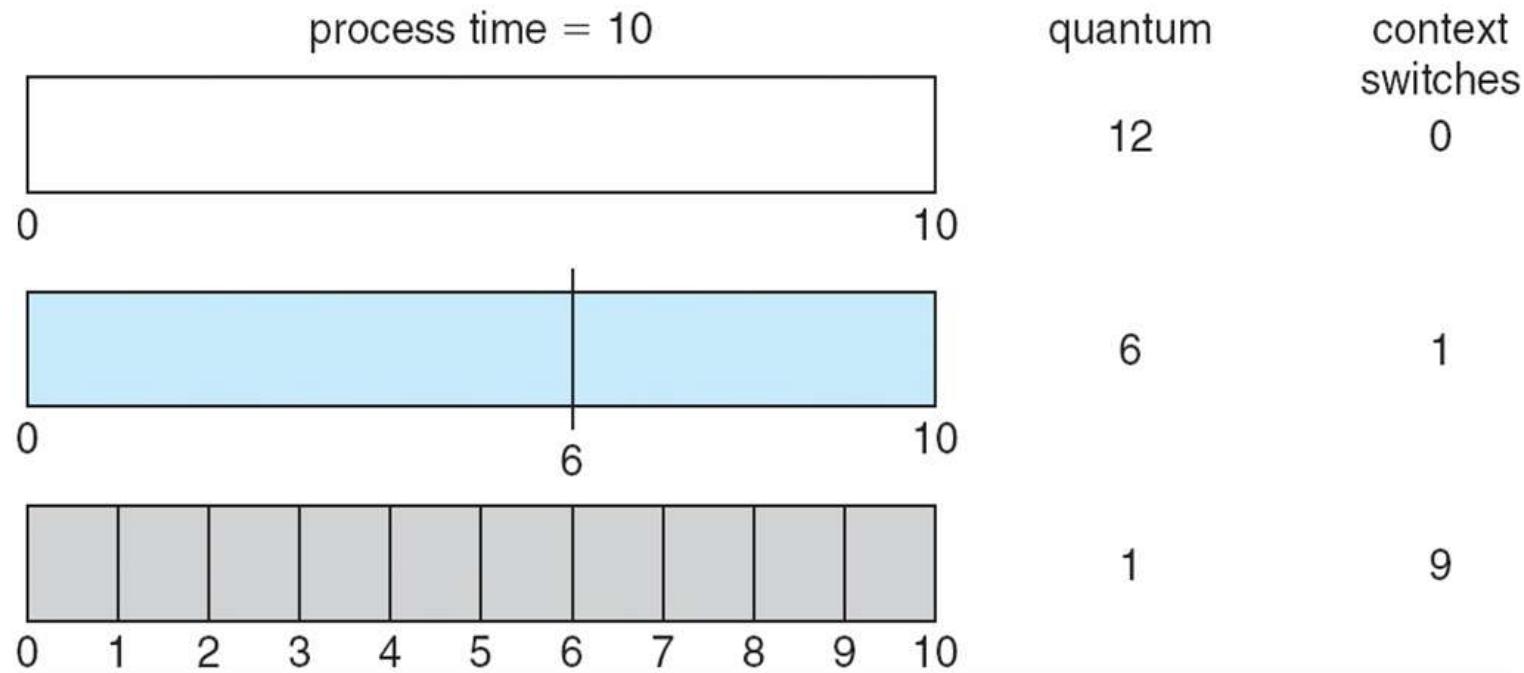
Assumendo un processo di 10 unità di tempo

Se il quanto è 12 il processo completa senza context switch

Se il quanto è 6 ne occorre uno

Se il quanto è 1 ne occorrono 9

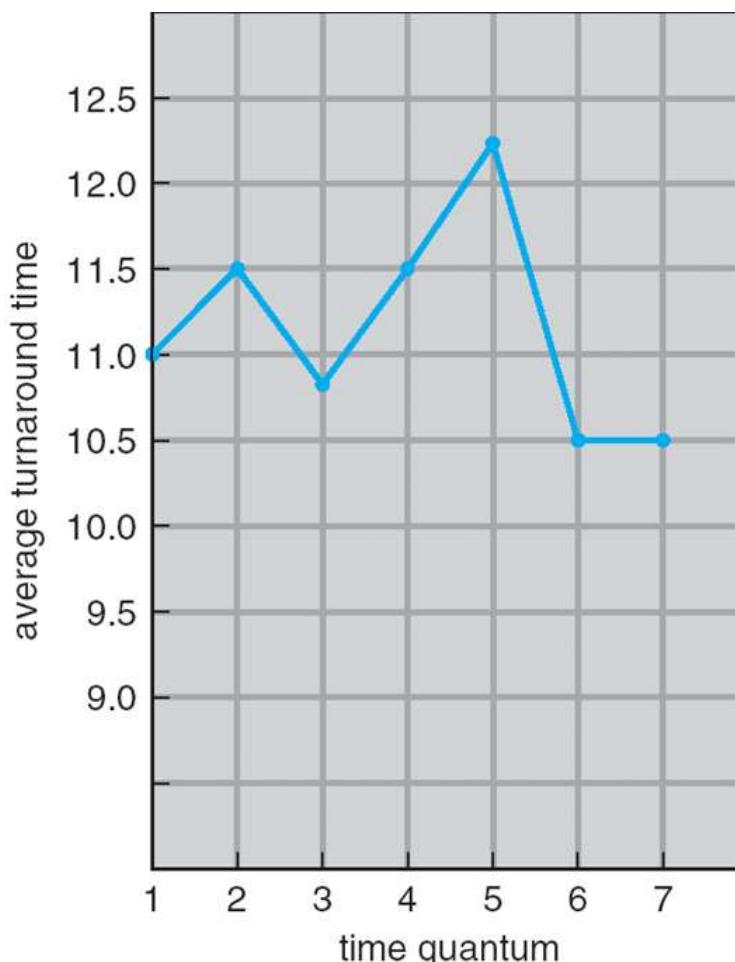
Se cs è 10% del quanto, per ogni quanto si perde 10% di CPU



Tempo di Turnaround varia con il Time Quantum

Tempo di turnaround varia con la dimensione del quanto e non incrementa sempre con la dimensione

Dipende dal tempo di completamento dei processi



| process | time |
|---------|------|
| P_1 | 6 |
| P_2 | 3 |
| P_3 | 1 |
| P_4 | 7 |

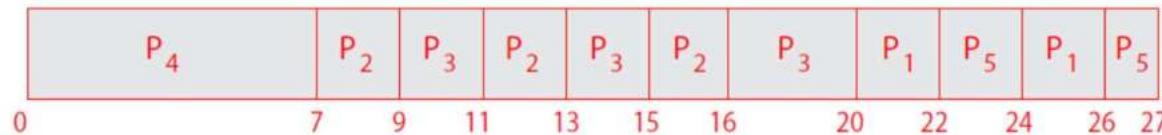
Regola empirica: l'80% dei CPU burst dovrebbero essere più corti di q

RR con Priorità

Si segue la priorità, processi con pari priorità con RR

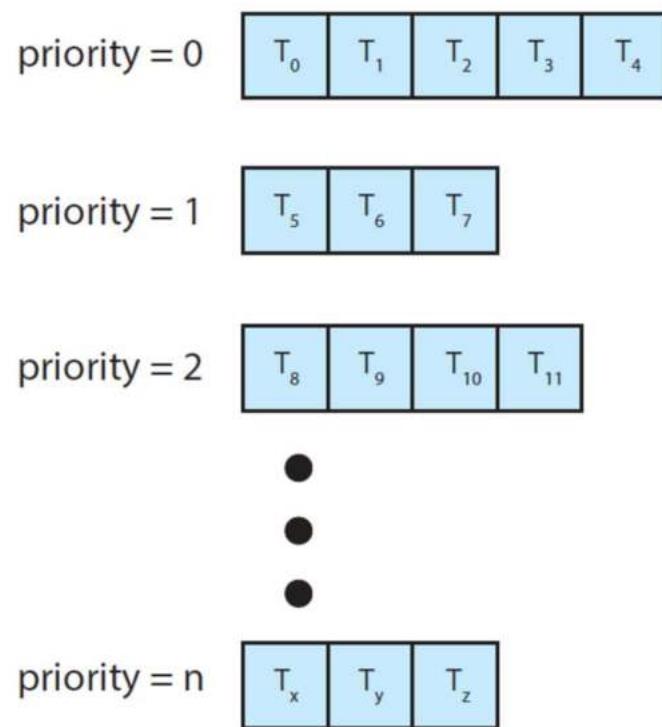
| Process | Burst Time | Priority | millisecondi |
|---------|------------|----------|--------------|
| P_1 | 4 | 3 | |
| P_2 | 5 | 2 | |
| P_3 | 8 | 2 | |
| P_4 | 7 | 1 | |
| P_5 | 3 | 3 | |

Quantum 2 millisecondi



Code Multiple

- Code separate per le differenti priorità
- Assegnazione delle code ai processi (di solito statico)
- Ogni coda ha il suo scheduling

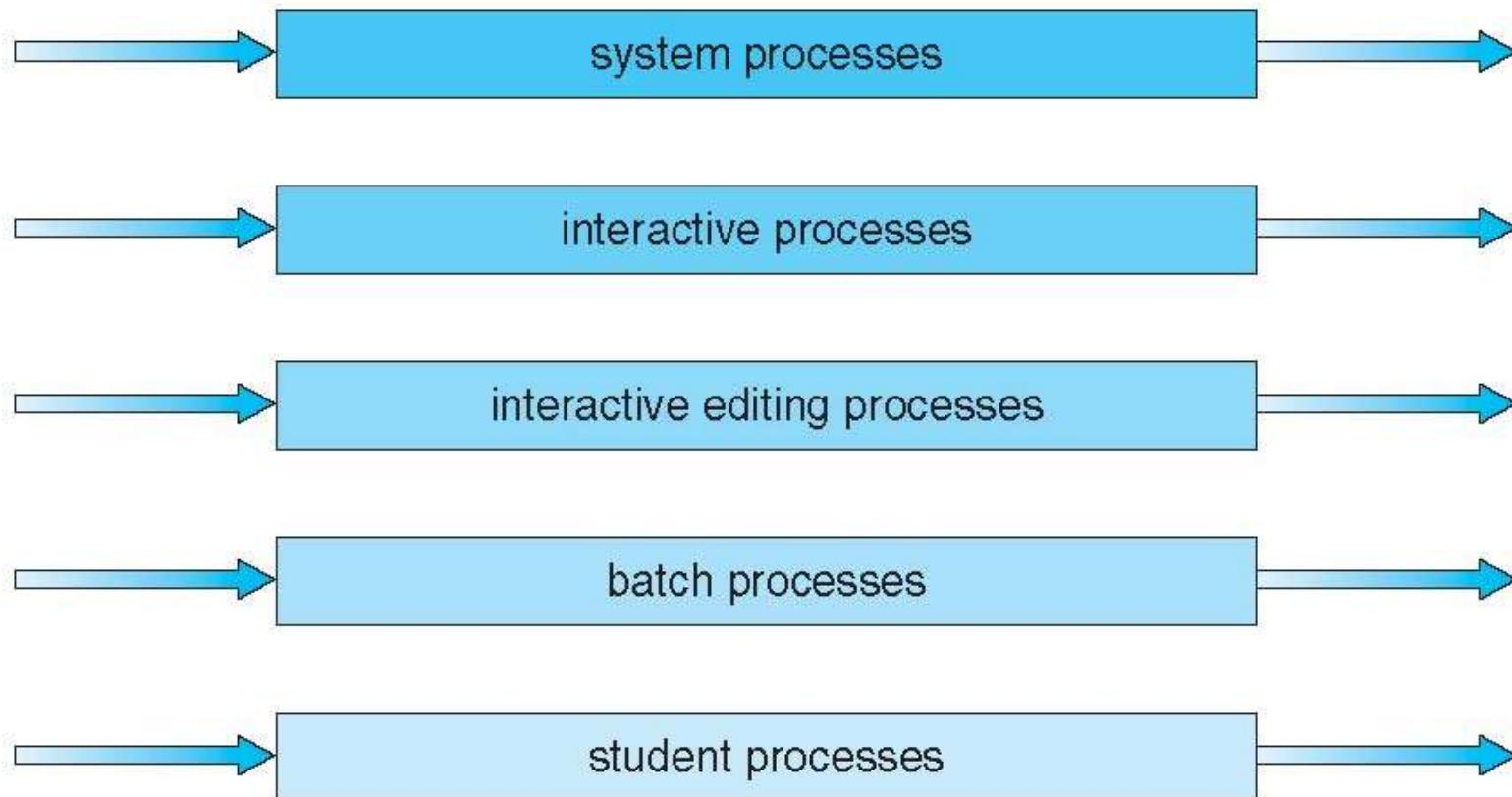


Code Multiple

- La Coda Ready partizionata in diverse code, e.g.:
 - **foreground** (interattiva)
 - **background** (batch)
- Processi assegnati alle code in modo fisso (non passaggio tra code)
- Ogni coda ha un suo algoritmo di scheduling:
 - foreground – RR
 - background – FCFS
- Scheduling tra code:
 - Scheduling a priorità fissata (i.e., serve tutti dal foreground e poi dal background). Possibilità di starvation.
 - Time slice – ogni coda prende un certo quanto di CPU che può schedulare tra i suoi processi
 - ▶ i.e., 80% ai processi foreground in RR 20% ai processi in background in FCFS

Code Multiple

highest priority



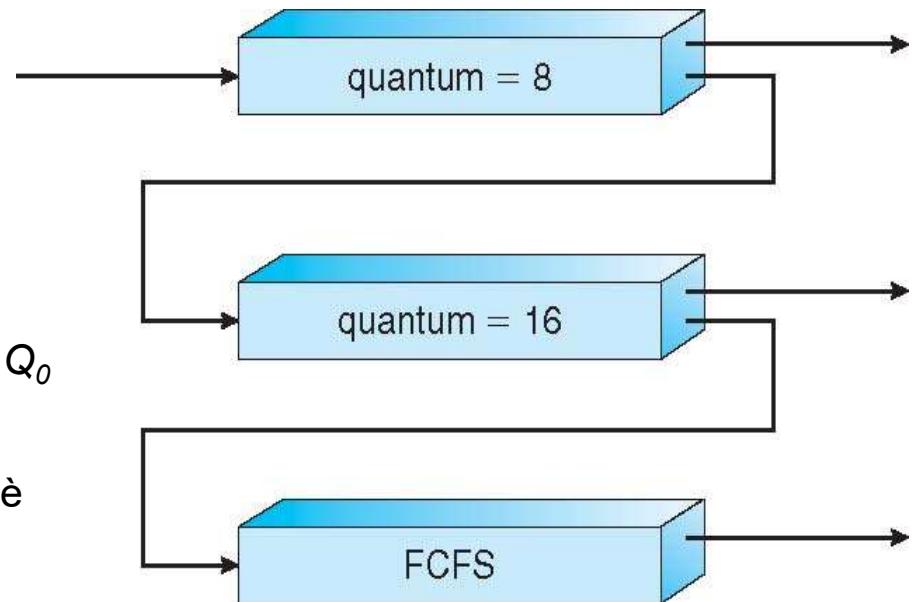
lowest priority

Code Multiple con Feedback

- Un processo può spostarsi tra le code
 - Processi I/O bound con priorità più elevata
 - aging (aumento di priorità nel tempo) per evitare starvation
- Lo scheduler di code multiple con feedback definito da:
 - Numero di code
 - Algoritmi di scheduling per ogni coda
 - Metodi per determinare quando alzare la priorità di processo
 - Metodi per determinare quando abbassare la priorità un processo
 - Metodi per determinare quale coda assegnare ad un processo quando richiede un servizio
- È lo schedulatore più complesso e flessibile

Esempio di Coda Multiple con Feedback

- Si considerino tre code:
 - Q_0 – RR quanto di tempo 8 msec
 - Q_1 – RR quanto di tempo 16 msec
 - Q_2 – FCFS
- Scheduling
 - Prelazione con priorità 0, 1, 2
 - Nuovo processo in Ready messo in coda Q_0
 - Il processo riceve 8 msec di CPU
 - Se non finisce in 8 msec, il processo è mosso in coda Q_1
 - Se vuota Q_0
 - il primo processo in Q_1 riceve 16 msec
 - Se non completa viene prelazionato e si sposta sulla coda Q_2
 - Se vuote Q_0 e Q_1 vengono eseguiti i processi in coda Q_2 con FCFS
 - Schedulatore che dà priorità a processi rapidi e mette in coda quelli più lunghi



Scheduling di Thread

- Distinguere tra thread user-level e kernel-level
- Moderni SO schedulano thread kernel-level, non processi
- Thread user-level gestiti da libreria e kernel non ne è al corrente
- Per essere eseguiti i thread user-level devono essere mappati in kernel-level

Scheduling di Thread

- Per essere eseguiti i thread user-level devono essere mappati in kernel-level
- Modelli many-to-one e many-to-many, la libreria dei thread gestisce gli user-level threads per poi lanciare i lightweight processes (LWPs)
 - Lo schema è detto **Process-Contention Scope (PCS)**: la competizione per lo CPU è tra thread dello stesso processo
 - Tipicamente scheduling con priorità (definita dal programmatore)
- Per decidere il kernel thread da schedulare in CPU il kernel usa un **System-Contention Scope (SCS)**
 - Competizioni tra tutti i thread nel sistema
 - Sistemi che utilizzano modelli one-to-one (Linux, Windows) usano SCS

Pthread Scheduling

- API permettono di specificare PCS o SCS durante la creazione dei thread
- POSIX Pthread permette di specificare entrambe le modalità:
 - PTHREAD_SCOPE_PROCESS usa PCS scheduling
 - PTHREAD_SCOPE_SYSTEM usa SCS scheduling
- Limitato da OS – Linux e Mac OS X permettono solo PTHREAD_SCOPE_SYSTEM

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* first inquire on the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Unable to get scheduling scope\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM");
        else
            fprintf(stderr, "Illegal scope value.\n");
    }
}
```

Pthread Scheduling API

```
/* set the scheduling algorithm to PCS or SCS */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, runner, NULL);

/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);

}

/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */

    pthread_exit(0);
}
```

Scheduling Multi-Processore

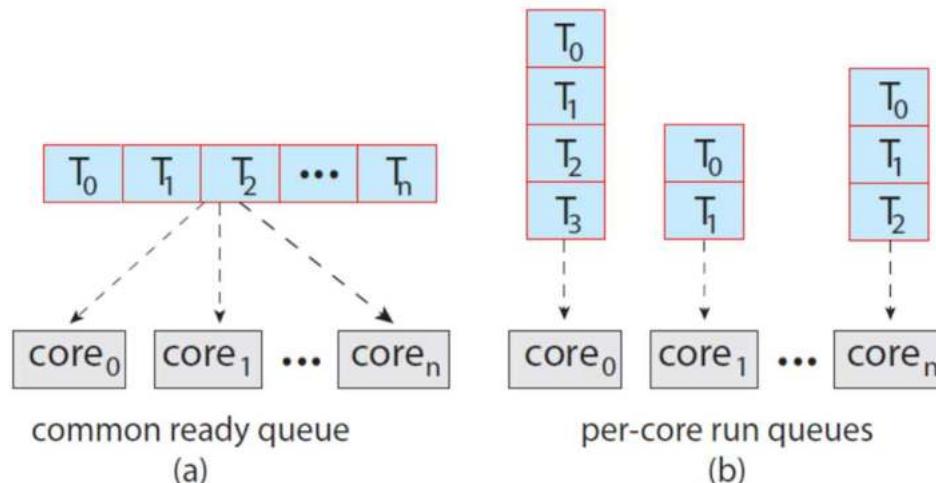
- CPU scheduling più complesso se ci sono più CPU
- Diverse architetture disponibili:
 - **Multicore CPU**
 - **Multithreaded cores**
 - **Sistemi NUMA**
 - **Multiprocessamento eterogeneo**
- Per i primi tre casi assumiamo processori omogeni (stesse capacità)

Scheduling Multi-Processore

- CPU scheduling più complesso se ci sono più processori
- **Homogeneous processors** – processori identici per funzionalità
- **Asymmetric multiprocessing** – un solo processore (master) prende decisioni di scheduling e accede alle strutture dati di sistema senza condivisione di dati
 - Semplifica la gestione, ma il processore server master fa da collo di bottiglia
- **Symmetric multiprocessing (SMP)** – approccio standard: ogni processore fa self-scheduling

Scheduling Multi-Processore

- CPU scheduling più complesso se ci sono più CPU
- **Homogeneous processors** – processori identici per funzionalità
- **Asymmetric multiprocessing** – un solo processore (master) prende decisioni di scheduling e accede alle strutture dati di sistema senza condivisione di dati.
 - Semplifica la gestione, ma il processore server master fa da collo di bottiglia
- **Symmetric multiprocessing (SMP)** – approccio standard: ogni processore fa self-scheduling
 - tutti i processi in una coda ready
 - ognuno ha una sua coda privata di processi ready



Scheduling Multi-Processore

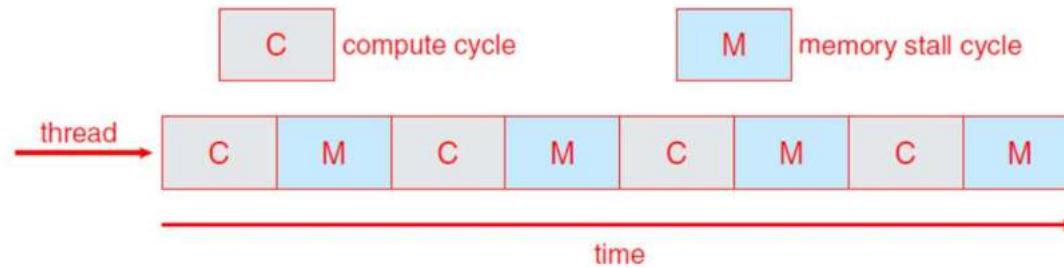
- CPU scheduling più complesso se ci sono più CPU
- **Homogeneous processors** – processori identici per funzionalità
- **Asymmetric multiprocessing** – un solo processore (master) prende decisioni di scheduling e accede alle strutture dati di sistema senza condivisione di dati. Semplifica la gestione, ma il processore server master fa da collo di bottiglia
- **Symmetric multiprocessing (SMP)** – approccio standard: ogni processore fa self-scheduling
 - tutti i processesi in una coda ready oppure ognuno
 - ha una sua coda privata di processi ready
 - Attualmente la soluzione più commune/standard in SMP sono le code private
 - Tutti i sistemi supportano SMP (Linux, Windows, MacOS, Android, etc.)

Scheduling Multi-Processore

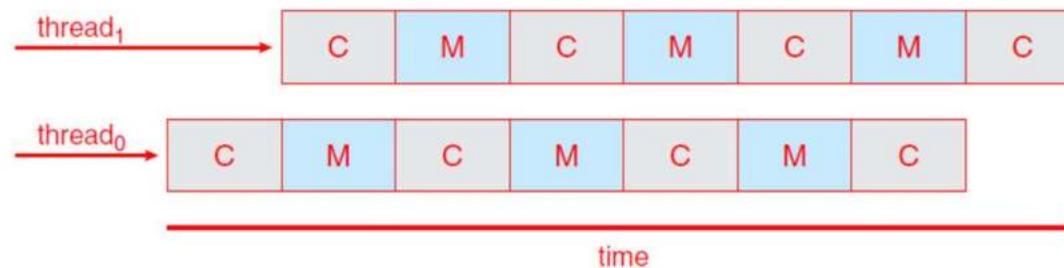
- CPU scheduling più complesso se ci sono più CPU
- **Homogeneous processors** – processori identici per funzionalità
- **Asymmetric multiprocessing** – un solo processore (master) prende decisioni di scheduling e accede alle strutture dati di sistema senza condivisione di dati. Semplifica la gestione, ma il processore server master fa da collo di bottiglia
- **Symmetric multiprocessing (SMP)** – approccio standard: ogni processore fa self-scheduling
 - tutti i processesi in una coda ready oppure ognuno ha una sua coda privata di processi ready
 - Attualmente la soluzione più commune/standard in SMP sono le code private
 - Tutti i sistemi supportano SMP (Linux, Windows, MacOS, Android, etc.)

Scheduling Multi-Processore

- Sistemi multicore
- Ogni core appare al SO come una CPU separata
- Core veloci, ma problemi di **memory stall** attesa di dati pronti in memoria

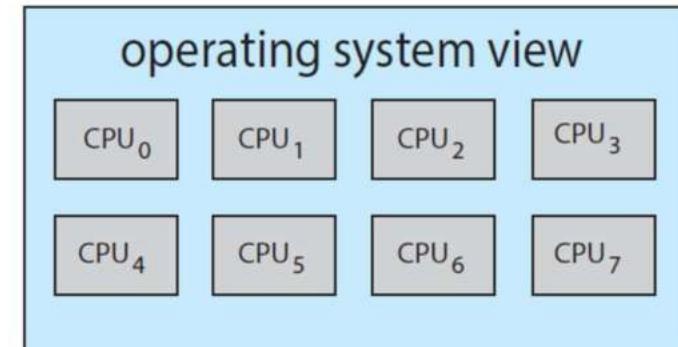
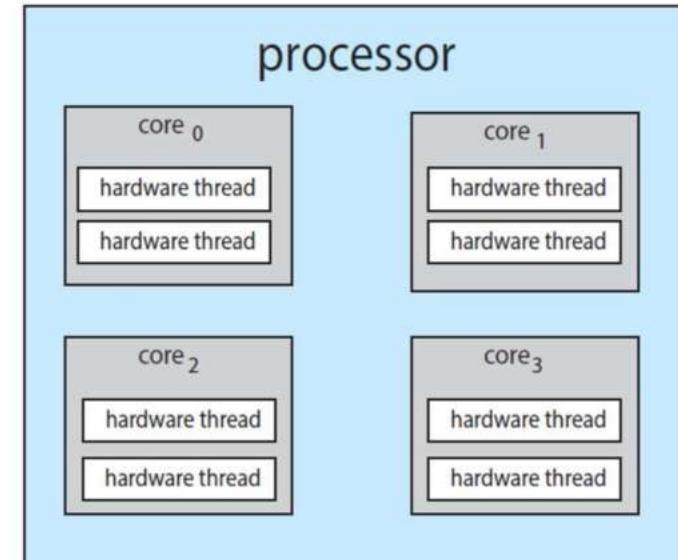


- Più thread pronti per core: **chip multithreading (CMT)**



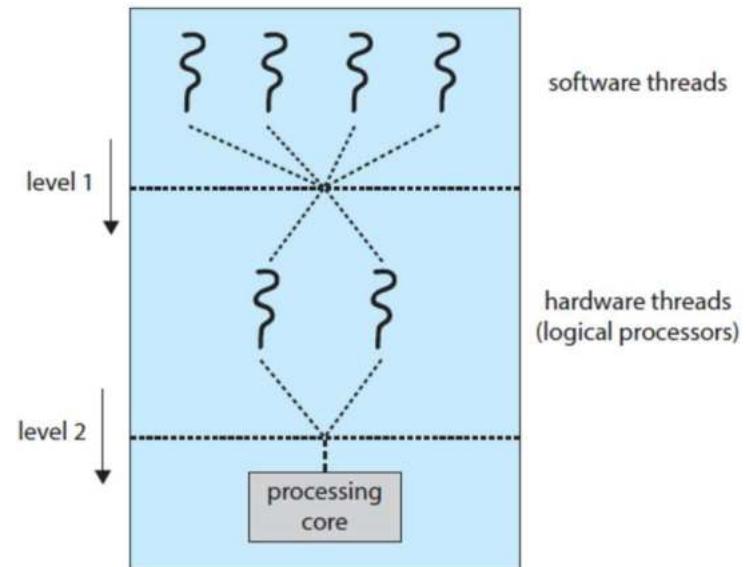
Scheduling Multi-Processore

- Sistemi multicore
- Ogni core appare al SO come una CPU separata
- Core veloci, ma problemi di **memory stall** attesa di dati pronti in memoria
- Più thread pronti per core: **chip multithreading (CMT)**
- Per OS è come avere più CPU
intel chiama hyperthreading
- Es. i7 ha 2 thread per core, 4 core
Oracle Spark M7 8 thread per core
su 8 core, come 64 CPU



Scheduling Multi-Processore

- Sistemi multicore
- Ogni core appare al SO come una CPU separata
- Core veloci, ma problemi di **memory stall** attesa di dati pronti in memoria
- Più thread pronti per core: **chip multithreading (CMT)**
- Per OS è come avere più CPU
intel chiama hyperthreading
- Es. i7 ha 2 thread per core, 4 core
Oracle Spark M7 8 thread per core
su 8 core, come 64 CPU
- Due livelli di scheduling:
 - Software threads (SO)
 - Hardware threads



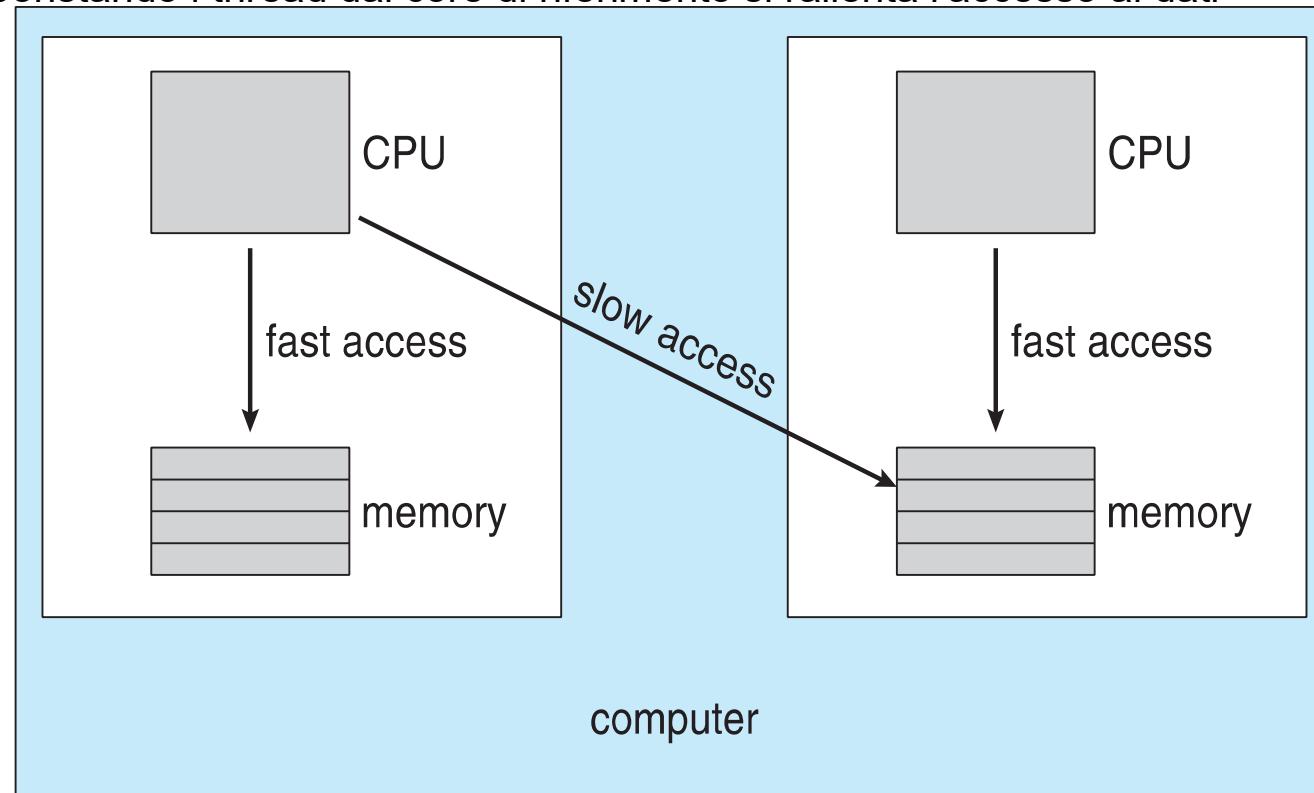
Multiple-Processor Scheduling – Load Balancing

- Se Symmetric Multiprocessing (SMP) e code private:
 - **Load balancing:** necessario bilanciare il carico tra i processori
- Due metodi di Load balancing:
 - **Push migration** – un processo verifica continuamente il carico dei processori, se trova uno sbilanciamento sposta/spinge il task dalla CPU sovraccarica alle altre
 - **Pull migration** – i processori in idle prendono/tirano i task in attesa sulle code dei processori occupati
 - Esempio: Linux implementa entrambe le strategie
- **Processor affinity** – processo ha affinità per il processore su cui è in running
 - **soft affinity** – tentativo di mantenere il processore del thread
 - **hard affinity** – specifica il sottoinsieme di processori per il thread
 - Esempio, Linux implementa soft, ma supporta entrambi
 - Con code private per core più semplice affinity

NUMA e CPU Scheduling

□ Non-uniform Memory Access (NUMA)

- Due chip con la propria CPU e memoria
- Accessi in memoria a velocità diversa
- C'è un conflitto tra problematiche di accesso in memoria e Load balancing
 - ▶ Sponstando i thread dai core di riferimento si rallenta l'accesso ai dati



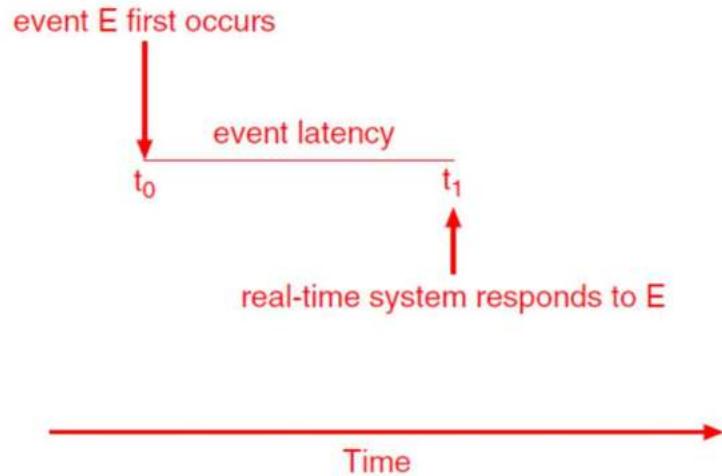
Gli algoritmi di memory-placement possono considerare l'affinity

Heterogeneous Multiprocessing

- Alcuni sistemi hanno architetture multicore non omogenee
 - Diverso clock, diverso consumo di energia, etc.
 - Non asimmetrici, perché core omogenei per capacità di calcolo
 - Ma diversi consumi: Big core e Little core (processori ARM)
 - Big core veloci ma consumano
 - Little core lenti ma economici
 - Vantaggio per sistemi mobile
 - **Algoritmi di scheduling finalizzati al risparmio di energia**

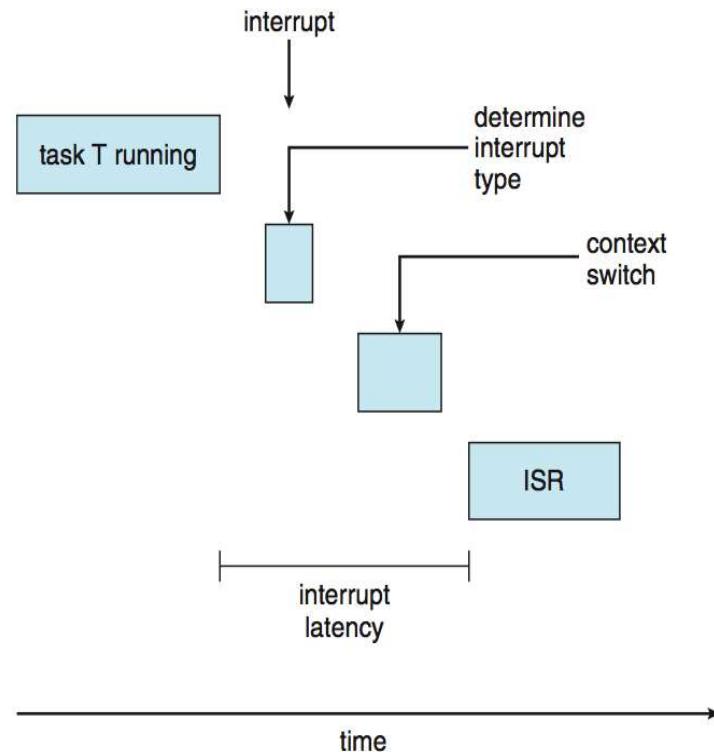
Real-Time CPU Scheduling

- Sistemi Real-Time
- **Soft real-time system**
 - non garanzie su quando un processo verrà schedulato, solo processi critici preferiti a non critici
- **Hard real-time system**
 - task servito prima di una deadline
- Latenza di un evento:
 - Tempo trascorso tra evento e risposta del sistema



Real-Time CPU Scheduling

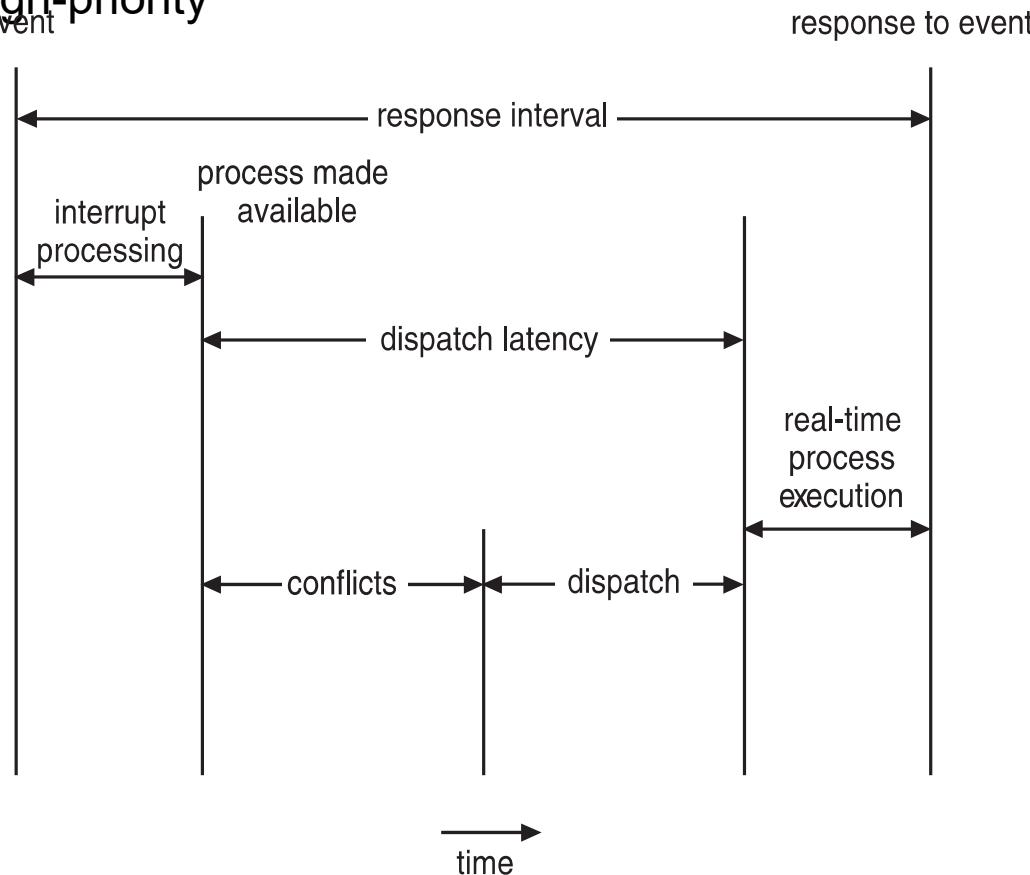
- Sistemi Real-Time
- **Soft real-time system**
 - non garanzie su quando un processo verrà schedulato, solo processi critici preferiti a non critici
- **Hard real-time system**
 - task servito prima di una deadline
- Due tipi di latenze incidono sulla prestazione
 1. **Interrupt latency** – tempo dall'arrivo dell'interrupt allo start della routine di servizio
 2. **Dispatch latency** – tempo di scheduling per fare lo switch di un altro processo



Real-Time CPU Scheduling

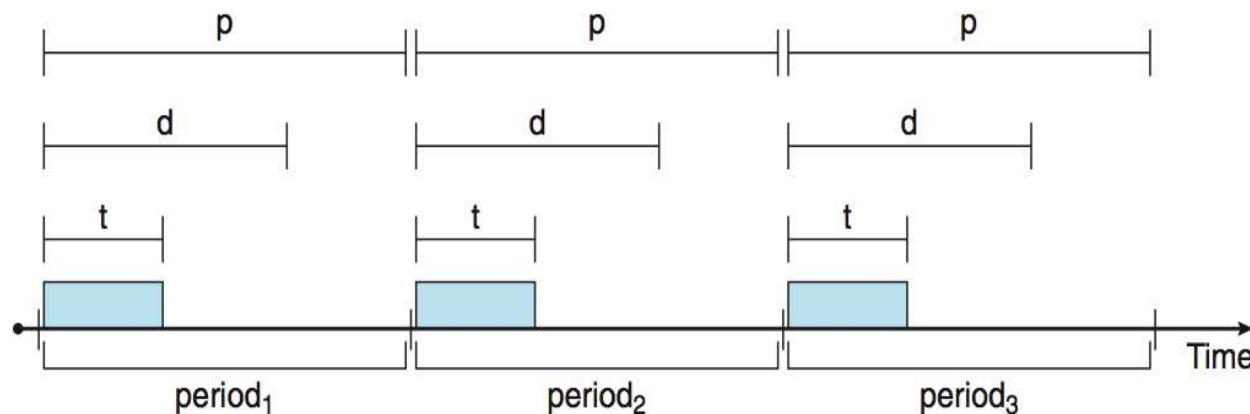
- Dispatch latency: fase di conflitto e fase di dispatch del processo
- Fase di gestione del **conflitto** durante la dispatch latency:
 1. Prelazione di ogni processo in kernel mode
 2. Rilascio delle risorse occupate dai processi low-priority quando richieste dai processi high-priority

Latenza relativa al dispatch: periodo di tempo necessario al dispatcher per bloccare un processo e avviare un altro



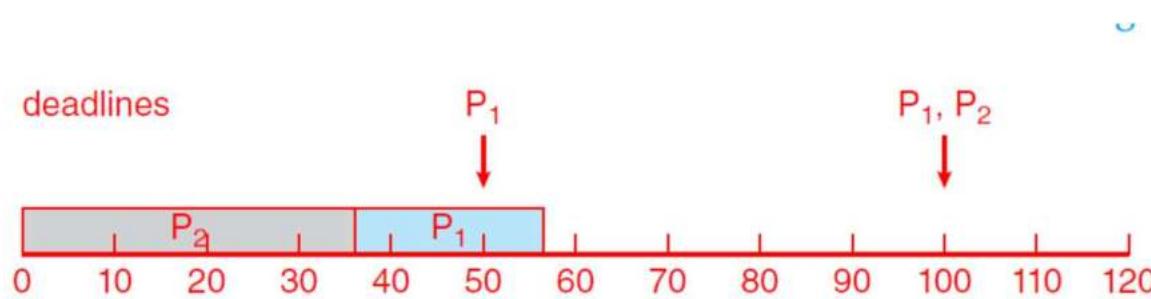
Priority-based Scheduling

- Nel real-time scheduling lo scheduler deve supportare preemptive e priority-based scheduling
 - Ma questo garantisce solo il soft real-time
- Per hard real-time deve anche garantire le deadlines
- Consideriamo task **periodici**
 - Richiedono la CPU a intervalli costanti
 - Otttenuta la CPU ha un tempo di processo t , deadline d , e periodo p
 - $0 \leq t \leq d \leq p$
 - **Rate** di un task periodico $1/p$



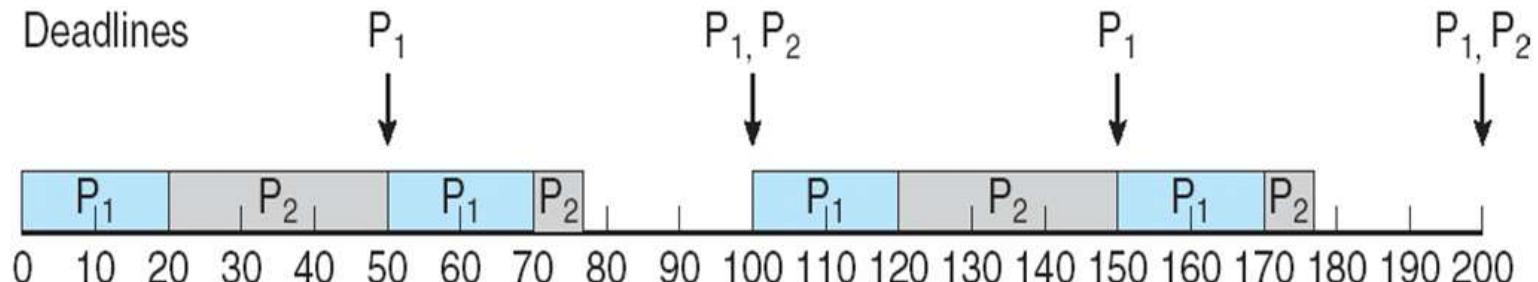
Rate Monotonic Scheduling

- Priorità assignata in base all'inverso del periodo
 - Periodi brevi = priorità alta;
 - Periodi lunghi = priorità bassa
 - Priorità alta a chi interviene più spesso
- Esempio:
 - Periodi 50 e 100, tempo di processamento 20, 35
 - Deadline: terminare prima del prossimo periodo
 - Uso CPU per P_1 di è $20/50 = 0.4$, per P_2 di è $35/100 = 0.35$, tot 0.75
 - Assumendo P_2 con priorità più alta di P_1 deadline persa per P_1



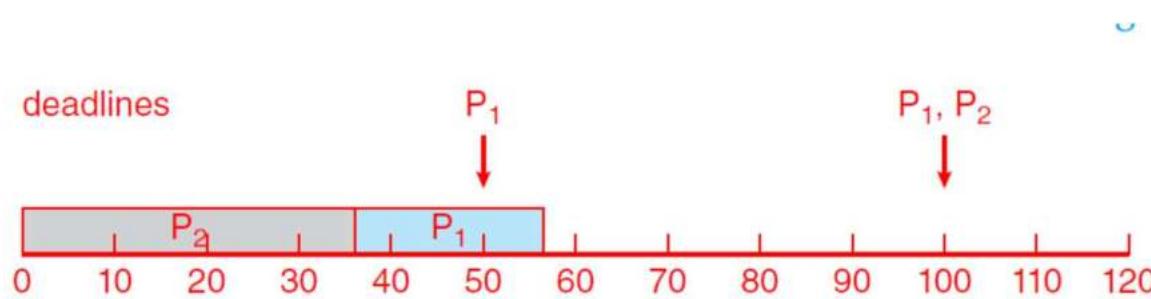
Rate Monotonic Scheduling

- Priorità assignata in base all'inverso del periodo
- Periodi brevi = priorità alta;
- Periodi lunghi = priorità bassa
- Esempio:
 - Periodi 50 e 100, tempo di processamento 20, 35
 - Deadline: terminare prima del prossimo period
 - Uso CPU per P_1 di è $20/50 = 0.4$, per P_2 di è $35/100 = 0.35$, tot 0.75
 - Assumendo RMS:
 - ▶ P_1 con priorità più alta di P_2 entrambe le deadline rispettate



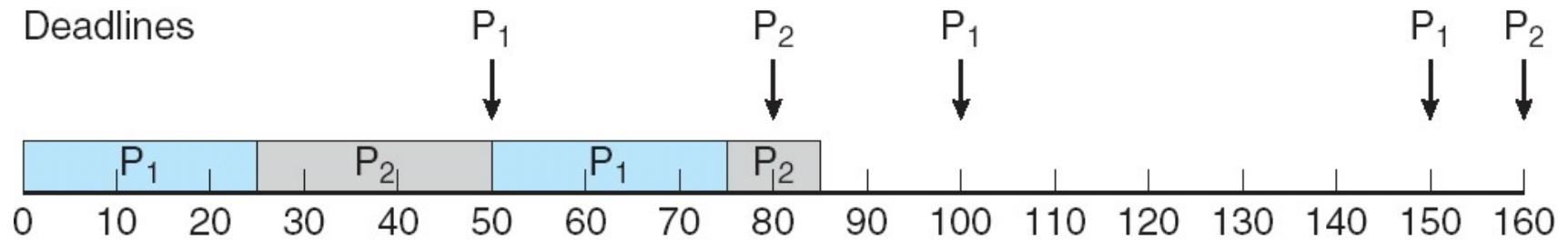
Rate Monotonic Scheduling

- Priorità assignata in base all'inverso del periodo
 - Periodi brevi = priorità alta;
 - Periodi lunghi = priorità bassa
 - Priorità alta a chi interviene più spesso
- Esempio:
 - Periodi 50 e 100, tempo di processamento 20, 35
 - Deadline: terminare prima del prossimo periodo
 - Uso CPU per P_1 di è $20/50 = 0.4$, per P_2 di è $35/100 = 0.35$, tot 0.75
 - Assumendo P_2 con priorità più alta di P_1 deadline persa per P_1



Deadline persa con il Rate Monotonic Scheduling

- Se P_1 ha periodo 50 e CPU brust 25 e P_2 periodo 80 e CPU brust 35
- Utilizzo CPU $25/50 + 35/80 = 0.94$
- P_2 interrotta da P_1 quando riprende finisce a 85, ma doveva finire ad 80



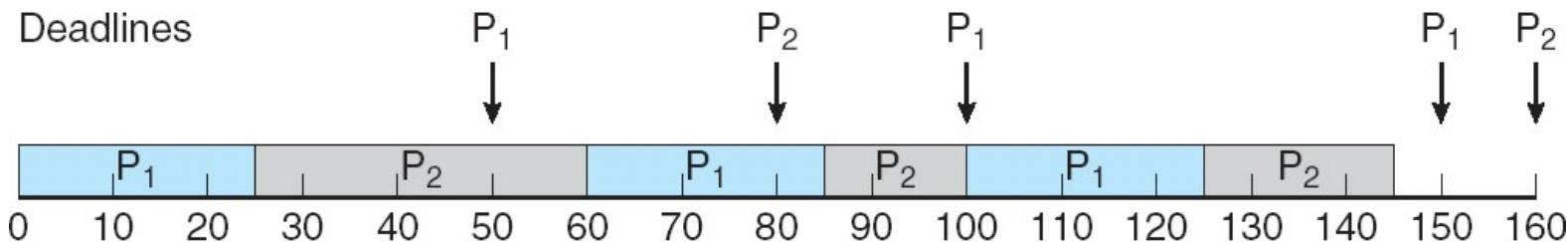
- Non sfrutta completamente la CPU

$$N(2^{1/N} - 1)$$

- Con 1 processo 100%, con 2 processi 83%, con al crescere di N 69%

Earliest Deadline First Scheduling (EDF)

- Priorità sono assegnate secondo le deadline:
 - prima la deadline, più alta è la priorità;
 - più lontana è la deadline, più bassa è la priorità



P1: p1 = 50, t1 = 25; P2: p2 = 80, t2= 35

Non richiede la periodicità dei processi e neppure CPU brust costante

Ogni processo deve dichiarare la deadline quando diventa runnable

Teoricamente ottimo (se utilizzo sotto 100% rispetta la deadline) però va considerate il context switch e il costo della gestione dell'interrupt

Earliest Deadline First Scheduling (EDF)

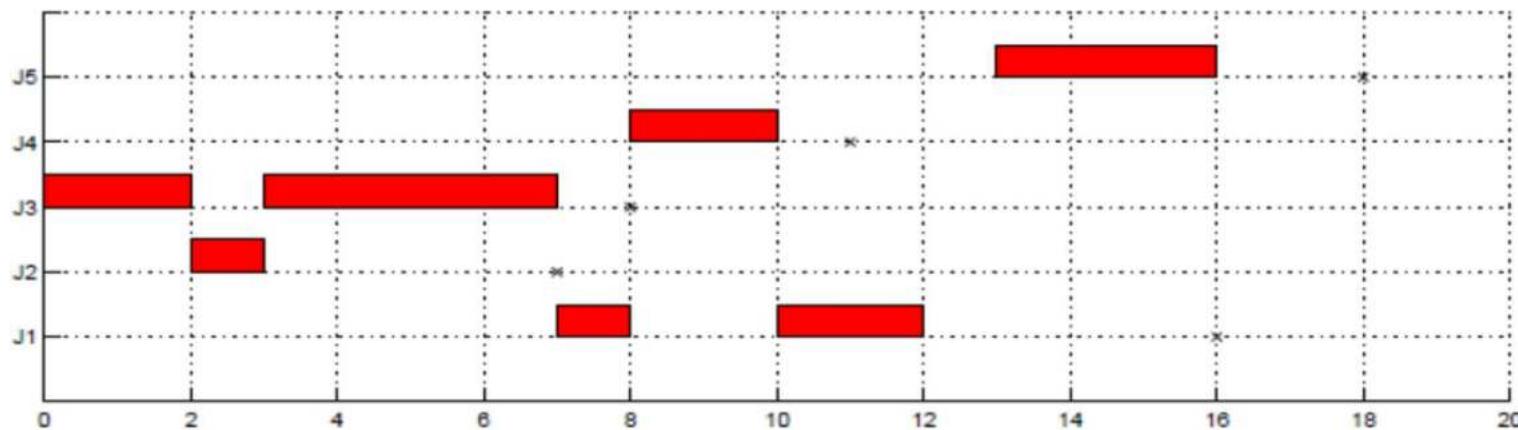
- Priorità sono assegnate secondo le deadline
- Esempio:
 - Tracciare il Gantt secondo EDF verificando se le scadenze sono rispettate (scheduling ammissibile)

| Processo | Tempo di arrivo | Esecuzione | Scadenza |
|----------|-----------------|------------|----------|
| J_1 | 0 | 3 | 16 |
| J_2 | 2 | 1 | 7 |
| J_3 | 0 | 6 | 8 |
| J_4 | 8 | 2 | 11 |
| J_5 | 13 | 3 | 18 |

Earliest Deadline First Scheduling (EDF)

- Priorità sono assegnate secondo le deadline
- Esempio:
 - Tracciare il Gantt secondo EDF verificando se le scadenze sono rispettate (ammissibile)

| Processo | Tempo di arrivo | Esecuzione | Scadenza |
|----------|-----------------|------------|----------|
| J_1 | 0 | 3 | 16 |
| J_2 | 2 | 1 | 7 |
| J_3 | 0 | 6 | 8 |
| J_4 | 8 | 2 | 11 |
| J_5 | 13 | 3 | 18 |



Scheduling a Quote Proporzionali

- Quote proporzionali di CPU preallocate per le applicazioni
- T quote di CPU devono essere preallocate tra tutti i processi
- Un'applicazione riceve N quote con $N < T$
 - L'algoritmo assicura l'applicazione riceverà N / T del tempo totale del processore
- Esempio $T = 100$ da dividere su A, B, C. A che riceve 50, B riceve 15, e C riceve 20. Rimangono quote per altri processi
- Algoritmo di admission control assegna le quote rimanenti ai nuovi processi nel rispetto delle quote già assegnate

POSIX Real-Time Scheduling

POSIX.1b standard API per gestire real-time threads

Definisce due classi di scheduling per real-time threads:

1. SCHED_FIFO - threads sono schedulati con strategia FCFS con coda FIFO.
No time-slicing per thread con priorità uguale
2. SCHED_RR - simile a SCHED_FIFO ma con time-slicing per thread di pari priorità

Definisce due funzioni per prendere e settare la politica di scheduling:

1. `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
2. `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`

POSIX Real-Time Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
    int i, policy;
    pthread_t_tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* get the current scheduling policy */
    if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
        fprintf(stderr, "Unable to get policy.\n");
    else {
        if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
        else if (policy == SCHED_RR) printf("SCHED_RR\n");
        else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
    }
}
```

POSIX Real-Time Scheduling API (Cont.)

```
/* set the scheduling policy - FIFO, RR, or OTHER */
if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
    fprintf(stderr, "Unable to set policy.\n");

/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, runner, NULL);

/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);

}

/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```

Virtualizzazione e Scheduling

- Con Virtualizzazione si schedula con SO ospiti multipli sulla CPU(s)
- Ogni ospite esegue il suo scheduling
 - Ma in realtà non ha una sua CPU e sta schedulando pensando di avere l'intera CPU
 - Può portare a risposte rallentate
 - Lo stesso clock del Sistema può essere rallentato
- Si possono perdere i benefici di un buon algoritmo di scheduling implementato sul SO ospite

Esempi di SO

- Linux scheduling
- Windows scheduling
- Solaris scheduling

Scheduling Linux prima della versione 2.5

- Prima della versione kernel 2.5 variazione dell'algoritmo di scheduling standard di UNIX
- Code multiple con feedback gestite con RR
 - Prelazione sulle code con priorità
 - Priorità assegnata sulla base dell'utilizzo di CPU (maggiore utilizzo, minore priorità)
 - Periodicamente il kernel calcola la CPU utilizzata da un processo dall'ultimo controllo, valore inversamente proporzionale alla priorità
- Metodo non flessibile e poco adatto a Symmetric Multiprocessing (SMP)

Scheduling Linux versione 2.5

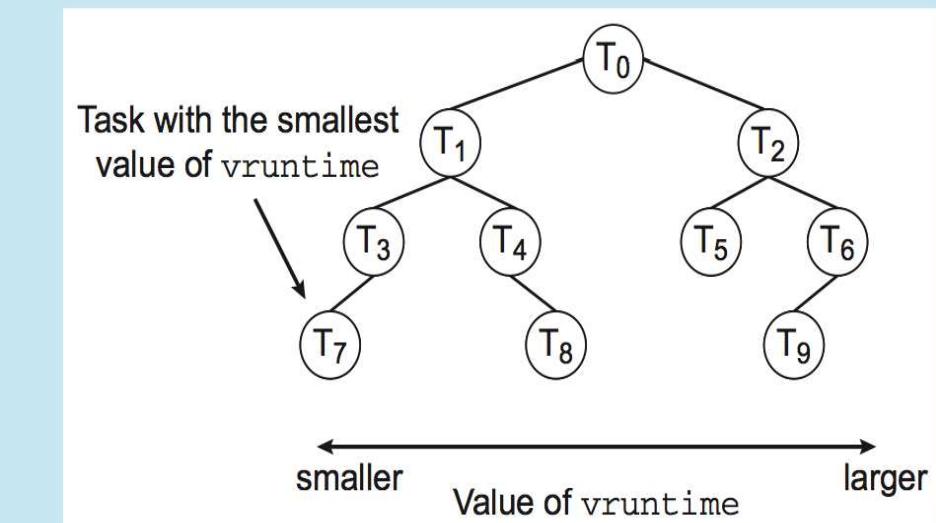
- Con la versione 2.5 si propone uno scheduler a tempo costante $O(1)$ indipendente dal numero di task nel sistema:
 - Preemptive e priority based
 - Due intervalli di priorità: processi time-sharing (detti **nice**) e real-time
 - **Real-time** da 0 a 99 e valori **nice** da 100 a 140
 - Si mappano in priorità globali con valori bassi che indicano alta priorità
 - Priorità più alte prendono time-slice q maggiori
 - I task sono in esecuzione finché hanno il time slice (**active**)
 - Quando scade non vanno in esecuzione finché tutti gli altri tasks non usano lo slice
 - Tutti i task eseguibili tracciati con code per-CPU
 - Due array di priorità (active, expired)
 - Task indicizzati con la priorità
 - Finiti gli attivi gli array sono scambiati
 - Buon funzionamento, ma tempi di risposta non soddisfacenti per processi in time-sharing

Linux Scheduling in Versione 2.6.23 +

- **Completely Fair Scheduler** (CFS)
- **Diverse classi di scheduling**
 - Ognuna con priorità specifica
 - Scheduler prende la più alta nella classe più alta
 - Non quantum based con allocazione fissa, ma basata su proporzione di CPU time
 - 2 classi di scheduling, altre possono essere aggiunte
 1. default
 2. real-time
- Quantum calcolato sulla base di un **nice value** da -20 a +19
 - Più basso, più alta la priorità
 - Calcolo della **latenza target** – intervallo di tempo durante il quale un task deve andare in run almeno una volta
 - La latenza target può aumentare con il numero dei task attivi
- CFS scheduler mantiene un **tempo di esecuzione virtuale** per ogni task in variabile **vruntime** (per quanto è stato eseguito)
 - Associato a fattore di decadimento basato sulla priorità del task
 - ▶ a basse priorità corrisponde alto fattore di decadimento
 - ▶ Default: virtual run time = actual run time
- Per decidere il prossimo task lo scheduler prende il task con il minore virtual run time

Performance CFS

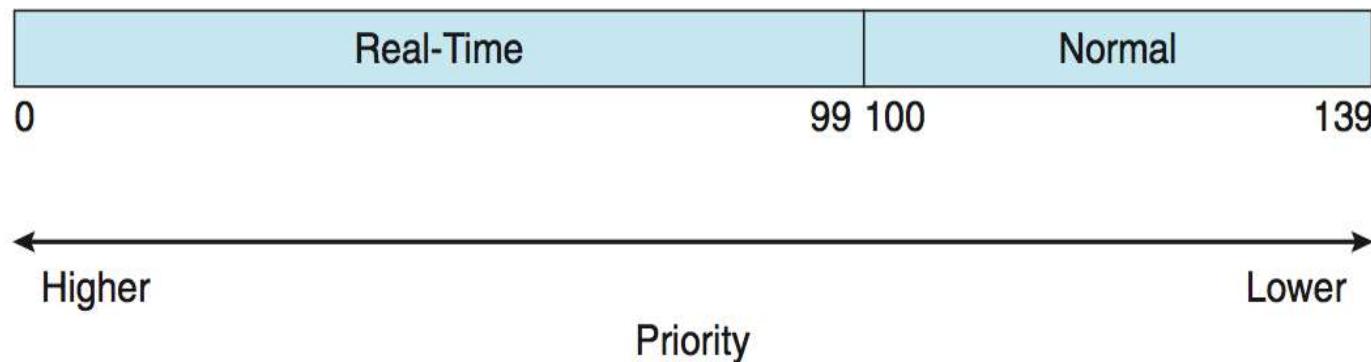
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(\lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

Linux Scheduling

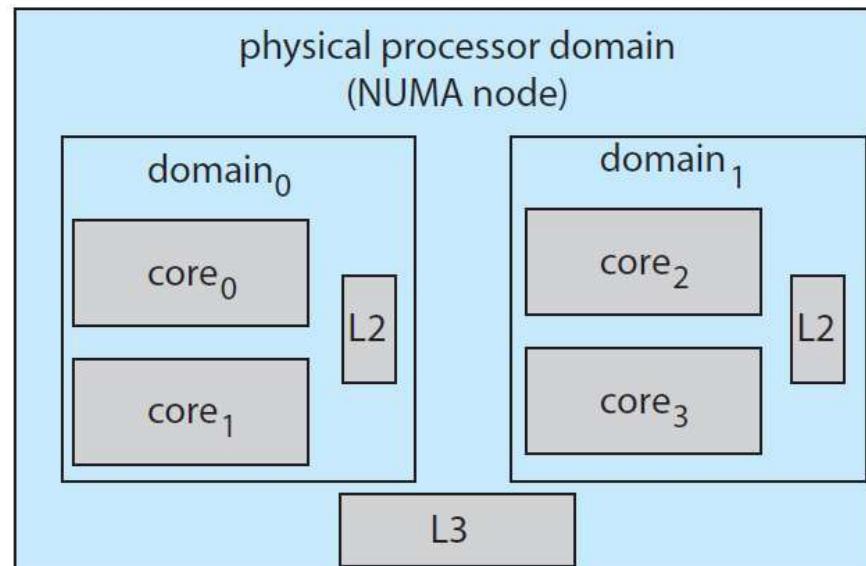
- Real-time scheduling in POSIX.1b
 - Task real-time con priorità statiche
- Real-time hanno valori più bassi di quelli dei task normali e sono mappati in uno schema di priorità globale



- Le priorità relative dei processi real-time sono assicurate
- Il kernel non fornisce garanzia sui tempi di attesa dei processi pronti
- Se interruzione per real-time arriva mentre il kernel serve una chiamata di sistema, il processo real-time attende

Linux Scheduling

- Scheduler CSF supporta bilanciamento del carico, è compatibile con NUMA - reduce al minimo migrazione dei thread
- Il carico del thread è dato da una combinazione di priorità e tasso medio di utilizzo della CPU
 - Carico basso, ma alta priorità simile a carico alto e bassa priorità
 - Somma dei load indica il carico della coda e si può ribilanciare
 - Domini di scheduling per ridurre il costo del bilanciamento
 - Bilanciamento prima nel dominio



Windows Scheduling

- Windows usa un priority-based preemptive scheduling
- Thread a priorità più alta vengono lanciati
- **Dispatcher** è lo scheduler
- Thread eseguono fino al (1) blocco, (2) fine del time slice, (3) prelazionato da un thread a priorità più alta
- I thread real-time possono prelazionare i non-real-time
- 32-livelli di schema di priorità
- **Variable class** 1-15, **real-time class** 16-31
- Priorità 0 è il thread di gestione della memoria
- Code per ogni priorità
- Cerca i thread in ogni coda
- Se non trova un runnable thread, allora lancia il **thread idle**

Windows Priority Classes

- Win32 API identifica molte classi di priorità per i processi
 - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS,
ABOVE_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS,
BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
 - Tutti variabili eccetto REALTIME
- Un thread in una priority class ha una priorità relativa
 - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL,
LOWEST, IDLE
- Priority class e relative priority si combinano per dare una priorità numerica
- La priorità di base è NORMAL per la classe di appartenenza
- Se il quantum si esaurisce la priorità si abbassa, ma mai sotto la base

Windows Priorities

| | real-time | high | above normal | normal | below normal | idle priority |
|---------------|-----------|------|--------------|--------|--------------|---------------|
| time-critical | 31 | 15 | 15 | 15 | 15 | 15 |
| highest | 26 | 15 | 12 | 10 | 8 | 6 |
| above normal | 25 | 14 | 11 | 9 | 7 | 5 |
| normal | 24 | 13 | 10 | 8 | 6 | 4 |
| below normal | 23 | 12 | 9 | 7 | 5 | 3 |
| lowest | 22 | 11 | 8 | 6 | 4 | 2 |
| idle | 16 | 1 | 1 | 1 | 1 | 1 |

Windows Priority Classes

- Se il thread esce dal wait la priorità viene aggiornata a seconda di cosa si attendeva:
 - Es. I/O favorito, memoria meno
- Processi in foreground hanno un 3x boost di priorità
- Windows 7 ha aggiunto uno **user-mode scheduling (UMS)**
 - Le applicazioni creano e gestiscono thread indipendente dal kernel
 - Permette un uso più efficiente per un largo numero di thread

Solaris

- Priority-based scheduling
- Sei classi
 - Time sharing (default) (TS)
 - Interactive (IA)
 - Real time (RT)
 - System (SYS)
 - Fair Share (FSS)
 - Fixed priority (FP)
- Ogni thread in una classe
- Ogni classe ha il suo algoritmo di scheduling
- Il default è time sharing con multi-level feedback queue
 - Il time slice è variabile, la priorità è alta quanto piccolo
 - Processi interattivi a priorità più alta dei CPU bound

Solaris Dispatch Table

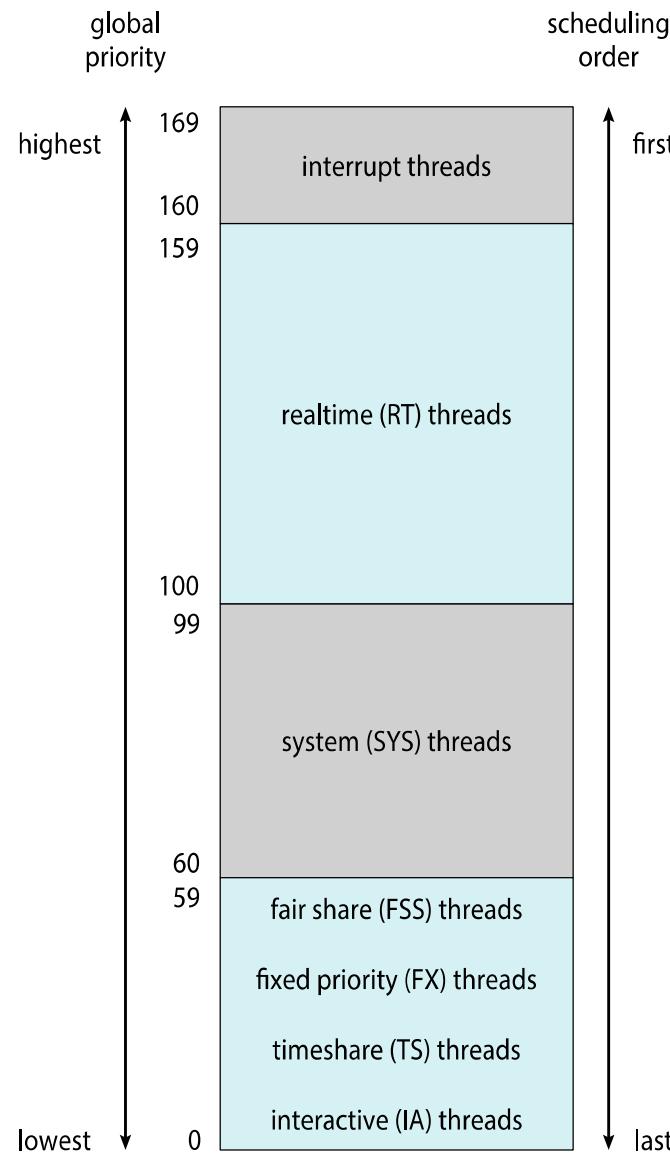
Time-sharing e processi interattivi

| priority | time quantum | time quantum expired | return from sleep |
|----------|--------------|----------------------|-------------------|
| 0 | 200 | 0 | 50 |
| 5 | 200 | 0 | 50 |
| 10 | 160 | 0 | 51 |
| 15 | 160 | 5 | 51 |
| 20 | 120 | 10 | 52 |
| 25 | 120 | 15 | 52 |
| 30 | 80 | 20 | 53 |
| 35 | 80 | 25 | 54 |
| 40 | 40 | 30 | 55 |
| 45 | 40 | 35 | 56 |
| 50 | 40 | 40 | 58 |
| 55 | 40 | 45 | 58 |
| 59 | 20 | 49 | 59 |

Priorità abbassata quando il tempo scade e alzata quando trona dal wait

Solaris Scheduling

Mappatura delle priorità delle classi in priorità globali



Solaris Scheduling

- Scheduler converte priorità class-specific priorities in priorità globali per-thread
 - Thread con priorità più alta lanciati
 - finché (1) bloccati, (2) usano il time slice, (3) preempted da thread a priorità maggiore
 - Threads con stessa priorità selezionati con RR

Valutazione Algoritmi

- Come selezionare algoritmi di CPU-scheduling per un SO?
- Determinare un criterio, poi valutare gli algoritmi
 - **Esempio 1:** massimizzare l'utilizzo della CPU, con tempo massimo di risposta inferiore a 300 millisecondi
 - **Esempio 2:** massimizzare la produttività, con tempo di completamento proporzionale (in media) al tempo di esecuzione effettivo

Valutazione Algoritmi

- **Valutazione analitica**

- Definisce una formula o dati da valutare

- **Modelli deterministicici**

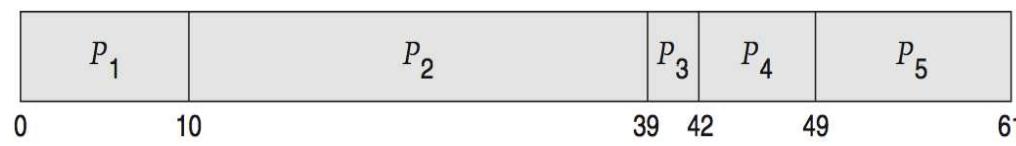
- Assume un particolare carico e definisce le performance di ogni algoritmo per quel workload
 - Considera 5 processi che arrivano al tempo 0:

| Process | Burst Time |
|---------|------------|
| P_1 | 10 |
| P_2 | 29 |
| P_3 | 3 |
| P_4 | 7 |
| P_5 | 12 |

- Quale algoritmo è il migliore in termini di tempi di attesa per FF, SJF e RR?

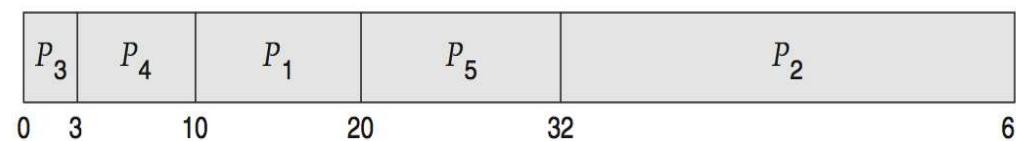
Valutazione Deterministica

- Per ogni algoritmo calcola il minimo tempo di attesa medio
- Semplice e veloce, ma richiede numeri esatti per l'input e si applica solo a questi inputs
 - FCS is 28ms:

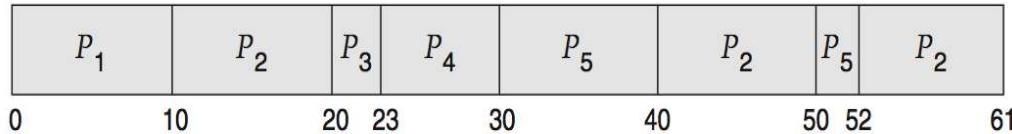


| Process | Burst Time |
|----------------|------------|
| P ₁ | 10 |
| P ₂ | 29 |
| P ₃ | 3 |
| P ₄ | 7 |
| P ₅ | 12 |

- Non-preemptive SJF is 13ms:

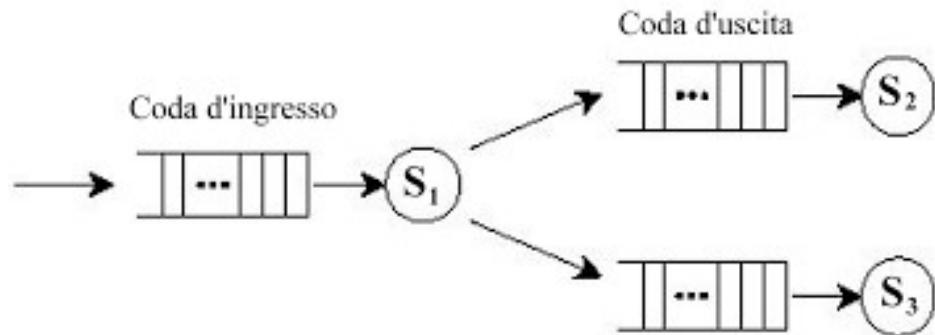


- RR is 23ms:



Modelli di Code

- Descrive gli arrivi dei processi, CPU e I/O bursts probabilisticamente
 - Di solito distribuzioni esponenziali e calcolo di medie
 - Calcola medie di throughput, utilizzo, waiting time, etc
- Sistemi di Computer descritti come reti di server, ognuno con una coda di processi waiting
 - Tempi di arrivo e frequenze di servizio
 - Si calcola utilizzo, media lunghezza delle code, medi tempi di attesa, etc.
- Astrazione ed assunzioni ne limitano l'utilizzo



Formula di Little

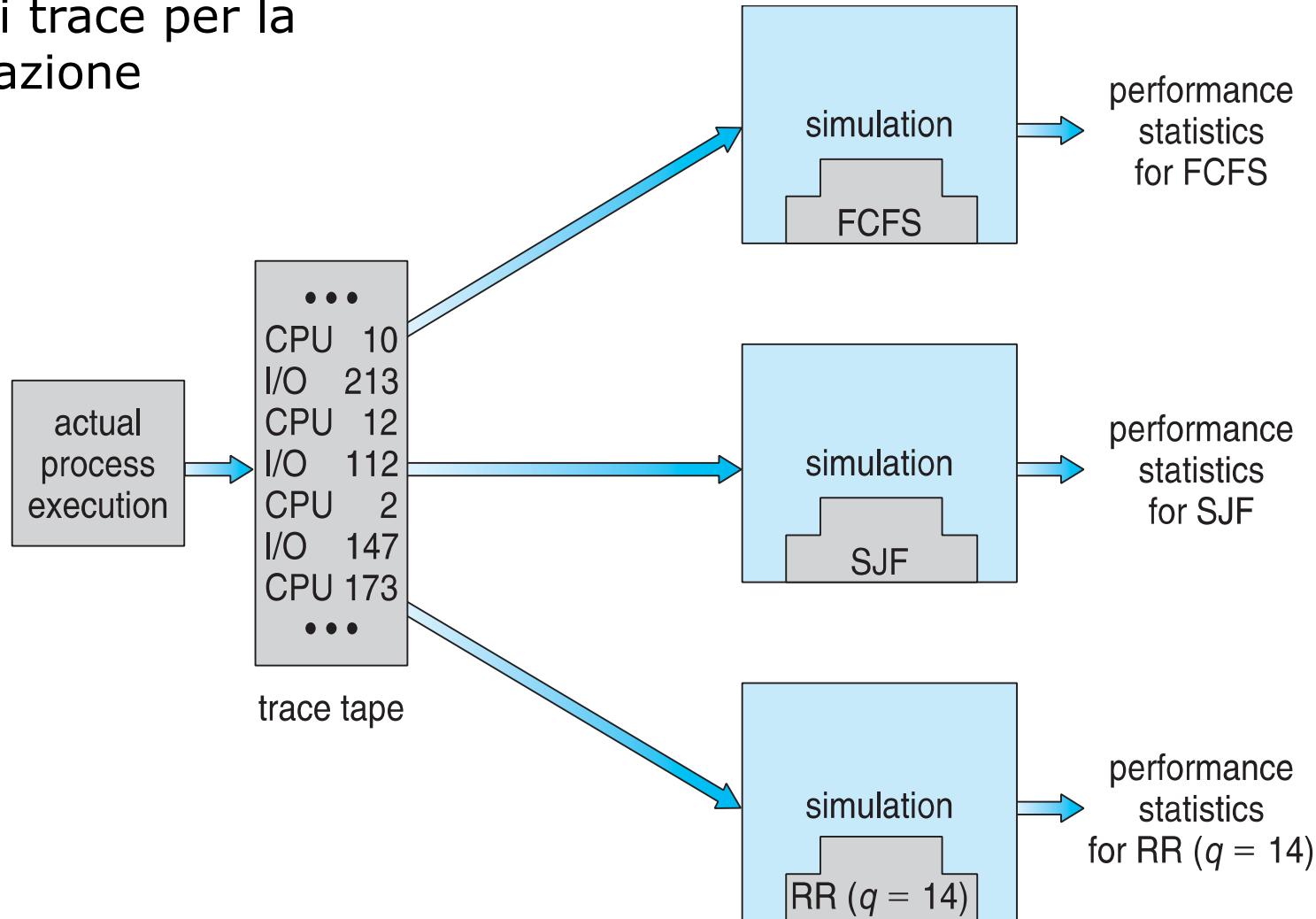
- n = media lunghezza coda
 - W = medio tempo di attesa in coda
 - λ = media frequenza di arrivi in coda
-
- Little's law – in stato stazionario, processi che lasciano la coda devono essere pari a quelli che arrivano, quindi:
$$n = \lambda \times W$$
 - Valido per ogni algoritmo di scheduling e distribuzione degli arrivi
 - Esempio: se in media 7 processi arrivano per secondo, e normalmente 14 processi sono in coda, allora il wait time medio per processo = 2 secondi

Simulazioni

- Modelli di code limitati, **simulazioni** più accurate
 - Modelli di computer system
 - Clock variable
 - Statistiche che indicano le performance
 - Dati per la simulazione ottenuti da
 - ▶ Generatori random
 - Distribuzioni definite matematicamente o empiricamente
 - Se empiricamente eventi reali in sistemi reali

Valuazione di Scheduler con Simulazione

Uso di trace per la simulazione



Implementazione

- Anche le simulazione hanno accuratezza limitata
- Si può direttamente implementare un nuovo scheduler e testarlo in sistemi reali
 - Alto costo, alto rischio
- Scheduler flessibili possono essere modificati per-sito o per-sistema
- API per modificare le priorità sono a disposizione

Obiettivi

- Presentare il concetto di sincronizzazione di processi
- Introdurre il problema della sezione critica
- Soluzioni software e hardware al problema della sezione critica
- Problemi classici di sincronizzazione
- Strumenti per risolvere il problema della sincronizzazione

Background

- Processi possono essere eseguiti concurrentemente
 - Interrotti in ogni momento con esecuzione incomplete
- Accessi concorrenti a dati condivisi possono portare ad inconsistenze sui dati
- La consistenza richiede meccanismi per assicurare l'esecuzione ordinata di processi cooperanti

Background

- Illustrazione del problema:
 - produttore consumatore con memoria limitata
 - Soluzione vista permetteva occupazione BUFFER_SIZE - 1

Bounded-Buffer – Produttore-Consumatore

- Buffer condiviso tra processi in memoria condivisa

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Buffer circolare con indici `in` (prossima free) e `out` (prima posizione full)
- Soluzione corretta, ma limite su `BUFFER_SIZE-1`
 - Buffer pieno $((in + 1) \% BUFFER_SIZE) == out$
 - Buffer vuoto `in == out`

Bounded-Buffer – Produttore

```
item next_produced;  
while (true) {  
    /* produce an item in next_produced */  
    while (((in + 1) % BUFFER_SIZE) == out)  
        ; /* do nothing */  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
}
```

Bounded Buffer – Consumatore

```
item next_consumed;  
while (true) {  
    while (in == out)  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
  
    /* consume the item in next_consumed */  
}
```

Produttore-Consumatore

- Illustrazione del problema:
 - produttore consumatore con memoria limitata
 - Soluzione vista permetteva occupazione BUFFER_SIZE - 1
 - Si può introdurre un intero **counter** che conta i buffer pieni
 - Inizialmente **counter** = 0.
 - Poi incrementato dal produttore dopo che produce un elemento nel buffer e decrementato del consumatore dopo che consuma un elemento dal buffer

Produttore

```
while (true) {  
    /* produce an item in next_produced */  
  
    while (counter == BUFFER_SIZE) ;  
        /* do nothing */  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

Consumatore

```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
    /* consume the item in next_consumed */  
}
```

Corsa Critica

- Operazioni non atomiche
- Se i due processi eseguono counter++ e counter-- insieme
- **counter++** può essere implementato come

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **counter--** può essere implementato come

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Interfogliate le due esecuzioni, inizialmente “count = 5”:

| | |
|---|-----------------|
| S0: producer register1 = counter | {register1 = 5} |
| S1: producer register1 = register1 + 1 | {register1 = 6} |
| S2: consumer register2 = counter | {register2 = 5} |
| S3: consumer register2 = register2 - 1 | {register2 = 4} |
| S4: producer counter = register1 | {counter = 6 } |
| S5: consumer counter = register2 | {counter = 4} |

Corsa Critica

- Il problema della corsa critica è pervasivo
- In particolare in sistemi multicore dove multithreading è molto enfatizzato
- Occorrono meccanismi di sincronizzazione e coordinazione dei processi

Problema della Sezione Critica

- Considera un sistema di n processi $\{p_0, p_1, \dots p_{n-1}\}$
- Ogni processo ha un segmento di **sezione critica** nel codice
 - Processi potrebbero cambiare variabili comuni, aggiornare tabelle condivise, scrivere file, etc.
 - Quando un processo entra in sezione critica gli altri processi non dovrebbero andare nella loro sezione critica
- **Problema della sezione critica** richiede un protocollo di interazione per risolverlo
 - Ogni processo deve chiedere il permesso per entrare nella sezione critica.
 - Il codice che implemente la richiesta è la **entry section**
 - Dopo la sezione ci sarà una **exit section**
 - Il codice rimanente che segue è la **remainder section**

Critical Section

- Struttura generale di un processo con sezione critica P_i

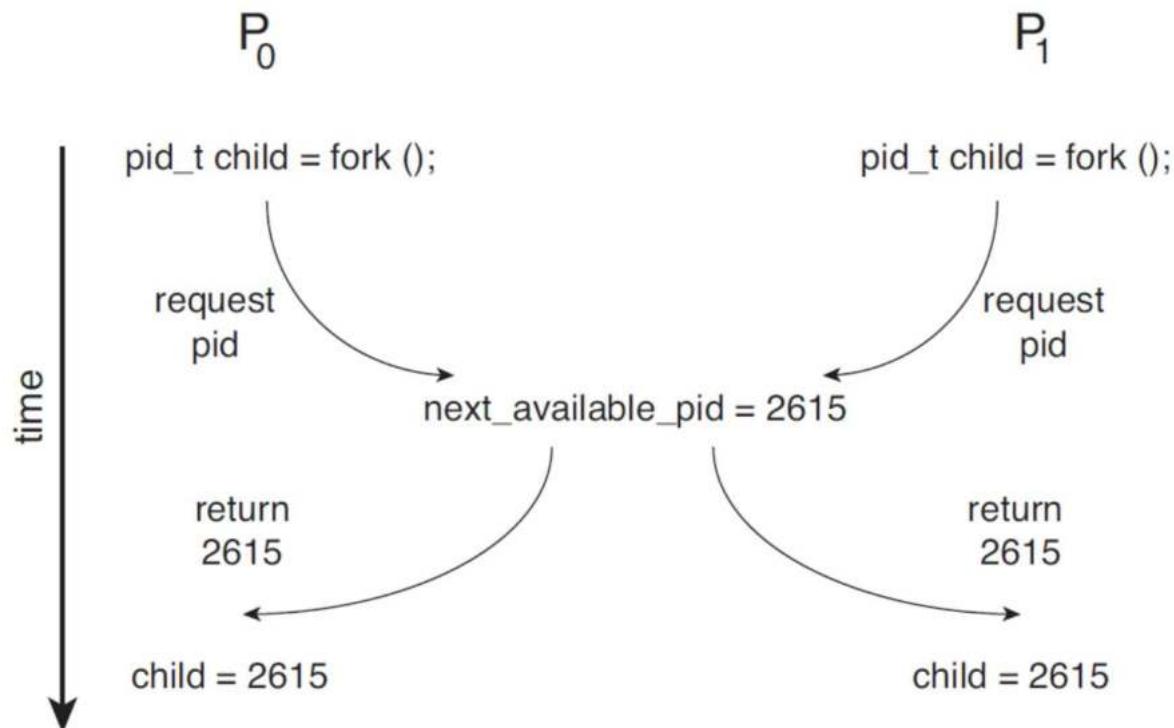
```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

Problema della Sezione Critica

1. **Mutua Esclusione** - se il processo P_i esegue la sua sezione critica nessun altro processo può eseguire la sua sezione critica
2. **Progresso** - Se nessun processo è in esecuzione nella sua sezione critica ed esistono alcuni processi che desiderano entrare nella loro sezione critica, solo i processi non in remainder section decideranno la selezione dei processi che entreranno successivamente nella sezione critica, che non può essere posticipata indefinitamente
3. **Bounded Waiting** - Deve esistere un limite al numero di volte in cui altri processi possono entrare nelle loro sezioni critiche dopo che un processo ha fatto una richiesta di entrare nella sua sezione critica prima che tale richiesta sia concessa
 - Assunto che ogni processo esegue a velocità nonzero
 - Senza assunzioni sulla **velocità relativa** degli n processi

Corse Critiche nel Kernel

- I processi P_0 e P_1 creano processi figlio utilizzando `fork()`
- Corse critiche su tabella di file aperti nel sistema
- Corse critiche sulla variabile kernel che rappresenta il prossimo identificatore di processo disponibile (pid), `next_available_pid`
 - Senza mutua esclusione, lo stesso pid potrebbe essere assegnato a due processi diversi



Gestione della Sezione Critica

Due approcci

- kernel preemptive o non-preemptive
- **Preemptive** – permette la prelazione del processo quando è in kernel mode. Critici con SMP, permettono time sharing.
- **Non-preemptive** – esegue finché in kernel mode, si blocca, o volontariamente rilasciano la CPU
 - ▶ Non si consente l'interruzione forzata di processi attivi in modalità di sistema
 - ▶ WindowsXP/2000
 - ▶ Essenzialmente senza race condition in kernel mode

Soluzione di Peterson

- Buona descrizione algoritmica del problema
- Fornisce soluzione sezione critica per due processi
- Si assume che **load** e **store** siano istruzioni atomiche in linguaggio macchina (non interrompibili)
- Due processi condividono due variabili:
 - `int turn;`
 - `Boolean flag[2]`
- La variable **turn** indica il processo di turno per entrare nella sezione critica
- Il **flag** array è usato per indicare se un processo è pronto per entrare in critical section.
 - `flag[i] = true` significa che il processo P_i è pronto

Algoritmo per il Processo P_i

```
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
        critical section  
    flag[i] = false;  
        remainder section  
} while (true);
```

P_0 : flag[0]=true
 P_1 : flag[1]=true
 P_1 : turn=0
 P_1 : while(flag[0] && turn=0);
 P_0 : turn=1
 P_0 : while(flag[1] && turn=1);
...

Soluzione — Mutua esclusione garantita dal valore di **turn**; P_i entra nella sezione critica (progresso) al massimo dopo un ingresso da parte di P_j (attesa limitata): **alternanza stretta**

Soluzione di Peterson

- Si può provare che i requisiti sono soddisfatti:

1. Mutua esclusione

P_i entra in sezione critica solo se:

`flag[j] = false oppure turn = i`

2. Requisito di progresso è soddisfatto
3. Requisito bounded-waiting è soddisfatto

Soluzione di Peterson

- In sistemi multithreaded ci sono problemi
- Esempio, due thread condividono le variabili:

```
boolean flag = false;  
int x = 0;
```

- Il primo thread esegue

```
while (!flag)  
    ;  
print x;
```

- Il secondo esegue

```
x = 100;  
flag = true;
```

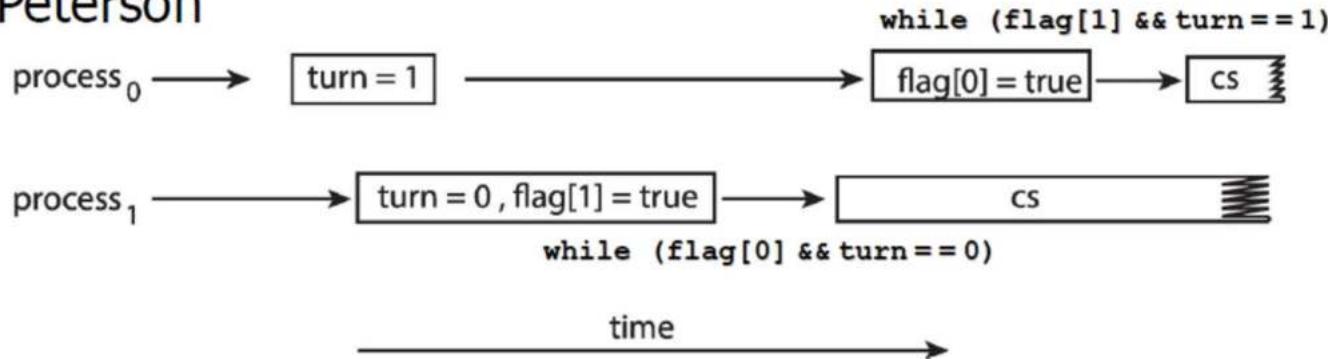
- Ci si aspetta che il primo thread stampi 100, ma non sono garantite le dipendenze tra le variabili x e flag,
- Il thread 2 potrebbe aver cambiato l'ordine delle assegnazioni e stampare 0

Soluzione di Peterson

- In sistemi multithreaded si potrebbe avere accesso concorrente alla sezione

```
while(1) {
    turn = j;
    flag[i] = true;
    while (flag[j] && turn == j);
        sezione critica
    flag[i] = false;
        sezione non critica
}
```

Effetto dello scambio di istruzioni nella soluzione di Peterson



Hardware per Sincronizzazione

- Molti sistemi forniscono supporto hardware per implementare il codice della sezione critica
- Tutte le soluzioni che seguono si basano sul **locking**
 - Proteggere le sezioni critiche con i lock
- Uniprocessori – possono disabilitare gli interrupt
 - Il codice corrente eseguirebbe senza prelazione
 - Troppo inefficiente su sistemi multiprocessori
 - ▶ OS che lo implementano non scalabile
- Macchine moderni forniscono hardware per gestire
 - ▶ **Barriere di memoria**
 - ▶ **Istruzioni hardware**
 - ▶ **Varibili atomiche**
- Usati direttamente o utilizzati per costruire meccanismi di sincronizzazione

Soluzioni basate su Locks

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (TRUE);
```

Barriere di Memoria

- Un **modello di memoria** stabilisce quello che la memoria garantisce
- Modelli di due tipi:
 - Fortemente ordinate
 - ▶ modifica di un processore immediatamente visible per gli altri
 - Debolmente ordinate
 - ▶ modifica non vista da tutti gli altri
- Si possono forzare cambiamenti in memoria propagati ad altri processori (barriera di memoria o recinzione di memoria)
 - Meccanismi di basso livello utilizzati da sviluppatori del kernel
 - Esempio
 - ▶ Thread 1 `while (!flag)
 memory_barrier();
 print x;`
 - ▶ Thread 2 `x = 100;
 memory_barrier();
 flag = true;`

Istruzioni atomiche

- Macchine moderne forniscono hardware per istruzioni atomiche
 - **Atomic** = non-interrompibili
 - Eseguibili sequenzialmente
- Consideriamo due tipi di istruzioni
 - Test memory (word) and set (value)
 - Swap contents of two memory words

test_and_set

Definizione:

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

1. Eseguite atomicamente
2. Restituisce il valore originale (parametro)
3. Setta il nuovo valore del parametro passato a “TRUE”.

Soluzione usando test_and_set()

- Variabile condivisa boolean lock initializzata a FALSE
- Soluzione:

```
do {  
    while (test_and_set(&lock))  
        ; /* do nothing */  
        /* critical section */  
    lock = false;  
    /* remainder section */  
} while (true);
```

Istruzioni atomiche

- Macchine moderne forniscono hardware per istruzioni atomiche
 - **Atomiche** = non-interrompibili
 - Eseguibili sequenzialmente
- Consideriamo due tipi di istruzioni
 - Test memory (word) and set (value)
 - Swap contents of two memory words

test_and_set

Definizione:

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

1. Eseguite atomicamente
2. Restituisce il valore originale (parametro)
3. Setta il nuovo valore del parametro passato a “TRUE”.

Soluzione usando test_and_set()

- Variabile condivisa boolean lock inizializzata a FALSE
- Soluzione:

```
do {  
    while (test_and_set(&lock))  
        ; /* do nothing */  
        /* critical section */  
    lock = false;  
    /* remainder section */  
} while (true);
```

compare_and_swap

Definizione:

```
int compare_and_swap(int *value, int expected, int new_value) {  
    int temp = *value;  
  
    if (*value == expected)  
        *value = new_value;  
    return temp;  
}
```

1. Eseguite atomicamente
2. Restituisce il valore originale del parametro in “value”
3. Setta la variabile “value” al valore di “new_value” ma solo se “value” ==“expected”. Cioè lo swap avviene solo con questa condizione.

Soluzione con compare_and_swap

- Intero condiviso “lock” initializzato a 0;
- Soluzione:

```
do {  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
    /* critical section */  
    lock = 0;  
    /* remainder section */  
} while (true);
```

Non soddisfa attesa limitata
(bounded-waiting)

Bounded-waiting Mutual Exclusion con CAS

```
while (true) {
    waiting[i] = true;
    key = 1;
    while (waiting[i] && key == 1)
        key = compare_and_swap(&lock, 0, 1);
    waiting[i] = false;

    /* critical section */

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock = 0;
    else
        waiting[j] = false;

    /* remainder section */
}
```

Bounded-waiting Mutual Exclusion con test_and_set

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section */
} while (true);
```

Mutex Locks

- Le soluzioni precedenti sono complicate e generalmente inaccessibili ai programmatori di applicazioni
- I progettisti di SO forniscono strumenti software per risolvere i problemi della sezione critica
- Il più semplice è il mutex lock (**mutual exclusion**)
- Protegge una sezione critica prendendo **acquire()** un lock e poi rilasciandolo **release()**
 - Variabile booleana indica se il lock è disponibile o no
- Chiamate **acquire()** e **release()** atomiche
 - Solitamente implementate con istruzioni hardware atomiche
- ... ma richiede **busy waiting**
 - Il lock bloccante è chiamato **spinlock** perché il processo in attesa gira
 - Vantaggio: non fa context-switch, quindi per brevi attese è ok
 - Vantaggio: su multicore durante uno spin l'esecuzione continua

acquire() e release()

- `acquire() {`
 `while (!available)`
 `; /* busy wait */`
 `available = false;;`
 `}`
- `release() {`
 `available = true;`
 `}`
- `do {`
 `acquire lock`
 `critical section`
 `release lock`
 `remainder section`
`}` `while (true);`

Semafori

- Strumenti di sincronizzazione più sofisticati (dei mutex locks) per sincronizzare processi.
- Semaforo **S** – variable intera
- Modificata con due operazioni indivisibili (atomiche)
 - **wait()** e **signal()**
 - ▶ Originariamente detti **P()** e **V()** da Dijkstra (Proberen: to test, Verhogen: to increment)
- Definizione di **wait()**

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

- Definizione di **signal()**

```
signal(S) {  
    S++;  
}
```

Operazioni Atomiche:

- Nessun altro processo può modificare il valore mentre sono in esecuzione
- Nel caso di wait sia test che decremento senza interruzioni

Semafori Uso

- Distinzione tra due tipi di semafori:
- **Semaforo contatore** – valore intero varia su un dominio non ristretto
- **Semaforo binario** – valore intero varia tra 0 e 1
 - Come il **mutex lock**
- Controllo accesso a numero di risorse pari al valore inizializzato
- Può risolvere diversi problemi di sincronizzazione
- Esempio – vincoli di scheduling
 - Considera P_1 e P_2 che richiedono S_1 prima di S_2

Crea un semaforo “**synch**” inizializzato a 0

P1 :

| | | |
|-----------------------|-------------|-------------|
| $S_1;$ | Thread join | Thread exit |
| signal(synch); | | |

P2 :

| | | |
|---------------------|---------|-----------|
| wait(synch); | wait(S) | signal(S) |
| $S_2;$ | | |

- Può implementare un semaforo contatore S con un semaforo binario

Produttori Consumatori

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

Due semafori per vincoli di scheduling
tra produttore e consumatore + un lock

```
while (true) {
    . .
    /* produce an item in next_produced */
    . .
    wait(empty);
    wait(mutex);

    . .
    /* add next_produced to the buffer */
    . .
    signal(mutex);
    signal(full);
}

while (true) {
    wait(full);
    wait(mutex);

    . .
    /* remove an item from buffer to next_consumed */
    . .
    signal(mutex);
    signal(empty);

    . .
    /* consume the item in next_consumed */
    . .
}
```

Problemi con i Semafori

□ Produttori-Consumatori

```
#define N 100                                /* number of slots in the buffer */
typedef int semaphore;                      /* semaphores are a special kind of int */
semaphore mutex = 1;                        /* controls access to critical region */
semaphore empty = N;                         /* counts empty buffer slots */
semaphore full = 0;                          /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();                /* TRUE is the constant 1 */
        down(&empty);                       /* generate something to put in buffer */
        down(&mutex);                      /* decrement empty count */
        insert_item(item);                  /* enter critical region */
        up(&mutex);                        /* put new item in buffer */
        up(&full);                         /* leave critical region */
        /* increment count of full slots */
    }
}
```

Problemi con i Semafori

□ Produttori-Consumatori

```
void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

/* infinite loop */
/* decrement full count */
/* enter critical region */
/* take item from buffer */
/* leave critical region */
/* increment count of empty slots */
/* do something with the item */

Implementazione Semaforo

- Garantire che due processi non eseguano `wait()` e `signal()` sullo stesso semaforo nello stesso tempo

- Problema della sezione critica dove il codice di `wait` e `signal` sono in sezione critica

- Definizioni precedenti con busy waiting

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

- Come evitare il busy waiting?

Implementazione senza Busy waiting

- Ogni semaforo ha una coda di attesa
- Ogni entry in coda di attesa ha due dati:
 - valore (di tipo intero)
 - puntatore al prossimo record in lista
- ```
typedef struct{
 int value;
 struct process *list;
} semaphore;
```
- Due operazioni:
  - **block** – mette il processo che chiede l'operazione nella appropriata coda di attesa
  - **wakeup** – toglie uno dei processi in coda di attesa lo mette nella coda ready

# Implementazione senza Busy waiting

---

```
wait(semaphore *S) {
 S->value--;
 if (S->value < 0) {
 add this process to S->list;
 block();
 }
}

signal(semaphore *S) {
 S->value++;
 if (S->value <= 0) {
 remove a process P from S->list;
 wakeup(P);
 }
}
```

# Implementazione Semaforo

---

- In questo caso il valore diverge dalla definizione classica perché può essere negativo
  - Invertito ordine di decremento e attesa (definizione classica inverso)
  - Se negativo indica il numero di processi in attesa da risvegliare
- La lista dei processi in attesa si può ottenere con puntatori ai Process Control Block (PCB)
- Rimane il problema della sezione critica per il codice di **wait** e **signal**
  - Per singoli processori si può inibire l'interrupt
  - Per multicore molto inefficiente, meglio usare `compare_and_swap`
  - ... ma non eliminato il **busy waiting** spostato dalla entry alla critica section
    - ▶ Dove il codice è breve quindi poco busy waiting se la sezione critica è raramente occupata
    - ▶ Caso molto differente per le applicazioni che potrebbero spendere molto tempo in sezioni critiche

# Deadlock e Starvation

---

- **Deadlock** – due o più processi sono in attesa per un evento che può essere causato solo da uno dei processi in attesa
- Siano  $S$  e  $Q$  due semafori initializzati ad 1

|                         |                         |
|-------------------------|-------------------------|
| $P_0$                   | $P_1$                   |
| <code>wait(S);</code>   | <code>wait(Q);</code>   |
| <code>wait(Q);</code>   | <code>wait(S);</code>   |
| ...                     | ...                     |
| <code>signal(S);</code> | <code>signal(Q);</code> |
| <code>signal(Q);</code> | <code>signal(S);</code> |

- **Starvation – blocco indefinito**
  - Un processo potrebbe non essere più rimosso dalla coda del semaforo su cui è sospeso
- **Priority Inversion**
  - problema di scheduling quando un processo a bassa priorità tiene un lock necessario ad un processo a più alta priorità
  - Risolto con un protocollo di **priority-inheritance**

# Problemi con i Semafori

---

- Uso scorretto delle operazioni dei semafori da parte dei programmatori:
  - Inversione di signal e wait
    - ▶ signal (mutex) .... wait (mutex)
    - ▶ Violazione della mutua esclusione
  - Scambio di signal con wait
    - ▶ wait (mutex) ... wait (mutex)
    - ▶ Blocco permanente
  - Omissione di wait (mutex) o signal (mutex) (o entrambi)
- Sono quindi possibili deadlock e starvation
- Si introducono costrutti di sincronizzazione di alto livello per affrontare questi problemi

```
signal(mutex);
...
critical section
...
wait(mutex);
wait(mutex);
...
critical section
...
wait(mutex);
```

# Problemi con i Semafori

---

- Semantica dei semafori non intuitiva
- Costrutti complessi molto dipendenti dal contest e dalle inizializzazioni
- Facilmente portano ad errori

# Monitor

---

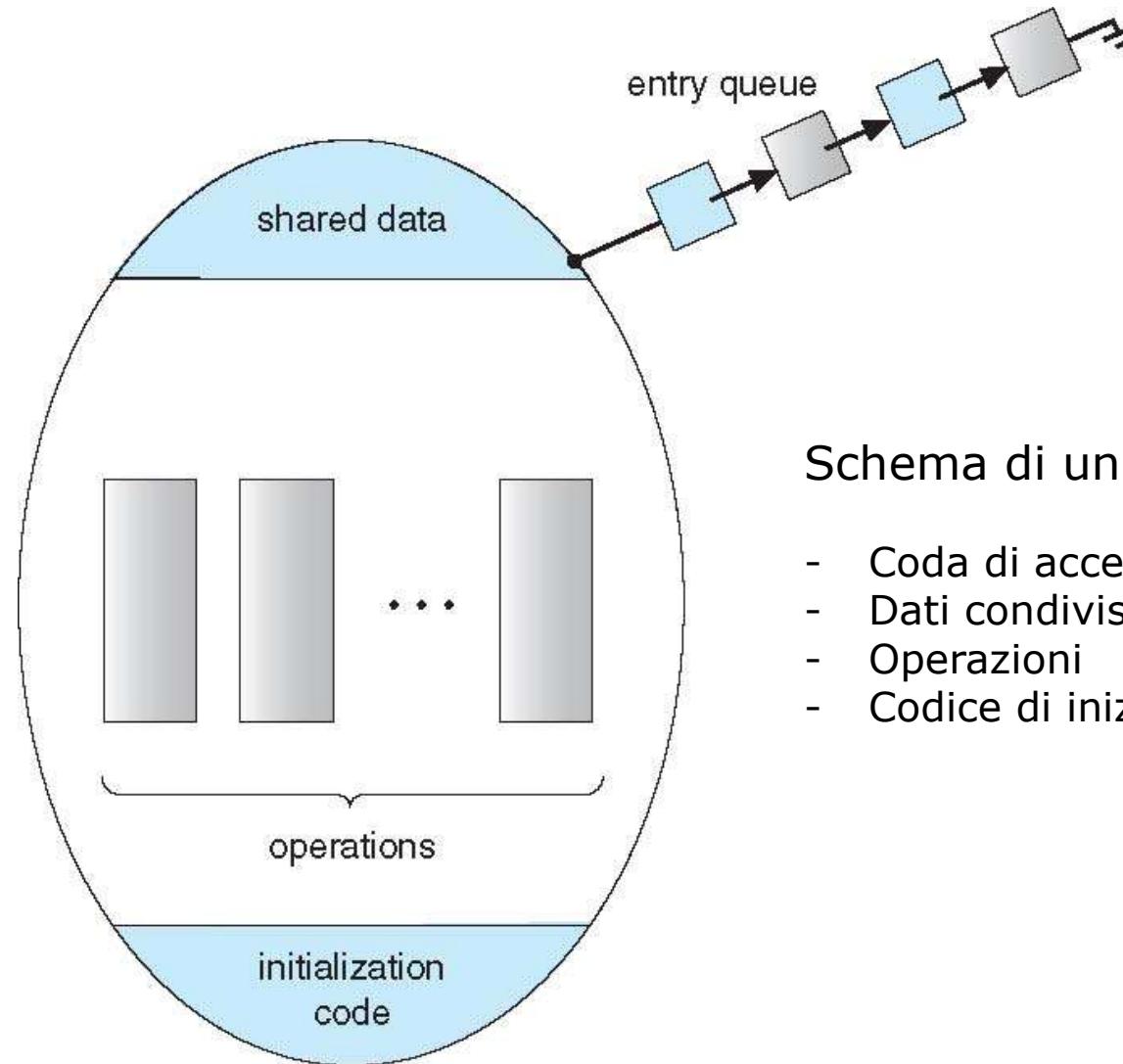
- Astrazione di alto livello che fornisce un meccanismo di sincronizzazione (introdotti nel 1974 da Hoare)
- *Tipo di dato astratto*, variabili condivise accessibili con procedure predefinite
- Solo un processo alla volta può essere attivo nel monitor
- Dati privati
- ... ma non potente abbastanza da modellare alcuni schemi di sincronizzazione
  - Introdotte le variabili di condizione

```
monitor monitor-name
{
 // shared variable declarations
 procedure P1 (...) { }

 procedure Pn (...) { }

 Initialization code (...) { ... }
}
```

# Rappresentazione di un Monitor



Schema di un Monitor:

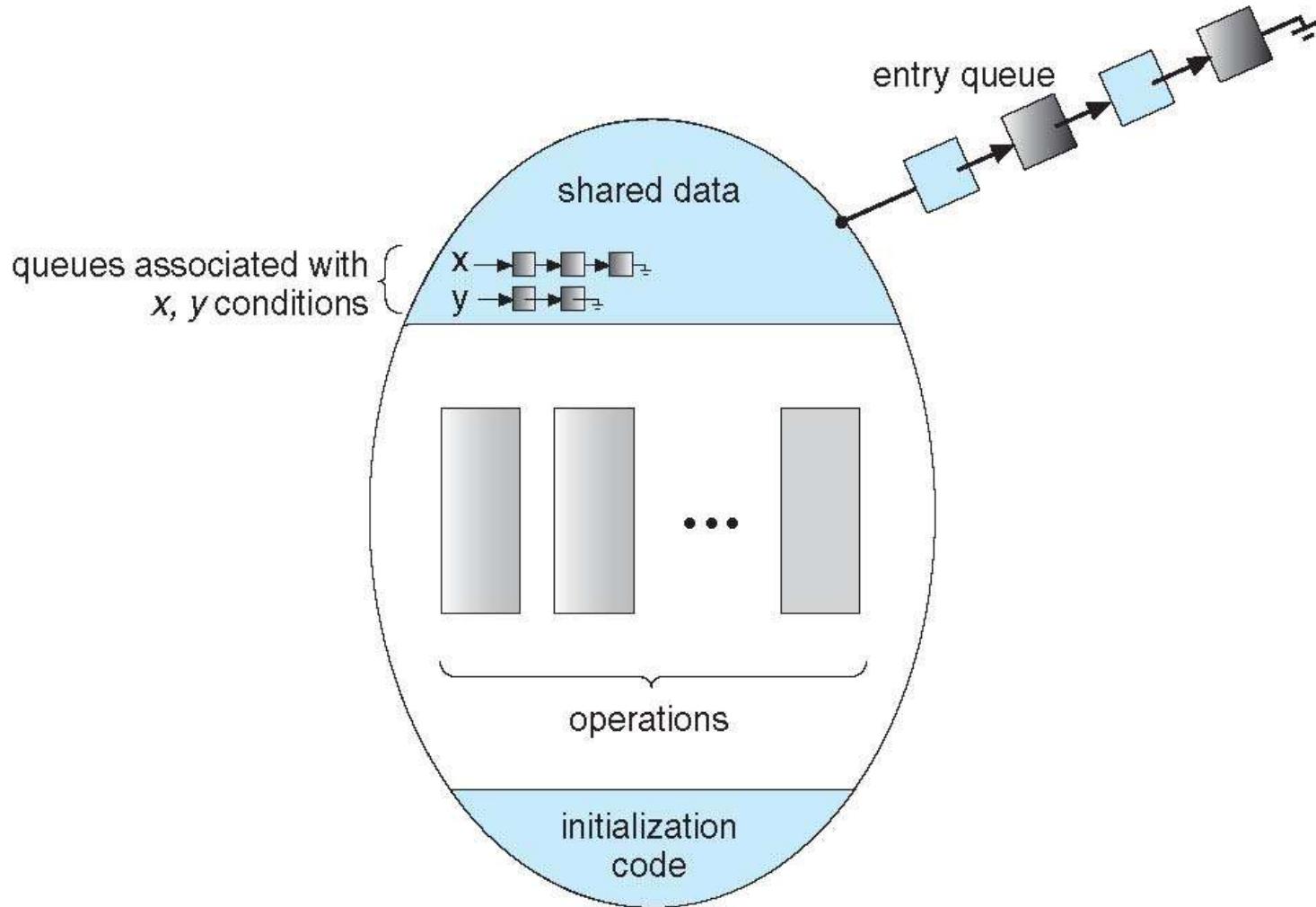
- Coda di accesso
- Dati condivisi
- Operazioni
- Codice di inizializzazione

# Variabili di Condizione

---

- **condition x, y;**
- Due operazioni sono permesse su una variabile di condizione:
  - **x.wait()** – il processo è sospeso finché non avviene **x.signal()**
  - **x.signal()** – riprende uno dei processi (se c'è) che ha invocato **x.wait()**
    - ▶ Se non c'è **x.wait()** sulla variabile, non c'è alcun effetto

# Monitor con Variabili di Condizione



# Monitor

---

- Mutua esclusione richiede un lock
  - Altri costrutti di sincronizzazione sono le variabili di condizione
- 
- Incapsula i dati condivisi da proteggere
  - Prende il mutex
  - Lavora sui dati condivisi
  - Rilascia il lock quando non può procedure
  - Riprende il lock per continuare
  - Rilascia il lock alla fine

# Scelte su Condition Variables

---

- Se il processo P invoca **x.signal()** e il processo Q è sospeso in **x.wait()** che succede?
  - Q e P non possono eseguire in parallelo nel monitor.
    - ▶ Se Q viene ripreso, P deve attendere
- Tra le opzioni abbiamo:
  - **Signal and wait** – P aspetta finché Q o lascia il monitor o aspetta per un'altra condizione
  - **Signal and continue** – Q aspetta finché P lascia il monitor o aspetta per un'altra condizione
  - Entrambe hanno pros e cons – implementatore del linguaggio decide
    - ▶ La seconda lascia al processo in esecuzione, ma la variabile potrebbe cambiare
  - Monitor implementato in Concurrent Pascal fa compromesso
    - ▶ P esegue signal e subito lascia il monitor, quindi Q è ripreso
    - ▶ Implementato in altri linguaggi (Mesa, C#, Java)

# Monitor Implementati con Semafori

---

- Variabili

```
semaphore mutex; // (initially = 1)
semaphore next; // (initially = 0)
int next_count = 0;
```

- Ogni procedura *F* sostituita da Signal and Wait

```
 wait(mutex);
 ...
 body of F;
 ...
 if (next_count > 0)
 signal(next)
 else
 signal(mutex);
```

- La mutua esclusione nel monitor è assicurata

# Monitor Implementation – Condition Variables

---

- Per ogni condition var **x**, abbiamo:

```
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

- La **x.wait** implementata come:

```
x_count++;
if (next_count > 0)
 signal(next);
else
 signal(mutex);
wait(x_sem);
x_count--;
```

# Monitor Implementation

---

- La `x.signal` implementata come:

```
if (x_count > 0) {
 next_count++;
 signal(x_sem);
 wait(next);
 next_count--;
}
```

# Ripresa dei Processi nel Monitor

---

- Se molti processi accodati sulla condizione x, e avviene un x.signal() quali riprendere?
- FCFS spesso non adeguato
- **conditional-wait** come x.wait(c)
  - Dove c è il **priority number**
  - Processo con numero più basso (highest priority) è il prossimo schedulato

# Single Resource allocation

---

- Allocata una singola risorsa tra processi in competizione usando i numeri di priorità che specificano il tempo massimo di utilizzo della risorsa

```
R.acquire(t);
...
access the resource;
...
```

```
R.release;
```

- Con R istanza di tipo **ResourceAllocator**

# Monitor per Singola Risorsa

---

```
monitor ResourceAllocator
{
 boolean busy;
 condition x;
 void acquire(int time) {
 if (busy)
 x.wait(time);
 busy = TRUE;
 }
 void release() {
 busy = FALSE;
 x.signal();
 }
 initialization code() {
 busy = FALSE;
 }
}
```

# Liveness

---

- Un processo che prova ad entrare in sezione critica potrebbe attendere indefinitivamente
- Violato **Progresso e Attesa Limitata**
- **Liveness:**
  - insieme di proprietà che un sistema deve soddisfare per garantire che i processi facciano progressi durante il ciclo di vita della loro esecuzione.
  - Un processo che attende indefinitamente è un esempio di “mancanza di liveness” (liveness failure).
- Esempi di situazioni che possono portare a liveness failure:
  - Ciclo infinito
  - Attesa indefinita
  - Deadlock
  - Inversione di priorità

# Deadlock e Starvation

---

- **Deadlock** – due o più processi sono in attesa per un evento che può essere causato solo da uno dei processi in attesa
- Siano  $S$  e  $Q$  due semafori initializzati ad 1

|                         |                         |
|-------------------------|-------------------------|
| $P_0$                   | $P_1$                   |
| <code>wait(S);</code>   | <code>wait(Q);</code>   |
| <code>wait(Q);</code>   | <code>wait(S);</code>   |
| ...                     | ...                     |
| <code>signal(S);</code> | <code>signal(Q);</code> |
| <code>signal(Q);</code> | <code>signal(S);</code> |

- **Starvation – blocco indefinito**
  - Un processo potrebbe non essere più rimosso dalla coda del semaforo su cui è sospeso
- **Priority Inversion**
  - problema di scheduling quando un processo a bassa priorità tiene un lock necessario ad un processo a più alta priorità
  - Risolto con un protocollo di **priority-inheritance**

# Inversione di Priorità

---

- Processo ad alta priorità che deve accedere a risorse del kernel tenute da un processo a parità più bassa
- Più complicato se il processo è prelazionato da un altro ancora
- Tre processi  $L < M < H$ 
  - H chiede il semaforo S tenuto da L
  - H deve aspettare L
  - ... M diventa runnable e prelaziona L
  - Il processo M influenza l'attesa di un processo H (a più alta priorità) su S tenuta da L
- Priority-inheritance protocol
  - Quando processi accedono ad una risorsa richiesta da un processo ad alta priorità, ereditano la priorità del processo. Poi tornano ai valori originali
  - L avrebbe ereditato la priorità di H bloccando M, rientrato in priorità L al rilascio, la competizione tra H ed M è vinta da H

# Inversione di Priorità

---

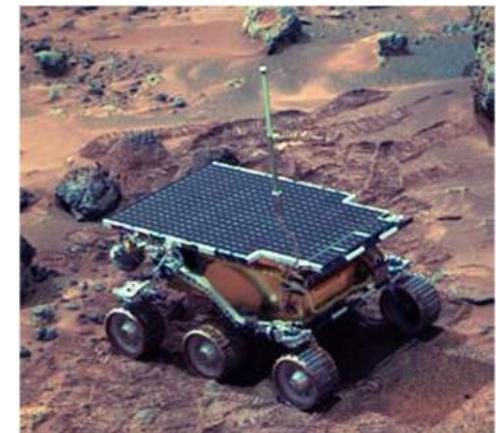
- L'inversione di priorità può avere effetti a catena causando un fallimento sistematico
- Il caso del Mars Pathfinder
  - Prima missione ad aver trasportato un rover, Sojourner, sul pianeta.
  - Lancio: 4 dicembre 1996
  - Arrivo: 4 luglio 1997 (7 mesi dopo)
  - Lander operava come stazione meteorologica
  - Rover usato per analizzare il suolo e le rocce del sito di atterraggio ed effettuare esperimenti sulla superficie
  - [https://space.skyrocket.de/doc\\_sdat/mars\\_pathfinder.htm](https://space.skyrocket.de/doc_sdat/mars_pathfinder.htm)



# Inversione di Priorità

---

- L'inversione di priorità può avere effetti a catena causando un fallimento sistemico
  - Il caso del Mars Pathfinder
    - Dopo i primi test il software di Sojourner fa continui "reset"
    - Inizializzato hw, sw, comunicazione
- 
- La raccolta dati dal bus era affidata a due task:
    - Il gestore delle transizioni del bus (`bc_sched`)
    - Il responsabile di acquisizione dati (`bc_dist`)
  - Una sequenza di operazioni corrette richiedeva che i due task si alternassero in esecuzione, durante ogni ciclo di 8Hz
    - Quando un task iniziava, per prima cosa controllava che l'altro task avesse finito; altrimenti, generava un **reset del sistema**.



# Inversione di Priorità

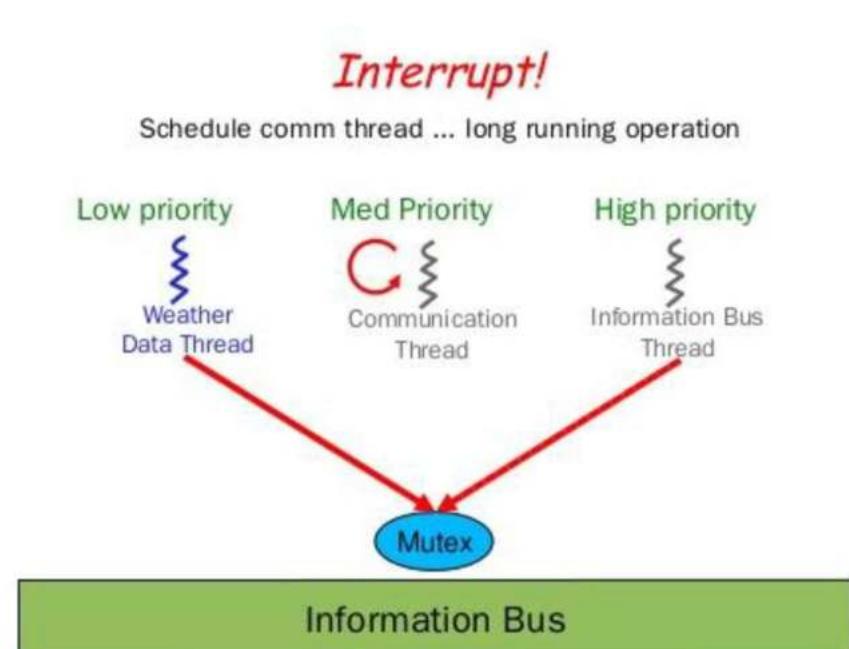
- L'inversione di priorità può avere effetti a catena causando un fallimento sistemico
- Il caso del Mars Pathfinder
  - Dopo i primi test il software di Sojourner fa continui "reset"
  - Inizializzato hw, sw, comunicazione

Il problema riguardava tre task:

- bc\_dist (priorità 3),
- Un qualsiasi task relativo alle operazioni della stazione spaziale (priorità 4), e
- Il task della stazione meteo (priorità 5)

Il task della stazione meteo riceveva i dati da bc\_sched, per fare questo, doveva **prendere in uso esclusivo (lock) una risorsa, cioe` il bus.**

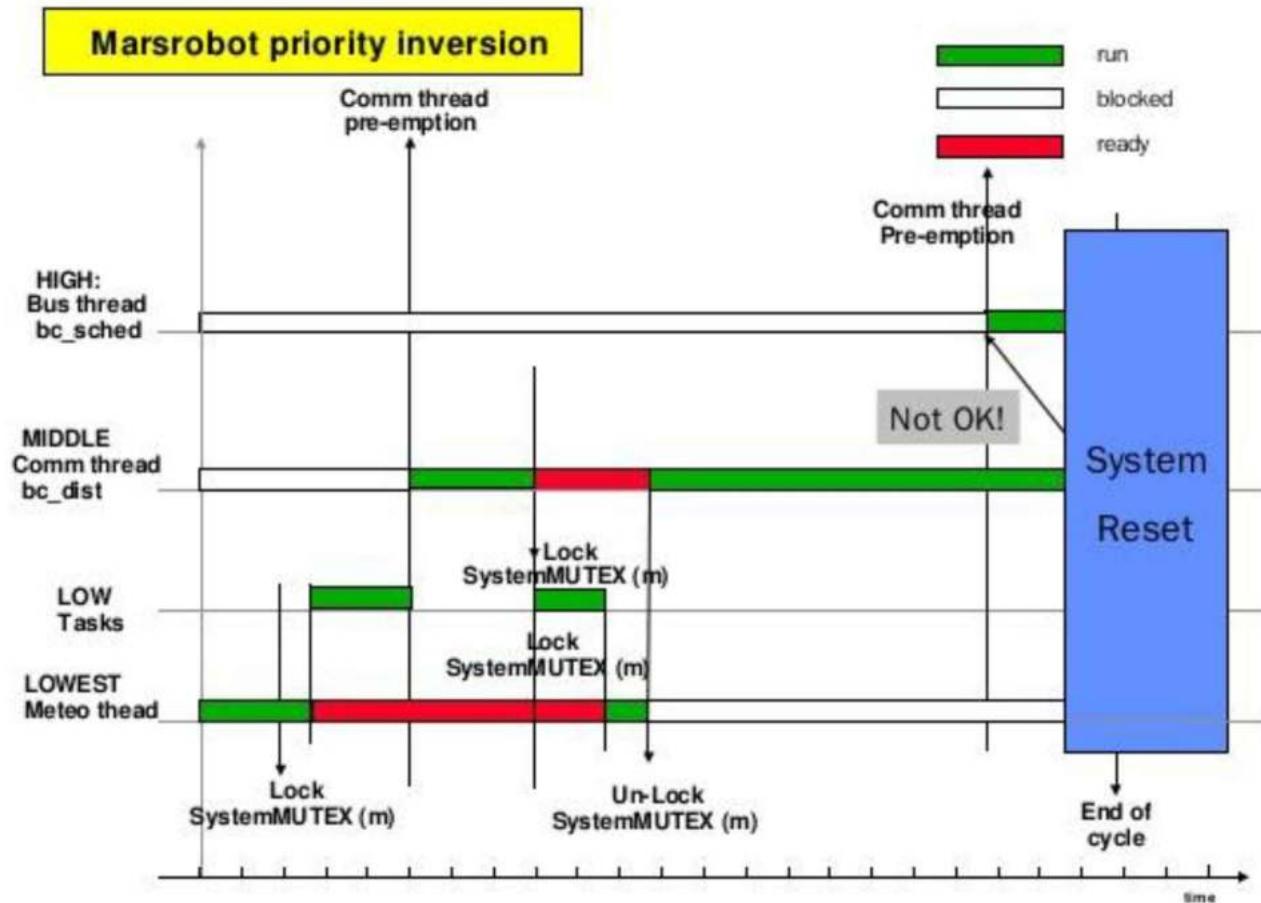
- La sua esecuzione veniva sospesa dal task con priorità 4, prima che rilasciasse il lock,
- Successivamente, bc\_dist non riusciva ad acquisire quella risorsa, e forzava il reset.



# Inversione di Priorità

## Il caso del Mars Pathfinder

- Dopo i primi test il software di Sojourner fa continui “reset”
- Inizializzato hw, sw, comunicazione



<https://www.slideshare.net/gabrielladodero/quando-i-computer-non-funzionano-su-marte>

# Inversione di Priorità

---

- Il caso del Mars Pathfinder
  - Dopo i primi test il software di Sojourner fa continui “reset”
  - Inizializzato hw, sw, comunicazione
  - SO su Sojourner era VxWorks (real-time)
  - Variabile globale per priority inheritance su tutti i semafori
  - La variabile settata su Sojourner e il problema è stato risolto

<https://www.slideshare.net/gabrielladodero/quando-i-computer-non-funzionano-su-marte>

# Livelli di Contesa

---

- Strumenti per il problema della sezione critica e per sincronizzare l'attività dei processi possono essere valutati a seconda del livello di contesa.
- Alcuni strumenti funzionano meglio con un certo livello di contesa
- Strumenti basati su istruzioni CAS per avere lock-free
  - Approccio ottimistico (prima testi poi verifichi collisioni)
  - Pessimistico (prima lock poi test)
- Linee guida per distinguere le prestazioni della sincronizzazione basata su `compare_and_swap()` (CAS) e della sincronizzazione tradizionale (come i lock mutexe i semafori) al variare del livello di contesa:
  - Nessuna contesa. Sebbene entrambe le opzioni siano generalmente veloci, la protezione CAS sarà leggermente più veloce della sincronizzazione tradizionale.
  - Contesa moderata. La protezione CAS sarà più veloce e in alcuni casi molto più veloce rispetto alla sincronizzazione tradizionale.
  - Alta contesa. Con carichi molto elevati, la sincronizzazione tradizionale sarà in definitiva più veloce della sincronizzazione basata su CAS.

# Sistemi I/O

# I/O Systems

---

- Overview
- I/O Hardware
- Applicazioni di interfacce I/O
- Kernel I/O Subsystem
- Transformazione di I/O Requests in operazioni hardware
- STREAMS
- Performance

# Obiettivi

---

- Esplora la struttura di un sottosistema I/O di un SO
- Discutere i principi dell'hardware I/O
- Dettagliare le performance di I/O hardware e software

# Overview

---

- Gestione I/O è uno dei compiti fondamentali di un OS
  - Importante aspetto delle operazioni di un computer
  - Dispositivi I/O variano molto, ma nello stesso tempo interface standard
  - Vari metodi per controllare
  - Gestione della performance
  - Nuovi tipi di dispositivi sono frequenti
- Porte, bus, controllori di dispositivo connettono i vari dispositivi
- **Device driver** encapsulano dettagli dei dispositivi
  - Presentano interfacce per un accesso uniforme ai sottosistemi I/O
  - Ruolo analogo a quello delle chiamate di sistema

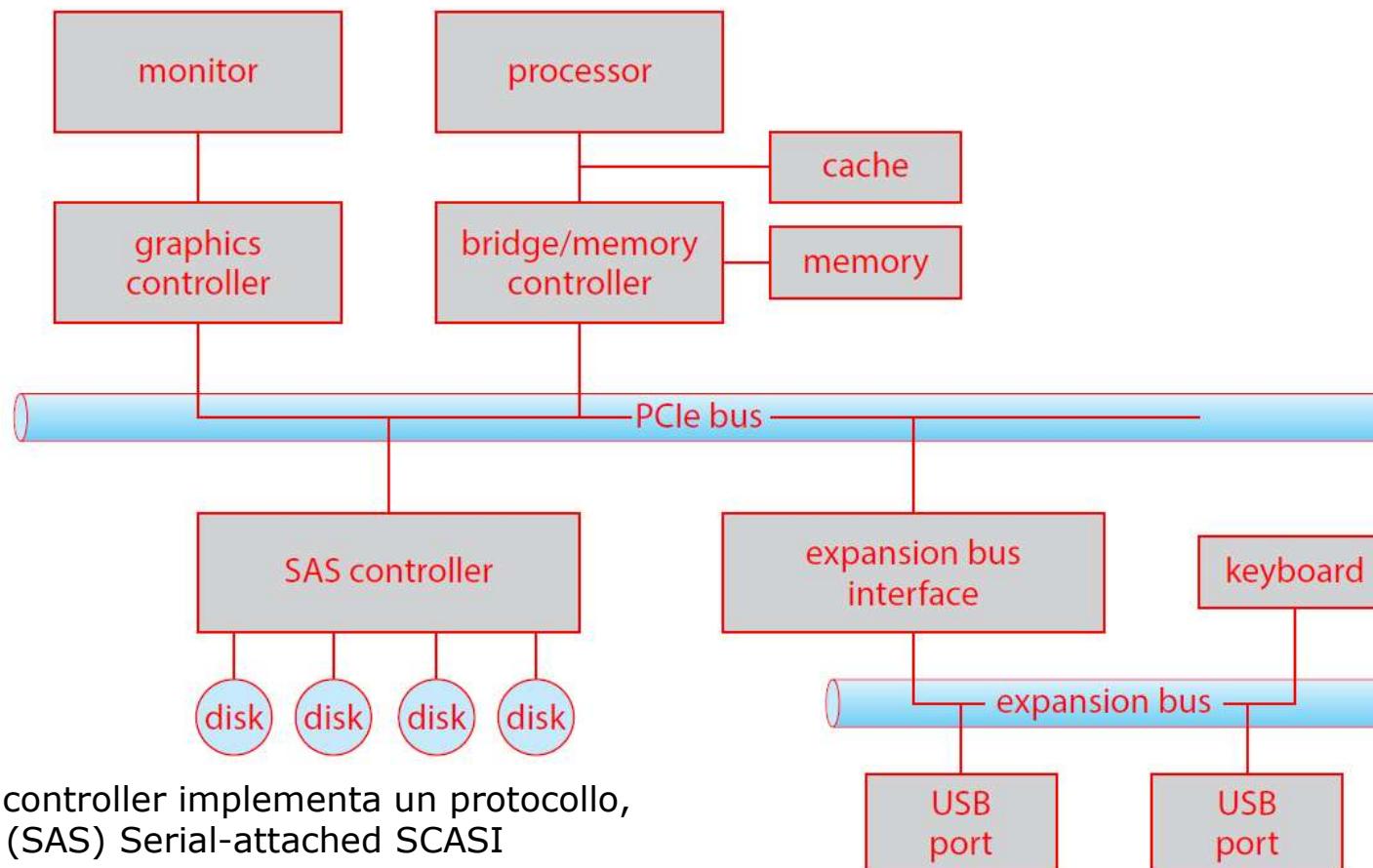
# I/O Hardware

---

- Grande varietà di dispositivi I/O
  - Storage (dischi)
  - Trasmissione (connessione di rete)
  - Interfacce uomo-macchina (schermi, tastiera, audio, mouse)
- Concetti
  - segnali da/a dispositivo I/O a/da interfaccia con computer
  - **Porta** – punto di connessione con il computer (es., porta seriale)
  - **Bus - daisy chain** (A connesso con B connesso con C, etc.)
    - ▶ **PCI** bus tipico in PC e servers (PCI Express (**PCle**)) per dispositivi veloci
    - ▶ **expansion bus** connette i dispositivi “lenti” (tastiera, porte seriali o USB)
  - **Controller (host adapter)** – electronica che gestisce porta, bus, dispositivi
    - ▶ A volte integrate, a volte circuito separato (host adapter)
    - ▶ Contiene processore, microcodice, memoria privata, bus controller, etc

# Una Tipica Struttura dei PC Bus

PCI composta da linee, con  $2 + 2$  connessioni trasportando pacchetti nelle due direzioni  
 $xN$  per indicare N linee ( $N=1,2,4,8, 16, 32, \dots$ )



# I/O Hardware

---

- Il processore invia comandi e dati al controller di un dispositivo I/O?
- Attraverso registri di dati e comandi
  - I controllori hanno registri per i comandi, indirizzi, e dati da scrivere, o leggono i dati dai registri dopo l'esecuzione dei comandi
  - Data-in register, data-out register, status register, control register
  - Registri di 1-4 byte oppure FIFO chip può espandere
- I dispositivi hanno indirizzi, usati per
  - Istruzioni I/O che chiedono il trasferimento di dati ad un indirizzo di una porta I/O
  - **Memory-mapped I/O**
    - ▶ Registri del controllo del dispositivo mappato sull'address space del processore
    - ▶ La CPU richiede operazioni standard di scrittura/lettura su registri mappati in memoria
    - ▶ In particolare per grandi address space (grafica)
      - Porte per operazioni di base
      - Il controller ha regioni memory-mapped per il contenuto dello schermo

# Porte dei dispositivi I/O su PC

---

| I/O address range (hexadecimal) | device                    |
|---------------------------------|---------------------------|
| 000–00F                         | DMA controller            |
| 020–021                         | interrupt controller      |
| 040–043                         | timer                     |
| 200–20F                         | game controller           |
| 2F8–2FF                         | serial port (secondary)   |
| 320–32F                         | hard-disk controller      |
| 378–37F                         | parallel port             |
| 3D0–3DF                         | graphics controller       |
| 3F0–3F7                         | diskette-drive controller |
| 3F8–3FF                         | serial port (primary)     |

# Polling

---

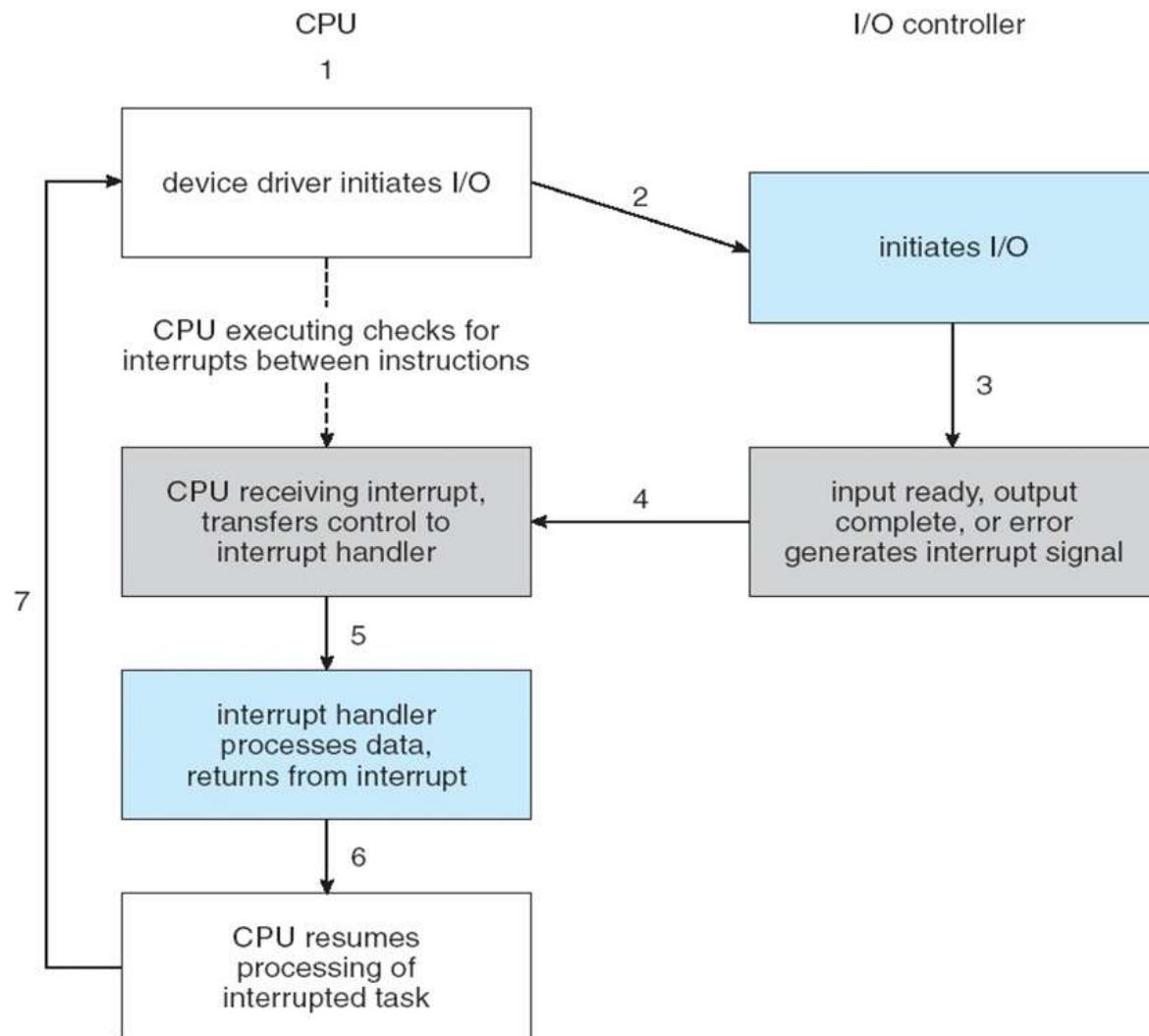
- Si assume un host che interagisce con un controller
- For each byte of I/O
  - 1. Host legge il busy bit dallo status register finché è 0
  - 2. L'host setta read o write bit e se write copia i dati nel data-out register
  - 3. L'host setta il command-ready bit
  - 4. Il controller setta il busy bit, esegue il trasferimento
  - 5. Il controller toglie il busy bit, error bit, command-ready bit quando ha fatto
- Nello step 1 l'host è in un ciclo di **busy-wait** in attesa di I/O dal device
  - Ragionevole se il dispositivo è veloce
  - inefficiente se lento
  - CPU può andare in switch su altri task?
    - ▶ ... ma se perde i cicli i dati possono essere sovrascritti o perduti

# Interrupt

---

- Polling può avvenire in un ciclo di 3 istruzioni
  - Read status, logical-and per estrarre lo status bit, branch se non zero
  - Come si può essere più efficienti se non-zero non frequente?
- CPU hardware ha una **Interrupt-request line**
  - Controller innesca interrupt (raises)
  - Controllato dal processore dopo ogni istruzione
  - Se interrupt (catches), salva lo stato e chiama interrupt handler routine (dispatches)
- **Interrupt handler routine** gestisce gli interrupt
  - Controlla la causa dell'interrupt, serve, clears l'interrupt e ripristina lo stato
  - **Interrupt vector** per eseguire l'interrupt con il gestore corretto

# Ciclo per Interrupt-Driven I/O



# Interrupt

---

- Importante gestione attentativa degli interrupt
- Anche un PC quasi idle gestisce continuamente interrupt
- In figura comando latency macOS su desktop
  - ... in 10 secondi 23000 interrupts

| SCHEDULER          |    | INTERRUPTS |
|--------------------|----|------------|
| total_samples      | 13 | 22998      |
| delays < 10 usecs  | 12 | 16243      |
| delays < 20 usecs  | 1  | 5312       |
| delays < 30 usecs  | 0  | 473        |
| delays < 40 usecs  | 0  | 590        |
| delays < 50 usecs  | 0  | 61         |
| delays < 60 usecs  | 0  | 317        |
| delays < 70 usecs  | 0  | 2          |
| delays < 80 usecs  | 0  | 0          |
| delays < 90 usecs  | 0  | 0          |
| delays < 100 usecs | 0  | 0          |
| total < 100 usecs  | 13 | 22998      |

# Interrupt

---

- Nei sistemi moderni gestione più complessa
  - Posticipare la gestione dell'interrupt durante operazioni critiche
  - Modo efficiente per fare il dispatch dell'handler giusto
  - Mecchanismi di priorità nella gestione
  - Gestione separata di alcune eccezioni diverse da I/O (traps, es. page fault)
- **Interrupt controller hardware** e CPU gestiscono
  - Due linee di interruzione **maskable** e **nonmaskable** (gli I/O mascherabili)
  - Gestione di indirizzi di interrupt per mappare la risposta su **interrupt vector**
  - In partica però più dispositivi che address, quindi interrupt concatenati se più di un device allo stesso interrupt number
    - ▶ Quando arriva un interruzione su quell numero tutti i dispositivi interrogati finsché non si trova quello giusto
  - Livelli di priorità, un interrupt a più alto livello può prelazionarne uno di più basso
  - First level interrupt handler e Second Level interrupt handler per dividera la gestione dell'interrupt
    - ▶ Il primo va context switch, state storage e accodamento, il secondo serve la richiesta

# Intel Pentium Processor Event-Vector Table

Fino a 32 non maskable

| vector number | description                            |
|---------------|----------------------------------------|
| 0             | divide error                           |
| 1             | debug exception                        |
| 2             | null interrupt                         |
| 3             | breakpoint                             |
| 4             | INTO-detected overflow                 |
| 5             | bound range exception                  |
| 6             | invalid opcode                         |
| 7             | device not available                   |
| 8             | double fault                           |
| 9             | coprocessor segment overrun (reserved) |
| 10            | invalid task state segment             |
| 11            | segment not present                    |
| 12            | stack fault                            |
| 13            | general protection                     |
| 14            | page fault                             |
| 15            | (Intel reserved, do not use)           |
| 16            | floating-point error                   |
| 17            | alignment check                        |
| 18            | machine check                          |
| 19–31         | (Intel reserved, do not use)           |
| 32–255        | maskable interrupts                    |

# Interrupt

---

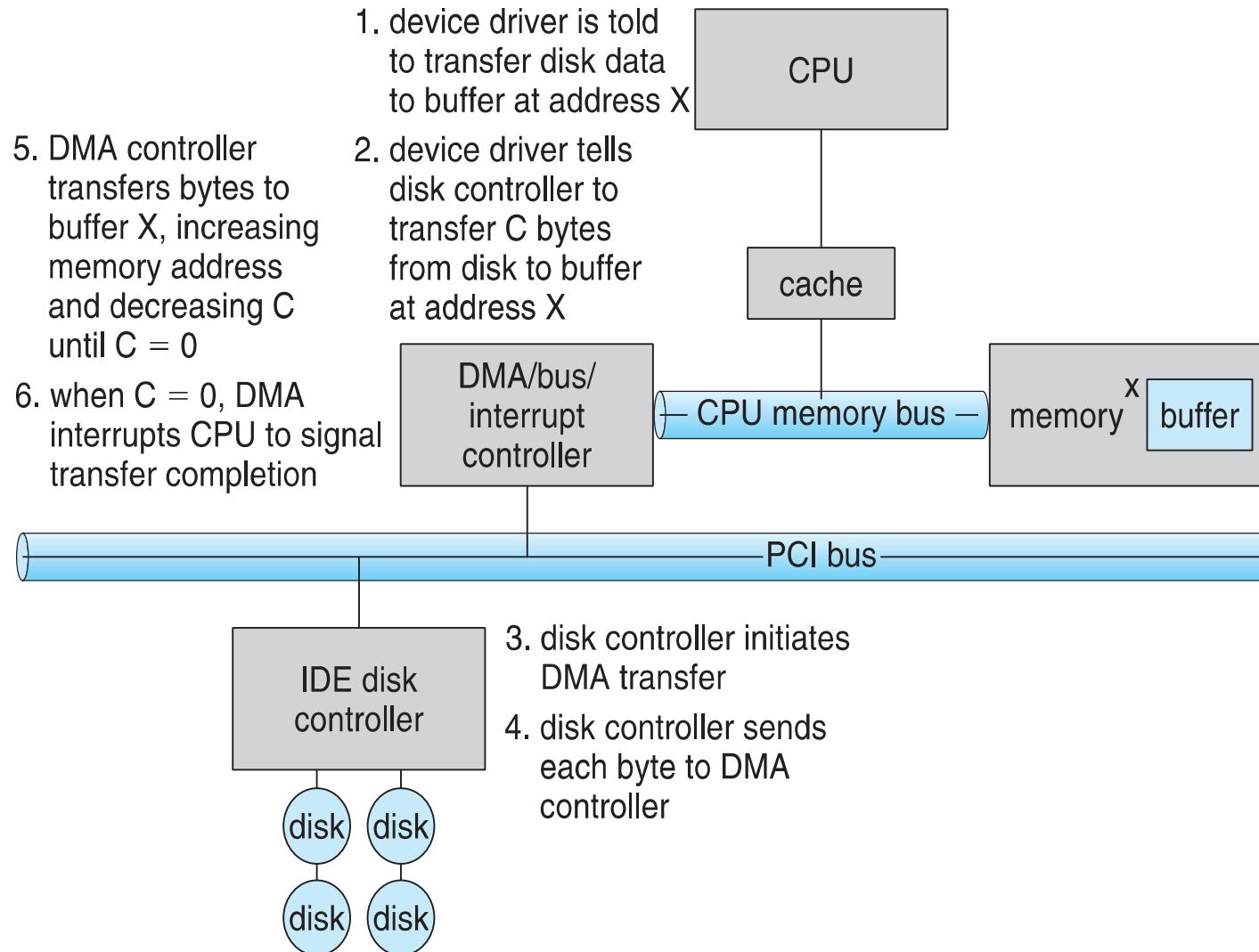
- Interrupt anche usato per gestire le **eccezioni**
  - Terminazione di processo, crash di sistema dovuto a hardware error
- Page fault eseguito per memory access error
  - handler gestisce il processo di caricamento della pagina
- System call eseguite con un **trap** per innescare il kernel ed eseguire la richiesta
  - Gestita come interruzione a bassa priorità
- Sistemi Multi-CPU possono processare interrupt concorrentemente
  - Se SO progettato per gestirli

# Direct Memory Access

---

- Usato per evitare **programmed I/O** (un byte alla volta) per grandi trasferimenti di dati
- Richiede un **DMA** controller
- Bypassa la CPU per trasferire dati direttamente tra dispositivi I/O e memoria
- SO scrive un command block in memoria per DMA
  - Scrive la locazione del command block al controller del DMA
  - Indirizzo sorgente e destinazione
  - Read o write mode
  - Numero di bytes da trasferire
- Bus mastering del DMA controller
  - DMA prende il bus alla CPU per il trasferimento
    - ▶ **Cycle stealing** prende il bus rubando cicli alla CPU mentre non lo utilizza
- Quando ha finito, interrupt per segnalare il completamento

# Sei Passi per Eseguire il Trasferimento DMA



# Direct Memory Access

---

- Handshaking tra device controller e DMA controller
  - DMA request e DMA acknowledge
    - Device controller solleva una **DMA request**
    - DMA controller chiede il bus alla CPU
    - DMA controller prende il bus, mette l'indirizzo richiesto sul memory address e manda il segnale di **DMA acknowledge**
    - Il Controller del dispositivo allora trasferisce i dati e rimuove il DMA request
    - Quando la DMA ha finito il trasferimento manda interrupt alla CPU

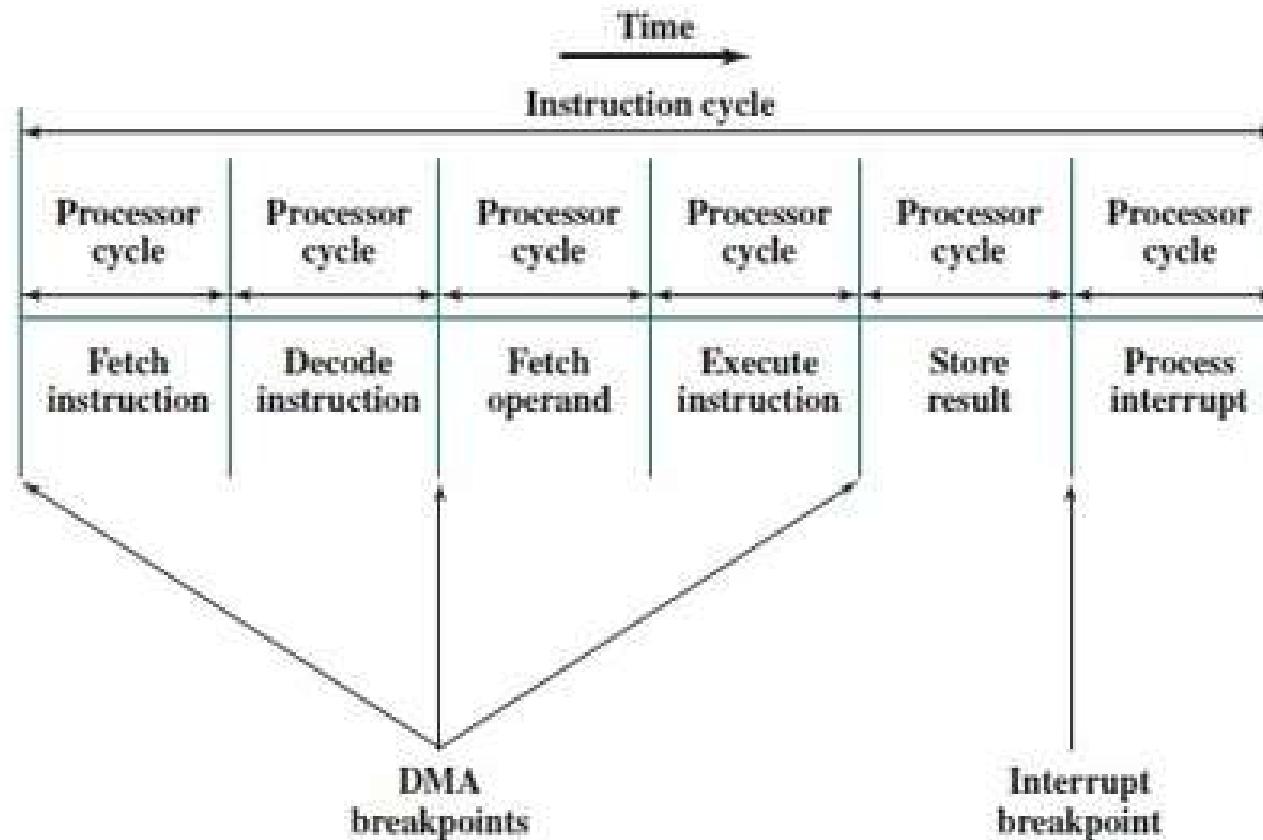
# Direct Memory Access

---

- Handshaking tra device controller e DMA controller
  - DMA request e DMA acknowledge
    - Device controller solleva una **DMA request**
    - DMA controller chiede il bus alla CPU
    - DMA controller prende il bus, mette l'indirizzo richiesto sul memory address e manda il segnale di **DMA acknowledge**
    - Il Controller del dispositivo allora trasferisce i dati e rimuove il DMA request
    - Quando la DMA ha finito il trasferimento manda interrupt alla CPU
- Target address nel Kernel Space (in user space si potrebbe modificare)
  - Però allora poi doppia scrittura per portarlo in User Space
  - Direttamente in user space
- In alcuni casi il DMA accede gli indirizzi fisici, versioni consapevoli dei virtual addresses possono essere anche più efficienti - **DVMA**

# Cycle Stealing

- Bus mastering del DMA controller
  - DMA prende il bus alla CPU per il trasferimento
    - ▶ **Cycle stealing** prende il bus rubando cicli alla CPU mentre non lo utilizza

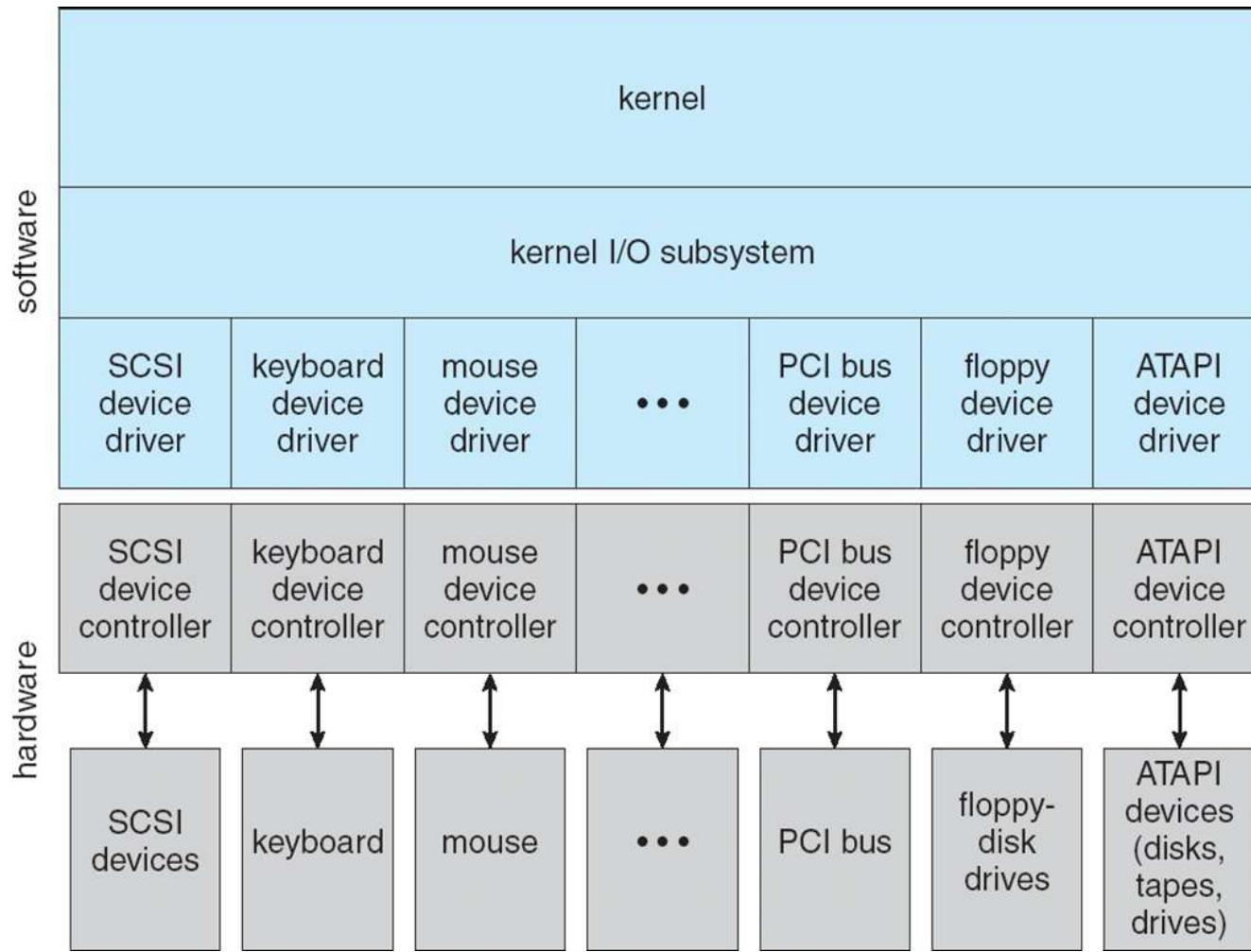


# Interfacce per Applicazioni I/O

---

- Interfaccia uniforme per gestire I/O molto differenti
- Astrazione, encapsulamento e stratificazione
- I/O system call encapsulano comportamenti dei device in classi generiche
- Device-driver layer nascondono le differenze tra I/O controller dal kernel
- Nuovi device driver con protocolli già implementati non necessitano di lavoro extra
- Ogni SO ha i suoi sottosistemi I/O e device driver

# Strutturazione del Kernel per I/O



# Interfacce per Applicazioni I/O

---

- Ogni SO ha i suoi sottosistemi I/O e device driver
- Il venditore li deve mettere a disposizione
- Device driver variano in diverse dimensioni
  - **Character-stream o block**
    - byte a byte o a blocchi come unità di trasferimento
  - **Sequential o random-access**
    - accesso sequenziale ai dati del dispositivo o qualunque
  - **Synchronous o asynchronous** (o entrambi)
    - Trasferimento dati sincronizzato (tempi prevedibili) altrimenti irregolare e non coordinato
  - **Sharable o dedicated**
    - Può essere usato contemporaneamente da più thread o no?
  - **Speed of operation**
    - Velocità di trasferimento da pochi byte a Gb per secondo
  - **read-write, read only, o write only**
    - alcuni dispositivi permettono solo lettura altri scritti solo una volta, etc.

# Caratteristiche dei Dispositivi I/O

| aspect             | variation                                                         | example                               |
|--------------------|-------------------------------------------------------------------|---------------------------------------|
| data-transfer mode | character<br>block                                                | terminal<br>disk                      |
| access method      | sequential<br>random                                              | modem<br>CD-ROM                       |
| transfer schedule  | synchronous<br>asynchronous                                       | tape<br>keyboard                      |
| sharing            | dedicated<br>sharable                                             | tape<br>keyboard                      |
| device speed       | latency<br>seek time<br>transfer rate<br>delay between operations |                                       |
| I/O direction      | read only<br>write only<br>read-write                             | CD-ROM<br>graphics controller<br>disk |

# Caratteristiche dei Dispositivi I/O

---

- Dettagli dei dispositivi gestiti dai device drivers
- Approssimativamente i dispositivi I/O possono essere raggruppati dagli SO in categorie (con chiamate di sistema per l'accesso)
  - Block I/O
  - Character I/O (Stream)
  - Memory-mapped file access
  - Network sockets
  - Poi anche accesso a time-of-the-day, timer, graphics, video, audio, etc.

# Caratteristiche dei Dispositivi I/O

---

- Per manipolazione diretta di un dispositivo con funzionalità definite nei device driver spesso a disposizioni una sys call **escape / back door**
  - In Unix **ioctl()** call (I/O control) manda bit arbitrari a un device control register e dati al device data register
    - ▶ Tre argomenti id del dispositivo, intero che identifica il comando, indirizzo della struttura da gestire
    - ▶ Id in Linux/Unix è dato da due numeri (tipo, istanza)
    - ▶ Esempio, in Linux per vedere i dispositivi SSD

```
% ls -l /dev/sda*
```

```
brw-rw---- 1 root disk 8, 0 Mar 16 09:18 /dev/sda
brw-rw---- 1 root disk 8, 1 Mar 16 09:18 /dev/sda1
brw-rw---- 1 root disk 8, 2 Mar 16 09:18 /dev/sda2
brw-rw---- 1 root disk 8, 3 Mar 16 09:18 /dev/sda3
```
    - ▶ In questo caso 8 è il tipo, 0, 1, 2, 3 sono le istanze

# Device Block e Character

---

- Le interfacce ai block device permettono l'accesso alle unità disco e altri dispositivi con memoria a blocchi
  - I comandi includono read, write, seek (se random access)
  - accesso tramite file-system
  - **raw I/O** (array di blocchi), **direct I/O** (disabilita buffering e locking)
  - Possibile accesso a memory-mapped file
    - ▶ Non read/write etc. ma accesso simile ad accesso in memoria (array di bytes in memoria)
    - ▶ System call restituisce l'indirizzo di memoria del file mappato
    - ▶ file mappati su virtual memory e i cluster ottenuti con demand paging
  - DMA
- Character device (stream) includono tastiera, mouse, modem, porte seriali
  - Comandi tipici sono **get()** , **put()**
  - Librerie permettono la gestione ed il buffering di linee di caratteri (spaziatura)
  - ... anche per dispositivi di output (stampanti, schede audio, etc.)

# Device Block e Character

---

- Le interfacce ai block device permettono l'accesso alle unità disco e altri dispositivi con memoria a blocchi
  - I comandi includono read, write, seek (se random access)
  - accesso tramite file-system
  - Chiamate di sistema

```
#include <sys/types.h> /*data types*/
#include <sys/stat.h> /* data returned by stat()*/
#include <fcntl.h> /* file control options */

int open(const char *pathname, int oflag, ... /* mode_t mode */);

- Restituisce il descrittore del file, -1 in caso di errore;
- Permette sia di aprire un file già esistente che di creare il file nel caso in cui questo non esista;
- pathname è il pathname (assoluto o relativo) del file;
- oflag permette di specificare le opzioni, mediante costanti definite in <fcntl.h>, combinate con “|” (or bit-a-bit);
- mode è il modo del file ed è un parametro opzionale, utilizzato solo nel caso di creazione del file (“...” modo ISO C per dire variabile).

```

# Device Block e Character

---

- Le interfacce ai block device permettono l'accesso alle unità disco e altri dispositivi con memoria a blocchi
  - I comandi includono read, write, seek (se random access)
  - accesso tramite file-system
  - Chiamate di sistema

```
#include <unistd.h>

ssize_t read(int filedes, void *buf, size_t nbytes);

• Restituisce:

- il numero di byte effettivamente letti
- 0 se ci troviamo alla fine del file
- -1 in caso di errore

- L'operazione di lettura avviene partendo dall'offset corrente del file
 - L'offset viene incrementato opportunamente

```

# Device Block e Character

---

- Le interfacce ai block device permettono l'accesso alle unità disco e altri dispositivi con memoria a blocchi
  - I comandi includono read, write, seek (se random access)
  - accesso tramite file-system
  - Chiamate di sistema

```
#include <unistd.h>

ssize_t write(int filedes, void *buff, size_t nbytes);
```

- Restituisce:
  - il numero di byte effettivamente scritti
  - -1 in caso di errore
- L'operazione di scrittura avviene partendo dall'offset corrente del file
  - L'offset viene incrementato opportunamente

# Network Device

---

- Abbastanza diversi da block e character da meritare una loro interfaccia
- Diverso soprattutto l'indirizzamento e le performance
- Linux, Unix, Windows e molti altri SO hanno le interfacce **socket**
  - Separa il protocollo di rete dall'operazione sulla rete
  - Prima definisce il tipo di connessione e indirizzo poi permette la connessione
  - Primitive di send receive o read e write
  - Include la funzionalità **select()** per leggere da più socket (limitando busy waiting e polling)
- Molti altri strumenti sono a disposizione
  - In UNIX pipes, FIFOs, streams, queues, mailboxes, etc.

# Network Device

---

## □ Socket in Linux

```
#include <sys/socket.h>

int socket(int family, int type, int protocol);
```

- apre un socket, allocando una voce nella tabella dei file del kernel e permettendo la specifica del dominio, dello stile e del protocollo di comunicazione;
- *family* è la famiglia cui appartiene il protocollo (dominio);
- *type* definisce il tipo di comunicazione,
- *protocol* specifica il protocollo della famiglia;
- restituisce `-1` in caso di insuccesso, un numero **positivo** (il **socket descriptor**) altrimenti.

# Network Device

---

## □ Socket in Linux

```
int socket(int famiglia, int tipo, int protocollo);
```

- Crea un socket di una determinata categoria
- Nel nostro caso:
  - famiglia = PF\_LOCAL oppure PF\_INET
    - (PF = *Protocol Family*)
  - tipo = SOCK\_STREAM
  - protocollo = 0
- Restituisce il descrittore del socket, oppure -1

# Network Device

## □ Socket in Linux

|                                                                                                                                  |                                                                                                                                                                                                                                                                              |
|----------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Valori per <i>family</i></b><br>(AF_) PF_UNIX<br>(AF_) PF_INET<br>(AF_) PF_INET6<br>(AF_) PF_APPLETALK<br>(AF_) PF_X25<br>... | <b>Tipo di protocolli</b><br>Protocolli per comunicazioni locali su sistemi UNIX<br>Protocolli per Internet (vers. 4)<br>Protocolli per Internet (vers. 6)<br>Protocolli per LAN Appletalk<br>Protocolli dello standard ITU X.25 per reti a commutazione di pacchetti        |
| <b>Valori per <i>type</i></b><br>SOCK_STREAM<br>SOCK_DGRAM<br>SOCK_RAW<br>SOCK_SEQPACKET<br>...                                  | <b>Tipo di trasmissione</b><br>a flusso, sequenziale ed affidabile<br>a pacchetti di lungh. max. fissata, non sequenziale e non affidabile<br>accesso a basso livello (costruzione “manuale” dei pacchetti)<br>a pacchetti di lungh. max. fissata, sequenziale ed affidabile |
| <b>Valori per <i>protocol</i></b><br>IPPROTO_TCP<br>IPPROTO_UDP<br>IPPROTO_RAW<br>IPPROTO_ICMP<br>...                            | <b>Protocollo della famiglia INET</b> (definiti in <netinet/in.h>)<br>TCP<br>UDP<br>IP<br>ICMP                                                                                                                                                                               |

# Network Device

---

## □ Socket in Linux

```
int fd = socket(PF_LOCAL, SOCK_STREAM, 0);
```

```
if (fd<0) perror("socket"), exit(1);
```

```
int fd = socket(PF_INET, SOCK_STREAM, 0);
```

```
if (fd<0) perror("socket"), exit(1);
```

```
int fd = socket(PF_INET, SOCK_DGRAM, 0);
```

```
if (fd<0) perror("socket"), exit(1);
```

# Network Device

---

## □ Socket in Linux

```
int fd = socket(PF_LOCAL, SOCK_STREAM, 0);
```

```
if (fd<0) perror("socket"), exit(1);
```

```
int fd = socket(PF_INET, SOCK_STREAM, 0);
```

```
if (fd<0) perror("socket"), exit(1);
```

```
int fd = socket(PF_INET, SOCK_DGRAM, 0);
```

```
if (fd<0) perror("socket"), exit(1);
```

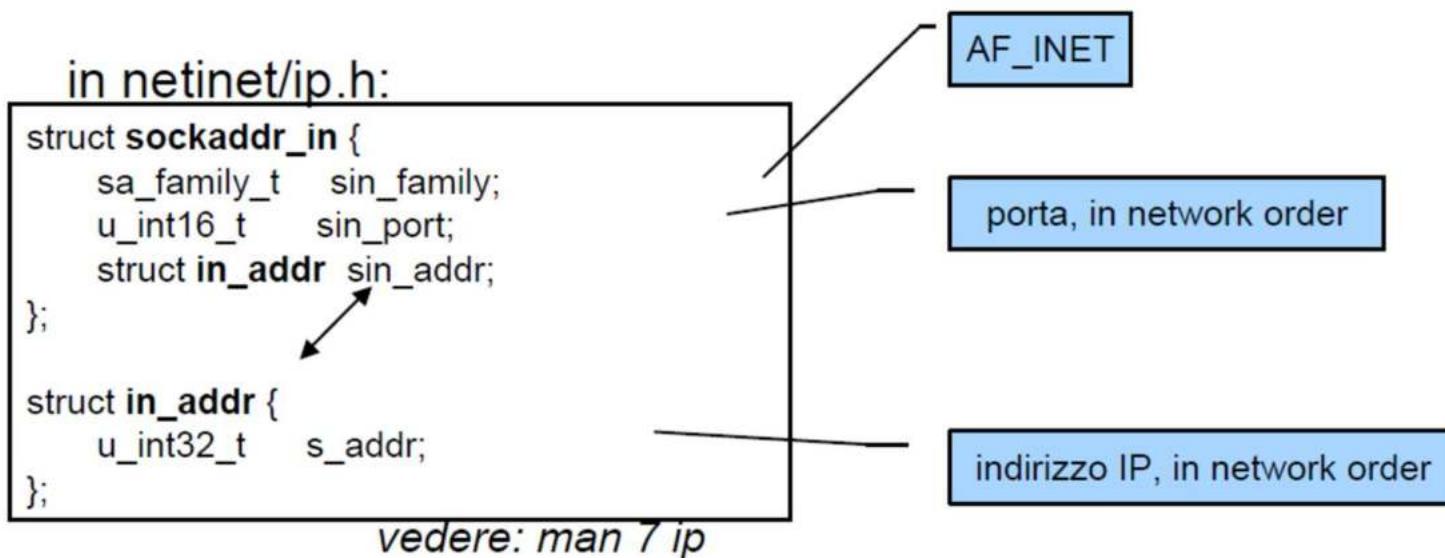
# Network Device

- Socket in Linux
  - Indirizzi TCP - bind

```
struct sockaddr_in mio_indirizzo;

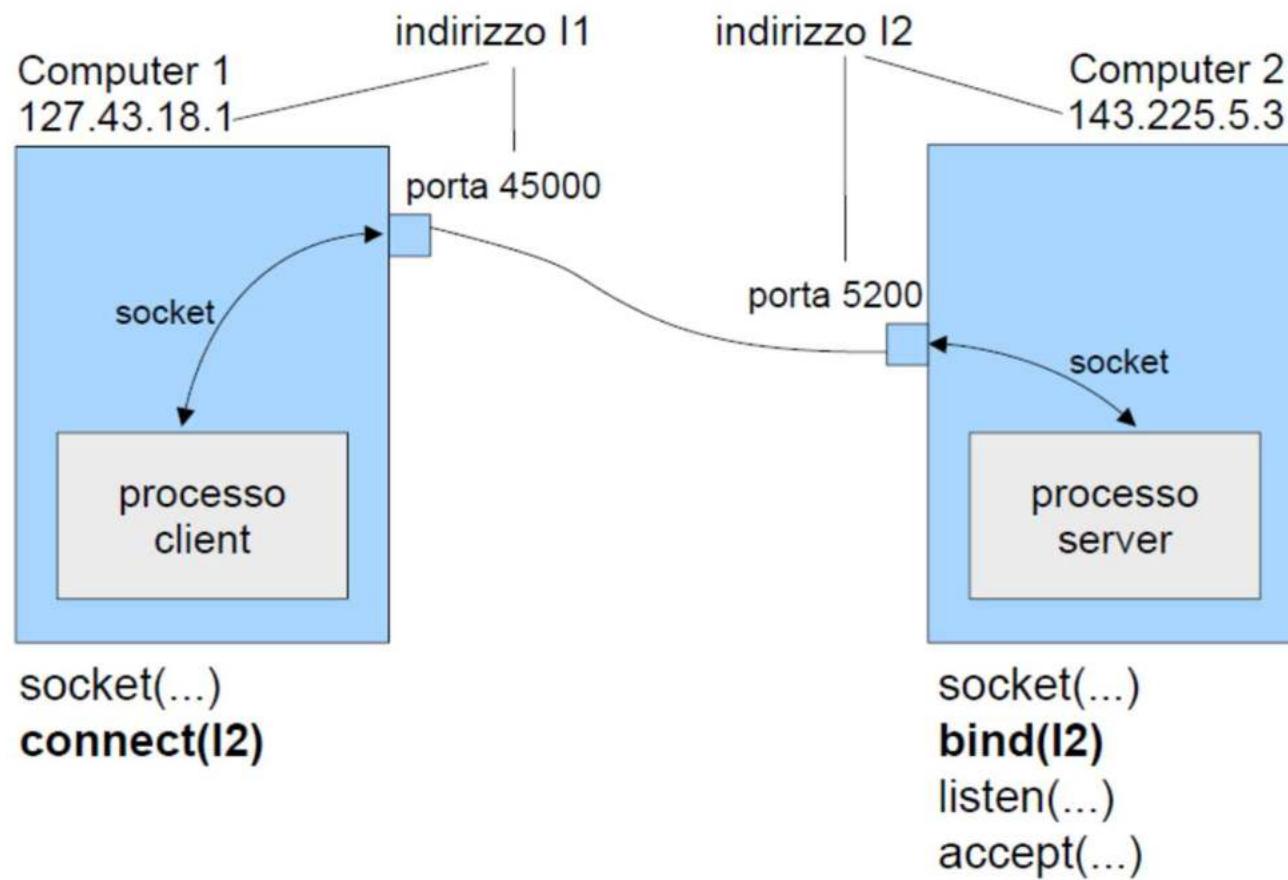
mio_indirizzo.sin_family = AF_INET;
mio_indirizzo.sin_port = htons(5200);
mio_indirizzo.sin_addr.s_addr = htonl(INADDR_ANY);

bind(fd, (struct sockaddr *) &mio_indirizzo, sizeof(mio_indirizzo));
```



# Network Device

## □ Socket in Linux



# Network Device

---

- Socket in Linux

- Server

```
int fd1, fd2;
struct sockaddr_in mio_indirizzo;

mio_indirizzo.sin_family = AF_INET;
mio_indirizzo.sin_port = htons(5200);
mio_indirizzo.sin_addr.s_addr = htonl(INADDR_ANY);

fd1 = socket(PF_INET, SOCK_STREAM, 0);
bind(fd1, (struct sockaddr *) &mio_indirizzo, sizeof(mio_indirizzo));

listen(fd1, 5);
fd2 = accept(fd1, NULL, NULL);
...
close(fd2);
close(fd1);
```

# Network Device

---

- Socket in Linux

- Client

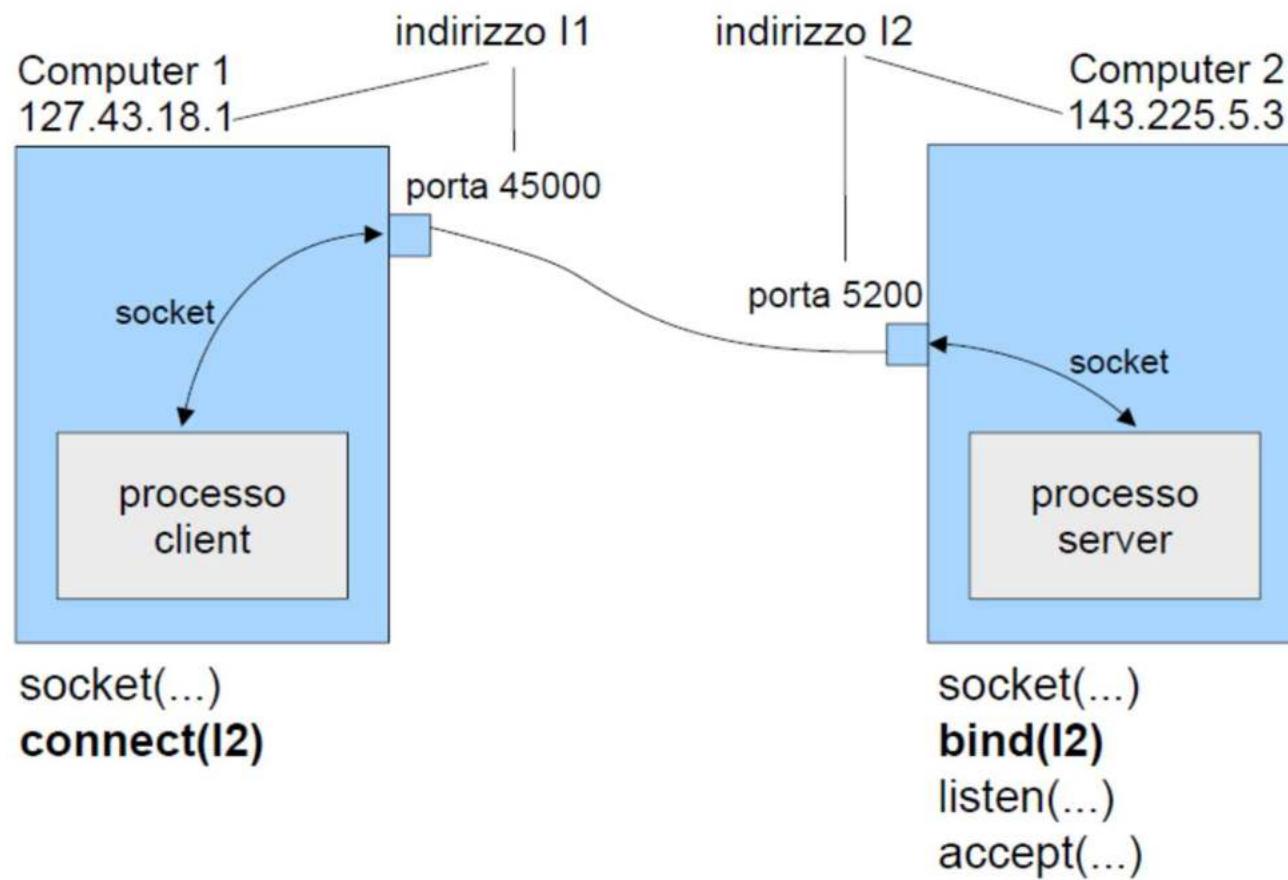
```
int fd;
struct sockaddr_in mio_indirizzo;

mio_indirizzo.sin_family = AF_INET;
mio_indirizzo.sin_port = htons(5200);
inet_aton("143.225.5.3", &mio_indirizzo.sin_addr);

fd = socket(PF_INET, SOCK_STREAM, 0);
connect(fd, (struct sockaddr *) &mio_indirizzo,
sizeof(mio_indirizzo));
...
close(fd);
```

# Network Device

## □ Socket in Linux



# Network Device

---

- Socket in Linux

- scrittura

```
#include <unistd.h>
ssize_t write(int fd,
 const void *buf,
 size_t count);
```

- invia il contenuto del buffer buf al socket specificato
- si usa esclusivamente con SOCK\_STREAM
- restituisce il numero di byte inviati oppure -1 in caso di errore
- è la stessa funzione che consente la scrittura su file

# Network Device

---

## □ Socket in Linux

### □ lettura

```
#include <unistd.h>
ssize_t read(int fd,
 void *buf,
 size_t count);
```

- solo per socket connessi (SOCK\_STREAM)
- legge un messaggio di lunghezza massima `len` dal socket
- se non c'è alcun messaggio, il programma rimane sospeso
  - chiamata *bloccante*
- la funzione ritorna il numero di byte letti, -1 in caso di errore
- è la stessa funzione che consente la lettura da un file

# Clock e Timer

---

- Forniscono funzionalità:
  - Get current time,
  - Get elapsed time
  - Set Timer (per innescare un'operazione al tempo T)
- Usati intensamente ma le sys call non standardizzate tra SO
- Hardware:
  - **Programmable interval timer** usati per timing e periodic interrupt
    - Scheduler per attribuire gli slice, operazioni periodiche su disco o in rete
    - Accesso agli utenti
    - Diversi clock anche clock virtuali
- Risoluzione tipica 1/60 sec (anche più alta risoluzione)
  - Clock diversi, es. high-performance event timer (HPET) 10 megahertz
  - Network Time Protocol (NTP) per correggere il drift the clock
- `ioctl()` (su UNIX) gestisce aspetti strani di I/O come clock e timer

# Clock e Timer

---

- Esempio programma che legge il tempo

```
#include <time.h>
...
char buffer[26];
time_t ora;
time(&ora);
printf(" Ora esatta : %s\n",
 ctime_r(&ora, buffer));
```

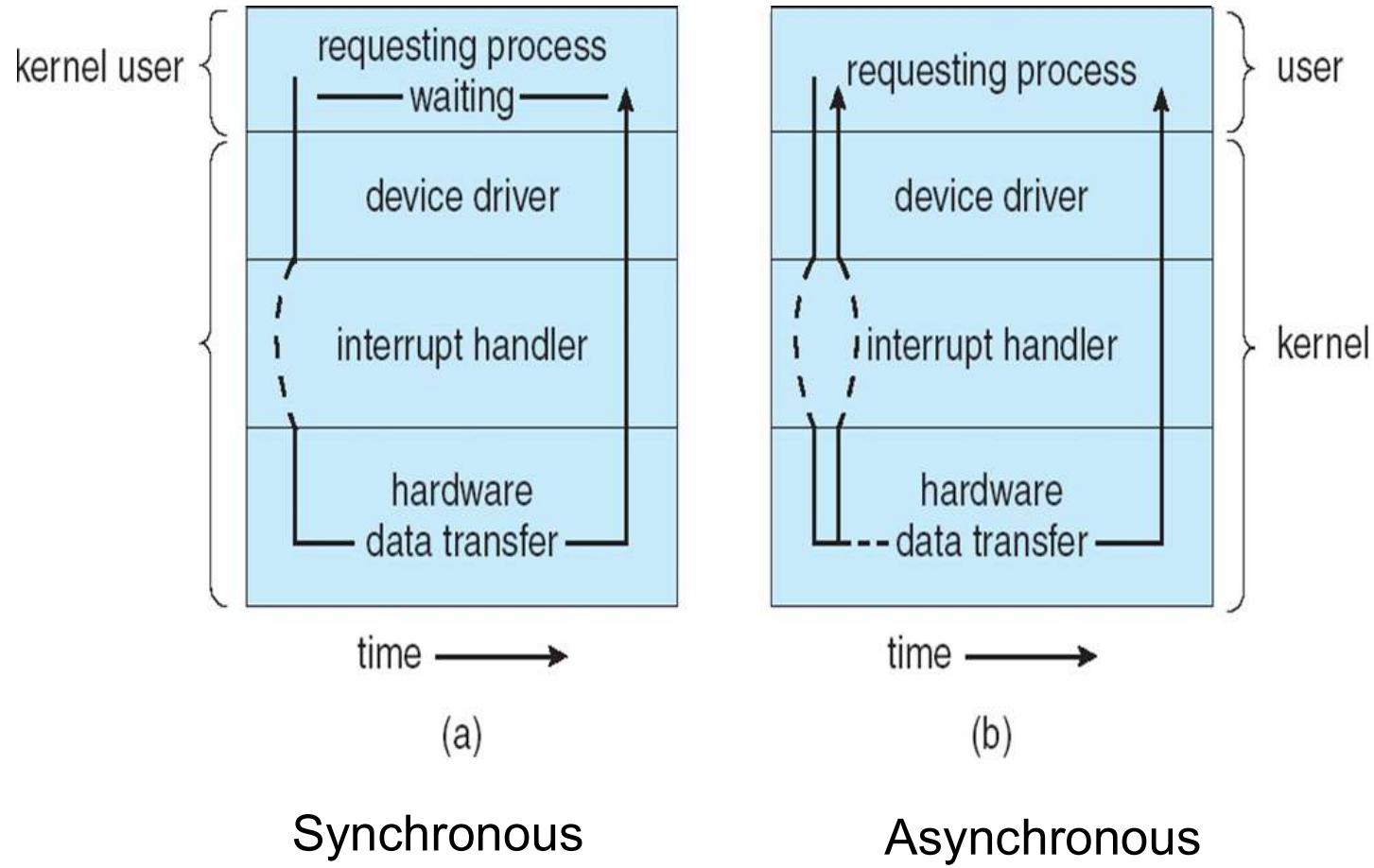
- la funzione `time` restituisce l'ora in un formato interno (`time_t`)
- la funzione `ctime_r` trasforma il formato interno in stringa; ha bisogno di un buffer di (almeno) 26 caratteri

# Nonblocking e Asynchronous I/O

---

- **Blocking** - processi sospesi finché I/O non è completato
  - Facile da usare e capire
  - Problematico in alcune circostanze
- **Nonblocking** - I/O call ritorna appena possibile
  - User interface (keyboard and screen)
  - data copy (buffered I/O)
  - Implementata via multi-threading
  - Ritorna rapidamente con la conta dei bytes read o written
  - `select()` per vedere se dati pronti poi `read()` o `write()` per il trasferimento
- **Asynchronous** - processo gira mentre I/O esegue
  - Difficile da usare
  - I/O subsystem segnala al processo quando I/O è completata

# Due Metodi I/O



# Vectored I/O

---

- Vectored I/O permette ad una system call di fare operazioni multiple di I/O (anche detto scatter/gather I/O)
- Prende dati da più buffer e li scrive in uno stream
- Esempio, Unix `readve()` accetta un vettore di buffer multipli per leggere o scrivere
- Questo metodo scatter-gather migliore di multiple chiamate di I/O
  - Minore context switching e system call overhead
  - Alcune versioni forniscono atomicity
    - ▶ Evita per esempio di preoccuparsi di threads multipli durante i read / write

# Vectored I/O

---

- Vectored I/O permette ad una system call di fare operazioni multiple di I/O (anche detto scatter/gather I/O)
- Prende dati da più buffer e li scrive in uno stream
- Esempio, Unix `readv()` accetta un vettore di buffer multipli per leggere o scrivere

```
#include <sys/uio.h>
ssize_t readv (int fd, const struct iovec *iov, int count);
```

```
#include <sys/uio.h>
ssize_t writev (int fd, const struct iovec *iov, int count);
```

Each `iovec` structure describes an independent disjoint buffer, which is called a *segment*:

```
struct iovec {
 void *iov_base; /* pointer to start of buffer */
 size_t iov_len; /* size of buffer in bytes */
};
```

# Vectored I/O

## □ Vectored I/O

```
int main () {
 struct iovec iov[3]; ssize_t nr; int fd, i;

 char *buf[] = { "The term buccaneer comes from the word boucan.\n",
 "A boucan is a wooden frame used for cooking meat.\n",
 "Buccaneer is the West Indies name for a pirate.\n" };

 fd = open ("buccaneer.txt", O_WRONLY | O_CREAT | O_TRUNC);
 if (fd == -1) { perror ("open"); return 1; } /* fill out three iovec structures */

 for (i = 0; i < 3; i++) {
 iov[i].iov_base = buf[i]; iov[i].iov_len = strlen(buf[i]) + 1;
 }
 /* with a single call, write them all out */
 nr = writev (fd, iov, 3);

 if (nr == -1) {
 perror ("writev");
 return 1;
 }
 printf ("wrote %d bytes\n", nr);
 if (close (fd)) { perror ("close"); return 1; }

 return 0;
}
```

\$ ./writev wrote 148 bytes

# Vectored I/O

## □ Vectored I/O

```
int main () {
 char foo[48], bar[51], baz[49];

 struct iovec iov[3]; ssize_t nr; int fd, i;

 fd = open ("buccaneer.txt", O_RDONLY);

 if (fd == -1) { perror ("open"); return 1; }

 /* set up our iovec structures */
 iov[0].iov_base = foo; iov[0].iov_len = sizeof (foo);
 iov[1].iov_base = bar; iov[1].iov_len = sizeof (bar);
 iov[2].iov_base = baz; iov[2].iov_len = sizeof (baz);

 /* read into the structures with a single call */
 nr = readv (fd, iov, 3);

 if (nr == -1) { perror ("readv"); return 1; }

 for (i = 0; i < 3; i++)
 printf ("%d: %s", i, (char *) iov[i].iov_base);

 if (close (fd)) { perror ("close"); return 1; }

 return 0;

}
```

```
$./readv
```

- 0: The term buccaneer comes from the word boucan.
- 1: A boucan is a wooden frame used for cooking meat.
- 2: Buccaneer is the West Indies name for a pirate.

# Sottosistema I/O del Kernel

---

- Il Kernel fornisce molti servizi per la gestione dell'I/O che sono realizzati a partire dai dispositivi e dei driver relativi

scheduling

gestione del  
buffer

gestione delle  
cache

gestione delle  
code di spooling

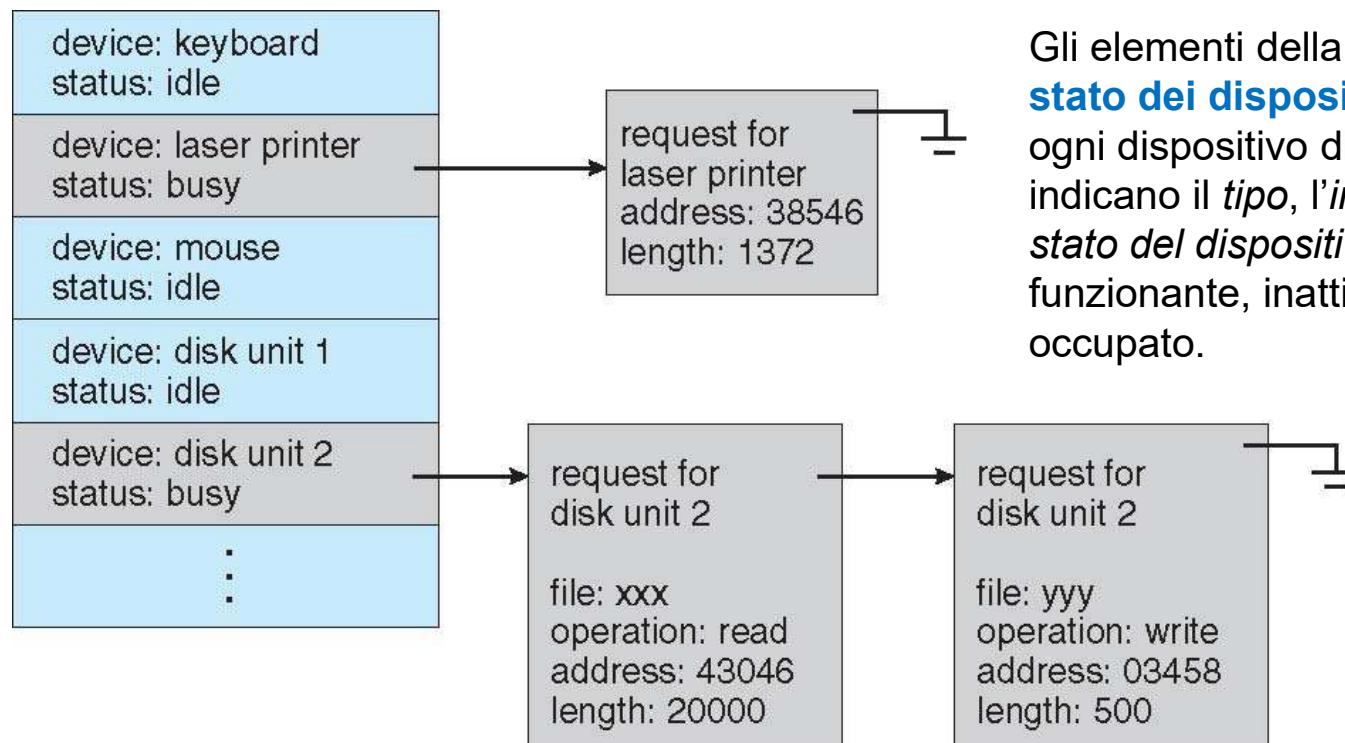
riservazione dei  
dispositivi

gestione degli  
errori →  
protezione  
dell'I/O

# Sottosistema I/O del Kernel

## □ I/O Scheduling

- Riordina le richieste di I/O sulle code per-device
- Alcuni SO mantengono un Quality Of Service (i.e., IPQOS) per le richieste oppure precedenza a richieste delay-sensitive
- SO deve mantenere traccia di diverse richieste asincrone, per questo mantiene le code su tabella dello stato dei dispositivi (device status table)



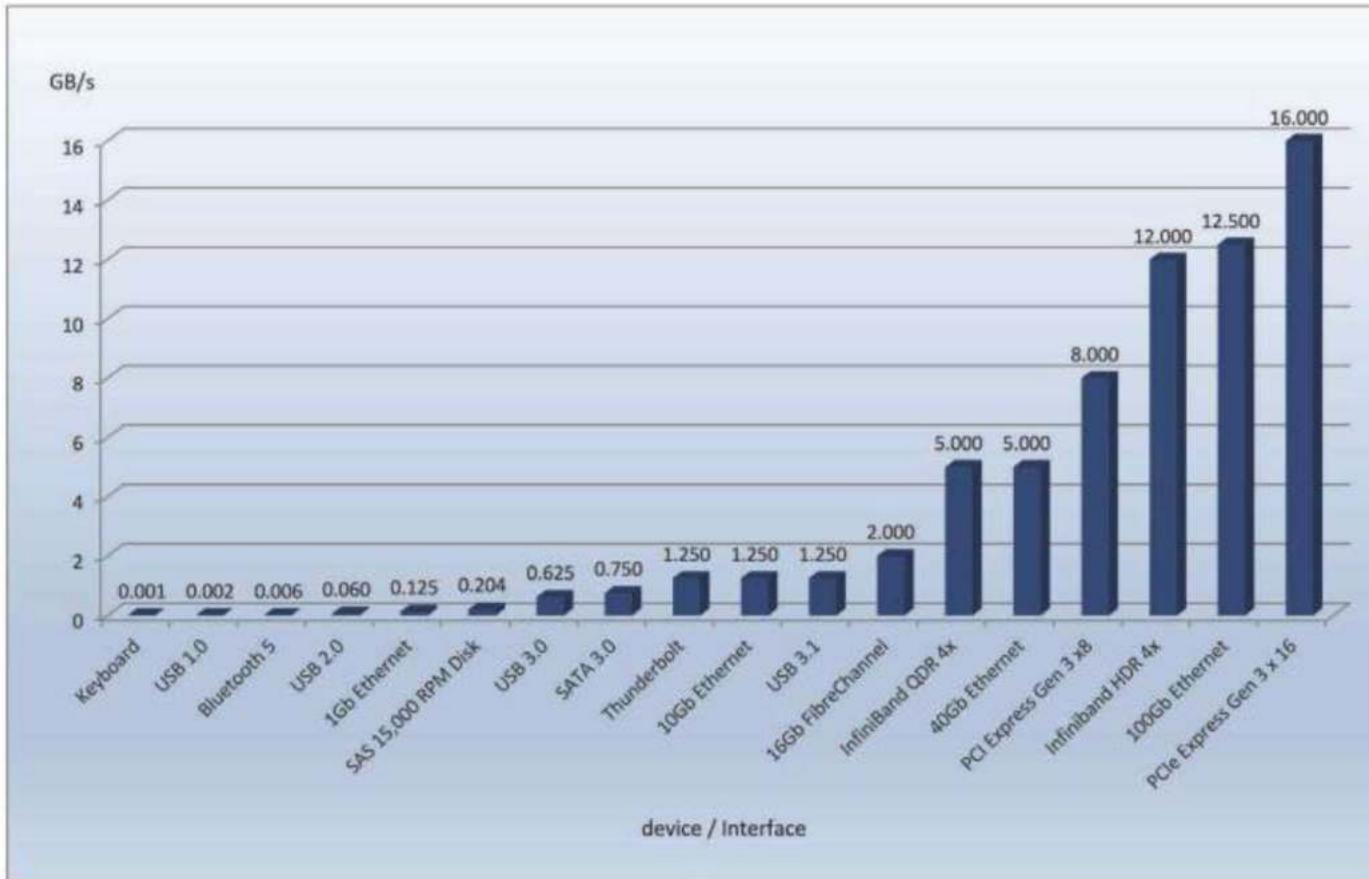
# Sottosistema I/O del Kernel

---

- **Buffering** - immagazzina dati in memoria durante il trasferimento dati tra dispositivi.
- Tre motivi principali:
  - Per gestire la differenza di velocità dei device
    - Esempio raccolta dati da modem per copiare su disco
    - **Double buffering** – due buffer, uno per raccogliere, l'altro per processare (disaccoppia produttore e consumatore del dato)
    - Full / viene processato, not-full / viene usato
  - Per gestire la differenza di dimensione di dati da trasferire
    - Esempio riassembaggio di pacchetti da rete
  - Per mantenere una “copia semantica”
    - Il dato da copiare su buffer per copiare il contenuto a tempo di chiamata
    - Il dato bufferizzato nel Kernel e non modificato da processo utente
    - Copy-on-write può essere usato for efficienza in qualche caso

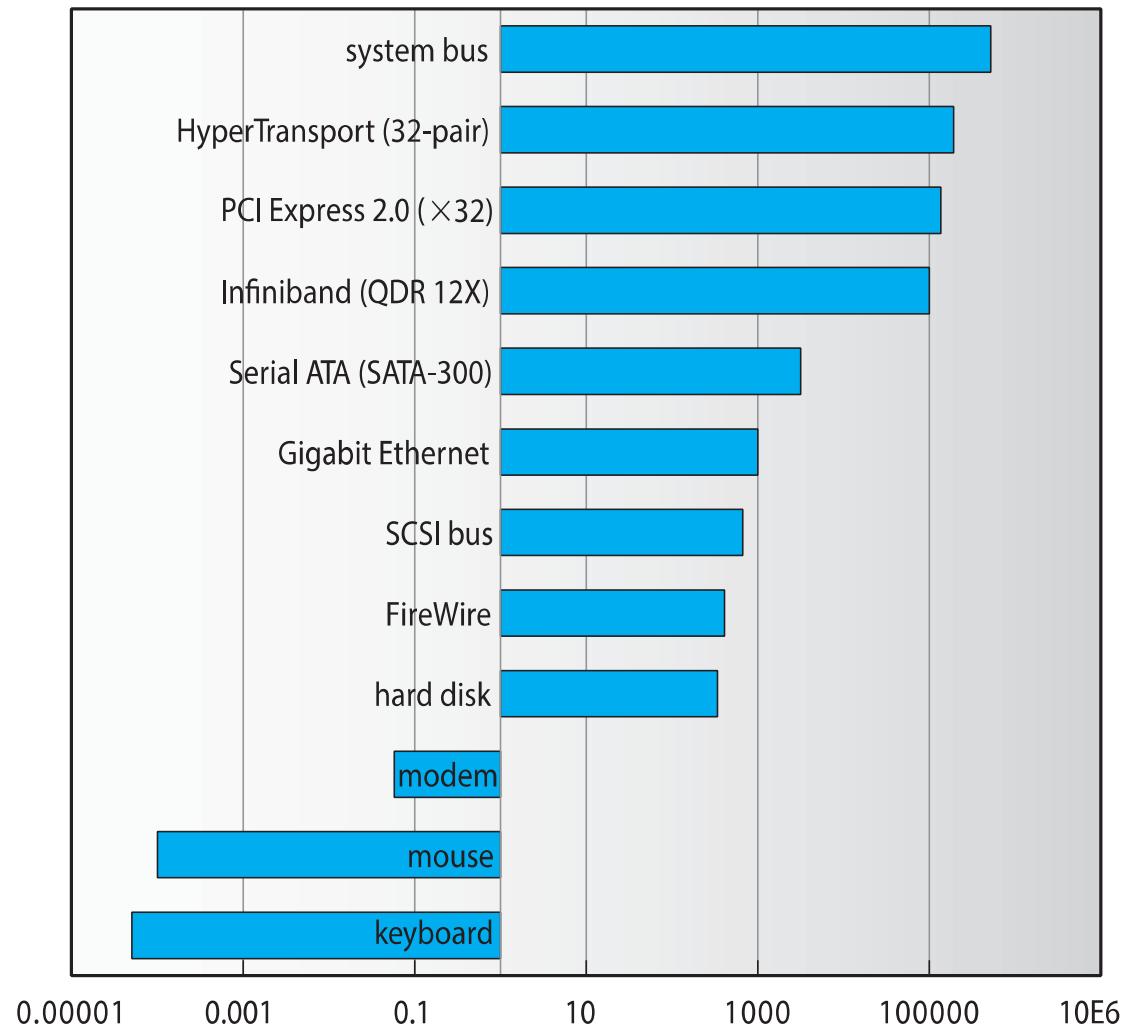
# Sottosistema I/O del Kernel

- **Buffering** - immagazzina dati in memoria durante il trasferimento tra dispositivi.



Le enormi differenze di velocità tra i dispositivi tipici di un calcolatore

# Sun Enterprise 6000 Device-Transfer Rates



# Sottosistema I/O del Kernel

---

- **Caching** – dispositivo più veloce che tiene una copia del dato
  - Es. copia su disco e su cache primaria e secondaria della CPU
  - Sempre solo una copia (buffer non copia)
  - Curciale per la performance
  - A volte combinato con il buffering
- **Spooling** -
  - Simultaneous Peripheral Operations On-line
  - Spool è un buffer che mantiene gli output per un dispositivo (es., printer)
  - Se il device può servire solo una richiesta alla volta (non interleaving)
  - Gestione delle code sui dispositivi
- **Device reservation** - accesso esclusivo ad un dispositivo
  - System call per allocation e de-allocation

# Sottosistema I/O del Kernel

---

- **Error Handling**
- SO può proteggere e recuperare da diversi errori transitori
- SO può recuperare da: lettura disco, fallimento di scrittura, etc.
  - Ritenta un read o write, per esempio
  - Alcuni sistemi più sofisticati – Solaris FMA, AIX
    - ▶ Traccia di frequenza di errori
  - Hardware possono fornire errori più dettagliati non sempre usati da SO
    - ▶ Es. SCASI, tre livelli di errore (sense key, additional sense code, add sense code qualifie)
- In UNIX chiamata di sistema forniscono un error number (errno) quando la richiesta di I/O fallisce (centinaia)
  - Es. bad pointer, out of range, file not open
- Gli error log di sistema mantengono i report dei problemi

# Sottosistema I/O del Kernel

---

## □ Error Handling

### □ Linux

- Molte system call restituiscono -1 in caso di errore
- Per avere piu' informazioni, si usa la variabile globale **errno** (error number)
- La funzione **perror**(const char \*) stampa la stringa passata come parametro, e poi un messaggio in base al valore corrente di errno
- Esempi:

```
int fd = open("prova.txt", O_RDONLY);
if (fd<0) perror("errore di open");
```

```
int fd;
if ((fd=open("prova.txt", O_RDONLY)) < 0)
 perror("errore di open");
```

# Protezione I/O

- Processo utente può accidentalmente o volontariamente tentare di corrompere le operazioni con istruzioni illegali I/O
- Tutte le istruzioni I/O definite come privilegiate
  - I/O deve essere fatta via system call
    - ▶ Tutte Memory-mapped e locazioni delle porte I/O devono essere protette

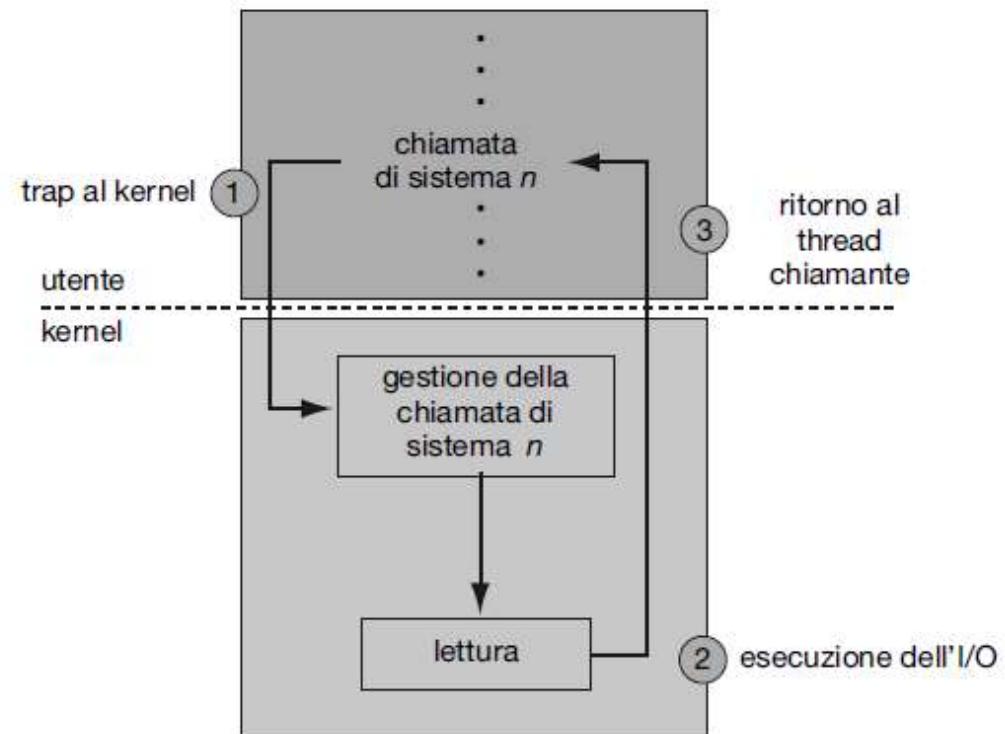


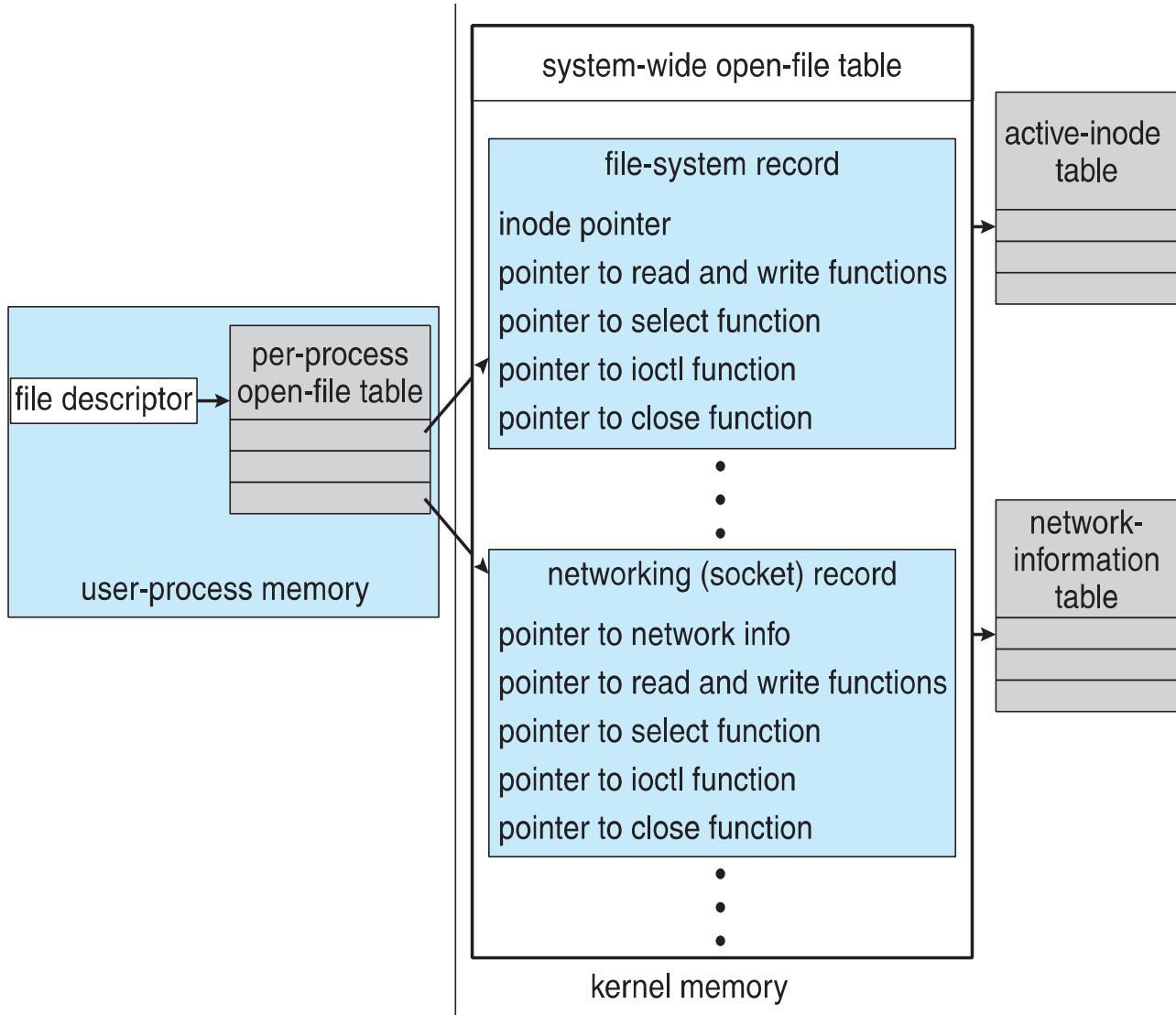
Figura 12.12 Uso delle chiamate di sistema per eseguire I/O.

# Strutture Dati del Kernel

---

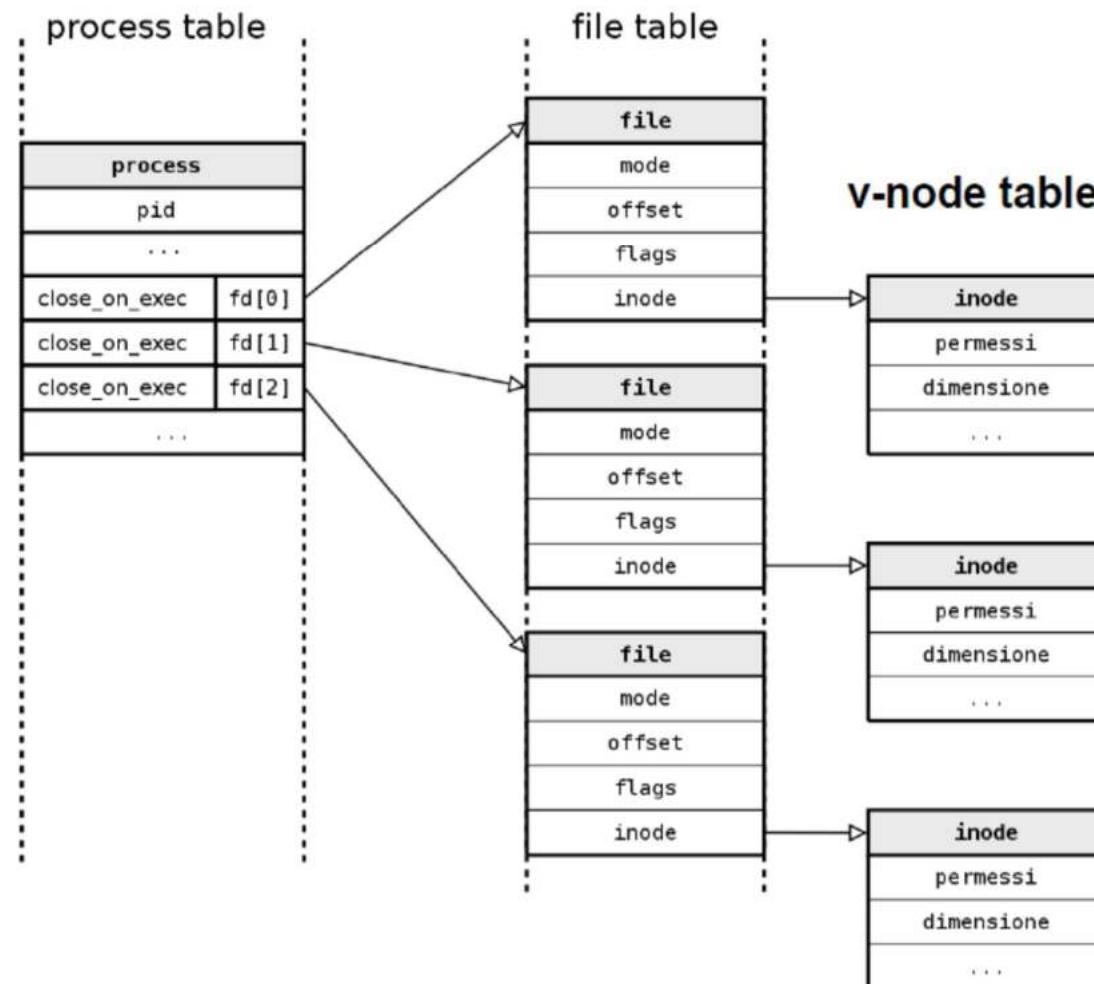
- Kernel mantiene info di stato per tutte le componenti I/O, incluse tabelle dei open file, connessione di rete, stato dei character device, etc.
- Molte strutture dati complesse per tracciare buffer, allocazione di memoria, “dirty” block
- Unix permette un accesso da file system ad unità eterogenee:
  - User files, raw data, etc.
  - Read() uniforme, ma semantica differente a seconda del tipo
- Si definiscono strutture dati uniformi per gestire queste differenze
- Ad esempio la struttura dei file aperti di UNIX può gestire tipi di diversi di file aperti con puntatori a strutture differenti

# UNIX - Strutture I/O del Kernel



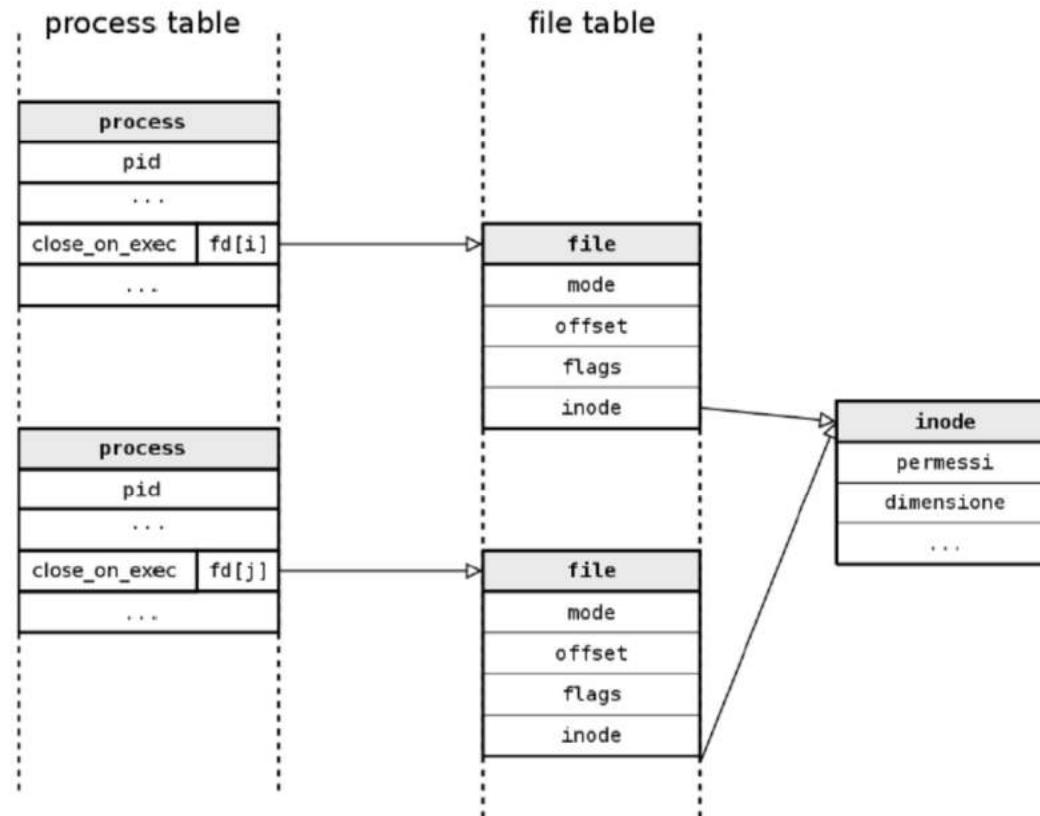
# UNIX - Strutture I/O del Kernel

Un processo con 3 descrittori aperti



# UNIX - Strutture I/O del Kernel

Due processi che accedono allo stesso file



# Strutture Dati del Kernel

---

- Kernel mantiene info di stato per tutte le componenti I/O, incluse tabelle dei open file, connessione di rete, stato dei character device, etc.
- Molte strutture dati complesse per tracciare buffer, allocazione di memoria, “dirty” block
- A volte metodi di message passing per implementare I/O
  - Windows usa message passing invece che chiamate di procedura
    - Messaggio con info I/O passato da user mode a kernel
    - Messaggio modificato se passa per il device driver e indietro al processo
    - Pros: semplice progettazione e flessibilità
    - Cons: overhead di gestione

# Power Management

---

- Non strettamente pertinenza di I/O, ma riguarda I/O
- Computers e devices usano elettricità, generano calore, richiedono raffreddamento
- SO possono aiutare a gestire e migliorare l'uso dell'energia
- Mobile computing ha power management come una caratteristica fondamentale del SO

# Power Management

---

- Per esempio, Android implementa 3 componenti:
  - Power collapse – mette il dispositivo in “very deep sleep”
    - Uso marginale di energia
    - Solo per respondere a stimuli esterni (button press, incoming call)
  - Component-level power management
    - Usa relazioni tra componenti
    - Costruisce alberi di device che rappresentano la topologia fisica dei device
    - CPU -> System bus -> I/O subsystem -> {flash, USB storage}
    - Device driver tracciano lo stato del device, se in uso
    - Componenti non usati – spenti
    - Tutti i device in un ramo dell’albero non usati – spenti
  - Wake locks – come altri lock ma previene sleep del device quando c’è il lock

# Power Management

---

- Power Management è parte di Device Management
  - Al boot Device tree in RAM
  - Device tree usato per caricare i device driver
- Moderni computer usano ACPI (Avanced Configuration and Power Interface)
  - Firmware code
  - Standard per gestire hardware
  - Fornisce il controllo dei dispositivi al SO
  - Routine per scoprire lo stato dei dispositivi e gestirli

# Concetti principali del sottosistema di I/O del kernel

Il sistema per l'I/O coordina un'ampia raccolta di servizi disponibili per le applicazioni e per altre parti del kernel:

gestione dello spazio  
dei nomi per file e  
dispositivi

controllo  
dell'accesso ai file e  
ai dispositivi

controllo delle  
operazioni

allocazione dello  
spazio per il file  
system

allocazione dei  
dispositivi

gestione dei buffer,  
delle cache e delle  
code di spooling

scheduling dell'I/O

controllo dello stato  
dei dispositivi,  
gestione degli errori  
e procedure di  
ripristino

configurazione e  
inizializzazione dei  
driver dei dispositivi

gestione energetica  
dei dispositivi dell'I/O

# Richiesta I/O e Operazioni Hardware

---

- Vediamo come il SO serve una richiesta I/O da processo utente
- Consideramo il problema di lettura di un file dal disco:
  - Applicazione si riferisce ad un filename
  - Il Sistema gestisce il file system per trovare la locazione del file
    - ▶ Es. In Unix si trova inode number e inode ha info su allocazione file
  - Come determinare il device che ha il file?
    - ▶ In windows nome volume nel pathname, in Unix mount table che associa prefissi a nomi di device
  - Traduce il nome alla rappresentazione del device
  - Legge fisicamente i dati da disco a buffer
  - Rende i dati disponibili al processo richiedente
  - Restituisce il controllo al processo

# Ciclo di vita di una richiesta I/O

Richiesta di una read() con open già eseguito

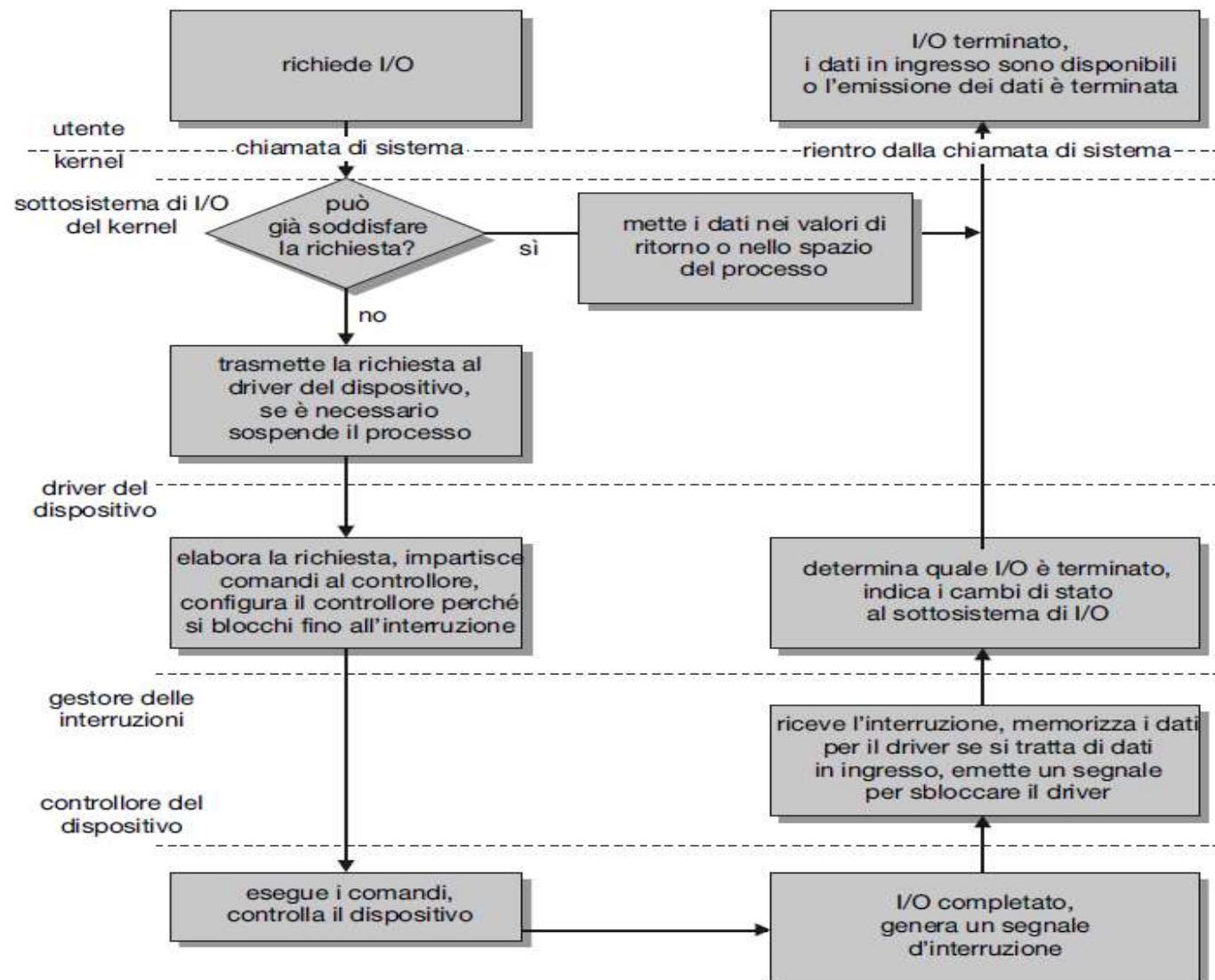


Figura 12.14 Schema d'esecuzione di una richiesta di I/O.

# STREAMS

- **STREAMS** – canale di comunicazione full-duplex tra processo user-level e dispositivo in Unix System V e oltre:
  - stream head, driver end, stream module

**STREAMS** è una realizzazione e una metodologia che permettono di sviluppare in modo modulare e incrementale i driver e i protocolli di rete.

Utilizzando gli *stream*, i driver possono essere organizzati in una catena, attraverso cui passano i dati in maniera sequenziale e bidirezionale per l'elaborazione

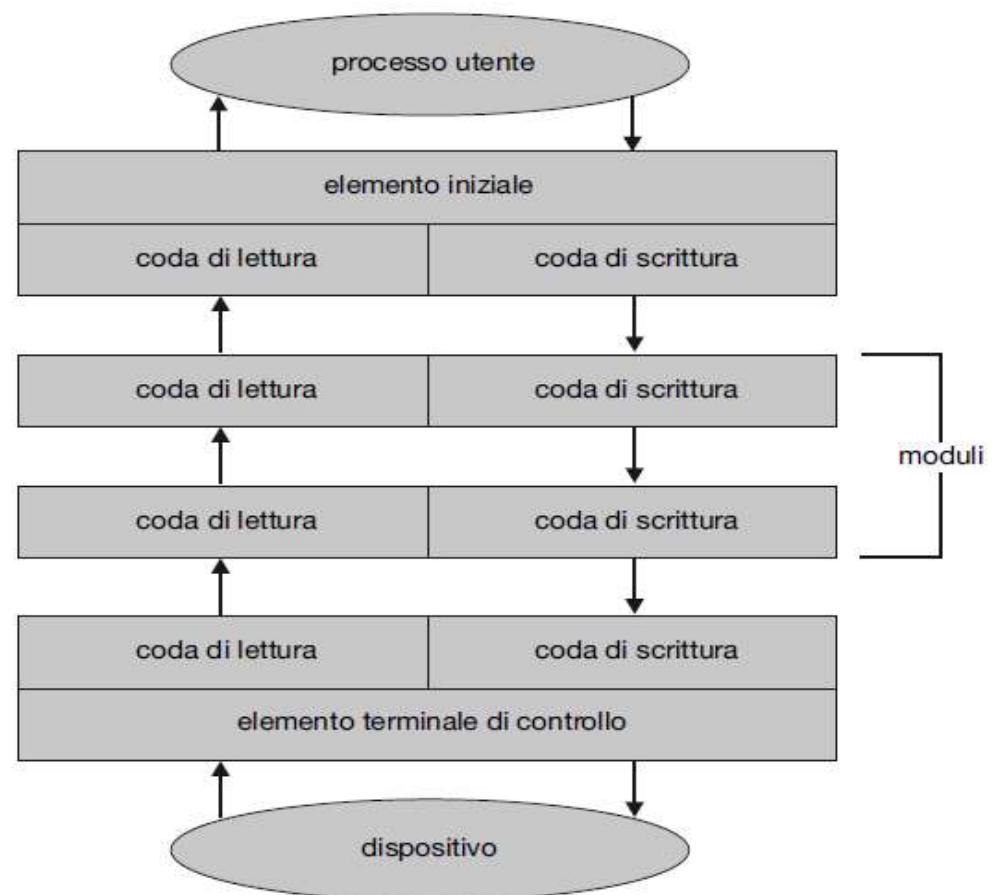


Figura 12.15 Struttura di STREAMS.

# Prestazioni

---

- Gestione I/O come principale fattore per la performance di Sistema:
  - Richiede CPU per eseguire device driver, kernel I/O code
  - Context switche dovuti ad interrupt
  - Copia dei dati
  - Traffico di rete

# Prestazioni

A causa dei molti strati di software presenti fra un dispositivo fisico e l'applicazione, le **chiamate di sistema per l'I/O** sono **onerose** in termini di utilizzazione della CPU. Anche il **traffico di una rete** può portare a un **alto numero di cambi di contesto**; si consideri il login remoto da un calcolatore a un altro.

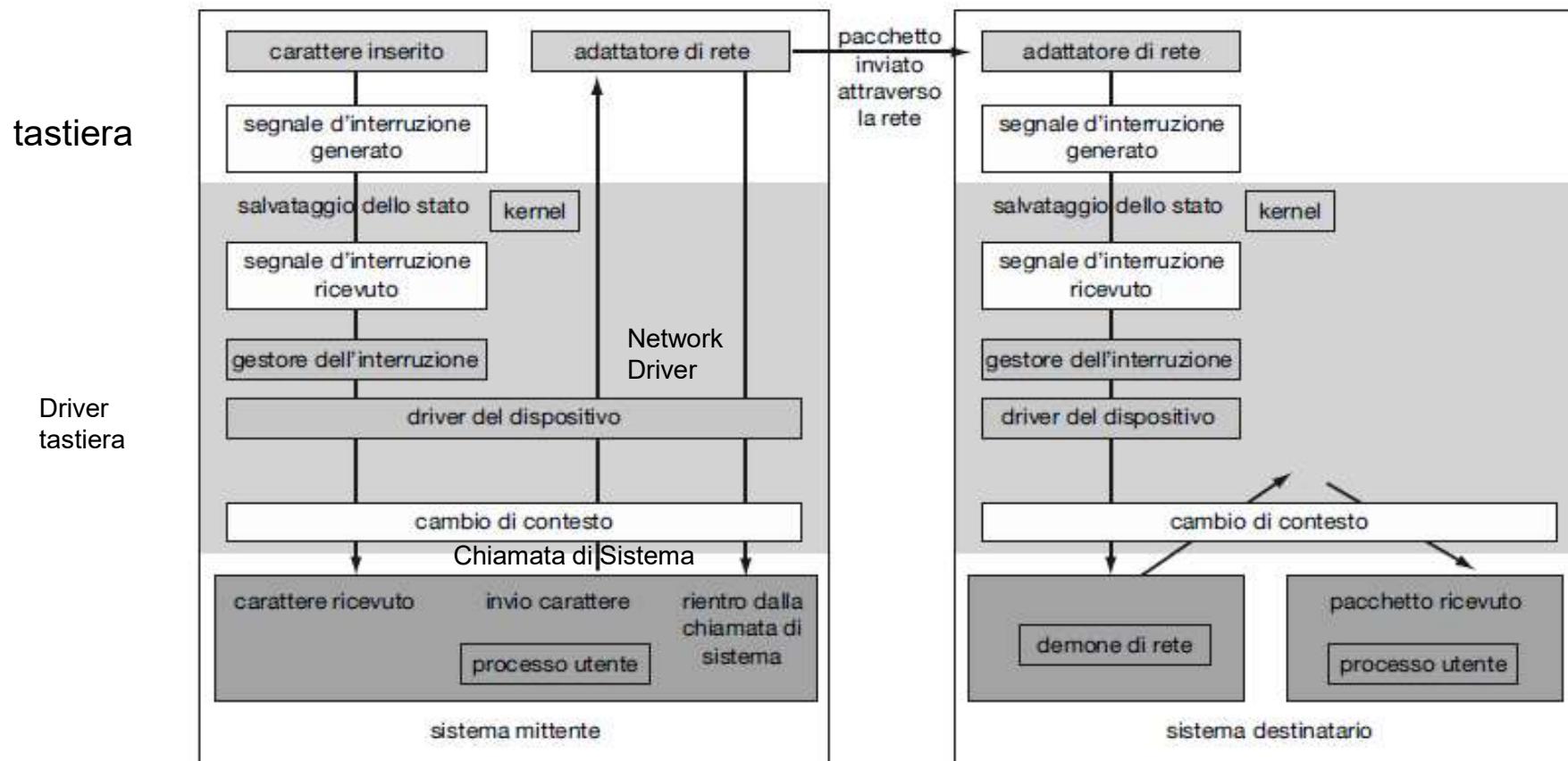


Figura 12.16 Comunicazione tra calcolatori.

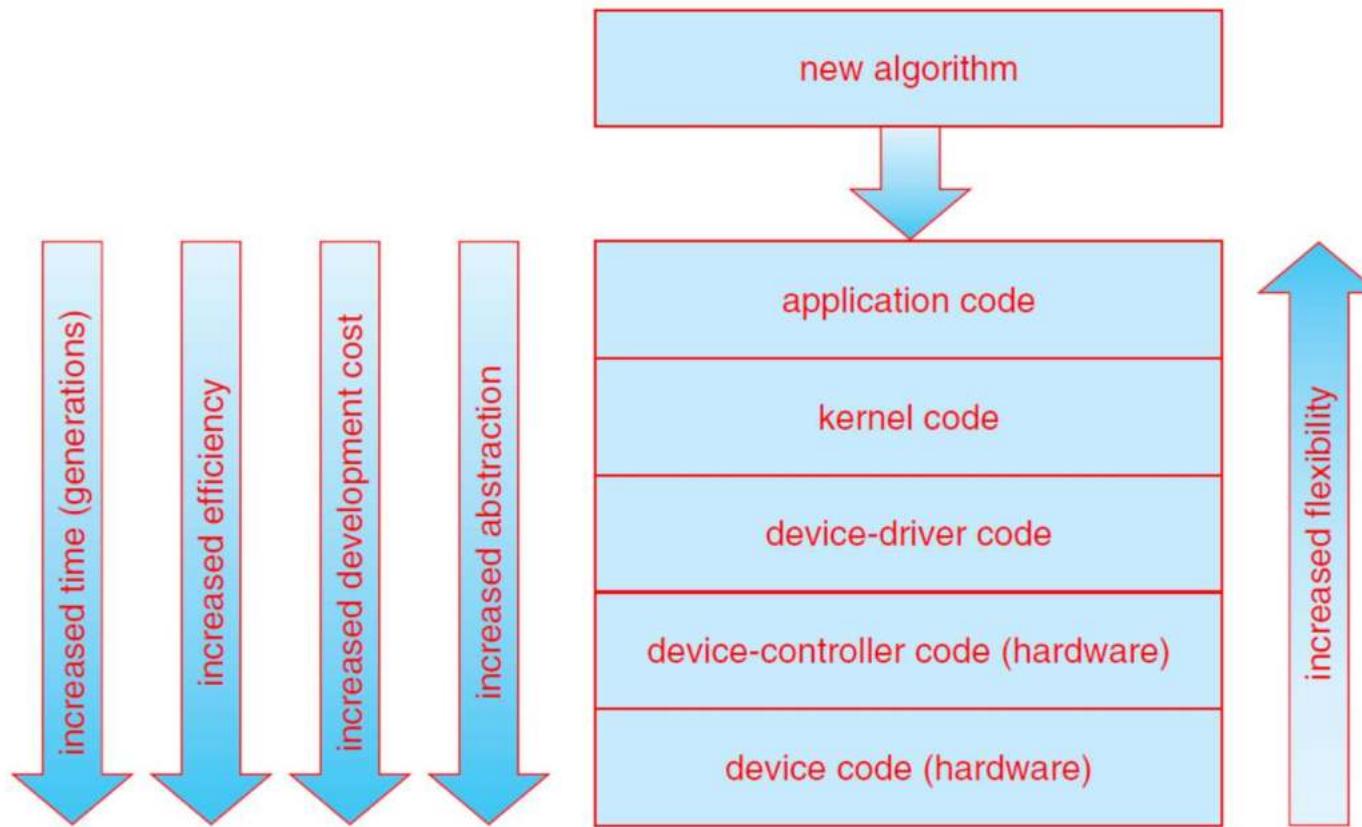
# Migliorare la Prestazione

---

- Ridurre il numero di context switch
- Ridurre il data copying mentre i dati passano dai dispositivi alla memoria
- Ridurre il numero degli interrupt usando grandi transferimenti di dati, smart controller e polling quando è possibile
- Usare quando possibile il DMA o altri canali diretti di comunicazione che bypassano la CPU per data copying semplice
- Usare dispositivi con smarter hardware per delegare il processamento semplice di dati e farlo in concorrenza con la CPU senza impegnare il bus
- Bilanciare CPU, memory, bus, e I/O performance perché ogni sbilanciamento crea un collo di bottiglia

# Progressione delle funzionalità dei Device

- Dove implementare le funzionalità dei dispositivi?
- Gestite al livello hardware? Device driver? Software applicativo?
- La tendenza è la seguente:



# Prestazioni e Latenza dei Dispositivi

- Dispositivi aumentano di velocità e capacità
- Per gestire occorrono sistemi I/O sempre migliori
  - Rapida lettura e scrittura
- Le NVM aumentano di prestazione ...

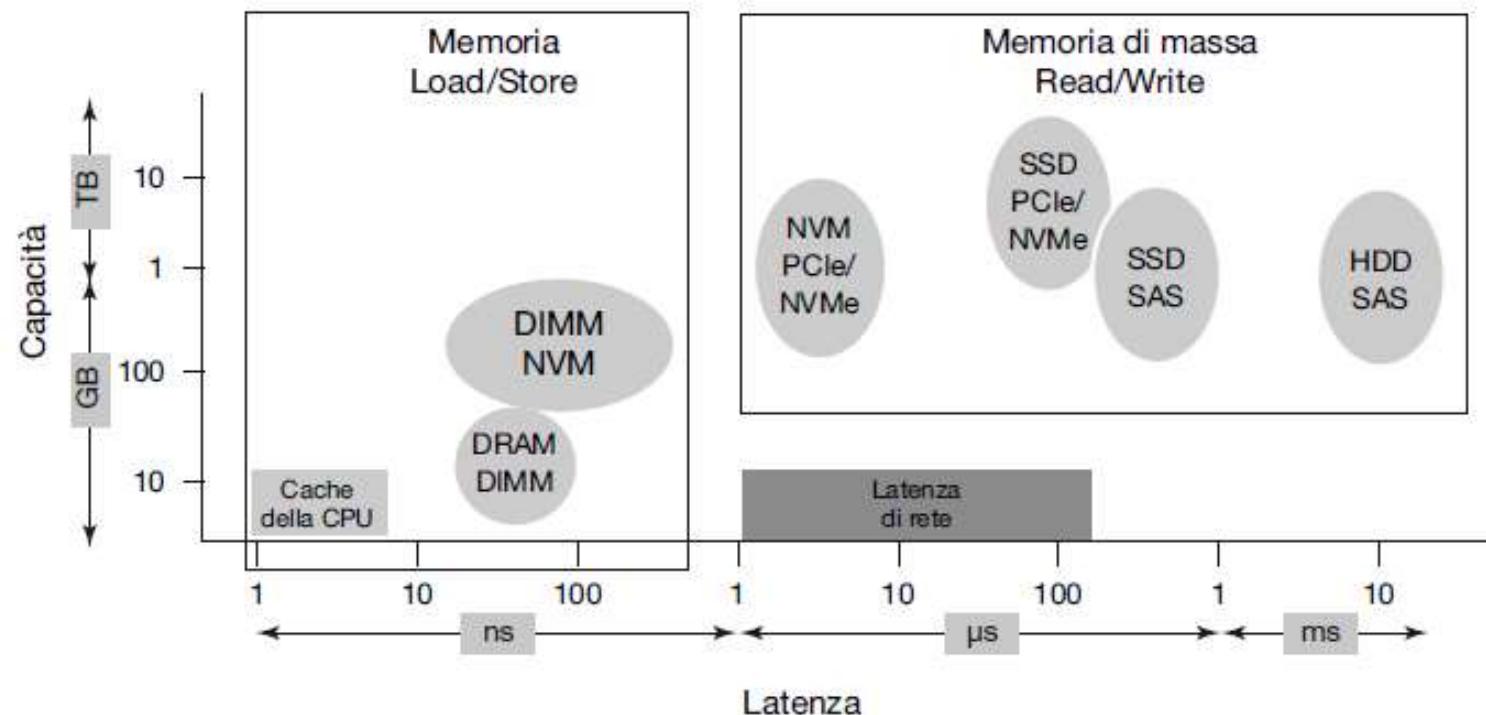


Figura 12.18 Prestazioni di I/O dei dispositivi di memorizzazione (e latenza di rete).

# The Linux System



# The Linux System

---

- Linux History
- Design Principles
- Kernel Modules
- Process Management
- Scheduling
- Memory Management
- File Systems
- Input and Output
- Interprocess Communication
- Network Structure
- Security

# Objectives

---

- Principles of Linux's design
- Linux process model
- Memory management in Linux
- Linux file systems and I/O devices

# History

---

- Linux is a modern, free operating system based on UNIX standards
  - developed as a small but self-contained kernel in 1991 by Linus Torvalds
  - major design goal of UNIX compatibility
  - released as open source
  - Designed to run efficiently and reliably on common PC hardware, but also runs on a variety of other platforms
- The core Linux operating system **kernel** is entirely original, but it can run much existing free UNIX software, resulting in an entire UNIX-compatible operating system free from proprietary code
- **Linux system** has many **Linux distributions** including the kernel, applications, and management tools

# Linux Kernel

---

- Version 0.01 (May 1991)
  - no networking, only 80386-compatible Intel processors and on PC hardware
  - extremely limited device-drive support
  - supported only the Minix file system
- Linux 1.0 (March 1994) included new features:
  - Support for UNIX' s standard TCP/IP networking protocols
  - BSD-compatible socket interface for networking programming
  - Device-driver support for running IP over an Ethernet
  - Enhanced file system
  - Support for a range of SCSI controllers for high-performance disk access
  - Extra hardware support
- Version 1.2 (March 1995) was the final PC-only Linux kernel
- Kernels with odd version numbers are **development kernels**, those with even numbers are **production kernels**

# Linux 2.0

---

- Released in June 1996, 2.0 added two major new capabilities:
  - Support for multiple architectures (including 64-bit)
  - Support for multiprocessor architectures
- Other new features included:
  - Improved memory-management code
  - Improved TCP/IP performance
  - Support for internal kernel threads, for handling dependencies between loadable modules, and for automatic loading of modules on demand
  - Standardized configuration interface
  - Available for Motorola 68000-series processors, Sun Sparc systems, and for PC and PowerMac systems
- 2.4 and 2.6 increased SMP support, added journaling file system, preemptive kernel, 64-bit memory support
- 3.0 released in 2011, 20<sup>th</sup> anniversary of Linux, improved virtualization support, new page write-back facility, improved memory management, new Completely Fair Scheduler

# Linux System

---

- Many tools developed as part of
  - Berkeley's BSD OS, MIT X Window System, and the Free Software Foundation's GNU project
- The main **system libraries** were started by the **GNU project**, with improvements provided by the Linux community
- Linux system maintained by developers collaborating over the Internet

# Linux Distributions

---

- Standard, precompiled sets of packages, or **distributions**, include:
  - basic Linux system
  - system installation and management utilities
  - ready-to-install packages of common UNIX tools
- Modern distributions include advanced package management

# Linux Licensing

---

- The Linux kernel is distributed under the **GNU General Public License** (GPL), the terms of which are set out by the Free Software Foundation
  - Not **public domain**, in that not all rights are waived
- Anyone using Linux, or creating derivative of Linux, may not make the derived product proprietary
  - Software released under GPL may not be redistributed as a binary-only product
  - Can sell distributions, but must offer the source code too

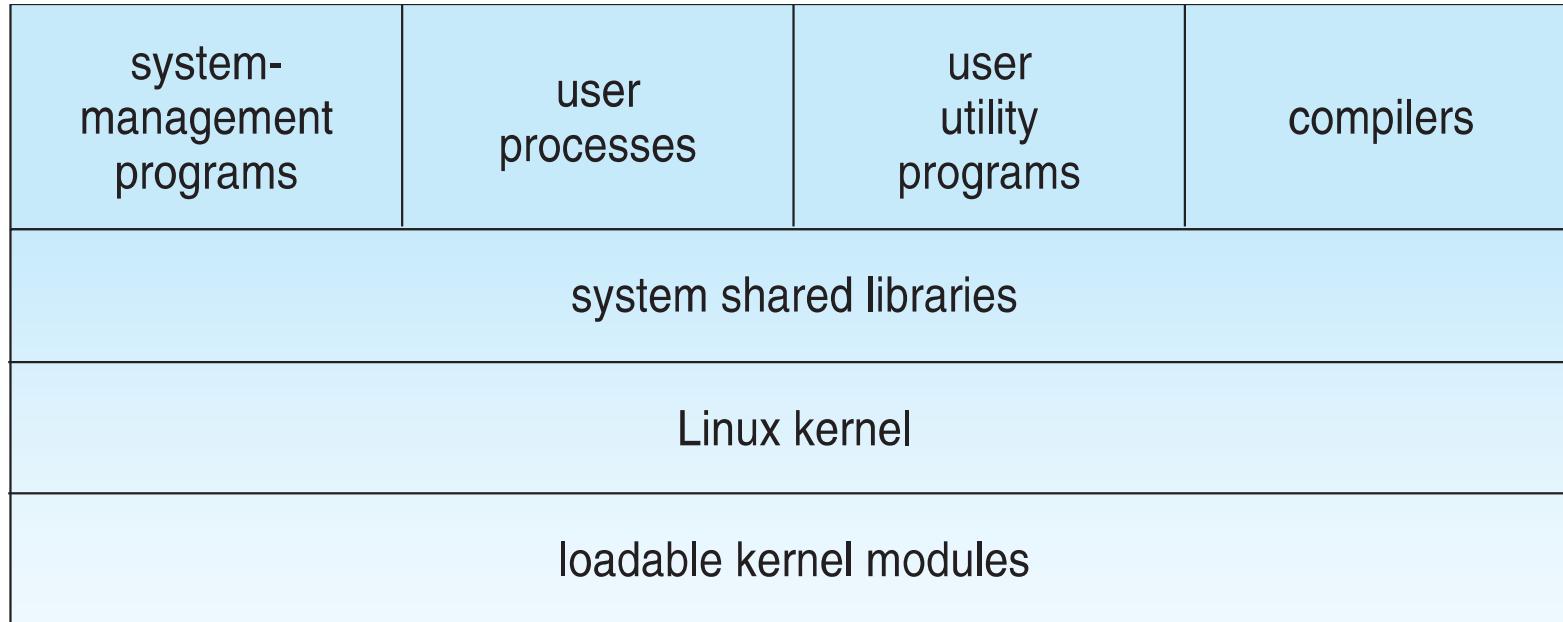
# Design Principles

---

- Multiuser, multitasking system with a full set of UNIX-compatible tools
- File System adheres to traditional UNIX semantics, and it fully implements the standard UNIX networking model
- Main design goals are:
  - speed, efficiency, and standardization
- Linux is designed to be compliant with the relevant **POSIX** documents
  - supports pthreads and a subset of POSIX real-time process control

# Components of a Linux System

---



# Components of a Linux System

---

- Linux is composed of three main bodies of code
- Most important distinction is between the **kernel** and all other components
- The **kernel** is responsible for maintaining the key abstractions of the OS
  - Kernel code executes in *kernel mode* with full access to all the physical resources of the computer
  - All kernel code and data structures are kept in the same single address space

# Components of a Linux System

---

- **System libraries**

- standard set of functions through which applications interact with the kernel
  - implement much of the OS functionality that does not need the kernel privileges

- **System utilities**

- individual specialized management tasks

- User-mode programs

- including multiple **shells** like the **bourne-again (bash)**

# Kernel Modules

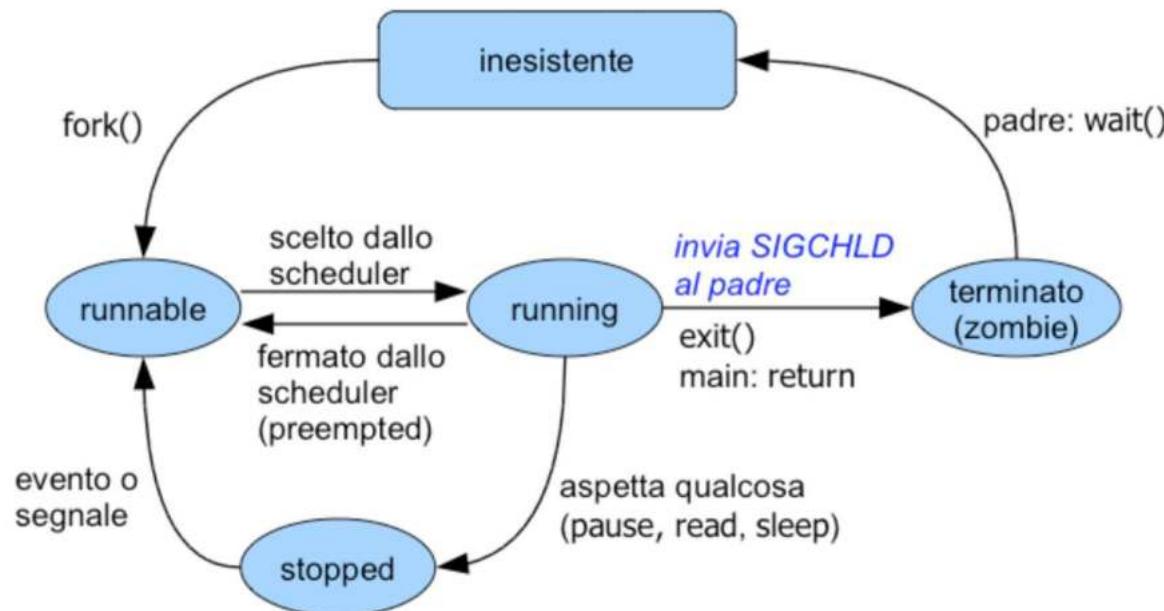
---

- Sections of kernel code that can be compiled, loaded, and unloaded independent of the rest of the kernel
  - Kernel not recompiled
  - Kernel modules typically implement
    - ▶ device driver, a file system, or a networking protocol
  - The module interface allows third parties to write and distribute device drivers or file systems
  - Kernel modules permits a Linux system with a standard, minimal kernel
- Four components to Linux module support:
  - ▶ **module-management system:** to manage module loading and communication
  - ▶ **module loader and unloader:** to load/unload a module into memory
  - ▶ **driver-registration system:** to tell the rest of the kernel that a new driver is available
  - ▶ **conflict-resolution mechanism:** allows device drivers to reserve resources

# Process Management

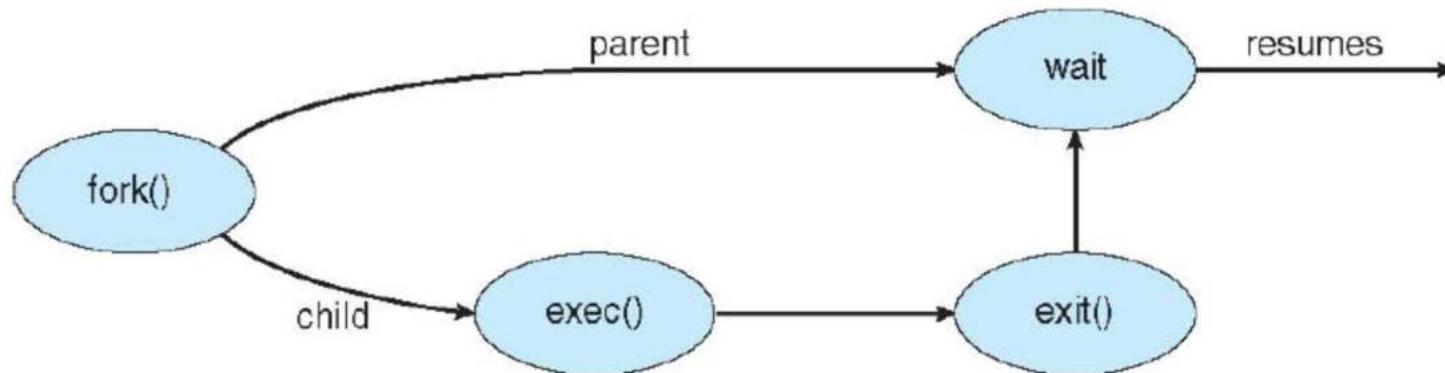
- UNIX process management separates the creation of processes and the running of a new program into two distinct operations.
  - The `fork()` system call creates a new process
  - A new program is run after a call to `exec()`

## Ciclo di vita del processo



# Process Management

- UNIX process management separates the creation of processes and the running of a new program into two distinct operations.
  - The `fork()` system call creates a new process
  - A new program is run after a call to `exec()`



# Process Management

---

- UNIX process management separates the creation of processes and the running of a new program into two distinct operations.
  - The `fork()` system call creates a new process
  - A new program is run after a call to `exec()`
- This model has the advantage of simplicity
  - Not necessary to specify details of the environment of a new program
  - The new program simply runs in its existing environment
- Under Linux, process properties fall into three groups:
  - process's identity, environment, and context

# Process Identity

---

- **Process ID (PID) –**
  - The unique identifier for the process
- **Credentials -**
  - Each process has a user ID and one or more group IDs that determine the process' s rights to access system resources and files
- **Personality -**
  - Not traditionally on UNIX systems, under Linux each process has a personality identifier that can slightly modify the semantics of certain system calls
- **Namespace –** Specific view of file system hierarchy
  - Most processes share common namespace and operate on a shared file-system hierarchy
  - But each can have unique file-system hierarchy with its own root directory and set of mounted file systems

# Process Environment

---

- The process's environment is inherited from its parent; it is composed of two vectors:
  - The **argument vector** lists the command-line arguments used to invoke the running program
  - The **environment vector** is a list of “NAME=VALUE” pairs that associates named environment variables with arbitrary textual values
- Passing environment variables among processes and inheriting variables by a process's children are flexible means of passing data
  - The environment is not held in kernel memory but is stored in the process's own user-mode address space as the first datum at the top of the process's stack
  - The environment-variable mechanism provides a customization of the OS that can be set on a per-process basis

# Process Environment

---

## Ambiente di un processo

- L'ambiente di un processo è un insieme di stringhe (terminate da \0).
- Un ambiente è rappresentato da un vettore di puntatori a caratteri terminato da un puntatore nullo.
- Ogni puntatore (che non sia quello nullo) punta ad una stringa della forma: identificatore = valore
- Per accedere all'ambiente da un programma C, è sufficiente aggiungere il parametro `envp` a quelli del main:

```
/* showmyenv */
#include <stdio.h>
main(int argc, char **argv, char **envp)
{
 while(*envp)
 printf("%s\n", *envp++);
}
```

- oppure usare la variabile globale seguente:  
`extern char **environ;`

# Process Environment

---

## L'ambiente di un processo

- L'ambiente di default di un processo coincide con quello del processo padre.
- Per specificare un nuovo ambiente è necessario usare una delle due varianti seguenti della famiglia `exec`, memorizzando in `envp` l'ambiente desiderato:

```
execle(path, arg0, arg1, ..., argn, (char *)0, envp);
execve(path, argv, envp);
```

# Process Environment

---

## L'ambiente di un processo

```
/* setmyenv */
#include <unistd.h>
#include <stdio.h>
main()
{ char *argv[2], *envp[3];
 argv[0] = "setmyenv";
 argv[1] = (char *)0;
 envp[0] = "var1=valore1";
 envp[1] = "var2=valore2";
 envp[2] = (char *)0;
 execve("./showmyenv", argv, envp);
 perror("execve fallita");
}
```

- **Eseguendo il programma precedente si ottiene quanto segue:**

```
$./setmyenv
var1=valore1
var2=valore2
```

# Process Environment

---

## L'ambiente di un processo

- Esiste una funzione della libreria standard che consente di cercare in environ il valore corrispondente ad una specifica variabile d'ambiente:

```
#include <stdlib.h>
char *getenv(const char *name);
```

- getenv prende come argomento il nome della variabile da cercare e restituisce il puntatore al valore (ciò che sta a destra del simbolo =) o NULL se non lo trova:

```
#include <stdio.h>
#include <stdlib.h>
main()
{
 printf("PATH=%s\n", getenv("PATH"));
}
```

- Dualmente, putenv consente di modificare o estendere l'ambiente:  
`putenv("variabile=valore");`

# Process Environment

## Esempio 7

```
/* exec.c: Dimostra il funzionamento di due funzioni exec */
#include <sys/types.h> /* per il tipo pid_t */
#include <unistd.h> /* per le funzioni fork execle ed execlp */
#include <sys/wait.h> /* per la funzione waitpid */
#include <stdio.h> /* per la funz. printf */
#include <stdlib.h> /* per la funzione exit */

char *ambiente[] = { "USER = nobody", "PATH=/tmp", NULL };

int main(void)
{
 pid_t pid;
 Esecuzione del programma echoall con
 specifica del pathname e dell'ambiente
 if ((pid = fork()) < 0) printf("errore fork"), exit(1);

 else if (pid == 0) { /* 1mo figlio */
 if (execle("/home/gio/echoall", "echoall", "1mo arg",
 "2ndo e ultimo", NULL, ambiente) < 0)
 printf("errore di execle"), _exit(1);
 }
}
```

G Schmid-Processi Unix →

# Process Environment

## Esempio 7 (cont.)

```
/* pid >0, genitore */
if ((pid = waitpid(pid, NULL, 0)) < 0)
 printf("errore di waitpid"), exit(1);

if ((pid = fork()) < 0)
 printf("errore di fork"), exit(1);

else if (pid == 0) { /* 2ndo figlio */
 if (execlp("echoall", "echoall", "1mo e ultimo", NULL) < 0)
 printf("errore di execlp"), _exit(1);
}

/* pid >0, genitore */
if (wait(NULL) != pid)
 printf("errore di wait"), exit(1);
exit(0);
}
```



Esecuzione del programma echoall senza specifica del pathname e dell'ambiente

# Process Environment

## Esempio 7 (cont.)

```
/* echoall.c: Il programma eseguito da exec.c */

#include <stdlib.h> /* per la funzione exit */
#include <stdio.h> /* per la funzione printf */

void main(int argc, char *argv[])
{
 int i;
 char **ptr;
 extern char **environ;

 for (i=0; i<argc; i++) /* effettua l'eco di tutti gli argomenti dalla
 linea di comando... */
 printf("argv[%d] : %s\n", i , argv[i]);

 for (ptr=environ; *ptr != 0; ptr++)
 /* e di tutte le stringhe
 d'ambiente */
 printf("%s\n", *ptr);

 exit(0);
}
```

# Process Context

---

- The (constantly changing) state of a running program at any point in time
- The **scheduling context**
  - information that the scheduler needs to suspend and restart the process
  - Process state
  - scheduling priority
  - signals waiting to be delivered to the process
  - process's kernel stack, a separate area of kernel memory reserved for use by kernel-mode code
- The kernel maintains **accounting** information about the resources consumed by each process
- The **file table** is an array of pointers to kernel file structures
  - When making file I/O system calls, processes refer to files by their index into this table, the **file descriptor (fd)**

# Process Context

---

- Whereas the file table lists the existing open files, the **file-system context** applies to requests to open new files
  - The current root and default directories to be used for new file searches are stored here
- The **signal-handler table** defines the routine in the process' s address space to be called when specific signals arrive
- The **virtual-memory context** of a process describes the full contents of the its private address space

# Process Control Block

---

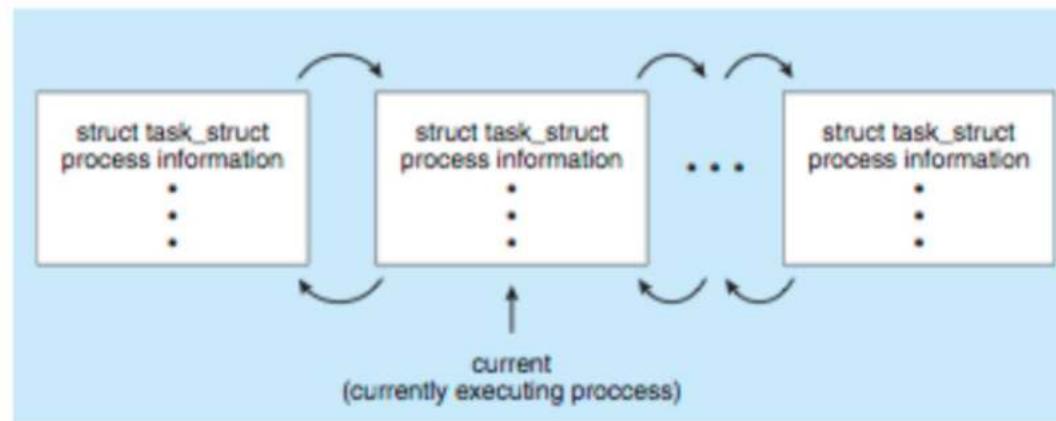
- In the kernel, each process is associated with a process control block
  - process number (pid)
  - process state
  - program counter
  - CPU registers
  - CPU scheduling information
  - memory-management data
  - accounting data
  - I/O status
- Linux's PCB is defined in struct `task_struct`:  
<http://lxr.linux.no/linux+v3.2.35/include/linux/sched.h#L1221>

# Process Control Block

- Represented by the C structure **task\_struct**

```
pid_t pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice; /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process*/

...
```



# Processes and Threads

---

- Linux uses the same internal representation for processes and threads
  - a thread is a new process that shares the same address space as its parent
  - Both are called **tasks** by Linux
- Distinction when a new thread is created by the `clone()` system call
  - `fork()` creates a new task with its own entirely new task context
  - `clone()` creates a new task with its own identity, but that is allowed to share the data structures of its parent
  - Using `clone()` gives an application fine-grained control over exactly what is shared between two threads
  - If none of these flags is set when `clone()` is invoked, the associated resources are not shared, resulting in functionality similar to that of the `fork()` system call

| flag                       | meaning                            |
|----------------------------|------------------------------------|
| <code>CLONE_FS</code>      | File-system information is shared. |
| <code>CLONE_VM</code>      | The same memory space is shared.   |
| <code>CLONE_SIGHAND</code> | Signal handlers are shared.        |
| <code>CLONE_FILES</code>   | The set of open files is shared.   |

# Processes and Threads

---

- Linux uses the same internal representation for processes and threads
  - a thread is a new process that shares the same address space as its parent
  - Both are called **tasks** by Linux
- Distinction when a new thread is created by the `clone()` system call

```
#include <sched.h>
int clone (int (*fn) (void *), void *child_stack, int flags, void *arg);
```

# Scheduling

---

- The job of allocating CPU time to different tasks within an OS
  - As in UNIX systems, Linux scheduling supports preemptive multitasking
- As of 2.5, scheduling algorithm – preemptive, priority-based, known as O(1)
  - can schedule processes within a constant amount of time (regardless of how many processes are running)
  - Improvement O(n) scheduler, which schedule processes in an amount of time that scales linearly based on the amounts of processes
  - Real-time range
  - Nice value
    - Had problems with interactive performance
- 2.6 introduced **Completely Fair Scheduler (CFS)**

# Scheduling Linux prima della versione 2.5

---

- Prima della versione kernel 2.5 variazione dell'algoritmo di scheduling standard di UNIX
- Code multiple con feedback gestite con RR
  - Prelazione sulle code con priorità
  - Priorità assegnata sulla base dell'utilizzo di CPU (maggiore utilizzo, minore priorità)
  - Periodicamente il kernel calcola la CPU utilizzata da un processo dall'ultimo controllo, valore inversamente proporzionale alla priorità
  - Selezione richiede scansione dei processi in attesa:  $O(n)$
- Non scalabile con molti processi
  - Problema osservato con JVM
- Metodo non flessibile e poco adatto a Symmetric Multiprocessing (SMP)
  - Singola coda con multipli core

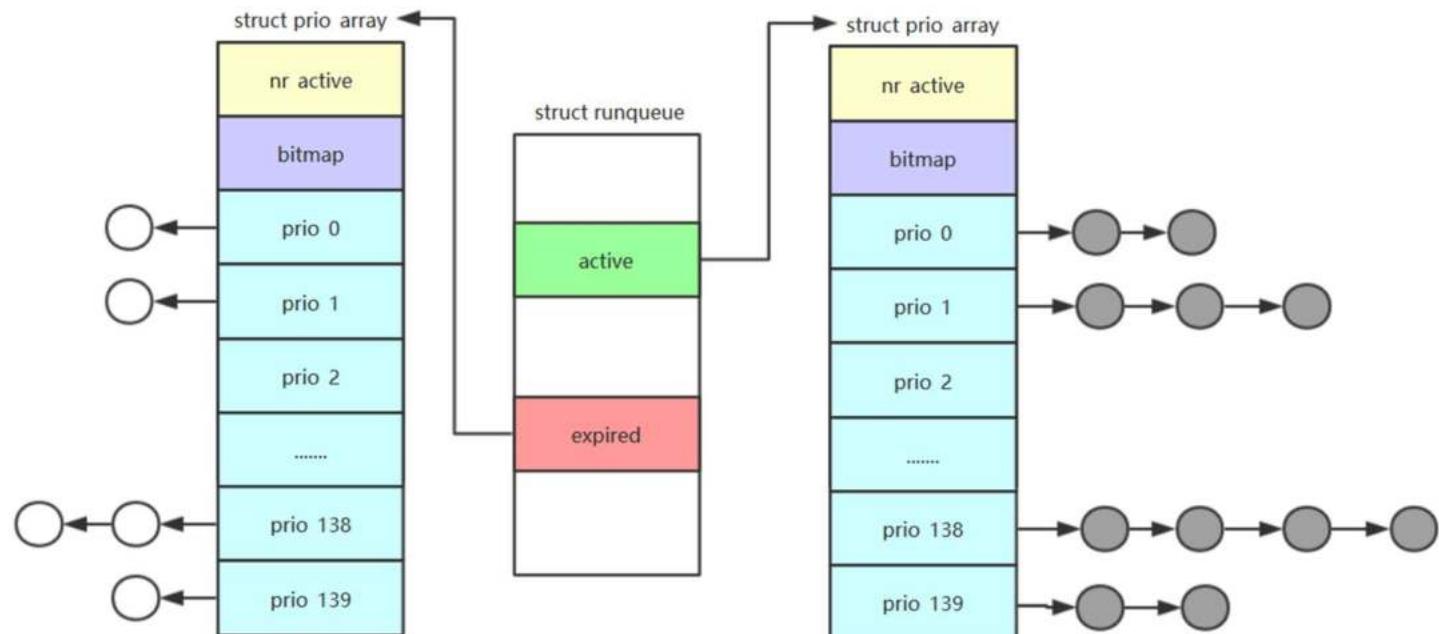
# Scheduling Linux versione 2.5

---

- La versione 2.5 propone uno scheduler a **tempo costante  $O(1)$**  indipendente dal numero di task nel sistema:
  - Preemptive e priority based
  - Due intervalli di priorità: processi time-sharing (detti **nice**) e real-time
  - **Real-time** da 0 a 99 e valori **nice** da 100 a 140
  - Priorità più alte prendono time-slice q maggiori
  - I task sono in esecuzione finché hanno il time slice (**active**)
  - Quando scade non vanno in esecuzione finché tutti gli altri tasks non usano lo slice

# Scheduling Linux versione 2.5

- La versione 2.5 propone uno scheduler a **tempo costante  $O(1)$**  indipendente dal numero di task nel sistema:
  - Preemptive e priority based
  - Due intervalli di priorità: processi time-sharing (detti **nice**) e real-time
  - **Real-time** da 0 a 99 e valori **nice** da 100 a 140
  - Priorità più alte prendono time-slice q maggiori
  - I task sono in esecuzione finché hanno il time slice (**active**)
  - Quando scade non vanno in esecuzione finché tutti gli altri tasks non usano lo slice



# Scheduling Linux versione 2.5

---

- La versione 2.5 propone uno scheduler a **tempo costante  $O(1)$**  indipendente dal numero di task nel sistema:
  - Preemptive e priority based
  - Due intervalli di priorità: processi time-sharing (detti **nice**) e real-time
  - **Real-time** da 0 a 99 e valori **nice** da 100 a 140
  - Priorità più alte prendono time-slice q maggiori
  - I task sono in esecuzione finché hanno il time slice (**active**)
  - Quando scade non vanno in esecuzione finché tutti gli altri tasks non usano lo slice
  - Tutti i task eseguibili tracciati con code per-CPU
- Buon funzionamento, ma tempi di risposta non soddisfacenti per processi in time-sharing
  - ▶ Dipendenza tra timeslice e priority
  - ▶ Priority e timeslice non uniformi
  - ▶ Complicata euristica per distinguere tra processi CPU e I/O bound

# Completely Fair Scheduler (CFS)

---

- Eliminates traditional, common idea of fixed time slice
  - Instead, all tasks allocated portion of processor's time
- CFS calculates how long a process should run as a function of total number of tasks
  - $N$  runnable tasks means each gets  $1/N$  of processor's time
  - Then weights each task with its nice value
    - ▶ Smaller nice value -> higher weight (higher priority)

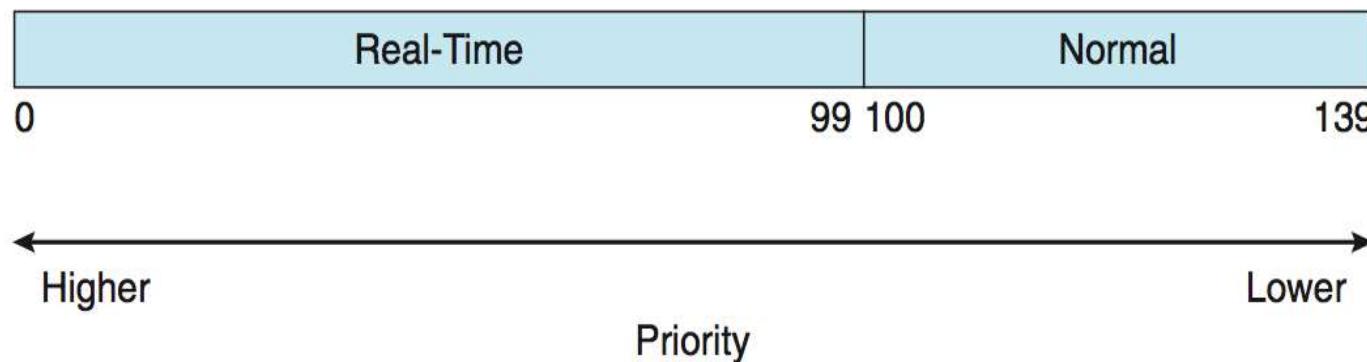
# Linux Scheduling in Versione 2.6.23 +

---

- **Completely Fair Scheduler** (CFS)
- **Diverse classi di scheduling**
  - Ognuna con priorità specifica
  - Scheduler prende la più alta nella classe più alta
  - Non quantum based con allocazione fissa, ma basata su proporzione di CPU time
  - 2 classi di scheduling, altre possono essere aggiunte
    1. default
    2. real-time
- Quantum calcolato sulla base di un **nice value** da -20 a +19
  - Più basso, più alta la priorità
  - Calcolo della **latenza target** – intervallo di tempo durante il quale un task deve andare in run almeno una volta
  - La latenza target può aumentare con il numero dei task attivi
- CFS scheduler mantiene un **tempo di esecuzione virtuale** per ogni task in variabile **vruntime** (per quanto è stato eseguito)
  - Associato a fattore di decadimento basato sulla priorità del task
    - ▶ a basse priorità corrisponde alto fattore di decadimento
    - ▶ Default: virtual run time = actual run time
- Per decidere il prossimo task lo scheduler prende il task con il minore virtual run time

# Real-Time Scheduler

- Real-time scheduling in POSIX.1b
  - Task real-time con priorità statiche
- Real-time hanno valori più bassi di quelli dei task normali e sono mappati in uno schema di priorità globale



- Le priorità relative dei processi real-time sono assicurate
- Il kernel non fornisce garanzia sui tempi di attesa dei processi pronti
- Se interruzione per real-time arriva mentre il kernel serve una chiamata di sistema, il processo real-time attende

# Real-Time Scheduler

---

- Real-time scheduling in POSIX.1b
  - fixed-priority preemptive scheduling policy
- SCHED\_FIFO: This is a fixed-priority preemptive scheduling policy, in which processes with the same priority are treated in first-in-first-out (FIFO) order. At least 32 priority levels must be available for this policy.
- SCHED\_RR: This policy is similar to SCHED\_FIFO, but uses a time-sliced (round robin) method to schedule processes with the same priorities. It also has 32 priority levels.
- SCHED\_OTHER: It is an implementation-defined scheduling policy.

# Real-Time Scheduler

---

- Simpler than the fair scheduling employed for standard time-sharing threads
- Linux implements the two real-time scheduling classes:
  - First-Come First Served (FCFS)
  - Round-Robin
- Each thread has a priority in addition to its scheduling class
- The scheduler always runs the thread with the highest priority
- Among threads of equal priority, it runs the thread that has been waiting longest
- FCFS threads continue to run until they either exit or block, whereas a round-robin thread will be preempted after a while and will be moved to the end of the scheduling queue
- Linux's real-time scheduling is soft—rather than hard—real time.
  - ▶ The scheduler offers strict guarantees about the relative priorities of real-time threads
  - ▶ The kernel does not offer any guarantees about how quickly a real-time thread will be scheduled once that thread becomes runnable

# Kernel Synchronization

---

- The way the kernel schedules its own operations is different from the way it schedules threads
- A request for kernel-mode execution can occur in two ways:
  - A running program may request an OS service
    - either explicitly via a system call, or implicitly, for example, when a page fault occurs
  - A device driver may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt
- The problem for the kernel is that all these tasks may try to access the same internal data structures
  - kernel synchronization involves more than thread scheduling
  - Kernel tasks should run without violating the integrity of shared data

# Kernel Synchronization

---

- Linux uses two techniques to protect critical sections:
  1. Normal kernel code is **nonpreemptible** (until 2.6)
    - when a time interrupt is received while a process is executing a kernel system service routine, the kernel's **need\_resched** flag is set so that the scheduler will run once the system call has completed and control is about to be returned to user mode
  2. The second technique applies to **critical sections** that occur in an interrupt service routines
    - By using the processor's interrupt control hardware to disable interrupts during a critical section, the kernel guarantees that it can proceed without the risk of concurrent access of shared data structures
- Provides spin locks, semaphores, and reader-writer versions of both
  - ▶ Behavior modified if on single processor or multi:

| single processor           | multiple processors |
|----------------------------|---------------------|
| Disable kernel preemption. | Acquire spin lock.  |
| Enable kernel preemption.  | Release spin lock.  |

# Symmetric Multiprocessing

---

- Linux 2.0 was the first Linux kernel to support **SMP** hardware
  - separate processes or threads can execute in parallel on separate processors
- Until version 2.2, to preserve the kernel's **nonpreemptible synchronization** requirements, SMP imposes the restriction
  - single kernel spinlock ("big kernel lock"), that only one processor at a time may execute kernel-mode code
- Later releases implement more scalability by splitting single spinlock into **multiple locks**, each protecting a small subset of kernel data structures
- Version 3.0 adds even **more fine-grained locking**, processor affinity, and load-balancing

# Memory Management

---

- Linux's physical memory-management system has two components:
  - **Physical memory**
    - allocating and freeing memory pages, groups of pages, and small blocks of RAM
  - **Virtual memory**
    - handling virtual memory, memory mapped into the address space of running processes

## □ Physical memory

- Splits memory into different **zones** due to hardware characteristics
  - Architecture specific, for example on x86

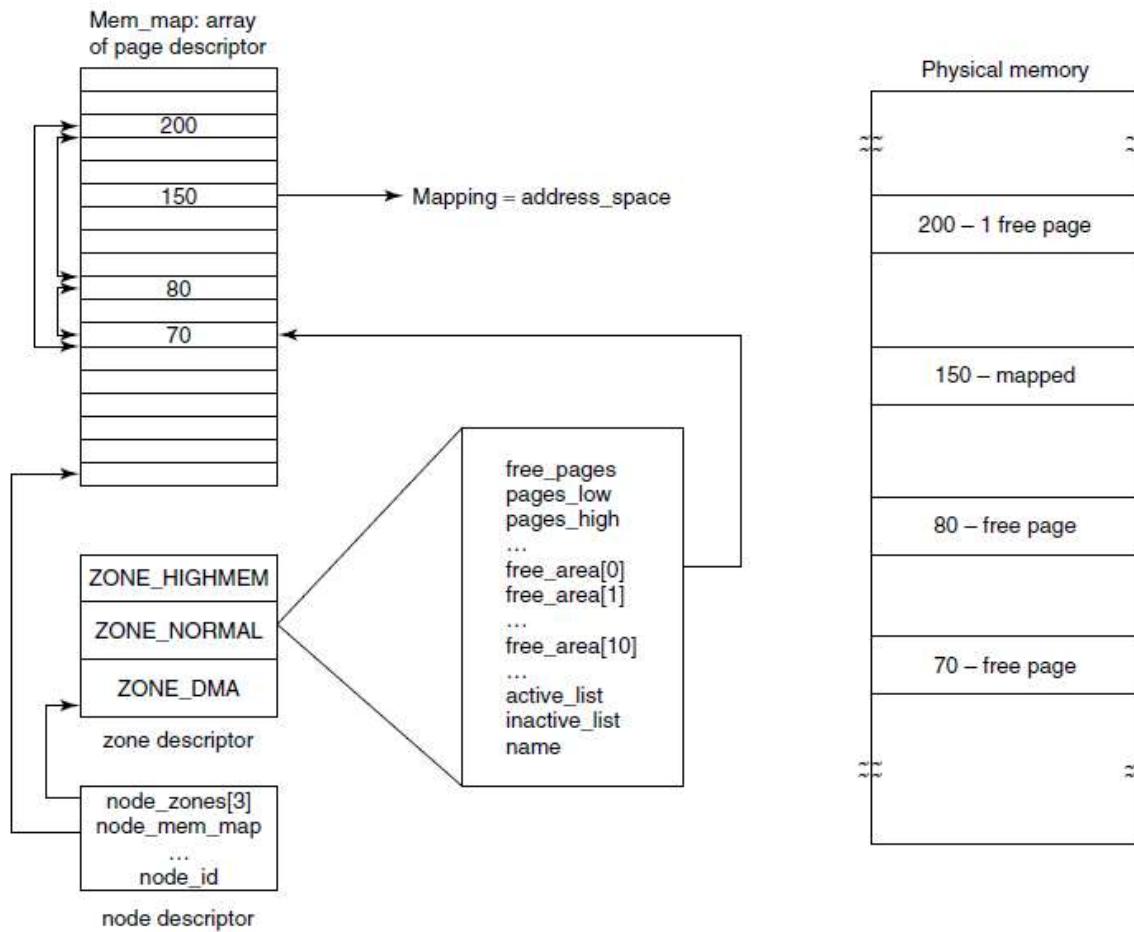
| zone         | physical memory |
|--------------|-----------------|
| ZONE_DMA     | < 16 MB         |
| ZONE_NORMAL  | 16 .. 896 MB    |
| ZONE_HIGHMEM | > 896 MB        |

- A modern, 64-bit architecture has a small 16-MB ZONE DMA (for legacy devices) and all the rest of its memory in ZONE NORMAL, with no "high memory"
- When a request physical memory arrives, kernel satisfies it with the appropriate zone

# Memory Management

## □ Physical memory

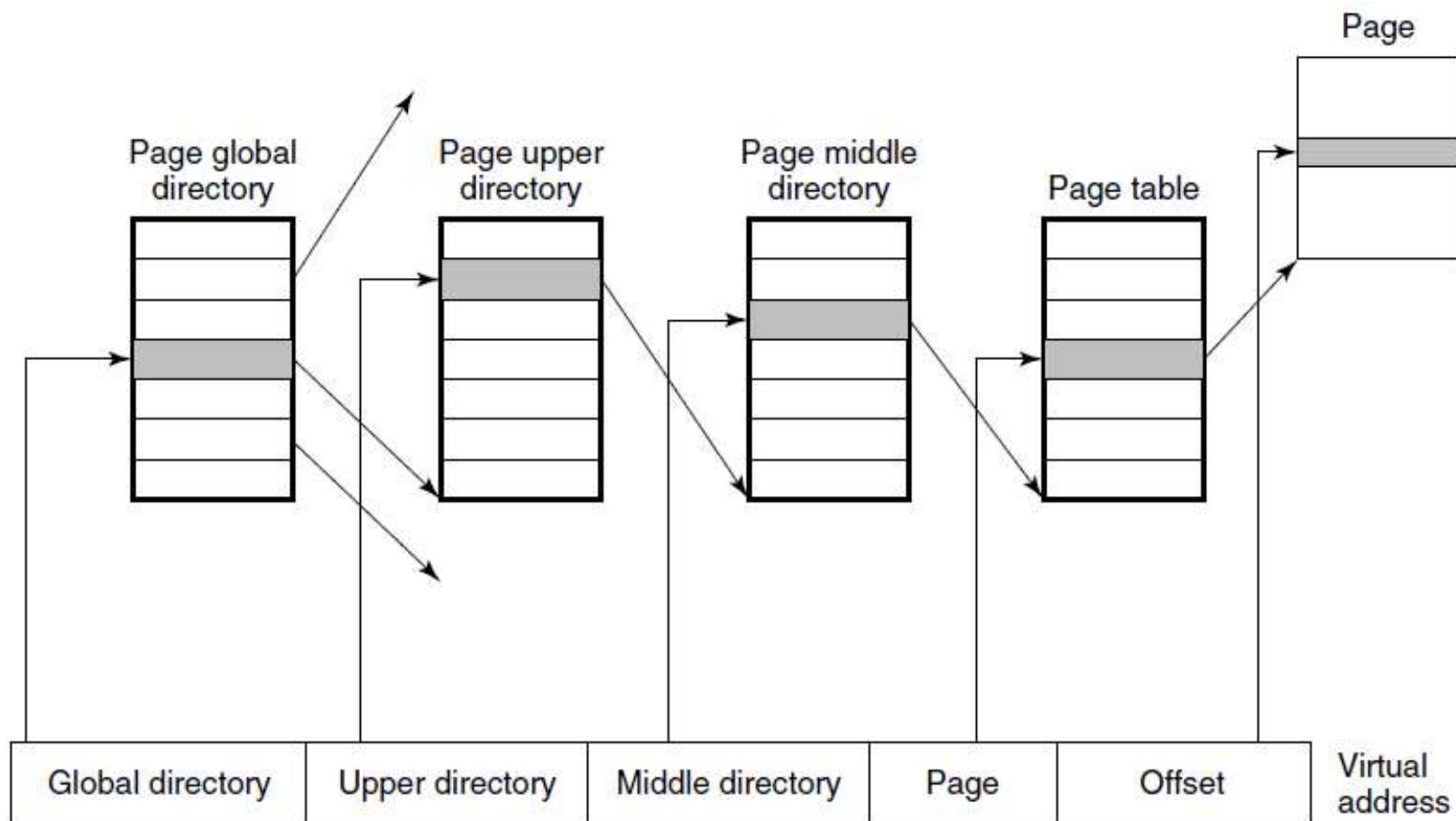
- The kernel maintains a map of the main memory which contains all information about the use of the physical memory in the system, such as its zones
  - ▶ Each page descriptor contains a pointer to the address space (links for free pages)



# Memory Management

## □ Physical memory

- In order for the paging mechanism to be efficient on both 32- and 64-bit architectures, Linux makes use of a four-level paging scheme



# Memory Management

---

## □ Physical memory

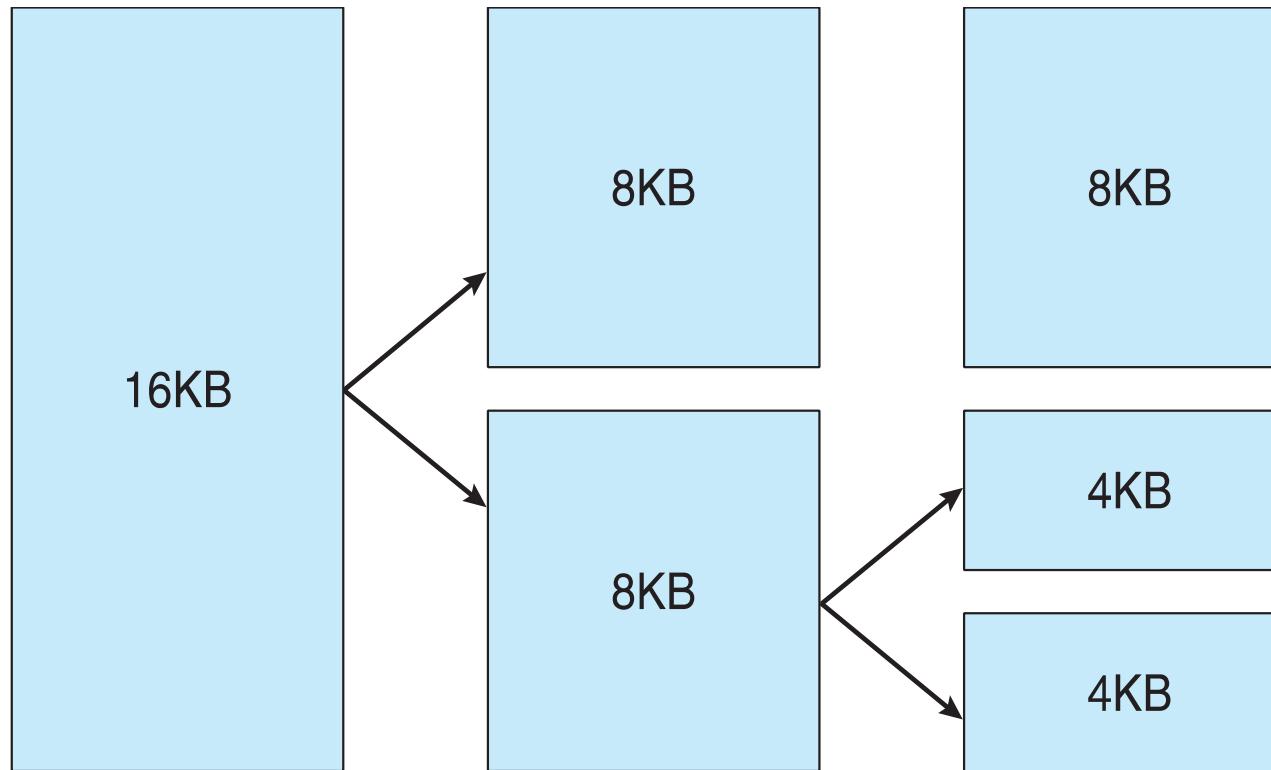
- The kernel itself is fully hardwired; no part of it is ever paged out.
- The rest of memory is available for user pages, the paging cache, and other purposes.

# Managing Physical Memory

---

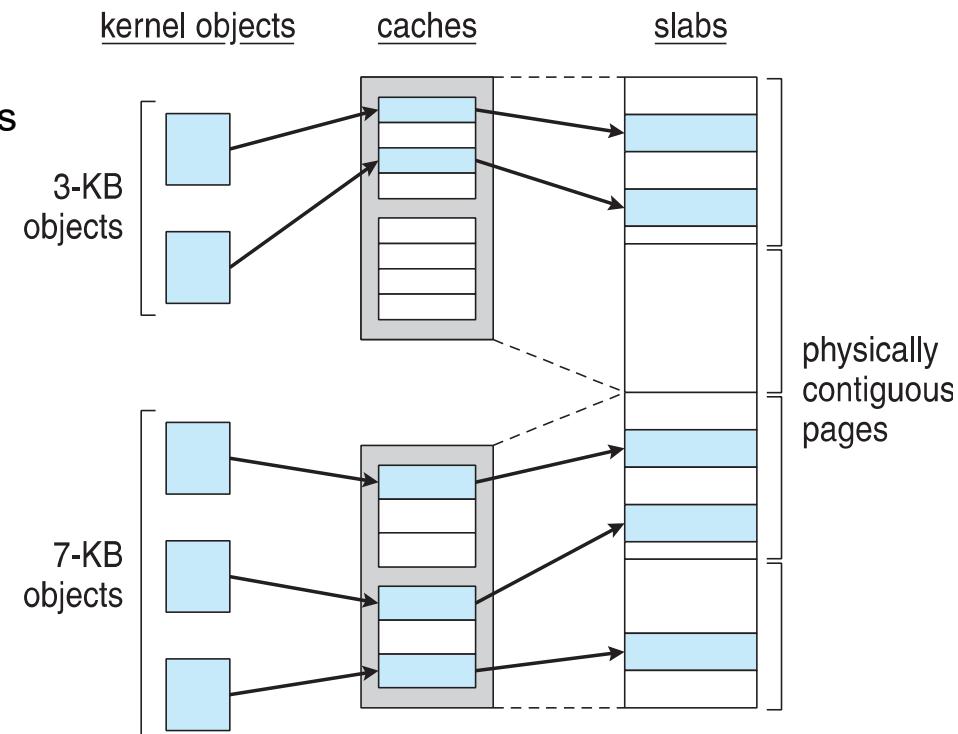
- **Page allocator**
  - Primary **physical-memory manager** in the Linux kernel
  - PA allocates and frees all physical pages
  - PA can allocate ranges of **physically-contiguous pages** on request
  - Each zone has its own allocator
- PA uses a **buddy-heap algorithm** to keep track of available physical pages
  - Each allocatable memory region is paired with an adjacent partner
  - When two allocated regions are freed up they are combined in a larger region
  - If a small memory request cannot be satisfied, then a larger free region will be subdivided into two partners regions
  - smallest size allocatable under this mechanism is a single physical page

# Splitting of Memory in a Buddy Heap



# Managing Physical Memory

- Buddy algorithm leads to considerable internal fragmentation
  - if you want a 65-page chunk, you have to ask for and get a 128-page chunk
- Another strategy adopted by Linux for allocating kernel memory is the **slab allocator**
  - **slab** is used for allocating memory for kernel data structures and is made up of one or more physically contiguous pages.
  - A **cache** consists of one or more slabs
  - Each cache is populated with **objects** that are instantiations of the kernel data structure
    - ▶ Es. cache representing inodes stores instances of inode structures, cache representing process descriptors stores instances of process descriptor structures



# Slab Allocator in Linux

---

- Esempio: per process descriptor è di tipo `struct task_struct`
- Appross. 1.7KB di memoria
- Nuovo task -> alloca new struct dalla cache
  - Userà `free struct task_struct`
- Slab può essere in tre possibili stati
  1. Full – tutto usato
  2. Empty – tutto libero
  3. Partial – mix di free e used
- Su richiesta, slab allocator
  1. Usa `free struct` in partial slab
  2. Se non c'è, prende uno degli empty slab
  3. Se non c'è empty slab, crea un nuovo empty slab

# Slab Allocator in Linux

---

- Slab iniziato in Solaris, ora diffuso sia per Kernel mode e memoria user in molti SO, Linux da 2.2 prima Buddy
- Linux 2.2 aveva SLAB, ora ha allocatori SLOB e SLUB
  - SLOB per sistemi con limitata memoria
    - Simple List Of Blocks – mantiene 3 liste di oggetti per piccoli, medi e grandi
    - Oggetti 256 byte, 1024 byte, più di una pagina
    - First fit-policy per l'allocazione
  - SLUB è uno SLAB ottimizzato,
    - Introdotto da 2.6.24 per sostituire SLAB
    - usa metadati memorizzati in una page structure, etc.

# Managing Physical Memory

---

- Two other main subsystems in Linux do their own management of physical pages:
  - **page cache** and **virtual memory system**
    - Page cache is kernel's main cache for files and main mechanism for I/O to block devices is performed
    - File systems of all types perform their I/O through the page cache
    - The page cache stores entire pages of file contents and is not limited to block devices (it can also cache networked data)
    - The virtual memory system manages the contents of process's virtual address space
  - Page cache and virtual memory system interact closely
    - Reading a page of data into the page cache requires mapping pages in the page cache using the virtual memory system

# Virtual Memory

---

- The Linux virtual memory system is responsible for maintaining the address space accessible to each process:
  - Creates pages of virtual memory on demand
  - Manages the loading of those pages from disk
  - Manages their swapping back out to disk as required
- The virtual memory manager maintains two separate views of a process address space: set of ***separate regions*** and as a ***set of pages***
  - **Logical view** describing the layout of the address space
    - the address space consists of a set of nonoverlapping regions, each region representing a continuous, page-aligned subset of the address space
    - Each region is described by a single ***vm\_area\_struct*** structure
      - properties of the region, e.g., process's read, write, and execute permissions in the region, any files associated with the region etc.
  - **Physical view** of each address space
    - Stored in the hardware ***page tables*** for the process
    - Page table entries identify the exact current location of each page of VM

# Virtual Memory

- Logical view describing the layout of the address space
  - the address space consists of a set of nonoverlapping regions, each region representing a continuous, page-aligned subset of the address space
  - Each region is described by a single ***vm\_area\_struct*** structure

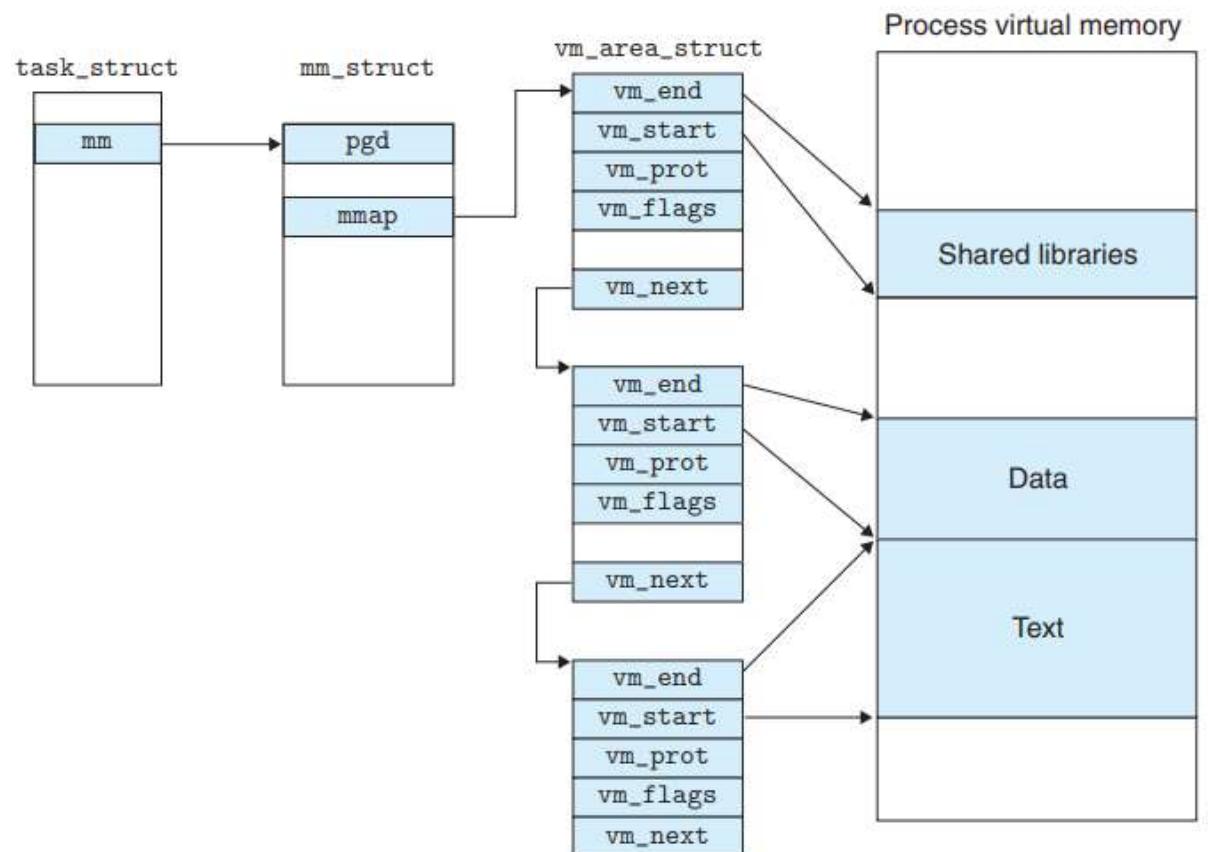
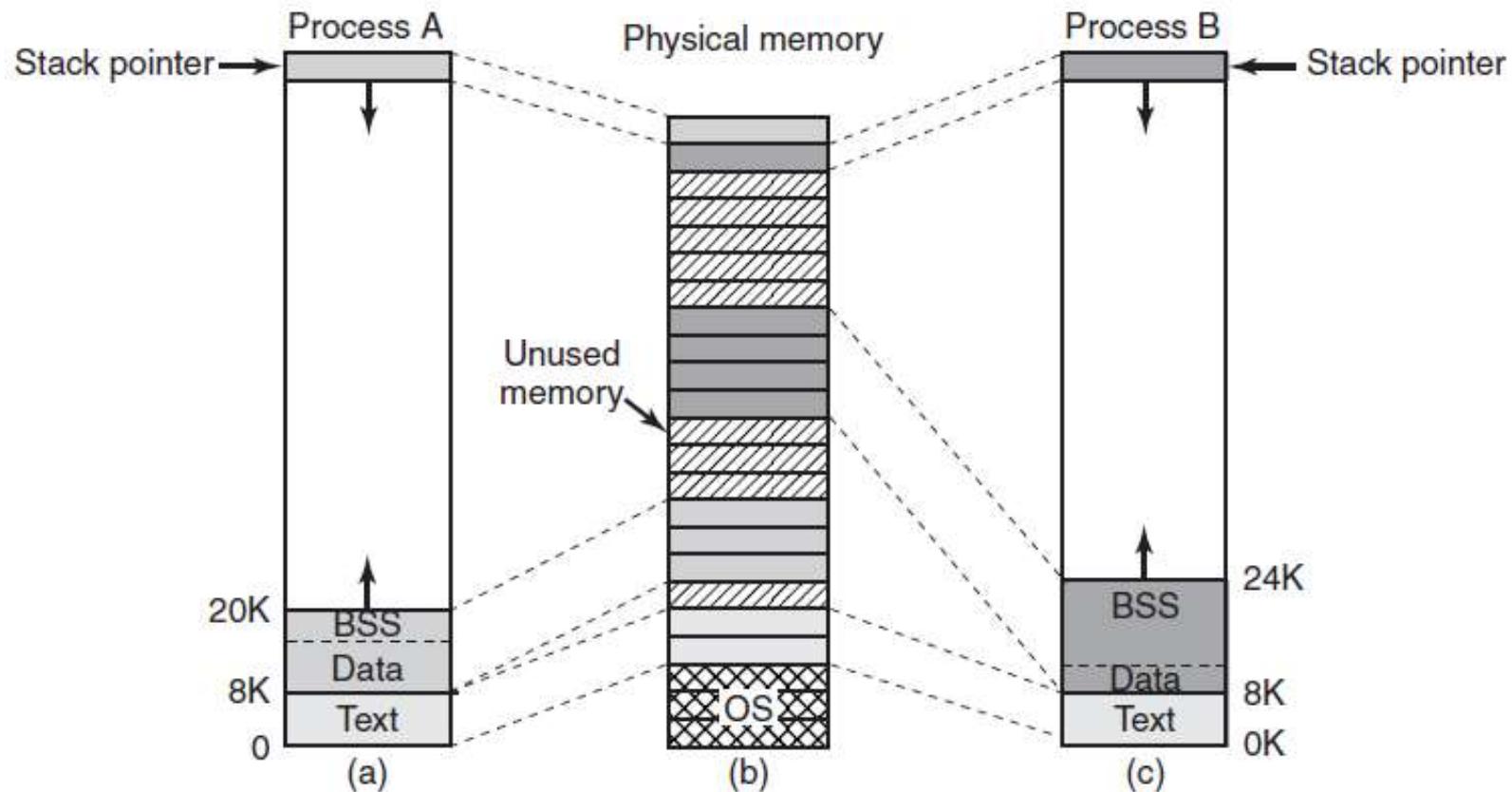


Figure 9.27 How Linux organizes virtual memory.

# Virtual Memory

- Logical view describing the layout of the address space
  - the address space consists of a set of nonoverlapping regions, each region representing a continuous, page-aligned subset of the address space

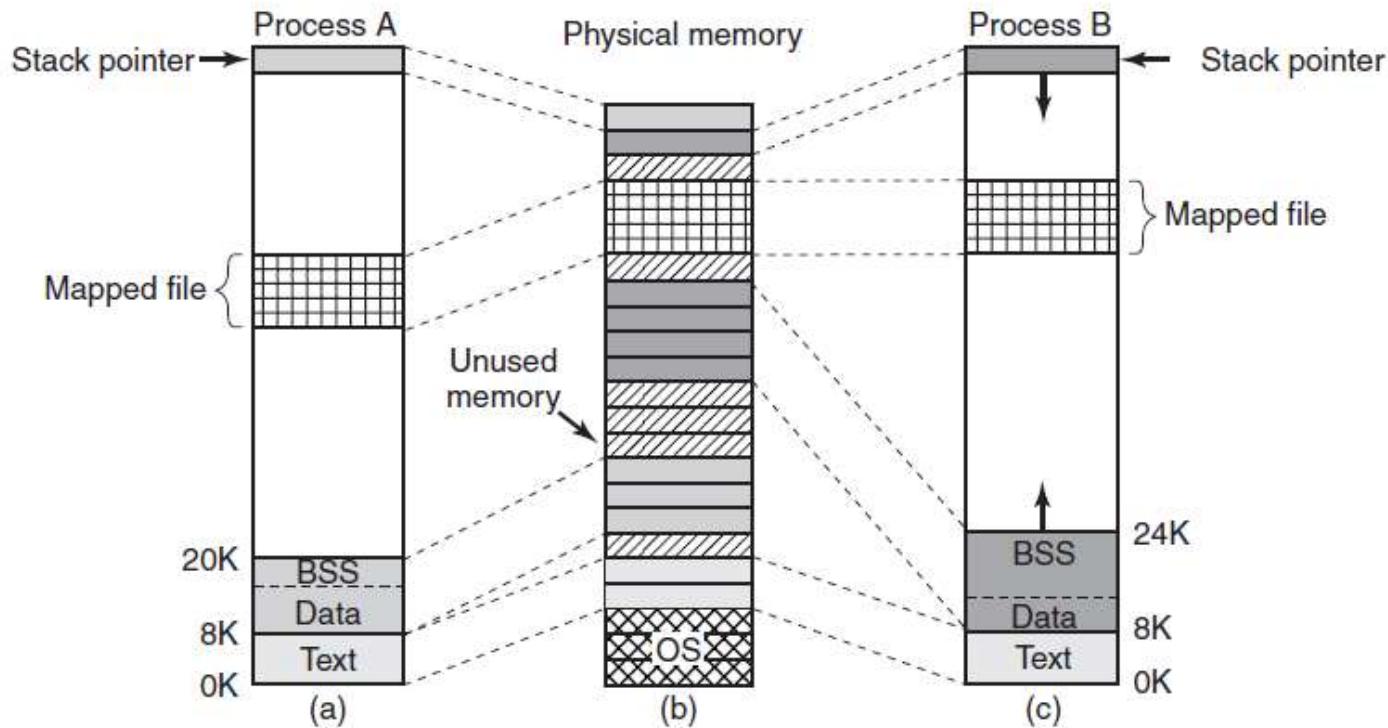


- (a) Process A's virtual address space. (b) Physical memory. (c) Process B's virtual address space.

Tanenbaum & Bo, Modern Operating Systems:4th ed., (c) 2013 Prentice-Hall, Inc.

# Virtual Memory

- Logical view describing the layout of the address space
  - the address space consists of a set of nonoverlapping regions, each region representing a continuous, page-aligned subset of the address space



- Two processes can share a mapped file

Tanenbaum & Bo, Modern Operating Systems:4th ed., (c) 2013 Prentice-Hall, Inc.

# Memory Management

---

## □ System Call

| System call                                               | Description              |
|-----------------------------------------------------------|--------------------------|
| <code>s = brk(addr)</code>                                | Change data segment size |
| <code>a = mmap(addr, len, prot, flags, fd, offset)</code> | Map a file in            |
| <code>s = unmap(addr, len)</code>                         | Unmap a file             |

Some system calls relating to memory management. The return code `s` is `-1` if an error has occurred; `a` and `addr` are memory addresses, `len` is a length, `prot` controls protection, `flags` are miscellaneous bits, `fd` is a file descriptor, and `offset` is a file offset.

Tanenbaum & Bo, Modern Operating Systems:4th ed., (c) 2013 Prentice-Hall, Inc.

# Virtual Memory

---

- Linux implements several types of virtual memory regions
- The kernel creates a new virtual address space
  1. When a process runs a new program with the `exec()` system call
  2. Upon creation of a new process by the `fork()` system call

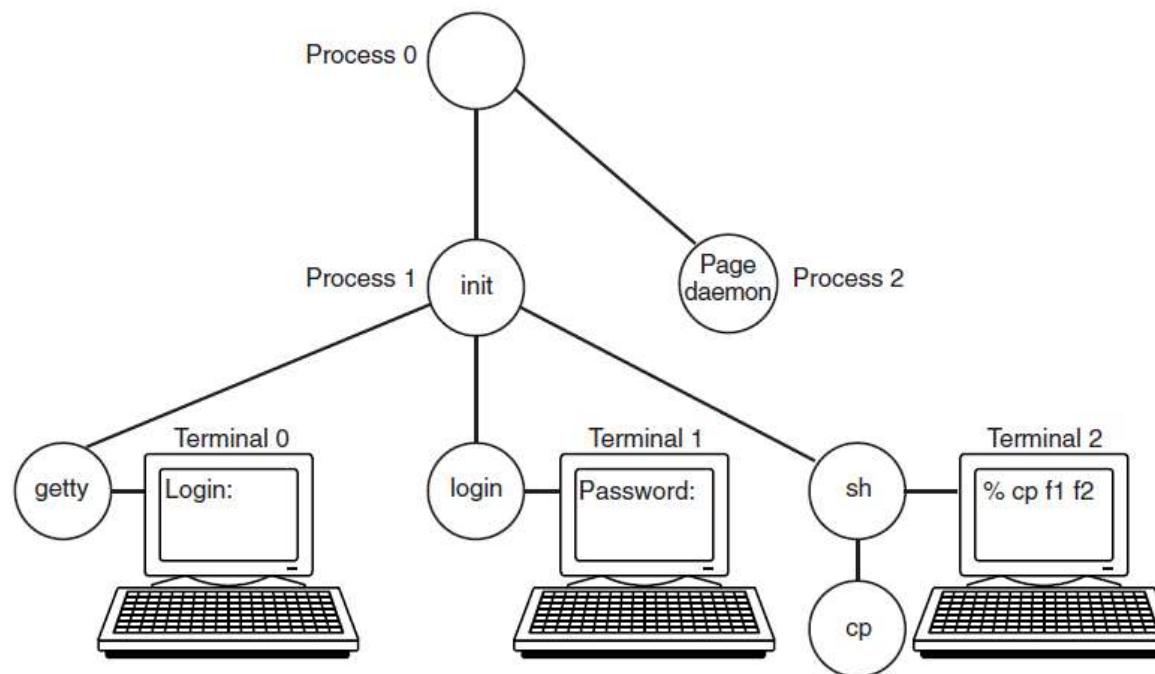
# Virtual Memory

---

- On executing a new program, the process is given a new, completely empty virtual-address space
  - program-loading routines populate the address space with virtual-memory regions
- Creating a new process with `fork()` involves creating a complete copy of the existing process virtual address space
  - The kernel copies the parent process `vm_area_struct` descriptors,
  - Then creates a new set of page tables for the child
  - The parent page tables are copied directly into the child, with the reference count of each page covered being incremented
  - After the fork, the parent and child **share the same physical pages** of memory in their address spaces
  - The areas are marked as read/write, but the pages are marked as **read only**
  - If either process tries to write on a page, a protection fault occurs and the kernel sees that the area is logically writable but the page is not writeable
  - it gives the process a copy of the page and marks it read/write (copy on write)

# Swapping and Paging

- The VM paging system relocates pages of memory from physical memory out to disk when the memory is needed
- Paging is implemented partly by the kernel and partly by a process called the page daemon
  - The page daemon is process 2 (process 0 is the idle process- traditionally called the swapper—and process 1 is init)



# Swapping and Paging

---

- The VM paging system relocates pages of memory from physical memory out to disk when the memory is needed
- Paging is implemented partly by the kernel and partly by a process called the page daemon
  - The page daemon is process 2 (process 0 is the idle process- traditionally called the swapper—and process 1 is init)
  - Like all daemons, the page daemon runs periodically
    - Once awake, it looks around to see if there is any work to do
    - If it sees that the number of pages on the list of free memory pages is too low, it starts freeing up more pages
- Linux is a fully demand-paged system with no prepaging and no working-set concept

# Swapping and Paging

---

- The VM paging system relocates pages of memory from physical memory out to disk when the memory is needed
- Paging is implemented partly by the kernel and partly by a new process called the page daemon.
  - The page daemon is process 2 (process 0 is the idle process- traditionally called the swapper—and process 1 is init)
- The VM paging system can be divided into two sections:
  - **Page replacement** algorithm decides which pages to write out to disk and when
    - ▶ modified version of the standard clock (or second-chance) algorithm
  - **Paging mechanism** carries out the transfer and pages data back into physical memory
    - ▶ Page out to either swap device or normal files (slower)
    - ▶ **Bitmap** used to track used blocks in swap space (kept in physical memory)

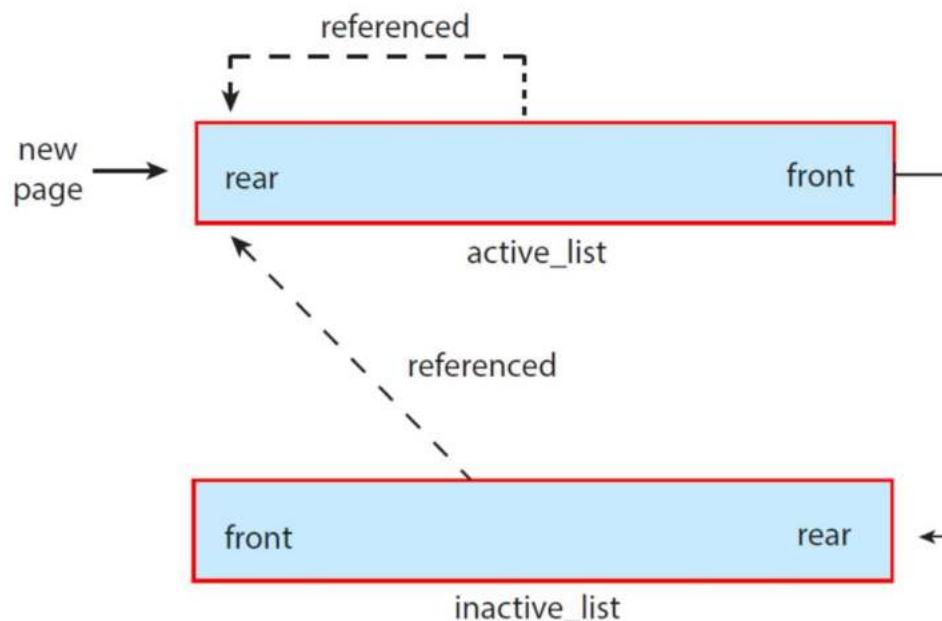
# Page Replacement

---

- **Demand paging with free frame list**
- **Global page replacement with clock approx LRU**
- PFRA (Page Frame Reclaiming Algorithm)
  - At boot time, init starts up a page daemon, kswapd and configures them to run periodically
    - ▶ Each time kswapd awakens, it checks to see if there are enough free pages available
    - ▶ If there is enough memory, it goes back to sleep, although it can be awakened early if more pages are suddenly needed
    - ▶ If the available memory for any of the zones ever falls below a threshold, kswapd initiates the page frame reclaiming algorithm
  - PFRA uses a clock-like algorithm to select old pages for eviction within a certain category
    - ▶ During PFRA, pages are moved between the active and inactive list

# Page Replacement

- Due liste di pagine **active\_list** e **inactive\_list**, le inactive sono elegibili per essere riusate
  - Un **accessed** bit per stabilire chi ha avuto un riferimento
  - Nuova pagina nel retro della active list (come ogni pagina con accesso in active)
  - Il bit è periodicamente resettato
  - Last Recently Used emerge sul fronte, sul retro retrocede all'inactive
  - Un demone kswapd periodicamente verifica se occorre liberare memoria, se la memoria libera va sotto soglia scorre la **inactive\_list** e libera frame



# Executing and Loading User Programs

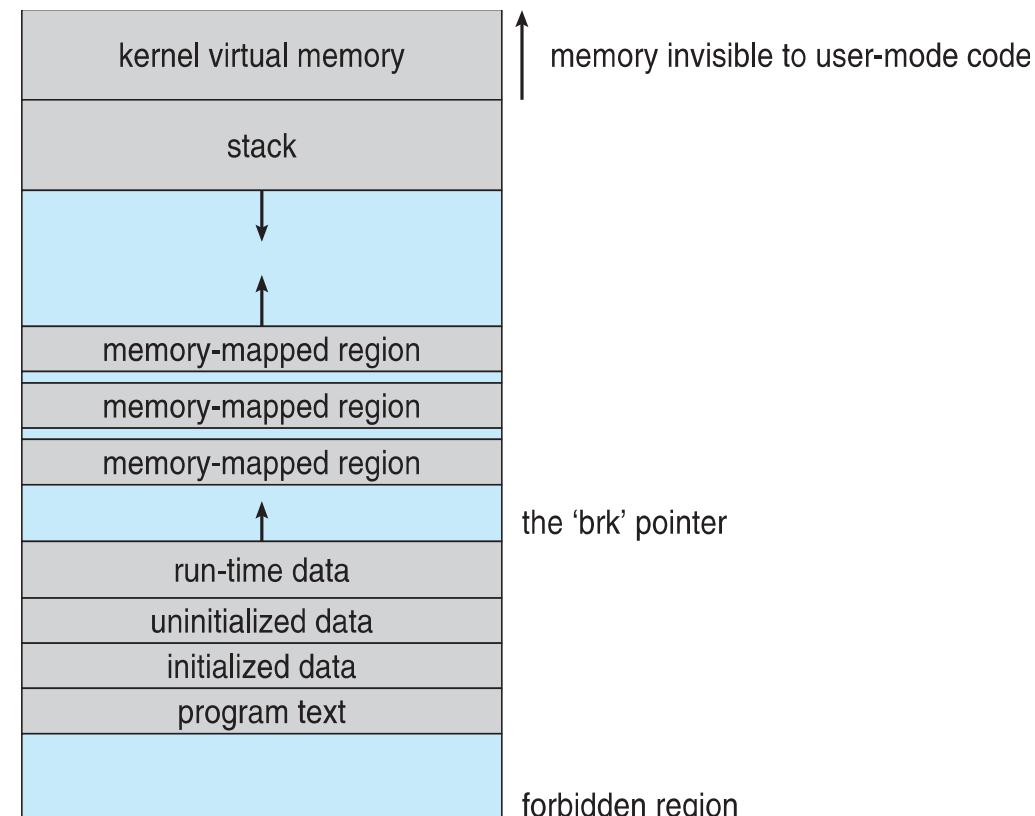
---

- Kernel execution of user programs is triggered by a call to the exec()
  - Verify that the calling process has permission rights to the file being executed
  - Kernel invokes a loader routine to start running the program
- There is no single routine in Linux for loading a new program
  - Linux maintains a table of functions for loading programs
  - The registration of multiple loader routines allows Linux to support both the **ELF** and **a.out** binary formats

# Executing and Loading User Programs

## □ Mapping of Programs into Memory

- Initially, binary-file pages are mapped into virtual memory
  - When a program tries to access a given page will a **page fault** result in that page being loaded into physical memory
- An ELF binary file consists of a **header** and several **page-aligned** sections
- The ELF loader reads the header and maps the sections of the file into regions of virtual memory
  - Regions to be initialized are:  
stack, program text, data regions
  - Stack top, text bottom
  - beyond fixed-sized regions  
variable-sized region that  
programs can expand (brk)



Once mappings have been set up, the loader initializes the process program-counter register with the starting point recorded in the ELF header and the process can be scheduled

# Executing and Loading User Programs

---

- gcc produces executable files in the ELF file format
- Command readelf read parts of an elf file

- Header

- ▶ list the highest level header in the ELF file

```
% readelf -h a.out
```

- Sections

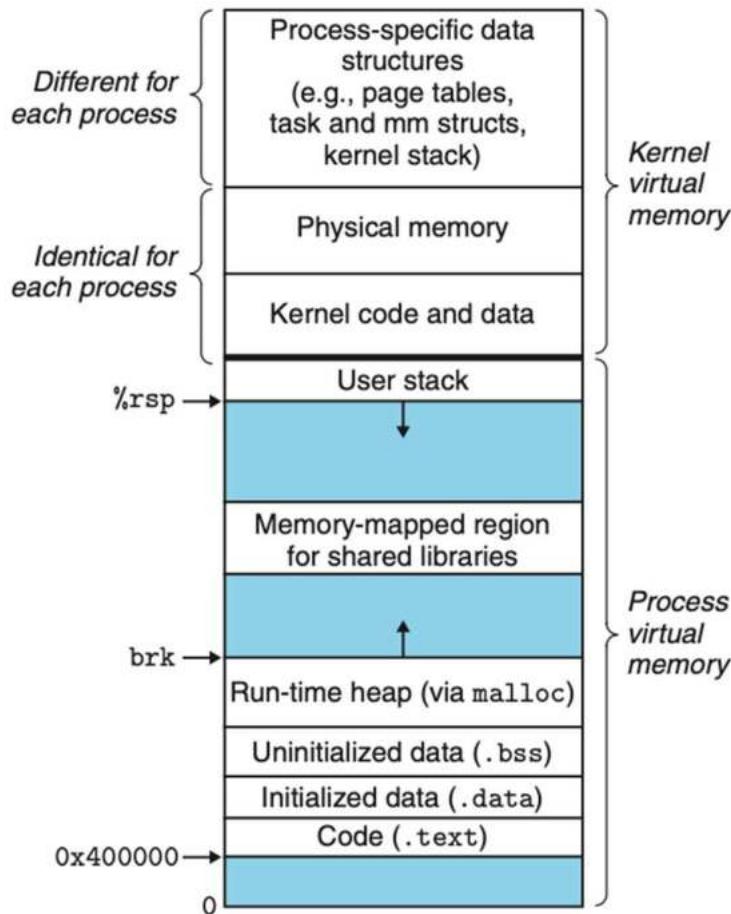
- ▶ lists the sections of the process' address space that are specified from the a.out file

```
% readelf -S a.out
```

# Kernel Virtual Memory

- Mapping of Programs into Memory
  - Initially, binary-file pages are mapped into virtual memory

**Figure 9.26**  
The virtual memory of a Linux process.



# Static and Dynamic Linking

---

- Once the program has been loaded and has started running, all the necessary contents of the bin file are into the process virtual address space
- Programs also need to run functions from the system libraries and these library functions must also be loaded
  - A program whose necessary library functions are embedded directly in the program's executable binary file is **statically** linked to its libraries
  - The main disadvantage of static linkage is that every program generated must contain copies of exactly the same common system library functions
- *Dynamic* linking is more efficient in terms of both physical memory and disk-space usage because it loads the system libraries into memory only once

# Dynamic Linking

---

- Linux implements dynamic linking
  - The programs `ld.so` and `ld-linux.so` find and load the shared libraries needed by a program, prepare the program to run, and then run it (`so` = shared object)
  - The program `ld.so` handles `a.out` binaries; `ld-linux.so*` handles ELF
  - both have the same behavior, and use the same support files and programs:
    - `ldd` prints the shared libraries required by each program
    - `ldconfig` configure dynamic linker run-time bindings

# File Systems

---

- UNIX files can be anything capable of handling the input or output of a stream of data
- To the user, Linux file system appears as a hierarchical directory tree obeying UNIX semantics
- Internally, the kernel hides implementation details and manages the multiple different file systems via an abstraction layer, that is, the virtual file system (VFS)
- The Linux VFS is designed around object-oriented principles and is composed of four components:
  - A set of definitions that define what a file object is allowed to look like
    - The **inode object** structure represent an individual file
    - The **file object** represents an open file
    - The **superblock object** represents an entire file system
    - A **dentry object** represents an individual directory entry

# File Systems

---

- To the user, Linux's file system appears as a hierarchical directory tree obeying UNIX semantics
- Internally, the kernel hides implementation details and manages the multiple different file systems via an abstraction layer:
  - Virtual File System (VFS)
- The Linux VFS is designed around object-oriented principles and layer of software to manipulate those objects with a set of operations on the objects
  - Every object contains a pointer to a function table that lists the addresses of the actual functions for the operations
  - For example for the file object operations include (from `struct_file_operations` in `/usr/include/linux/fs.h`)
    - int `open(...)` — Open a file
    - `ssize_t read(...)` — Read from a file
    - `ssize_t write(...)` — Write to a file
    - `int mmap(...)` — Memory-map a file

# File Systems

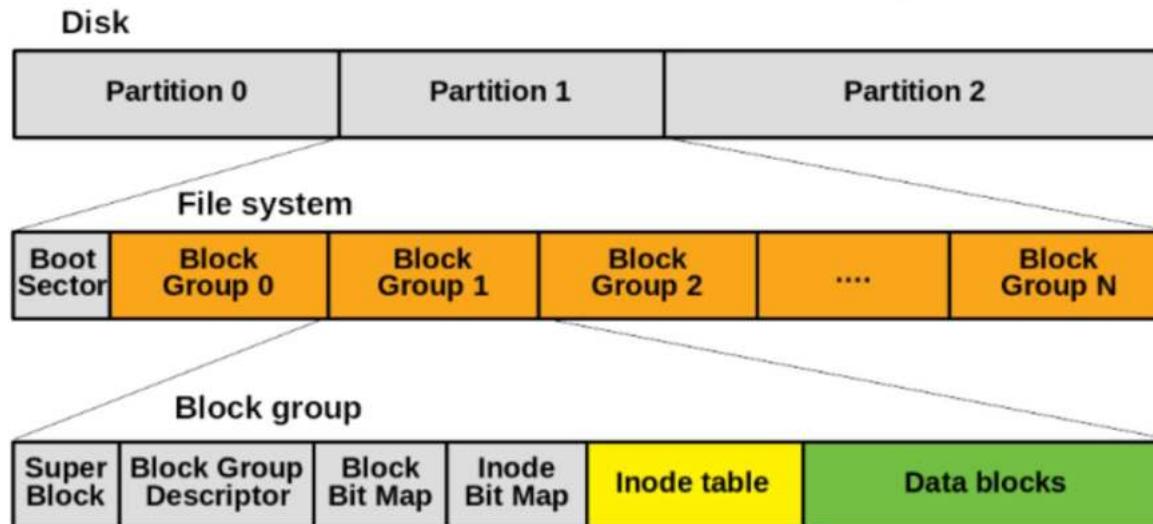
---

- **inode** and **file objects** are the mechanisms used to access files
- An **inode object** is a data structure containing pointers to the disk blocks that contain the actual file contents
- A **file object** represents a point of access to the data in an **open file**
  - keeps track of offset, permissions (for example, read or write), etc.
  - one file object for every instance of an open file
- **Superblock object** represents a connected set of files that form a FS
  - The main responsibility is to provide access to inodes
- **Dentry object** represents a directory entry
- Operations on directories
  - creating, deleting, and renaming a file in a directory
  - The system calls for these directory operations do not require that the user open the files

# The Linux ext File System

- **ext3, ext4** are standard on disk file system for Linux
  - Uses a mechanism similar to that of BSD Fast File System (FFS) for locating data blocks belonging to a specific file
  - Supersedes older **extfs**, **ext2** file systems
  - **ext4** adding features like extents
  - many other file system choices with Linux distros

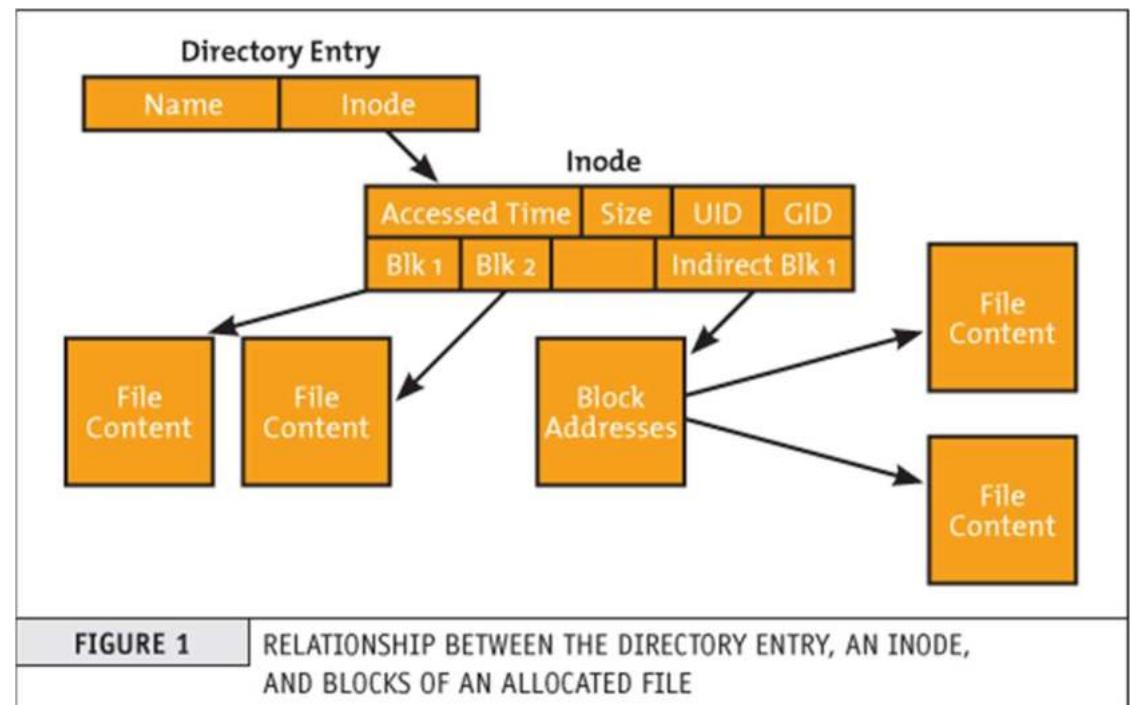
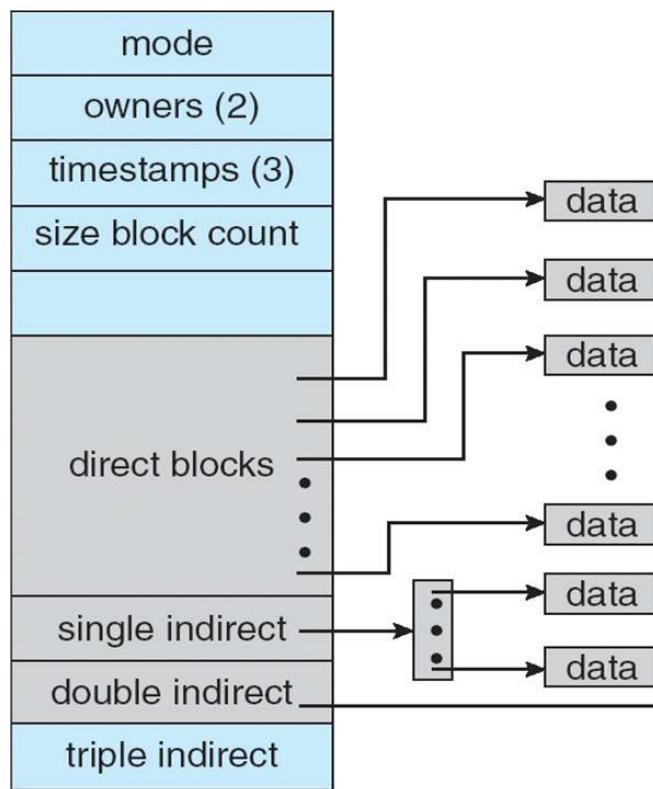
Ext2 on a disk consists of many **groups of disk blocks** (of the same size, located sequentially one after the other). Block groups reduce **file fragmentation**.



Source: <https://computing.ece.vt.edu/~changwoo/ECE-LKP-2019F/I/lec21-fs.pdf>

# The Linux File System

- Inode
- Directory entry
- Block addresses



# The Linux File System

---

## □ Evolution

- First FS of Linux was Minix
- Minix filesystem, written by Andrew S. Tanenbaum and part of Tanenbaum's Minix operating system (educational purposes)
- Minix has the following structure:
  - A boot sector in the first sector of the hard drive on which it is installed.
  - The boot block includes a very small boot record and a partition table.
  - The first block in each partition is a superblock that contains the metadata that defines the other filesystem structures and locates them on the physical disk assigned to the partition
  - An inode bitmap block, which determines which inodes are used and which are free
  - The inodes, which have their own space on the disk. Each inode contains information about one file, including the locations of the data blocks, i.e., zones belonging to the file
  - A zone bitmap to keep track of the used and free data zones.
  - A data zone, in which the data is stored
- an inode is a 256-byte block on the disk and stores data about the file
  - The inode also contains data that points to the location of the file's data on the hard drive

# The Linux File System

---

## □ Evolution

- The original EXT filesystem (Extended) was written by Rémy Card and released with Linux in 1992 to overcome some size limitations of the Minix filesystem
  - ▶ the metadata was based on the Unix filesystem (UFS), which is also known as the Berkeley Fast File System (FFS)
  - ▶ quickly superseded by the EXT2 filesystem
- Like Minix, EXT2 has a boot sector in the first sector of the hard drive on which it is installed
  - ▶ it includes a small boot record and a partition table

# The Linux File System

---

## □ Evolution

- Like Minix, EXT2 has a boot sector in the first sector of the hard drive on which it is installed
  - ▶ it includes a small boot record and a partition table

The **main features of ext2** affecting its **performance**:

- When creating the system, the administrator can choose the **optimal block size** (in the range of 1 KB to 4 KB), depending on the **expected average file size**.
- When creating a system, the administrator can set the **number of inodes** for a particular partition size, depending on the **number of files** expected.
- Disk blocks are divided into **groups** including adjacent tracks, thanks to which reading a file located within a single group is associated with a **short seek time**.
- The file system **preallocates** disk blocks for **regular files**, so as the file grows, blocks are already reserved for it in physically adjacent areas, which **reduces file fragmentation**.
- Thanks to the careful implementation, it is stable and flexible.
- Defined in **/fs/ext2**.

# The Linux File System

---

## □ Evolution

- The original EXT filesystem (Extended) was written by Rémy Card and released with Linux in 1992 to overcome some size limitations of the Minix filesystem
  - the metadata was based on the Unix filesystem (UFS), which is also known as the Berkeley Fast File System (FFS)
  - quickly superseded by the EXT2 filesystem
- Like Minix, EXT2 has a boot sector in the first sector of the hard drive on which it is installed
  - it includes a small boot record and a partition table
- In EXT3 we have the journal, which records in advance the changes that will be performed to the filesystem.
  - The rest of the disk structure is similar to EXT2

# The Linux ext3 File System

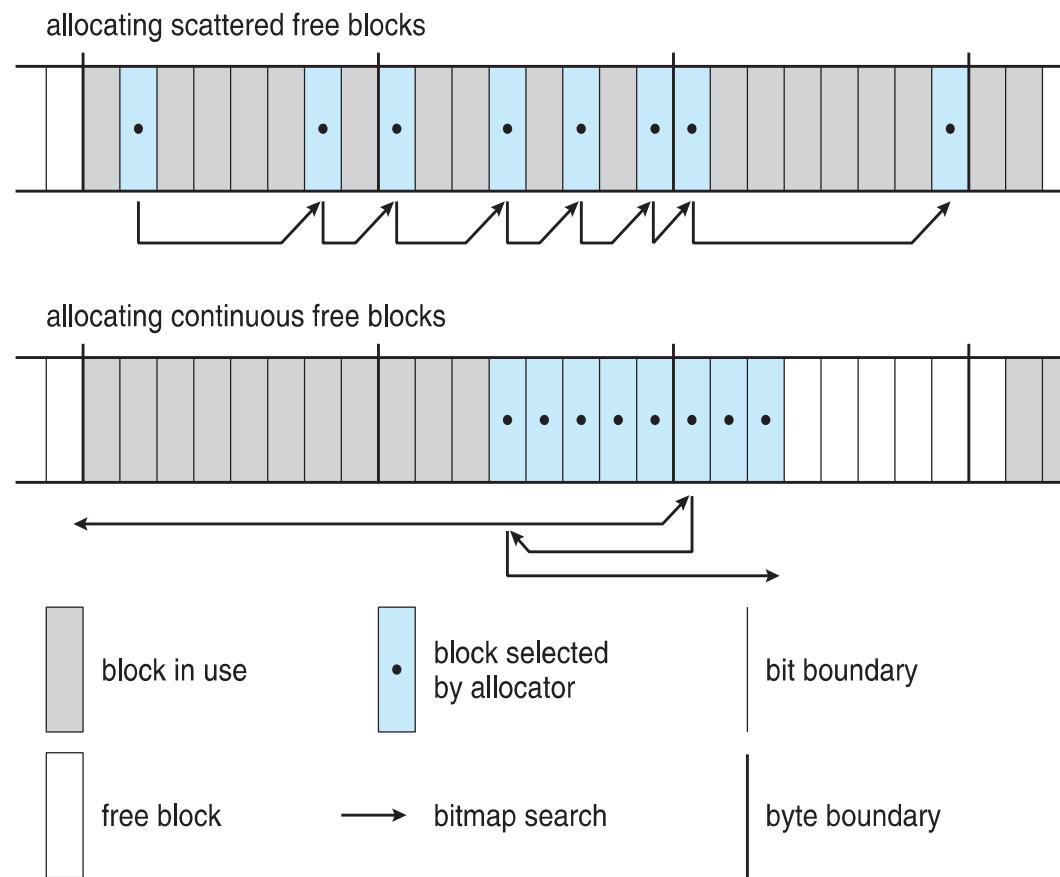
---

- The main differences between ext3 and UFS concern their disk allocation policies
  - In UFS and BFFS, the disk is allocated to files in blocks of 8Kb, with blocks subdivided into fragments of 1Kb to store small files or partially filled blocks at the end of a file
  - ext3 does **not use fragments**; it performs its allocations in smaller units
    - ▶ The default block size on ext3 varies as a function of total size of file system with support for 1, 2, 4 and 8 KB blocks
- ext3 uses **cluster allocation policies** designed to place **logically adjacent** blocks of a file into physically adjacent blocks on disk, so that it can submit an I/O request for several disk blocks as a single operation on a **block group**
  - ▶ Ext3 file system is partitioned into 128 MB block group chunks
  - ▶ Each block group maintains a single block bitmap to describe data block availability inside this block group
  - ▶ This way allocation on different block groups can be done in parallel
  - ▶ attempts to allocate the file to the block group of the file inode
  - ▶ tries to keep allocations **physically contiguous** if possible, reducing fragmentation
  - ▶ Maintains **bit map** of free blocks in a block group, searches for free byte to allocate at least **8 blocks at a time**

# Ext3 Block-Allocation Policies

First, ext3 searches for an entire free byte in the bitmap; if it fails to find one, it looks for any free bit

Once a free block has been identified, the search is extended backward until an allocated block is encountered



# Journaling

---

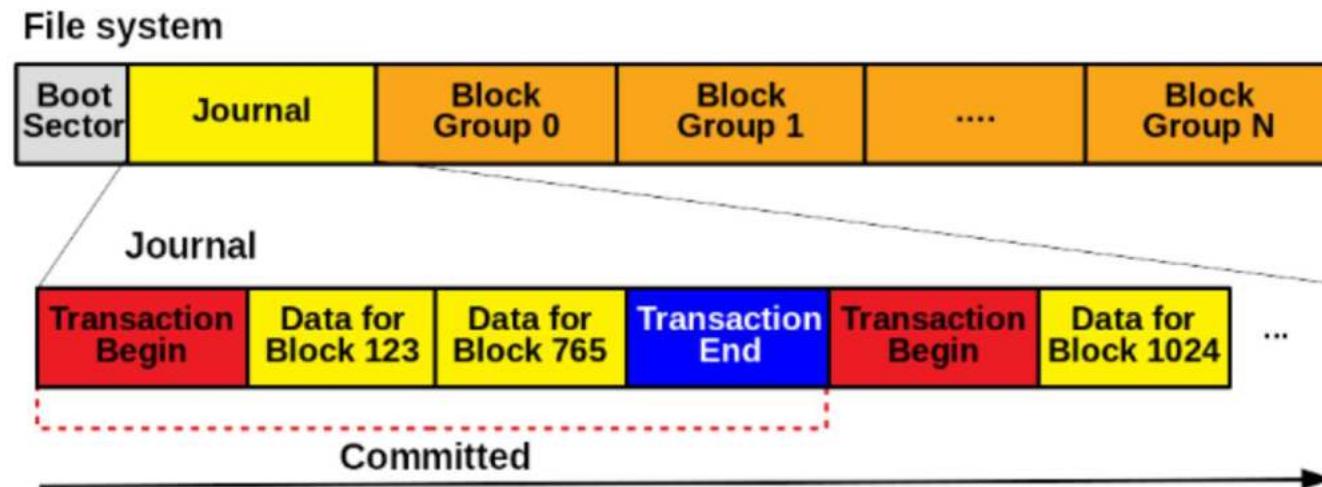
- ext3/ext4 implements **journaling**, with file system updates first written to a log file in the form of **transactions**
  - Once in log file, considered committed
  - Over time, log file transactions replayed over file system to put changes in place
- On system crash, some transactions might be in journal but not yet placed into file system
  - Must be completed once system recovers
  - No other consistency checking is needed after a crash (much faster than older methods)
- Improves write performance on hard disks by turning random I/O into sequential I/O

# Journaling

- ▣ ext3/ext4 implements **journaling**, with file system updates first written to a log file in the form of **transactions**
  - ▣ Once in log file, considered committed
  - ▣ Over time, log file transactions replayed over file system to put changes in place

Journal is logically a **fixed-size, circular array**.

- Implemented as a **special file** with a **hard-coded inode number**.
- Each journal **transaction** is composed of a **begin** marker, **log**, and **end** marker.



Source: <https://computing.ece.vt.edu/~changwoo/ECE-LKP-2019F/I/lec21-fs.pdf>

# The Linux File System

---

## □ Evolution

- The original EXT filesystem (Extended) was written by Rémy Card and released with Linux in 1992 to overcome some size limitations of the Minix filesystem
  - the metadata was based on the Unix filesystem (UFS), which is also known as the Berkeley Fast File System (FFS)
  - quickly superseded by the EXT2 filesystem
- Like Minix, EXT2 has a boot sector in the first sector of the hard drive on which it is installed
  - it includes a small boot record and a partition table
- In EXT3 we have the journal, which records in advance the changes that will be performed to the filesystem.
  - The rest of the disk structure is similar to EXT2
- The EXT4 filesystem primarily improves performance, reliability, and capacity
  - data allocation was changed from fixed blocks to extents
  - EXT4 reduces fragmentation by scattering newly created files across the disk so that they are not only in one location at the beginning of the disk
  - delayed allocation to allow the filesystem to collect all the data being written to the disk before allocating space to it

# The Linux Proc File System

---

- Proc file system (procfs) is virtual file system created on fly when system boots and is dissolved at time of system shut down
- It presents information about processes and other system information in a hierarchical file-like structure
- Typically, it is mapped to a mount point named **/proc** at boot time
- It acts as an interface to internal data structures about running processes in the kernel
  - ps uses the proc file system to obtain its data, without using any specialized system calls
  - `/proc/PID/environ`, `/proc/PID/cwd`, `/proc/PID/status`
- **/proc** also includes non-process-related system information
  - although in the 2.6 kernel much of that information moved to a separate pseudo-file system, **sysfs**, mounted under **/sys**
  - `/proc/buddyinfo`, `/proc/cpuinfo`

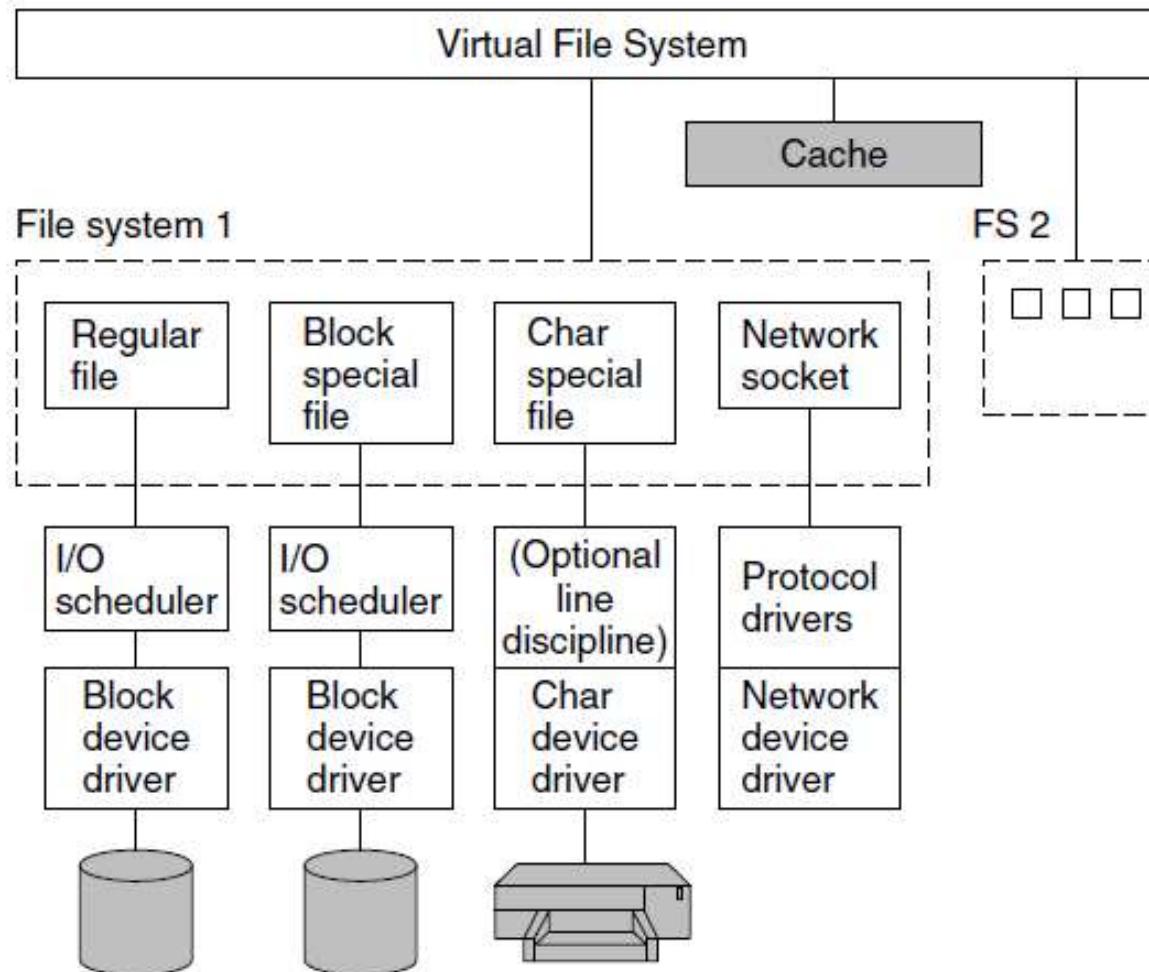
# Input and Output

---

- All I/O devices are made to look like files and are accessed as such with the same read and write system calls that are used to access all ordinary files.
- Users can open an access channel to a device in the same way they open any other file
  - devices can appear as objects within the file system
  - The devices into the file system as what are called special files
    - ▶ assigned a path name, usually in /dev.
- Linux splits all devices into three classes:
  - **block devices** allow random access to completely independent, fixed size blocks of data
    - ▶ Block devices are typically used to store file systems, but direct access to a block device is also allowed
  - **character devices** include most other devices; they don't need to support the functionality of regular files
    - ▶ character devices are accessed serially (mouse, keyboard)
  - **network devices** are interfaced via the kernel's networking subsystem

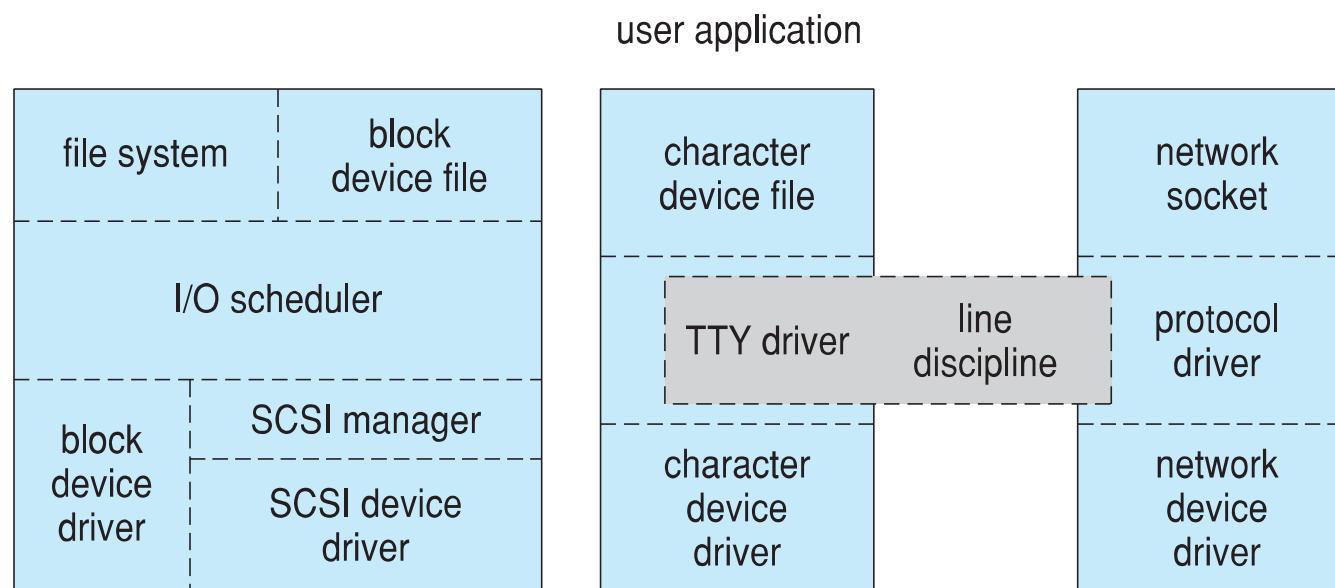
# Input and Output

- All I/O devices are made to look like files and are accessed as such with the same read and write system calls that are used to access all ordinary files.



# Device-Driver Block Structure

Overall structure of the device driver system



# Block Devices

---

- Provide the main interface to all disk devices in a system
- The block buffer cache serves two main purposes:
  - it acts as a pool of buffers for active I/O
  - it serves as a cache for completed I/O
- The **request manager** manages the reading and writing of buffer contents to and from a block device driver
- Kernel 2.6 introduced **Completely Fair Queueing (CFQ)**
  - Now the default scheduler
  - Different from elevator algorithms
  - Maintains set of lists, one for each process by default
  - Uses C-SCAN algorithm, with round robin between all outstanding I/O from all processes
  - Four blocks from each process put on at once

# Character Devices

---

- A device driver which does not offer random access to fixed blocks of data
- A character device driver must register a set of functions which implement the driver's various file I/O operations
- The kernel performs almost no preprocessing of a file read or write request to a character device, but simply **passes on** the request to the device
- The main exception to this rule is the special **subset of character device drivers** which implement **terminal devices**, for which the kernel maintains a standard interface

# Character Devices

---

- The kernel maintains a standard interface to these drivers by means of a set of `tty_struct` structures.
  - Each structure provides buffering and flow control on the data stream from the terminal device and feeds those data to a **line discipline**
- **Line discipline** is an interpreter for the information from the terminal device
  - The most common line discipline is `tty discipline`
    - It glues the terminal's data stream onto standard input and output streams of user's running processes, allowing processes to communicate directly with the user's terminal
    - E.g., local line editing can be done (i.e., erased characters and lines can be removed)
    - carriage returns can be mapped onto line feed
    - A process can put the line in raw mode, the line discipline will be bypassed
    - Not all devices have line disciplines.
  - Several processes may be running simultaneously
    - `tty_line` discipline responsible for attaching and detaching terminal's input and output from various processes connected to it as processes are suspended or awakened

# Interprocess Communication

---

- Like UNIX, Linux informs processes that an event has occurred via **signals**
- There is a limited number of signals, and they cannot carry information:
  - Only the fact that a signal occurred is available to a process
- The Linux kernel does not use signals to communicate with processes running in kernel mode, rather, communication within the kernel is accomplished via scheduling states and **wait\_queue** structures
- Also implements System V Unix semaphores
  - Process can wait for a signal or a semaphore
  - Semaphores scale better
  - Operations on multiple semaphores can be atomic

# Passing Data Between Processes

---

- The **pipe** mechanism allows a child process to inherit a communication channel to its parent
  - data written to one end of the pipe can be read at the other
- Shared memory offers an extremely fast way of communicating
  - any data written by one process to a shared memory region can be read immediately by any other process that has mapped that region into its address space
- To obtain synchronization, shared memory must be used in conjunction with another interprocess-communication mechanism

# Network Structure

---

- Networking is a key area of functionality for Linux
  - It supports the standard Internet protocols for UNIX to UNIX communications
  - It also implements protocols native to non-UNIX operating systems, in particular, protocols used on PC networks, such as Appletalk and IPX
- Networking in the Linux kernel is implemented by three layers of software:
  - The socket interface
    - User applications perform all networking requests through the socket interface
  - Protocol drivers
    - The protocol layer decides to which socket or device it will send the packet
  - Network device drivers
- Internet protocol suite most important protocols in the Linux net. system
  - It implements routing between different hosts anywhere on the network
  - On top of the routing protocol are built the UDP, TCP and ICMP protocols

# Security

---

- Authentication and Access control
- Authentication has typically been performed through the use of a publicly readable password file
- The **pluggable authentication modules (PAM)** system is available under Linux
  - PAM is based on a shared library that can be used by any system component that needs to authenticate users
- Access control under UNIX systems, including Linux, is performed through the use of unique numeric identifiers (**uid** and **gid**)
- Access control is performed by assigning objects a *protections mask*, which specifies which access modes—read, write, or execute—are to be granted to processes with owner, group, or world access

# Security

---

- The UNIX implementation of **setuid** distinguishes between a process's real and effective UID:
  - The real UID is that of the user running the program; the effective UID is that of the file's owner
  - It implements the POSIX specification's saved ***user-id*** mechanism, which allows a process to repeatedly drop and reacquire its effective uid
- Linux provides another mechanism that allows a client to selectively pass access to a single file to some server process without granting it any other privileges

# Lezione 4: Threads & Concorrenza



# Obiettivi

---

- Introdurre la nozione di thread
- Multithreaded programming
- Supporti per threads in Windows e Linux

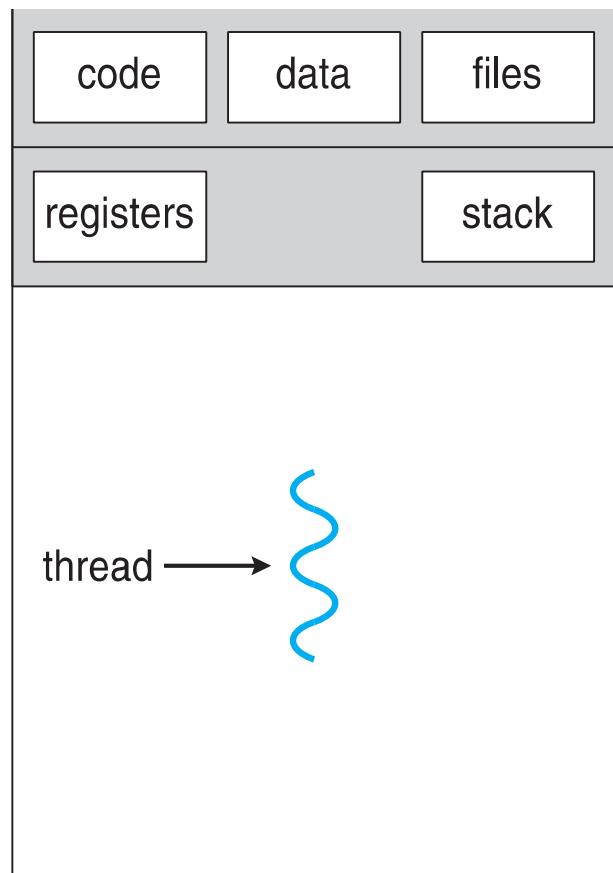
# Motivazioni

---

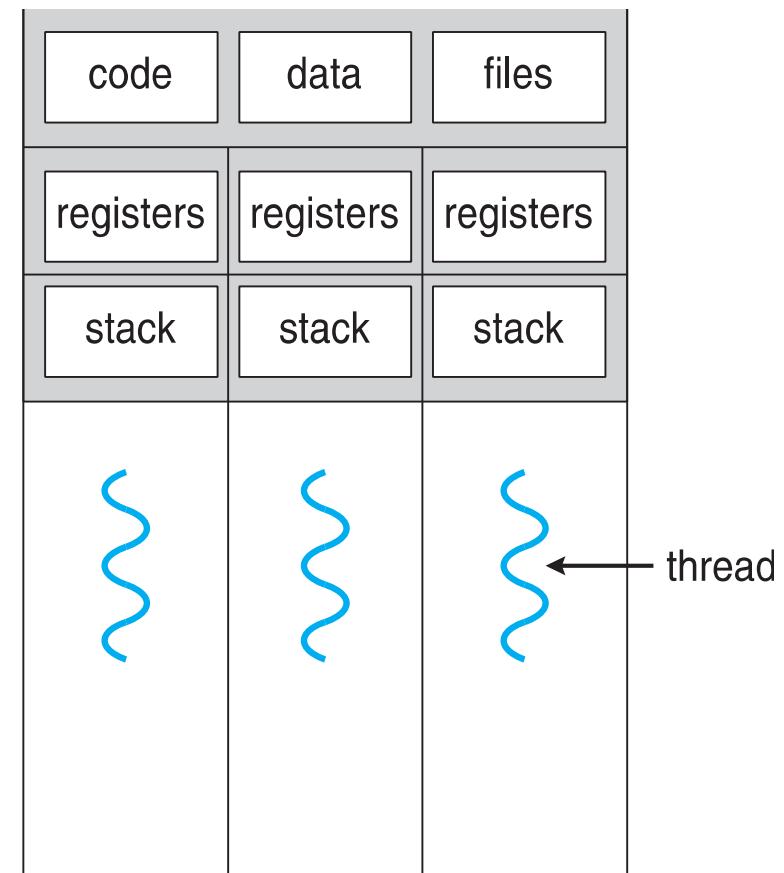
- Le applicazioni sono generalmente multithreaded
- Task multipli in applicazioni implementati da thread separati
  - Update del display
  - Fetch data
  - Spell checking
  - Risposta in rete
- La creazione di processo è pesante, il thread è leggero
- Il kernel è generalmente multithreaded

# Processi Single e Multithreaded

- Unità base di utilizzo di CPU
  - Caratterizzati da Thread ID, PC, Registri e Stack
  - Minima identità di task



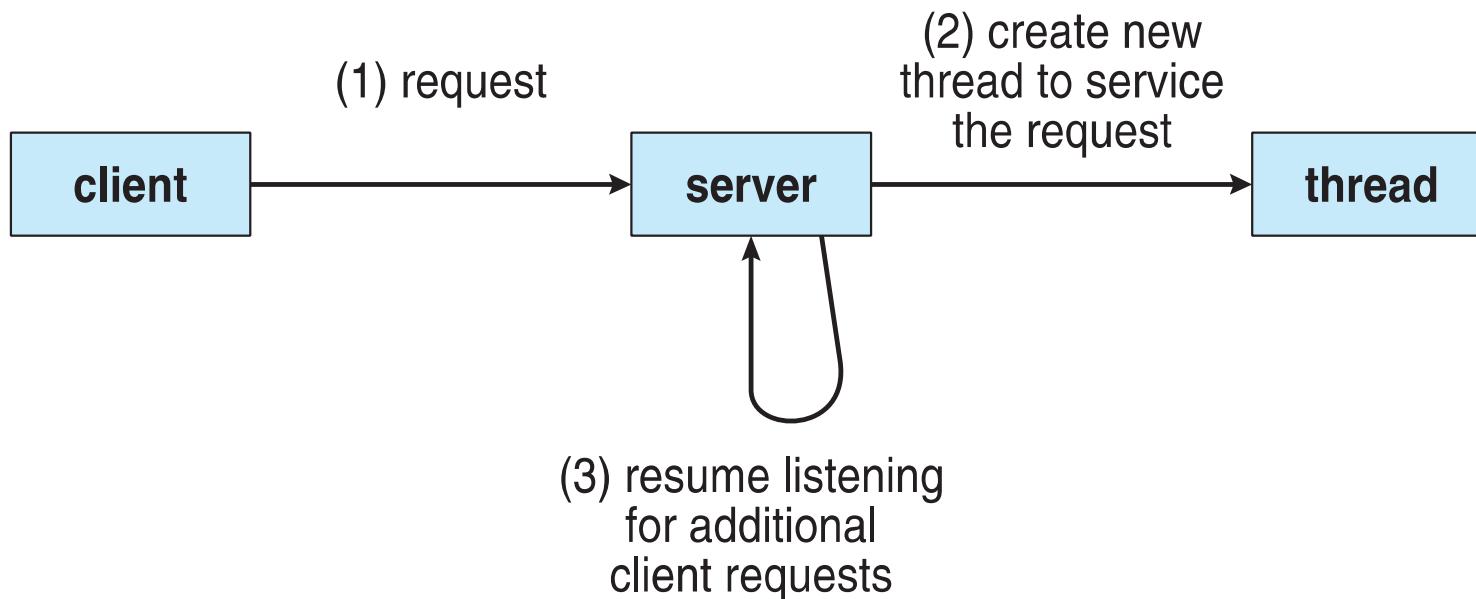
single-threaded process



multithreaded process

# Architettura di Server Multithread

- Molto utilizzate nelle architetture client-server
  - Per ogni client si genera un thread che gestisce il client
  - Per ogni richiesta di un client un thread
  - Server può gestire tante richieste eseguendo molti task leggeri



# Benefici

---

- **Risposta**
  - Esecuzione continua se una parte del processo è bloccata (importante per le interfacce utente)
- **Risorse**
  - Thread condividono le risorse di un processo, più semplice di shared memory o message passing
- **Economia**
  - Più leggero di una creazione di processo, il **thread switching** ha minore overhead del **context switching**
- **Scalabilità**
  - I processi possono avvantaggiarsi delle architetture multiprocessore

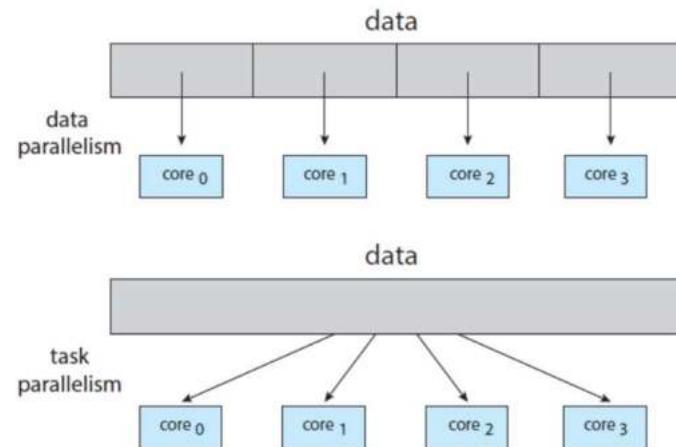
# Multicore Programming

---

- Sistemi multithread forniscono meccanismi utili per sistemi multicore
- Sistemi **Multicore** o **multiprocessori** pongono diversi problemi:
  - **Come dividere le attività sui core**
    - ▶ Computazioni parallelizzabili su più core
  - **Come bilanciare i calcolo sui core**
  - **Come splittare i dati**
  - **Come gestire le dipendenze dei dati sui core**
    - ▶ Dipendenze possono richiedere sincronizzazioni
  - **Come fare test e debugging**
    - ▶ Più tracce di esecuzioni da testare
- **Parallelismo**
  - Più di un task simultaneamente distribuiti su più unità di calcolo
- **Concorrenza**
  - Consente il progresso contemporaneo di più di un task
  - Se processore/core singolo, concorrenza data dallo scheduler che intrallaccia le esecuzioni

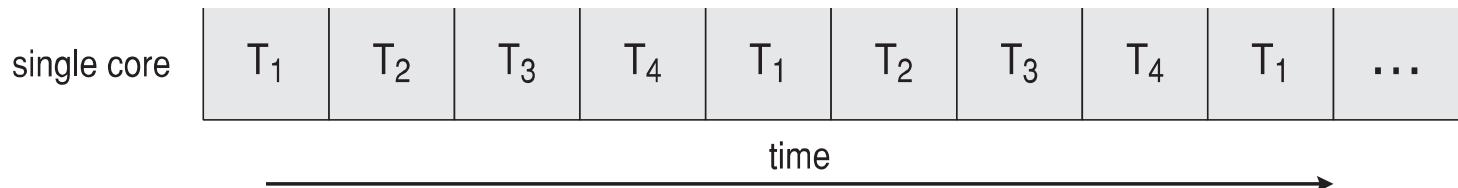
# Multicore Programming

- Tipi di parallelismo
  - **Parallelismo di dati** – sottoinsiemi dei dati su core multipli con stessa operazione per ognuno
    - ▶ Ad esempio calcolo della somma dei numeri su array, si può dividere il compito
  - **Parallelismo di task** – thread sui core dove ciascuno svolge un'operazione particolare
    - ▶ Esempio media e varianza di un set di dati
- Al crescere del numero dei thread aumenta il supporto architetturale
  - CPU hanno core e **hardware thread**
  - Esempio, Oracle SPARC T4 con 8 core e 8 hardware threads per core

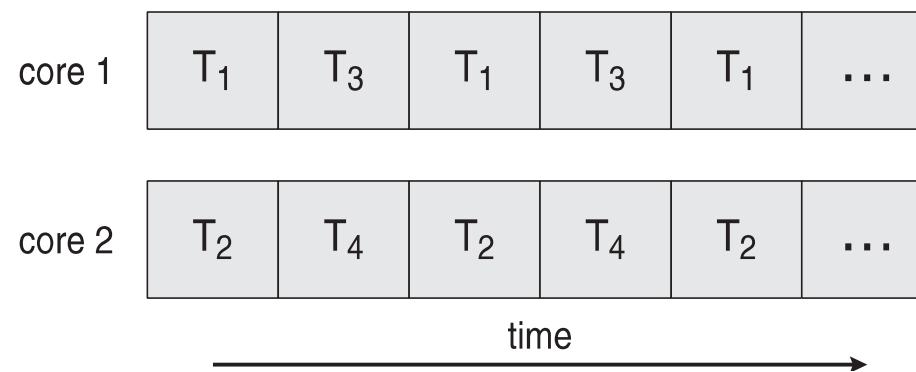


# Concorrenza vs. Parallelismo

- Esecuzione concorrente su sistema single-core:



- Parallelismo su sistema multi-core:



# Legge di Amdahl

---

- Guadagno teorico in performance con l'aggiunta di un core per un'applicazione che ha sia componenti seriali che paralleli
  - $S$  è la porzione seriale
  - $N$  sono i core per il processamento

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

speed-up rispetto ad un singolo core

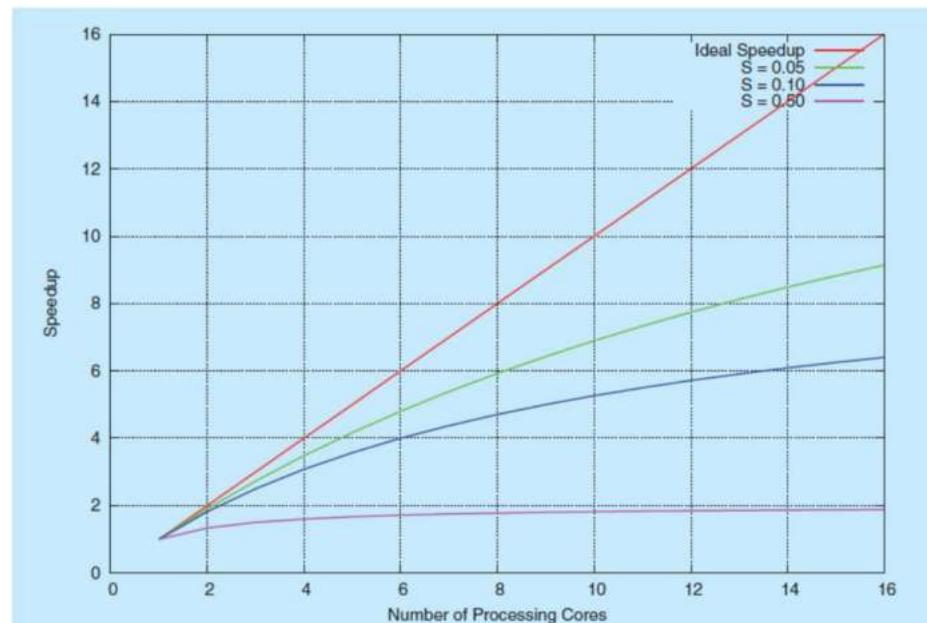
- Esempio: se l'applicazione è al 75% parallela e 25% seriale, passando da 1 a 2 core si ha uno speed-up di 1.6 volte
- Con  $N$  che tende ad infinito lo speed-up tende a  $1 / S$

# Legge di Amdahl

- Guadagno teorico in performance (latenza) con l'aggiunta di un core per un'applicazione che ha sia componenti seriali che paralleli
  - $S$  è la porzione di calcolo seriale (non parallelizzabile)
  - $N$  sono i core per il processamento

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- La porzione seriale di un'applicazione ha effetto molto marcato sul guadagno di performance con un core aggiuntivo



# User Thread e Kernel Thread

- **User thread** – gestita da librerie di thread di livello utente

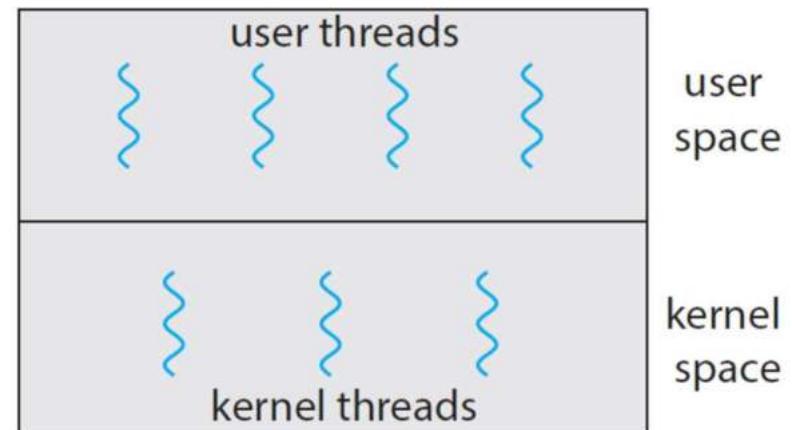
- Tre principali:

- POSIX **Pthreads**
- Windows threads
- Java threads

- **Kernel thread** - supportati dal Kernel

- Esempi – tutti gli SO general purpose inclusi:

- Windows
- Solaris
- Linux
- Tru64 UNIX
- Mac OS X



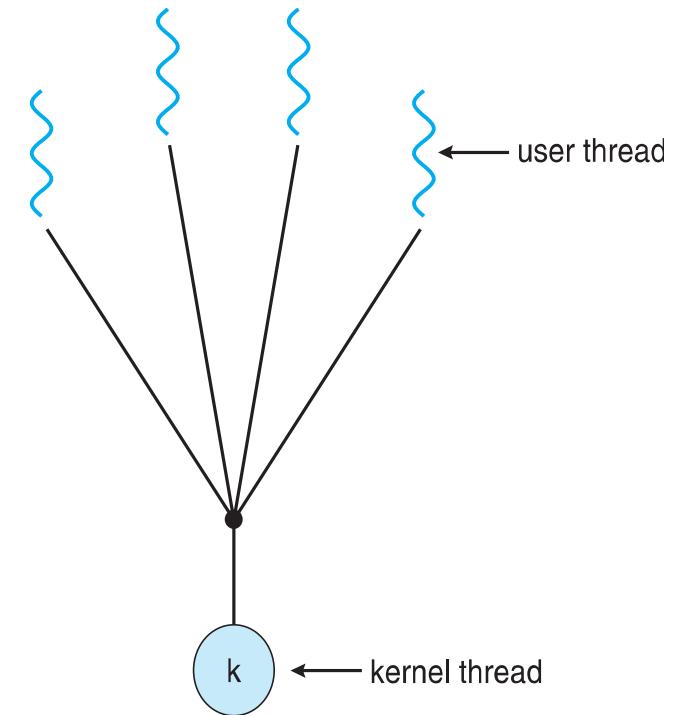
# Modelli di Multithreading

---

- Diversi modi di gestire la relazione tra User e Kernel thread
  - Many-to-One
  - One-to-One
  - Many-to-Many

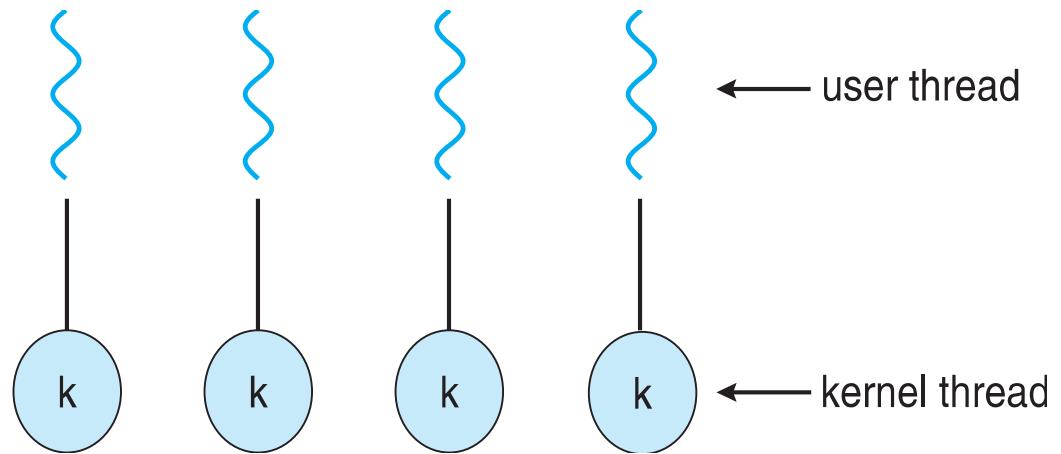
# Many-to-One

- Tanti thread user mappati su un singolo kernel thread
- Però un thread bloccante (es. durante una system call) può bloccare tutto
- Problemi di parallelismo con sistemi multicore perché uno solo alla volta è in esecuzione nel Kernel
- Pochi sistemi utilizzano questo approccio:
  - **Solaris Green Threads** library
  - **GNU Portable Threads** library



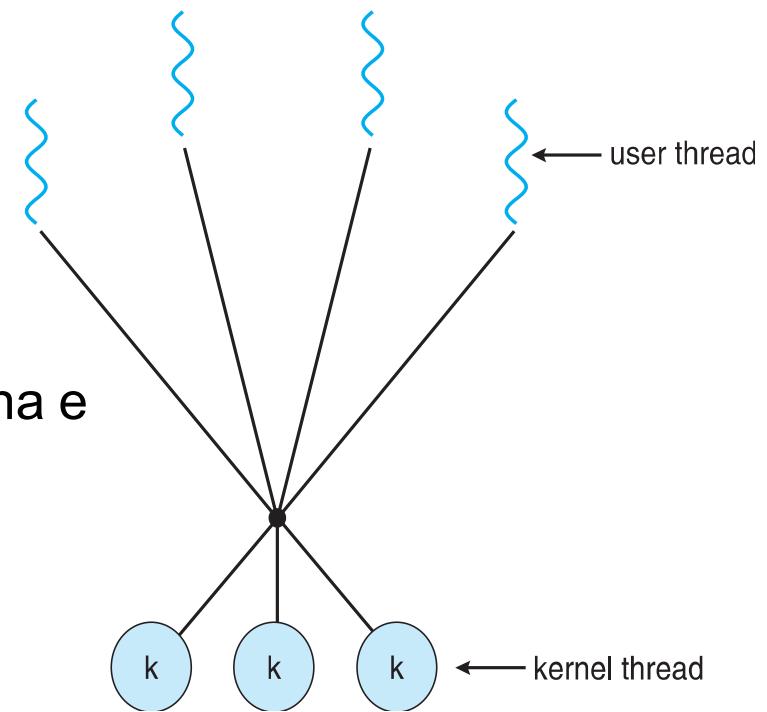
# One-to-One

- Ogni thread user-level mappato su kernel thread
  - La creazione di thread user-level crea un kernel thread
  - Maggiore concorrenza di many-to-one
  - Il numero di thread per processo può essere limitato per evitare overhead
  - Esempi:
    - Windows
    - Linux
    - Solaris 9 and later
  - Utente deve controllare il numero dei thread



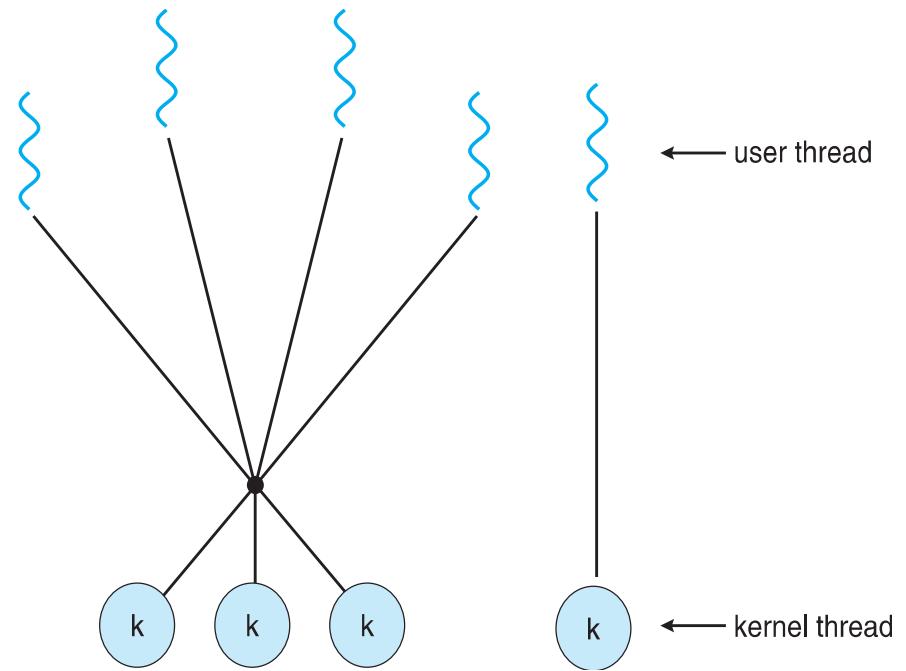
# Many-to-Many

- Molti thread user-level sono mappati su molti thread del kernel
- L'OS può creare un numero sufficiente di thread del kernel
  - Però può gestirne il numero e il grado di parallelismo
- Esempi:
  - Solaris dalla versione 9
  - Windows con *ThreadFiber* package
- Il numero può dipendere dalla macchina e dall'applicazione
- L'utente può lanciare più thread controlla il sistema



# Two-level Model

- Variante del Many-to-Many che permette ad un user-thread di essere legato (**bound**) ad un kernel thread
- Esempi:
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 e precedenti



# Librerie per Thread

---

- **Librerie per i thread (Thread library)**
  - API per creare e gestire i thread
- Due modi principali di implementarle
  - Libreria interamente nello user space
    - ▶ I thread sono implementati al livello utente (gestiti con chiamate di funzione utente)
  - Libreria al livello Kernel supportata dall'SO
    - ▶ I thread sono gestiti con chiamate di Sistema (chiamata di sistema)

# Pthreads

---

- Estensione di POSIX standard (IEEE 1003.1c) fornisce le API per la creazione e sincronizzazione dei thread
- Esiste sia in versione user-level sia kernel-level
- **Specifico**, non **implementazione**
- API specifica il comportamento delle funzioni di libreria, implementazione lasciata allo sviluppatore
- Tipiche in UNIX (Solaris, Linux, Mac OS X)

# Esempio Pthreads

---

- Esempio somma dei primi N numeri
  - Per sempio per  $N = 5$ ,  $\text{Sum} = 1 + 2 + 3 + 4 + 5 = 15$
  - Si può delegare il calcolo ad uno o più thread
  - Abbiamo thread sincroni e asincroni
    - Nel caso sincrono il genitore aspetta i figli
    - Nel caso asincrono il genitore continua subito

# Esempio Pthreads

---

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
 pthread_t tid; /* the thread identifier */
 pthread_attr_t attr; /* set of thread attributes */

 if (argc != 2) {
 fprintf(stderr,"usage: a.out <integer value>\n");
 return -1;
 }
 if (atoi(argv[1]) < 0) {
 fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
 return -1;
 }
}
```

# Esempio Pthreads

---

```
/* get the default attributes */
pthread_attr_init(&attr);
/* create the thread */
pthread_create(&tid,&attr,runner,argv[1]);
/* wait for the thread to exit */
pthread_join(tid,NULL);

printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
 int i, upper = atoi(param);
 sum = 0;

 for (i = 1; i <= upper; i++)
 sum += i;

 pthread_exit(0);
}
```

# Codice Pthreads per il join di 10 Thread

---

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
 pthread_join(workers[i], NULL);
```

# Pthreads

---

Per creare thread addizionali relativi ad uno stesso processo, Posix prevede la funzione:

```
#include <pthread.h>

int pthread_create (pthread_t *tid, const pthread_attr_t *attr,
 void * (*start_func) (void *), void *arg);
```

- se la chiamata ha successo, *tid* punta al thread ID;
- *attr* permette di specificare gli attributi del thread (se *attr* = NULL, gli attributi sono quelli di default);
- *start\_func* è l'indirizzo della funzione di avvio;
- *arg* è l'indirizzo dell'argomento accettato dalla funzione di avvio;
- restituisce 0 in caso di successo, un intero positivo – secondo le convenzioni di *<sys/errno.h>* – in caso di errore.

# Pthreads

---

```
typedef void (*thread_start)(void *);

int pthread_create(pthread_t *tid,
 const pthread_attr_t *attributes
 thread_start start,
 void *argument);
```

- Restituisce 0 se OK, un codice d'errore altrimenti
- tid = argomento di ritorno, conterrà il tid del nuovo thread
- attributes = attributi del thread (vedere dopo)
- start = indirizzo della funzione da cui partire
- argument = l'argomento passato alla funzione start

# Pthreads

---

```
void pthread_exit(void *status);
```

- termina il thread corrente, con valore di uscita status
- altri thread possono raccogliere il valore di uscita usando `pthread_join` (vedere slide successiva)
- fare attenzione che i dati puntati da `ret` sopravvivano alla terminazione del thread!
  - `status` non deve puntare allo stack (no variabili locali)
  - Ok uso di variabili globali o allocate dinamicamente

# Pthreads

---

Un thread può attendere per la terminazione di un altro thread relativo allo stesso processo:

```
#include <pthread.h>

int pthread_join (pthread_t *tid, void **status);
```

- *tid* è l'ID del thread del quale si vuole attendere la terminazione;
- *status* punta al valore restituito dal thread per cui si è atteso, indicante il suo stato di terminazione (se *status* = NULL, tale stato non viene restituito);
- restituisce 0 in caso di successo, un intero positivo – secondo le convenzioni di *<sys/errno.h>* – in caso di errore.

# Pthreads

---

```
/* thread_create: stampa i TID del main thread e di due altri
thread */

#include <pthread.h>
#include <stdio.h>
#include <errno.h>

void *start_func(void *arg) /* funzione di avvio */
{
 printf("%s", (char *)arg);
 printf(" and my TID is: %d\n", (int)pthread_self());
}

int main(void)
{
 int en;
 pthread_t tid1, tid2;
 char *msg1 = "Hello world, I am thread #1";
 char *msg2 = "Hello world, I am thread #2";

 printf("The launching process has PID:%d\n", (int)getpid());

 printf("The main thread has TID:%d\n", (int)pthread_self());
```

# Pthreads

---

```
/* thread_create: stampa i TID del main thread e di due altri
thread */

#include <pthread.h>
#include <stdio.h>
#include <errno.h>

void *start_func(void *arg) /* funzione di avvio */
{
 printf("%s", (char *)arg);
 printf(" and my TID is: %d\n", (int)pthread_self());
}

int main(void)
{
 int en;
 pthread_t tid1, tid2;
 char *msg1 = "Hello world, I am thread #1";
 char *msg2 = "Hello world, I am thread #2";

 printf("The launching process has PID:%d\n", (int)getpid());

 printf("The main thread has TID:%d\n", (int)pthread_self());
```

# Pthreads

---

```
/* crea il 1mo thread */
if ((en = pthread_create(&tid1, NULL, start_func, msg1)!=0))
 errno=en, perror("pthread_create"), exit(1);

/* crea il 2ndo thread */
if ((en = pthread_create(&tid2, NULL, start_func, msg2)!=0))
 errno=en, perror("pthread_create"), exit(2);

/* attende per il 1mo */
if ((en = pthread_join(tid1, NULL)!=0))
 errno=en, perror("pthread_join"), exit(1);

/* attende per il 2ndo */
if ((en = pthread_join(tid2, NULL)!=0))
 errno=en, perror("pthread_join"), exit(2);

return 0;
}
```

# Pthreads

---

```
typedef struct foo{
 int a;
 int b;
} myfoo;

myfoo test; // Variabile GLOBALE

void stampa(char *st, struct foo *test){
 printf("%s: tid=%d a=%d b=%d\n", st, pthread_self(), test->a, test->b);
}

void *fun1(void *arg){
 myfoo test2 = {1,2}; // Variabile LOCALE
 printf("%s %d\n", arg, pthread_self());
 stampa(arg, &test2);
 pthread_exit((void *)&test2);
}
```

# Pthreads

---

```
void *fun2(void *arg){
 test.a = 3;
 test.b = 4; // Variabile GLOBALE
 printf("%s %d\n", arg, pthread_self());
 stampa(arg, &test);
 pthread_exit((void *)&test);
}

void *fun3(void *arg){
 myfoo *test3;
 test3=malloc(sizeof(struct foo)); // Variabile allocata dinamicamente
 test3->a = 5;
 test3->b = 6;
 printf("%s %d\n", arg, pthread_self());
 stampa(arg, test3);
 pthread_exit((void *)test3); //c
}
```

# Pthreads

---

```
int main(void){
 char st[100];
 pthread_t tid1;
 pthread_t tid2;
 pthread_t tid3;

 myfoo *b; // PUNTATORE alla struttura (non allocata)

 pthread_create(&tid1, NULL, fun1, "Thread 1"); // Locale
 pthread_join(tid1, (void *)&b);
 stampa("Master ", b);

 pthread_create(&tid2, NULL, fun2, "Thread 2"); // Globale
 pthread_join(tid2, (void *)&b);
 stampa("Master ", b);

 pthread_create(&tid3, NULL, fun3, "Thread 3"); // Dinamica
 pthread_join(tid3, (void *)&b);
 stampa("Master ", b);
}
```

# Pthreads

---

```
Thread 1: 1077283760
 // Locale
Thread 1: a=1 b=2
Master : a=1075156600 b=1077281896
```

```
Thread 2: 1077283760
 // Globale
Thread 2: a=3 b=4
Master : a=3 b=4
```

```
Thread 3: 1077283760
 // Dinamica
Thread 3: a=5 b=6
Master : a=5 b=6
```

# Windows Multithreaded in C

```
#include <windows.h> Windows API
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
 DWORD Upper = *(DWORD*)Param;
 for (DWORD i = 0; i <= Upper; i++)
 Sum += i;
 return 0;
}

int main(int argc, char *argv[])
{
 DWORD ThreadId;
 HANDLE ThreadHandle;
 int Param;

 if (argc != 2) {
 fprintf(stderr,"An integer parameter is required\n");
 return -1;
 }
 Param = atoi(argv[1]);
 if (Param < 0) {
 fprintf(stderr,"An integer >= 0 is required\n");
 return -1;
 }
}
```

DWORD 32-bit unsigned int

LPVOID puntatore void

Funzione di avvio del thread come in Pthread

# Windows Multithreaded in C

---

```
/* create the thread */ Creazione del thread come in
ThreadHandle = CreateThread(Pthread
 NULL, /* default security attributes */
 0, /* default stack size */
 Summation, /* thread function */
 &Param, /* parameter to thread function */
 0, /* default creation flags */
 &ThreadId); /* returns the thread identifier */

if (ThreadHandle != NULL) { Attesa del thread come in Pthread
 /* now wait for the thread to finish */
 WaitForSingleObject(ThreadHandle, INFINITE);

 /* close the thread handle */
 CloseHandle(ThreadHandle);

 printf("sum = %d\n", Sum);
}
}
```

# Thread in Java

---

- thread in Java gestiti dalla JVM
- Tipicamente implementati con il modello di thread fornito dal sistema operativo utilizzato
  - Esempio in Unix Pthreads
- I thread java possono essere creati in due modi:
  - Estendendo la classe Thread con override del metodo run()

```
public class ThreadA extends Thread{
 @Override
 public void run(){
 System.out.println("Codice del thread A");
 }
}
```

- Implementando l'interfaccia Runnable

```
public interface Runnable
{
 public abstract void run();
}
```

# Thread in Java

---

- thread in Java gestiti dalla JVM
- Tipicamente implementati con il modello di thread fornito dal sistema operativo utilizzato
  - Esempio in Unix Pthreads
- I thread java possono essere creati in due modi:
  - Estendendo la classe Thread con override del metodo run()

```
public class Thread1 extends Thread{
 @Override
 public void run(){
 System.out.println("Codice del thread 1");
 }
}
```

```
public class ProvaThread1 {
 public static void main(String[] args) {
 Thread1 thread1 = new Thread1();
 thread1.start();
 }
}
```

thread1.start(); rende il thread eleggibile per l'esecuzione.

Una volta in esecuzione parte il run()

# Thread in Java

---

- thread in Java gestiti dalla JVM
- Tipicamente implementati con il modello di thread fornito dal sistema operativo utilizzato
  - Esempio in Unix Pthreads
- I thread java possono essere creati in due modi:
  - Estendendo la classe Thread con override del metodo run()
  - Implementando l'interfaccia Runnable

```
public interface Runnable
{
 public abstract void run();
}
```

```
public class Thread2 implements Runnable{
 public void run(){
 System.out.println("Codice del thread 2");
 }
}
```

# Thread in Java

---

- thread in Java gestiti dalla JVM
- Tipicamente implementati con il modello di thread fornito dal sistema operativo utilizzato
  - Esempio in Unix Pthreads
- I thread java possono essere creati in due modi:
  - Estendendo la classe Thread con override del metodo run()
  - Implementando l'interfaccia Runnable

```
public class Thread2 implements Runnable{
 public void run(){
 System.out.println("Codice del thread 2");
 }
}

public class ProvaThread2 {
 public static void main(String[] args) {
 Thread thread = new Thread(new Thread2());
 thread.start();
 }
}
```

# Java Multithreaded Program

```
class Sum
{
 private int sum;

 public int getSum() {
 return sum;
 }

 public void setSum(int sum) {
 this.sum = sum;
 }
}

class Summation implements Runnable
{
 private int upper;
 private Sum sumValue;

 public Summation(int upper, Sum sumValue) {
 this.upper = upper;
 this.sumValue = sumValue;
 }

 public void run() {
 int sum = 0;
 for (int i = 0; i <= upper; i++)
 sum += i;
 sumValue.setSum(sum);
 }
}
```

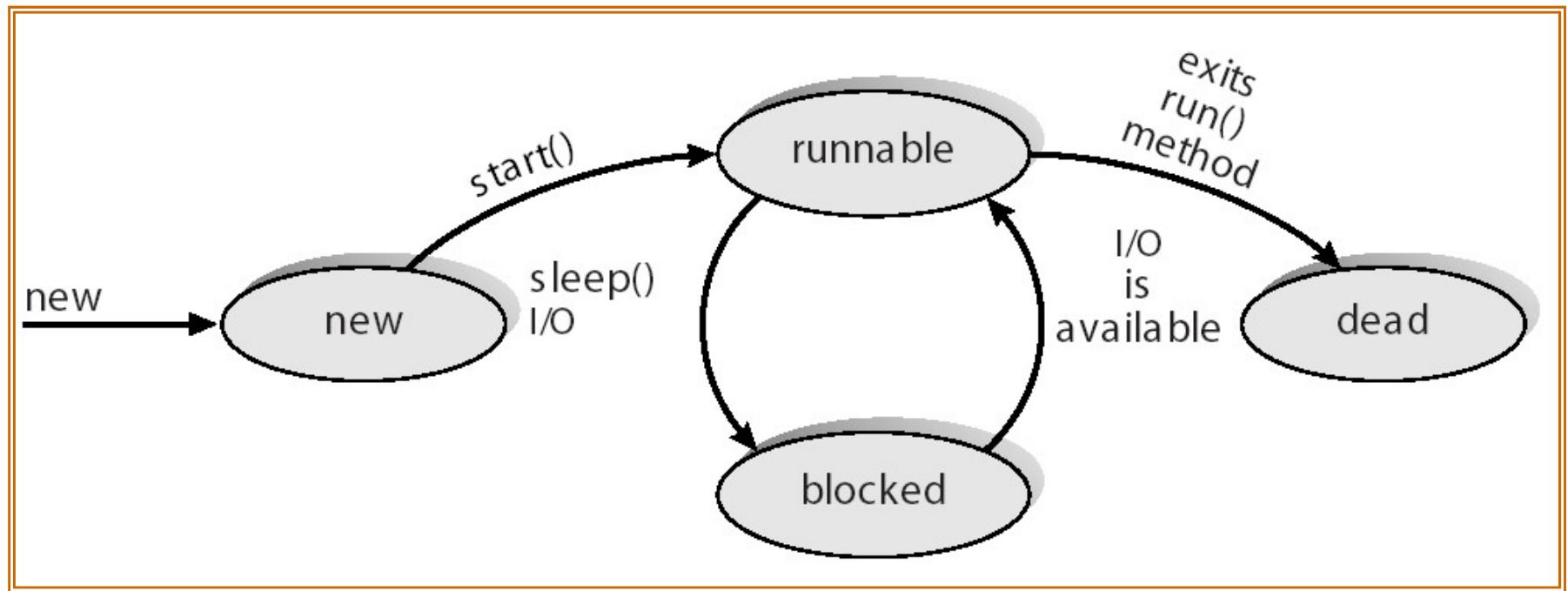
Summation implementa  
l'interfaccia Runnable

Il metodo run implementa  
il thread

# Java Multithreaded Program

```
public class Driver
{
 public static void main(String[] args) {
 if (args.length > 0) {
 if (Integer.parseInt(args[0]) < 0)
 System.err.println(args[0] + " must be >= 0.");
 else {
 Sum sumObject = new Sum();
 int upper = Integer.parseInt(args[0]);
 Thread thrd = new Thread(new Summation(upper, sumObject));
 thrd.start(); Creazione del thread
 try { Lancio con start
 thrd.join(); Raccolto con join
 System.out.println
 ("The sum of "+upper+" is "+sumObject.getSum());
 } catch (InterruptedException ie) { }
 }
 }
 else
 System.err.println("Usage: Summation <integer value>"); }
 }
```

# Java Multithreaded Program



# Threading implicito

---

- Tecnica sempre più popolare al crescere della complessità dei sistemi multithread
  - Creazione e gestione dei thread fatta da compilatori e librerie run-time invece che dai programmatori
  - Il programmatore identifica dei **task** non dei thread
- 
- Tre metodi considerati
    - Thread Pools
    - OpenMP
    - Grand Central Dispatch
- 
- Altri metodi includono
    - Microsoft Threading Building Blocks (TBB)
    - `java.util.concurrent` package

# Pool di Thread

---

- Si crea un numero di thread in attesa in un pool
  - Richiesto un thread per servizio, se non c'è attesa.
- Vantaggio:
  - Solitamente più veloce nel servire una richiesta quando il thread esiste già
  - Il numero dei thread per l'applicazione applicazione è limitato dalla dimensione del pool
  - Separa il task da eseguire dal meccanismo per creare il task e permette differenti strategie di esecuzione del task
    - ▶ i.e. I task possono essere schedulati per avviarsi periodicamente
- Windows API per supportare i thread pool:

```
DWORD WINAPI PoolFunction(VOID Param) {
 /*
 * this function runs as a separate thread.
 */
}
```

# Pool di Thread

---

```
DWORD WINAPI PoolFunction(VOID Param) {
 /* This function runs as a separate thread */
}
```

- Un puntatore a `PoolFunction()` è passato ad una delle funzioni delle API nel thread pool e un thread dal pool la esegue.
- Una di queste funzioni è `QueueUserWorkItem` con 3 parametri
  - `LPTHREAD_START_ROUTINE Function` – il puntatore alla funzione da eseguire come thread
  - `PVOID Param` – i parametri passati alla funzione
  - `ULONG Flags` – flag che indicano come il thread pool deve creare e gestire l'esecuzione del thread.

Esempio di chiamata:

```
QueueUserWorkItem(&PoolFunction, NULL, 0);
```

che porta uno dei thread del pool ad invocare la `PoolFunction()`

# OpenMP

---

- Direttive di compilazione ed API per C, C++, FORTRAN
- Supporto per programmazione parallela in ambienti in shared-memory
- Identifica **regioni parallele** – blocchi di codice parallelizzabili

**#pragma omp parallel**

Crea tanti thread quanti sono i core

Esegui il loop in parallelo

```
#pragma omp parallel for
 for(i=0;i<N;i++) {
 c[i] = a[i] + b[i];
}
```

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
 /* sequential code */

 #pragma omp parallel
 {
 printf("I am a parallel region.");
 }

 /* sequential code */

 return 0;
}
```

# Grand Central Dispatch

---

- Tecnologia Apple per Mac OS X e iOS
  - Esensioni di C, C++, API, e librerie run-time
  - Permette l'identificazione di sezioni parallele (task)
  - Gestisce i dettagli del threading
- 
- Blocchi in sezioni “^{ }” - ^{ printf("I am a block") ; }
  - Gestisce code di dispatch
    - Blocchi in coda di dispatch
    - Assegna i blocchi a un thread disponibile nel thread pool quando è rimosso dalla coda

# Grand Central Dispatch

---

- Due tipi di code di dispatch:

- seriali – blocchi rimossi in ordine FIFO e attesa di completamento, coda per processo, chiamata **main queue**
  - ▶ Programmatori possono creare ulteriori code seriali nel programma
- concorrenti – rimossi in ordine FIFO ma più rimozioni alla volta
  - ▶ Tre code di sistema con priorità low, default, high

```
dispatch_queue_t queue = dispatch_get_global_queue
 (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_async(queue, ^{ printf("I am a block."); });
```

# Problematiche Threading

---

- Semantica delle chimate **fork()** ed **exec()**
- Signal handling
  - Sincrono e asincrono
- Cancellazione thread
  - Asincrono o deferred
- Thread-local storage
- Attivazione Scheduler

# Semantica di fork() ed exec()

---

- **fork()** duplica solo il thread chiamante o tutti i thread del processo?
  - Per alcune versioni di UNIX due versioni di fork
- **exec()** funziona normalmente – sovrascrive il processo in esecuzione inclusi i thread

# Signal Handling

---

**Segnali** in UNIX notifica eventi a processi

- Un **signal handler** usato per processare i segnali
  - 1. Generato da evento
  - 2. Mandato ad un processo
  - 3. Signal handler:
    - 1. default
    - 2. user-defined
- Ogni segnale ha un **default handler** che il kernel esegue
  - **User-defined signal handler** sovrascrive il default
  - Per single-threaded, segnali mandati al processo

# Signal Handling

---

- Un “segnale” e’ un interrupt “software”
  - La terminologia corretta e’ “exception” mentre “interrupt” e’ usata solo per gli interrupt “hardware”
- Consente la comunicazione asincrona tra processi e/o tra device e processo
- Ogni segnale ha un proprio nome
  - Tutti i nomi cominciano per “SIG”
  - Definiti in <signal.h>
  - Associati ad interi positivi

# Segnali (Unix)

---

- **Caratteristiche dei segnali**
  - Ogni segnale ha un identificatore
    - Identifieri di segnali iniziano con i tre caratteri SIG
    - Es. **SIGABRT** è il segnale di abort
  - Numero segnali: 15-40, a seconda della versione di UNIX
    - POSIX: 18
    - Linux: 38
  - I nomi simbolici corrispondono ad un intero positivo
    - Definizioni di costanti in **bits/signum.h**

# Signal (Unix)

---

## Pressione di tasti speciali sul terminale

- Es: Premere il tasto **ctrl-c** genera il segnale **SIGINT**

## Eccezioni hardware

- Divisione per 0 (**SIGFPE**)
- Riferimento non valido a memoria (**SIGSEGV**)
- L'interrupt viene generato dall'hardware, e catturato dal kernel; questi invia il segnale al processo in esecuzione

## System call **kill**

- Permette di spedire un segnale ad un altro processo
- Limitazione: uid del processo che esegue **kill** deve essere lo stesso del processo a cui si spedisce il segnale, oppure 0 (root)

# Segnali (Unix)

---

| <i>nome</i> | <i>significato</i>                | <i>default</i>  |
|-------------|-----------------------------------|-----------------|
| SIGINT      | interruzione da tastiera (Ctrl-c) | terminare       |
| SIGSTOP*    | stop al processo                  | <i>fermare</i>  |
| SIGKILL*    | terminazione forzata              | terminare       |
| SIGQUIT     | quit da tastiera (Ctrl-y)         | terminare       |
| SIGTERM     | terminazione da tastiera (Ctrl-\) | terminare       |
| SIGCHLD     | figlio terminato o fermato        | <i>ignorare</i> |
| SIGALRM     | suona la sveglia!                 | terminare       |
| SIGSEGV     | segmentation fault                | terminare       |
| SIGUSR1     | a disposizione dell'utente        | terminare       |

# Segnali (Unix)

---

- I segnali vengono inviati in modo asincrono.
  - Non e' possibile sapere quando il processo ricevera' un segnale.
- E' possibile indicare al kernel *l'azione da intraprendere* quando un segnale e' generato per un processo:
  - Ignora: Valida per quasi tutti i segnali tranne SIGKILL e SIGSTOP.
  - "Catch" del segnale: Indicare una procedura da eseguire (signal handler). Ad esempio:
    - SIGCHLD: esegui le operazioni associate alla termiazione di un figlio
    - SIGINT: (CTRL-C) "cancella file temporanei"...
    - **Non e' possibile intercettare SIGKILL o SIGSTOP.**
  - Default: Eseguire l'azione di default.

# Segnali (Unix)

---

- Un handler (gestore) è una funzione del tipo:

```
void funzione(int num_segnale) {
 printf("%d", num_segnale);
}
```

Una volta che l'handler termina, si torna al punto in cui il programma era stato interrotto.

# Segnali (Unix)

---

```
typedef void (*sighandler_t)(int);
```

```
sighandler_t signal(int signum, sighandler_t handler);
```

- `signal(SIGINT, foo)` imposta la funzione `foo` come handler del segnale SIGINT
- si puo' anche richiedere di ignorare il segnale
  - `signal(SIGINT, SIG_IGN)`
- oppure ritornare alla reazione di default
  - `signal(SIGINT, SIG_DFL)`

# Segnali (Unix)

---

```
int kill(pid_t pid, int sig);
```

- `kill(127, SIGINT)` invia il segnale SIGINT al processo il cui pid e' 127
- restituisce 0 in caso di successo e -1 in caso di errore

# Segnali (Unix)

---

```
void foo(int num_segnale);
int main(void){

 int n=0; int buf[100];
 alarm(5);
 signal(SIGALRM,foo);

 while (n<=0){
 printf("Digitare qualcosa:\n");
 alarm(1);
 if ((n=read(STDIN_FILENO,buf,10))<0)
 perror("Read error");
 alarm(0);
 }
}

void foo(int num_segnale) {
 alarm(1);
 printf("Vuoi muoverti a digitare qualcosa ???\n");
}
```

# Signal Handling

---

Se multi-threaded?

- ❑ Invia il segnale al thread a cui si applica
- ❑ Invia a tutti i thread del processo
- ❑ Invia a specifici thread del processo
- ❑ Assegna ad un thread sepecifico il compito di recevere tutti i segnali del processo

# Signal Handling

---

```
int pthread_kill(pthread_t tid, int signo);
```

- manda il segnale signo al thread specificato da tid
  - se e' impostato un handler, viene eseguito nel thread tid
  - se non e' impostato un handler, e il comportamento di default e' di terminare il processo, vengono comunque terminati tutti i thread
- restituisce 0 se OK, un codice d'errore altrimenti

# Signal Handling

---

```
signal(SIGUSR1, usr1);
pthread_create(&tid1, NULL, fun, "Thread 1");
pthread_create(&tid2, NULL, fun, "Thread 2");
pthread_create(&tid3, NULL, fun, "Thread 3");
sleep(1);
pthread_kill(tid1, SIGUSR1); (USR1 = User defined signal)
pthread_kill(tid2, SIGUSR1);
pthread_kill(tid3, SIGUSR1);

sigemptyset(&set); // Configura la maschera SOLO nel master thread
sigaddset(&set, SIGUSR1);
sigprocmask(SIG_SETMASK, &set, NULL);
sleep(1);
while (i++<10){
 sleep(1);
 kill(pid, SIGUSR1); // il segnale e' intercettato da un thread
}
```

# Signal Handling

---

Thread id=1077283760 ricevuto segnale  
Thread id=1079385008 ricevuto segnale  
Thread id=1081486256 ricevuto segnale  
Thread id=1077283760 ricevuto segnale

# Cancellazione Thread

---

- Terminare un thread prima che sia finito
- Il thread da cancellare è il **target thread**
- Due approcci:
  - **Asynchronous cancellation** termina il target thread subito
  - **Deferred cancellation** permette al target thread di controllare periodicamente se deve essere cancellato
- Codice Pthread per creare e cancellare un thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

# Cancellazione Thread

---

- La cancellazione effettiva depende dallo stato del thread

| Mode         | State    | Type         |
|--------------|----------|--------------|
| Off          | Disabled | –            |
| Deferred     | Enabled  | Deferred     |
| Asynchronous | Enabled  | Asynchronous |

- Se il thread ha la cancellazione disabilitata questa rimane pending finché il thread non la consente
- Il default è deferred:
  - Cancellazione avviene solo quando il thread raggiuge un **cancellation point**
    - ▶ Allora **cleanup handler** è invocato
- Su Linux la cancellazione dei thread gestita con segnali

# Cancellazione Thread (Posix)

---

- In ogni istante, un thread può essere cancellabile o non cancellabile
- Quando partono tutti i thread sono cancellabili
- Quando un altro thread chiama `pthread_cancel`
  - se il thread è cancellabile, viene cancellato
  - se non è cancellabile, la richiesta di cancellazione viene memorizzata, in attesa che il thread diventi cancellabile

# Cancellazione Thread (Posix)

---

```
int pthread_setcancelstate(int state, int *oldstate);
```

- imposta la cancellabilità a state e restituisce la vecchia cancellabilità in oldstate
- state e oldstate possono assumere i valori:
  - PTHREAD\_CANCEL\_ENABLE
  - PTHREAD\_CANCEL\_DISABLE
- restituisce 0 se OK, un codice d'errore altrimenti

# Thread-Local Storage

---

- **Thread-local storage (TLS)**
  - Ogni thread con la sua copia dei dati
  - Utile quando non si controlla direttamente la creazione dei thread (i.e., usando i thread pool)
- Differenza rispetto a variabili locali
  - Variabili locali visibili solo per una singola invocazione di funzione
  - TLS visibile per più invocazioni di funzione
- Simile ai dati **static**
  - TLS unico per ogni thread

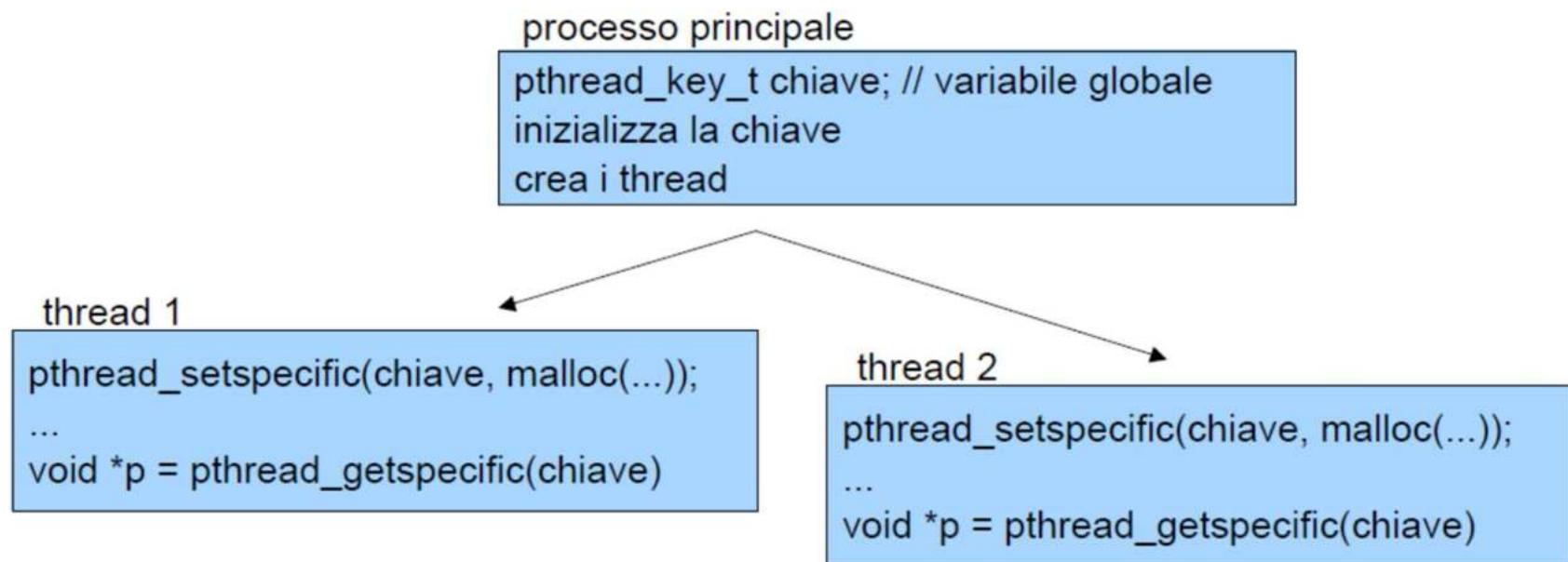
# Thread-Specific Data (Posix)

---

- Ogni thread possiede un'area di memoria privata, la TSD area, indicizzata da chiavi
- La TSD area contiene associazioni tra le chiavi ed un valore di tipo void\*
  - diversi thread possono usare le stesse chiavi ma i valori associati variano di thread in thread
  - inizialmente tutte le chiavi sono associate a NULL

# Thread-Specific Data (Posix)

- associare a una stessa chiave, dati diversi per ciascun thread



# Thread-Specific Data (Posix)

---

- `int pthread_key_create(...)`
  - per creare una chiave TSD
- `int pthread_key_delete(...)`
  - per deallocare una chiave TSD
- `int pthread_setspecific(...)`
  - per associare un certo valore ad una chiave TSD
- `void * pthread_getspecific(...)`
  - per ottenere il valore associato ad una chiave TSD

# Thread-Specific Data (Posix)

---

```
int pthread_key_create(pthread_key_t *key,
 void (*destructor)(void *));
```

- crea una chiave per dati privati
- key è l'indirizzo della chiave da inizializzare
- destructor è un puntatore alla funzione distruttore che deve essere chiamata alla terminazione di un thread (`pthread_exit()`)
- restituisce 0 se OK, un codice d'errore altrimenti

# Thread-Specific Data (Posix)

---

```
int pthread_setspecific(pthread_key_t *key,
 const void* val);
```

- associa l'indirizzo val alla chiave key, per il thread chiamante
  - restituisce 0 se OK, un codice d'errore altrimenti
- 

```
void* pthread_getspecific(pthread_key_t *key);
```

- restituisce l'indirizzo associato alla chiave key nel thread chiamante
  - restituisce NULL se nessun indirizzo è stato associato a key

# Thread-Specific Data (Posix)

---

```
#include ...

static pthread_key_t thread_log_key; /* tsd key per thread */

void write_to_thread_log (const char* message); //Scrive log
void close_thread_log (void* thread_log); //Chiude file log
void* thread_function (void* args); //Eseguita dai thread

int main() {
 // Crea una chiave da associare al log Thread-Specific
 // Crea 5 thread che facciano il lavoro
 // Aspetta che tutti finiscano
 return 0;
}
...
```

# Thread-Specific Data (Posix)

```
...
int main() {
 int i;
 pthread_t threads[5];
 // Crea una chiave da associare al puntatore TSD al log file
 pthread_key_create(&thread_log_key, close_thread_log);

 for (i = 0; i < 5; ++i) // thread che faccia il lavoro
 pthread_create(&(threads[i]), NULL, thread_function, NULL);

 for (i = 0; i < 5; ++i) // Aspetta che tutti finiscano
 pthread_join(threads[i], NULL);
 return 0;
}
```

# Thread-Specific Data (Posix)

```
...
void write_to_thread_log (const char* message) {
 FILE* thread_log = (FILE*)pthread_getspecific(thread_log_key);
 fprintf (thread_log, "%s\n", message);
}

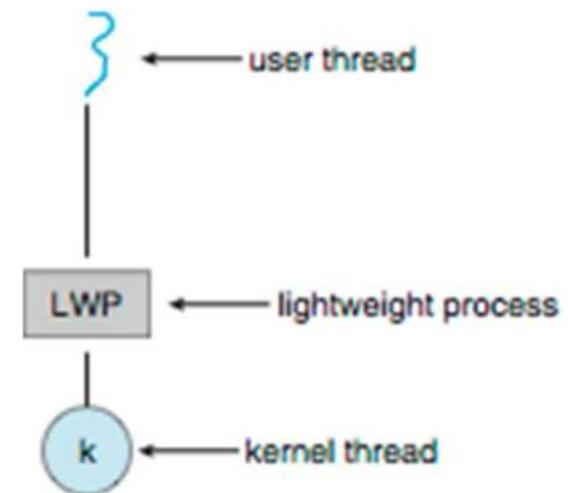
void close_thread_log (void* thread_log) {
 fclose ((FILE*) thread_log);
}

void* thread_function (void* args) {
 char thread_log_filename[20];
 FILE* thread_log;
 sprintf(thread_log_filename,"thread%d.log", (int)pthread_self ());
 thread_log = fopen (thread_log_filename, "w");
 /* Associa la struttura FILE TSD a thread_log_key. */
 pthread_setspecific (thread_log_key, thread_log);

 write_to_thread_log ("Thread starting.");
 /* Fai altro lavoro qui... */ return NULL;
}
```

# Attivazioni Scheduler

- Sia modelli Many-to-Many che Two-level richiedono di mantenere un numero appropriato di kernel thread allocati per l'applicazione
- Tipicamente usata una struttura dati intermedia tra user e kernel thread – **lightweight process (LWP)**
  - Come un processore virtuale su cui il processo può schedulare l'esecuzione di user thread
  - Ogni LWP associato ad un kernel thread
- Le attivazioni dello scheduler forniscono **upcalls** – un meccanismo di comunicazione tra il kernel e **upcall handler** nella libreria thread
- Questa comunicazione consente ad un'applicazione di mantenere il corretto numero di kernel thread



# Operating System Examples

---

- Windows Threads
- Linux Threads

# Windows Threads

---

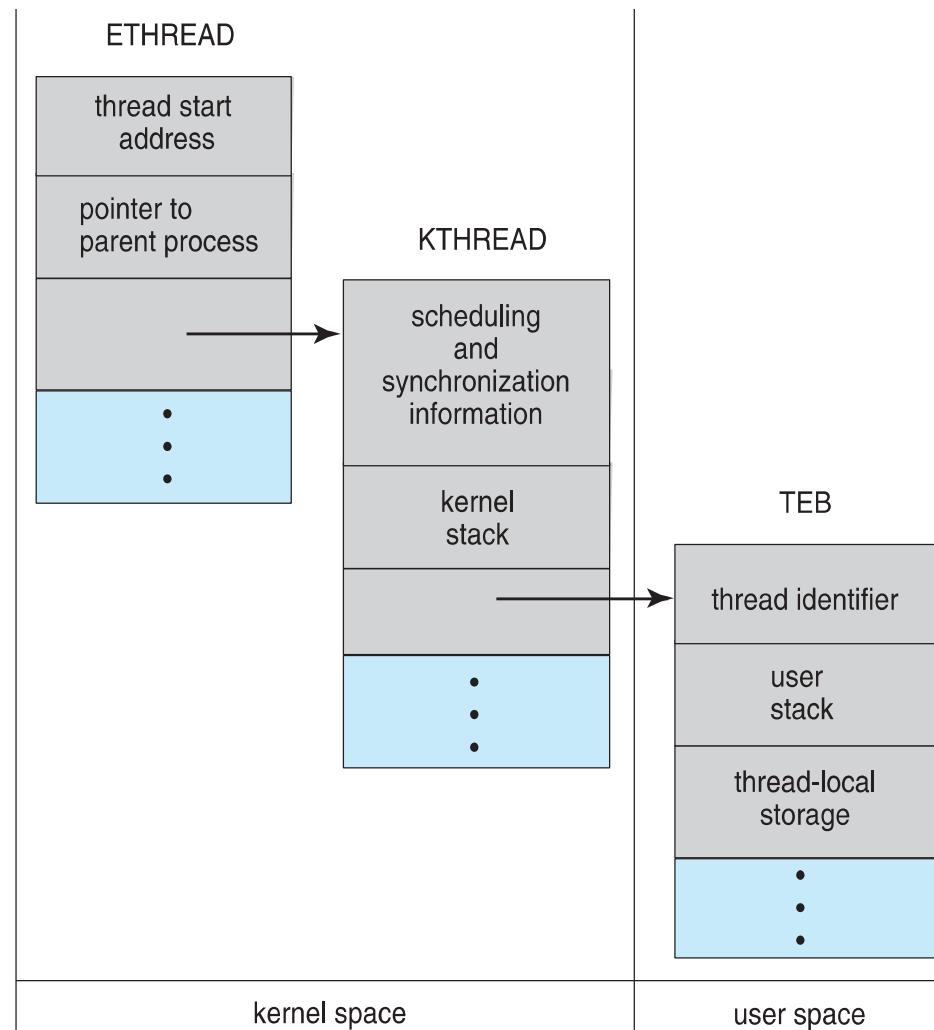
- Implementa il mapping one-to-one, kernel-level
- Ogni thread contiene
  - Un thread id
  - Register set rappresenta lo stato del processore
  - User e kernel stacks separati per i thread eseguiti in user mode o kernel mode
  - Area di private data storage usati da run-time libraries e dynamic link libraries (DLLs)
- Il register set, stacks, e private storage area sono detti il **context** del thread

# Windows Threads

---

- La struttura dati primaria del thread include:
  - ETHREAD (executive thread block) – include puntatore al processo al quale il thread appartiene e a KTHREAD nel kernel space
  - KTHREAD (kernel thread block) – info di scheduling e sincronizzazione, kernel-mode stack, puntatore a TEB nel kernel space
  - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, nello user space

# Windows Threads Data Structures



# Thread Linux

---

- Linux si riferisce a **tasks** piuttosto che a **threads**
  - Unificando la gestione di processi e thread
- Thread creation è gestita dalla chiamata di sistema **clone()**
  - Usata per implementare **pthread\_create**
- **clone()** permette ad un child task di condividere l'address space del parent task (processo)
  - Flag di controllo:

| flag          | meaning                            |
|---------------|------------------------------------|
| CLONE_FS      | File-system information is shared. |
| CLONE_VM      | The same memory space is shared.   |
| CLONE_SIGHAND | Signal handlers are shared.        |
| CLONE_FILES   | The set of open files is shared.   |

- **struct task\_struct** punta alle strutture dati del processo (condivise o uniche)

# Unix Shell

- La shell è un interprete di comandi
- Si interpone tra l'utente ed il sistema operativo
- In sistemi Unix qualsiasi operazione può essere eseguita da una sequenza di comandi shell

Tra le Shell più utilizzate ci sono:

- |                       |           |             |
|-----------------------|-----------|-------------|
| • Bourne shell,       | /bin/sh   | (Bell Labs) |
| • Bourne-again shell, | /bin/bash | (Linux)     |
| • Cshell,             | /bin/csh  | (Berkeley)  |
| • Korn shell,         | /bin/ksh  | (Bell Labs) |
| • TENEX C shell       | /bin/tcsh | (BBN tech)  |

Il file /etc/shells contiene l'elenco delle shell installate dall'amministratore e disponibili a tutti gli utenti.

# Shell Interattiva

- Comunicazione tra utente e shell avviene tramite comandi o script:
- Nome comando built-in oppure
- Nome di un file eseguibile oppure
- Nome di Script, cioè file ASCII presente nel sistema dotato del premesso di esecuzione.

# Classi di Comandi

- reperire informazioni su comandi, programmi, file....
- gestire filesystem
- operare su file e directory
- elaborare testi
- sviluppare software
- operare in remoto
- ....(comandi “di utilità” vari)
- amministrare il sistema
  - utenti e gruppi
  - dispositivi
  - software

# Formato dei comandi

**comando [ argomento ...]**

Gli argomenti possono essere:

- opzioni o flag (-)
- parametri

separati da almeno un separatore (di default il carattere spazio)

L'ordine delle **opzioni** e', in genere, irrilevante

L'ordine dei **parametri** e', in genere, rilevante

**Attenzione: Unix e' CASE SENSITIVE**

# Esempi di Comandi

| Comando               | Significato                                                         |
|-----------------------|---------------------------------------------------------------------|
| <b>ls</b>             | visualizza la lista di file nella directory, come il comando in DOS |
| <b>dir</b>            | cambia directory                                                    |
| <b>cd directory</b>   | cambia password                                                     |
| <b>passwd</b>         | visualizza il tipo di file o il tipo di file con nome filename      |
| <b>file filename</b>  | riversa il contenuto di textfile sullo screen                       |
| <b>cat textfile</b>   | visualizza la directory di lavoro corrente                          |
| <b>pwd</b>            | lascia la sessione                                                  |
| <b>exit or logout</b> | leggi pagine manuale su <b>command</b>                              |
| <b>man command</b>    | leggi pagine Info su <b>command</b>                                 |
| <b>...</b>            |                                                                     |

# Gestione delle Directory

**mkdir**

Crea una directory

**rmdir**

Elimina una directory vuota

**pwd**

Emette il percorso della directory corrente

**ls**

Elenca il contenuto di una o più directory

# mkdir (make directory)

mkdir [options] directory

Crea una directory secondo le opzioni

Alcune opzioni:

-m mode Indica le protezioni per la directory da creare

-p Crea, se assente, l'intero percorso indicato

Esempi:

mkdir -m 600 tracce\_esami

mkdir -p Iso/11-12/lezione1/

# rmdir (remove directory)

rmdir [options] directory

Rimuove la directory indicata.

Le directory possono essere rimosse se

- (a) sono vuote e
- (b) possibile scrivere nella directory padre.

Opzioni:            -p        Rimuove l'intero percorso indicato

Esempi:

Crea la directory lezione1 nella directory corrente

rmdir lezione1

Rimuove, dalla directory corrente, il path indicato

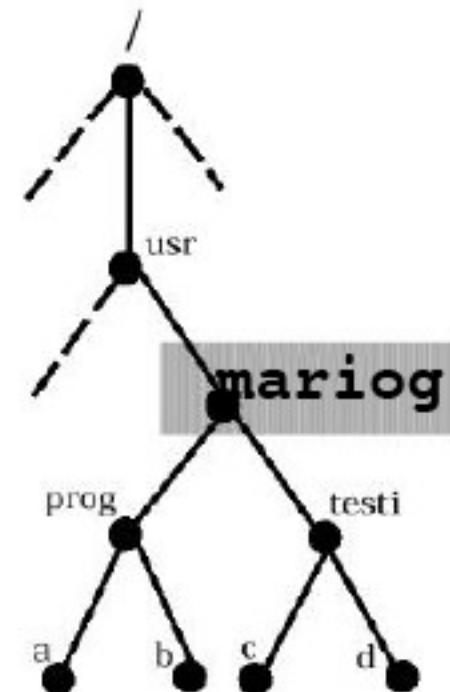
rmdir -p SO1/lezione1/

# pwd (print working directory)

**pwd** "printworking directory"

stampa il path della directory corrente

```
% pwd
/usr/mariog
%
```



# cd (change directory)

cd [directory]

Cambia la directory corrente a quella indicata.

Se non viene passato nessun argomento, la directory corrente diventa la home directory.

Esempio:

```
lso:~>cd
```

```
lso:~>pwd
```

```
/home/lso
```

```
lso:~>cd esempio/esempiocd
```

```
lso:~>pwd
```

```
/home/lso/esempio/esempiocd
```

# ls (list)

**ls [options] [directory]**

Elenca i file contenuti nella directory specificata.

Se la directory non e' indicata, viene elencato il contenuto della directory corrente.

Alcune opzioni:

- a Elenca anche i file nascosti
- l Formato esteso con informazioni su modo, proprietario, dimensione, etc dei file
- s fornisce la dimensione in blocchi dei file
- t lista i file nell'ordine di modifica (prima il file modificato per ultimo)
- 1 elenca i file in una singola colonna
- F aggiunge / al nome delle directory e \* al nome dei file eseguibili
- R si chiama ricorsivamente su ogni subdirectory

# Is – campi del formato esteso

| Totale dimensione occupata (in blocchi) |                     |                   |             |                    |          |  |
|-----------------------------------------|---------------------|-------------------|-------------|--------------------|----------|--|
|                                         | Riferimenti al file | Dimensione (byte) | Nome        |                    |          |  |
| <b>so1:~&gt;ls -l</b>                   |                     |                   |             |                    |          |  |
| total 12                                |                     |                   |             |                    |          |  |
| -rw-rw-r--                              | 1 lso               | lso               | 10          | Mar 4 13:29        | a        |  |
| -rw-rw-r--                              | 1 lso               | lso               | 10          | Mar 4 14:12        | b        |  |
| <b>drwxrwxr-x</b>                       | <b>2 lso</b>        | <b>lso</b>        | <b>4096</b> | <b>Mar 4 14:29</b> | <b>c</b> |  |

Diagramma di un output del comando ls -l con annotazioni:

- Tipo:** Indica il tipo di file (d, l, c, b, -).
- Permessi:** Indica i permessi di lettura (r), scrittura (w) e esecuzione (x) per utente, gruppo e altri.
- Riferimenti al file:** Indica il numero di riferimenti al file.
- Proprietario:** Indica il proprietario del file.
- Gruppo primario:** Indica il gruppo primario del file.
- Dimensione (byte):** Indica la dimensione del file in byte.
- Data ultima modifica:** Indica la data e l'ora della ultima modifica del file.
- Nome:** Indica il nome del file.

Legenda per i simboli dei permessi:  
(r)ead, (w)rite, e(x)ecute, (s)et uid bit

# cp (copy)

**cp [options] source... target**

copia un file in un altro file oppure uno o più file in una directory

Se vengono specificati solo i nomi di due file, il primo viene copiato sul secondo

Se vengono specificati solo due nomi, e se il secondo nome indicato è una directory, source viene copiato con lo stesso nome nella directory target. Se source è una directory, la copia avviene solo con opzioni particolari.

Se vengono indicati più di due nomi, il file target deve essere una directory e vengono generate le copie dei source in target. In mancanza di opzioni particolari, le directory non vengono copiate.

Alcune opzioni

-r se source e target sono directory, copia ricorsivamente source, i suoi file e le sue subdirectory in target

-i opera in modo interattivo, chiedendo una conferma se la copia comporta la cancellazione di un target preesistente

# Esempi

- Copia il file pippo nella directory corrente nel file /tmp/pippo.back

```
cp pippo /tmp/pippo.back
```

- Copia il file /tmp/pippo.back e la directory dir nella directory nuovadir

```
cp -r /tmp/pippo.back dir nuovadir
```

- Copia il file pippo nel file pippo2

```
cp pippo pippo2
```

# mv (move)

**mv [options] source... destination**

rinomina (sposta) file o directory.

Se vengono specificati solo i nomi di due elementi, source viene rinominato in destination, oppure in destination/source, a seconda che destination indichi un file o una directory. Qualora destination denoti un file preesistente, questo non sarà più accessibile come tale, e non sarà più accessibile in alcun modo se destination era il suo unico nome.

Se vengono indicati più di due elementi, destination deve essere una directory, e source\_1...source\_n vengono rinominati come destination/source\_1...destination/source\_n.

Nel caso che source e destination appartengono a due diversi file system, il comando effettua un vero e proprio spostamento dati tra i due file system. In tal caso vengono spostati solo i file ordinari, quindi: né collegamenti, né directory.

Alcune opzioni:

-i il comando chiede conferma all'utente qualora destination è un file preesistente

# rm (remove)

**rm [options] file...**

elimina i file o le directory indicati come argomento.

Alcune opzioni:

-i chiede conferma prima di rimuovere ogni file

-R rimuove ricorsivamente i file e le sottodirectory

Esempi d' uso

Elimina i file pippo nella directory corrente e /tmp/pippo.back

rm pippo /tmp/pippo.back

Elimina la directory nuovadir e tutto il suo contenuto

rm -R nuovadiru

Elimina tutti i file nella directory corrente

rm \*

# cat (concatenate)

---

`cat [options] [file...]`

concatena i file indicati come argomento, visualizzandoli attraverso lo standard output

## **Alcune opzioni:**

- n      fa precedere ogni linea di output dal numero progressivo che identifica la posizione della linea nel file concatenato
- b      come l'opzione precedente, ma omette la numerazione delle linee bianche
- v      mostra anche i caratteri non stampabili, ad eccezione dei caratteri di tabulazione, nuova linea e ritorno a capo

# wc (word count)

`wc [options] [file...]`

fornisce il numero dei codici di interruzione di riga (in pratica il numero delle righe), delle parole o dei caratteri contenuti in `file`. Senza opzioni fornisce, nell'ordine suddetto, ciascuna delle precedenti informazioni.

## **Alcune opzioni:**

- c                emette solo il numero complessivo di caratteri di `file`.
- w                emette solo il numero complessivo di parole in `file`.
- l                emette solo il numero di righe in `file`.

## **Esempi di esecuzione**

```
gio$ wc which_manpage
132 239 2083 which_manpage
```

```
gio$ wc -c which_manpage
2083 which_manpage
gio$
```

# - Compilazione -

# - Fasi della compilazione -

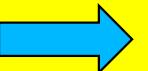
- Il compilatore GNU C in ambiente Unix, per generare il file eseguibile dal file sorgente, passa attraverso le seguenti fasi:
  - Fase 1) **Preprocessore**
    - Il preprocessore legge un sorgente C e produce in output un altro sorgente C, dopo avere **espanso in linea le macro**, e **incluso i file**. (#define MAX 100) (#include <nomefile>)
  - Fase 2) **Compilazione**
    - Traduzione del codice sorgente ricevuto dal preprocessore in codice assembly (codice molto vicino al linguaggio macchina)
  - Fase 3) **Assembler** (creazione del file oggetto)
    - Crea il codice oggetto (file con il suffisso .o)
  - Fase 4) **Linker** (creazione del file eseguibile)
    - Unisce le funzioni definite in altri file sorgenti o definite in librerie, con la funzione main() per creare il file eseguibile.

## - Il compilatore GNU **gcc** in ambiente Unix -

# - Compilazione -

- Per compilare in ambiente Unix, un programma C scritto nel file **source.c** si utilizza un compilatore GNU per sorgenti C, che viene invocato con il comando **gcc**:

Compilazione:

\$ **gcc** source.c  **a.out** (eseguibile prodotto dal compilatore)

Opzioni comando gcc:

- flag “**-o**”: \$ gcc **-o eseguibile.out sorgente.c**
- flag “**-v**” : Per sapere la versione del compilatore
- flag “**-E**”: Serve per invocare solo il preprocessore
- flag “**-c**”: Per creare il codice oggetto senza chiamare il linker
- flag “**-Wall**”: per aumentare il livello dei messaggi prodotti in fase di compilazione
- \$ **man gcc** (manuale in linea)

## - Il messaggi del compilatore -

# – I messaggi del compilatore –

- Il compilatore invia dei messaggi all' operatore:
  - **Messaggi di avvertimento** (warning messages)
    - I messaggi di avvertimento indicano la presenza di parti di codice presumibilmente mal scritte o di problemi che potrebbero avvenire in seguito, durante l'esecuzione del programma. I messaggi di avvertimento non interrompono comunque la compilazione.
  - **Messaggi d'errore** (error messages)
    - I messaggi di errore invece indicano qualcosa che deve essere necessariamente corretto e causano l'interruzione della compilazione.
- Per **inibire** tutti i messaggi di **warning** usare l'opzione **-w**
  - \$ gcc -w -o eseguibile.out source.c
- Per avere il **massimo livello** di **warning** usare l'opzione **-Wall**
  - \$ gcc -Wall -o eseguibile.out source.c

# - Compilare con opzione debug -

- Per effettuare il debug di un programma, utilizzate sempre l'opzione **-g**:
  - `$ gcc -Wall -g -o eseguibile.out source.c`
- L'opzione **-g** fa sì che il programma eseguibile contenga informazioni supplementari che permettono al debugger di collegare le istruzioni in linguaggio macchina che si trovano nell'eseguibile alle righe del codice corrispondenti nei sorgenti C

# Esempi di Sincronizzazione



# Classici Problemi di Sincronizzazione

---

- Problemi classici usati per testare schemi di sincronizzazione
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

# Bounded-Buffer Problem

---

- $n$  buffer, ognuno può contenere un articolo
- Semaforo **mutex** initializzato al valore 1
- Semaforo **full** initializzato al valore 0
- Semaforo **empty** initializzato al valore n

# Bounded Buffer Problem

---

- La struttura del produttore

```
do {
 ...
 /* produce an item in next_produced */
 ...
 wait(empty);
 wait(mutex);
 ...
 /* add next produced to the buffer */
 ...
 signal(mutex);
 signal(full);
} while (true);
```

# Bounded Buffer Problem

---

- La struttura del consumatore

```
Do {
 wait(full);
 wait(mutex);
 ...
 /* remove an item from buffer to next_consumed */
 ...
 signal(mutex);
 signal(empty);
 ...
 /* consume the item in next consumed */
 ...
} while (true);
```

# Readers-Writers Problem

---

- Un insieme di dati è condiviso tra n processi concorrenti
  - Readers – leggono solo i dati; **non** aggiornano
  - Writers – leggono e scrivono
- **Problema** – permettere a lettori multipli di leggere allo stesso tempo
  - solo un singolo scrittore può accedere i dati condivisi allo stesso tempo
- Diverse variazioni – basate su priorità
- Dati condivisi:
  - Data set
  - Semaforo binario **rw\_mutex** inizializzato a 1
  - Semaforo binario **mutex** inizializzato a 1
    - protegge **read\_count**
  - Intero **read\_count** inizializzato a 0

# Readers-Writers Problem

---

- Struttura scrittore

```
do {
 wait(rw_mutex);
 ...
 /* writing is performed */
 ...
 signal(rw_mutex);
} while (true);
```

# Readers-Writers Problem

---

## □ Struttura del lettore

```
do {
 wait(mutex);
 read_count++;
 if (read_count == 1)
 wait(rw_mutex);
 signal(mutex);

 ...
 /* reading is performed */

 ...

 wait(mutex);
 read_count--;
 if (read_count == 0)
 signal(rw_mutex);
 signal(mutex);
} while (true);
```

Se uno scrittore è in sezione critica con n lettori in attesa, n-1 su mutex, 1 su rw\_mutex

# Readers-Writers Problem

## □ Struttura del lettore

```
do {
 wait(mutex);
 read_count++;
 if (read_count == 1)
 wait(rw_mutex);
 signal(mutex);

 ...
 /* reading is performed */

 ...

 wait(mutex);
 read_count--;
 if (read_count == 0)
 signal(rw_mutex);
 signal(mutex);
} while (true);
```

Primo lettore



Se uno scrittore è in sezione critica con n lettori in attesa, n-1 su mutex, 1 su rw\_mutex

# Readers-Writers Problem

## □ Struttura del lettore

Altri lettori in attesa  **do {**

Primo lettore aspetta 

```
 wait(mutex);
 read_count++;
 if (read_count == 1)
 wait(rw_mutex);
 signal(mutex);

 ...
/* reading is performed */

 ...

 wait(mutex);
 read_count--;
 if (read_count == 0)
 signal(rw_mutex);
 signal(mutex);

} while (true);
```

Se uno scrittore è in sezione critica con n lettori in attesa, n-1 su mutex, 1 su rw\_mutex

# Readers-Writers Problem

## □ Struttura del lettore

```
Altro lettore do {
 
 wait(mutex);
 read_count++;
 if (read_count == 1)
 wait(rw_mutex);
 signal(mutex);

 ...
 /* reading is performed */

 ...

 wait(mutex);
 read_count--;
 if (read_count == 0)
 signal(rw_mutex);
 signal(mutex);
} while (true);
```

Primo lettore  
prende  il lock di rw\_mutex

Il primo lettore prende  
il lock di rw\_mutex e  
permette l'accesso agli  
altri lettori

# Readers-Writers Problem

## □ Struttura del lettore

```
do {
 wait(mutex);
 read_count++;
 if (read_count == 1)
 wait(rw_mutex);
 signal(mutex); sblocco
 ...
 /* reading is performed */
 ...
 wait(mutex);
 read_count--;
 if (read_count == 0)
 signal(rw_mutex);
 signal(mutex);
} while (true);
```

Altri lettori evitano il wait su **rw\_mutex** e si alternano solo su **mutex**

Sbloccato dal primo lettore gli altri lettori evitano la wait su **rw\_mutex**

Tutti i lettori in mutua esclusione su **read\_count** usando **mutex**

# Readers-Writers Problem

## □ Struttura del lettore

```
do {
 wait(mutex);
 read_count++;
 if (read_count == 1)
 wait(rw_mutex);
 signal(mutex); sblocco
 ...
 /* reading is performed */
 ...
 wait(mutex);
 read_count--;
 if (read_count == 0)
 signal(rw_mutex);
 signal(mutex);
} while (true);
```

Ultimo lettore lascia il `rw_mutex`



L'ultimo lettore rilascia `rw_mutex` quindi o scrittore o primo lettore potrà prenderlo

# Readers-Writers Problem

---

- **Read-write lock** introdotti per risolvere questo tipo di problema:
  - Lock in read mode o write mode
  - Più reader in possono prendere in read mode
  - Un solo writer in write mode
- Utili:
  - ▶ Quando facile identificare quali processi leggono solo dati condivisi e quali processi scrivono solo dati condivisi.
  - ▶ Quando si hanno più lettori che scrittori.
    - I blocchi di lettura e scrittura generalmente richiedono più sovraccarico per essere stabiliti di semafori o blocchi di mutua esclusione.
    - Consentire più lettori compensa il sovraccarico dovuto dal blocco lettore-scrittore.

# Readers-Writers Problem Variazioni

---

- **Primo problema** – i lettori non aspettano altri processi lettori per accedere al dato
- **Secondo problema** – una volta che lo scrittore è pronto esegue la scrittura ASAP
- In entrambi i casi si può avere starvation portando anche a più variazioni (nel primo caso lo scrittore, nel secondo i lettori)
- Problema è risolto su alcuni sistemi dal kernel fornendo un reader-writer lock

# Problema della Cena dei Filosofi

---



- Filosofi pensano e mangiano
- Non interagiscono con i vicini, a volte provano a prendere 2 bacchette (una alla volta) per mangiare
  - Due per mangiare, poi le rilasciano quando hanno finito
- Se 5 filosofi
  - Dati condivisi
    - ▶ Ciotola di riso (data set)
    - ▶ Semaforo **chopstick [5]** inizializzato a 1

# Problema della Cena dei Filosofi

---

- filosofo *i*:

```
do {
 wait (chopstick[i]);
 wait (chopStick[(i + 1) % 5]);

 // eat

 signal (chopstick[i]);
 signal (chopstick[(i + 1) % 5]);

 // think

} while (TRUE);
```

- Problema di questo algoritmo?

# Problema della Cena dei Filosofi

---

- filosofo *i*:

```
do {
 wait (chopstick[i]);
 wait (chopStick[(i + 1) % 5]);

 // eat

 signal (chopstick[i]);
 signal (chopstick[(i + 1) % 5]);

 // think

} while (TRUE);
```

- Problema di questo algoritmo?
- Se tutti i filosofi prendono contemporaneamente la bacchetta sinistra deadlock

# Problema della Cena dei Filosofi

---

- filosofo *i*:

```
do {
 wait (chopstick[i]);
 wait (chopStick[(i + 1) % 5]);

 // eat

 signal (chopstick[i]);
 signal (chopstick[(i + 1) % 5]);

 // think

} while (TRUE);
```

- Rimedi:

- Permetti di sedere solo a 4 filosofi
- Permetti di prendere le bacchette solo se sono entrambe disponibili
- Soluzione asimmetrica (numero di dispari di filosofi)

# Soluzione con Monitor

---

```
monitor DiningPhilosophers
{
 enum { THINKING, HUNGRY, EATING} state [5] ;
 condition self [5];
 void pickup (int i) {
 state[i] = HUNGRY;
 test(i);
 if (state[i] != EATING) self[i].wait;
 }
 void putdown (int i) {
 state[i] = THINKING;
 // test left and right neighbors
 test((i + 4) % 5);
 test((i + 1) % 5);
 }
}
```

stati usati per coordinare le azioni

test valuta e cambia stato in EATING

Self[i] Permette ad *i* di ritardare quando è affamato ma non può prendere entrambe le bacchette.

Un filosofo alla volta mangia (quando ha fame) prendendo entrambe le bacchette. Se ha fame controlla prima se ci sono entrambe le bacchette.

# Problema della Cena dei Filosofi

---

```
void test (int i) {
 if ((state[(i + 4) % 5] != EATING) &&
 (state[i] == HUNGRY) &&
 (state[(i + 1) % 5] != EATING)) {
 state[i] = EATING ;
 self[i].signal () ;
 }
}

initialization_code() {
 for (int i = 0; i < 5; i++)
 state[i] = THINKING;
}
```

Verifica se i vicini di i mangiano nel caso sia affamato. Se non mangiano si seta in EATING. Dopo il controllo sblocca i

# Soluzione Cena Filosofi

---

- Ogni filosofo  $i$  invoca **pickup()** e **putdown()**

```
DiningPhilosophers.pickup(i);
```

EAT

```
DiningPhilosophers.putdown(i);
```

- Non deadlock, ma starvation è possibile

# Esempi di Sincronizzazione

---

- Solaris
- Windows
- Linux
- Pthreads

# Solaris

---

- Implementa una varietà di lock per supportare multitasking, multithreading (inclusi real-time thread) e multiprocessing
- Usa **adaptive mutexes** per efficienza per piccolo frammenti di codice
  - Inizia come semaforo standard spin-lock
  - Se lock tenuto, con thread eseguito su altra CPU, spins
  - Se lock tenuto da thread non in run, si blocca e va in sleep in attesa di un segnale di ripresa
- Usa **condition variables**
- Usa **readers-writers** lock
- Usa **turnstiles** (tornelli) per ordinare la lista dei thread in attesa di prendere un adaptive mutex o un reader-writer lock (lock per lettori e scrittori)

# Windows

---

- Usa una interrupt masks per proteggere l'accesso alle risorse condivise su sistemi uniprocessori
- Usa **spin-lock** su sistemi multiprocessore
  - Spinlocking-thread mai prelazionati
- Usa dei **dispatcher object** che possono essere utilizzati come mutexes, semafori, eventi, e timer
  - **Eventi**
    - ▶ Un evento simile ad una condition variable
  - Timer notifica a uno o più thread quando scade il tempo
  - **dispatcher object** signaled-state (oggetto disponibile) o non-signaled state (thread si blocca)

# Linux

---

- Linux:
  - Prima del kernel versione 2.6, disabilitati gli interrupt per implementare brevi sezioni critiche
  - Versione 2.6 e successive, completamente preemptive
- Linux fornisce:
  - Semafori
  - atomic integer
  - spinlocks
  - versioni di reader-writer
- Su CPU singola spinlocks sostituiti da abilitazione/disabilitazione della prelazione nel kernel

# Linux

---

- Linux kernel:
  - Lock non ricorsivi (due blocchi stalla)
  - Perlazione abilitata o disabilitata
    - ▶ Preempt\_disable()
    - ▶ Preempt\_enable()
- Contatore dei lock()
  - ▶ preempt count
  - ▶ Se zero allora è safe prelazionare il kernel

# Pthreads

---

- Pthreads API sono OS-independent
- Forniscono:
  - mutex locks
  - condition variable
- Estensioni includono:
  - read-write locks
  - spinlocks

## Mutex Posix

Un mutex Posix è caratterizzato dalle seguenti proprietà:

- è una variabile di tipo `pthread_mutex_t` che può essere inizializzata con diversi attributi ([tipo](#), [scopo](#), etc.)
- può assumere solo i due stati alternativi [chiuso](#) (locked) o [aperto](#) (unlocked);
- può essere chiuso solo da [un](#) processo alla volta, ed il processo che chiude il mutex ne diviene il [possessore](#) fino alla successiva chiusura;
- può essere riaperto solo dal proprio [possessore](#);
- deve essere [condiviso](#) tra tutti i processi che intendono sincronizzare l'accesso ad una regione critica ([blocco cooperativo](#)).

## I mutex

- Un mutex è un semaforo binario (rosso o verde)
- Un mutex è mantenuto in una struttura

`pthread_mutex_t`

- Tale struttura va allocata e inizializzata
- Per inizializzare:

- se la struttura è allocata staticamente:

```
pthread_mutex_t c = PTHREAD_MUTEX_INITIALIZER
```

- se la struttura è allocata dinamicamente (e.g. se si usa `malloc`): chiamare `pthread_mutex_init`

## Inizializzare e distruggere un mutex

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex,
 const pthread_mutexattr_t *attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- inizializza e distrugge un mutex, rispettivamente
- Quando inizializzato è in stato aperto
- restituiscono 0 se OK, un codice d'errore altrimenti
- attr può essere NULL (attributi di default)

# Mutex

---

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_trylock (pthread_mutex_t *mutex);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

- acquisiscono e rilasciano il semaforo
- restituiscono 0 se OK, un codice d'errore altrimenti
- se il semaforo è occupato (locked)...
  - ...lock blocca il thread finché il semaforo si libera
  - ...trylock invece non blocca, ma restituisce subito l'errore EBUSY

# Mutex

---

```
myfoo test; // Variabile GLOBALE
pthread_mutex_t sem=PTHREAD_MUTEX_INITIALIZER;

void *inc(void *arg){ // incrementa a e b
 pthread_mutex_lock(&sem);
 test.a++;
 test.b++;
 printf("tid=%d a=%d b=%d\n" , pthread_self(), test.a, test.b);
 pthread_mutex_unlock(&sem);
 pthread_exit((void *)&test);
}

int main(void){
 char st[100];
 pthread_t tid;
 int i=0;
 myfoo *b;

 while (i++<10){
 pthread_create(&tid, NULL, inc, NULL); // Thread concorrenti
 }
}
```

# Mutex

---

```
tid=1077283760 a=1 b=1
tid=1096194992 a=2 b=2
tid=1079385008 a=3 b=3
tid=1081486256 a=4 b=4
tid=1083587504 a=5 b=5
tid=1085688752 a=6 b=6
tid=1087790000 a=7 b=7
tid=1094093744 a=8 b=8
tid=1091992496 a=9 b=9
tid=1089891248 a=10 b=10
```

# Mutex

---

```
typedef struct foo{
 int a;
 int b;
 pthread_mutex_t sem;
} myfoo;

myfoo *test; // Variabile GLOBALE

myfoo *init_struct(){
 struct foo *fp;

 if ((fp=malloc(sizeof(myfoo)))==NULL)
 return(NULL);
 fp->a=0;
 fp->b=0;
 pthread_mutex_init(&fp->sem,NULL);
 return(fp);

}
```

# Mutex

---

```
void stampa(struct foo *test){
```

```
 printf("tid=%d a=%d b=%d\n", pthread_self(), test->a, test->b);
 fflush(stdout);
```

```
}
```

```
void *inc(void *arg){ // incrementa a e b
```

```
 pthread_mutex_lock(&test->sem);
 test->a=test->a+2;
 test->b++; // Variabile GLOBALE
 stampa(test);
 pthread_mutex_unlock(&test->sem);
 pthread_exit((void *)&test);
}
```

```
tid=1077283760 a=2 b=1
```

```
tid=1079385008 a=4 b=2
```

```
tid=1081486256 a=6 b=3
```

```
tid=1083587504 a=8 b=4
```

```
tid=1085688752 a=10 b=5
```

```
tid=1087790000 a=12 b=6
```

```
tid=1089891248 a=14 b=7
```

```
tid=1091992496 a=16 b=8
```

```
tid=1094093744 a=18 b=9
```

```
tid=1096194992 a=20 b=10
```

```
Master:tid=1075181248 a=20 b=10
```

```
int main(void){
```

```
 char st[100];
```

```
 pthread_t tid;
```

```
 int i=0;
```

```
 myfoo *b;
```

```
 test=init_struct();
```

```
 while (i++<10){
```

```
 pthread_create(&tid, NULL, inc, NULL); // Globale
```

```
}
```

```
 sleep(1);
```

```
 printf("Master:");
```

```
 stampa(test);
```

```
 pthread_mutex_destroy(&test->sem);
```

```
}
```

## Tipologie di Mutex

- fast: semantica classica, pertanto un thread che esegue due `mutex_lock()` consecutivi sullo stesso mutex causa uno stallo
- recursive: conta il numero di volte che un thread blocca il mutex e lo rilascia solo se esegue un pari numero di `mutex_unlock()`
- error-checking: controlla che il thread che rilascia il mutex sia lo stesso thread che lo possiede

# Mutex

- inizializzazione di un mutex

- statica, macro per inizializzare un mutex:

```
fastmutex = PTHREAD_MUTEX_INITIALIZER;
recmutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
errchkmutex = PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
```

- dinamica, chiamata di libreria:

```
int pthread_mutex_init(pthread_mutex_t *mp,
 const pthread_mutexattr_t *mattr);
```

- **mp** è una mutex precedentemente allocato
  - **mattr** sono gli attributi del mutex: **NULL** per il default
  - restituisce sempre 0

# Mutex

## Attributi Mutex

- Si può scegliere il tipo usando queste macro:

- fast: PTHREAD\_MUTEX\_FAST\_NP
- recursive: PTHREAD\_MUTEX\_RECURSIVE\_NP
- error checking: PTHREAD\_MUTEX\_ERRORCHECK\_NP

e le funzioni per fissare/conoscere il tipo:

```
#include <pthread.h>
int pthread_mutexattr_settype(pthread_mutexattr_t *attr,
 int kind);
```

- restituisce 0 oppure un intero ≠0 in caso di errore

```
int pthread_mutexattr_gettype(const pthread_mutexattr_t *attr,
 int *kind);
```

- restituisce sempre 0

## Limiti del MUTEX

- Se un thread attende il verificarsi di una condizione su una risorsa condivisa con altri thread
- Con i soli mutex sarebbe necessario un ciclo del tipo:

```
while(1) {
 lock(mutex);
 if (<condizione sulla risorsa condivisa>)
 break;
 unlock(mutex);
 ...
}
<sezione critica>;
unlock(mutex);
```

Attesa, e verifica ciclica sulla var, finchè la condizione verificata non rompe il ciclo, quindi sblocco.

## Limiti MUTEX

- I mutex sono come strumento di cooperazione risultano:
  - inefficienti
  - inelegantie per risolvere elegantemente problemi di cooperazione servono altri strumenti
- Le variabili condizione sono un'implementazione delle variabili condizione teorizzate da Hoare

# Varibile di Condizione

---

## Variabili di Condizione

- strumenti di sincronizzazione tra thread che consentono di:
  - attendere passivamente il verificarsi di una condizione su una risorsa condivisa
  - segnalare il verificarsi di tale condizione
- la condizione interessa sempre e comunque una risorsa condivisa
- pertanto le variabili condizioni possono sempre associarsi al mutex della stessa per evitare corse critiche sul loro utilizzo

```
int pthread_cond_wait(pthread_cond_t *cond,
 pthread_mutex_t *mutex);
```

# Varibile di Condizione

---

## Usare una condition variable

- thread che aspetta una condizione
- thread che rende la condizione vera

**mutex\_lock(m)**

while (condizione falsa)

**cond\_wait(c, m)**

*fa' qualcosa*

**mutex\_unlock(m)**

**mutex\_lock(m)**

*rendi la condizione vera*

**cond\_broadcast(c)**

**mutex\_unlock(m)**

# Varibile di Condizione

---

## Attendere una condition variable

```
int pthread_cond_wait(pthread_cond_t *cond,
 pthread_mutex_t *mutex);
```

- attende che cond sia segnalata come vera
- restituisce 0 se OK, un codice d'errore altrimenti
- il mutex protegge la condizione
  - deve essere acquisito prima di chiamare cond\_wait
  - durante l'attesa, cond\_wait rilascia il mutex
  - finita l'attesa, cond\_wait riprende il mutex

# Varibile di Condizione

---

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond,
 pthread_mutex_t *mutex);
```

- `cond` è una variabile condizione
- `mutex` è un mutex associato alla variabile
- 1. al momento della chiamata il mutex dove essere bloccato
- 2. rilascia il mutex, il thread chiamante rimane in attesa passiva di una segnalazione sulla variabile condizione
- 3. nel momento di una segnalazione, la chiamata restituisce il controllo al thread chiamante, e questo rientra in competizione per acquisire il mutex
- restituisce 0 in caso di successo oppure un codice d'errore ≠0

# Varibile di Condizione

---

```
...
/* Chiama do_work() mentre flag è settato, altrimenti si
 blocca in attesa che venga segnalato un cambiamento nel
 suo valore */

void* thread_function (void* thread_arg) {
 while (1) {
 // Attende segnale sulla variabile condizione
 pthread_mutex_lock(&thread_flag_mutex);
 while (!thread_flag)
 pthread_cond_wait(&thread_flag_cv, &thread_flag_mutex);
 pthread_mutex_unlock(&thread_flag_mutex);
 do_work (); /* Fa qualcosa */
 }
 return NULL;
}_
```

# Variabile di Condizione

---

```
int pthread_cond_signal(pthread_cond_t *cond);
```

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- signal risveglia esattamente un thread in attesa su una condition variable
- broadcast risveglia tutti i thread in attesa su una condition variable
- restituiscono 0 se OK, un codice d'errore altrimenti
- attr può essere NULL (attributi di default)

# Varibile di Condizione

---

## Segnalazione

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
```

- uno dei thread che sono in attesa sulla variabile condizione cond viene risvegliato
  - se più thread sono in attesa, ne viene scelto uno ed uno solo effettuando una scelta non deterministica
  - se non ci sono thread in attesa, non accade nulla

```
--> // Setta flag a FLAG_VALUE
void set_thread_flag (int flag_value) {
 // Lock del mutex su flag
 pthread_mutex_lock (&thread_flag_mutex);
 // cambia il valore del flag
 thread_flag = FLAG_VALUE;
 // segnala a chi è in attesa che
 // il valore di flag è cambiato
 pthread_cond_signal (&thread_flag_cv);
 // unlock del mutex
 pthread_mutex_unlock (&thread_flag_mutex);
```

# Varibile di Condizione

---

```
pthread_mutex_t sem=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;

void *dec(void *arg){

 while(1){
 pthread_mutex_lock(&sem);
 while (test.a==0)
 pthread_cond_wait(&cond, &sem);

 printf("CONSUMATORE 1 a=%d \n", test.a);
 test.a--;
 pthread_mutex_unlock(&sem);
 nanosleep(&t2,NULL);
 }
}

void *dec2(void *arg){

 while(1){
 pthread_mutex_lock(&sem);
 while (test.a==0)
 pthread_cond_wait(&cond, &sem);

 printf("CONSUMATORE 2 a=%d, b=%d \n", test.a, test.b);
 test.a--;
 test.b--;
 pthread_mutex_unlock(&sem);
 nanosleep(&t2,NULL);
 }
}
```

# Varibile di Condizione

```
void *inc(void *arg){ // incrementa a e b

 int j=0;
 while (j++<10){
 pthread_mutex_lock(&sem);
 test.a++;
 test.b++;
 printf("PRODUTTORE tid=%d a=%d b=%d\n", pthread_self(),test.a, test.b);

 pthread_cond_signal(&cond);
 pthread_mutex_unlock(&sem);
 nanosleep(&t2,NULL);
 }
 pthread_exit((void *)&test);
}

int main(void){
 pthread_t tid;
 pthread_create(&tid, NULL, inc, NULL);
 pthread_create(&tid, NULL, dec, NULL);
 pthread_create(&tid, NULL, dec2,NULL);
 sleep(5);
}
```

PRODUTTORE tid=1077283760 a=1 b=1  
CONSUMATORE 1 a=1  
PRODUTTORE tid=1077283760 a=1 b=2  
CONSUMATORE 1 a=1  
PRODUTTORE tid=1077283760 a=1 b=3  
CONSUMATORE 1 a=1  
PRODUTTORE tid=1077283760 a=1 b=4  
CONSUMATORE 2 a=1, b=4  
PRODUTTORE tid=1077283760 a=1 b=4  
CONSUMATORE 1 a=1  
PRODUTTORE tid=1077283760 a=1 b=5  
CONSUMATORE 1 a=1

# Semafori

---

- Appartengono all'estensione Posix SEM
  - #include <semaphore.h>
  - Named e unnamed
- 
- *Named*

```
#include <semaphore.h>
sem_t *sem;
/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);

/* acquire the semaphore */
sem_wait(sem);
/* critical section */
/* release the semaphore */
sem_post(sem);
```

# Semafori

---

- Appartengono all'estensione Posix SEM
  - #include <semaphore.h>
  - Named e unnamed
- 
- *Unnamed*
    - A pointer to the semaphore
    - A flag indicating the level of sharing
    - The semaphore's initial value

```
#include <semaphore.h>
sem_t sem;
/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

flag 0, shared only by threads belonging to the process that created the semaphore.

nonzero value, shared between separate processes by placing it in a region of shared memory.

```
/* acquire the semaphore */
sem_wait(&sem);
/* critical section */
/* release the semaphore */
sem_post(&sem);
```

# Java Monitor

---

- Monitor per implementare il buffer limitato
- I metodi insert e remove sono sincronizzati
  - Invocarli significa acquisire il lock

```
public class BoundedBuffer<E>
{
 private static final int BUFFER_SIZE = 5;

 private int count, in, out;
 private E[] buffer;

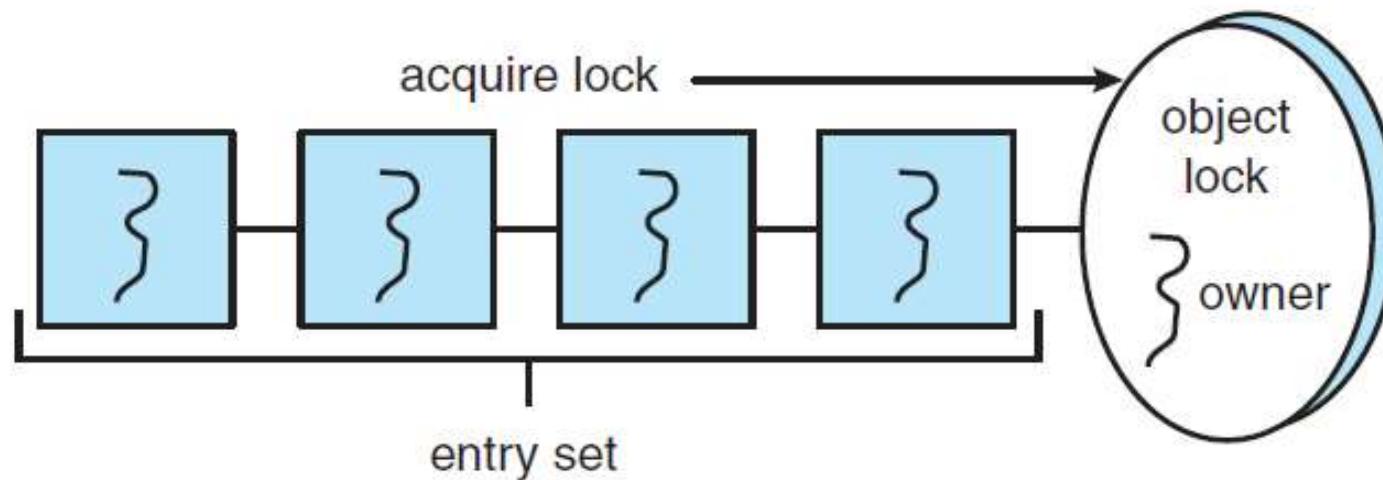
 public BoundedBuffer() {
 count = 0;
 in = 0;
 out = 0;
 buffer = (E[]) new Object[BUFFER_SIZE];
 }

 /* Producers call this method */
 public synchronized void insert(E item) {
 /* See Figure 7.11 */
 }

 /* Consumers call this method */
 public synchronized E remove() {
 /* See Figure 7.11 */
 }
}
```

# Java Monitor

- Monitor per implementare il buffer limitato
- I metodi insert e remove sono sincronizzati
  - Invocarli significa acquisire il lock



# Java Monitor

---

- Monitor per implementare il buffer limitato
- I metodi insert e remove sono sincronizzati
  - Insert deve attendere con wait() se il buffer è pieno

```
/* Producers call this method */
public synchronized void insert(E item) {
 while (count == BUFFER_SIZE) {
 try {
 wait();
 }
 catch (InterruptedException ie) { }
 }

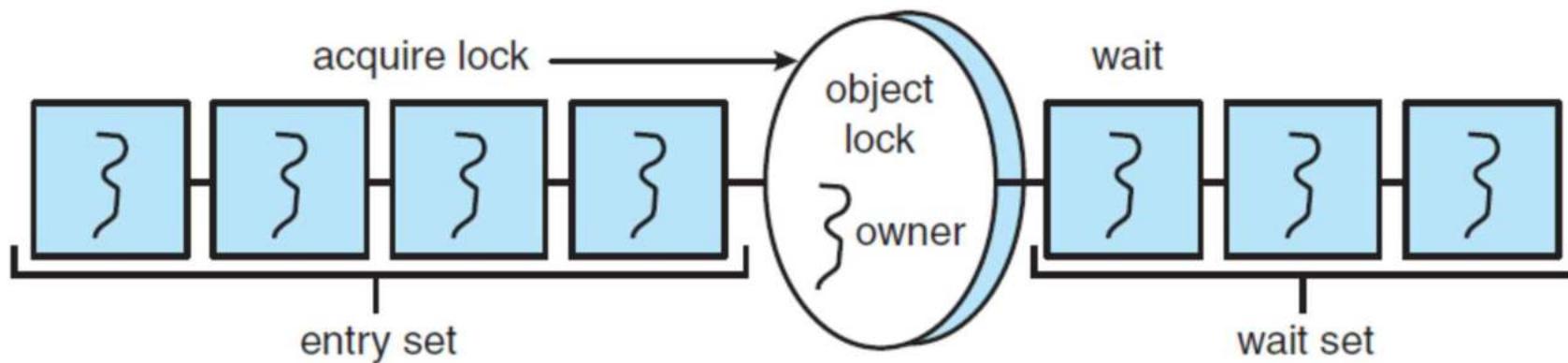
 buffer[in] = item;
 in = (in + 1) % BUFFER_SIZE;
 count++;

 notify();
}
```

1. The thread releases the lock for the object.
2. The state of the thread is set to blocked.
3. The thread is placed in the wait set for the object.

# Java Monitor

- Monitor per implementare il buffer limitato
- I metodi insert e remove sono sincronizzati
  - Insert deve attendere con wait() se il buffer è pieno



# Java Monitor

---

- Monitor per implementare il buffer limitato
- I metodi insert e remove sono sincronizzati
  - Remove deve attendere con wait() se il buffer è vuoto

```
/* Consumers call this method */
public synchronized E remove() {
 E item;

 while (count == 0) {
 try {
 wait();
 }
 catch (InterruptedException ie) { }
 }

 item = buffer[out];
 out = (out + 1) % BUFFER_SIZE;
 count--;

 notify();
}

return item;
}
```

1. Picks an arbitrary thread T from the list of threads in the wait set
2. Moves T from the wait set to the entry set
3. Sets the state of T from blocked to runnable

# Java: Reentrant Lock

---

- Strumento di sincronizzazione più semplice in Java API
- Gestito da un thread per sincronizzare
- Esiste anche ReentrantReadWriteLock

```
Lock key = new ReentrantLock();

key.lock();
try {
 /* critical section */
}
finally {
 key.unlock();
}
```

# Java: semafori

---

## □ Semafori contatori in Java API

```
Semaphore(int value);

Semaphore sem = new Semaphore(1);

try {
 sem.acquire();
 /* critical section */
}
catch (InterruptedException ie) { }
finally {
 sem.release();
}
```

# Java: variabili di condizione

---

- Variabili di condizione in Java API
- Associate ad un lock
- Solo unnamed

```
Lock key = new ReentrantLock();
Condition condVar = key.newCondition();
```

```
Lock lock = new ReentrantLock();
Condition[] condVars = new Condition[5];

for (int i = 0; i < 5; i++)
 condVars[i] = lock.newCondition();
```

# Java: variabili di condizione

---

## □ Variabili di condizione in Java API

```
/* threadNumber is the thread that wishes to do some work */
public void doWork(int threadNumber)
{
 lock.lock();

 try {
 /**
 * If it's not my turn, then wait
 * until I'm signaled.
 */
 if (threadNumber != turn)
 condVars[threadNumber].await();

 /**
 * Do some work for awhile ...
 */

 /**
 * Now signal to the next thread.
 */
 turn = (turn + 1) % 5;
 condVars[turn].signal();
 }
 catch (InterruptedException ie) { }
 finally {
 lock.unlock();
 }
}
```

# Approcci Alternativi

---

- Con i sistemi multicore sempre più complesso lo sviluppo di applicazioni multithreaded senza race conditions
  - Transactional Memory
  - OpenMP
  - Functional Programming

# Transactional Memory

---

- Una **memory transaction** è una sequenza di operazioni read-write in memoria eseguite atomicamente
- Se tutte le operazioni in sequenza sono eseguite allora si conferma l'operazione
- Altrimenti si ripristina la situazione precedente
- Software Transactional Memory (STM)
- Hardware Transactional Memory (HTM)

```
void update()
{
 /* read/write memory */
}
```

# OpenMP

---

- OpenMP è un set di direttive per il compilatore e API a supporto del parallel programming.

```
void update(int value)
{
 #pragma omp critical
 {
 count += value
 }
}
```

Il codice in **#pragma omp critical** è la critical section ed è eseguito in modo atomico.

<https://www.openmp.org/>

# Functional Programming

---

- I linguaggi di programmazione funzionale offrono un paradigma diverso rispetto ai linguaggi procedurali in quanto non mantengono uno stato.
- Le variabili vengono trattate come immutabili e non possono cambiare stato una volta assegnato un valore.
- C'è un crescente interesse per linguaggi funzionali come Erlang e Scala per il loro approccio alla gestione delle gare di dati.

# Lezione 2: Sistemi Operativi

## Strutture

# Obiettivi

---

- Definire i servizi forniti da un Sistema Operativo
- Introdurre le **chiamate di sistema** che forniscono i servizi del sistema
- Discutere la strutturazione di un Sistema Operativo

# Servizi degli SO

---

- I sistemi operativi forniscono un ambiente per l'esecuzione di programmi e servizi a **programmi** e **utenti**
- Un insieme di servizi del sistema operativo fornisce funzioni utili per l'utente:

*Interfaccia  
con l'utente*

*Esecuzione  
di un  
programma*

*Operazioni  
di I/O*

*Gestione  
del file  
system*

*Comunicazioni*

*Rilevamento di  
errori*

# Servizi di Sistemi Operativi

---

- I sistemi operativi forniscono un ambiente per l'esecuzione di programmi e servizi a programmi e utenti
- Un insieme di servizi del sistema operativo fornisce funzioni utili per l'utente:
  - **Interfaccia Utente** – quasi tutti i sistemi le forniscono ([UI](#)).
    - ▶ Diversi tipi **Command-Line (CLI)**, **Graphics User Interface (GUI)**, **Batch**
  - **Esecuzione di Programmi** - Il sistema deve essere in grado di caricare un programma in memoria ed eseguire quel programma, terminare l'esecuzione, normalmente o in modo anomalo
  - **Operazioni I/O** - Un programma in esecuzione può richiedere I/O, che può coinvolgere un file o un dispositivo I/O.
    - ▶ L'S.O. deve nascondere la complessità del dispositivo offrendo agli utenti un interfacciamento semplice ed uniforme

# Servizi degli SO

---

- Un insieme di servizi del sistema operativo fornisce funzioni utili per l'utente:
  - **Gestione File-system** - I programmi devono leggere e scrivere file e directory, crearli ed eliminarli, ricercarli, elencare le informazioni sui file, gestire i permessi. Accesso frequente richiede gestione oculata.
  - **Comunicazione** – I processi possono scambiare informazioni, sullo stesso computer o tra computer in una rete
  - **Rilevamento di Errori** – Il sistema operativo deve essere costantemente consapevole di possibili errori
    - ▶ Può verificarsi nella CPU e nell'hardware di memoria, nei dispositivi I/O, nel programma utente
    - ▶ Per ogni tipo di errore, il sistema operativo dovrebbe intraprendere l'azione appropriata per garantire un'elaborazione corretta e coerente
    - ▶ Le funzionalità di debug possono migliorare notevolmente le capacità dell'utente e del programmatore di utilizzare in modo efficiente il sistema

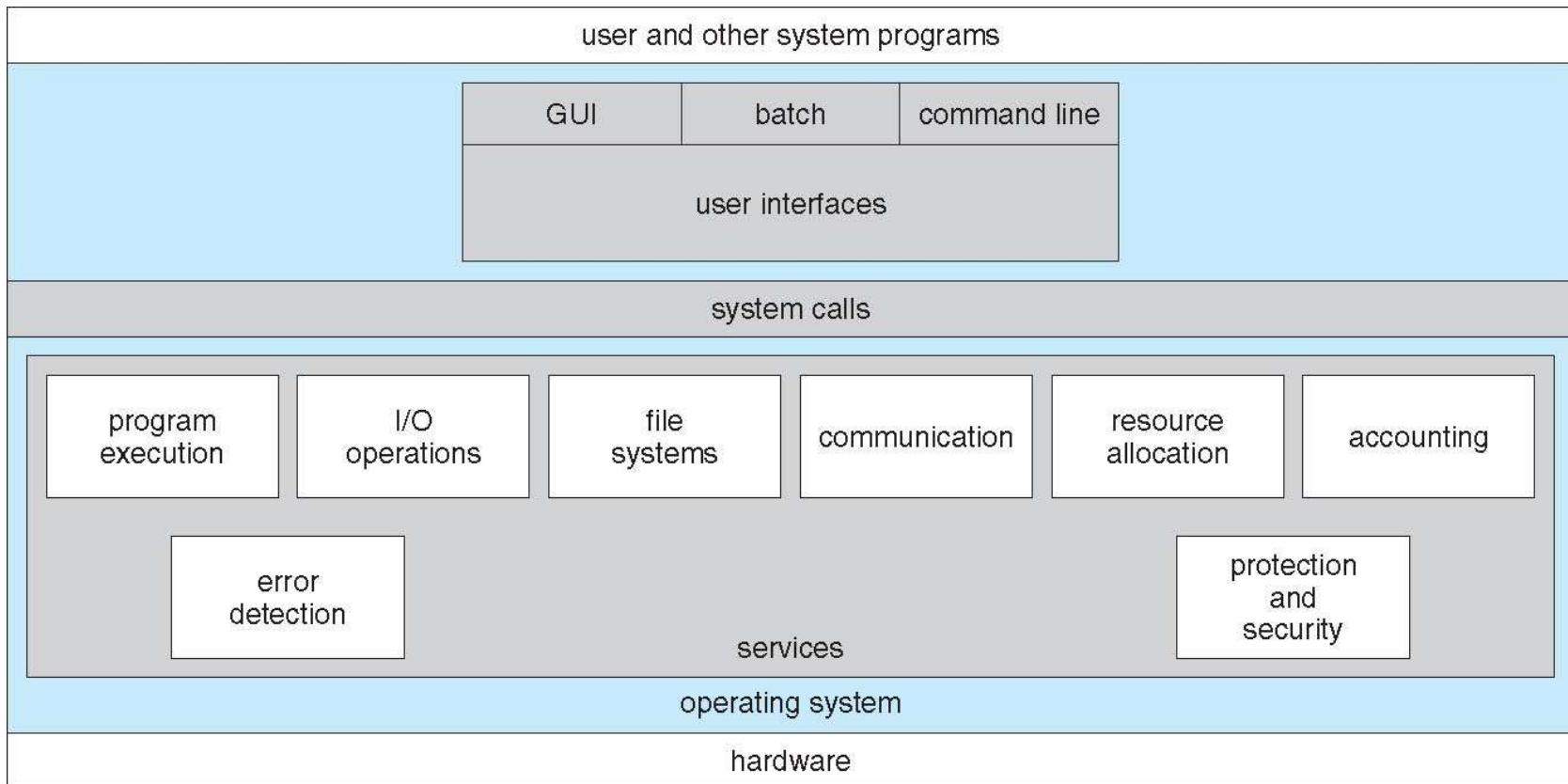
# Servizi degli SO

---

- Insieme di funzioni del sistema operativo per garantire il funzionamento efficiente del sistema stesso tramite la condivisione delle risorse
  - **Allocazione delle risorse** - quando più utenti o più lavori vengono eseguiti contemporaneamente, le risorse devono essere allocate a ciascuno di essi
    - ▶ Molti tipi di risorse - cicli CPU, memoria principale, file, dispositivi I/O.
  - **Contabilità** - tenere traccia di quali utenti utilizzano quanto e quali tipi di risorse del computer
  - **Protezione e sicurezza** - I proprietari delle informazioni archiviate in un sistema informatico multiutente o in rete devono poter controllare l'uso di tali informazioni, processi simultanei non devono interferire tra loro
    - ▶ La **protezione** deve garantire che l'accesso da parte di processi ed utenti alle risorse di sistema sia controllato
    - ▶ La **sicurezza** rispetto ad accessi esterni richiede l'autenticazione dell'utente e prevede la difesa dei dispositivi I/O esterni da accessi non consentiti

# Panoramica dei Servizi di un SO

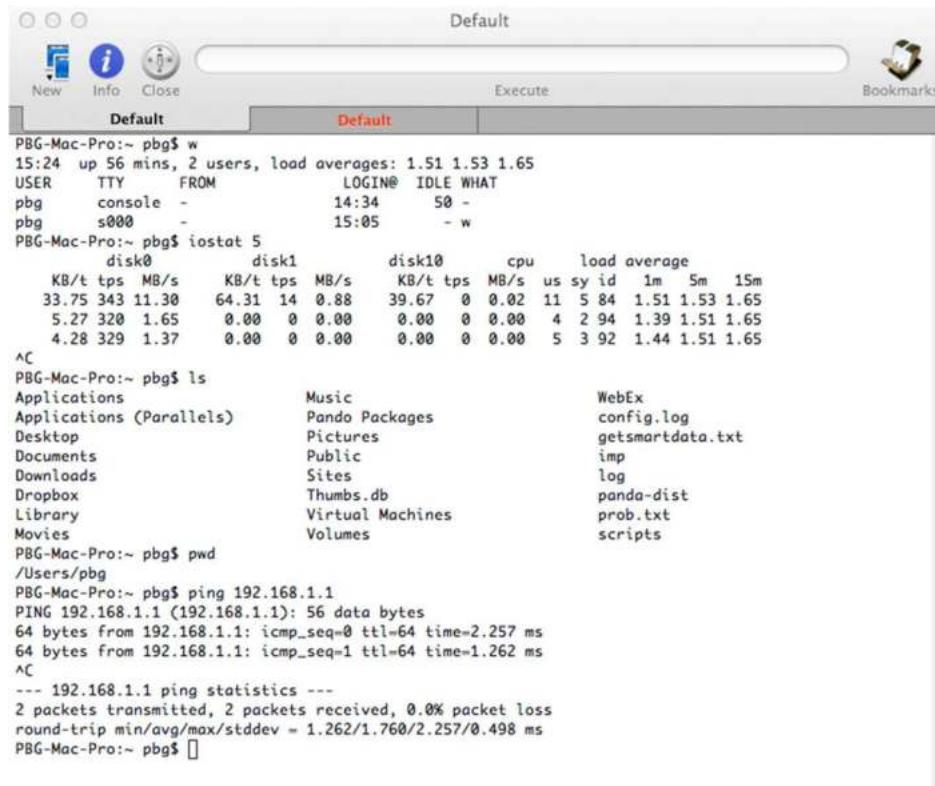
- I processi utente accedono ai servizi del Sistema Operativo mediante chiamate di sistema:
  - Istruzioni estese** che invocano servizi gestiti dal Kernel



# Interfaccia Utente - CLI

L'interprete dei comandi consente l'immissione diretta dei comandi

- A volte implementato nel kernel, a volte dal programma di sistema
- A volte più interfacce – **shells**
- Principalmente recupera un comando dall'utente e lo esegue

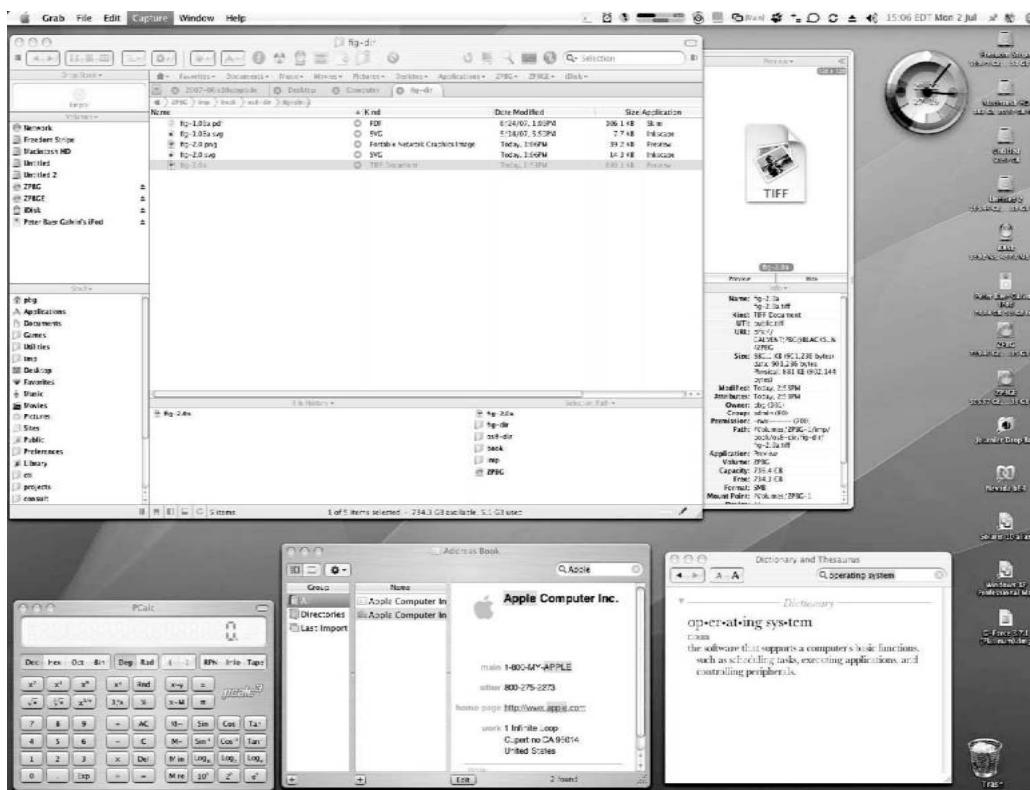


```
PBG-Mac-Pro:~ pbgs w
15:24 up 56 mins, 2 users, load averages: 1.51 1.53 1.65
USER TTY FROM LOGIN@ IDLE WHAT
pbgs console -
pbgs s000 -
PBG-Mac-Pro:~ pbgs iostat 5
 disk0 disk1 disk10 cpu load average
 KB/t tps MB/s KB/t tps MB/s KB/t tps MB/s us sy id 1m 5m 15m
 33.75 343 11.30 64.31 14 0.88 39.67 0 0.02 11 5 84 1.51 1.53 1.65
 5.27 320 1.65 0.00 0 0.00 0.00 0 0.00 4 2 94 1.39 1.51 1.65
 4.28 329 1.37 0.00 0 0.00 0.00 0 0.00 5 3 92 1.44 1.51 1.65
^C
PBG-Mac-Pro:~ pbgs ls
Applications Music WebEx
Applications (Parallels) Pango Packages config.log
Desktop Pictures getsmartdata.txt
Documents Public imp
Downloads Sites log
Dropbox Thumbs.db panda-dist
Library Virtual Machines prob.txt
Movies Volumes scripts
PBG-Mac-Pro:~ pbgs pwd
/Users/pbgs
PBG-Mac-Pro:~ pbgs ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: icmp_seq=0 ttl=64 time=2.257 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=1.262 ms
^C
--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 1.262/1.760/2.257/0.498 ms
PBG-Mac-Pro:~ pbgs
```

# Interfaccia Utente - GUI

# Interfaccia grafica (Graphic User Interface)

- Interfaccia che usa la metafora del desktop
  - Mouse, tastiera, schermo
  - Le icone rappresentano file, programmi, azioni, ecc
  - Inventata a Xerox PARC



# Altre Interfacce Utente

- Interfacce touch-screen
  - Senza mouse
  - Gesti per attivare azioni
- Comandi vocali
- Etc.



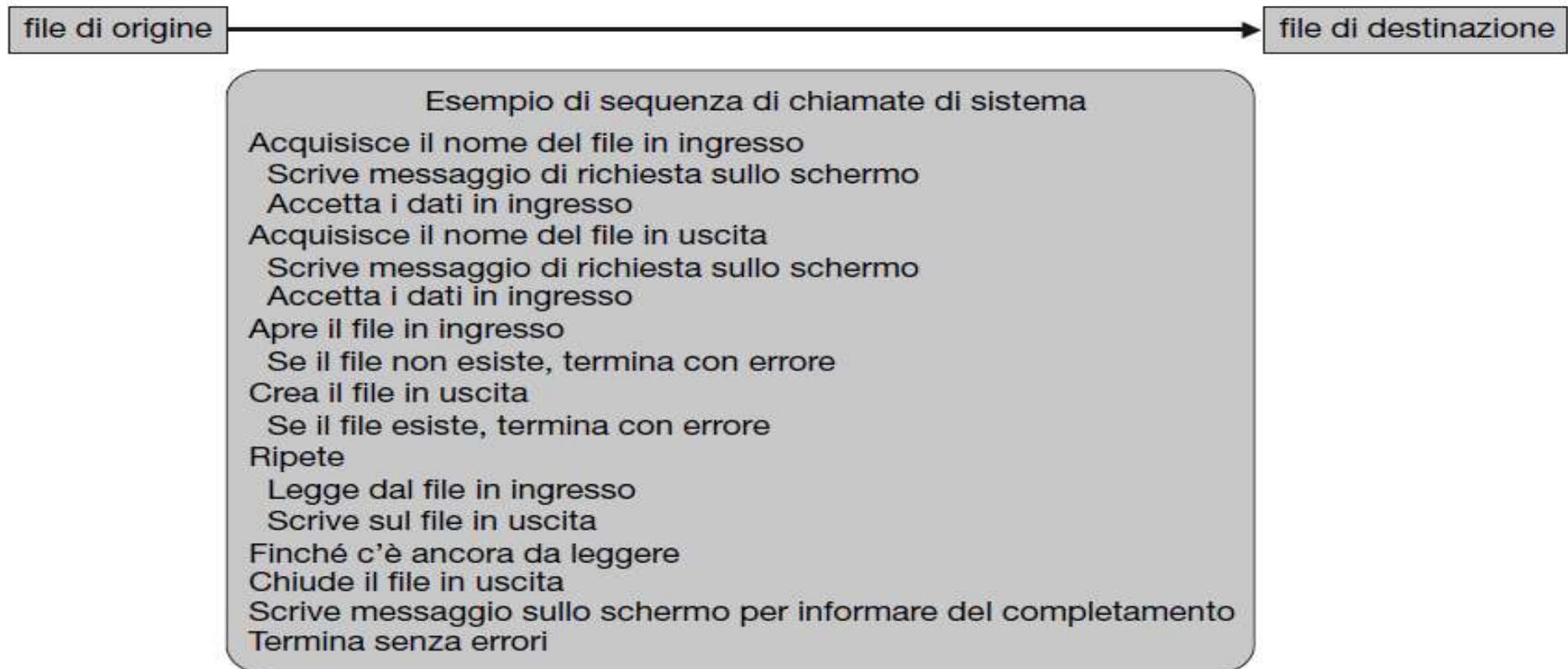
# Chiamate di Sistema

---

- Le **chiamate di sistema** (*system call*) costituiscono un'interfaccia per i servizi resi disponibili dal sistema operativo
- Tipicamente scritte in linguaggio di alto livello (C o C++)
- Chiamate dai programmi mediante **Application Programming Interface (API)**
- Tra le API più comuni citiamo:
  - Win32 API per Windows
  - POSIX API per POSIX-based systems (incluse tutte le versioni UNIX, Linux e Mac OS X) - **Portable Operating System Interface**
  - Java API per la Java virtual machine (JVM)

# Esempio di Chiamata di Sistema

- Chiamata di Sistema per copiare il contenuto di un file in un altro file



**Figura 2.5** Esempio d'uso delle chiamate di sistema.

# Esempio di API standard

**API:** Specifica un insieme di funzioni a disposizione del programmatore e dettaglia i parametri necessari per l'invocazione di queste funzioni insieme ai valori restituiti

## ESEMPIO DI API STANDARD

Come esempio di API standard consideriamo la funzione `read()` disponibile in Unix e Linux. L'API per questa funzione si può ottenere digitando

```
man read
```

da riga di comando. Una descrizione di questa API è la seguente:

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count)
```

valore nome parametri  
restituito della funzione

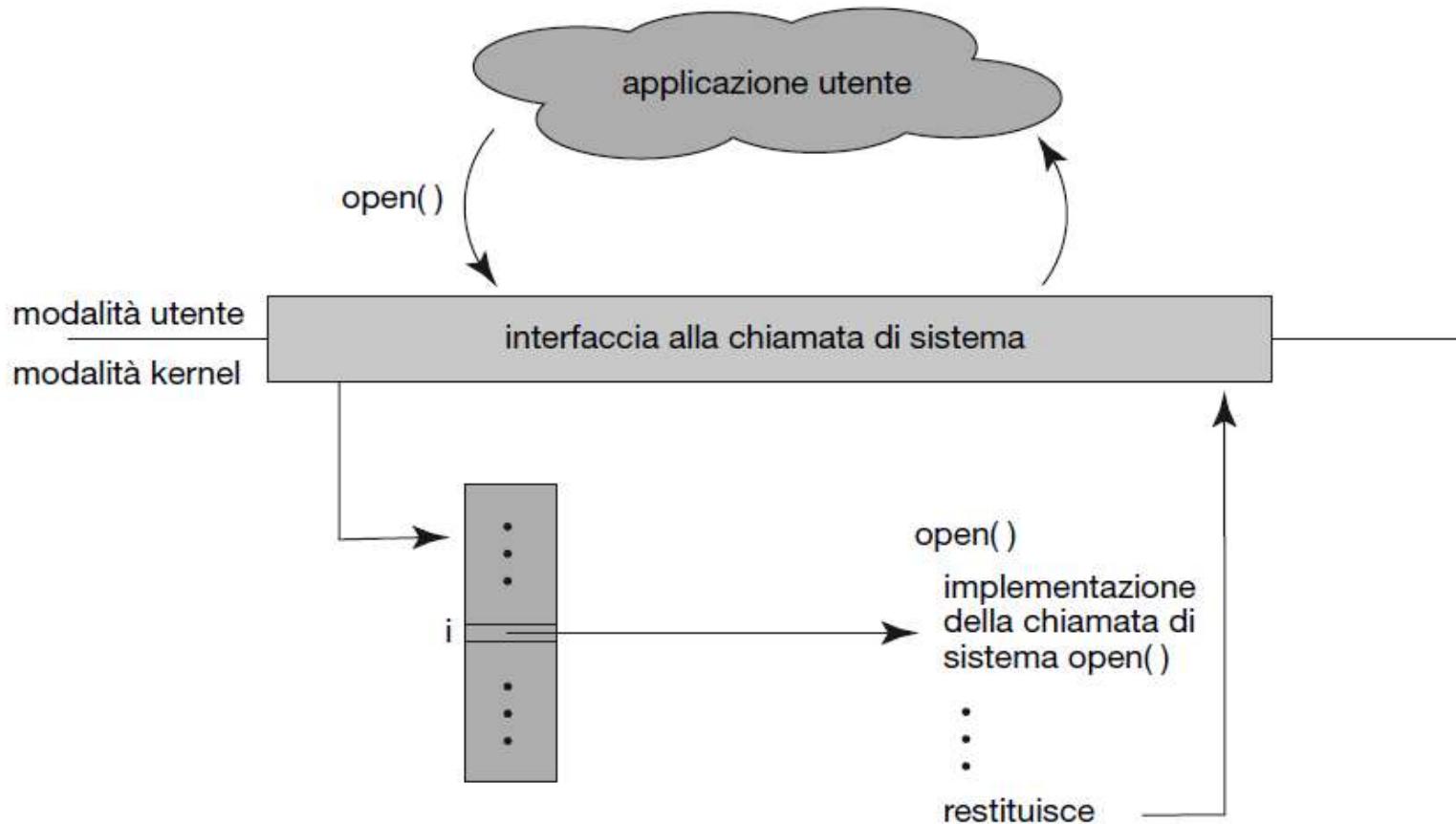
Un programma che utilizza la `read()` deve includere il file `unistd.h` che, tra le altre cose, definisce i tipi di dato `ssize_t` e `size_t`. I parametri passati alla `read()` sono i seguenti:

- `int fd` — il descrittore del file da leggere
- `void *buf` — un buffer nel quale vengono messi i dati letti
- `size_t count` — il massimo numero di byte da leggere e inserire nel buffer

Quando una `read()` è completata con successo viene restituito il numero di byte letti. La `read()` restituisce 0 in caso di fine del file e -1 quando si è verificato un errore.

# Relazione tra API e Chiamata

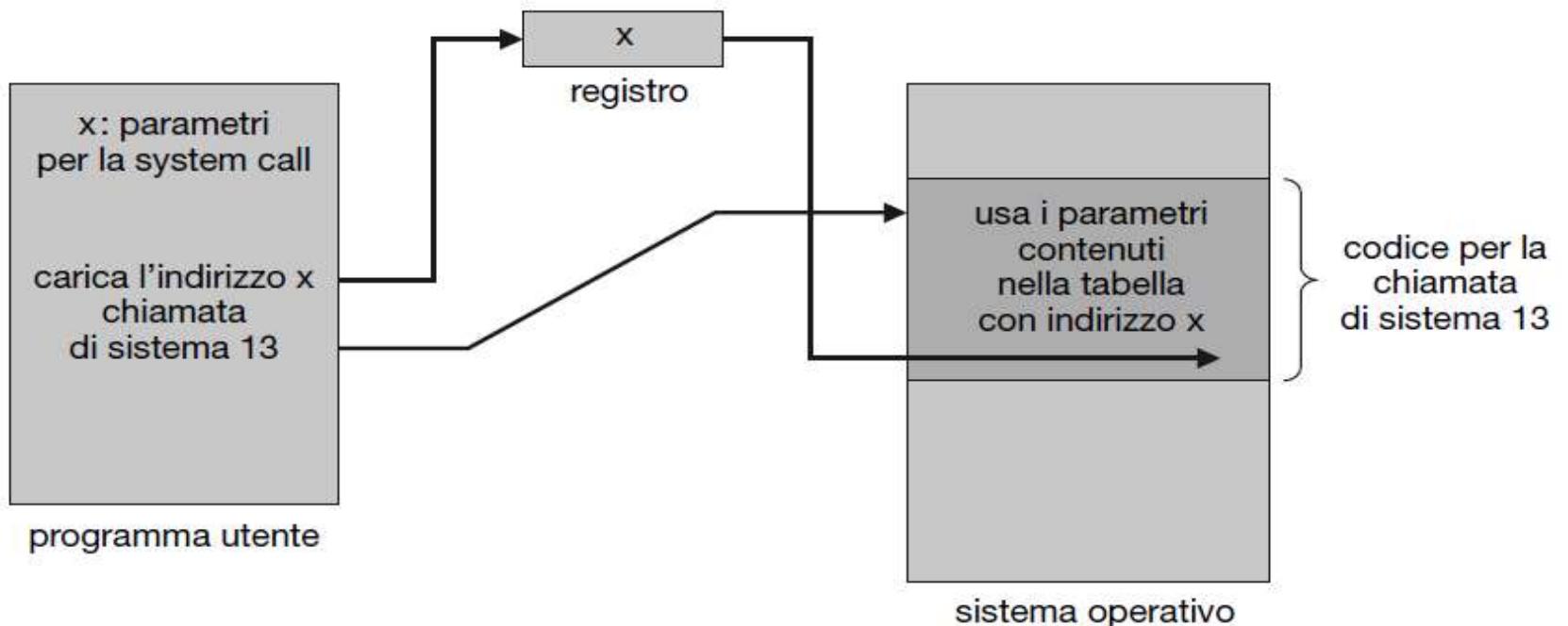
- Tipicamente un numero è associato per ogni chiamata (la **system-call interface** mantiene una tabella indicizzata)
- L'interfaccia invoca la chiamata nel kernel e restituisce lo status e valore di ritorno (dettagli nascosti gestiti da librerie di supporto)



**Figura 2.6** Gestione della chiamata di sistema `open()` invocata da un'applicazione utente.

# Passaggio dei parametri

- Tre metodi generali utilizzati per passare i parametri al sistema operativo
  - Passa direttamente i parametri nei registri (a volte troppi parametri)
  - Parametri in un blocco, o tabella, in memoria e indirizzo del blocco passato come parametro in un registro (Linux e Solaris)
  - Parametri inseriti o inseriti nello stack dal programma e estratti dallo stack dal sistema operativo



**Figura 2.7** Passaggio di parametri in forma di tabella.

# Tipi di Chiamate di Sistema

---

- Controllo di processi
  - Creare e terminare processi
  - Caricare, eseguire
  - Otttenere e settare attributi di processo
  - Attese di tempo
  - Attese di eventi/segnali
  - Allocare memoria
  - Dump memory se errore
  - Debugger per determinare bug
  - Meccanismi di regolazione per gestire l'accesso a dati condivisi

# Tipi di Chiamate di Sistema

---

- Controllo di processi

- Esempi POSIX:

```
#include <sys/types.h>
include <unistd.h>

pid_t getpid(void); identificativo del processo
pid_t getppid(void); identificativo del genitore

pid_t fork(void); pid_t wait(int *status);
pid_t vfork(void);

pid_t pid;

if ((pid=fork()) < 0)
 perror("fork"), exit(1);
else if (pid != 0) {
 // codice del padre
} else {
 // codice del figlio
}
```

# Tipi di Chiamate di Sistema

---

- Gestione di File
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- Gestione di dispositivi
  - request device, release device
  - read, write
  - get device attributes, set device attributes
  - logically attach or detach devices

# Tipi di Chiamate di Sistema

---

- Gestione informazione
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Comunicazione
  - create, delete communication connection
  - send, receive messages if **message passing model** to **host name** or **process name**
    - ▶ da **client** a **server**
  - **Shared-memory** creazione ed accesso a regioni di memoria condivisa
  - trasferire informazione di stato
  - Installare/disinstallare dispositivi remoti

# Tipi di Chiamate di Sistema

---

- Protezione
  - Controllo accesso a risorse
  - Gestione dei permessi
  - Gestione accesso utenti

# Tipi di Chiamate di Sistema

---

- Controllo dei processi
  - creazione e arresto di un processo
  - caricamento, esecuzione
  - terminazione normale e anormale
  - esame e impostazione degli attributi di un processo
  - attesa per il tempo indicato
  - attesa e segnalazione di un evento
  - assegnazione e rilascio di memoria
- Gestione dei file
  - creazione e cancellazione di file
  - apertura, chiusura
  - lettura, scrittura, posizionamento
  - esame e impostazione degli attributi di un file
- Gestione dei dispositivi
  - richiesta e rilascio di un dispositivo
  - lettura, scrittura, posizionamento
  - esame e impostazione degli attributi di un dispositivo
  - inserimento logico ed esclusione logica di un dispositivo
- Gestione delle informazioni
  - esame e impostazione dell'ora e della data
  - esame e impostazione dei dati del sistema
  - esame e impostazione degli attributi dei processi, file e dispositivi
- Comunicazione
  - creazione e chiusura di una connessione
  - invio e ricezione di messaggi
  - informazioni sullo stato di un trasferimento
  - inserimento ed esclusione di dispositivi remoti
- Protezione
  - visualizzazione dei permessi di un file
  - impostazione dei permessi di un file

**Figura 2.8** Tipi di chiamate di sistema.

# Esempi chiamate di sistema in Windows e Unix

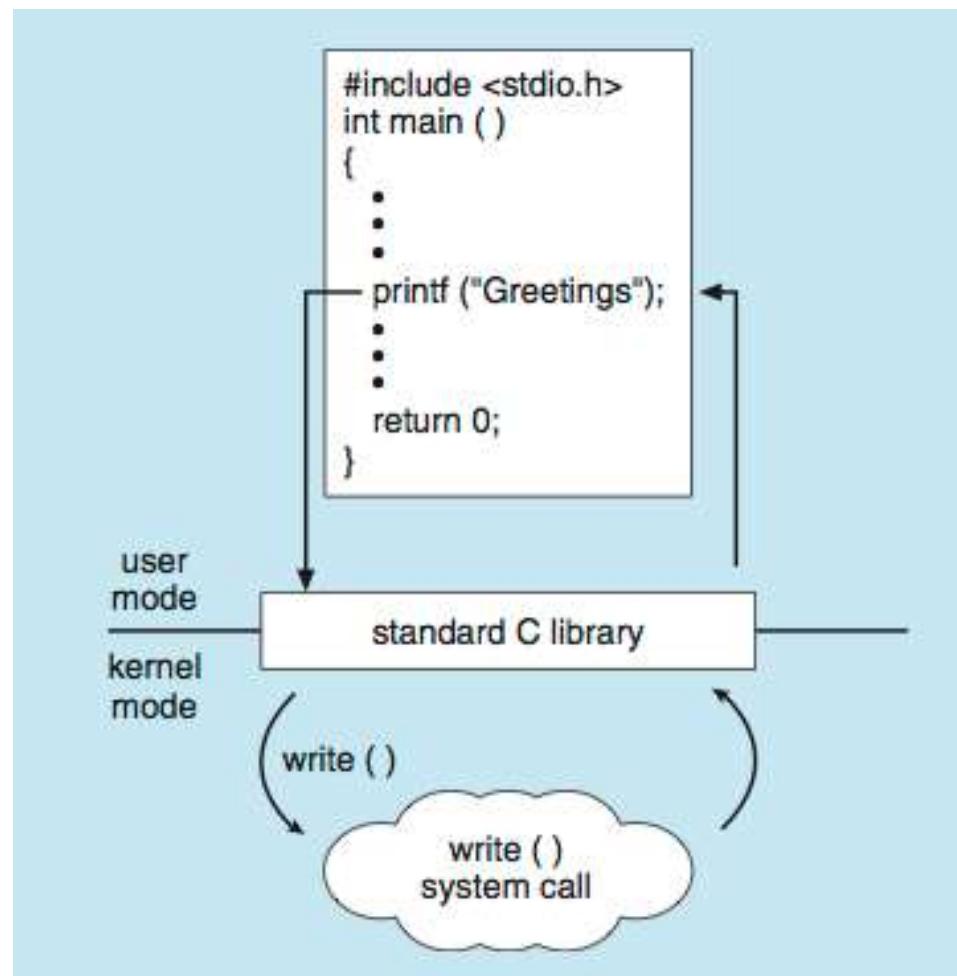


## ESEMPIO DI CHIAMATE DI SISTEMA DI WINDOWS E UNIX

|                                    | <b>Windows</b>                                                                      | <b>UNIX</b>                            |
|------------------------------------|-------------------------------------------------------------------------------------|----------------------------------------|
| <b>Controllo dei processi</b>      | CreateProcess()<br>ExitProcess()<br>WaitForSingleObject()                           | fork()<br>exit()<br>wait()             |
| <b>Gestione dei file</b>           | CreateFile()<br>ReadFile()<br>WriteFile()<br>CloseHandle()                          | open()<br>read()<br>write()<br>close() |
| <b>Gestione dei dispositivi</b>    | SetConsoleMode()<br>ReadConsole()<br>WriteConsole()                                 | ioctl()<br>read()<br>write()           |
| <b>Gestione delle informazioni</b> | GetCurrentProcessID()<br>SetTimer()<br>Sleep()                                      | getpid()<br>alarm()<br>sleep()         |
| <b>Comunicazione</b>               | CreatePipe()<br>CreateFileMapping()<br>MapViewOfFile()                              | pipe()<br>shm_open()<br>mmap()         |
| <b>Protezione</b>                  | SetFileSecurity()<br>InitializeSecurityDescriptor()<br>SetSecurityDescriptorGroup() | chmod()<br>umask()<br>chown()          |

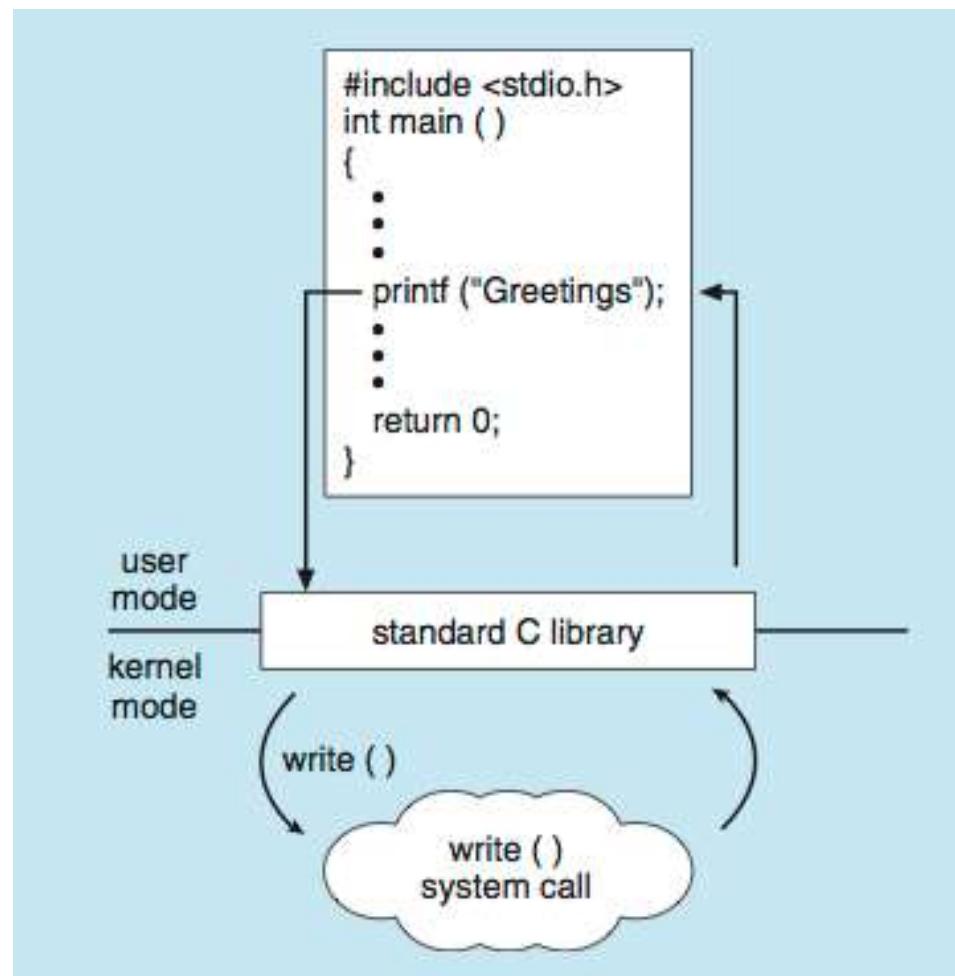
# Standard C Library

- Programma in C invoca una chiamata di libreria printf() che chiama la chiamata di sistema write()



# Standard C Library

- Programma in C invoca una chiamata di libreria printf() che chiama la chiamata di sistema write()



# Programmi di Sistema

---

- I programmi di sistema forniscono un ambiente conveniente per lo sviluppo e l'esecuzione del programma
- Possono essere suddivisi in:
  - Manipolazione di file
  - Informazioni sullo stato
  - Supporto del linguaggio di programmazione
  - Caricamento ed esecuzione del programma
  - Comunicazioni
  - Servizi in background
  - Programmi applicativi
- La maggior parte degli utenti di un SO usa soprattutto i programmi di sistema, meno le effettive chiamate di sistema

# Programmi di Sistema

---

- Forniscono un ambiente conveniente per lo sviluppo e l'esecuzione del programma
  - Alcuni di essi sono semplicemente interfacce utente per le chiamate di sistema; altri sono notevolmente più complessi
- **Gestione dei file:** crea, elimina, copia, rinomina, stampa, scarica, elenca e generalmente manipola file e directory
- **Informazioni sullo stato**
  - Informazioni al sistema: data, ora, quantità di memoria disponibile, spazio su disco, numero di utenti
  - Informazioni dettagliate su prestazioni, registrazione e debug
  - In genere, questi programmi formattano e stampano l'output sul terminale o su altri dispositivi di output

# Programmi di Sistema

---

- **Modifica file**
  - Editor di testo per creare e modificare file
  - Comandi speciali per cercare contenuti di file o eseguire trasformazioni del testo
- **Supporto del linguaggio di programmazione** - compilatori, assemblatori, debugger ed interpreti
- **Caricamento ed esecuzione del programma** - una volta assemblato o compilato un programma, deve essere caricato in memoria per essere eseguito. Il sistema può fornire loader, linkage editor, sistemi di debug, etc.
- **Comunicazioni** - meccanismi per creare connessioni virtuali tra processi, utenti e sistemi informatici

# Programmi di Sistema

---

## □ Servizi di Background

- Lanciati a tempo di boot
  - ▶ Alcuni per lo startup, poi terminate
  - ▶ Altri continuano a girare
- I programmi di sistema in costante esecuzione si dicono **servizi, sottosistemi, demoni**
- Esempi sono monitoraggio di errori, servizi di stampa, etc.
- Eseguiti in contesto user, non kernel

# Progettazione ed Implementazione di un SO

---

- La struttura di un SO può variare molto da sistema a sistema
- La progettazione parte da obiettivi e specifiche
- Dipende da hardware e tipo di sistema
- Distinguiamo **obiettivi utente** e **obiettivi di sistema**
  - Utente
    - ▶ comodo da usare, facile da imparare, affidabile, sicuro e veloce
  - Sistema
    - ▶ facile da progettare, implementare e mantenere, nonché flessibile, affidabile, privo di errori ed efficiente

# Progettazione ed Implementazione di un SO

---

- Distinguere:
  - **Politica:** **cosa** fare?
  - **Meccanismo:** **come** fare?
- La separazione della politica dal meccanismo consente flessibilità se le decisioni politiche devono essere modificate in un secondo momento
  - Es. Il Timer è un meccanismo di protezione della CPU, quale latenza è una politica
  - Se Meccanismo e Politica sono separati un meccanismo può supportare più politiche
    - ▶ Esempio meccanismo delle priorità che può supportare più politiche (preferenza di I/O-intensive vs CPU-intensive)
- Specificare e progettare un sistema operativo è un compito altamente creativo per l'ingegneria del software

# Implementazione

---

- Molto variabile
  - Primi SO in assembly language
  - Poi linguaggi di sistema come Algol, PL/1
  - Ora C, C++
- In realtà mix di linguaggi
  - Livelli più bassi in assembly
  - Corpo principale in C
  - Programmi di sistema in C, C++, linguaggi di scripting come PERL, Python, shell script
- Più alto livello più semplice è il **porting** verso altri hardware
  - Meno ottimizzato “sartorialmente”, quindi meno performante
  - Ma solo alcuni componenti critici:
    - ▶ interrupt handlers, I/O manager, memory manager, CPU scheduler
- **Emulazione** può permettere ad un SO di girare su hardware non-nativo

# Virtualizzazione

---

- Permette agli OS di lanciare applicazioni di altri OS
- **Emulazione** con CPU diverse e codice non nativo
  - Metodo più lento, non compilazione, ma **interpretazione**
- **Virtualizzazione** – sistema operativo compilato in modo nativo per la CPU, che esegue anche sistemi operativi guest compilati in modo nativo
  - **VMM** (Virtual Machine Manager/Monitor) fornisce servizi di virtualizzazione
  - Tipo 1:
    - VMM come Sistema operativo host, comunica direttamente con hardware (es. KVM, Oracle VR Server)
  - Tipo 2:
    - VMM installato sopra un SO host (es. VMWare, Oracle VirtualBox)

# Virtualizzazione

- **Virtualizzazione** – sistema operativo compilato in modo nativo per la CPU, che esegue anche sistemi operativi guest compilati in modo nativo
  - **VMM** (Virtual Machine Manager/Monitor) fornisce servizi di virtualizzazione
  - **Tipo 1:**
    - VMM come Sistema operativo host, comunica direttamente con hardware (es. KVM, Oracle VR Server)
  - **Tipo 2:**
    - VMM installato sopra un SO host (es. VMWare, Oracle VirtualBox)

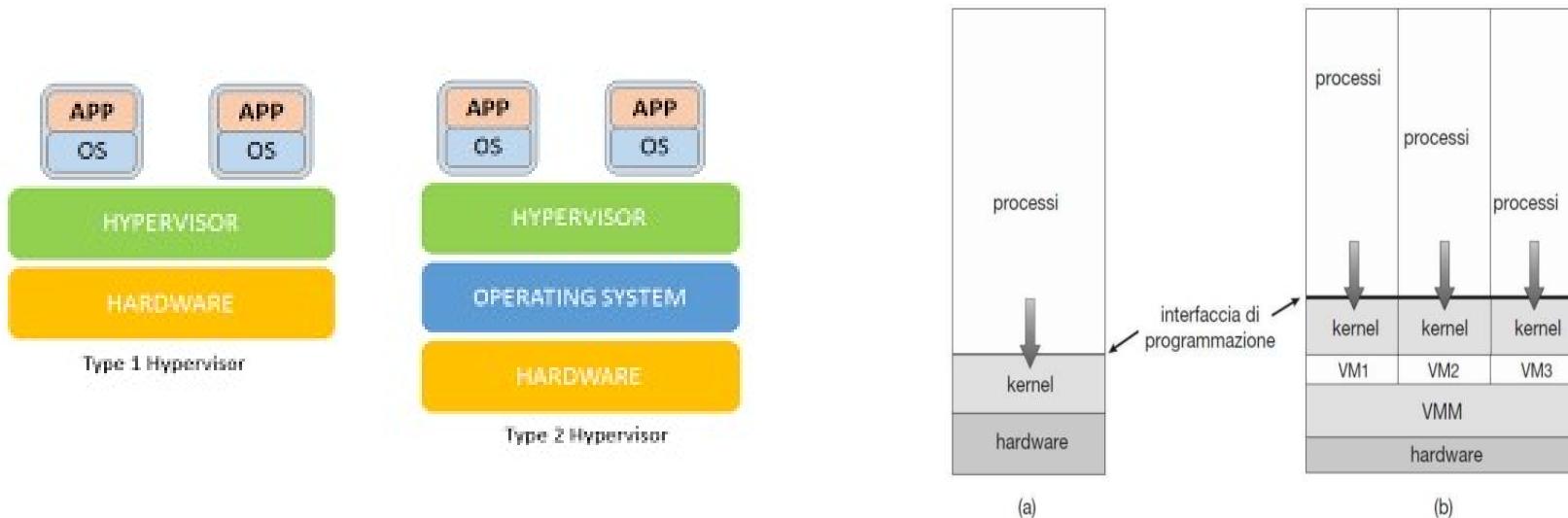


Figura 1.16 Un computer che ha in esecuzione (a) un singolo sistema operativo e (b) tre macchine virtuali.

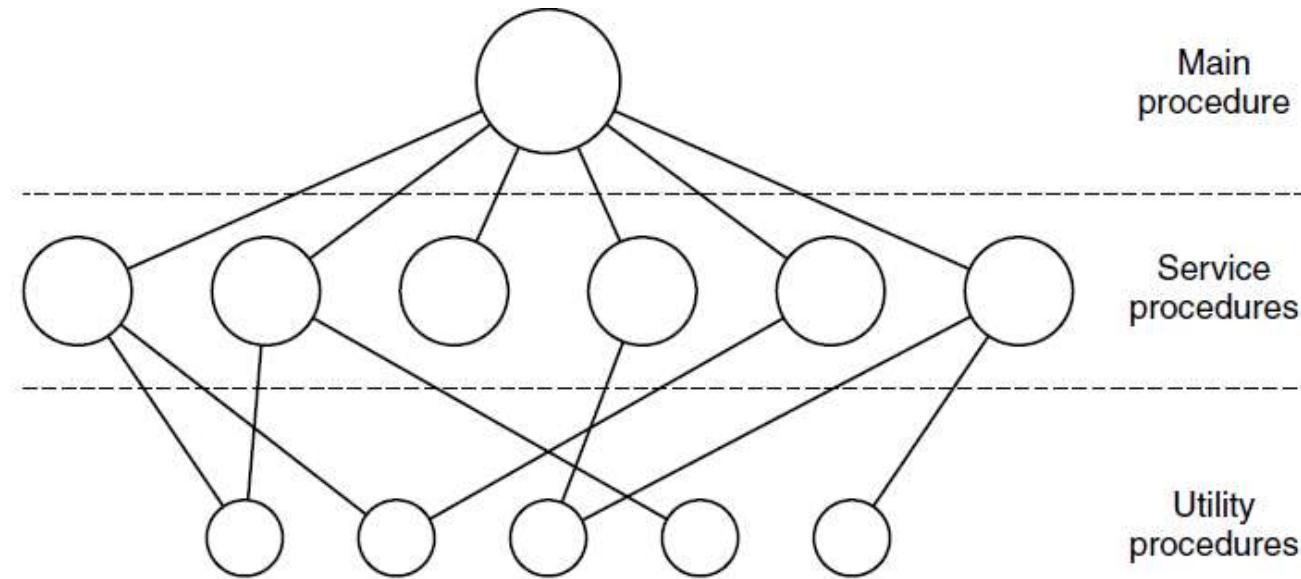
# Strutture di un Sistema Operativo

---

- Un Sistema Operativo è un software molto complesso
- Vari modi di strutturarolo
  - Sistemi monolitici
    - Struttura semplice – MS-DOS
    - Più complessa -- UNIX
  - Sistemi Stratificati
  - Sistemi a micro-kernel

# Sistemi Monolitici

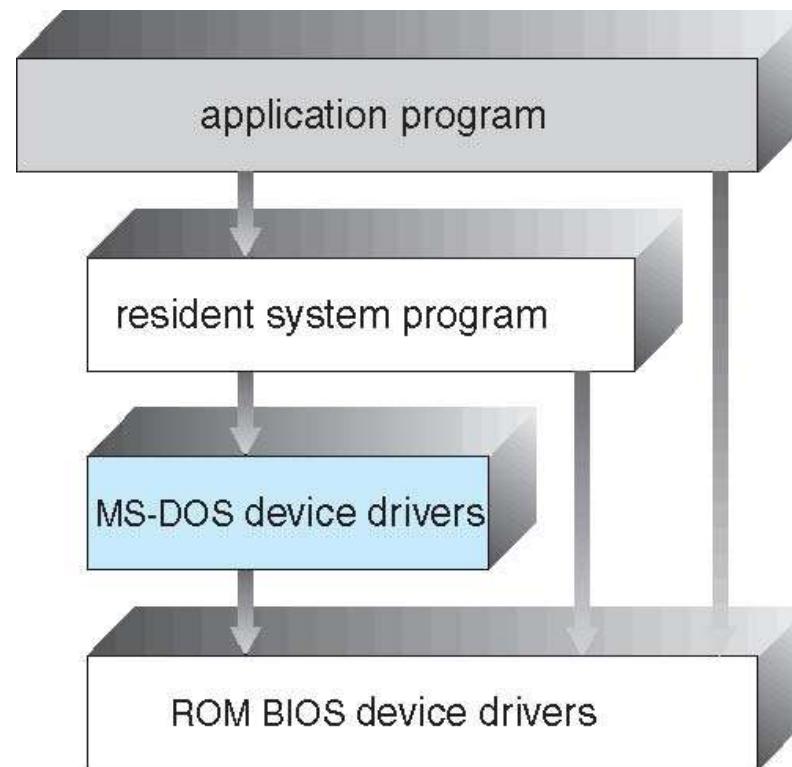
- Sistema Operativo è costituito da una collezione di procedure ognuna delle quali può chiamare qualsiasi altra
  - Un sistema monolitico viene anche chiamato sistema strettamente accoppiato (tightly coupled)
  - Difficili da sviluppare ed estendere
  - Alte prestazioni



Tanenbaum & Bo, Modern Operating Systems:4th ed., (c) 2013 Prentice-Hall, Inc. All rights reserved.

# Struttura Semplice - MS-DOS

- MS-DOS – funzionalità nel minimo spazio
  - Monoutente e monotask
  - Non suddiviso in moduli
  - Sebbene MS-DOS abbia una struttura, le interfacce ed livelli di funzionalità non sono ben separati
  - Programmi utente accedono a tutti i livelli



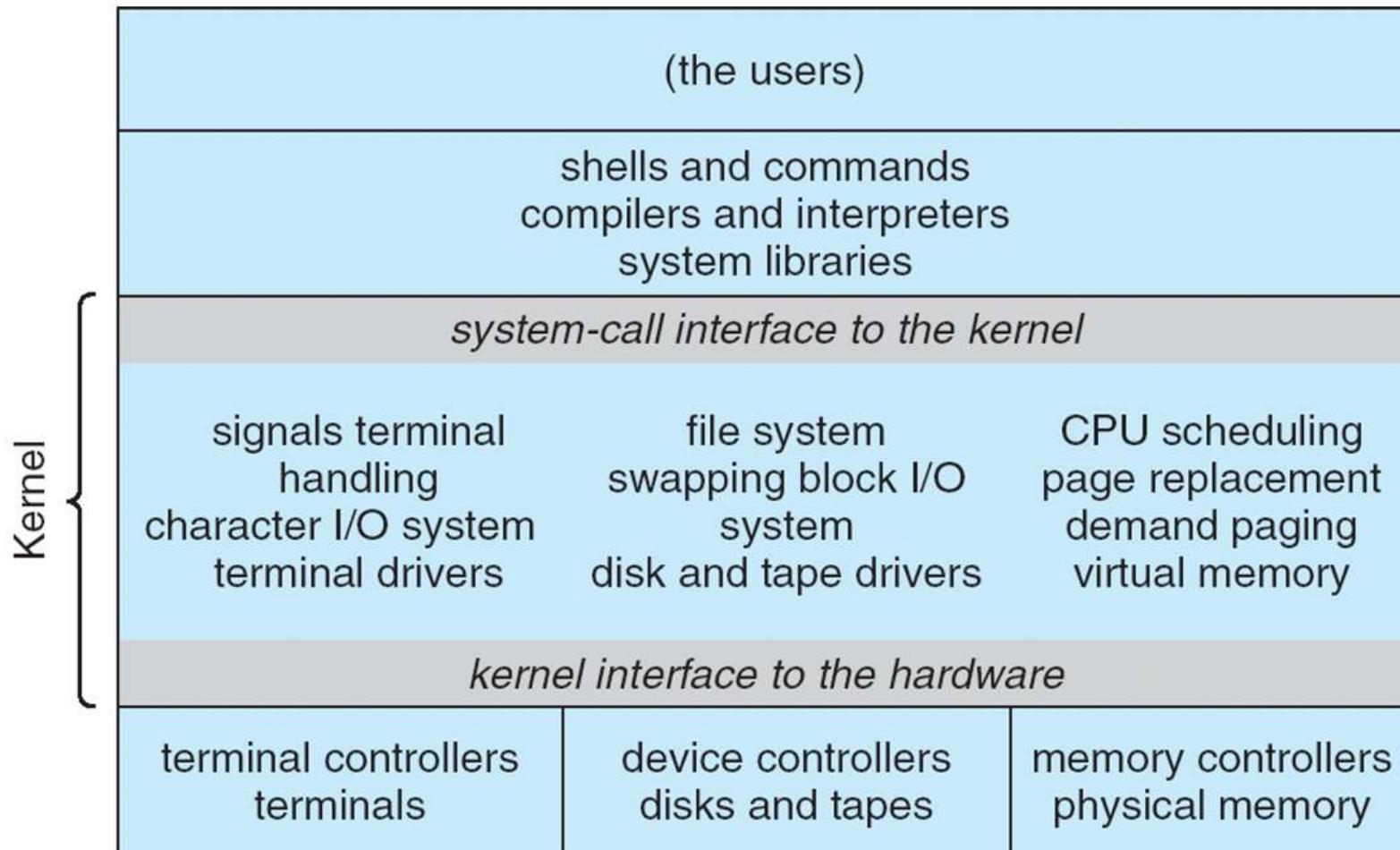
# Struttura non Semplice - UNIX

---

- UNIX: limitato dalla funzionalità hardware, il sistema operativo UNIX originale aveva una strutturazione limitata
- Il sistema operativo UNIX è costituito da due parti separabili
  - Programmi di sistema
  - Kernel
    - ▶ Tutto ciò che si trova al di sotto dell'interfaccia di chiamata di sistema e al di sopra dell'hardware fisico
    - ▶ Fornisce il file system, schedulazione della CPU, gestione della memoria e altre funzioni del sistema operativo (alto numero di funzioni in un livello)
    - ▶ Interfaccia standard per chiamate di sistema (POSIX)

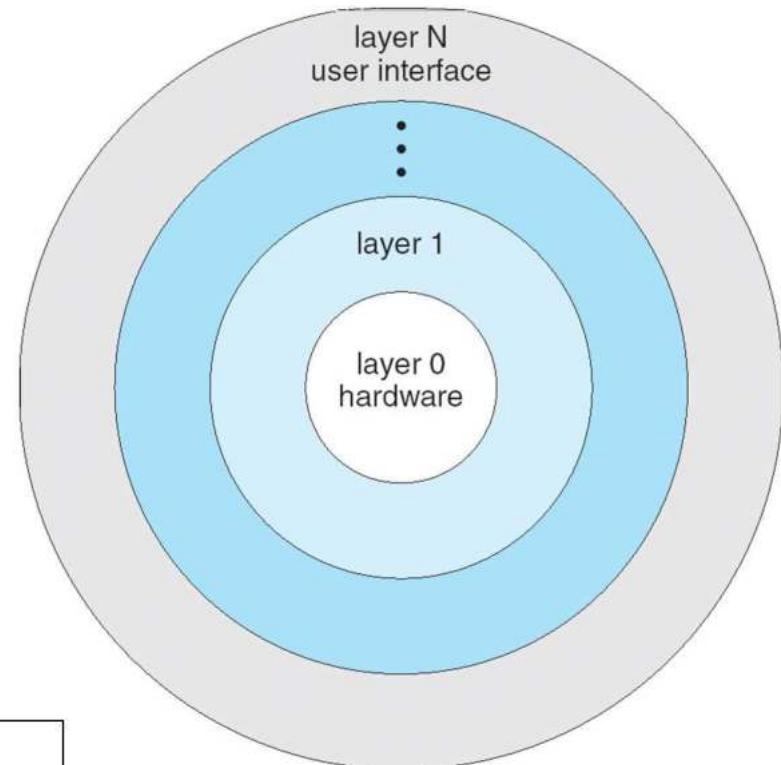
# Struttura di un Sistema UNIX

Architettura non semplice, ma non strutturato



# Approccio Stratificato

- Il sistema operativo è suddiviso in un numero di livelli (levels), ciascuno costruito sopra i livelli inferiori.
- Lo strato inferiore (layer 0), è l'hardware; il più alto (layer N) è l'interfaccia utente.
- Con la modularità ciascuno layer utilizza funzioni (operazioni) e servizi dei layer inferiori

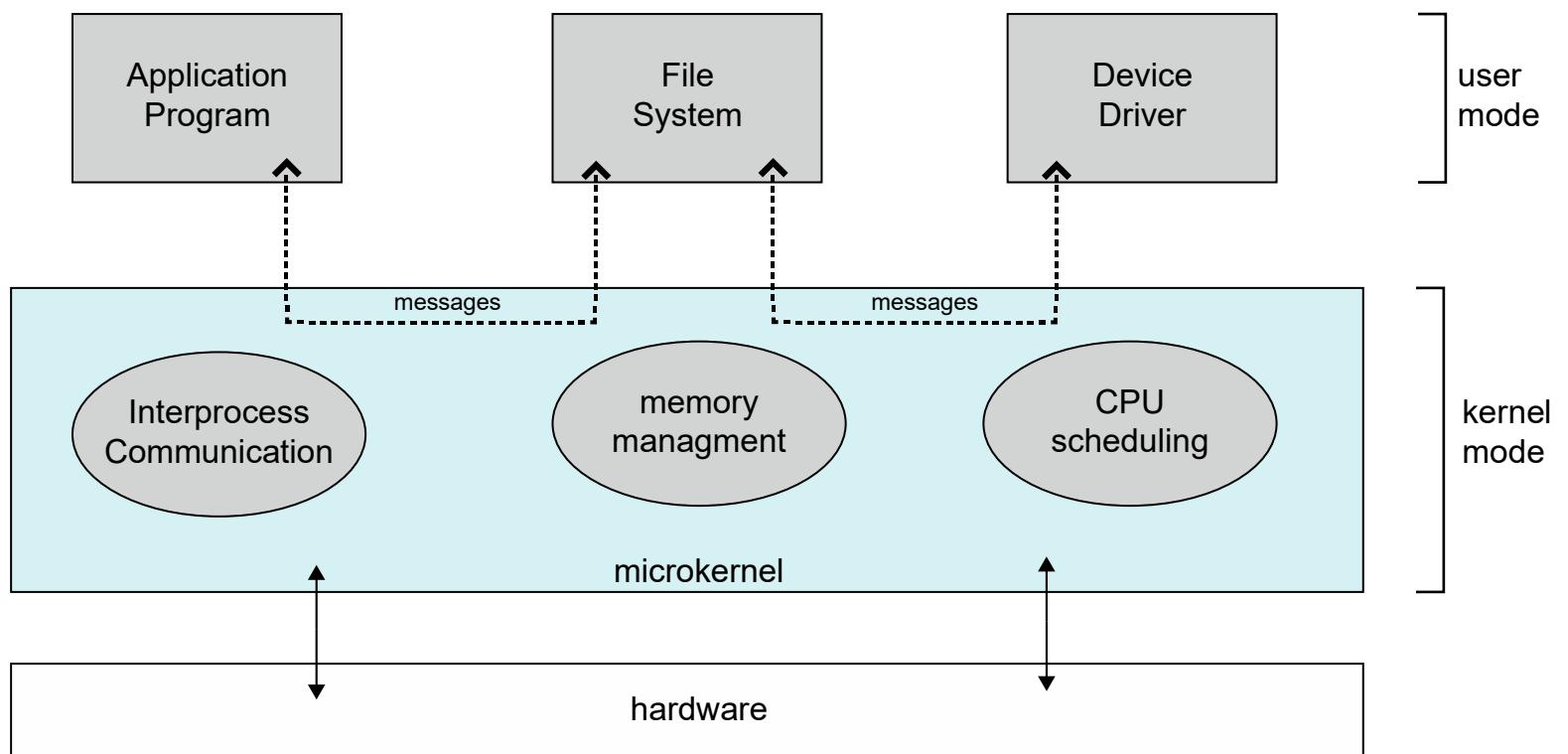


| Layer | Function                                  |
|-------|-------------------------------------------|
| 5     | The operator                              |
| 4     | User programs                             |
| 3     | Input/output management                   |
| 2     | Operator-process communication            |
| 1     | Memory and drum management                |
| 0     | Processor allocation and multiprogramming |

THE realizzato alla Technische Hogelschool Eindhoven in Olanda da Dijkstra nel 1968 per il computer Electrologica X8

# Struttura a Microkernel

- Verso metà anni '80 realizzato **Mach** con kernel strutturato in moduli secondo il cosiddetto **orientamento a microkernel**
  - Mac OS X kernel (**Darwin**) parzialmente basato su Mach
- Spostare più possibile dal kernel allo spazio utente
  - Funzioni di comunicazione tra i programmi client e i vari servizi in esecuzione nello spazio utente



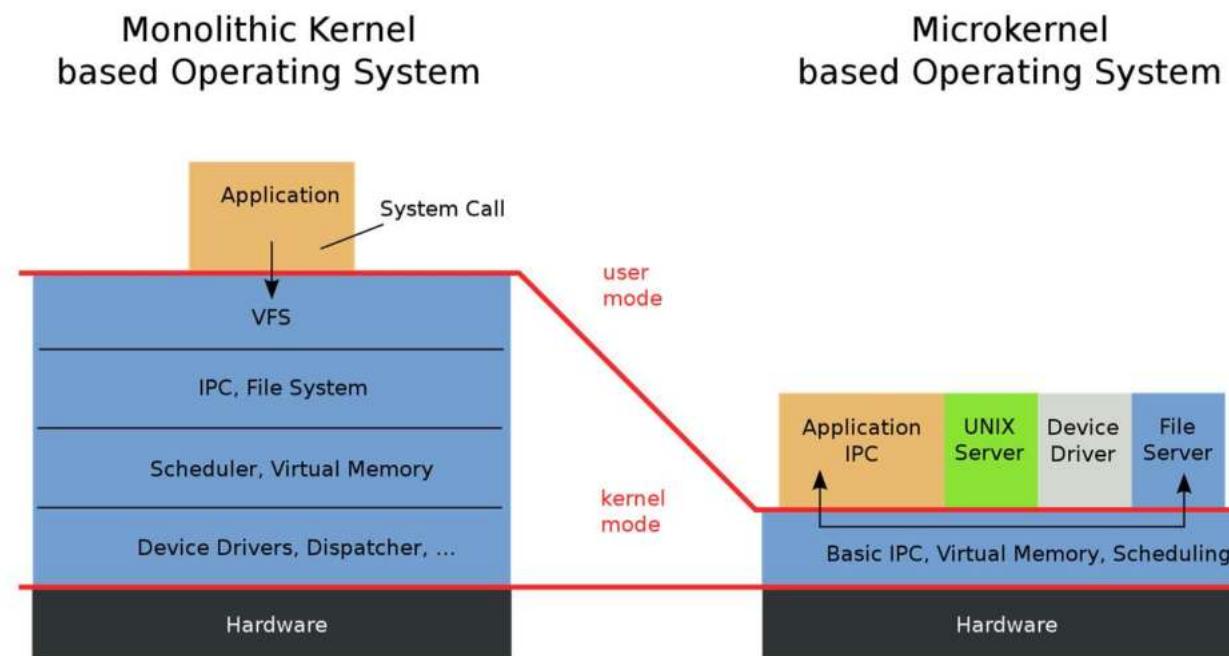
# Struttura a Microkernel

---

- Verso metà anni '80 realizzato **Mach** con kernel strutturato in moduli secondo il cosiddetto **orientamento a microkernel**
  - Mac OS X kernel (**Darwin**) parzialmente basato su Mach
- Spostare più possibile dal kernel allo spazio utente
  - Funzioni di comunicazione tra i programmi client e i vari servizi in esecuzione nello spazio utente
  - Separazione massima tra meccanismi e politiche
  - Kernel minimale:
    - meccanismo di comunicazione tra processi
    - gestione della memoria e dei processi
    - gestione dell'hardware di basso livello (driver)
    - tutto il resto gestito da processi in spazio utente (e.g., politiche di gestione file system, scheduling, memoria sono implementate da processi utente)

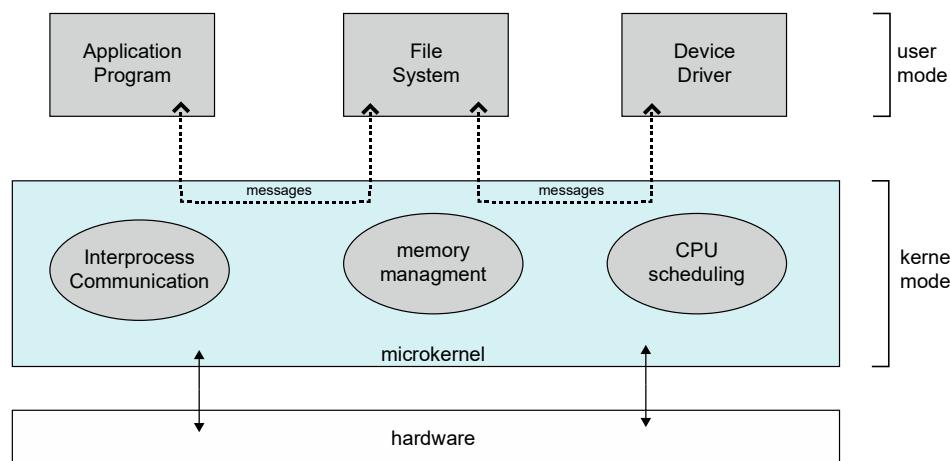
# Struttura a Microkernel

- Verso metà anni '80 realizzato **Mach** con kernel strutturato in moduli secondo il cosiddetto **orientamento a microkernel**
  - Mac OS X kernel (**Darwin**) parzialmente basato su Mach
- Spostare più possibile dal kernel allo spazio utente
  - Funzioni di comunicazione tra i programmi client e i vari servizi in esecuzione nello spazio utente
  - Kernel minimale:



# Struttura a Microkernel

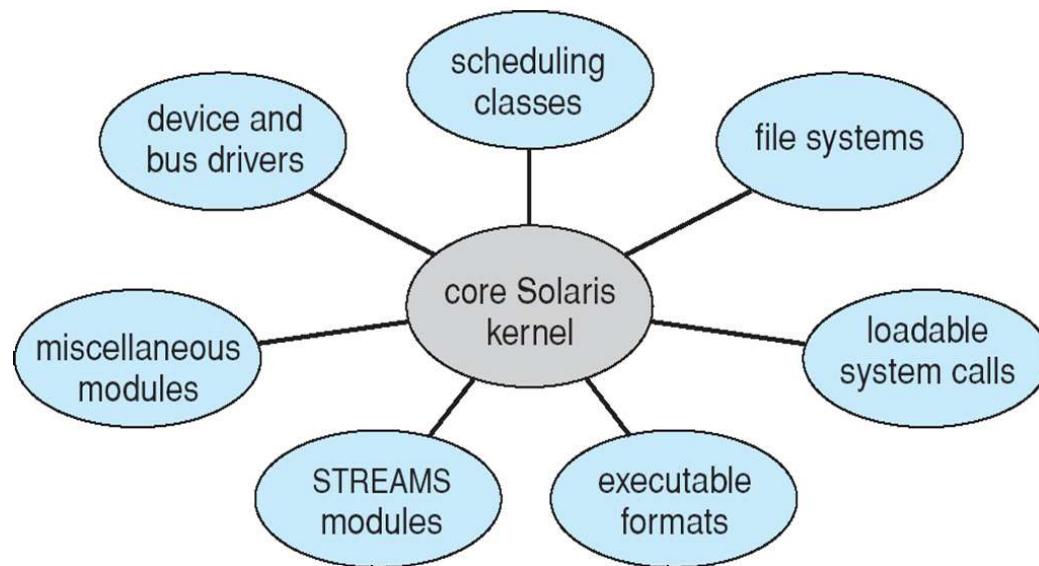
- Spostare più possibile dal kernel allo spazio utente
- Comunicazione tra moduli utente mediante **message passing**
- Vantaggi:
  - Facilità di estensione microkernel
  - Facilità di trasferimento dell'S.O. su nuove architetture
  - Più affidabile (meno codice eseguito in modalità kernel)
  - Più sicuro
- Svantaggi:
  - Sovraccarico delle prestazioni della comunicazione tra lo spazio utente e lo spazio del kernel



# Moduli

---

- Molti sistemi operativi moderni implementano moduli del kernel caricabili
  - Utilizza un approccio orientato agli oggetti
  - Ogni componente principale è separato
  - Ciascuno parla con gli altri tramite interfacce conosciute
  - Ciascuno è caricabile secondo necessità all'interno del kernel
- Simile al sistema a livelli ma con più flessibilità
- Simile a microkernel ma più efficiente perché non richiesto invio di messaggi



# Sistemi Ibridi

---

- La maggior parte dei sistemi operativi moderni non sono in realtà un modello puro
  - Il sistema ibrido combina più approcci per soddisfare le esigenze di prestazioni, sicurezza e usabilità
  - Kernel Linux e Solaris in un singolo spazio di indirizzi, quindi monolitico, modulari per il caricamento dinamico di funzionalità
  - Windows per lo più monolitiche, microkernel per diverse parti del sottosistema
  - Apple mac OS ibrido, stratificato

# Struttura a Irida

- Il sistema operativo **macOS** di Apple è progettato per funzionare principalmente su *computer desktop* e *laptop*, mentre **iOS** è un sistema operativo mobile progettato per *iPhone* e *iPad*
- Strato dell'interfaccia utente (**user experience**)
- Strato degli **ambienti applicativi**
- Strato degli **Ambienti di base**
- Strato degli **Ambiente kernel**

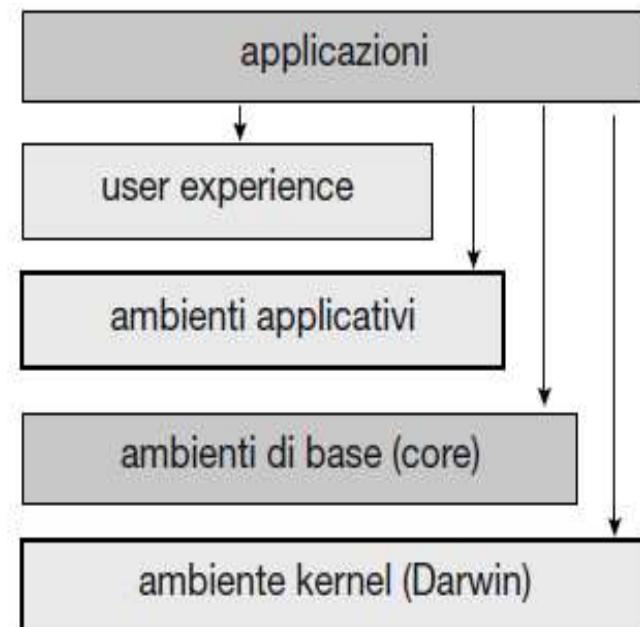


Figura 2.16 Architettura dei sistemi operativi macOS e iOS di Apple.

# Struttura a Irida

- Darwin è un sistema a strati costituito principalmente dal microkernel Mach e dal kernel BSD UNIX
  - Apple ha rilasciato il sistema operativo Darwin come **open-source**
  - Kernel extensions (kext) per modularità
  - iokit per sviluppo drivers

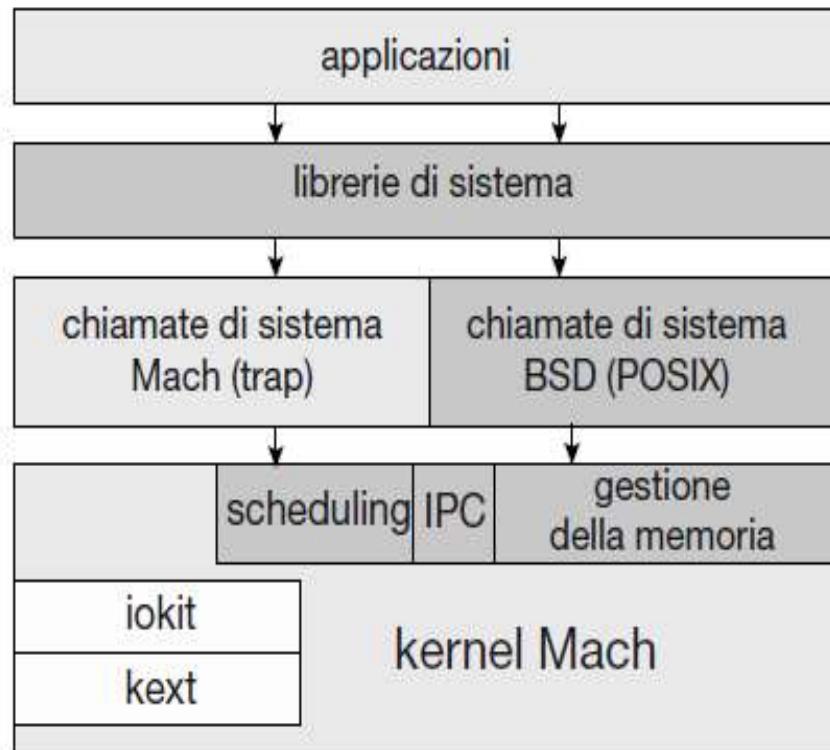


Figura 2.17 La struttura di Darwin.

# Android

---

- Sviluppato da Open Handset Alliance (Google) Open Source
- Basato su Linux kernel (modificato)
- Eseguito su diversi dispositivi: **strato di astrazione hardware** detto **HAL** (*hardware abstraction layer*)
- Applicazioni in Java (Andorid API) su Android RunTime (ART) o Java Native Interface (JNI)
- Ambiente runtime include librerie di base e Dalvik virtual machine
- Bionic standard C Lib per Android

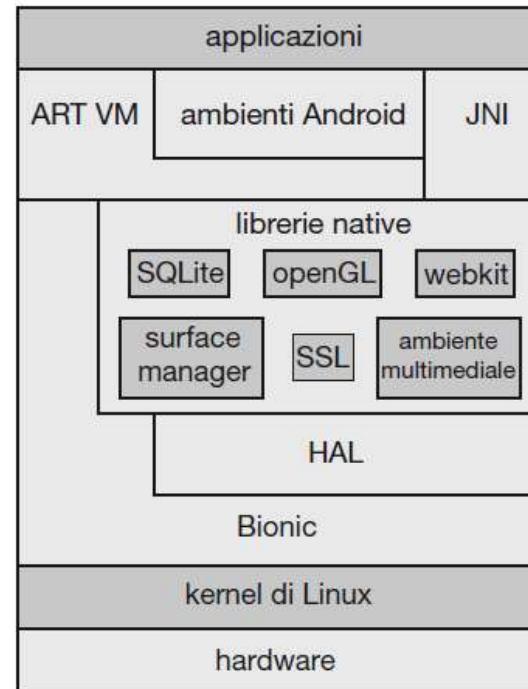


Figura 2.18 Architettura di Google Android.

# Generazione del SO

---

1. Sviluppa il codice del Sistema (oppure ottieni il codice sorgente)
2. Configura il Sistema Operativo per la piattaforma di destinazione
3. Compila l'ISO
4. Installa l'ISO
5. Fai il boot del nuovo SO

Esempio: The Linux Kernel Archives - [www.kernel.org](http://www.kernel.org)

The Linux Kernel Organization is a California Public Benefit Corporation established in 2002 to distribute the Linux kernel and other Open Source software to the public without charge

# Generazione del SO

---

SO progettati per funzionare su più macchine, occorre configurare per ogni macchina

Il programma **SYSGEN** (System Generator) ottiene informazione sulla configurazione del sistema hardware

- ▣ Usato per creare un kernel system-specific o system-tuned

# Boot di Sistema

---

- All'accensione l'esecuzione parte da una locazione di memoria fissata
  - Firmware ROM contiene il codice iniziale di boot
  - Piccolo frammento di codice – **bootstrap loader**, in **ROM** o **EEPROM** localizza il kernel, lo carica in memoria e lo avvia
  - A volte processa a due passi: il **boot block** è in locazione fissata e codice in ROM per caricare il bootstrap loader dal disco (BIOS e boot block)
  - La partizione del disco che contien il boot block è la partizione di avvio (**boot partition**)
- Oltre a caricare il kernel il programma di bootstrap
  - Power-on self-test (POST) - inizializza l'hardware (CPU, memoria e disposititivi)
  - Caricamento del Kernel
  - Il SO va in running