Cenni storici – 1

- **Anni ’50**: i S/O hanno origine con i primi elaboratori a programma memorizzato
  - Modalità di esecuzione *batch* (a lotti) gestita da un operatore umano
    - **Tutto** l’elaboratore a disposizione di **un solo** programma per tutto il tempo della sua esecuzione
    - Immissione di un programma mediante interruttori binari frontali o schede perforate
    - Emissione dei risultati mediante lampadine, testo stampato, schede perforate

Cenni storici – 2

- A partire dagli **anni ’60**
  - Nuovi compilatori, nuovi linguaggi di programmazione, nuovi strumenti di sviluppo
- Ancora gestione a lotti
  - Immissione ed emissione di programmi e dati ancora molto laboriose (= molto costose e molto lente)
    - Un programma può essere sia ingresso che uscita di una esecuzione
  - L’operatore umano è ancora necessario per eseguire le operazioni di ingresso / uscita

Cenni storici – 3

- L'esecuzione di più **lavori** in modalità a lotti può essere facilmente gestita da un S/O **residente**
  - Più caricamenti seguiti da una fase ininterrotta di lavoro e dal ritrovamento dei rispettivi risultati
  - Ordine di esecuzione **predeterminato**
    - Quello di caricamento
    - Secondo il livello di privilegio del richiedente
- Operazioni di I/O fino a 1.000 volte più lente dell'elaborazione
  - Esecuzione *off-line*
    - Senza richiedere tempo di elaboratore
  - Sovrapposizione tra I/O ed elaborazione

Cenni storici – 4

- Sovrapposizione sempre più conveniente al crescere delle prestazioni dei dispositivi di I/O
  - Tecnica detta di *spooling* (*Simultaneous Peripheral Operation On-Line*)
  - Si effettua *spooling* quando l'emissione o la ricezione di dati avviene in parallelo all'esecuzione di altri lavori
    - Esempi
      - Invio di una richiesta di stampa
      - Caricamento di un programma
      - Invio di un messaggio di posta elettronica
  - **Senza interrompere** il lavoro in corso

Cenni storici – 5

- **Multiprogrammazione**
  - Desiderabile poter eseguire diversi lavori simultaneamente
    - In ambito mono-processore il parallelismo è solo simulato
  - Occorre controllare l'assegnazione dell'accesso alle risorse della macchina
    - Esempio: per **quanti di tempo** in modalità *time-sharing* sotto il controllo del S/O

Definizione di S/O

- Un insieme di utilità progettate per
  - Offrire all'utente un'astrazione più semplice e potente della macchina *assembler*
    - Concetto di "**macchina virtuale**"
    - Più semplice da usare (p.es., senza bisogno di conoscenze di microprogrammazione ☺)
    - Più potente (p.es., usando la memoria secondaria per realizzare una più ampia memoria principale virtuale)
  - Gestire in maniera **ottimale** le risorse fisiche e logiche dell'elaboratore
    - Ottimalità è la minimizzazione dei tempi di attesa e la massimizzazione dei lavori svolti per unità di tempo

Nozione di processo

- Un processo è un **programma in esecuzione** e corrisponde a
  - L'insieme **ordinato** di **stati** assunti dal programma nel corso dell'esecuzione (sulla sua macchina virtuale)
    - Processo come "automa a stati"
  - L'insieme **ordinato** delle **azioni** effettuate dal programma nel corso dell'esecuzione (sulla sua macchina virtuale)
    - Processo come "attore" (operatore di azioni)

Realizzazione di processo

- Spazio di **indirizzamento logico**
  - La memoria della **macchina virtuale** che il processo può leggere e scrivere (*core*)
    - Memoria virtuale organizzata a pagine e/o segmenti
    - Programma eseguibile
    - Dati del programma
      - Organizzazione dell'informazione in forma di *file*
    - Aree di lavoro
    - Definizione del **contesto di esecuzione** ed area di salvataggio

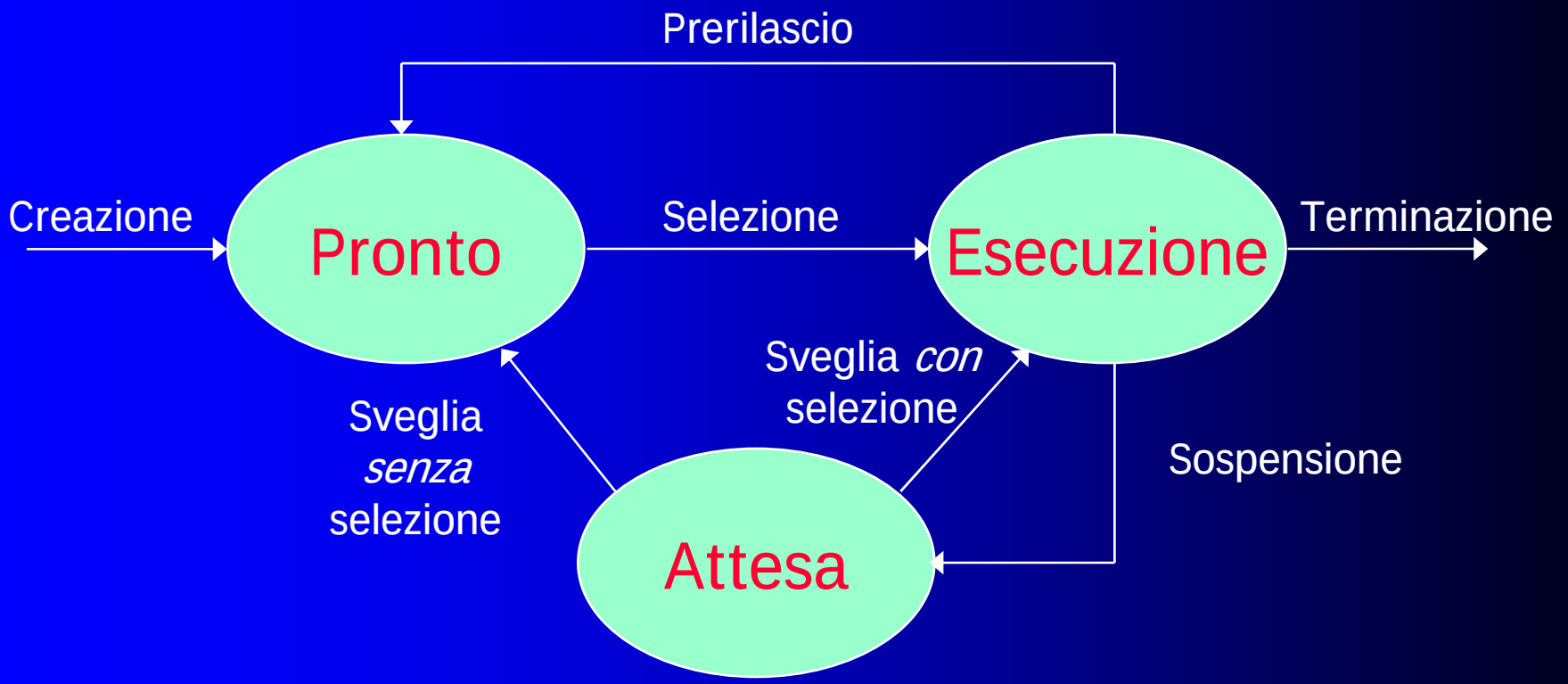
Caratteristiche di processi – 1

- In un sistema coesistono processi utente e di S/O
  - Possono cooperare tra loro ma hanno privilegi diversi
- I processi avanzano concorrentemente
  - Il S/O assegna loro le risorse necessarie secondo diverse politiche di ordinamento
    - A divisione di tempo
    - A livello di priorità (urgenza)
- I processi possono dover comunicare e sincronizzarsi tra loro
  - Il S/O deve fornire i meccanismi e i servizi necessari

Caratteristiche di processi – 2

- Un processo può creare processi “figli”
  - Esempio
    - Un processo interprete di comandi (*shell*) lancia un processo figlio per eseguire un comando di utente
- I processi vengono
  - **Creati** per eseguire un lavoro
  - **Sospesi** per consentire l’esecuzione di altri processi
  - **Terminati** al compimento del lavoro assegnato
    - Un processo figlio che sopravvive alla terminazione del processo padre è detto “orfano” e può essere molto dannoso

Stati di avanzamento di processo



Gestore dei processi – 1

- Costituisce il cuore o nucleo del S/O (*kernel*)
  - Gestisce ed assicura l'avanzamento dei processi
    - Stato di avanzamento
      - In esecuzione, pronto per l'esecuzione, sospeso in attesa di un evento (una comunicazione, la disponibilità di una risorsa, ...)
    - La scelta del processo del eseguire ad un dato istante si chiama ordinamento (*scheduling*)
    - Il gestore decide il cambio di stato dei processi
      - Per divisione di tempo
      - Per trattamento di eventi (p.es., risorsa libera / occupata)

Gestore dei processi – 2

- Compiti del nucleo di S/O
  - Gestire l'avanzamento dei processi
    - Registrando ogni transizione nel loro stato di attivazione
  - Gestire le interruzioni esterne (all'esecuzione corrente) causate da
    - Eventi di I/O
    - Situazioni anomale rilevate da altri processi o componenti del S/O
  - Consentire ai processi di accedere a risorse di sistema e di attendere eventi

Gestore dei processi – 3

- La politica di ordinamento deve essere equa (*fair* → *fairness*)
  - Processi pronti per eseguire devono avere l'opportunità di farlo
  - Processi in attesa di risorse devono avere l'opportunità di accederle
- I meccanismi e servizi di comunicazione e sincronizzazione devono essere efficaci
  - Il dato (o segnale) inviato da un processo mittente deve raggiungere il destinatario in un tempo breve e in modo sicuro

Definizione di risorsa

- **Risorsa** è qualsiasi elemento fisico (*hardware*) o logico (realizzato a *software*) necessario alla creazione, esecuzione e avanzamento di processi
- Le risorse possono essere
  - Durevoli (p.es., CPU)
  - Consumabili (p.es., memoria fisica)
  - Ad accesso divisibile o indivisibile
    - Divisibile se tollera alternanza con accessi di altri processi
    - Indivisibili se *non* tollera alternanza durante l'uso
  - Ad accesso individuale o molteplice
    - Molteplicità fisica o logica (virtualizzata)

Risorsa CPU

- Risorsa indispensabile per l'avanzamento di tutti i processi
- A livello fisico (*hardware*) corrisponde alla CPU
- A livello logico (sotto gestione *software*) può essere vista come una **macchina virtuale**
  - Offerta dal S/O alle sue applicazioni

Risorsa memoria

- Scrittura: risorsa ad accesso individuale
- Lettura: risorsa ad accesso multiplo
- La gestione *software* la **virtualizza** (usandola insieme alla memoria secondaria) attribuendone l'accesso ai vari processi secondo particolari politiche
- Se virtualizzata, diventa riutilizzabile e prerilasciabile
  - Altrimenti consumabile e indivisibile
- Gestione velocizzata con l'utilizzo di supporto *hardware*

Risorsa I/O

- Risorse generalmente riutilizzabili, non prerilasciabili, ad accesso individuale
- La gestione *software* ne facilita l'impiego nascondendone le caratteristiche *hardware* e uniformandone il trattamento
- L'accesso fisico ha bisogno di utilizzare programmi proprietari e specifici
  - ***BIOS***

Caricamento del S/O

Il S/O può risiedere

- Permanentemente in ROM
  - Soluzione tipica di sistemi di controllo industriale e di **sistemi dedicati**
- In memoria secondaria per essere caricato (tutto o in parte) in RAM all'attivazione di sistema (*bootstrap*)
  - Adatto a sistemi di elevata complessità oppure predisposti al controllo (alternativo) da parte di più istanze di S/O
  - In ROM risiede solo il caricatore di sistema (*bootstrap loader*)

Sincronizzazione tra processi – 1

- Processi **indipendenti** possono avanzare concorrentemente senza alcun vincolo di ordinamento reciproco
- In realtà molti processi condividono risorse e informazioni funzionali
  - Per gestire la loro condivisione servono meccanismi di **sincronizzazione di accesso**

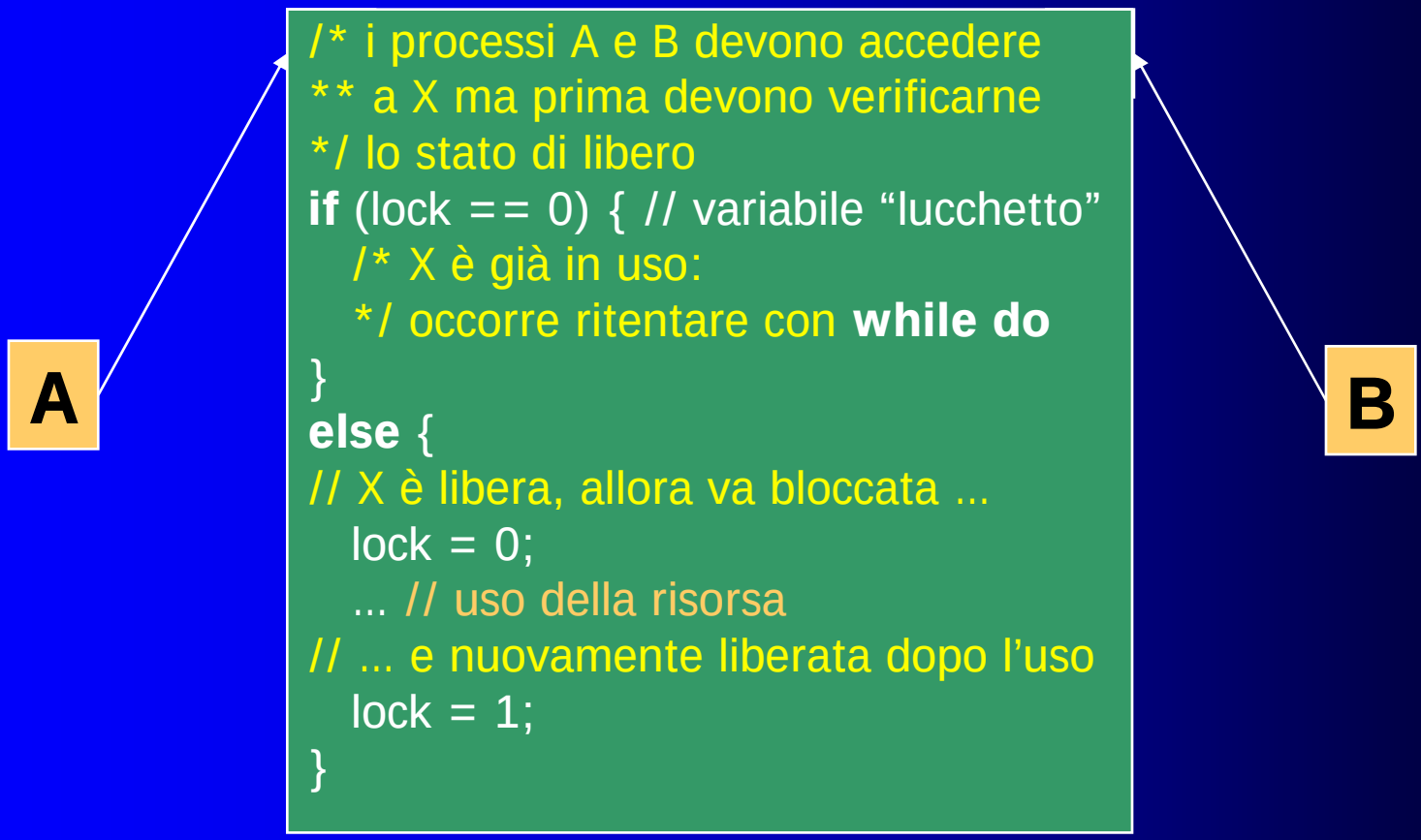
Sincronizzazione tra processi – 2

- Siano A e B due processi che condividono la variabile **X** inizializzata al valore **10**
  - Il processo A deve incrementare **X** di **2** unità
  - Il processo B deve decrementare **X** di **4** unità
- A e B leggono **concorrentemente** il valore di **X**
  - Il processo A scrive in X_A il proprio risultato (**12**)
  - Il processo B scrive in X_B il proprio risultato (**6**)
- Il valore finale in **X** è l'ultimo tra X_A e X_B a essere scritto!
- Il valore atteso in **X** invece era **8**
  - Ottenibile **solo** con sequenze (A;B) o (B;A) **indivise** di lettura e scrittura

Sincronizzazione tra processi – 3

- La modalità di accesso indivisa a una variabile condivisa viene detta **“in mutua esclusione”**
  - L’accesso consentito a un processo inibisce quello simultaneo di qualunque altro processo utente fino al rilascio della risorsa
- Si utilizza una variabile logica “lucchetto” (*lock*) che indica se la variabile condivisa è al momento in uso a un altro processo
  - Detta anche “struttura *mutex*” *(mutual exclusion*)

Sincronizzazione tra processi – 4



Questa soluzione **non** funziona! Perché?

Sincronizzazione tra processi – 5

- La soluzione appena vista è completamente inadeguata
  - Ciascuno dei due processi può essere prerilasciato **dopo** aver letto la variabile *lock* ma **prima** di esser riuscito a modificarla
    - Questa situazione è detta ***race condition*** e può generare pesanti inconsistenze
  - Inoltre l'algoritmo mostrato richiede **attesa attiva** che causa spreco di tempo di CPU a scapito di altre attività a maggior valore aggiunto
    - La tecnica di sincronizzazione tramite attesa attiva viena detta *busy wait*

Sincronizzazione ammissibile

- Una soluzione al problema della sincronizzazione di processi è ammissibile se soddisfa le seguenti 4 condizioni
  1. Garantire accesso esclusivo
  2. Garantire attesa finita
  3. Non fare assunzioni sull'ambiente di esecuzione
  4. Non subire condizionamenti dai processi esterni alla sezione critica

Soluzioni “esotiche” – 1

- Mutua esclusione con variabili condivise tramite **alternanza stretta** tra coppie di processi
  - Non ha bisogno di supporto dalla sua macchina virtuale

- Tre gravi difetti → soluzione non ammissibile!
  - Uso di attesa attiva (*busy wait*)
  - Condizionamento esterno alla sezione critica
  - Rischio di *race condition* sulla variabile di controllo

Soluzioni “esotiche” – 2

- Proposta da G. L. Peterson
  - Migliore (!) della precedente ma ingenua rispetto al funzionamento dei processori di oggi
  - Applica solo a coppie di processi

```
IN(int i) :: {
int j = (i – 1); // l’altro
flag[i] = TRUE;
turn = i;
while (flag[j] && turn == i)
 { // attesa attiva  };
```

```
OUT(int i) :: {
flag[i] = FALSE; }
```

```
Processo (int i) ::

while (TRUE) {
  IN(i);
  // sezione critica
  OUT(i);
  // altre attività
}
```

Soluzioni “esotiche” – 3

- La condizione di uscita in **IN**() impone che il processo `i` resti in attesa attiva fin quando il processo `j` non abbia invocato **OUT**()
  - Per cui `flag[j] = FALSE`
  - Si ha attesa attiva quando non vi sia coerenza tra la richiesta di `turn` e lo stato dell’altro processo
- Su `flag[]` non vi può essere scrittura simultanea che si ha invece su `turn`
  - La condizione di uscita è però espressa in modo da evitare il rischio di *race condition*
    - Non viene decisa solo dal valore assunto da `turn`!

Sincronizzazione tra processi – 6

- Tecniche complementari e/o alternative
  - **Disabilitazione delle interruzioni**
    - Previene il prerilascio dovuto all'esaurimento del quanto di tempo e/o la promozione di processi a più elevata priorità
    - Può essere inaccettabile per sistemi soggetti a interruzioni frequenti
  - Supporto *hardware* diretto: ***Test-and-Set-Lock***
    - Cambiare **atomicamente** valore alla variabile di *lock* se questa segnala "libero"
    - Evita situazioni di *race condition* ma comporta **sempre** attesa attiva

Sincronizzazione tra processi – 7

```
!! Chiamiamo regione critica la zona di programma
!! che delimita l'accesso e l'uso di una variabile
!! condivisa
enter_region:
 TSL R1, LOCK        !! modifica il valore di
                     !! LOCK (se vale 0) e lo pone in R1
 CMP R1, 0           !! verifica l'esito
 JNE enter_region    !! attesa attiva se =0
 RET                 !! altrimenti ritorna al chiamante
                     !! con possesso della regione critica
leave_region:
 MOV LOCK, 0         !! scrive 0 in LOCK (accesso libero)
 RET                 !! ritorno al chiamante
```

Sincronizzazione tra processi – 8

- Soluzione mediante **semaforo**
  - Dovuta a E. W. Dijkstra (1965)
  - Richiede accesso **indiviso** (atomico) alla variabile di controllo detta semaforo
    - Per questo la struttura semaforo si appoggia sopra una macchina virtuale meno potente che fornisce una modalità di accesso indiviso più primitiva
    - Semaforo **binario** (contatore Booleano che vale 0 o 1)
    - Semaforo **contatore** (consente tanti accessi simultanei quanto il valore iniziale del contatore)
  - La richiesta di accesso **P** (**up**) decrementa il contatore se questo non è già 0, altrimenti accoda il chiamante
  - L'avviso di rilascio **V** (**down**) incrementa di 1 il contatore e chiede al *dispatcher* di porre in stato di "pronto" il primo processo in coda sul semaforo

Sincronizzazione tra processi – 9

L'uso di una risorsa condivisa **R** è racchiuso entro le chiamate di **P** e **V** sul semaforo associato a **R**

```
Processo Pi ::
{ // avanzamento
  P(sem);
  /* uso di risorsa
     condivisa R */
  V(sem);
  // avanzamento
}
```

P(sem) viene invocata per richiedere accesso a una risorsa condivisa R

- Quale R tra tutte?

V(sem) viene invocata per rilasciare la risorsa

Sincronizzazione tra processi – 10

- Mediante uso intelligente di semafori binari più processi possono anche **coordinare** l'esecuzione di attività collaborative
  - Esempio con semaforo inizialmente bloccato

```
processo A ::
{// esecuzione indipendente
 …
 P(sem); // attesa di B
 // 2a parte del lavoro
 …
}
```

```
processo B ::
{// 1a parte del lavoro
 …
 V(sem); // rilascio di A
 // esecuzione indipendente
 …
}
```

Sincronizzazione tra processi – 11

- Il **semaforo binario** (*mutex*) è una struttura composta da un campo valore intero e da un campo coda che accoda tutti i **PCB** dei processi in attesa sul semaforo
  - PCB = *Process Control Block*
- L'accesso al campo valore deve essere **atomico**!

```
void P(struct sem){
  if (sem.valore = 1)
     sem.valore = 0; // busy
  else {
     suspend(self, sem.coda);
     schedule();}
}
void V(struct sem){
  sem.valore = 1 ; // free
  if not_empty(sem.coda){
     ready(get(sem.coda));
     schedule();}
}
```

Sincronizzazione tra processi – 12

- Il **semaforo contatore** ha la stessa struttura del *mutex* ma usa una logica diversa per il campo valore
  - (Valore > 0) denota disponibilità non esaurita
  - (Valore < 0) denota richieste pendenti
- Il valore iniziale denota la capacità massima della risorsa

```
void P(struct sem){
  sem.valore -- ;
  if (sem.valore < 0){
     suspend(self, sem.coda);
     schedule();}
}
void V(struct sem){
  sem.valore ++ ;
  if (sem.valore <= 0){
     ready(get(sem.coda));
     schedule();}
}
```

Monitor – 1

- L'uso di semafori a livello di programma è ostico e rischioso
  - Il posizionamento improprio delle **P** può causare situazioni di blocco infinito (*deadlock*) o anche esecuzioni erronee di difficile verifica (*race condition*)
  - È indesiderabile lasciare all'utente il pieno controllo di strutture così delicate

Esempio 1

```
#define N …                  /* posizioni del contenitore   */
typedef int semaforo;        /* P decrementa, V incrementa,
                                il valore 0 blocca la P     */

semaforo mutex = 1;
semaforo non-pieno = N;
semaforo non-vuoto = 0;

void produttore(){
   int prod;
   while(1){
      prod = produci();
      P(&non-pieno);
      P(&mutex);
      inserisci(prod);
      V(&mutex);
      V(&non-vuoto);
   }
}
```

```
void consumatore(){
   int prod;
   while(1){
      P(&non-vuoto);
      P(&mutex);
      prod = preleva();
            V(&mutex);
      V(&non-pieno);
      consuma(prod);
   }
}
```

Il corretto ordinamento di P e V è critico!

Monitor – 2

- Un diverso ordinamento delle **P** nel codice utente di Esempio 1 potrebbe causare situazioni di blocco infinito (*deadlock*)

Codice del produttore

```
P(&mutex);      // accesso esclusivo al contenitore
P(&non-pieno); // attesa spazi nel contenitore
```

- In questo modo il consumatore non può più accedere al contenitore per prelevarne prodotti, facendo spazio per l'inserzione di nuovi → stallo = *deadlock*

Monitor – 3

- Linguaggi evoluti di alto livello (e.g.: Concurrent Pascal, Ada, Java) offrono strutture **esplicite** di controllo delle regioni critiche, originariamente dette *monitor* (Hoare, ’74; Brinch-Hansen, ’75)
- Il *monitor* definisce la regione critica
- Il compilatore (non il programmatore!) inserisce il codice necessario al controllo degli accessi

Monitor – 4

- Un *monitor* è un aggregato di sottoprogrammi, variabili e strutture dati
- Solo i sottoprogrammi del *monitor* possono accederne le variabili interne
- Solo un processo alla volta può essere attivo entro il *monitor*
  - Proprietà garantita dai meccanismi del **supporto a tempo di esecuzione** del linguaggio di programmazione concorrente
    - Funzionalmente molto simile al *kernel* del sistema operativo
  - Il codice necessario è inserito dal compilatore direttamente nel programma eseguibile

Monitor – 5

- La garanzia di mutua esclusione da sola può **non** bastare per consentire sincronizzazione intelligente
- Due procedure operanti su variabili speciali (non contatori!) dette *condition variables*, consentono di modellare **condizioni logiche** specifiche del problema
  - `Wait(<cond>)` // forza l'attesa del chiamante
  - `Signal(<cond>)` // risveglia il processo in attesa
- Il segnale di risveglio **non ha memoria**
  - Va perso se nessuno lo attende

Esempio 2

```
monitor PC
 condition non-vuoto, non-pieno;
 integer contenuto := 0;
 procedure inserisci(prod : integer);
 begin
  if contenuto = N then wait(non-pieno);
  <inserisci nel contenitore>;
  contenuto := contenuto + 1;
  if contenuto = 1 then signal(non-vuoto);
 end;
 function preleva : integer;
 begin
  if contenuto = 0 then wait(non-vuoto);
  preleva := <preleva dal contenitore>;
  contenuto :=  contenuto – 1;
  if contenuto = N-1 then signal(non-pieno);
 end;
end monitor;
```

```
procedure Produttore;
begin
 while true do begin
  prod := produci;
  PC.inserisci(prod);
 end;
end;
```

```
procedure Consumatore;
begin
 while true do begin
  prod := PC.preleva;
  consuma(prod);
 end;
end;
```

Monitor – 6

- La primitiva **`Wait`** permette di bloccare il chiamante qualora le condizioni logiche della risorsa non consentano l'esecuzione del servizio
  - Contenitore pieno per il produttore
  - Contenitore vuoto per il consumatore
- La primitiva **`Signal`** notifica il verificarsi della condizione attesa al (primo) processo bloccato, risvegliandolo
  - Il processo risvegliato compete con il chiamante della **`Signal`** per il possesso della CPU
- **`Wait`** e **`Signal`** sono invocate in mutua esclusione
  - Non si può verificare *race condition*

Monitor – 8

- In ambiente locale si hanno 3 possibilità per supportare sincronizzazione tra processi
  1. Linguaggi concorrenti **con** supporto esplicito per strutture *monitor* (**alto livello**)
  - Linguaggi sequenziali **senza** supporto per *monitor* o semafori
    2. Uso di semafori tramite strutture primitive del sistema operativo e chiamate di sistema (**basso livello**)
    3. Realizzazione di semafori primitivi, in linguaggio *assembler*, senza supporto dal sistema operativo (**bassissimo livello**)
- Monitor e semafori **non sono utilizzabili** per realizzare scambio di informazione tra elaboratori
  - Perché?

Barriere

- Per sincronizzare **gruppi** di processi
  - Attività cooperative suddivise in fasi ordinate
- La barriera blocca **tutti** i processi che la raggiungono fino all'arrivo dell'ultimo
  - Si applica indistintamente ad ambiente locale e distribuito
- Non comporta scambio di messaggi esplicito
  - L'avvenuta sincronizzazione dice implicitamente ai processi del gruppo che tutti hanno raggiunto un dato punto della loro esecuzione

Problemi classici di sincronizzazione

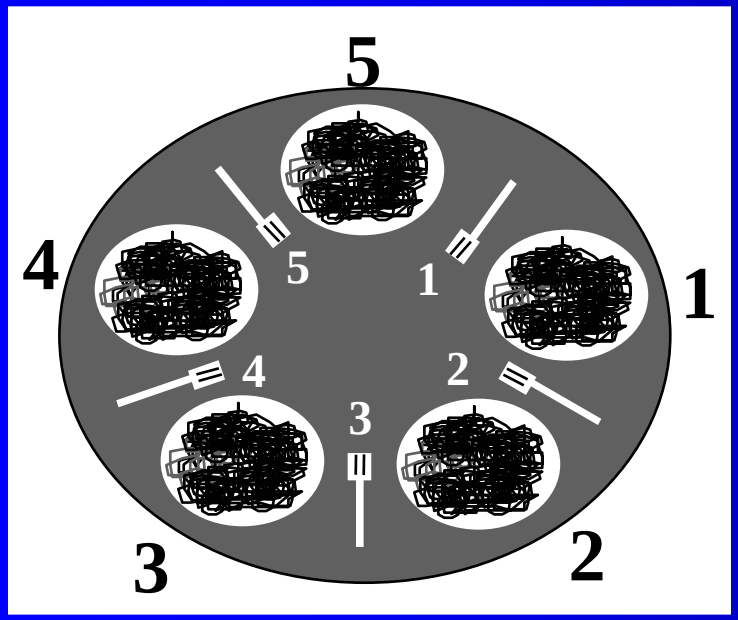
- Metodo per valutare l'efficacia e l'eleganza di modelli e meccanismi per la sincronizzazione tra processi
  - **Filosofi a cena** : accesso esclusivo a risorse limitate
  - **Lettori e scrittori** : accessi concorrenti a basi di dati
  - **Barbiere che dorme** : prevenzione di *race condition*
- Problemi pensati per rappresentare tipiche situazioni di rischio
  - Stallo con blocco (*deadlock*)
  - Stallo senza blocco (*starvation*)
  - Esecuzioni non predicibili (*race condition*)

Filosofi a cena – 1

- N filosofi sono seduti a un tavolo circolare
- Ciascuno ha davanti a se 1 piatto e 1 posata alla propria destra
- Ciascun filosofo necessita di 2 posate per mangiare
- L'attività di ciascun filosofo alterna pasti a momenti di riflessione

Filosofi a cena – 2

Soluzione A con stallo (*deadlock*)




```
void filosofo (int i){
  while (TRUE) {
    medita();
    P(f[i]);
    P(f[(i+1)%N]);
    mangia();
    V(f[(i+1)%N]);
    V(f[i]);
  };
}
```

L'accesso alla prima forchetta non garantisce l'accesso alla seconda!

Ogni forchetta modellata come un semaforo binario

Filosofi a cena – 3

Soluzione B con stallo (*starvation*)

```
void filosofo (int i){
 OK = FALSE;
 while (TRUE) {
  medita();
  while (!OK) {
   P(f[i]);
   if (!f[(i+1)%N]) {
    V(f[i]);
     sleep(T);}
   else {
    P(f[(i+1)%N]);
    OK = TRUE;
   };
  mangia();
  V(f[(i+1)%N]);
  V(f[i]);
 }
```

Un'attesa a durata costante difficilmente genera una situazione differente!

Filosofi a cena – 4

- Il problema ammette diverse soluzioni
  1. Utilizzare in soluzione A un semaforo a mutua esclusione per incapsulare gli accessi a *entrambe* le forchette
     - Funzionamento garantito
  2. In soluzione B, ciascun processo potrebbe attendere un tempo **casuale** invece che fisso
     - Funzionamento non garantito
  3. Algoritmi sofisticati, con maggiore informazione sullo stato di progresso del vicino e maggior coordinamento delle attività
     - Funzionamento garantito

Stallo
Condizioni necessarie e sufficienti

- **Accesso esclusivo a risorsa condivisa**
- **Accumulo di risorse**
  - I processi possono accumulare nuove risorse senza doverne rilasciare altre
- **Inibizione di prerilascio**
  - Il possesso di una risorsa deve essere rilasciato volontariamente
- **Condizione di attesa circolare**
  - Un processo attende una risorsa in possesso del successivo processo in catena

Stallo: prevenzione – 1

- Almeno tre strategie per affrontare lo stallo
  - **Prevenzione**
    - Impedire almeno una delle condizioni precedenti
  - **Riconoscimento e recupero**
    - Ammettere che lo stallo si possa verificare
    - Essere in grado di riconoscerlo
    - Possedere una procedura di recupero (sblocco)
  - **Indifferenza**
    - Considerare trascurabile la probabilità di stallo e **non** prendere alcuna precauzione contro di esso
      - Che succede se esso si verifica?

Stallo: prevenzione – 2

- Bisogna impedire il verificarsi di almeno una delle condizioni necessarie e sufficienti
  - Si può fare staticamente (prima di eseguire) oppure a tempo d'esecuzione
  1. **Accesso esclusivo alla risorsa**
     - Però alcune risorse non consentono alternative
  2. **Accumulo di risorse**
     - Però molti problemi richiedono l'uso simultaneo di più risorse
  3. **Inibizione del prerilascio**
     - Però alcune risorse non consentono di farlo
  4. **Attesa circolare**
     - Difficile da rilevare e complessa da evitare o sciogliere

Stallo: prevenzione – 3

- **Prevenzione sulle richieste di accesso**
  - A tempo d'esecuzione
    1. A ogni richiesta di accesso si verifica se questa possa portare allo stallo
       - In caso affermativo non è però chiaro cosa convenga fare
       - La verifica a ogni richiesta è un onere molto pesante
  - Prima dell'esecuzione
    2. All'avvio di ogni processo si verifica quali risorse essi dovranno utilizzare così da ordinarne l'attività in maniera conveniente

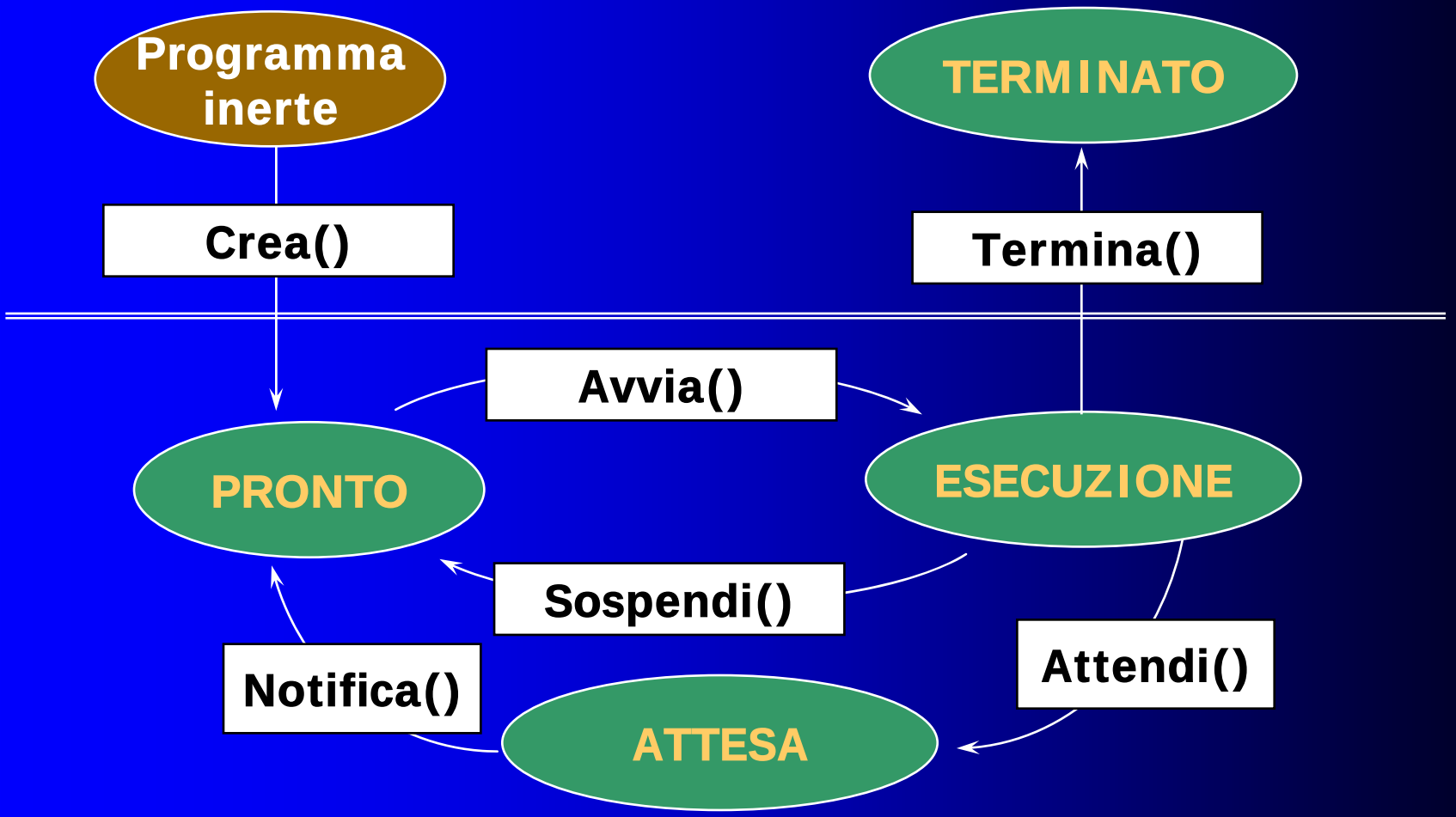
Stallo: riconoscimento – 1

- **A tempo d'esecuzione**
  - Assai oneroso
    - Occorre **bloccare** periodicamente l'avanzamento del sistema per analizzare lo stato di tutti i processi e verificare se quelli in attesa costituiscono una lista circolare chiusa
  - Lo sblocco di uno stallo comporta la terminazione forzata di uno dei processi in attesa
    - Il rilascio delle risorse liberate sblocca la catena di dipendenza circolare
- **Staticamente**
  - Può essere un problema non risolvibile!

Astrazione di processo

- Ogni processo è associato a un descrittore chiamato ***Process Control Block*** che ne specifica le caratteristiche distintive
  - Identificatore del processo
  - Contesto di esecuzione del processo
    - Tutte le informazioni necessarie a ripristinarne lo stato d'esecuzione dopo una sospensione o un prerilascio
  - Stato di avanzamento del processo
    - Puntatore (d)alla lista del processi in quello stato
  - Priorità
    - Iniziale, corrente
  - Diritti di accesso alle risorse e altri eventuali privilegi
  - Discendenza familiare
    - Puntatore al PCB del processo padre e degli eventuali processi figli
  - Puntatore alla lista delle risorse assegnate al processo
- Il PCB relaziona il processo alla sua **macchina virtuale**

Stati di avanzamento di processo – 1



Stati di avanzamento di processo – 2

- **Programma inerte**
  - Risiede in memoria secondaria
  - Un supervisore lo carica in memoria mediante una chiamata di sistema che crea il PCB corrispondente
- **PRONTO**
  - Una **macchina virtuale** è stata creata per il processo che ora è pronto per l'esecuzione in attesa del suo turno
- **ESECUZIONE**
  - Il processore è stato assegnato al processo selezionato la cui esecuzione può così avanzare

Stati di avanzamento di processo – 3

- **ATTESA**
  - Il processo è sospeso in attesa di una risorsa attualmente non disponibile o di un evento non ancora verificatosi
- **TERMINATO**
  - Il processo ha concluso regolarmente le sue operazioni e può rilasciare la sua **macchina virtuale**

Transizioni di stato – 1

- **Crea**()
  - Crea un PCB per il nuovo processo e gli assegna una macchina virtuale aggiornando la lista dei processi pronti (*ready list*)
- **Avvia**()
  - Manda in esecuzione il “primo” processo della lista dei pronti
    - La selezione dalla lista ” richiede un criterio non arbitrario
- **Sospendi**()
  - Il processo in esecuzione viene prerilasciato e il suo PCB ritorna nella lista dei pronti

Transizioni di stato – 2

- **Attendi**()
  - Il processo richiede l'uso di una risorsa o l'arrivo di un evento e viene sospeso se la risorsa è occupata o se l'evento non si è ancora verificato
- **Notifica**()
  - La risorsa richiesta dal processo bloccato è di nuovo libera o l'evento atteso si è verificato
    - Il PCB del processo ritorna nella lista dei pronti
- **Termina**()
  - Il processo in esecuzione termina il suo lavoro e rilascia la **macchina virtuale**

Ordinamento di processi

- Diversi metodi per decidere come alternare i processi in esecuzione
  - **Scambio cooperativo** (*cooperative* / *non pre-emptive switch*)
    - Il processo in esecuzione decide da solo quando cedere il controllo
      - Windows 3.1 ☹
  - **Scambio a prerilascio** (inconsapevole)
    - Il processo in esecuzione viene rimpiazzato
      - Da un processo appena arrivato con maggiore importanza (*priority-based pre-emptive*)
        - Sistemi a tempo reale
      - All'esaurimento del quanto di tempo (*time-sharing pre-emptive*)
        - Sistemi interattivi (Unix → Linux, Windows NT)
- Il prerilascio si realizza tramite un meccanismo **esterno** all'esecuzione dei processi
  - Un dispositivo (p.es., orologio) solleva una interruzione
  - Un gestore *software* la identifica e, se necessario, la notifica allo *scheduler*

Politiche e meccanismi – 1

- Lo *scheduler* è il componente del nucleo che decide l'ordinamento dei processi
  - È progettato **prima** dei processi che è chiamato a governare
- Bisogna perciò rendere il suo operato **parametrico** rispetto a specifici attributi assegnati ai processi
  - Per non doverlo cambiare al variare delle applicazioni
    - Basta configurare opportunamente gli attributi dei processi
  - Lo *scheduler* attua **meccanismi** forniti dal nucleo del S/O
- Il *dispatcher* è il componente del nucleo che attua le scelte di ordinamento dei processi
  - Opera su mandato dello *scheduler*
  - Deve essere molto efficiente perché opera a ogni scambio di contesto (*context switch*)
    - Salva il contesto del processo in uscita, installa quello del processo in entrata e gli affida il controllo della CPU

Politiche e meccanismi – 2

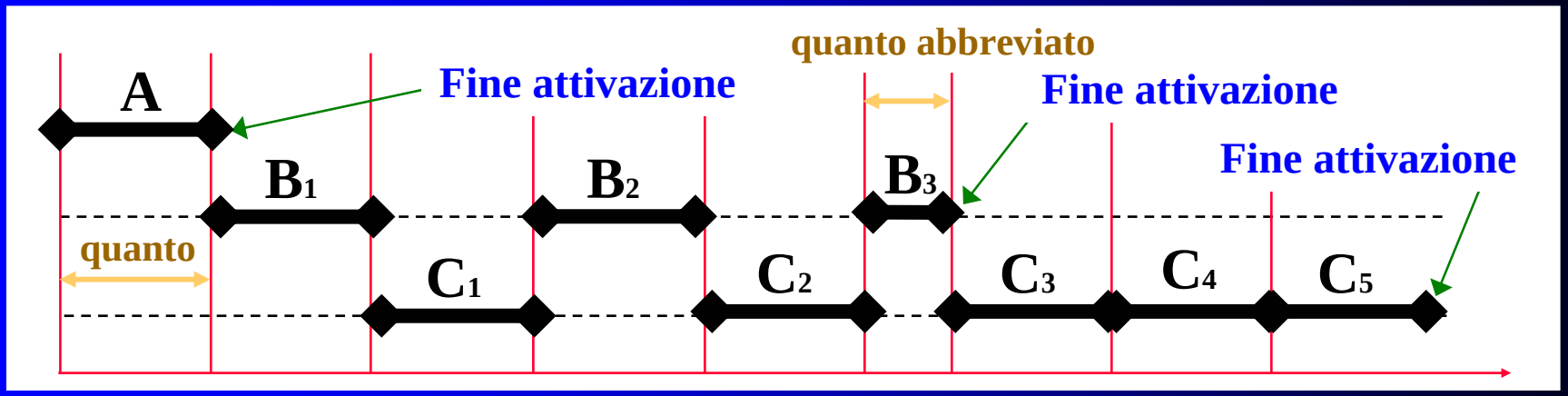
- L'applicazione decide le **politiche** di ordinamento fissando il valore degli attributi considerati dai meccanismi del nucleo
  - Per determinare l'ordinamento dei processi
  - Per influenzare l'attribuzione delle risorse
- L'efficienza delle politiche scelte si misura in termini di
  - **Percentuale di impiego utile della CPU**
    - Più i processi che il nucleo!
      - Il tempo di esecuzione di *scheduler* e *dispatcher* è sottratto ai processi
  - **Numero di processi avviati all'esecuzione per unità di tempo**
    - Misura di produttività (*throughput*)
  - **Durata di permanenza di un processo in stato di pronto**
    - Tempo di attesa
  - **Tempo di completamento** (*turn-around*)
  - **Reattività rispetto alla richiesta di avvio di un processo**
    - Tempo di risposta

Politiche e meccanismi – 3

- La garanzia di esecuzione dei processi dipende criticamente dalla **politica** di scambio adottata
  - Lo scambio cooperativo non offre alcuna garanzia
  - Gli utenti in genere richiedono equità di opportunità
    - *Fairness*
- I processi in stato di pronto sono registrati in una struttura detta **lista dei pronti** (*ready list*)
- La più semplice gestione della lista è con tecnica a coda (*First-Come-First-Served,* ***FCFS***)
  - Il primo processo a entrare in coda sarà anche il primo a essere avviato all'esecuzione
  - Molto facile da realizzare e da gestire

Politiche e meccanismi – 4

- Imponendo divisione di tempo (*time sharing*) sulla politica *FCFS* si ottiene una tecnica di rotazione detta ***round-robin***
- Vediamo l'applicazione di un quanto di tempo 2 su tre processi **A**, **B** e **C** con tempo di esecuzione 2, 5, 10 rispettivamente



Politiche e meccanismi – 5

- Le attività di un processo comprendono sequenze di azioni eseguibili dalla CPU intervallate da sequenze di azioni di I/O
- I processi si possono dunque classificare in
  - *CPU-bound*
    - Comprendenti attività lunga durata sulla CPU
  - *I/O- bound*
    - Comprendenti attività di breve durata sulla CPU intervallate da attività di I/O molto lunghe
- La politica FCFS penalizza i processi della classe *I/O-bound*
  - Cedendo la CPU durante le attività di I/O sono ritardati al ritorno dai processi che li hanno sostituiti

Politiche e meccanismi – 6

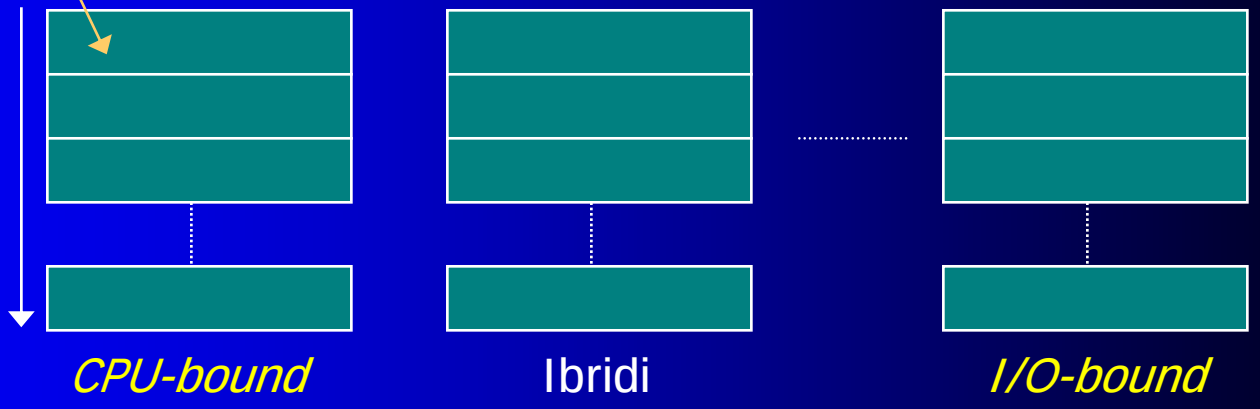
- **Esempio 1: politica di ordinamento a livelli**
  - A rotazione con priorità
- Scelta di politica – 1
  - Assegnare un dato livello di privilegio a ogni singolo processo
- Meccanismo impiegato
  - Attributo rappresentato da una priorità statica o dinamica registrata nel PCB
- Scelta di politica – 2
  - Processi distinti sulla base di determinate caratteristiche
    - Note a priori, p.es.: *CPU-bound*, *I/O-bound*
    - Acquisite a tempo d'esecuzione, p.es.: tempo cumulato (di esecuzione o di attesa)
- Meccanismo impiegato
  - Rilevazione del valore di un dato campo del PCB
- Scelta di politica – 3
  - Coda ordinata a priorità per ciascuna classe di processi
  - Selezione di coda *round-robin*

Politica a rotazione con priorità

Inserzione di processo
in coda di appartenenza
al proprio livello di priorità

Selezione di coda
a rotazione con divisione di tempo

Ordinamento in coda
per priorità decrescente



Politiche e meccanismi – 7

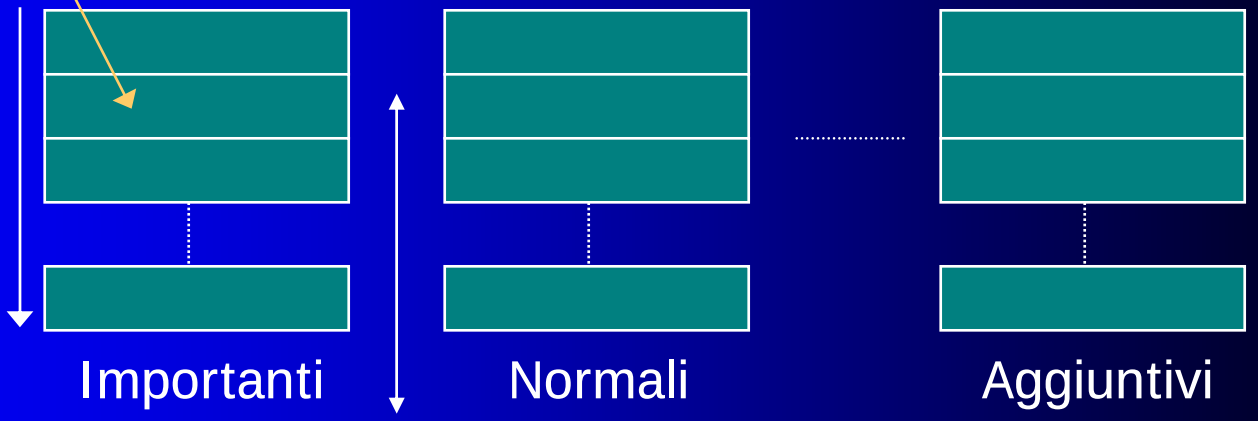
- **Esempio 2: politica di ordinamento a livelli**
  - A priorità con rotazione
- Scelta di politica – 1
  - Assegnare un dato livello di privilegio a ogni singolo processo
- Meccanismo impiegato
  - Attributo rappresentato da una priorità statica o dinamica registrata nel PCB
- Scelta di politica – 2
  - Processi distinti sulla base di determinate caratteristiche
    - Statiche o dinamiche
- Scelta di politica – 3
  - Coda FCFS per ciascuna classe di processi
  - Selezione di coda su base di priorità
  - Assegnazione di CPU con modalità *round-robin*

Politica a priorità con rotazione

Inserzione di processo
in coda corrispondente
alla propria classe di priorità

Selezione di coda
per priorità decrescente

Ordinamento in coda
con modalità FCFS



Selezione di processo
a rotazione per quanti di tempo

Politiche e meccanismi – 8

- I **meccanismi** per realizzare scelte di ordinamento e gestione dei processi risiedono nel nucleo
- Le **politiche** sono determinate fuori dal nucleo
  - Decise nello spazio delle applicazioni
  - Decidendo quali valori assegnare ai parametri di configurazione dei processi considerati dai meccanismi di gestione

Classificazione di sistemi – 1

- Diverse classi di sistemi concorrenti richiedono politiche di ordinamento di processi specifiche
- 3 classi generali
  - **Sistemi "a lotti"** (*batch*)
    - Ordinamento predeterminato; **lavori** di lunga durata e limitata urgenza; prerilascio non necessario
  - **Sistemi interattivi**
    - Grande varietà di attività; prerilascio essenziale
  - **Sistemi in tempo reale**
    - Lavori di durata ridotta ma con elevata urgenza; l'ordinamento deve riflettere l'importanza del processo; prerilascio possibile

Classificazione di sistemi – 2

- Caratteristiche desiderabili delle politiche di ordinamento
  - Per tutti i sistemi
    - **Equità** (*fairness*)
      - Nella distribuzione delle opportunità di esecuzione
    - **Coerenza** (*enforcement*)
      - Nell'applicazione della politica a tutti i processi
    - **Bilanciamento**
      - Nell'uso di tutte le risorse del sistema

Obiettivi specifici delle politiche

- Per i **sistemi a lotti**
  - Massimo prodotto per unità di tempo (*throughput*)
  - Massima rapidità di servizio per singolo lavoro (*turn-around*)
    - Media statistica
  - Massimo utilizzo delle risorse di elaborazione
- Per i **sistemi interattivi**
  - Rapidità di risposta per singolo lavoro
    - Rispetto alla percezione dell'utente
  - Soddisfazione delle aspettative generali dell'utente
- Per i **sistemi in tempo reale**
  - Rispetto delle scadenze temporali (*deadline*)
  - Predicibilità di comportamento (*predictability*)

Politiche di ordinamento – 1

- **Per sistemi a lotti**
  - **FCFS** (***First come first served***)
    - Senza prerilascio, senza priorità
    - Ordine di esecuzione = ordine di arrivo
    - Massima semplicità, basso utilizzo delle risorse
  - **SJB** (***Shortest job first***)
    - Senza prerilascio, richiede conoscenza dei tempi richiesti di esecuzione
    - Esegue prima il lavoro (*job*) più breve
    - Non è equo con i lavori non presenti all'inizio
  - **SRTN** (***Shortest remaining time next***)
    - Variante di SJB con prerilascio
    - Esegue prima il processo più veloce a completare
    - Tiene conto di nuovi processo quando essi arrivano
- In generale parliamo di lavori quando operiamo senza prerilascio e di processi quando operiamo con prerilascio

Politiche di ordinamento – 2.1

- **Per sistemi interattivi**
  - **OQ** : Ordinamento a quanti (***Round Robin***, **RR**)
    - Con prerilascio, senza priorità
    - Ogni processo esegue al più per un quanto alla volta
    - Lista circolare di processi
  - **OQP** : Ordinamento a quanti con priorità
    - Quanti diversi per livello di priorità
      - Come attribuire priorità a processi e come farle eventualmente variare
  - **GP** : Con garanzia per processo
    - Con prerilascio e con promessa di una data quantità di tempo di esecuzione (p.es. **1/n** per **n** processi concorrenti)
      - Le necessità di ciascun processo devono essere note (stimate) a priori
    - Esegue prima il lavoro maggiormente penalizzato rispetto alla promessa
      - Verifica periodica o a evento (soddisfacimento della promessa)

Politiche di ordinamento – 2.2

- **Per sistemi interattivi**
  - **SG**: Senza garanzia
    - Con prerilascio e priorità, opera sul principio della lotteria
      - Ogni processo riceve numeri da giocare
      - A priorità più alta corrispondono più numeri da giocare
      - A ogni scelta per assegnazione di risorsa, essa va al processo possessore del numero estratto
      - Le estrazioni avvengono periodicamente (= quanti) e/o a eventi (p.es. attesa di risorse non disponibili)
    - Comportamento impredicibile sul breve periodo, ma tende a stabilizzarsi statisticamente nel tempo
  - **GU** : Con garanzia per utente
    - Come GP ma con garanzia riferita a ciascun utente (possessore di più processi)

Politiche di ordinamento – 3.1

- **Per sistemi in tempo reale**
  - I sistemi in tempo reale sono sistemi concorrenti nei quali il valore corretto **deve** essere prodotto entro un tempo fissato
    - Oltre tale limite il valore prodotto ha utilità decrescente, nulla o addirittura negativa
  - L'ordinamento (*scheduling*) di processi deve fornire garanzie di completamento adeguate ai processi
    - Deve essere analizzabile staticamente (predicibile)
  - Il caso peggiore è sempre quando tutti i processi sono pronti insieme per eseguire all'istante iniziale (*critical instant*)

Politiche di ordinamento – 3.2

- **Per sistemi in tempo reale**
  - Modello semplice (***cyclic executive***)
    - L'applicazione consiste di un insieme fissato di processi periodici (ripetitivi) ed indipendenti con caratteristiche note
    - Ciascun processo è suddiviso in una sequenza ordinata di procedure di durata massima nota
    - L'ordinamento è costruito a tavolino come una sequenza di chiamate a procedure di processi fino al loro completamento
    - Un ciclo detto maggiore (*major cycle*) racchiude l'invocazione di tutte le sequenze di tutti i processi
    - Il ciclo maggiore è suddiviso in N cicli minori (*minor cycle*) di durata fissa che racchiude l'invocazione di specifiche sottosequenze

Esempio 1
Modello semplice senza suddivisione

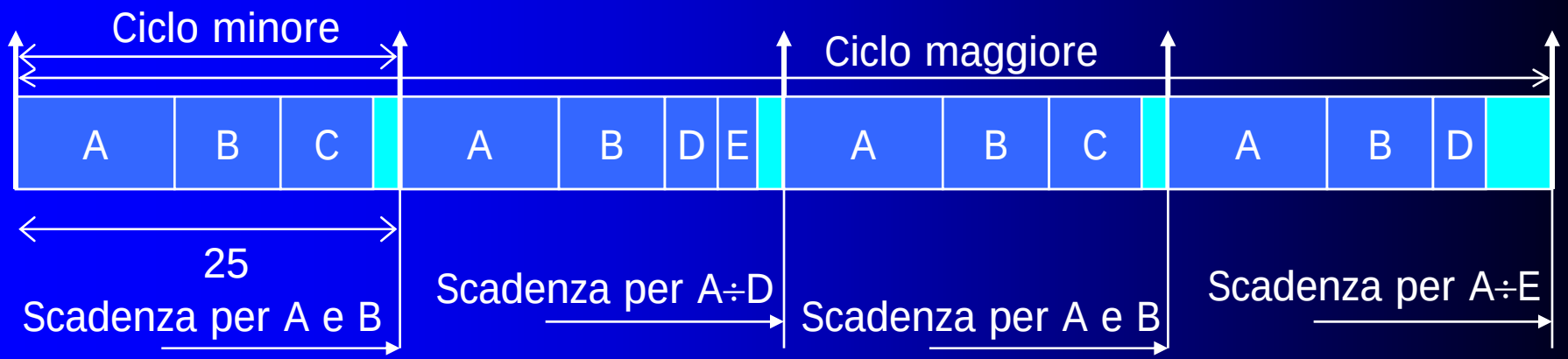
Processo	Periodo T	Durata C
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2

Conviene che i periodi siano armonici!

$$U = \Sigma_i (C_i / T_i) = 46/50 = 0.92$$

Ciclo maggiore di durata 100 → MCM di tutti i periodi

Ciclo minore di durata 25 → periodo più breve



Politiche di ordinamento – 3.3

- **Per sistemi in tempo reale**
  - **Ordinamento a priorità fissa**
    - Preferibilmente *con* prerilascio (a priorità!)
      - Processi periodici, indipendenti e noti
    - Assegnazione di priorità secondo il periodo (*rate monotonic*)
      - Per scadenza uguale a periodo (D = T), priorità maggiore per periodo più breve
      - Test di ammissibilità sufficiente ma non necessario per *n* processi indipendenti (Liu & Layland, 1973)

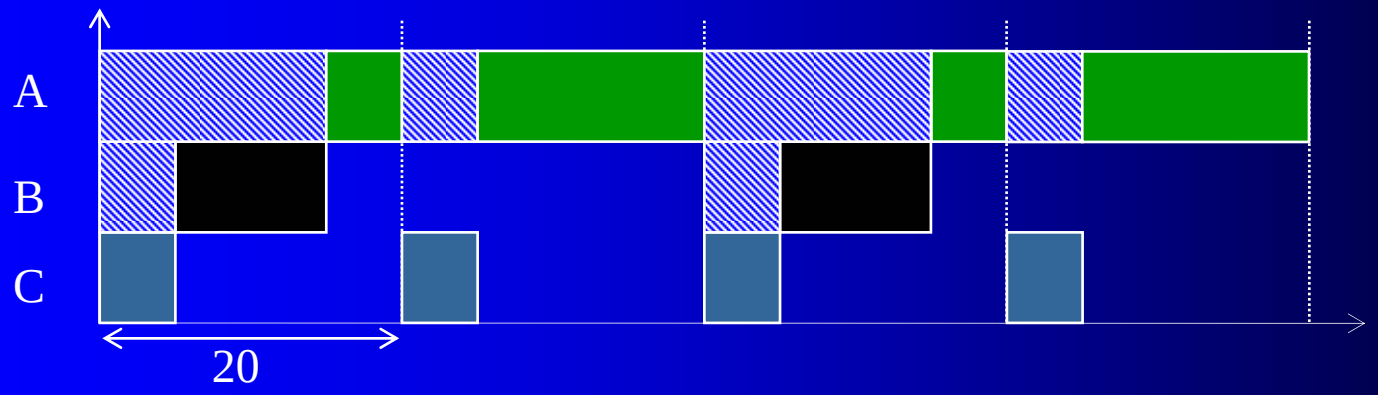
$$U = \sum_{i:=1}^{n} \left(\frac{C_i}{T_i} \right) \leq f(n) = n(2^{1/n} - 1)$$

Esempio 2

Caso semplice ordinamento a priorità

Processo	Periodo T	Durata C	Priorità
A	80	40	1 ← Bassa
B	40	10	2
C	20	5	3 ← Alta

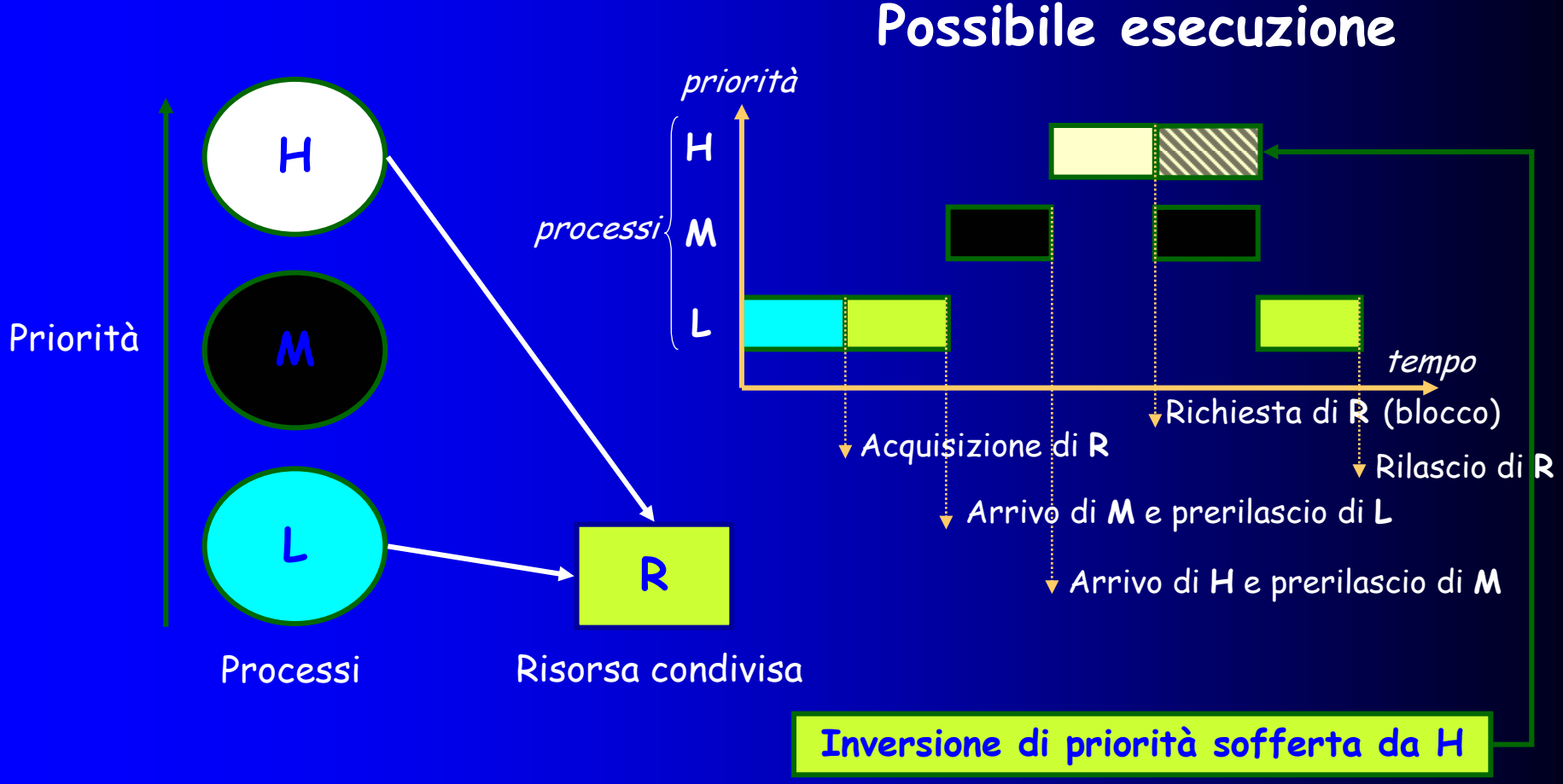
Il test di ammissibilità fallisce $U = 1 > f(3) = 0{,}78$ ma il sistema è ammissibile!



Politiche di ordinamento – 3.4

- **Per sistemi in tempo reale**
  - **Ordinamento a priorità fissa con prerilascio** e scadenza inferiore a periodo (D < T)
    - Assegnazione di priorità secondo la scadenza
    - Modello di processi generalizzato
      - Condivisione di risorse e processi sporadici
    - Rischio di **inversione di priorità**
      - Processi a priorità maggiore *bloccati* dall'esecuzione di processi a priorità minore
      - Effetto causato dall'accesso esclusivo a risorse condivise
      - Può condurre a blocco circolare (*deadlock*)

Inversione di priorità



Un caso reale: Mars Pathfinder

- **Inversione di priorità**
  - Priorità media attiva mentre quella alta è bloccata dalla bassa
  - Risultato: frequenti reset di sistema

- **Sistema Operativo VxWorks**
  - Ordinamento con prerilascio
  - Bus informazioni condiviso
    - **Mutex**
  - Gestione Bus
    - Alta priorità
  - Raccolta dati meteo
    - Bassa priorità
  - Trasmissione
    - Media priorità

Inversione di priorità, a parole

- **Esempio IP1**
  - Consideriamo tre processi **L**, **M**, **H** in ordine di priorità crescente
  - Assumiamo che condividano la risorsa **R** (*Mutex*)
  - **Inversione di priorità**
    - **L** si aggiudica **R**
    - **H** diviene attivo e vuole **R**
    - **H** deve attendere che **L** rilasci **R**
      - Il tempo d'uso di R ha durata prevedibile
    - **M** diviene attivo e blocca **L** (diverse priorità)
    - **H** deve attendere che **M** finisca ...
      - ... oltre che L esca dalla sezione critica ...
    - Cosa accade se nel frattempo si attivano altri processi a priorità intermedia tra **M** e **H**?

Politiche di ordinamento – 3.5

- **Soluzione: Innalzamento delle priorità**
  - Versione base (***Basic Priority Inheritance***)
  - La ***BPI*** non impedisce il *deadlock*
    1. L'innalzamento avviene solo quando un processo a priorità maggiore si blocca all'ingresso di una risorsa attualmente in possesso di un processo a priorità inferiore
    2. Il processo che possiede la risorsa (e che ha avuto l'innalzamento di priorità) può così terminare senza altre interruzioni
       - L'arrivo di un altro processo di priorità ancora superiore causa prerilascio e riporta la situazione al punto 1
  - **BPI** richiede il controllo di accesso e quindi causa catene di blocchi a tempo d'esecuzione
- Studiare l'uso di **BPI** sull'**esempio IP1**

Politiche di ordinamento – 3.6

- **Soluzione: Innalzamento della priorità**
  - Versione avanzata (***Immediate ceiling priority***)
    - Ogni processo j ha una **priorità statica di base** PB_j
    - Ogni risorsa condivisa i ha una priorità (*ceiling*) PC_i pari alla massima priorità dei processi che stanno attualmente richiedendo di usarla
    - Ogni processo j ha una **priorità dinamica** $P_j = \max\{PB_j, PC_i\}$ $\forall$ risorsa condivisa i in suo possesso
    - Un processo può acquisire una risorsa solo se la sua priorità dinamica corrente è maggiore del *ceiling* di tutte le risorse attualmente in possesso di altri processi
      - Un processo a priorità maggiore può essere bloccato *una sola volta* durante l'intera sua esecuzione (solo per la durata della sezione critica del processo a priorità più bassa)

Politiche di ordinamento – 3.7

- La tecnica **IPC** evita il *deadlock*
- **Esempio IP2**
  - Consideriamo tre processi **L**, **M**, **H** con priorità crescente
  - Assumiamo che tutti condividano le risorse **R1** e **R2** (entrambe *Mutex*)
    - Il *priority ceiling* di **R1** e **R2** è superiore alla priorità di **H**
  - **L** acquisisce **R1** e ne assume il *ceiling* poi si accinge a richiedere **R2**
  - **H** diventa pronto a questo istante e vorrebbe prerilasciare **L**
    - Ma non può perché la priorità di **H** non è superiore al *ceiling* di **R1** e **R2**
  - Quindi **H** resta pronto ma non riesce a prerilasciare **L**
  - **L** acquisisce anche **R2** e poi prosegue fino a rilasciare **R1** e **R2**
    - La priorità di **L** ritorna al valore originale
  - **H** ha ora priorità maggiore di **L** di ogni altro eventuale **M**
    - **H** può acquisire **R2** proseguire e completare
- La tecnica **IPC** impedisce il formarsi di catene di blocchi ...
  - I processi $\{H\}_i$ subiscono al più 1 blocco da parte di 1 processo L in possesso di risorsa R condivisa con {H}
    - Blocco = ritardo nel primo prerilascio

Politiche di ordinamento – 3.8

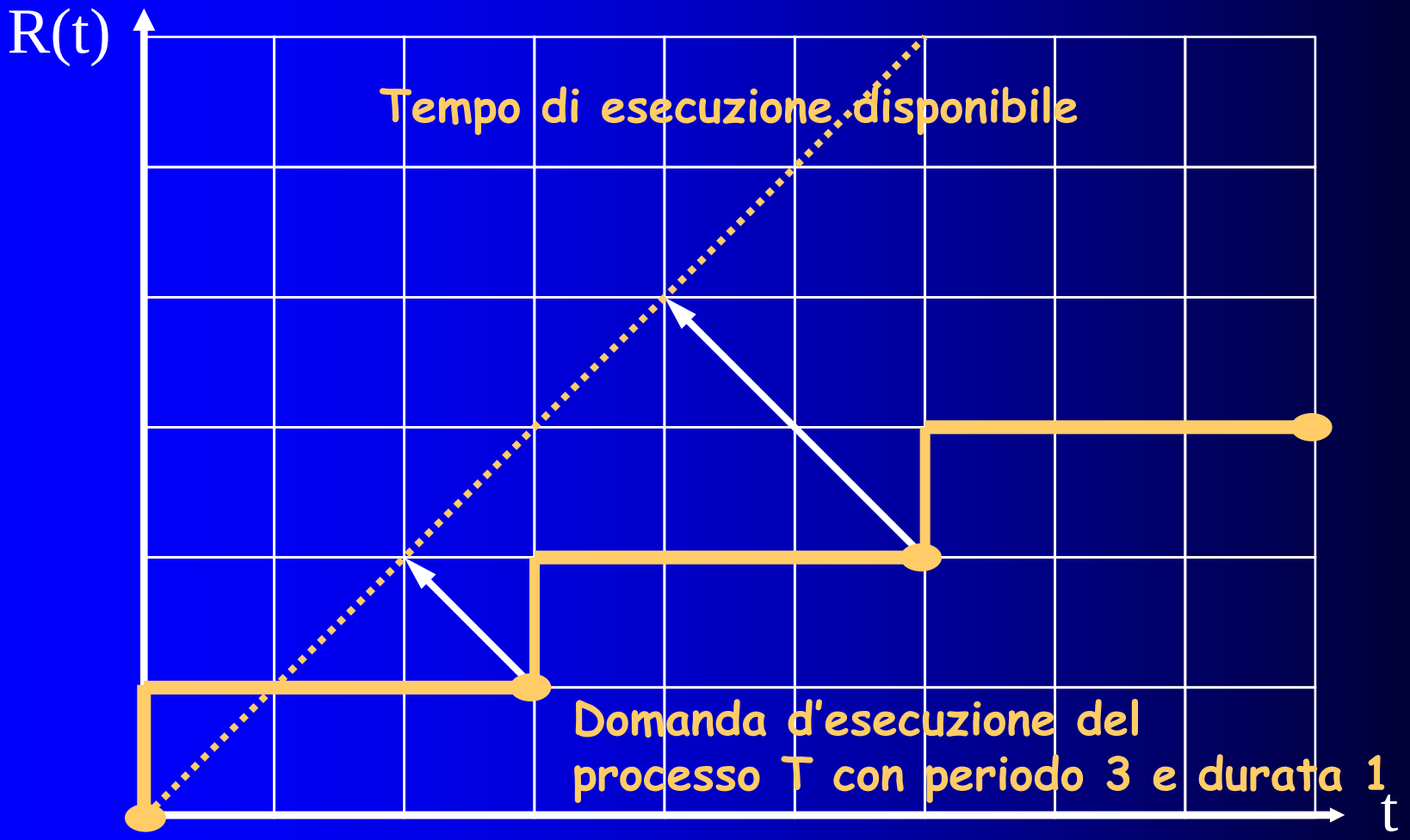
- **Per sistemi in tempo reale**
  - Calcolo del **tempo di risposta** R_i del processo i
    - **Tempo di blocco** del processo i
      - $B_i = \max_k\{C_k\}$ $\forall$ risorsa k usata da processi a priorità più bassa di i
    - **Interferenza** subita dal processo i da parte di tutti i processi j a priorità maggiore
      - $I_i = \Sigma_j \lceil R_i/T_j \rceil C_j$
    - $R_i = C_i + B_i + I_i$

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

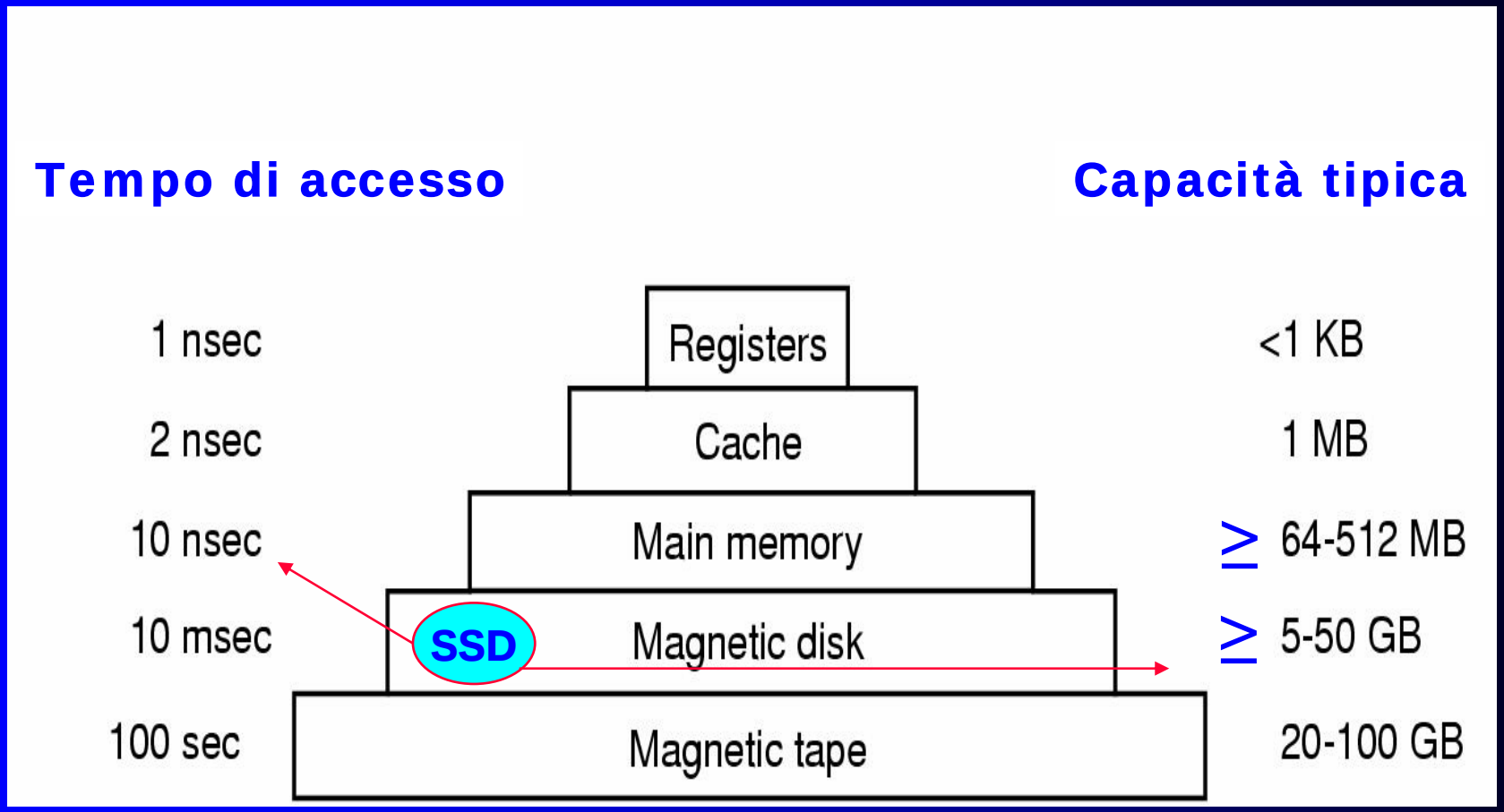
$$\omega_i^{k+1} := C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{\omega_i^k}{T_j} \right\rceil C_j$$

$$\omega_i^0 = C_i$$

Tempo di risposta **R**



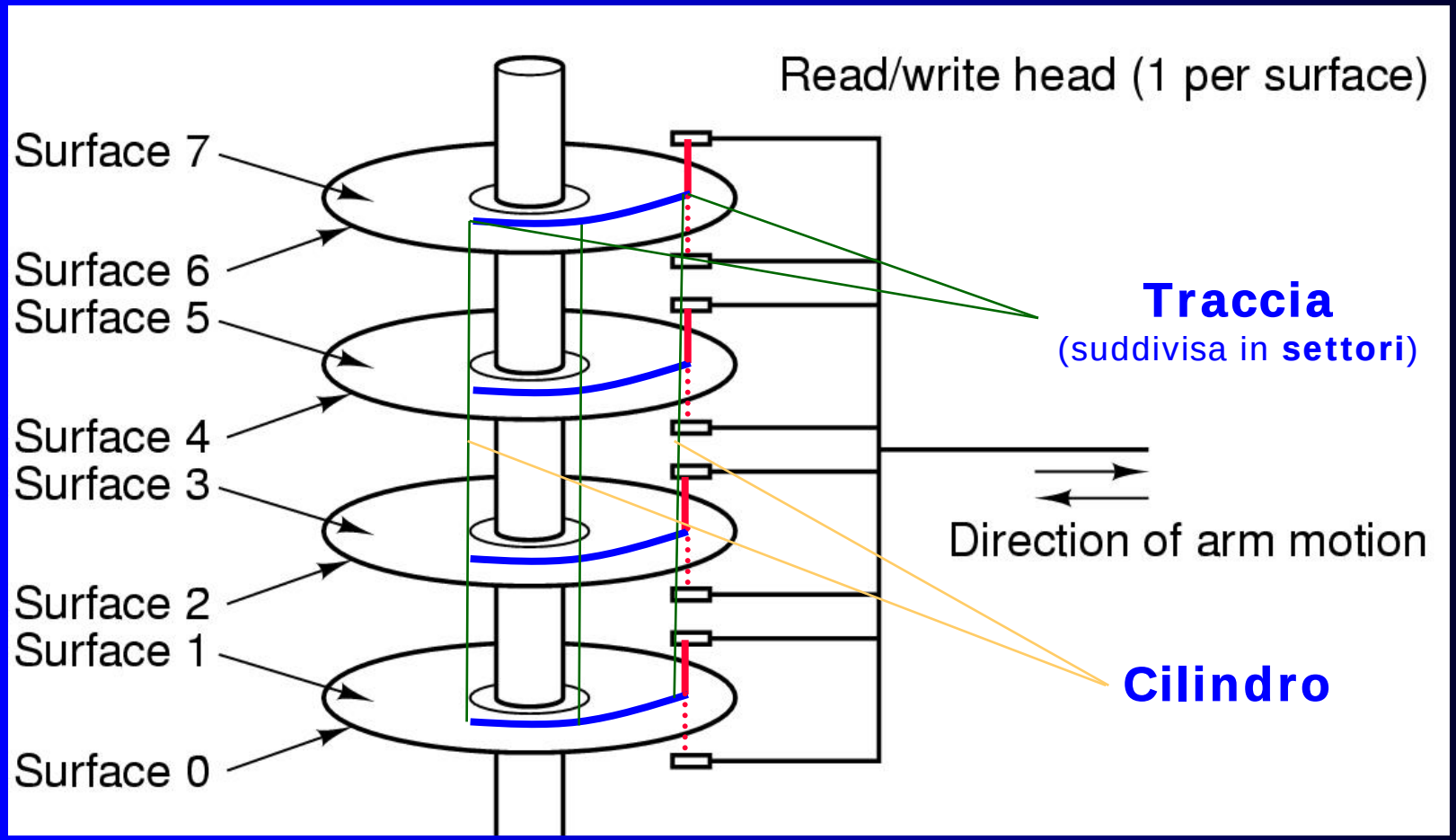
Gerarchia fisica di memoria – 1



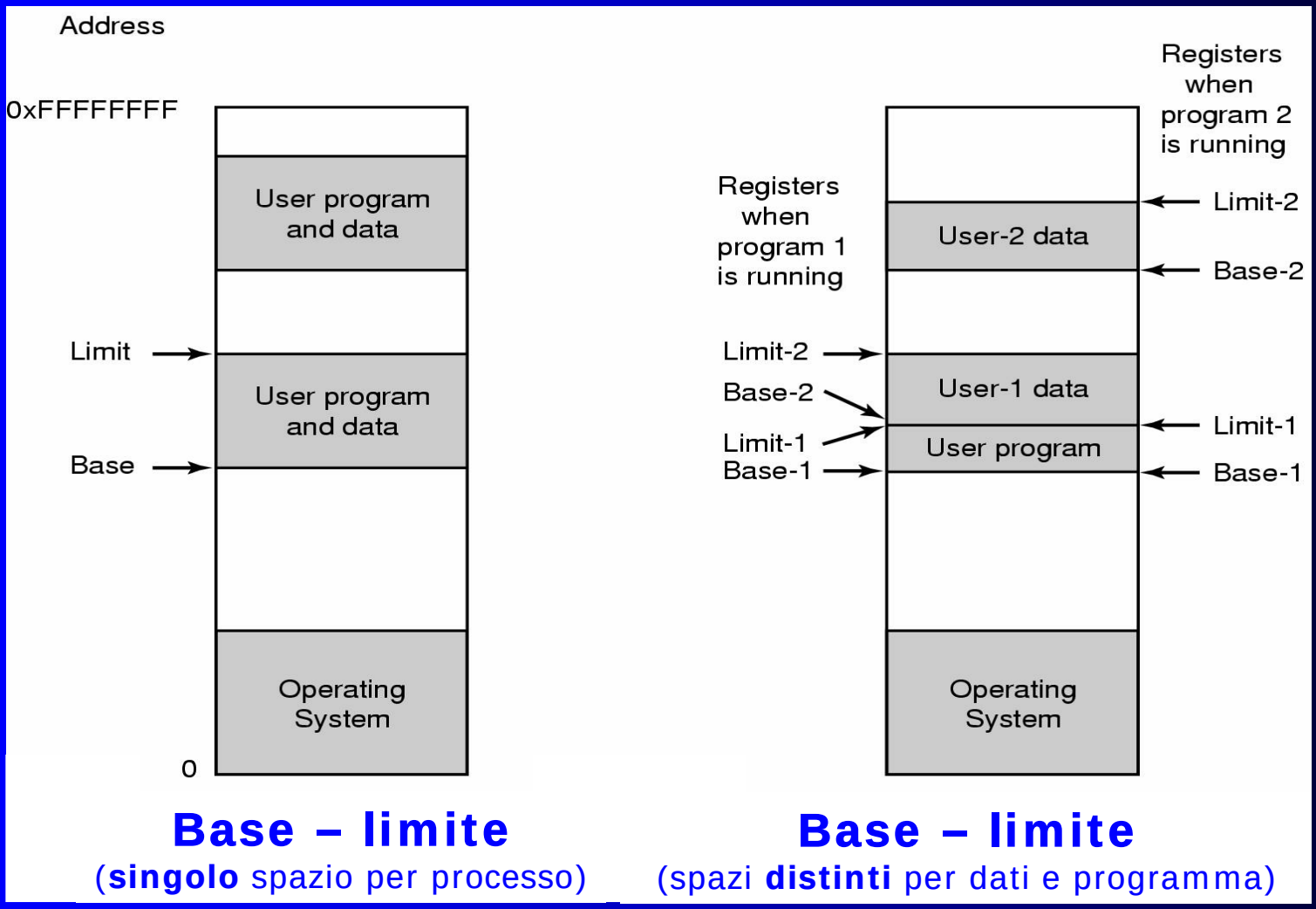
Gerarchia fisica di memoria – 2

- La ***cache*** è suddivisa in blocchi chiamati *line* con ampiezza tipica 64 B
  - Costi e benefici: *miss*, *hit*
  - Coerenza della memoria: *write-through*, *copy-back*
- I **dischi magnetici** hanno capienza 100 volte superiore e costo/*bit* 100 volte inferiore rispetto alla RAM
  - Tempo di accesso 1.000 volte peggiore
- La tecnologia SSD (***solid state drive***) offre una alternativa interessante
  - Capienza paragonabile ai dischi
  - Tempo d'accesso nell'ordine della RAM
    - Grazie all'assenza di parti meccaniche
  - Costo/*bit* >30 volte superiore ai dischi

Gerarchia fisica di memoria – 3



Vista logica della RAM – 1



Base – limite
(**singolo** spazio per processo)

Base – limite
(spazi **distinti** per dati e programma)

Vista logica della RAM – 2

- Nella sua forma più rudimentale la ripartizione della RAM tra processi distinti utilizza 2 registri speciali
  - **Base** e **limite**, i cui valori formano parte importante del contesto del processo
  - L'allocazione di un processo in RAM richiede **rilocazione** della sua **memoria virtuale**
    - Ampia 2^N *Byte* dove N è il numero di *bit* usati per un indirizzo
- In generale la gestione dello spazio di memoria virtuale dei processi utilizza un dispositivo **MMU** (*Memory Management Unit*) logicamente interposto tra CPU e memoria
  - Sotto la responsabilità del S/O

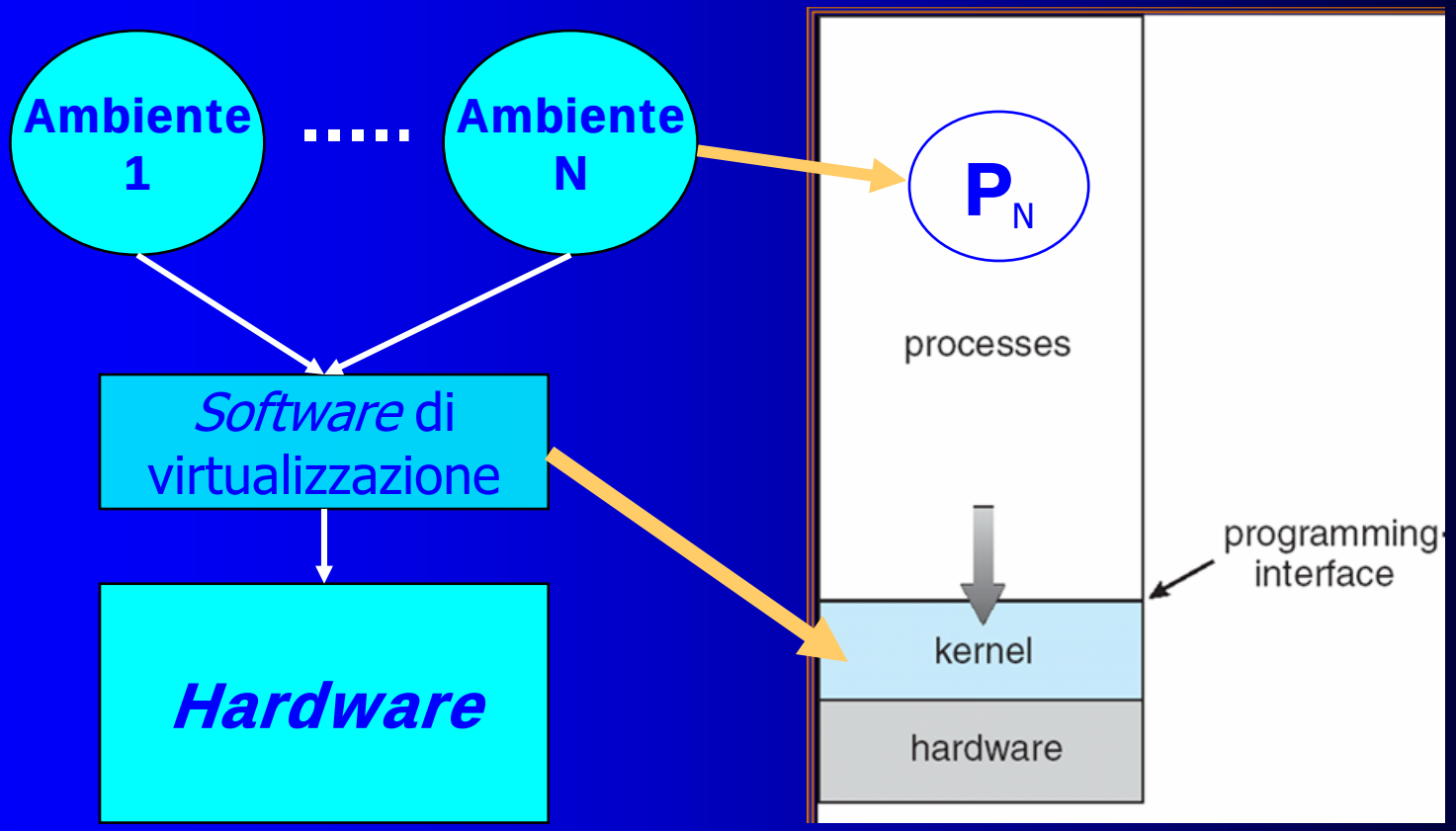
Vista logica della RAM – 3

- Il S/O assegna una porzione di RAM a ogni processo
- Un processo emette indirizzi logici **I** all'interno della propria memoria virtuale
  - Indirizzi contigui e relativi a una base logica "zero"
- L'indirizzo logico **I** va tradotto in un indirizzo fisico **M**
  - Traduzione (**rilocazione**) effettuata dalla MMU come **M** = **Base** + **I**
  - Se **M** < **Limite** allora l'indirizzo si trova davvero nella zona di RAM assegnata al processo
    - Altrimenti va cercato in memoria secondaria
    - La MMU deve allora effettuare un altro calcolo su **M** per capire la zona di disco da dove prelevarlo
    - Il caricamento di dati in RAM può comportare il rimpiazzo di altri dati
      - La RAM ha capacità limitata mentre i processi sono tanti e "esosi"

Macchina virtuale – 1

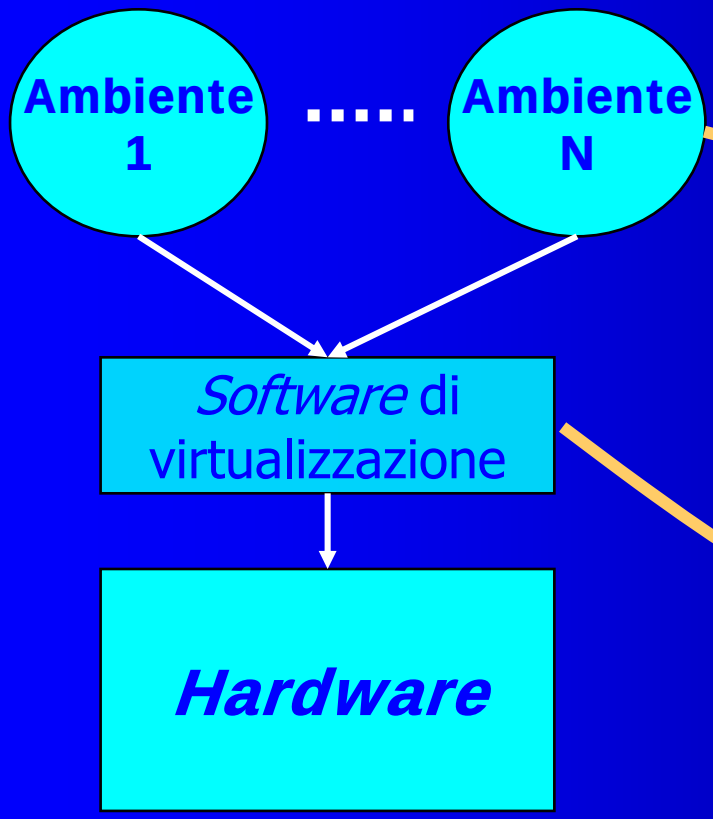
- Alla base del concetto stesso di multiprogrammazione
  - Condividere trasparentemente tra molti utenti una risorsa progettata per uso singolo
    - L'intero elaboratore o una parte di esso
    - Compito originario del sistema operativo
- La virtualizzazione porta grandi benefici
  - Fino al punto di consentire l'esecuzione simultanea di diversi sistemi operativi con le loro applicazioni
    - Richiede uno strato *software* dedicato tra l'*hardware* e i sistemi operativi ospitati oppure sopra il sistema operativo scelto come ospite

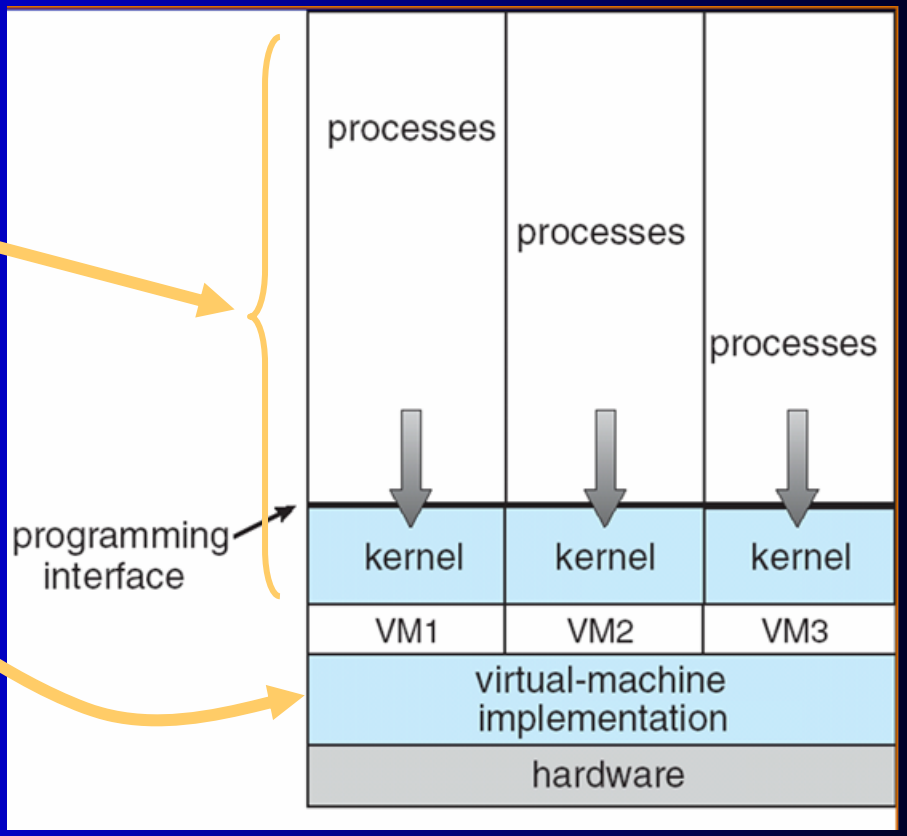
Macchina virtuale – 2



Silberschatz, Galvin and Gagne ©2005

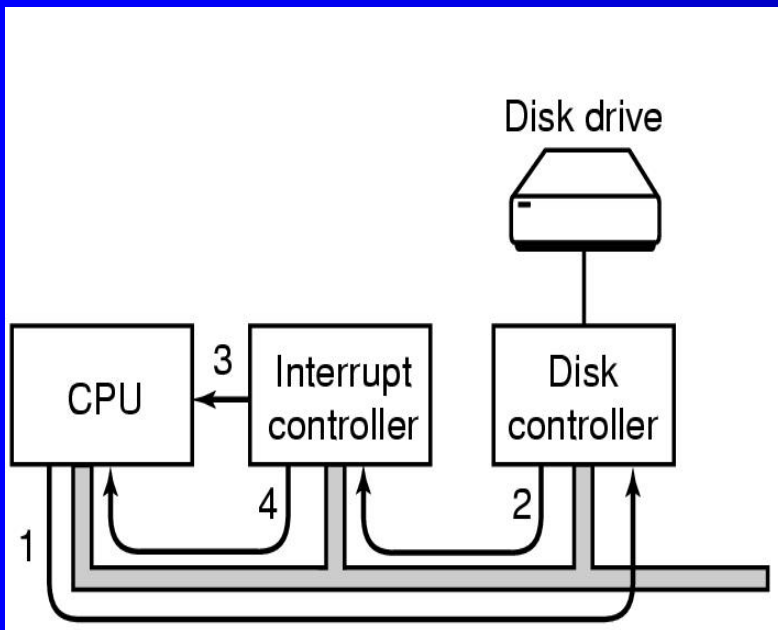
Macchina virtuale – 3



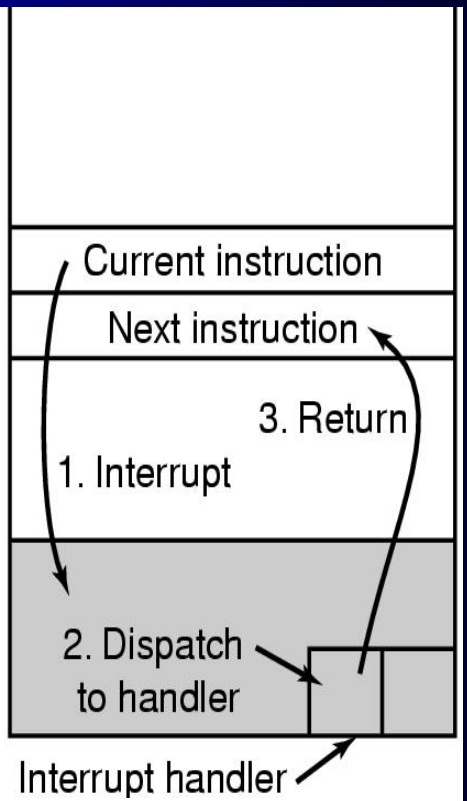


Silberschatz, Galvin and Gagne ©2005

Gestione dell'I/O – 1



Gestione di un dispositivo di I/O



Gestione delle interruzioni

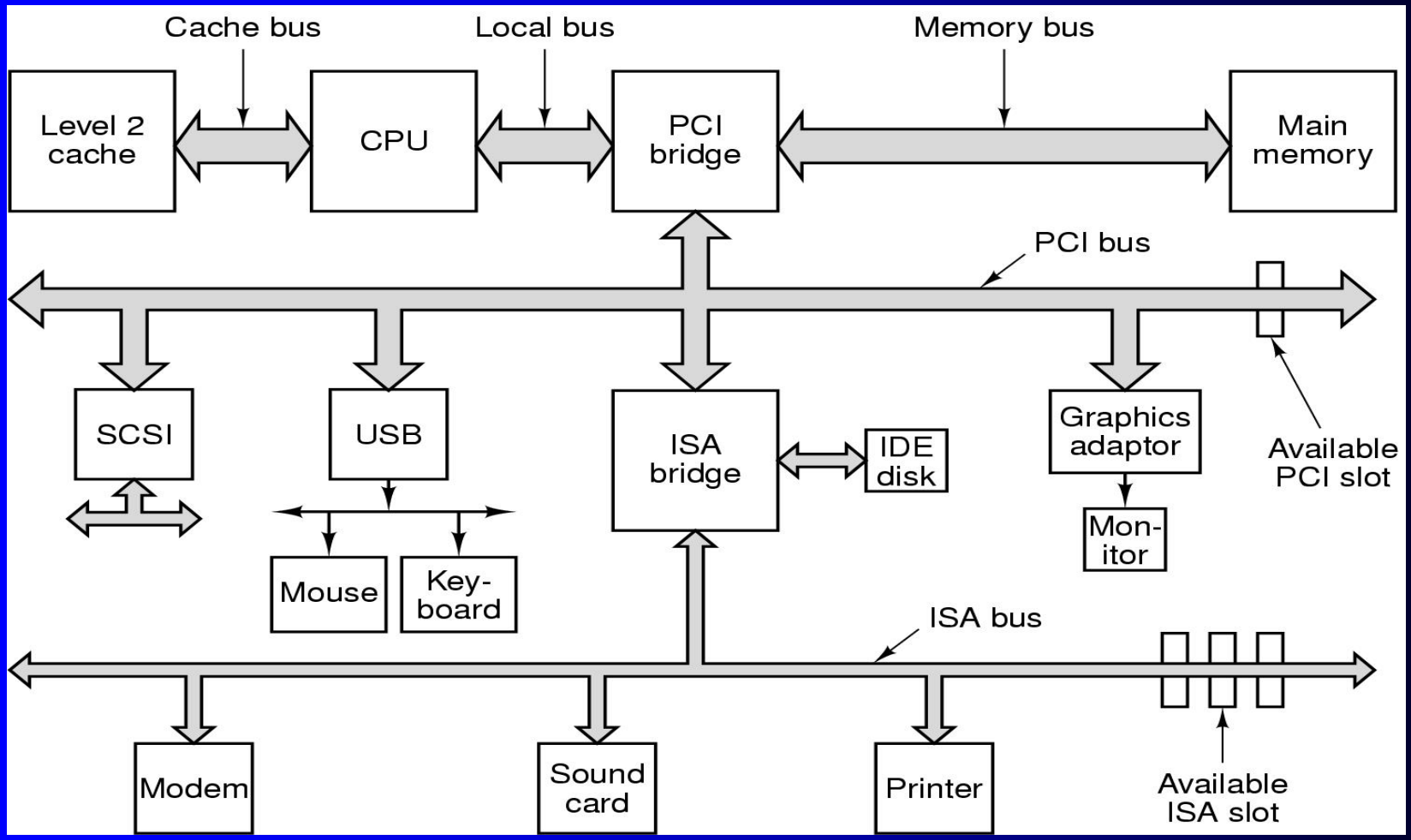
Gestione dell'I/O – 2

- L'uso delle interruzioni per l'interazione con i dispositivi di I/O risparmia il ricorso al *polling*
- L'interazione tipica avviene in 4 passi successivi
  1. Il gestore del dispositivo programma il controllore di dispositivo scrivendo nei suoi registri di interfaccia
  2. Il controllore agisce sul dispositivo e poi informa il controllore delle interruzioni
  3. Il controllore delle interruzioni asserisce un valore (*pin*) di notifica verso la CPU
  4. Quando la CPU si dispone a riceve la notifica il controllore delle interruzioni comunica anche l'identità del dispositivo
     - Così che il trattamento dell'interruzione sia attribuito al gestore appropriato

Gestione dell'I/O – 3

- All'arrivo di una interruzione
  - I registri PC e PSW sono posti sullo *stack* del processo corrente
  - La CPU passa al "modo operativo protetto"
  - Il parametro principale che denota l'interruzione serve come indice nel **vettore delle interruzioni**
    - Così si individua il gestore designato a servire l'interruzione
  - La parte **immediata** del gestore esegue nel contesto del processo interrotto
    - La parte del servizio meno urgente può essere invece **differita** e demandata a un processo dedicato

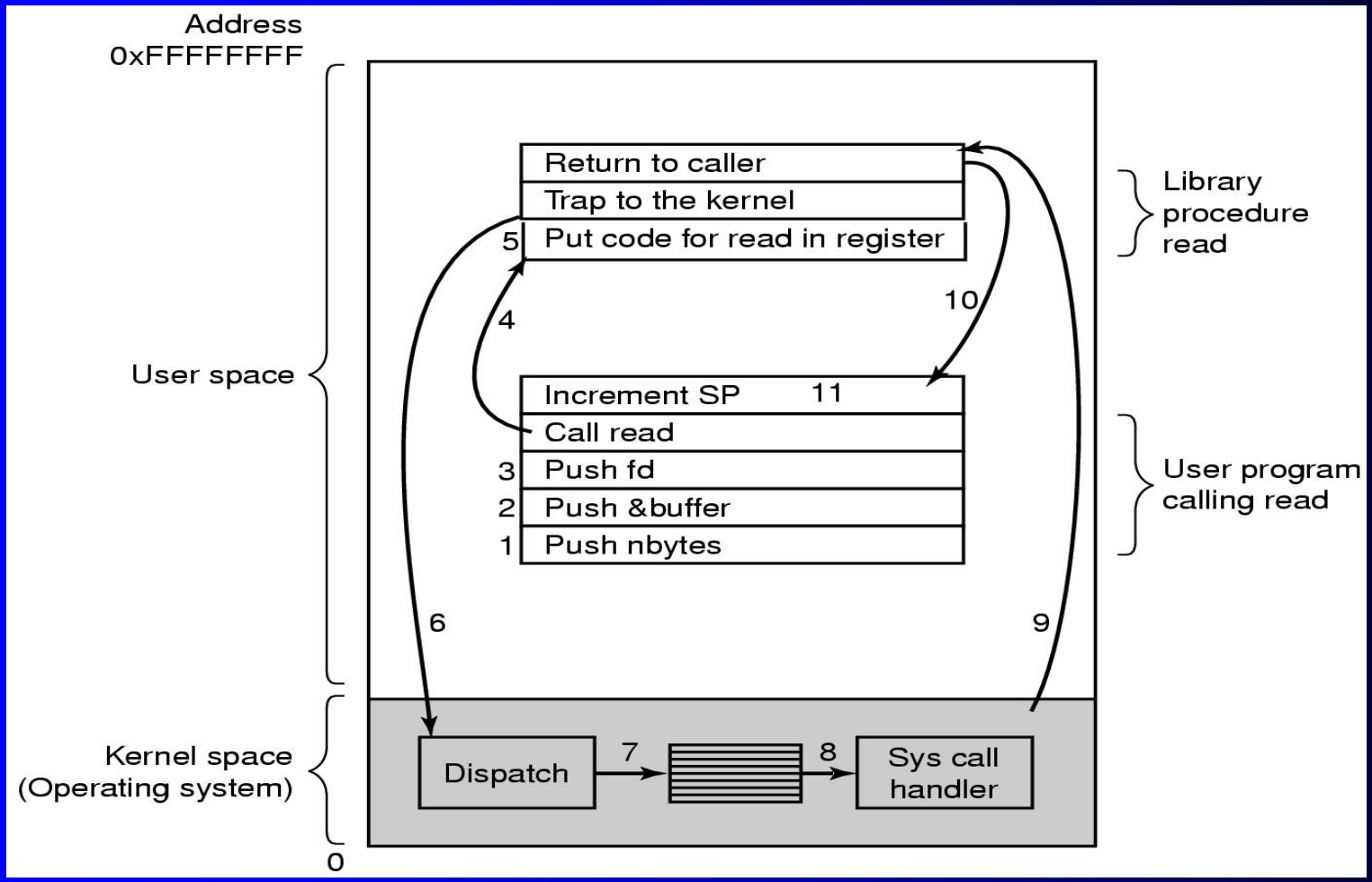
Pentium: architettura fisica – 1



Pentium: architettura fisica – 2

- **PCI** *bus* (*Peripheral Component Interconnect* 66 MHz, 8 B/ciclo)
  - Di vecchia concezione ma dotato di connettori per una grande varietà di dispositivi
- **USB** *bus* (*Universal Serial Bus*, ≤ 1.5 MB/s)
  - Per l'interconnessione di dispositivi lenti
  - 4 linee delle quali 2 di alimentazione del dispositivo
  - *Bus* con singolo *master* centrale predefinito che interroga @ 1 ms i dispositivi collegati (*polling*!)
- **SCSI** *bus* (*Small Computer System Interface* ≤ 160 MB/s)
  - Per l'interconnessione di dispositivi veloci

Chiamate di sistema – 1



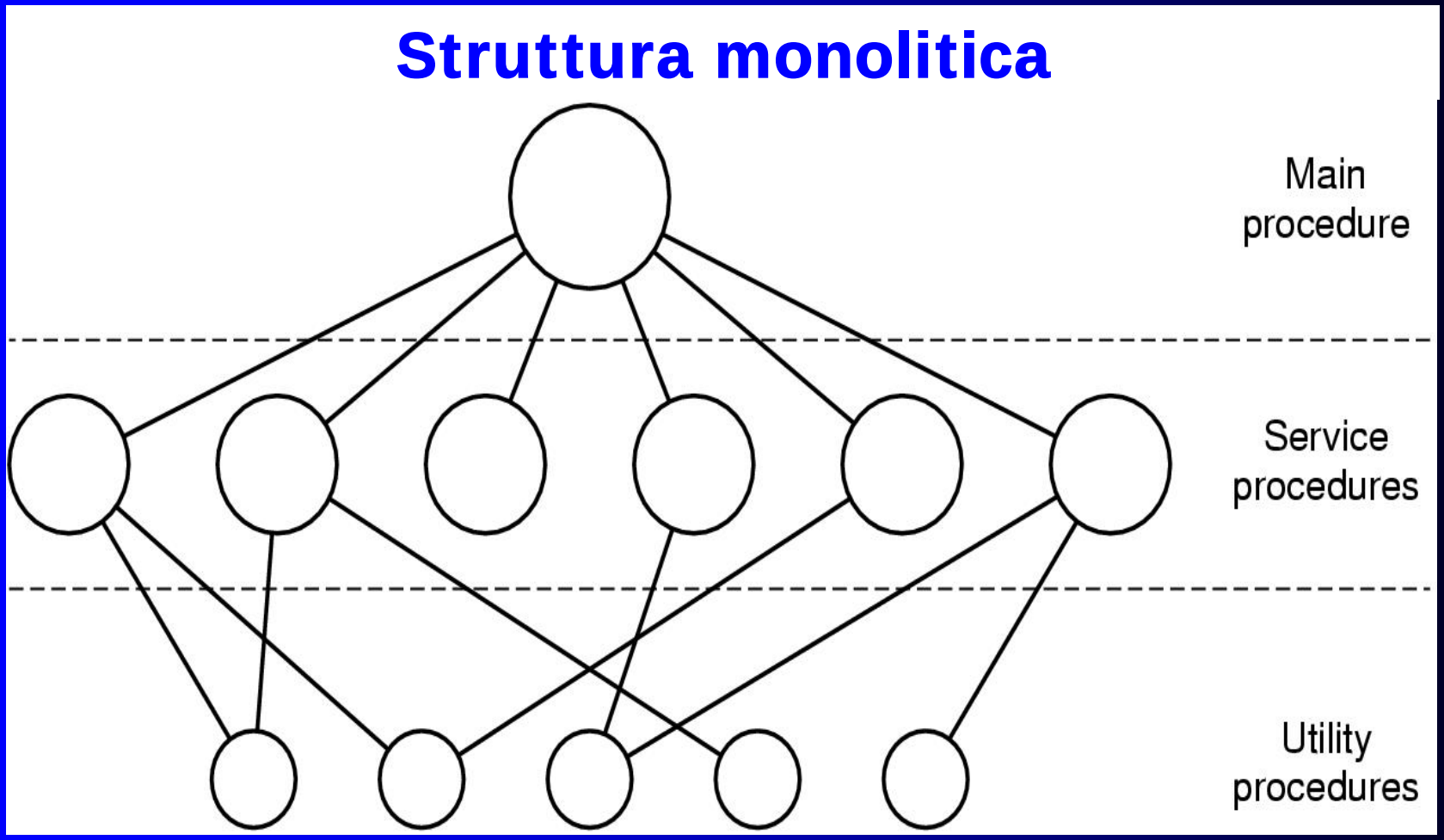
Chiamate di sistema – 2

- La maggior parte dei servizi del S/O sono eseguiti in risposta a invocazioni **esplicite** di processi
  - **Chiamata di sistema**
- Le chiamate di sistema sono nascoste in **procedure di libreria** predefinite note ai compilatori
  - L'applicazione **non** effettua direttamente chiamate di sistema
  - La procedura di libreria svolge il lavoro di preparazione necessario ad assicurare la corretta invocazione della chiamata di sistema
- La prima istruzione di una chiamata di sistema (***`trap`***) deve attivare il **modo** operativo **privilegiato**
  - Il parametro della chiamata designa l'azione da svolgere e la convenzione per trovare gli altri eventuali parametri
  - Il meccanismo complessivo è simile a quello già visto per il trattamento delle interruzioni
    - Le interruzioni sono **asincrone**
    - Le chiamate di sistema sono **sincrone**

Chiamate di sistema – 3

1. L'applicazione effettua una chiamata di sistema
   2. Pone sullo *stack* i parametri …
   3. … secondo una antica convenzione C/UNIX
4. Invoca la procedura di libreria (PL) che corrisponde alla chiamata
5. La PL notifica la chiamata al S/O
   - Scrivendo l'ID della chiamata in un luogo noto al S/O
6. La PL esegue l'istruzione ***`trap`***
   - L'esecuzione prosegue in modo operativo privilegiato
7. Il S/O individua la chiamata da eseguire
8. La esegue
9. Ritorna al chiamante oppure a un nuovo processo
   - Secondo quanto deciso dallo *scheduler*

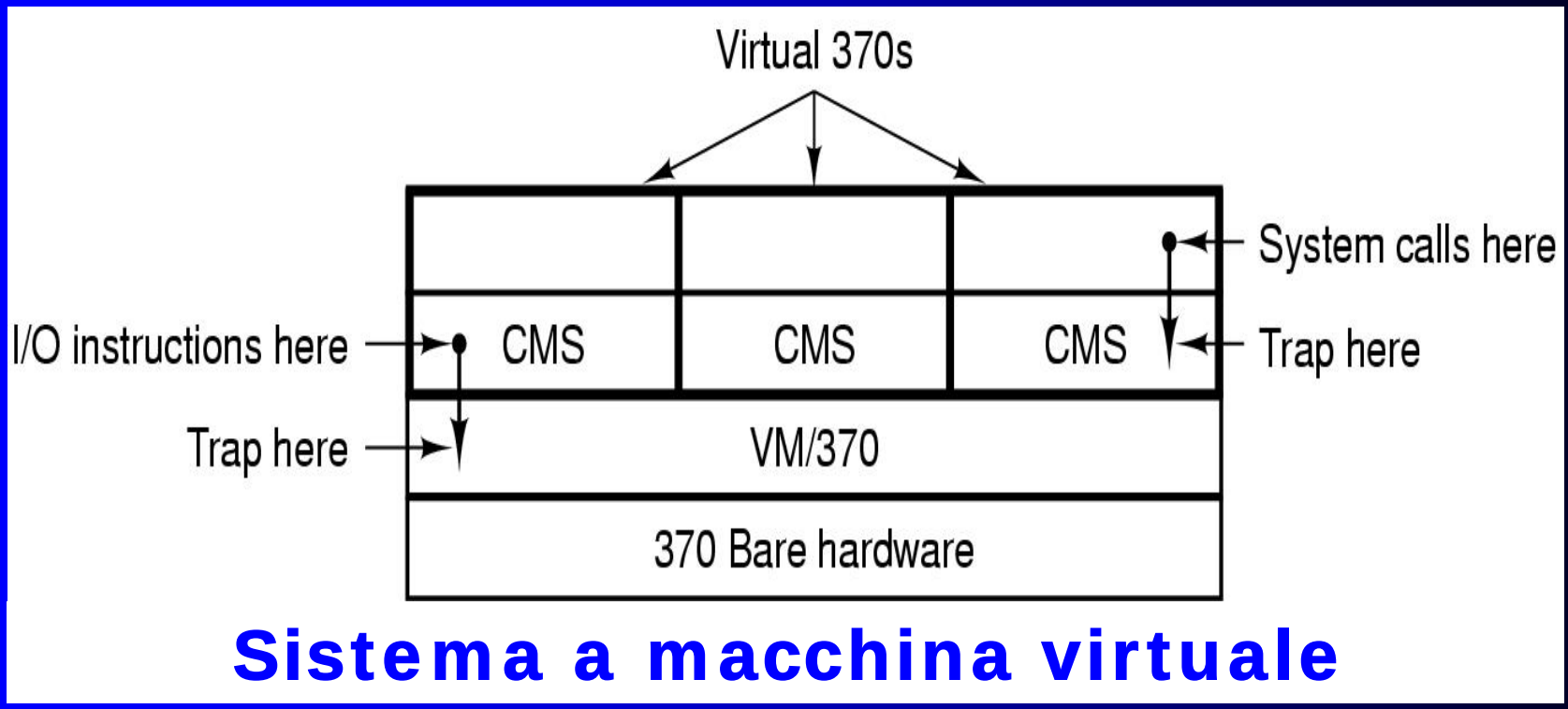
Architettura logica di S/O – 1



Architettura logica di S/O – 2

- Un'architettura monolitica **non** ha struttura
  - Il S/O è una collezione "piatta" di procedure
    - Ognuna delle quali può chiamarne qualunque altra
    - Nessuna forma di *information hiding*
  - Il S/O è un singolo .o che collega tutte le procedure che lo compongono
    - Viene incorporato (parzialmente!) negli eseguibili delle applicazioni che lo usano
- L'unica struttura riconoscibile in essa è data dalla convenzione di attivazione delle chiamate di sistema

Architettura logica di S/O – 3



Sistema a macchina virtuale

Architettura logica di S/O – 4

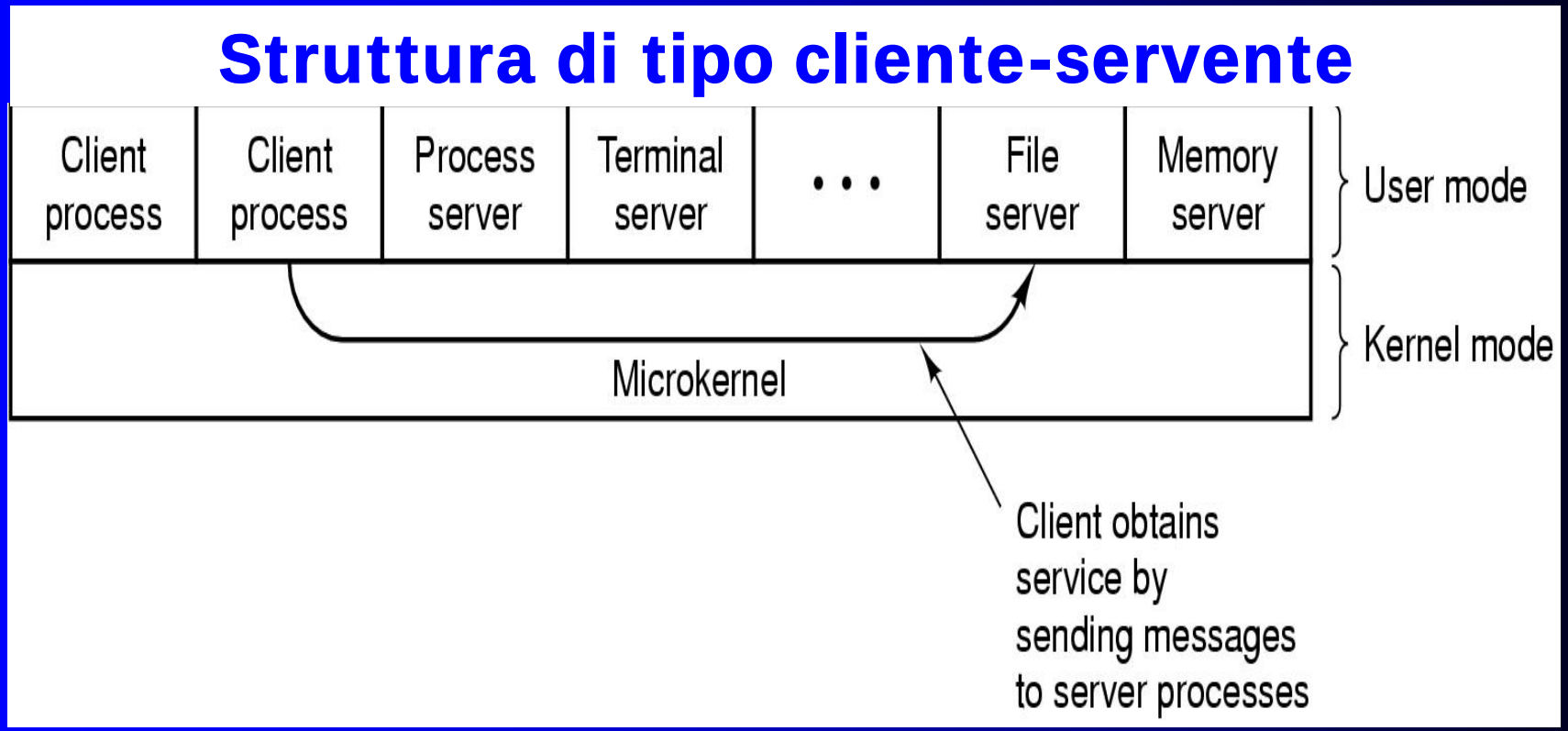
- Primi anni ’70
  - **VM/370** sviluppato da IBM *Scientific Center*, Cambridge, Massachussets
  - Basato sull’intuizione che un S/O a divisione di tempo in realtà realizza 2 fondamentali funzioni
    1. Multiprogrammazione
    2. Virtualizzazione dell’elaboratore fisico

    !
  - Separandole e ponendo 2. alla base si possono offrire copie identiche di **macchine virtuali** a S/O distinti che realizzano 1. secondo un criterio loro proprio

Architettura logica di S/O – 5

- **CMS** (*Conversational* [*Cambridge*] *Monitor System*)
  - S/O interattivo a divisione di tempo mono-utente
  - Esegue sopra una **macchina virtuale** realizzata da VM/370
- L'idea della virtualizzazione di elaboratori logici o fisici ha avuto notevole seguito
  - Intel: modo 8086 virtuale su Pentium
  - MS Windows & co.: ambiente virtuale di esecuzione MS-DOS
  - JVM: architettura portabile di elaboratore logico

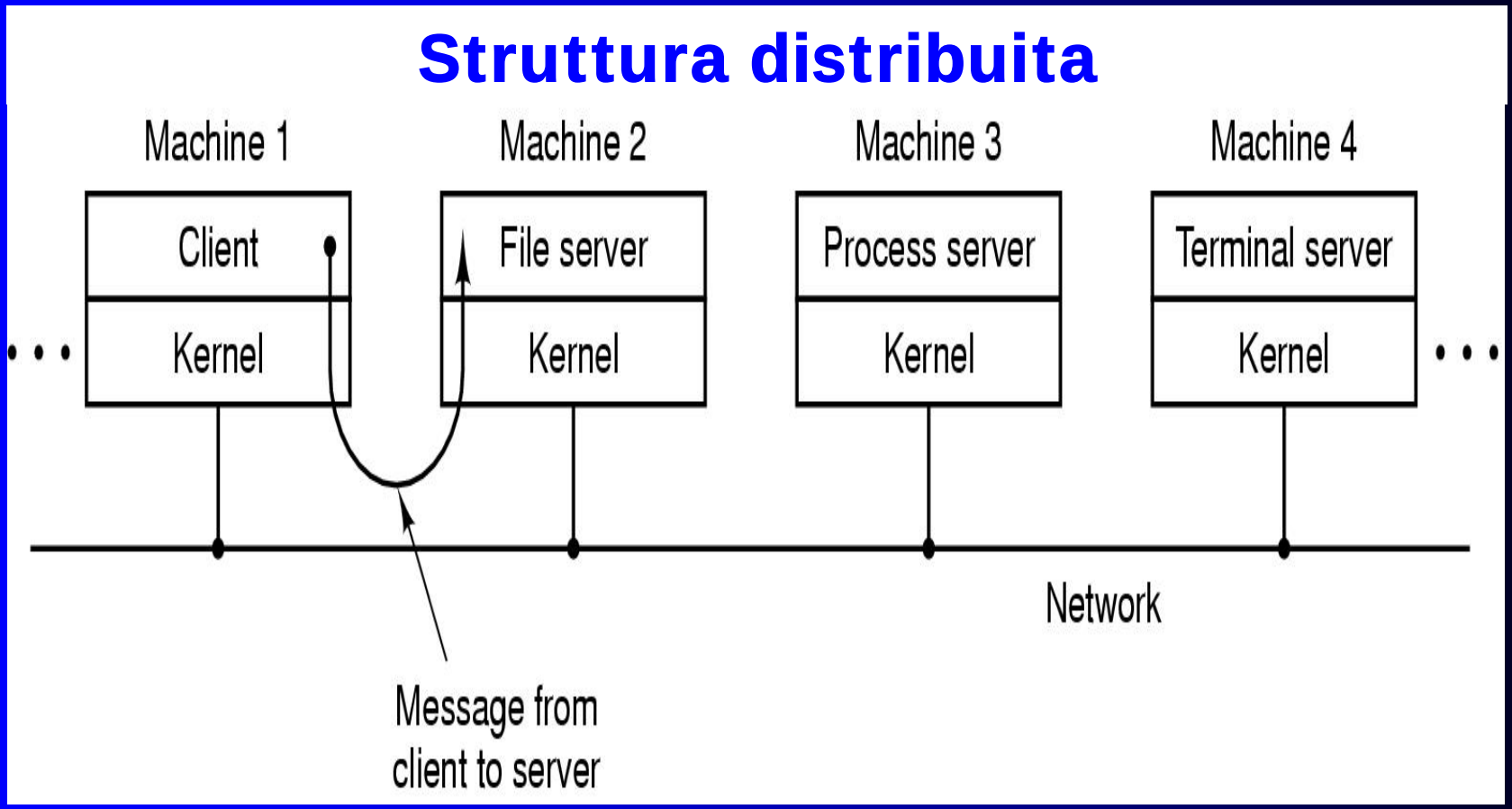
Architettura logica di S/O – 6



Architettura logica di S/O – 7

- L'architettura di S/O a modello cliente-servente è anche detta a ***micro-kernel***
- L'idea portante è di limitare al solo essenziale le responsabilità del nucleo delegando le altre a processi di sistema **nello spazio di utente**
  - I processi di sistema sono visti come serventi
  - I processi utenti sono visti come clienti
- Il ruolo del nucleo di S/O è di gestire i processi e supportare le loro comunicazioni
- Idea "pulita" ma prestazioni scadenti
  - Principalmente a causa della grande quantità di memoria che viene copiata da modo nucleo a modo utente

Architettura logica di S/O – 8



Unità metriche – 1

Exp.	Explicit	Prefix	Exp.	Explicit	Prefix
10^{-3}	0.001	milli	10^{3}	1,000	Kilo
10^{-6}	0.000001	micro	10^{6}	1,000,000	Mega
10^{-9}	0.000000001	nano	10^{9}	1,000,000,000	Giga
10^{-12}	0.000000000001	pico	10^{12}	1,000,000,000,000	Tera
10^{-15}	0.000000000000001	femto	10^{15}	1,000,000,000,000,000	Peta
10^{-18}	0.000000000000000001	atto	10^{18}	1,000,000,000,000,000,000	Exa
10^{-21}	0.000000000000000000001	zepto	10^{21}	1,000,000,000,000,000,000,000	Zetta
10^{-24}	0.000000000000000000000001	yocto	10^{24}	1,000,000,000,000,000,000,000,000	Yotta

Unità metriche – 2

Prefixes for binary multiples

In December 1998 the International Electrotechnical Commission (IEC), the leading international organization for worldwide standardization in electrotechnology, approved as an IEC International Standard names and symbols for prefixes for binary multiples for use in the fields of data processing and data transmission. The prefixes are as follows:

Prefixes for binary multiples

Factor	Name	Symbol	Origin	Derivation
2^{10}	kibi	Ki	kilobinary: $(2^{10})^1$	kilo: $(10^3)^1$
2^{20}	mebi	Mi	megabinary: $(2^{10})^2$	mega: $(10^3)^2$
2^{30}	gibi	Gi	gigabinary: $(2^{10})^3$	giga: $(10^3)^3$
2^{40}	tebi	Ti	terabinary: $(2^{10})^4$	tera: $(10^3)^4$
2^{50}	pebi	Pi	petabinary: $(2^{10})^5$	peta: $(10^3)^5$
2^{60}	exbi	Ei	exabinary: $(2^{10})^6$	exa: $(10^3)^6$

Examples and comparisons with SI prefixes

one **kibibit**	1 Kibit = 2^{10} bit = **1024 bit**
one **kilobit**	1 kbit = 10^3 bit = **1000 bit**
one **mebibyte**	1 MiB = 2^{20} B = **1 048 576 B**
one **megabyte**	1 MB = 10^6 B = **1 000 000 B**
one **gibibyte**	1 GiB = 2^{30} B = **1 073 741 824 B**
one **gigabyte**	1 GB = 10^9 B = **1 000 000 000 B**

It is suggested that in English, the first syllable of the name of the binary-multiple prefix should be pronounced in the same way as the first syllable of the name of the corresponding SI prefix, and that the second syllable should be pronounced as "bee."

It is important to recognize that the new prefixes for binary multiples are not part of the International System of Units (SI), the modern metric system.

Considerazioni preliminari – 1

- Nell'ottica degli utenti applicativi la memoria deve essere
  - Capiente
  - Veloce
  - Permanente (non volatile)
- Solo **l'intera** gerarchia di memoria nel suo insieme possiede tutte queste caratteristiche
- Il **gestore della memoria** è la componente di S/O incaricata di soddisfare le esigenze di memoria dei processi

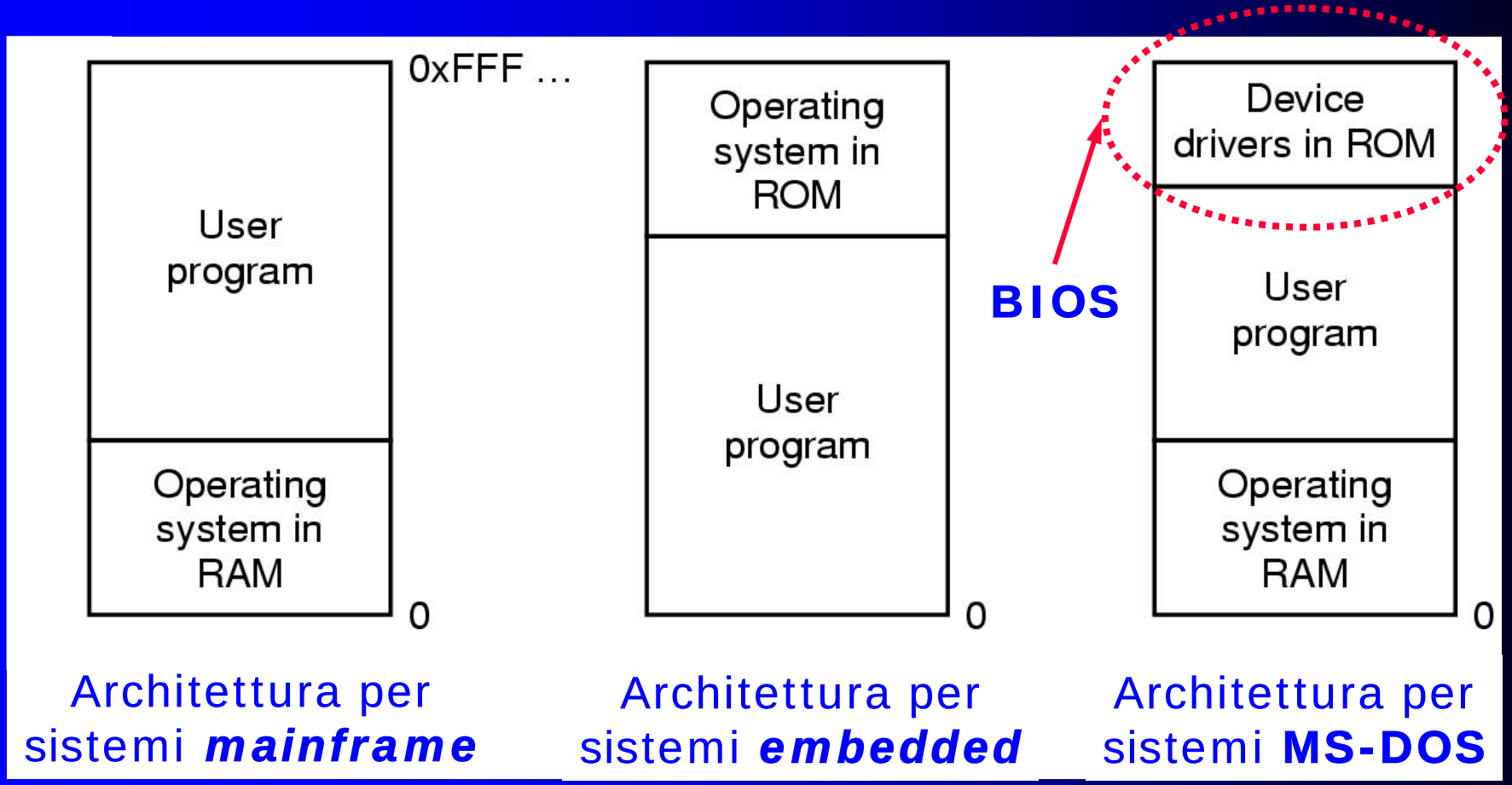
Considerazioni preliminari – 2

- Esistono due classi fondamentali di sistemi di gestione della memoria
  - Per processi allocati in modo fisso
  - Per processi destinati a migrare da memoria principale a disco durante l'esecuzione
- La memoria disponibile è in generale **inferiore** a quella necessaria per tutti i processi attivi simultaneamente

Sistemi monoprogrammati – 1

- Esegue un solo processo alla volta
- La memoria disponibile è ripartita solo tra quel processo e il S/O
- L'unica scelta progettuale rilevante è decidere **dove** allocare la memoria del S/O
  - Dati e primitive di servizio
- La parte di S/O ospitata in RAM è però solo quella che contiene l'ultimo comando invocato dall'utente

Sistemi monoprogrammati – 2



Architettura per sistemi ***mainframe***

Architettura per sistemi ***embedded***

Architettura per sistemi **MS-DOS**

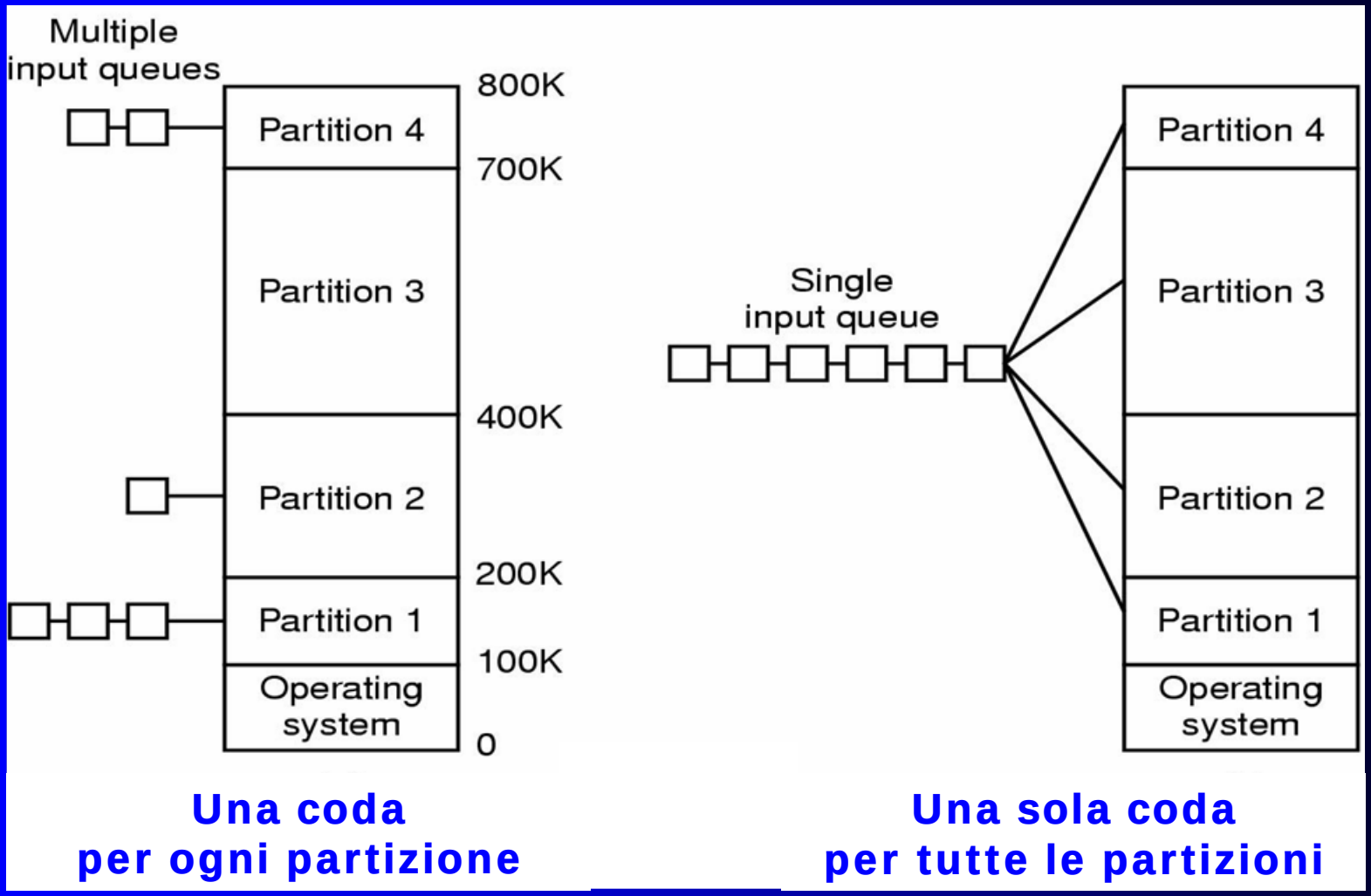
Sistemi multiprogrammati – 1

- La forma più rudimentale di gestione della memoria per questi sistemi crea una **partizione** per ogni processo
  - Staticamente all'avvio del sistema
  - Le partizioni possono avere dimensione diversa
- Il problema diventa assegnare dinamicamente processi a partizioni
  - Minimizzando la frammentazione interna

Sistemi multiprogrammati – 2

- A ogni nuovo processo (o lavoro) viene assegnata la partizione di dimensione più appropriata
  - Una coda di processi per ogni singola partizione
  - Scarsa efficacia nell'uso della memoria disponibile
- Assegnazione opportunistica
  - Una sola coda per tutte le partizioni
    - Quando si libera una partizione questa viene assegnata al processo a essa **più adatto** e più avanti nella coda
    - Oppure assegnata al "**miglior**" processo scandendo l'intera coda
      - I processi più "piccoli" sono discriminati quando invece meriterebbero di essere privilegiati in quanto più interattivi

Sistemi multiprogrammati – 3



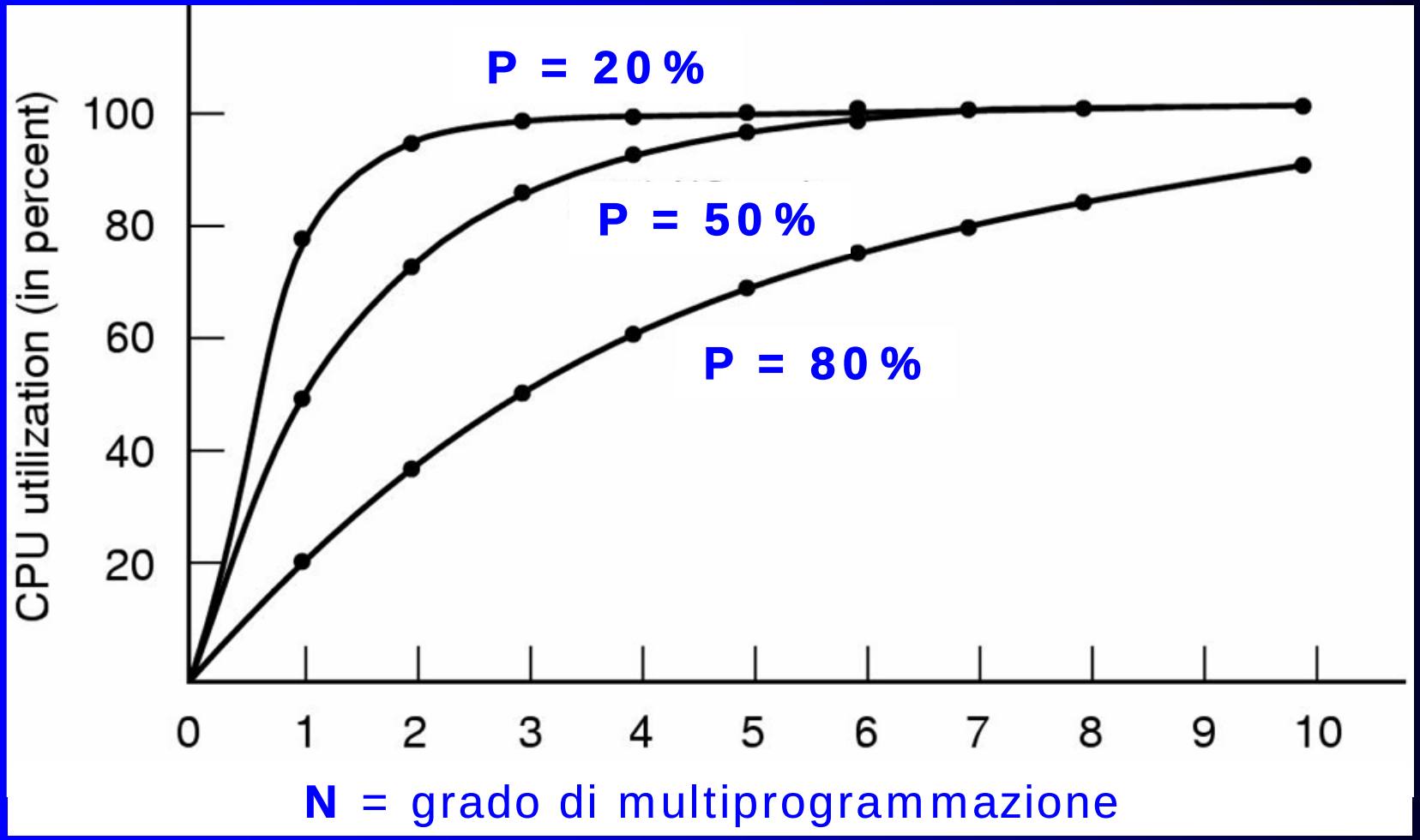
Una coda per ogni partizione

Una sola coda per tutte le partizioni

Valutazione dei vantaggi della multiprogrammazione – 1

- Stima **probabilistica** di quanti processi debbano eseguire concorrentemente per massimizzare l'utilizzazione della CPU
  - Sotto l'ipotesi che
    - Ogni processo impegni **P** % del suo tempo in attività di I/O
    - **N** processi simultaneamente in memoria
  - L'utilizzo stimato della CPU allora è $1 - P^N$

Valutazione dei vantaggi della multiprogrammazione – 2



Rilocazione e protezione

- **Rilocazione**
  - Interpretazione degli indirizzi emessi da un processo in relazione alla sua attuale collocazione in memoria
    - Occorre distinguere tra riferimenti **assoluti** permissibili al programma e riferimenti **relativi** da rilocare
- **Protezione**
  - Assicurazione che ogni processo operi soltanto nel suo spazio di memoria permissibile
    - Soluzione storica adottata da IBM
      - Memoria divisa in blocchi (2 kB) con codice di protezione per blocco (4 *bit*)
      - La PSW di ogni processo indica il suo codice di protezione
      - Il S/O blocca ogni tentativo di accedere a blocchi con codice di protezione diverso da quello della PSW corrente
    - Soluzione combinata (rilocazione + protezione)
      - Un processo può accedere memoria solo tra la **base** e il **limite** della partizione a esso assegnata
      - Valore **base** aggiunto al valore di ogni indirizzo riferito (operazione costosa)
      - Il risultato confrontato con il valore **limite** (operazione veloce)

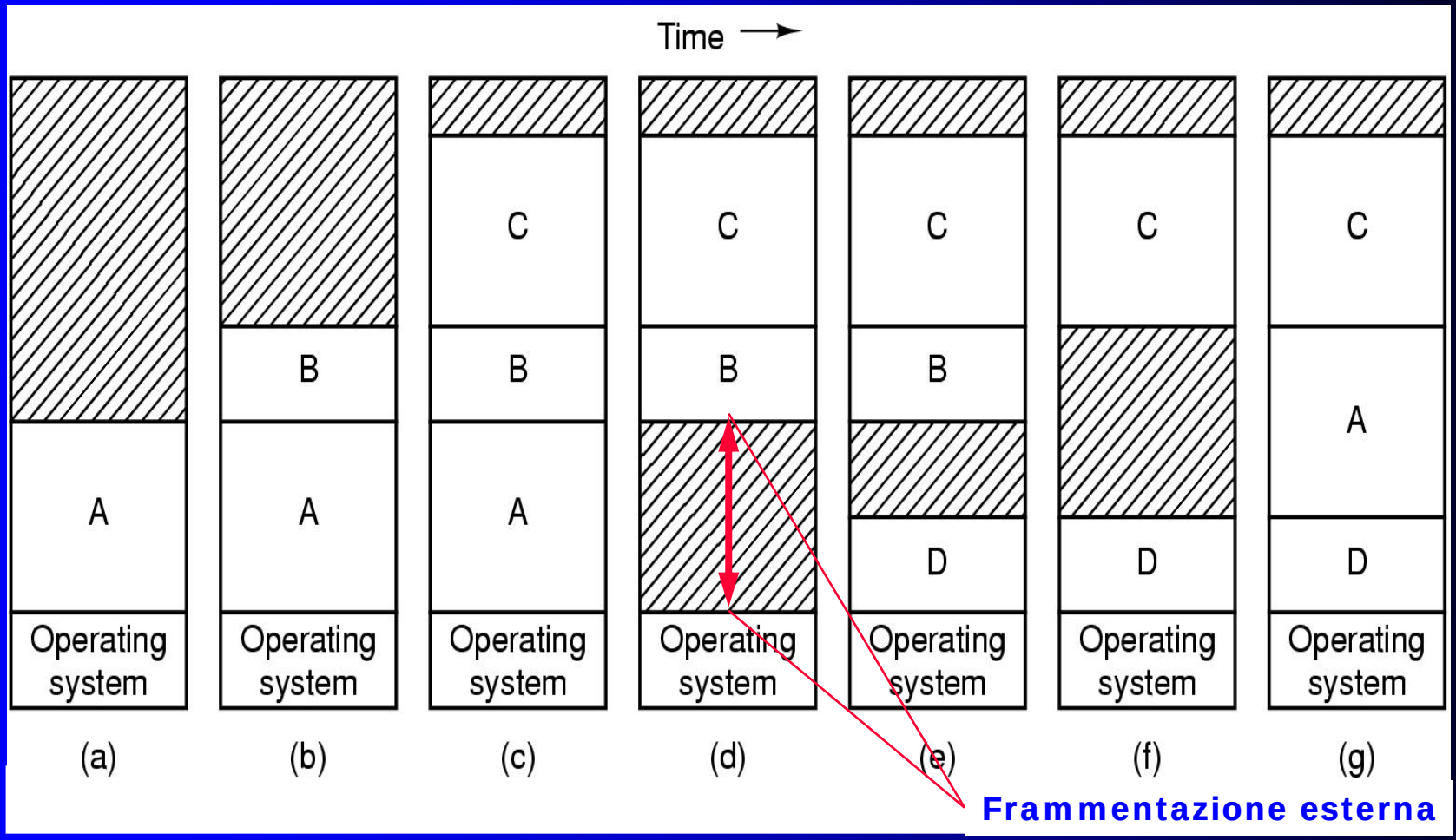
Swapping – 1

- La tecnica più rudimentale per alternare processi in memoria principale **senza** garantire allocazione fissa
- Trasferisce processi **interi** e assegna partizioni **diverse** nel tempo
- Il processo rimosso viene traslato in memoria secondaria
  - Ovviamente solo le sue parti modificate
    - Il codice non è modificabile!

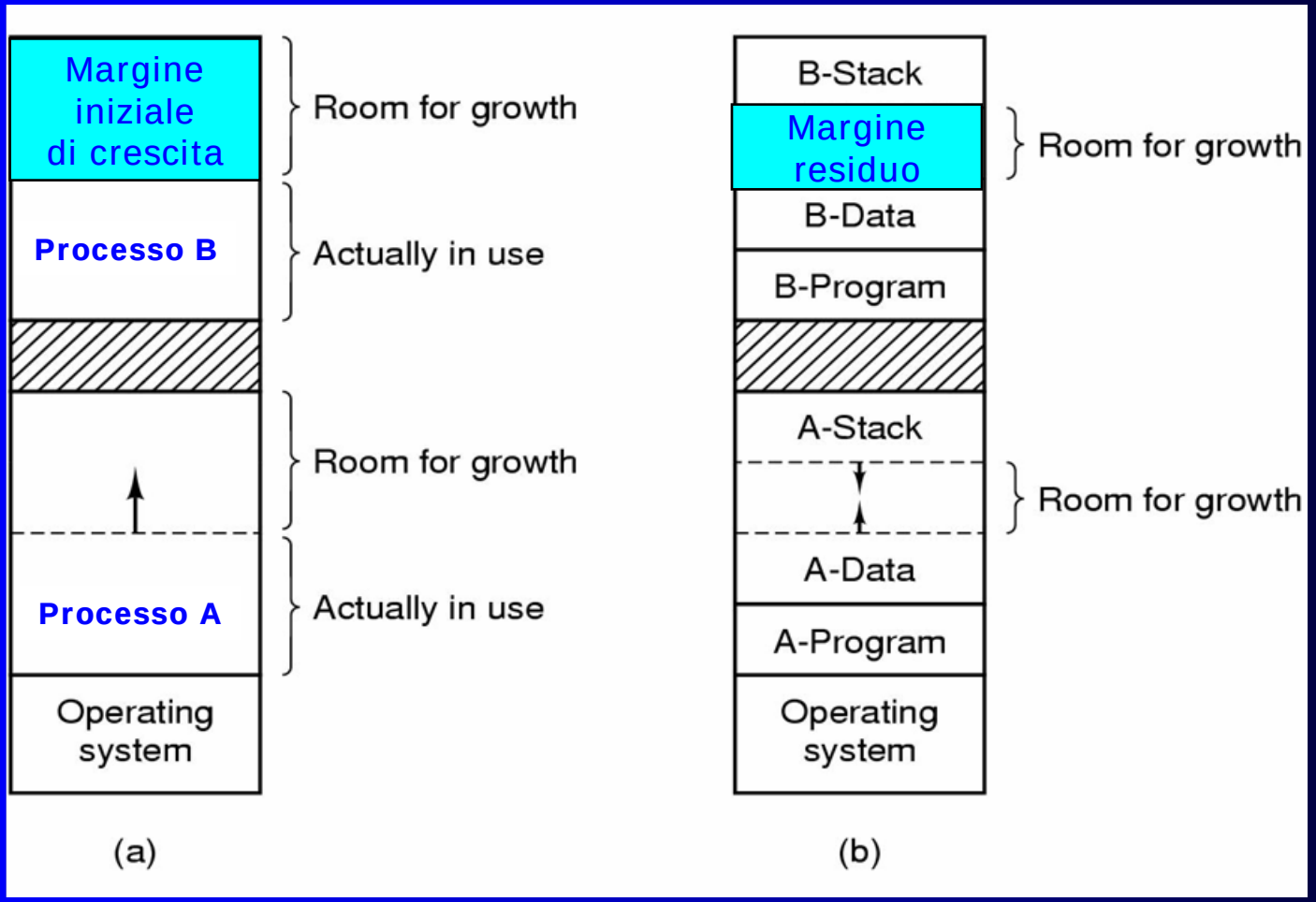
Swapping – 2

- Processi diversi richiedono partizioni di ampiezze diverse assegnate *ad hoc*
  - Rischio di frammentazione **esterna**
  - Occorre ricompattare periodicamente la memoria principale
    - Pagando un costo temporale importante!
      - Spostando 4 B in 40 ns. servono 5.37 s. per ricompattare una RAM ampia 512 MB
- Le dimensioni di memoria di un processo possono variare nel tempo!
  - Difficile ampliare dinamicamente l'ampiezza della partizione assegnata
  - Meglio assegnare **con margine**

Swapping – 3


Time
C
B
A
D
Operating system
(a)
(b)
(c)
(d)
(e)
(f)
(g)
Frammentazione esterna

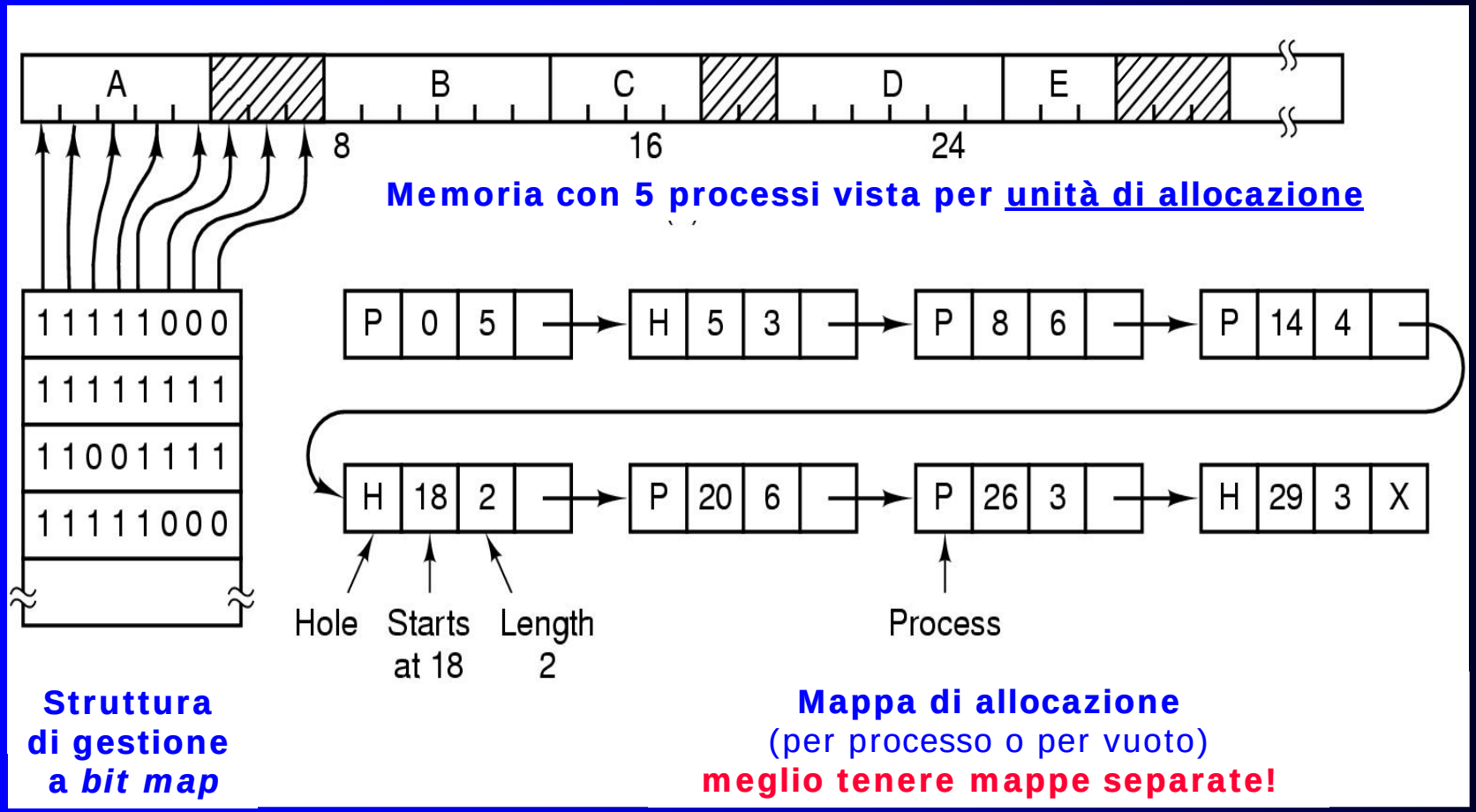
Swapping – 4


Margine iniziale di crescita
Room for growth
Processo B
Actually in use
Room for growth
Processo A
Actually in use
Operating system
(a)
B-Stack
Margine residuo
Room for growth
B-Data
B-Program
A-Stack
Room for growth
A-Data
A-Program
Operating system
(b)

Strutture di gestione – 1

- Quando la memoria principale viene allocata dinamicamente è essenziale tenere traccia del suo stato d'uso
- Due strategie principali
  - **Mappe di *bit***
    - Memoria vista come insieme di **unità di allocazione** (1 *bit* per unità)
      - Unità piccole → struttura di gestione grande
        - Esempio: Unità da 32 *bit* e RAM ampia 512 MB → struttura ampia 128 M *bit* = 16 MB → 3.1 % (= 1/32)

Strutture di gestione – 2



Strutture di gestione – 3

- La strategia alternativa usa **liste collegate**
  - Nella sua versione più semplice la memoria è vista a **segmenti**
    - Segmento = processo | spazio libero tra processi
    - Ogni elemento di lista rappresenta un segmento
      - Ne specifica punto di inizio, ampiezza e successore
      - Liste ordinate per indirizzo di base
- Varie **strategie di allocazione** di unità
  - *First fit* : il primo segmento libero ampio abbastanza
  - *Next fit* : come *First fit* ma cercando sempre avanti
  - *Best fit* : il segmento libero più adatto
  - *Worst fit* : sempre il segmento libero più ampio
  - *Quick fit* : liste diverse di ricerca per ampiezze “tipiche”

Memoria Virtuale – 1

- Una singola partizione o anche l'intera RAM sono presto divenute insufficienti per ospitare un intero processo
- La prima soluzione fu di suddividere il processo in parti chiamate *overlay*
  - Veniva caricata in RAM una parte alla volta
  - Non appena "consumata" le veniva sovrapposta la parte successiva
  - Suddivisione a cura del programmatore!

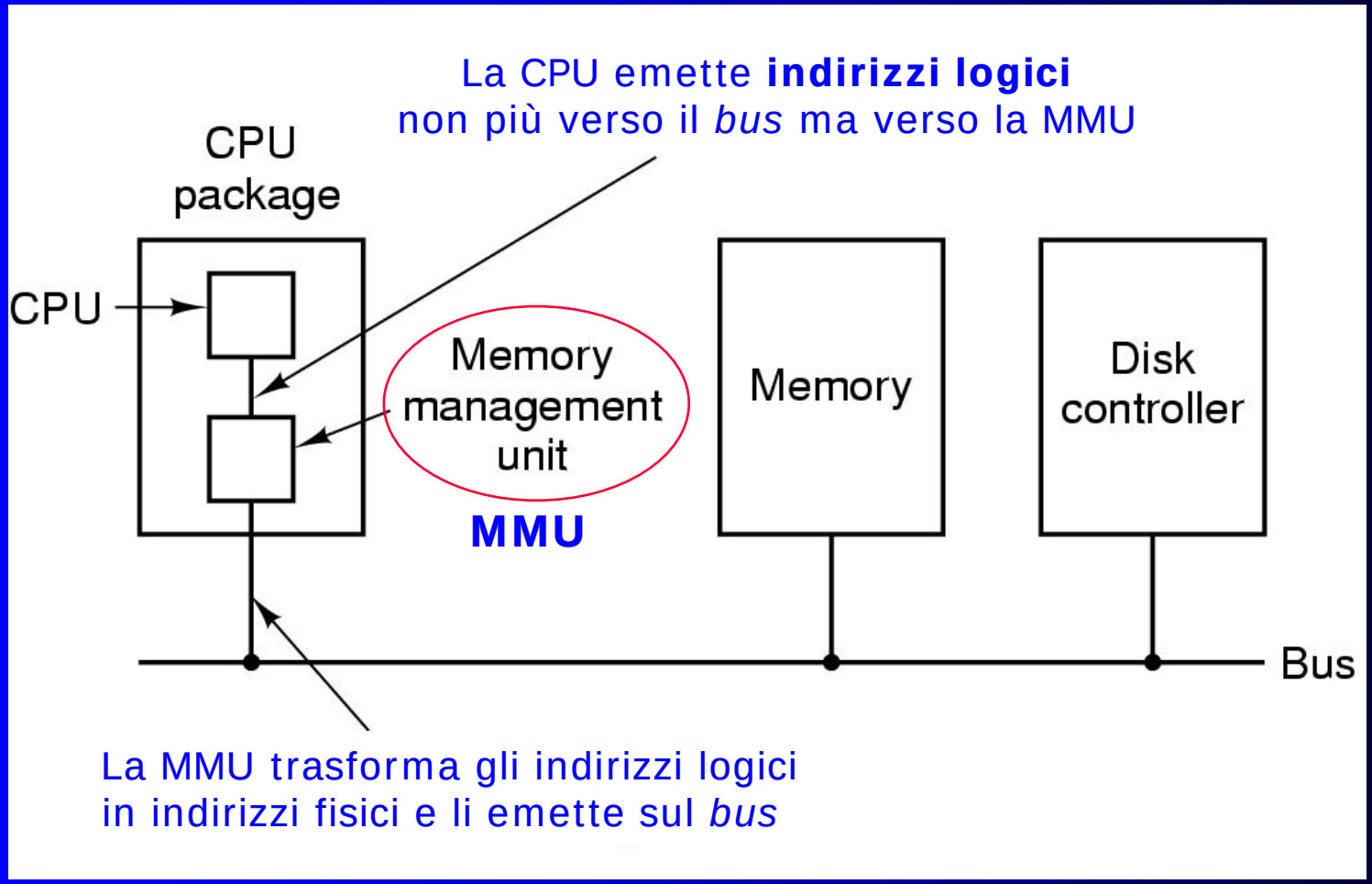
Memoria Virtuale – 2

- L'idea di **memoria virtuale** nasce nel '61
- Il principio cardine è che un singolo processo può avere ampiezza **maggiore** della RAM disponibile
  - Basta caricarne in RAM solo la parte strettamente necessaria lasciando il resto su disco
  - **Senza** intervento del programmatore
- In questo modo ogni processo ha un suo proprio spazio di memoria virtuale ampio a piacere
- Due tecniche alternative di gestione
  - **Paginazione**
  - **Segmentazione**

Memoria Virtuale – 3

- Gli indirizzi emessi dal processo **non** denotano più direttamente una locazione in RAM
  - Indirizzi logici interpretati da un'unita detta **MMU** che li mappa verso indirizzi fisici
    - **Prima** di essere emessi sul *bus*
  - Il tipo di interpretazione a carico della MMU dipende dalla tecnica adottata per la gestione della memoria virtuale

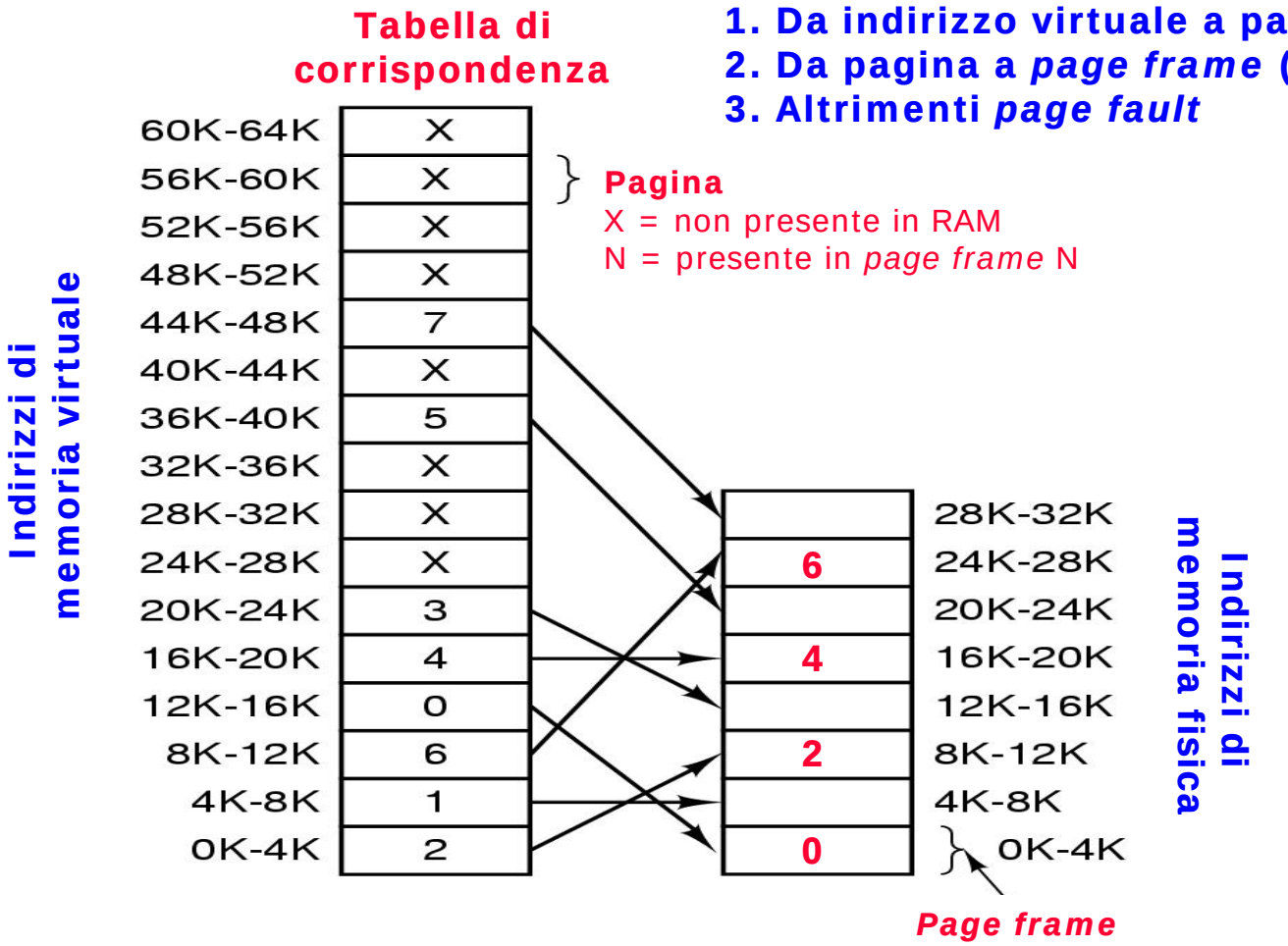
Memoria Virtuale – 4



Paginazione: premesse – 1

- La memoria virtuale è suddivisa in unità a dimensione fissa dette **pagine**
- La RAM è suddivisa in unità "contenitore" ampie come le pagine (*page frame*)
- I trasferimenti da e verso disco avvengono sempre in termini di pagine
- Di ogni pagina di una MV occorre sempre sapere se sia presente in RAM oppure no
  - *Bit* di presenza
  - Se una pagina è assente quando riferita si genera un evento *page fault* gestito dal S/O tramite **`trap`**

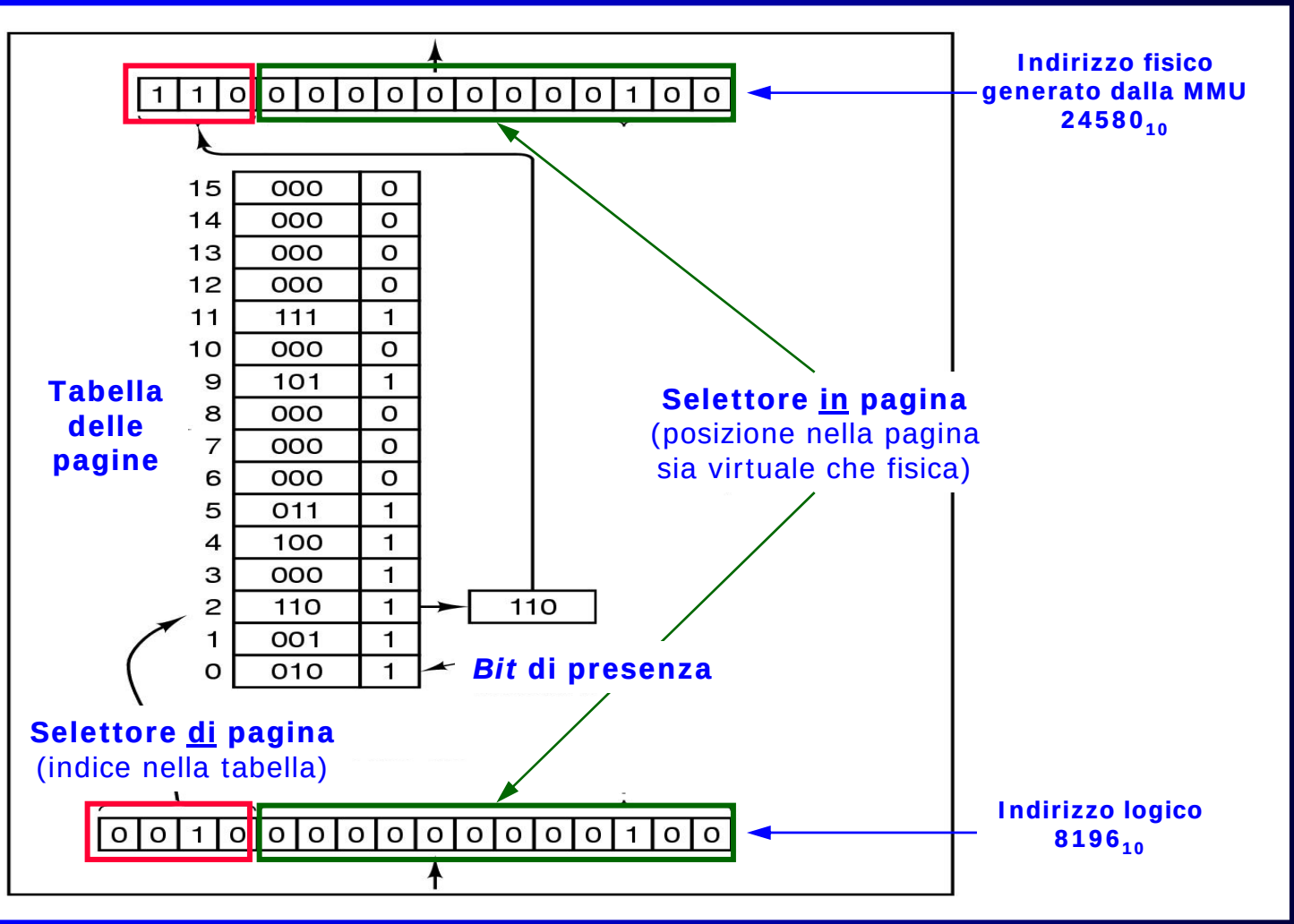
Paginazione: premesse – 2



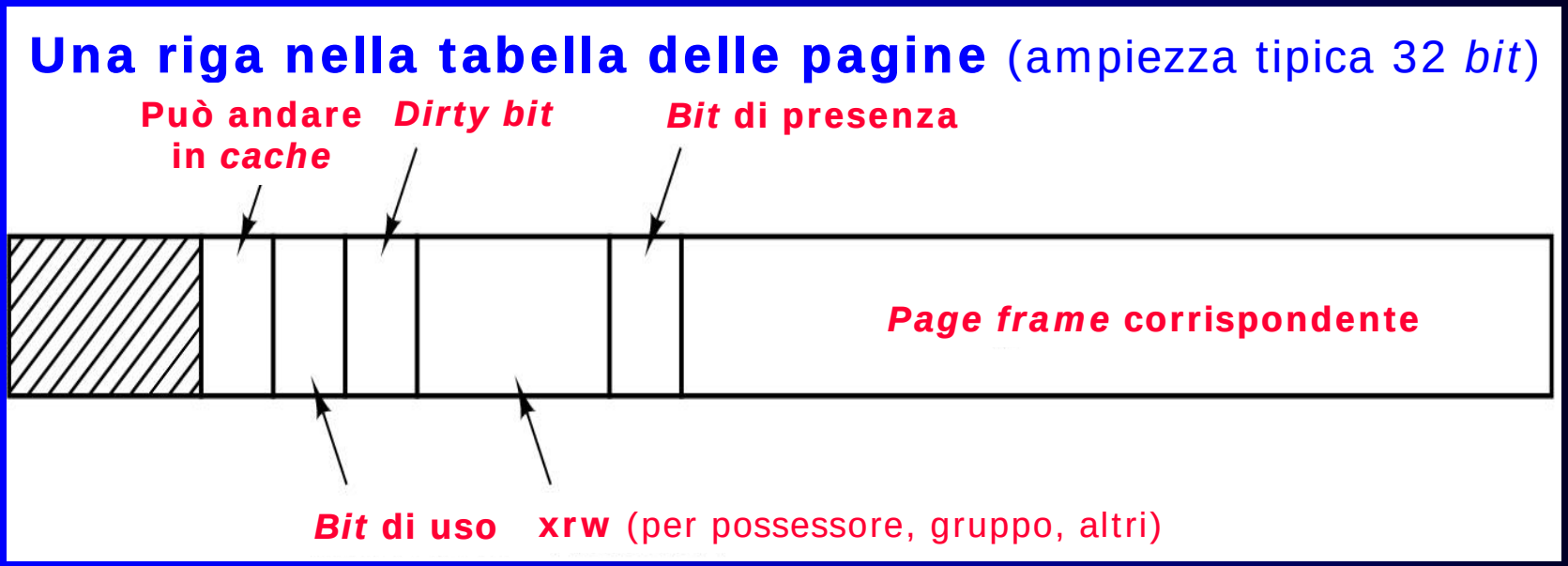
Paginazione: strutture – 1

- La traduzione da indirizzo logico a fisico avviene tramite una **tabella delle pagine**
  - Indicizzata per numero di pagina
    - Indirizzo $_{fisico} = \varphi$ (indirizzo $_{logico}$)
- La tabella può essere molto **grande**
  - Indirizzi logici da 32 *bit* e pagine da 4 KB $\rightarrow$ memoria virtuale da 4 GB = 1 M pagine!
- La traduzione deve essere molto **veloce**
  - Ogni indirizzo emesso dal processo (istruzione o operando) deve essere tradotto
    - Concettualmente come un vettore di registri caricato a ogni cambio di contesto
    - Oppure come una struttura sempre residente in RAM

Paginazione: strutture – 2



Paginazione: strutture – 3



- L'indirizzo di disco ove la pagina si trova quando non è in RAM **non** è nella tabella!
  - La tabella delle pagine serve alla MMU (*hardware*)
  - Il caricamento della pagina da disco viene effettuato dal S/O (*software*)
  - L'informazione dell'uno **non serve** all'altro

Paginazione: strutture – 4

- La tabella delle pagine è così grande che **non può** risiedere su registri
  - Dunque deve stare in RAM
  - Riferirla per ogni indirizzo emesso (istruzioni e operandi) ha un impatto devastante sulle prestazioni
- Serve una struttura supplementare più agile che ne sia come una *cache*
  - **Piccola** memoria associativa che consente scansione parallela (***Translation Lookaside Buffer***, TLB)
    - Ricerca su tutte le righe simultaneamente
  - Basata sull'osservazione che un processo in genere usa più frequentemente **poche** pagine

Paginazione: strutture – 5

- Ogni indirizzo emesso verso la MMU viene prima trattato con la TLB
  - Se la sua pagina è presente e l'accesso richiesto è permesso la traduzione avviene tramite TLB
    - **Senza** accedere alla tabelle delle pagine
  - Se non presente si ha l'equivalente di una *cache miss* e le informazioni richieste vengono caricate in TLB dalla tabella delle pagine
    - Rimpiazzando una cella in TLB e riflettendone il valore nella tabella delle pagine
      - Ma solo se cambiato!

Paginazione: strutture – 6

- Oggi le TLB sono prevalentemente realizzate in *software* invece che in *hardware* nelle MMU
  - Le prestazioni sono accettabili
  - La MMU ne guadagna in semplicità e riduzione di spazio che viene dedicato ad altri usi ritenuti più vantaggiosi (*cache*)
- Con le architetture a 64 *bit* però le tabelle delle pagine assumono dimensioni **proibitive**
  - 64 *bit* $\rightarrow$ memoria virtuale da 16 EB ($2^4 \times 2^{60}$ B)
    - (1 EB = 1 GB $\times$ 1 GB !)
  - Pagine da 4 KB $\rightarrow$ 4 P pagine
    - (1 P = 1 M $\times$ 1 G)
  - 32 *bit* per pagina in tabella $\rightarrow$ tabelle delle pagine di ampiezza 4 P $\times$ 4 B = 16 PB !
- Serve un'altra soluzione

Paginazione: strutture – 7

- La soluzione adottata impiega una **tabella invertita**
  - Non più una riga per pagina ma per *page frame* in RAM
    - Considerevole risparmio di spazio
      - Perché il numero di *page frame* è piccolo e fissato a priori
  - La traduzione da indirizzo logico a fisico diventa però molto più complessa
    - La pagina potrebbe risiedere in **qualunque** *page frame*
    - Bisognerebbe scandire l'intera tabella per trovarla
      - Per ogni indirizzo emesso dal processo!
      - Grande dispendio di tempo
  - Ricerca velocizzata dall'uso di TLB
  - E anche realizzando la tabella invertita come una tabella *hash* indicizzata da f_{Hash} (indirizzo $_{logico}$)
    - I dati relativi alle pagine i cui indirizzi logici indicizzino una stessa riga di tabella vengono collegati in lista

Paginazione: rimpiazzo – 1

- Quando si produce un *page fault* il S/O può dover **rimpiazzare** una pagina
  - Salvando su disco la pagina rimossa
    - Ma solo se modificata nell'uso
- Inopportuno rimpiazzare pagine in uso frequente
  - Altrimenti si paga prezzo doppio dovendole riportare troppo presto in RAM
- Problema del tutto analogo a quello della *cache*
  - Anche delle *cache* emulate a *software* per la gestione di informazioni logiche

Paginazione: rimpiazzo – 2

- La scelta perfetta **non** è realizzabile
  - Perché il S/O non ha modo di sapere quali pagine il processo accederà in futuro
    - Un po' come scegliere il processo più breve
- Le scelte realizzabili sono sempre e solo approssimazioni sotto-ottimali
  - Sulla base di osservazioni empiriche sull'uso recente delle pagine attualmente in RAM

Paginazione: rimpiazzo – 3

- **NRU** (*not recently used*)
  - Per ogni *page frame* vengono aggiornati
    - *Bit* M (*modified*), inizializzato a 0 dal S/0
    - *Bit* R (*referenced*), posto a 0 **periodicamente** dal S/O per stimare la frequenza d'uso
  - Le pagine nei *page frame* sono classificate in
    - **Classe 0**: non riferita, non modificata
    - **Classe 1**: non riferita, modificata
    - **Classe 2**: riferita, non modificata
    - **Classe 3**: riferita, modificata
  - NRU sceglie una pagina **a caso** nella classe non vuota a indice più vicino a 0

Paginazione: rimpiazzo – 4

- **FIFO**
  - Rimuove la pagina di ingresso più antico in RAM
    - Basta una lista ordinata di *page frame*
      - Ogni inserimento viene marcato in coda e la rimozione avviene dalla testa
- ***Second-Chance***
  - Corregge FIFO rimpiazzando solo le pagine con *bit* R = 0
    - Altrimenti il *page frame* viene considerato come appena caricato, posto in fondo alla coda e R viene posto a 0
    - Degenera in FIFO quando tutti i *page frame* siano stati recentemente riferiti

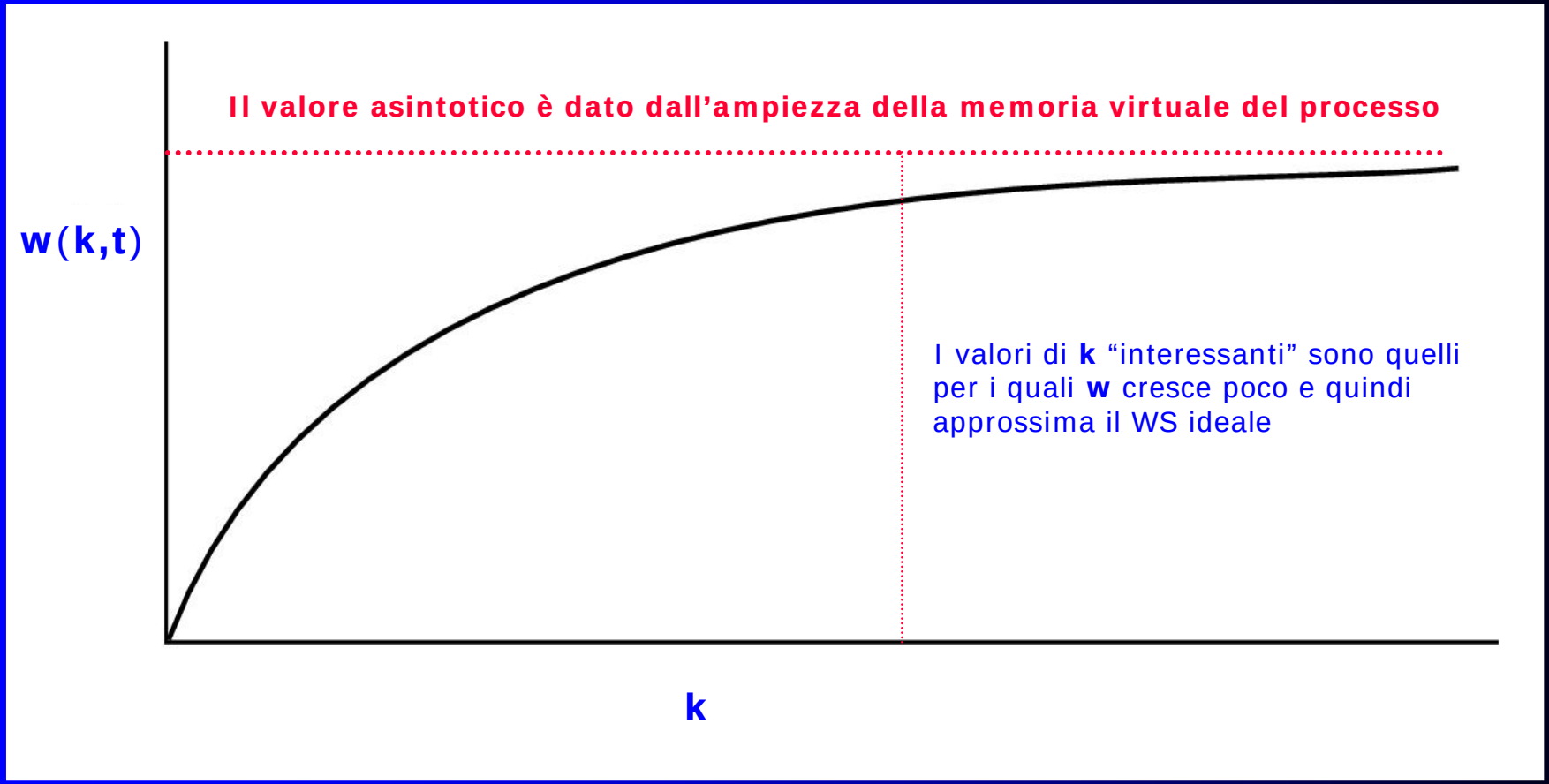
Paginazione: rimpiazzo – 5

- **Orologio**
  - Come SC ma i *page frame* sono mantenuti in una lista **circolare**
    - L'indice di ricerca si muove come una lancetta
- **LRU** (*least recently used*)
  - Necessita di *hardware* dedicato
- **Aging** (*not frequently used* modificato)
  - Realizzabile a *software*
  - Per ogni *page frame* aggiorna periodicamente un "contatore" **C** che cresce di più se **R** = 1
    - Non incrementa **C** con **R** ma gli inserisce **R** a sinistra
  - Approssima LRU con differenze importanti
    - Valuta solo periodicamente (a grana grossa)
    - Usando N *bit* per **C** perde memoria dopo N aggiornamenti

Paginazione: *working set* – 1

- Studi accurati mostrano come i processi emettano la maggior parte dei loro riferimento entro un ristretto spazio locale
  - **Località dei riferimenti**
- ***Working set*** (WS) è l'insieme di pagine che un processo ha in uso a un dato istante
  - **w**(**k, t**) è l'insieme di pagine che soddisfano i **k** riferimenti emessi al tempo **t**
    - Funzione monotona crescente

Paginazione: *working set* – 2



Paginazione: *working set* – 3

- Se si conoscesse il WS dei processi le pagine da rimpiazzare sarebbero quelle che **non** vi fossero comprese
- Conoscere precisamente il WS dei processi a tempo d'esecuzione è però **troppo costoso**
  - Quanto deve valere **k**?
  - Più facile fissare **t** come $(t, t + \Delta t)$
    - Considerando **t** come valore dell'effettivo tempo di esecuzione del processo (**tempo virtuale corrente**)
      - Non del tempo trascorso!
    - WS è fatto dalle pagine riferite dal processo nell'ultimo Δt

Paginazione: rimpiazzo - 6

- **WS approssimato**
  - Simile all'*Aging*
    - Ogni *page frame* in RAM ha un attributo temporale che indica se a un dato istante appare come riferita (R = 1)
      - L'attributo prende il valore **t** del **tempo virtuale corrente** del corrispondente processo all'arrivo di un *page fault*
      - R e M sono posti a 1 dall'*hardware*
      - R è posto a 0 se il *page frame* non risulta in uso a un controllo periodico o all'arrivo di un *page fault*
    - Al *page fault* sono rimpiazzabili le pagine con R = 0 e valore di attributo **antecedente** all'intervallo (**t** – Δt, **t**)
      - Per un Δt fissato
      - Se all'istante **t** tutti i *page frame* avessero R = 1 verrebbe rimpiazzata una pagina scelta a caso tra quelle con M = 0
    - Nel caso peggiore bisogna scandire l'**intera** RAM!

Paginazione: rimpiazzo - 7

- **WS approssimato con orologio**
  - *Page frame* organizzati in lista circolare
    - Come per l'orologio semplice
    - Ma con le informazioni del WS approssimato
  - Una "lancetta" indica il *page frame* corrente
    - Al *page fault* se R = 1 la lancetta avanza e R = 0
    - Se R = 0 si valuta l'attributo temporale
      - Se fuori da **w**(**k**,**t**) e con M = 0 allora rimpiazzo
      - Altrimenti il *page frame* va in una coda di trasferimento su disco e la lancetta avanza
        - Alla ricerca di un *page frame* rimpiazzabile direttamente
        - Quando la coda di trasferimento contiene **N** pagine si effettua trasferimento su disco
    - Se nessun *page frame* è rimpiazzabile allora si sceglie una pagina con M = 0 altrimenti quella cui punta la lancetta

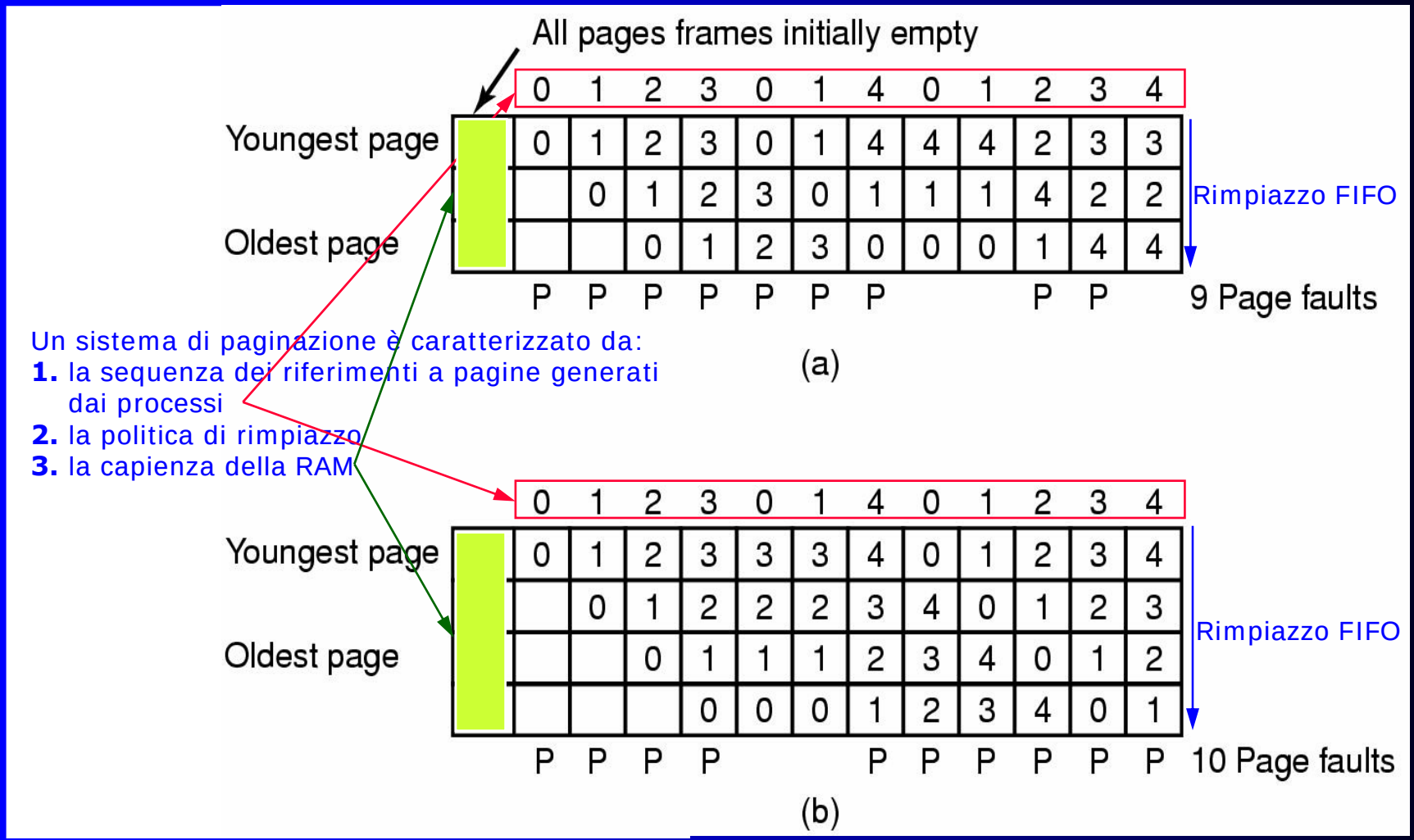
Paginazione: rimpiazzo - 8

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Paginazione: l'anomalia di Belady - 1

- Nel 1969 Lazlo Belady mostrò che la frequenza di *page fault* **non** sempre decresce al crescere dall'ampiezza della RAM
  - Un semplice contro-esempio usando FIFO come strategia di rimpiazzo
    - Sequenza di riferimenti: 0 1 2 3 0 1 4 0 1 2 3 4
    - RAM con 3 *page frame* : 9 *page fault*
    - RAM con 4 *page frame* : 10 *page fault*
- LRU è immune dall'anomalia di Belady
  - Ma la sua forma "pura" è irrealizzabile

Paginazione: l'anomalia di Belady - 2



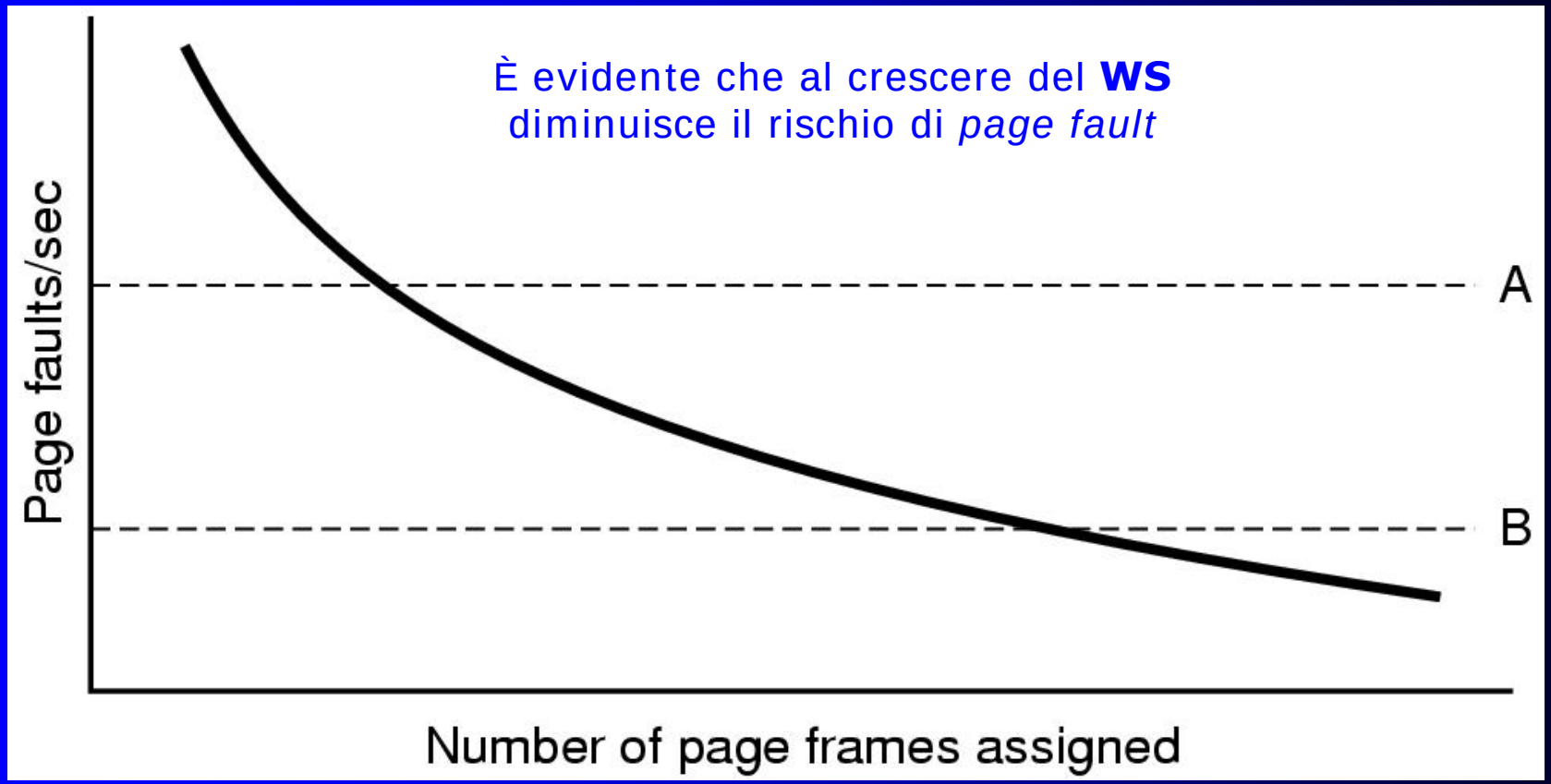
Paginazione: criteri di progetto - 1

- Nel rimpiazzare una pagina occorre scegliere consapevolmente tra
  - Politiche **locali**
    - Rimpiazzo nel WS del processo che ha causato il *page fault*
    - In tal caso ogni processo conserva una quota fissa di RAM
  - Politiche **globali**
    - La scelta avviene tra *page frame* senza distinzione di processo
    - L'allocazione di RAM a disposizione di ogni processo varia dinamicamente nel tempo

Paginazione: criteri di progetto - 2

- Le politiche **globali** sono più **efficaci**
  - Specialmente se l'ampiezza del WS può variare durante l'esecuzione
    - Però bisogna decidere **quanti** *page frame* assegnare a ogni singolo processo
- Le politiche **locali** hanno prestazioni **inferiori**
  - Se il WS di un processo cresce l'allocazione fissa causa rimpiazzi indesiderati
    - *Thrashing* (trebbiatura ☺)
      - Anche con RAM disponibile non usata da altri processi
  - Se il WS si riduce si ha invece spreco di memoria
- Non tutte le politiche si adattano all'uso in entrambe le varianti

Paginazione: criteri di progetto - 3



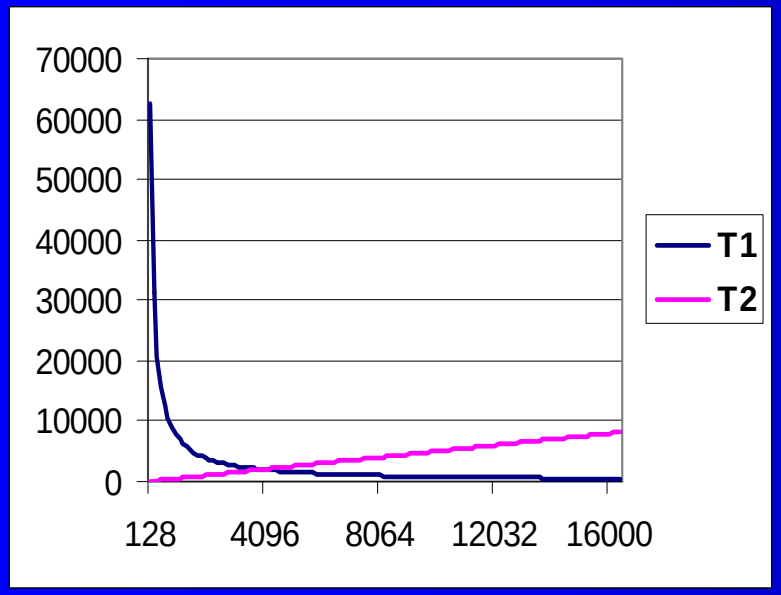
Paginazione: criteri di progetto - 4

- Quale dimensione di pagina?
  - Pagine **ampie**
    - Maggiore rischio di **frammentazione interna**
      - In media ogni processo lascia inutilizzata **metà** del suo ultimo *page frame*
  - Pagine **piccole**
    - Maggiore ampiezza della tabella delle pagine
- Il valore ottimo può essere definito matematicamente
  - **σ** B in media in uso per processo
  - **π** B per pagina
  - **ε** B per riga in tabella delle pagine
  - "Spreco" in funzione di **π** come $f(\boldsymbol{\pi}) = (\boldsymbol{\sigma} / \boldsymbol{\pi}) \times \boldsymbol{\varepsilon} + \boldsymbol{\pi} / 2$
    - Il minimo di $f(\boldsymbol{\pi})$ si ha per $\boldsymbol{\pi} = \sqrt{(2\ \boldsymbol{\sigma} \times \boldsymbol{\varepsilon})}$

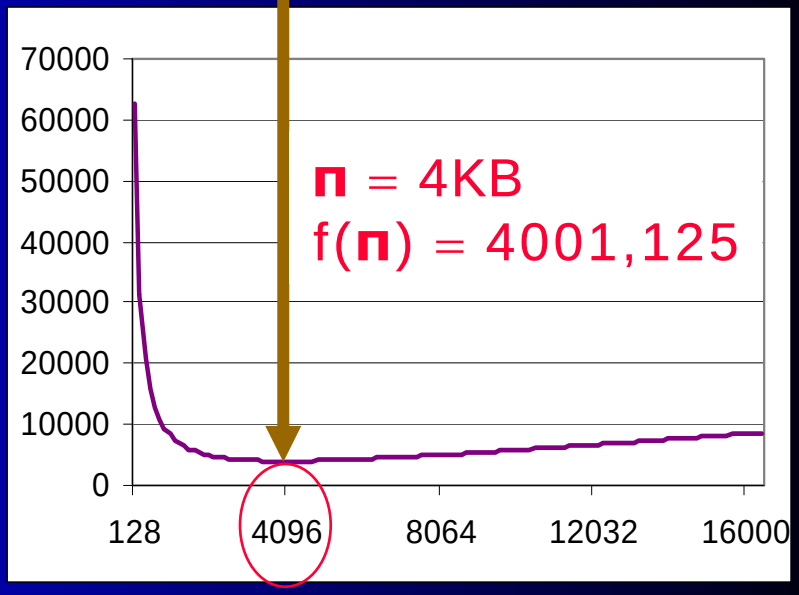
Paginazione: criteri di progetto - 5

- Per **σ** = 1 MB e **ε** = 8 B si ha **π** = 4 KB = 4096 B
- f(**π**) = (**σ** / **π**) × **ε** + **π** / 2

T1 **T2**



T1+T2



Paginazione: criteri di progetto - 6

- Come visto, per $\sigma = 1$ MB e $\varepsilon = 8$ B si ha $\pi = 4$ KB
- Per RAM di ampiezza crescente può convenire un valore di π maggiore
  - Ma di certo non linearmente
- In generale la memoria virtuale **non** è distinta per dati e istruzioni
  - Nella prima metà del ’70 vi sono stati elaboratori importanti (PDP-11) che fornivano invece spazi di indirizzamento distinti
    - *Programmed Data Processor* (2 KB *cache*, 2 MB RAM)

Paginazione: realizzazione – 1

- Il S/O compie azioni chiave
  - A ogni creazione di processo
    - Per determinare l'ampiezza della sua allocazione
    - Per creare la tabella delle pagine corrispondente
  - A ogni cambio di contesto
    - Per caricare la MMU e "pulire" la TLB
  - A ogni *page fault*
    - Per analizzare il problema e operare il rimpiazzo
  - A ogni terminazione di processo
    - Per rilasciarne i *page frame*
    - Per rimuoverne la tabella delle pagine

Paginazione: realizzazione – 2

- Per trattare un *page fault* bisogna capire quale riferimento è fallito
  - Per poter completare correttamente l'istruzione interrotta
- Il *Program Counter* dice a quale indirizzo il problema si è verificato
  - Ma non sa distinguere tra istruzione e operando
- Capirlo è compito del S/O
  - Orrendamente complicato dai molti effetti laterali causati dagli "acceleratori" *hardware*
  - Il S/O deve annullare lo stato erroneo e ripetere daccapo l'istruzione fallita

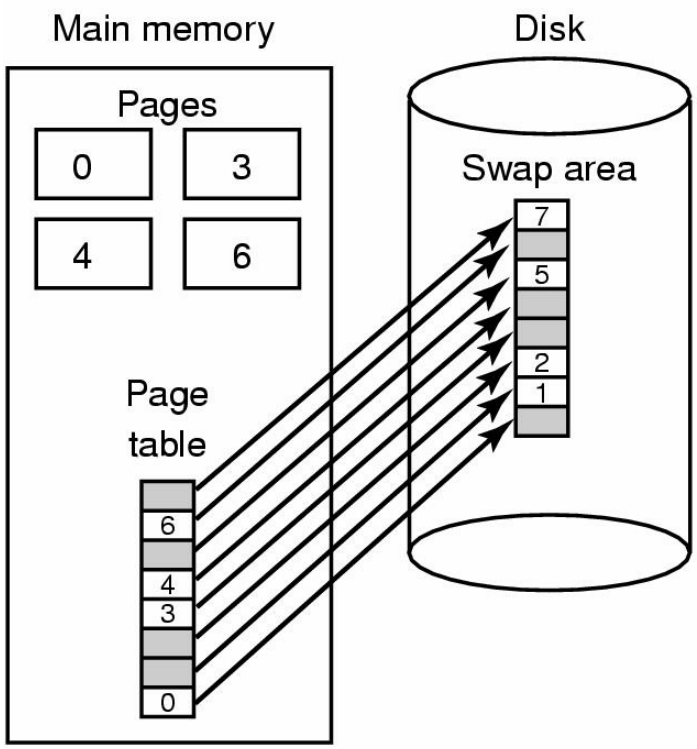
Paginazione: realizzazione – 3

- Un'area del disco può essere riservata per ospitare le pagine temporaneamente rimpiazzate
  - Area di *swap*
- Ogni processo ne riceve in dote una frazione
  - Che rilascia alla sua terminazione
  - I puntatori (base, ampiezza) a questa zona devono essere mantenuti nella tabella delle pagine del processo
    - Ogni indirizzo virtuale mappa nell'area di *swap* direttamente rispetto alla sua base

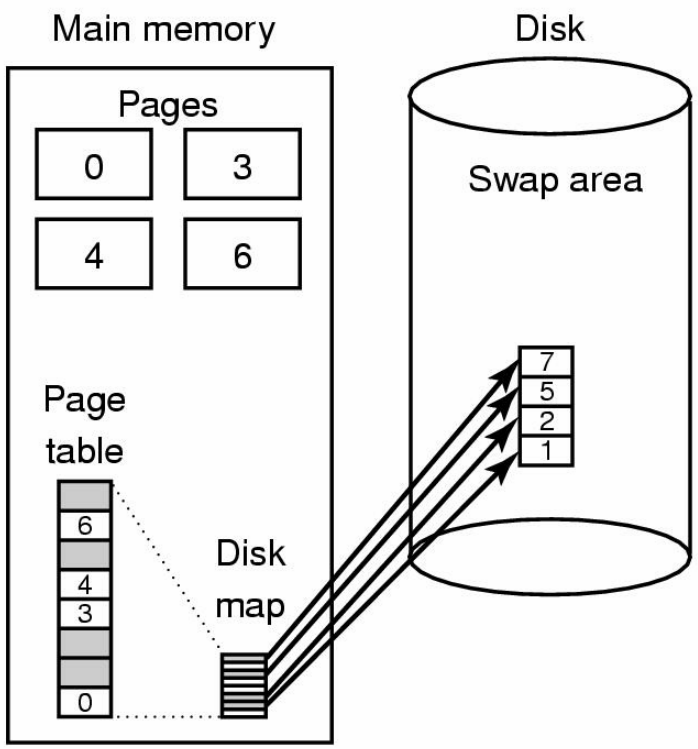
Paginazione: realizzazione – 4

- Idealmente
  - L'intera immagine del processo potrebbe andare subito nell'area di *swap* alla creazione del processo
  - Altrimenti potrebbe andare tutta in RAM e spostarsi nell'area di *swap* quando necessario
- Però sappiamo che i processi **non** hanno dimensione costante
  - Allora è meglio che l'area di *swap* sia frazionata per codice e dati
- Se l'area di *swap* **non** fosse riservata allora occorrerebbe ricordare in RAM l'indirizzo su disco di **ogni** pagina rimpiazzata
  - Informazione associata alla tabella delle pagine

Paginazione: realizzazione – 5



Area di *swap* pre-assegnata e mappata automaticamente dalla tabella delle pagine

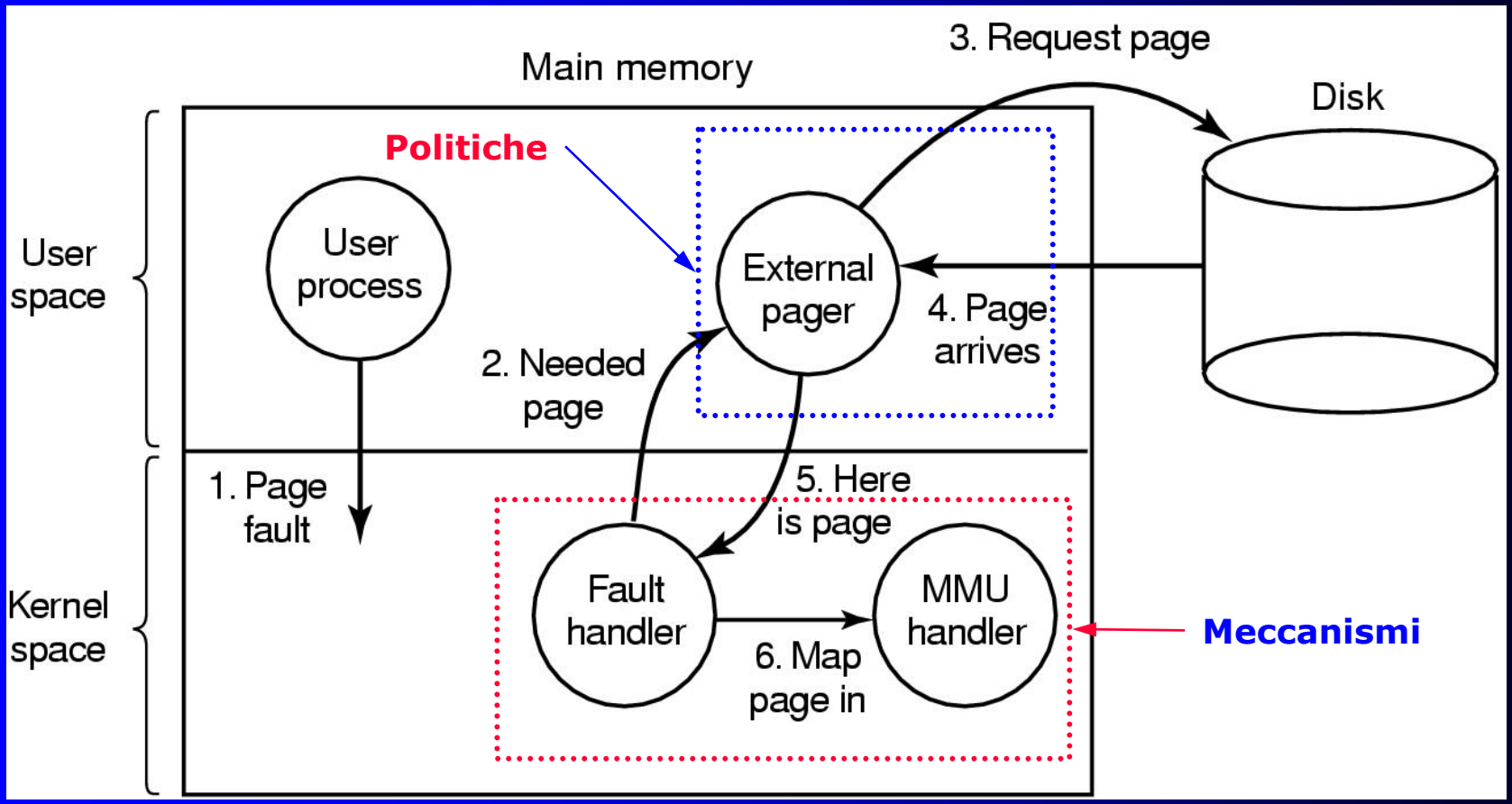


Area di *swap* assegnata a richiesta e mappata esplicitamente dalla tabella delle pagine

Paginazione: realizzazione – 6

- Per separare le politiche dai meccanismi
  - Conviene svolgere nel nucleo del S/O **solo** le azioni più delicate
    - Gestione della MMU
      - Specifica dell'architettura *hardware*
    - Trattamento **immediato** del *page fault*
      - Largamente indipendente dall'*hardware*
  - Demandando il resto della gestione a un processo esterno al nucleo
    - Scelta delle pagine e loro trasferimento
      - Trattamento **differito** del *page fault*

Paginazione: realizzazione – 7



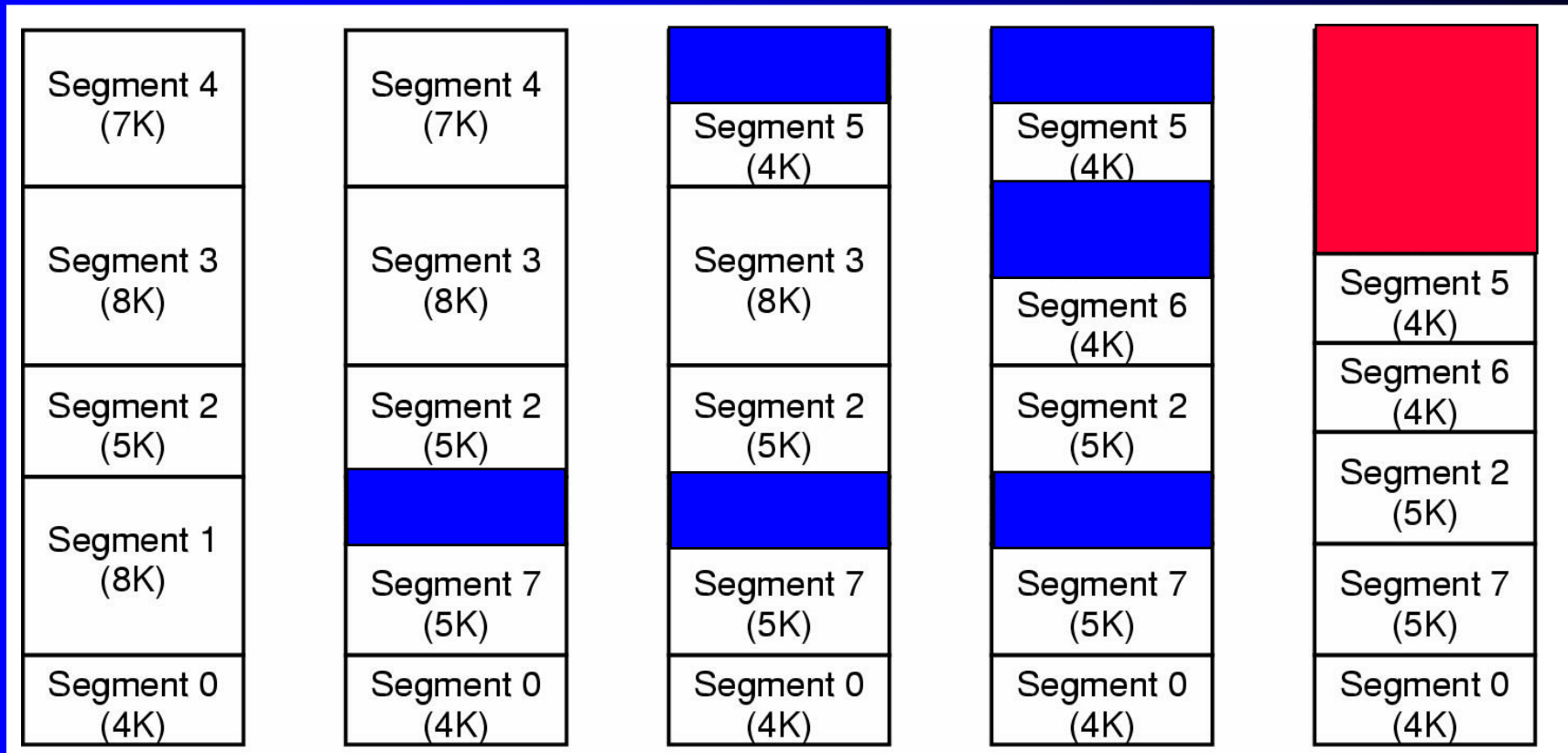
Segmentazione: premesse – 1

- Spazi di indirizzamento completamente **indipendenti** gli uni dagli altri
  - Per dimensione e posizione in RAM
    - Entrambe possono variare dinamicamente
- Entità **logica** nota al programmatore e destinata a contenere informazioni **coese**
  - Codice di una procedura
  - Dati di inizializzazione di un processo
  - *Stack* di processo
- Si presta a schemi di **protezione** specifica
  - Perché il **tipo** del suo contenuto può essere stabilito a priori
    - Ciò che **non** si può fare con la paginazione
- Causa frammentazione **esterna**

Segmentazione: premesse – 2

	Paginazione	Segmentazione
Il programmatore ne deve essere consapevole	No	Sì
Consente N spazi di indirizzamento lineari	N = 1	N ≥ 1
La sua ampiezza può eccedere la capacità della RAM	Sì	Sì
Consente di separare e distinguere tra codice e dati	No	Sì
Consente di gestire contenuti a dimensione variabile nel tempo	No	Sì
Consente di condividere parti di programmi tra processi	No	Sì
A quale obiettivo risponde	Consentire spazi di indirizzamento più grandi della RAM	Consentire la separazione logica tra aree dei processi e la loro protezione specifica

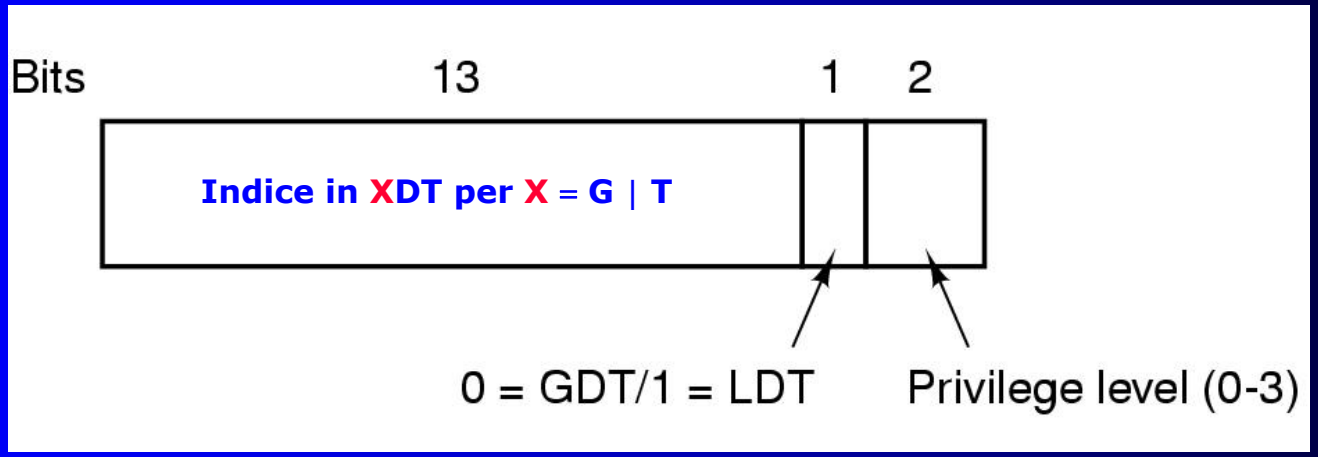
Segmentazione: realizzazione – 1



Segmentazione: realizzazione – 2

- Vista la grande ampiezza potenziale i segmenti sono spesso **paginati**
- Nel caso del Pentium di Intel
  - Fino a 16 K segmenti indipendenti (14 *bit*)
    - Ciascuno di ampiezza massima 4 GB (32 *bit*)
  - Una LDT per processo
    - *Local Descriptor Table*
      - Descrive i segmenti del processo
  - Una singola GDT per l'intero sistema
    - *Global Descriptor Table*
      - Descrive i segmenti del S/O

Segmentazione: realizzazione – 3



- 6 registri di segmento
  - Di cui 1 denota il segmento corrente
- LDT e GDT contengono $2^{13} = 8$ K descrittori di segmento
  - I descrittori di segmento sono espressi su 8 B
    - La **base** del segmento in RAM è espressa su 32 *bit*
    - Il **limite** su 20 *bit* per verificare la legalità dell'*offset* fornito dal processo
      - Consente ampiezza massima a 1 MB (per **granularità** a B)
      - Oppure 1 M pagine da 4 KB ovvero 4 GB (per granularità a pagine)

Segmentazione: realizzazione – 4

- L'indirizzo **lineare** ottenuto da (base di segmento + *offset*) può essere interpretato come
  - Indirizzo **fisico** se il segmento considerato non è paginato
  - Indirizzo **logico** altrimenti
    - Nel qual caso il segmento viene visto come una memoria virtuale paginata e l'indirizzo come virtuale in essa
      - 10 *bit* : indice in catalogo di tabelle delle pagine
        - 2^{10} righe da 32 *bit* ciascuna (base di tabella denotata)
      - 10 *bit* : indice in tabella delle pagine selezionata
        - 2^{10} righe da 32 bit ciascuna (base di *page frame*)
      - 12 *bit* : posizione nella pagina selezionata
        - *Offset* in pagina da 4 KB

Considerazioni generali – 1

- La maggior parte dell'informazione applicativa (i dati) ha **durata**, **ambito** e **dimensione** più ampi della vita delle applicazioni che la usano
  - Tre esigenze fondamentali
    - Persistenza dei dati
    - Possibilità di condividere dati tra applicazioni distinte
    - Nessun limite di dimensione fissato a priori
- Il *file system* è il servizio di S/O progettato per soddisfare questi bisogni

Considerazioni generali – 2

- Il termine *file* designa un insieme di dati correlati, residenti in **memoria secondaria** e trattati unitariamente
  - *File* = raccoglitore, *dossier*
- Dal punto di vita dell'utente il *file* è la più piccola unità di accesso alla memoria secondaria
- Il *file system* (FS) è la parte del S/O che si occupa delle operazioni sui *file*
  - Organizzazione
  - Gestione
  - Realizzazione
  - Accesso

Aspetti generali – 3

- La progettazione di FS affronta 2 problemi chiave
  - **Cosa** occorre offrire all'utente applicativo e secondo quali forme concrete
    - Modalità di **accesso** a *file*
    - Struttura **logica** e **fisica** di *file*
    - Operazioni ammissibili su *file*
  - **Come** realizzarlo in modo pratico ed economico
    - Garantendo la massima indipendenza delle operazioni dall'architettura fisica di supporto

Il *file*

- Il *file* è un concetto logico realizzato tramite **meccanismi di astrazione**
  - Per salvare informazione su memoria secondaria e potendola ritrovare in seguito senza conoscerne né la struttura logica e fisica né il funzionamento
  - All'utente non interessa come ciò avviene
  - Interessa invece poter designare le proprie unità di informazione mediante **nomi logici** unici e distinti
  - L'utente vede e tratta solo **nomi** di *file*
  - Le caratteristiche distintive di un *file* sono
    - Attributi (tra cui il nome utente)
    - Struttura interna
    - Operazioni ammesse

Attributi di *file* – 1

- **Nome**
  - Stringa composta da 8 – 255 caratteri, inclusi numeri e caratteri speciali
  - Con ≥ 1 estensioni che possono designare il “tipo” di *file* come visto dall’utente
    - **MS-DOS** (base di Windows 95 e Windows 98)
      - Nomi da 1 – 8 caratteri, con ≤ 1 estensione da 1 – 3 caratteri, **designante**, senza distinzione tra maiuscolo e minuscolo (*case insensitive*)
    - **UNIX** (base di GNU/Linux)
      - Nomi fino a 14 (ora 255) caratteri, *case sensitive*, con estensioni, solo **informative**, senza limite di numero e di ampiezza
  - In generale, l’utente può configurare presso il S/O l’associazione tra l’**ultima** estensione del *file* e il tipo applicativo corrispondente

Attributi di *file* – 2

- **Dimensione corrente**
- **Data di creazione**
  - Può non essere mostrata
- **Data di ultima modifica**
  - Indica la “freschezza” del contenuto
- **Creatore e possessore**
  - Possono essere distinti
    - P.es.: il compilatore crea *file* di proprietà dell’utente
- **Permessi di accesso**
  - Lettura, scrittura, esecuzione

Attributi di *file* – 3

Flag = bit

Protezione	Permesso di accesso al *file*	
Password	Chiave di accesso al *file*	
Creatore	Identità del processo che ha creato il *file*	
Proprietario	Identità del processo utilizzatore del *file*	
Uso	0 – lettura/scrittura	1 – sola lettura (*read-only*)
Visibilità	0 – normale	1 – *file* non visibile (*hidden*)
Livello	0 – normale	1 – *file* di sistema
Archiviazione	0 – salvato (*backed up*)	1 – non salvato
Tipo di contenuto	0 – ASCII	1 – binario
Tipo di accesso	0 – sequenziale	1 – casuale (*random*)
Permanenza	0 – normale	1 – da eliminare dopo l'uso (*temporary*)
Accesso esclusivo	0 – libero	≠ 0 – bloccato (*locked*)

Strutture dati di *file* – 1

Qui intervengono le caratteristiche interne del FS

- La struttura dei dati all'interno di un *file* può essere considerata da 3 punti di vista distinti
  - Livello **utente**
    - Il programma applicativo associa **autonomamente** significato al contenuto grezzo del *file*
  - Livello di **struttura logica**
    - A monte dell'interpretazione dell'utente il S/O organizza i dati grezzi in strutture logiche per facilitarne il trattamento
  - Livello di **struttura fisica**
    - Il S/O mappa le strutture logiche sulle strutture fisiche della memoria secondaria disponibile (p.es.: settori o blocchi su disco)
- Le possibili strutture **logiche** di un *file* sono
  - A sequenza di *byte*
  - A *record* di lunghezza e struttura interna fissa
  - A *record* di lunghezza e struttura interna variabile

Struttura logica di *file* – 1

- **Sequenza di *byte* (*byte stream*)**
  - La strutturazione logica più semplice e flessibile
    - La scelta di UNIX ($\rightarrow$ GNU/Linux) e MS Windows
  - Il programma applicativo sa come dare significato al contenuto informativo del *file*
    - Minimo sforzo per il S/O
  - L'accesso ai dati utilizza un puntatore relativo all'inizio del *file*
  - Lettura e scrittura operano su blocchi di *byte*

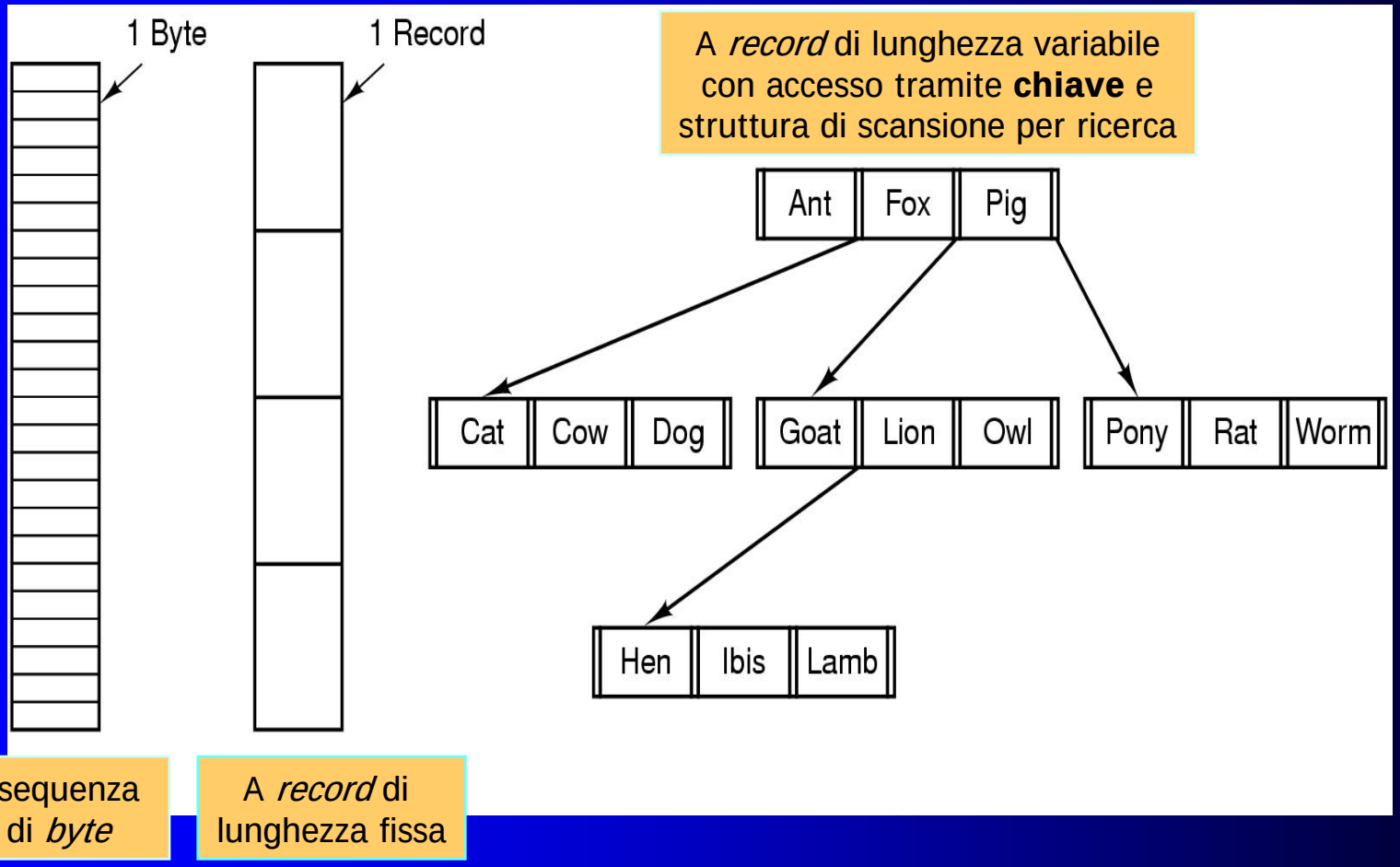
Struttura logica di *file* – 2

- ***Record* di lunghezza e struttura fissa**
  - Gli spazi non utilizzati sono riempiti da caratteri speciali (p.es.: NULL o SPACE)
  - Il S/O **deve** conoscere la struttura interna del *file*
    - Per muoversi al suo interno
  - L'accesso ai dati è sequenziale e utilizza un puntatore al *record* corrente
  - Lettura e scrittura operano su *record* singoli
  - Scelta ormai obsoleta e legata a specifiche limitazioni dell'architettura di sistema

Struttura logica di *file* – 3

- ***Record* di lunghezza e struttura variabile**
  - La struttura interna di ogni *record* viene descritta e identificata univocamente da una chiave (*key*) posta in posizione fissa e nota entro il *record*
  - Le chiavi vengono raccolte in una tabella a parte, ordinata per chiave, contenente anche i puntatori all'inizio di ciascun *record*
  - L'accesso ai dati avviene per chiave
  - Uso abbastanza diffuso in sistemi *mainframe*
    - Per applicazioni gestionali "massicce" e specializzate

Struttura logica di *file* – 4



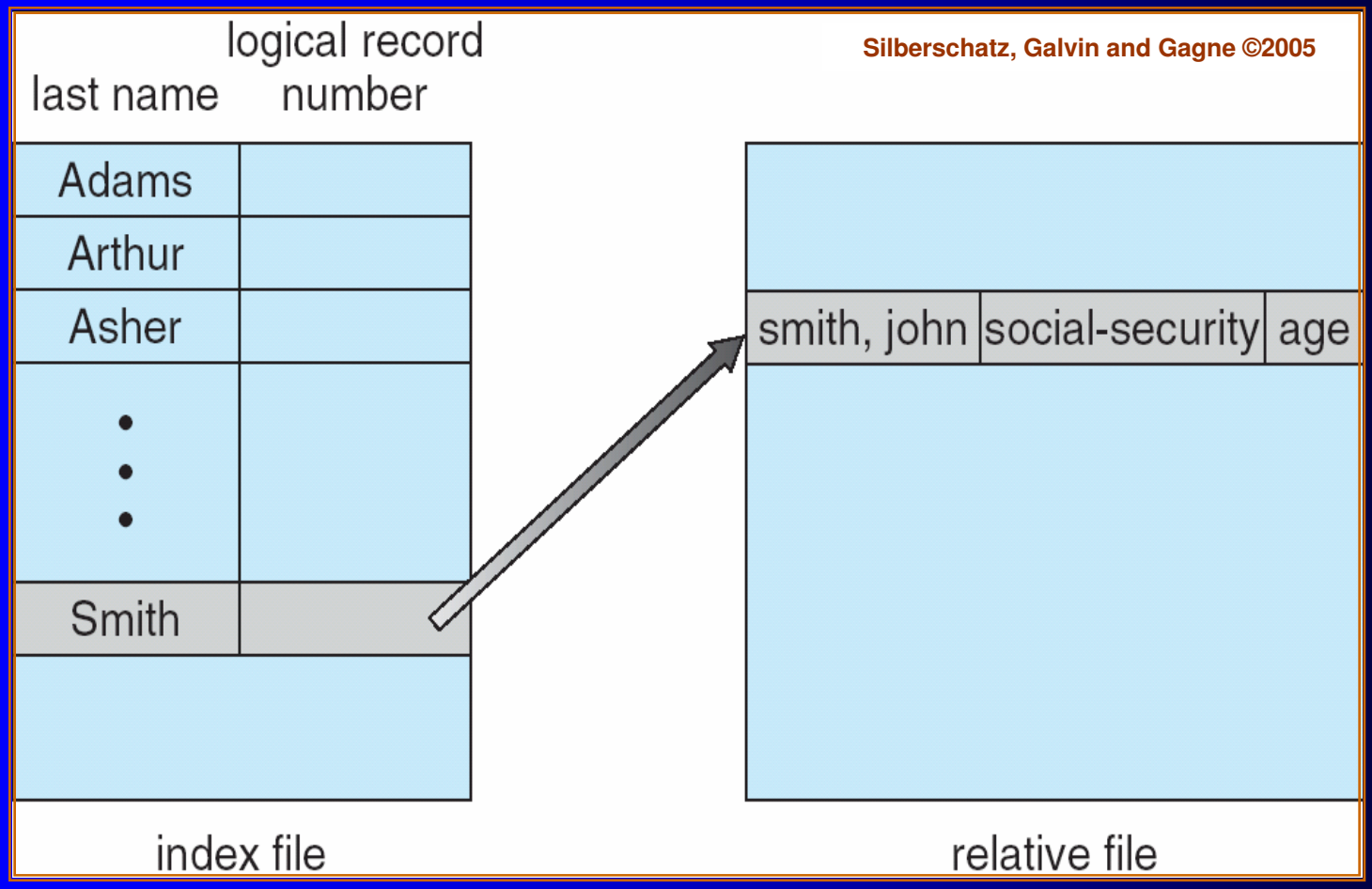
Modalità di accesso – 1

- **Accesso sequenziale**
  - Viene trattato un gruppo di *byte* (oppure un *record*) alla volta
  - Un puntatore indirizza il *record* (o gruppo) corrente e avanza a ogni lettura o scrittura
    - La lettura può avvenire in qualunque posizione del *file*
      - La posizione desiderata deve però essere raggiunta sequenzialmente
        - Come su un nastro
    - La scrittura può avvenire solo in coda al *file* (*Append*)
      - Ovviamente!
  - Sul *file* si può operare solo sequenzialmente
    - Ogni nuova operazione fa ripartire il puntatore dall'inizio

Modalità di accesso – 2

- **Accesso diretto**
  - Opera su *record* di dati posti in posizione **arbitraria** nel *file*
    - Posizione determinata rispetto alla base (*offset* = 0)
- **Accesso indicizzato** (per chiave)
  - Per ogni *file* una tabella di chiavi ordinate contenenti gli *offset* dei rispettivi *record* nel *file*
    - Informazione di navigazione non più nei *record* ma in una struttura a parte ad accesso veloce (*hash*)
      - Principio delle base di dati
  - Ricerca binaria della chiave e poi accesso diretto
  - Sistema **ISAM** (*indexed sequential access method*) di IBM
    - Consente accesso sia indicizzato che sequenziale
    - Tipico delle basi di dati

Modalità di accesso – 3



Accesso indicizzato per chiave

Classificazione

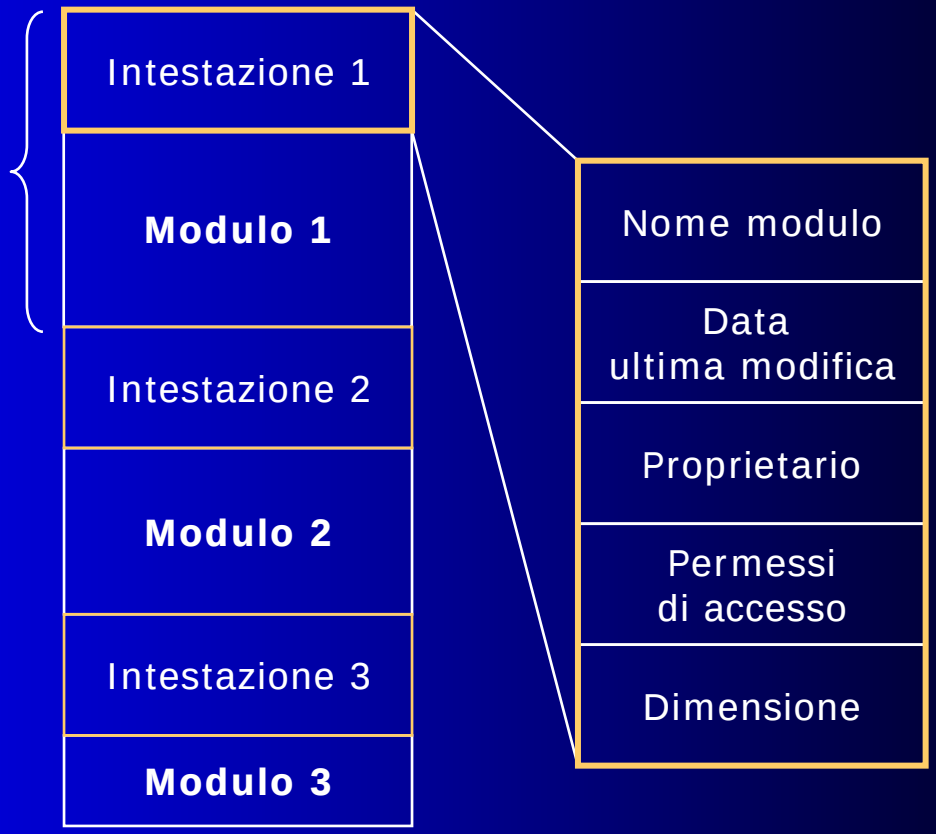
UNIX → GNU/Linux

Windows

- Il FS può trattare diversi **tipi** di *file*
  - Classificazione distinta da quella dell'utente!
  - *File* **regolari** (*regular*)
    - Sui quali l'utente può operare normalmente
      - Contenuto ASCII (testo) o binario (eseguibile)
  - *File* **catalogo** (*directory*)
    - Tramite i quali il FS permette di descrivere l'organizzazione di gruppi di *file*
  - *File* **speciali**
    - Con i quali il FS rappresenta dispositivi con caratteristiche distinte
      - Orientati a carattere (p.es.: terminale)
      - Orientati a blocco (p.es.: disco)

File binari in UNIX e GNU/Linux – 1

Struttura di un *file* eseguibile



Struttura di un *file* archivio
(**tar** : *tape archive*)

File binari in UNIX e GNU/Linux – 2

- Nei primi UNIX i primi 2 *Byte* di un *file* binario contenevano una istruzione per saltare l'intestazione (ampia 8 B) e arrivare direttamente all'eseguibile
  - Poi divenne un valore che designava il formato dell'eseguibile
    - P.es.: dati su pagina separata oppure no
  - Valore creato dal *linker/loader*
- Infine diventato "*magic number*" che specifica il **tipo** dei dati contenuti nel *file*
  - Java bytecode: 0xCAFEBABE (8 B)
  - PostScript: 0x2521 = '%!' (1 B)
  - ZIP: 'PK' (2 B)
  - JPEG: 0xFFD8 (1 B)
  - ...

Operazioni ammesse – 1

- **Creazione**
  - *File* inizialmente vuoto; inizializzazione attributi
- **Apertura**
  - Deve precedere il I uso; predispone le informazioni utili all'accesso
- **Cerca posizione** (*seek*)
  - Solo per accesso casuale
- **Cambia nome**
  - *Rename* (può implicare spostamento nella struttura logica del FS)
- **Distruzione**
  - Rilascio della memoria occupata
- **Chiusura**
  - Rilascio delle strutture di controllo usate per l'accesso e il salvataggio dei dati
- **Lettura. Scrittura**
  - *Read*, *write*, *append*
- **Trova attributi**
- **Modifica attributi**

Azioni più complesse (p.es.: copia) si ottengono tramite combinazione delle operazioni di base

Operazioni ammesse – 2

- **Sessione d'uso di un *file***
  - Si può accedere in uso solo a un *file* già aperto
  - All'apertura del *file* il S/O ne predispone uno specifico strumento di accesso (*handle*)
  - Dopo l'uso il *file* dovrà essere chiuso
  - UNIX (→ GNU/Linux) mantiene una tabella dei *file* aperti a due livelli
    - **Livello I**: informazioni sul *file* comuni a "famiglie" di processi
      - Da *handle* verso attributi, posizione su disco, punto di R/W
    - **Livello II**: dati specifici del particolare processo
      - Con puntatore alla voce corrispondente in tabella di livello I

Esempio d'uso con "chiamate di sistema"

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc,
         char *argv[]){

  FILE *fp;
  char dato;

  if (argc != 2) {
    printf("Nome del file?");
    exit(1);
  }

  // continua …
```

1/2

```c
  if ((fp = fopen(argv[1],"w"))
      == NULL){
   printf("File non aperto.\n");
   exit(1);
  }
   do {
    dato = getchar();
    if (EOF == putc(dato, fp)) {
     printf("Errore di lettura.\n");
    break;
    };
  } while (dato != 'c');
  fclose(fp);
}
```

2/2

File mappati in memoria

- Il S/O può mappare un *file* in memoria virtuale
  - Il *file* continua a risiedere in memoria secondaria
  - All'indirizzo di ogni suo dato corrisponde un indirizzo di memoria virtuale (base + *offset*)
    - Con segmentazione si **potrebbe** avere {*file* = segmento} potendo così usare lo stesso *offset* per entrambi
  - Le operazioni su *file* avvengono in memoria principale
    - Chiamata di indirizzo → *page fault* → caricamento → operazione → rimpiazzo di pagina → salvataggio in memoria secondaria
  - A fine sessione **tutte** le modifiche effettuate in memoria primaria devono essere riportate in memoria secondaria
- Riduce gli accessi a disco ma comporta problemi con la condivisione e con i *file* di enorme dimensione
  - Dove trovare la versione corrente dei dati: RAM o disco?
  - Cosa succede quando *file* > segmento?

Struttura di *directory* – 1

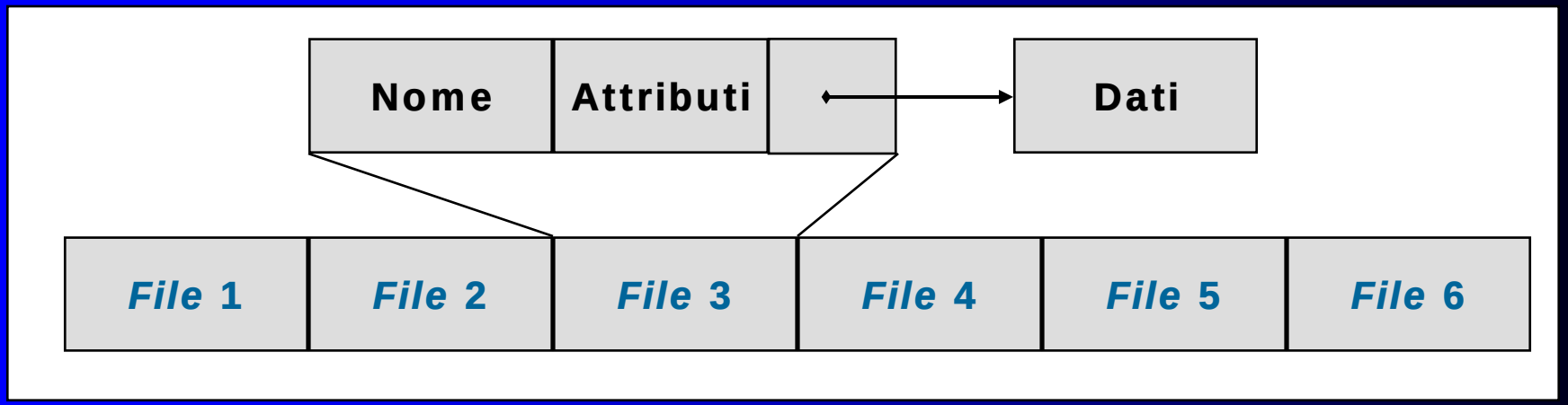
- Ogni FS usa *directory* (catalogo) o *folder* (cartella) per tener traccia dei suoi *file* regolari
- Le *directory* possono essere classificate rispetto all'organizzazione di *file* che esse consentono
  - A livello singolo
  - A due livelli
  - Ad albero
  - A grafo aciclico
  - A grafo generalizzato (ciclico)

Struttura di *directory* – 2

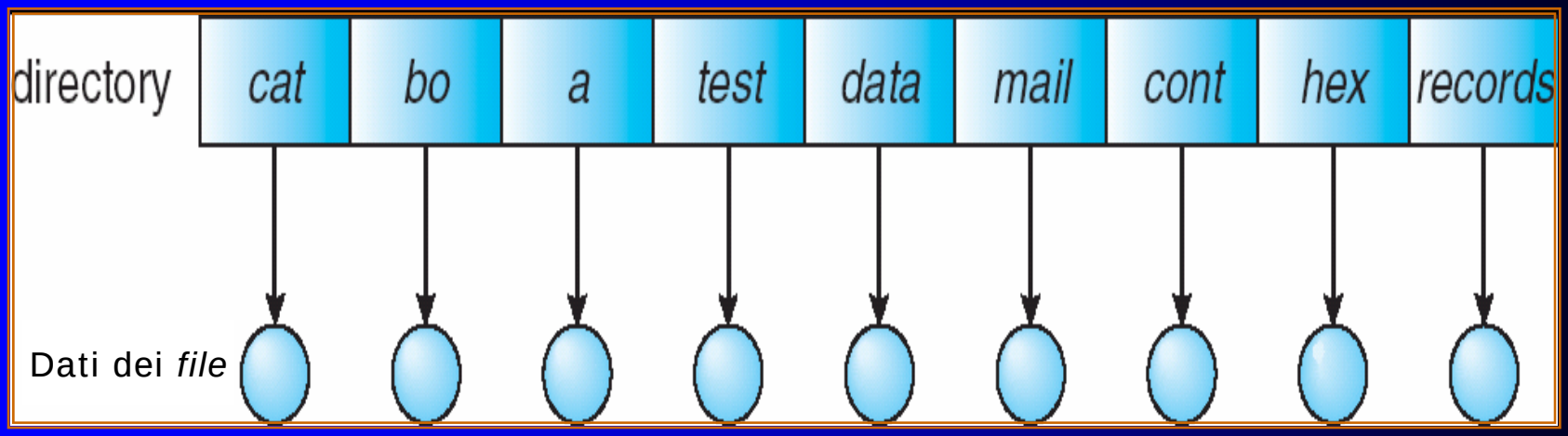
- Requisiti fondamentali a livello utente
  - **Efficienza**
    - Trovarvi un *file* deve essere facile e veloce
  - **Libertà di denominazione**
    - Più utenti devono poter ciascuno usare lo stesso nome per un *file* loro proprio
    - Lo stesso *file* deve poter essere “chiamato” con nomi diversi da utenti diversi
  - **Libertà di raggruppamento**
    - Creare gruppi logici di *file* sulla base di proprietà significative per l’utente

Struttura di *directory* – 3

- ***Directory* a livello singolo**
  - Tutti i *file* sono elencati su un'unica lista lineare
    - Ciascun *file* con il suo proprio nome
    - I nomi dei *file* devono pertanto essere **unici**
  - Semplice da capire e da realizzare
  - Gestione onerosa all'aumentare del numero di *file*



Struttura di *directory* – 4



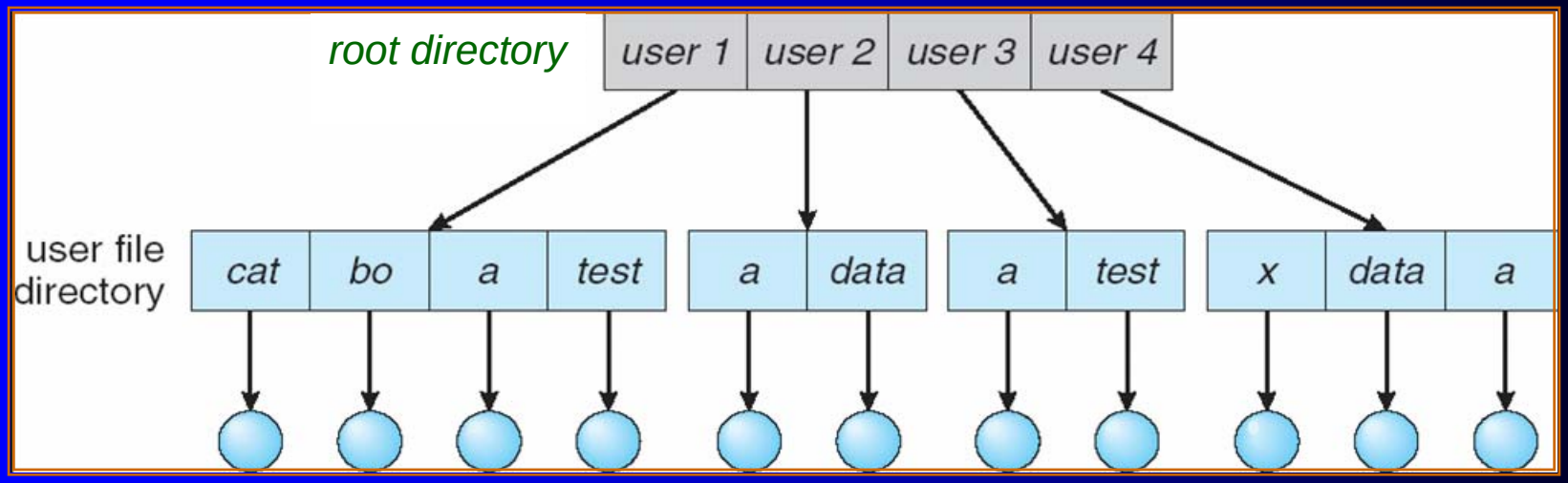
Silberschatz, Galvin and Gagne ©2005

- Requisiti **parzialmente** soddisfatti
  - Efficienza realizzativa (ma non di uso)
- Requisiti **non** soddisfatti
  - Libertà di denominazione
  - Libertà di raggruppamento

Struttura di *directory* – 5

- ***Directory* a due livelli**
  - Una *Root Directory* contiene una *User File Directory* (UFD) per ogni singolo utente di sistema
  - L'utente registrato può vedere **solo** la propria UFD
    - Le UFD di altri solo se esplicitamente autorizzato
    - Buona soluzione per isolare utenti in sistemi multiprogrammati
  - I *file* sono localizzati tramite percorso (*path name*)
  - I programmi di sistema possono essere copiati su tutte le UFD
  - Oppure (meglio!) essere posti in una *directory* di sistema condivisa e lì localizzati mediante **cammini di ricerca** predefiniti (*search path*)

Struttura di *directory* – 6



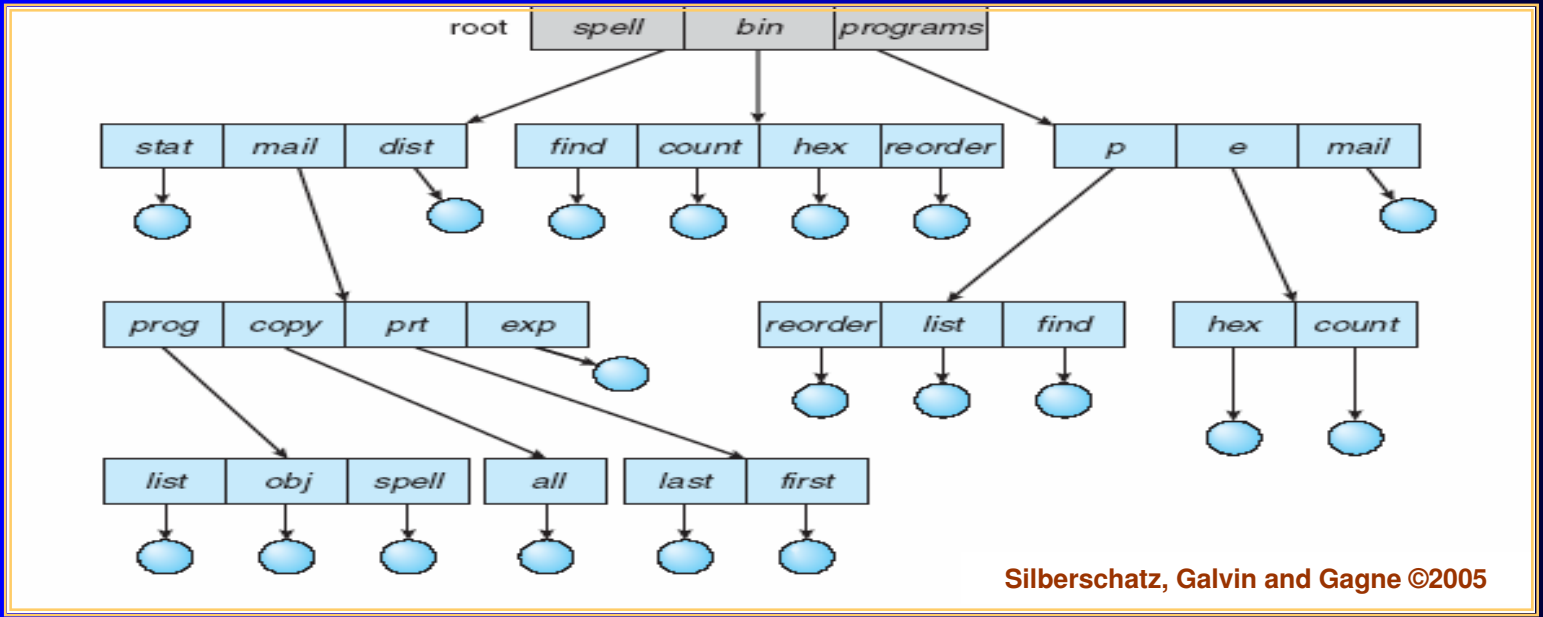
Silberschatz, Galvin and Gagne ©2005

- Requisiti parzialmente soddisfatti
  - Efficienza di ricerca
  - Libertà di denominazione
    - Ma non di riferimenti multipli allo stesso *file* utente
- Requisiti **non** soddisfatti
  - Libertà di raggruppamento

Struttura di *directory* – 7

- ***Directory* ad albero**
  - Numero arbitrario di livelli
  - Il livello superiore viene detto radice (*root*)
  - Ogni *directory* può contenere *file* regolari o *directory* di livello inferiore
  - Ogni utente ha la sua *directory* corrente che può cambiare con comandi di sistema
  - Se non si specifica il cammino (*path*) si assume come riferimento la *directory* corrente
  - Il cammino può essere **assoluto**
    - Espresso rispetto alla radice del FS
  - Oppure **relativo**
    - Rispetto alla posizione corrente

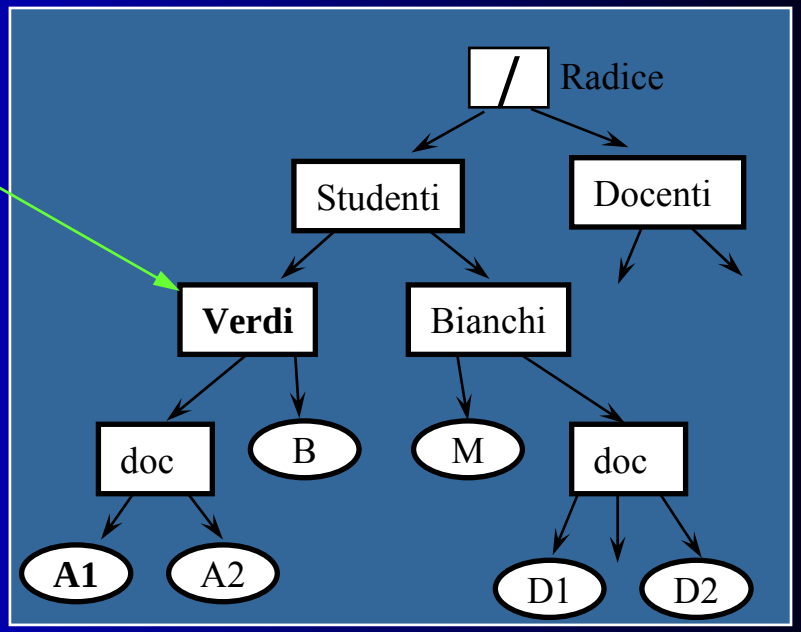
Struttura di *directory* – 8



- Requisiti **parzialmente** soddisfatti
  - Ricerca efficiente
  - Libertà di denominazione
    - Ma non di riferimenti multipli allo stesso *file* utente
  - Libertà di raggruppamento

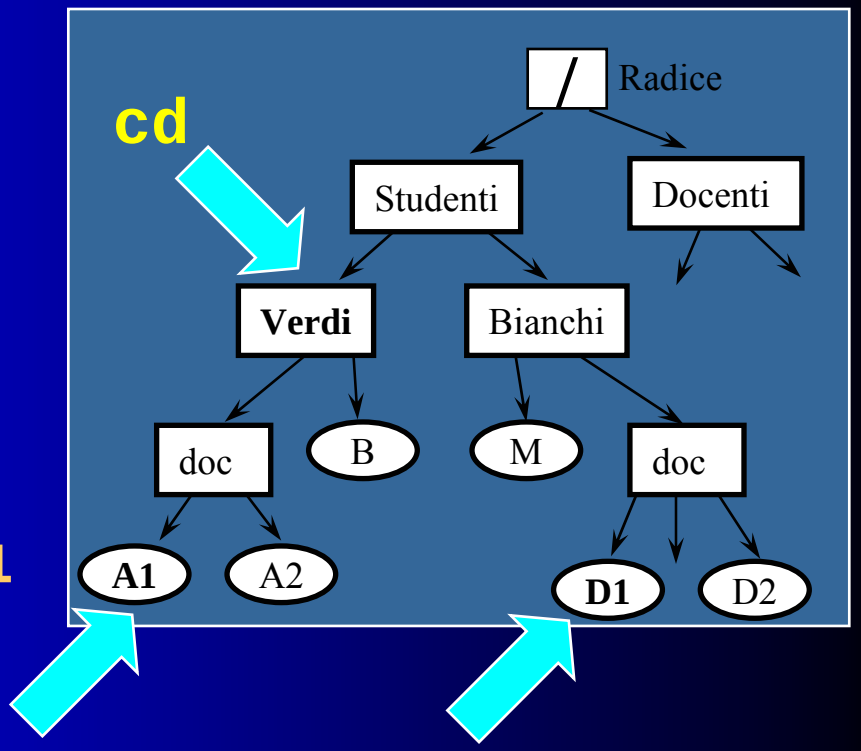
Esempio di *directory* ad albero – 1
(/ per UNIX e GNU/Linux, \ per MS Windows)

- Livello corrente
  - *Directory* **Verdi** = **.**
    - *Dot*
- Livello superiore
  - *Directory* padre = **..**
- Livello inferiore
  - *Directory* figlio = **./**
    - *Slash*



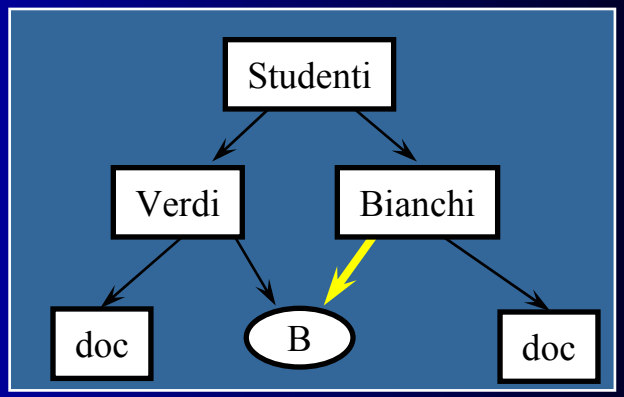
Esempio di *directory* ad albero – 2
(/ per UNIX e GNU/Linux, \ per MS Windows)

- Il *file* **A1** identificato come
  - **./doc/A1**
    - Cammino **relativo** a partire da **cd**
  - **/studenti/Verdi/doc/A1**
    - Cammino **assoluto** a partire dalla radice
- Il *file* **D1** di un altro ramo (purché condiviso da **Verdi**)
  - **../studenti/Bianchi/doc/D1**
    - Relativo
  - **/studenti/Bianchi/doc/D1**
    - Assoluto

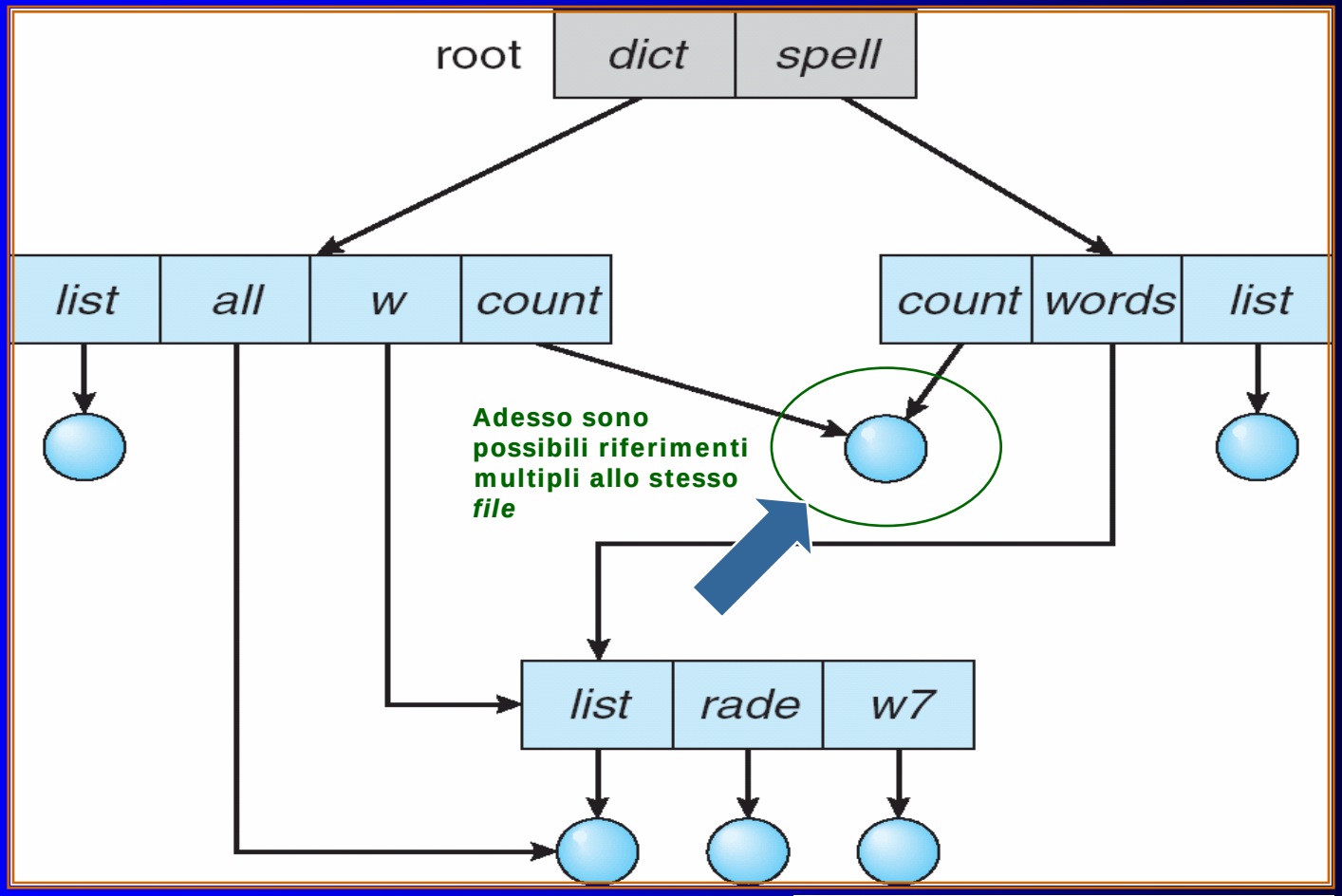


Struttura di *directory* – 9

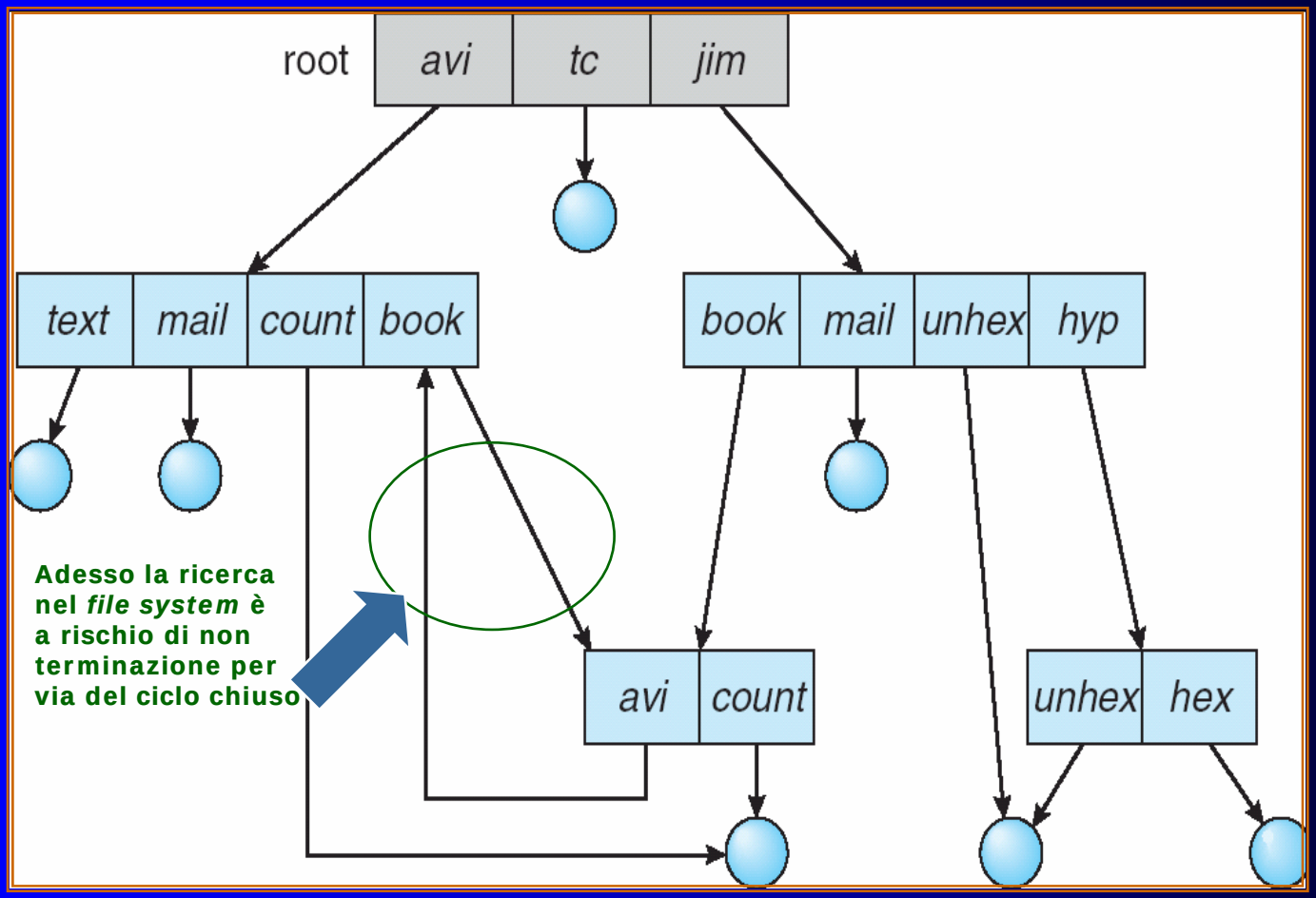
- ***Directory* a grafo**
  - Aciclico oppure ciclico (generalizzato)
  - L'albero diventa grafo quando si consente allo stesso *file* di appartenere simultaneamente a più *directory*
  - UNIX e GNU/Linux utilizzano collegamenti simbolici (***link***) tra il nome reale del *file* e la sua presenza virtuale
  - La forma generalizzata consente collegamenti ciclici e dunque riferimenti circolari
  - Un S/O potrebbe duplicare gli identificatori di accesso al *file* (*handle*) → nomi distinti
    - Questo però rende più difficile assicurare la coerenza del *file*



Struttura a grafo aciclico



Struttura a grafo generalizzato



Struttura di *directory* – 10

- ***Hard link***
  - Un puntatore diretto a un *file* **regolare** viene inserito in una *directory* a esso remota
    - Che deve risiedere nello **stesso** FS del *file*
  - In questo modo esistono **più vie d'accesso distinte** dirette allo stesso *file*
    - Per gestirli correttamente servono "**contatori di riferimento**"
      - Il proprietario del *file* non può più rimuoverlo quando vuole!
  - Rischio di cicli chiusi!
- ***Symbolic (soft) link***
  - Viene creato un *file* **speciale** il cui contenuto è il cammino del *file* originario
    - Chiamato *shortcut* in ambiente MS Windows
    - Il *file* originario può avere qualunque tipo e risiedere ovunque
      - Anche in un FS remoto
  - In questo modo esiste **1 sola via d'accesso** al *file* originario

Operazioni su *directory* GNU/Linux

Azione	Nome comando	Chiamata di sistema
Crea *directory*	`mkdir`	Create
Cancella *directory*	`rmdir`	Delete
Cambia nome a *directory*	`mv`	Rename
Apri, chiudi, leggi *directory*		Opendir, Closedir, Readdir
Crea collegamento a *file*	`ln`	Link
Rimuovi collegamento a *file*	`rm`	Unlink

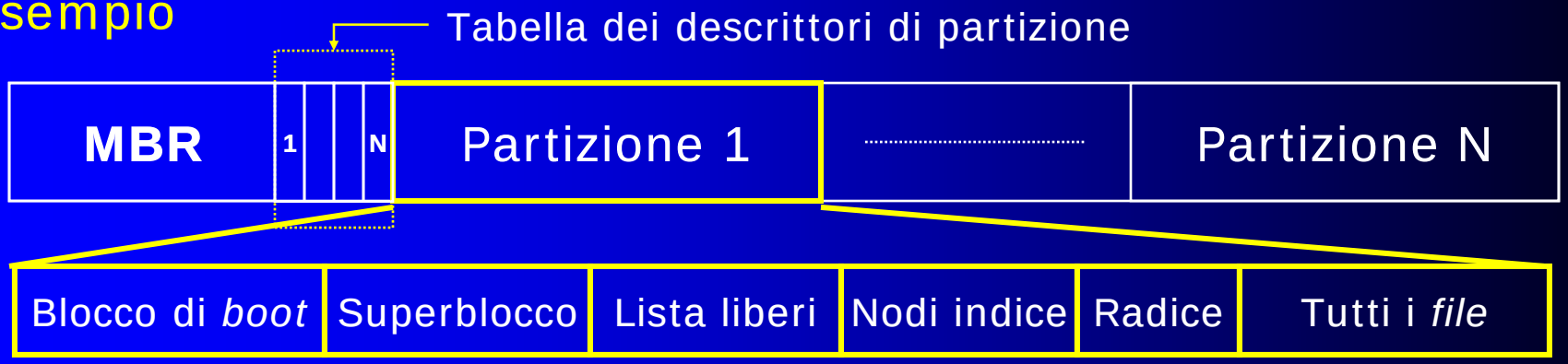
Realizzazione del *file system* – 1

- I *file system* (FS) sono memorizzati su disco
  - I dischi possono essere **partizionati**
  - Ogni partizione può contenere un FS distinto
- Il settore 0 del disco contiene le informazioni di inizializzazione del sistema
  - ***Master Boot Record***
  - L'inizializzazione è eseguita dal BIOS
  - L'**MBR** (ampio 512 B = 1 **settore**) specifica le partizioni presenti e identifica quella attiva
  - Il primo blocco di informazione di ogni partizione contiene le sue specifiche informazioni di inizializzazione (***boot block***)

Realizzazione del *file system* – 2

- L'unità informativa su disco è il **settore**
- I dischi vengono però letti e scritti a **blocchi** (*cluster* per Microsoft!) per velocizzare le operazioni
  - 1 blocco = N settori ($N \geq 1$)
  - Rischio consapevole di frammentazione interna
- La struttura interna di partizione è specifica del FS

Esempio



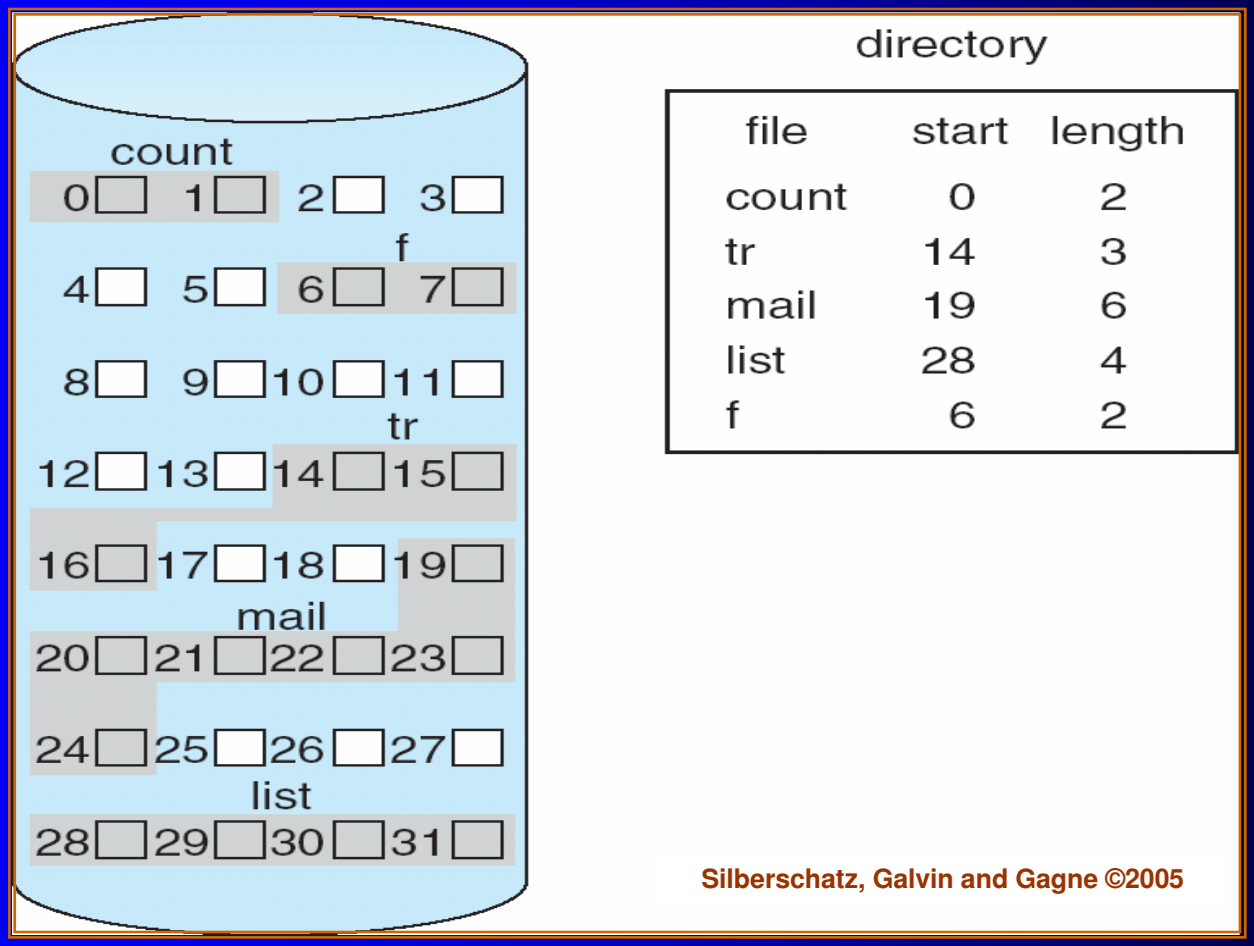
Realizzazione dei *file* – 1

- A **livello fisico** un *file* è un insieme di blocchi di disco
  - Occorre decidere quali blocchi assegnare a quale *file* e come tenerne traccia
- 3 strategie di allocazione di blocchi a *file*
  - Allocazione contigua
  - Allocazione a lista concatenata (*linked list*)
  - Allocazione a lista indicizzata

Realizzazione dei *file* – 2

- **Allocazione contigua**
  - Cerca di memorizzare i *file* su blocchi **consecutivi**
  - Ogni *file* è descritto dall'indirizzo del suo primo blocco e dal numero di blocchi utilizzati
  - Consente sia accesso sequenziale che diretto
  - Un *file* può essere letto e scritto con un solo accesso al disco
    - Ideale per CD-ROM e DVD
  - Ogni modifica di *file* comporta il rischio di frammentazione esterna
    - Ricompattazione periodica molto costosa
    - L'alternativa richiede l'utilizzo dei gruppi di blocchi liberi
      - Mantenere la lista dei blocchi liberi e la loro dimensione
        - Possibile ma oneroso
      - Conoscere in anticipo la dimensione massima dei nuovi *file*
        - Stima difficile e rischiosa

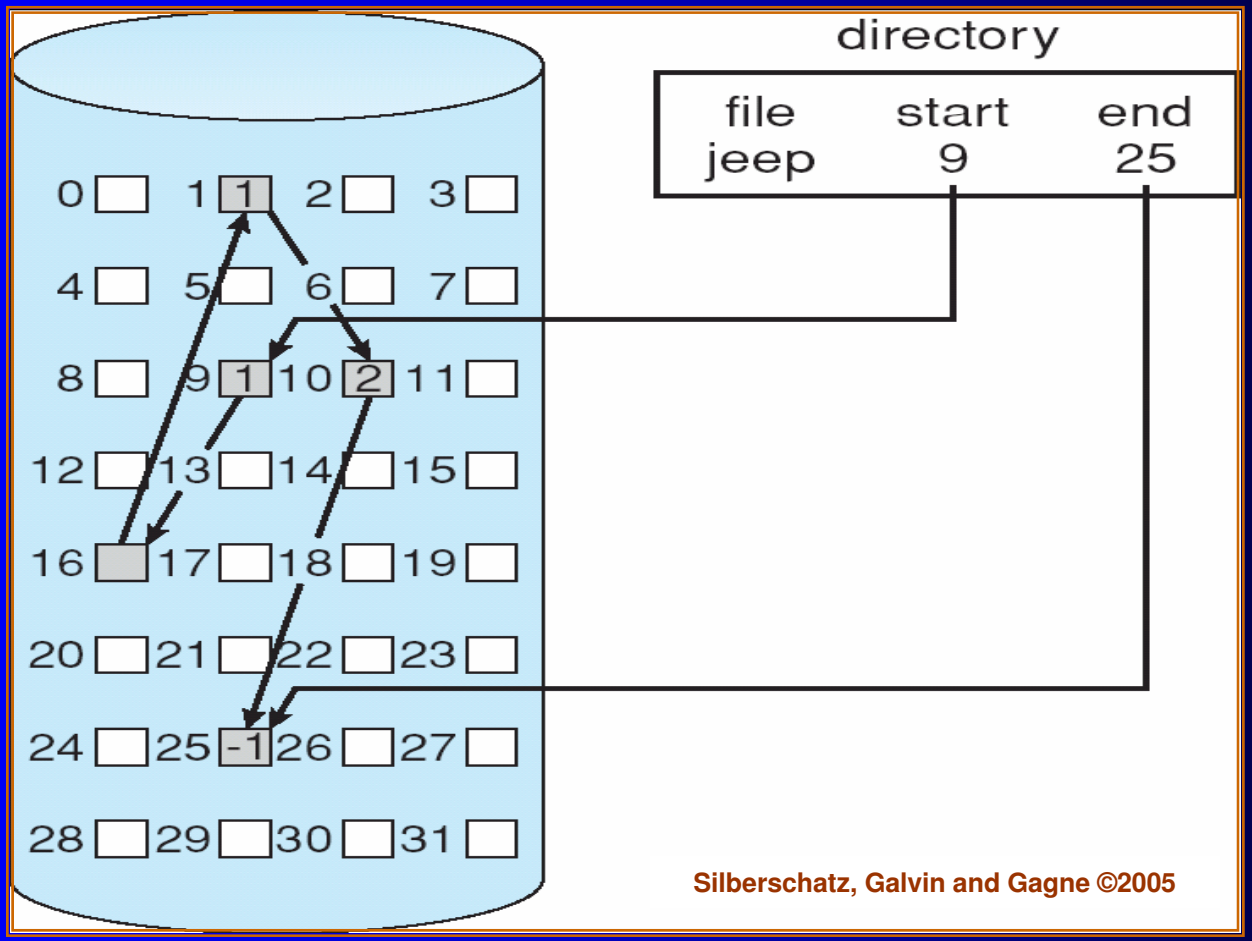
Allocazione contigua



Realizzazione dei *file* – 3

- **Allocazione a lista concatenata**
  - *File* come lista concatenata di blocchi
  - *File* identificato dal puntatore al suo primo blocco
    - Per alcuni S/O anche dal puntatore all'ultimo blocco del *file*
  - Ciascun blocco di *file* deve contenere il puntatore al blocco successivo (o un marcatore di fine lista)
    - Questo sottrae spazio ai dati
  - L'accesso sequenziale resta semplice da realizzare
    - Ma per *file* grandi può richiedere molte operazioni su disco
  - Accesso diretto molto più complesso e oneroso
    - Raggiungere qualunque posizione logica costa molte operazioni su disco
  - Un solo blocco guasto corrompe l'intero *file*

Allocazione a lista concatenata



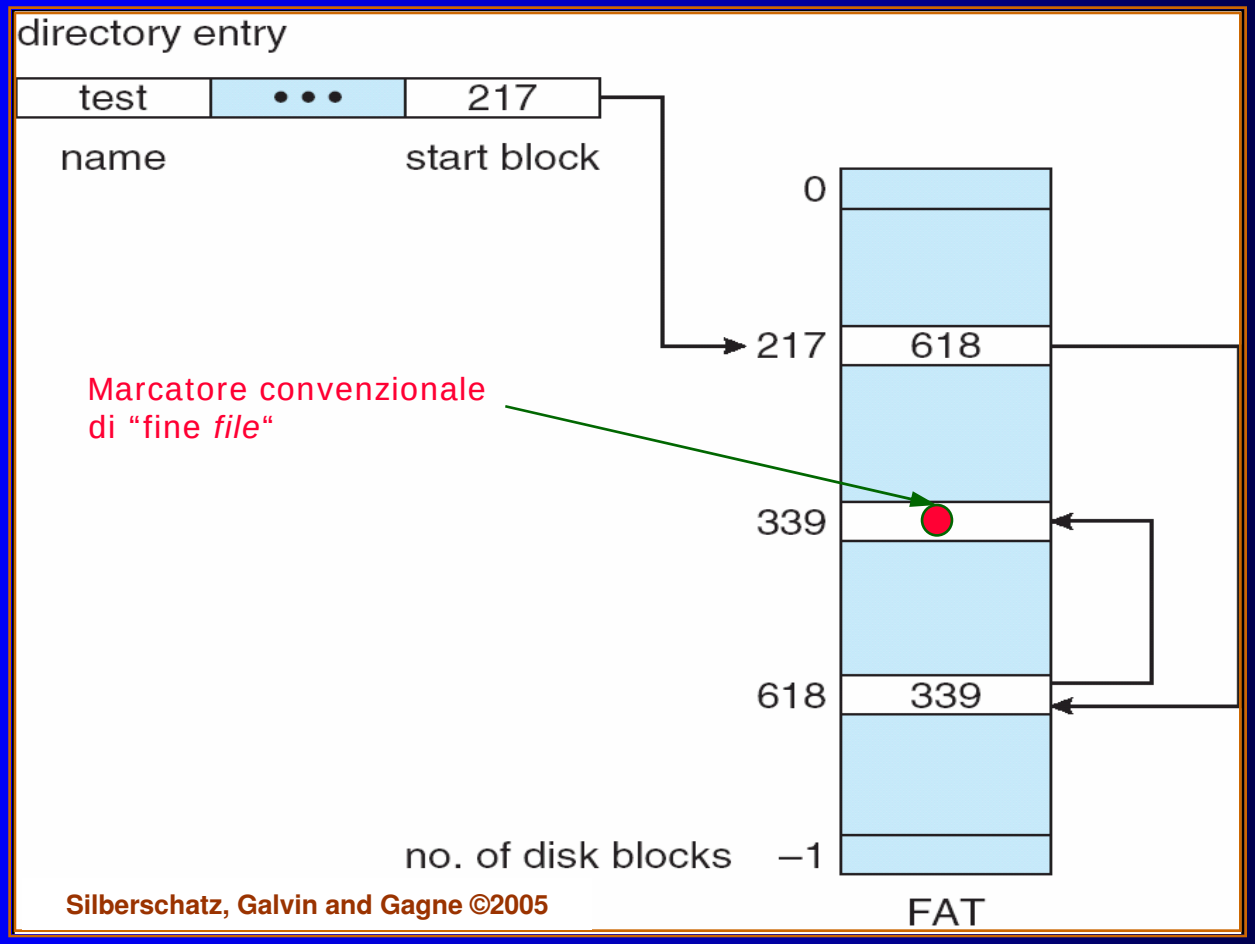
Realizzazione dei *file* – 4

- **Allocazione a lista indicizzata**
  - I puntatori ai blocchi sono memorizzati in strutture dedicate
    - Ciascun blocco contiene **solo dati**
  - Il *file* è descritto dall'insieme dei puntatori ai suoi dati
  - 2 strategie di organizzazione
    - Forma tabulare (**FAT**, *File Allocation Table*)
    - Forma indicizzata (nodo indice, *i-node*)
  - Non causa frammentazione esterna
  - Consente accesso sequenziale e diretto
  - La dimensione massima di ogni nuovo *file* non deve essere nota a priori
    - Il *file* può crescere a piacere

Allocazione a lista indicizzata – 1

- ***File Allocation Table***
  - La scelta progettuale di *MS-DOS*
    - Base del FS storico di *MS Windows*
- **FAT** = tabella ordinata di puntatori
  - Un puntatore $\forall$ blocco (*cluster*) del disco
  - La tabella cresce con l'ampiezza della partizione
- La porzione di FAT relativa ai *file* in uso **deve** sempre risiedere interamente in RAM
- Consente accesso diretto ai dati
- Un *file* è una catena di indici

Struttura **FAT**



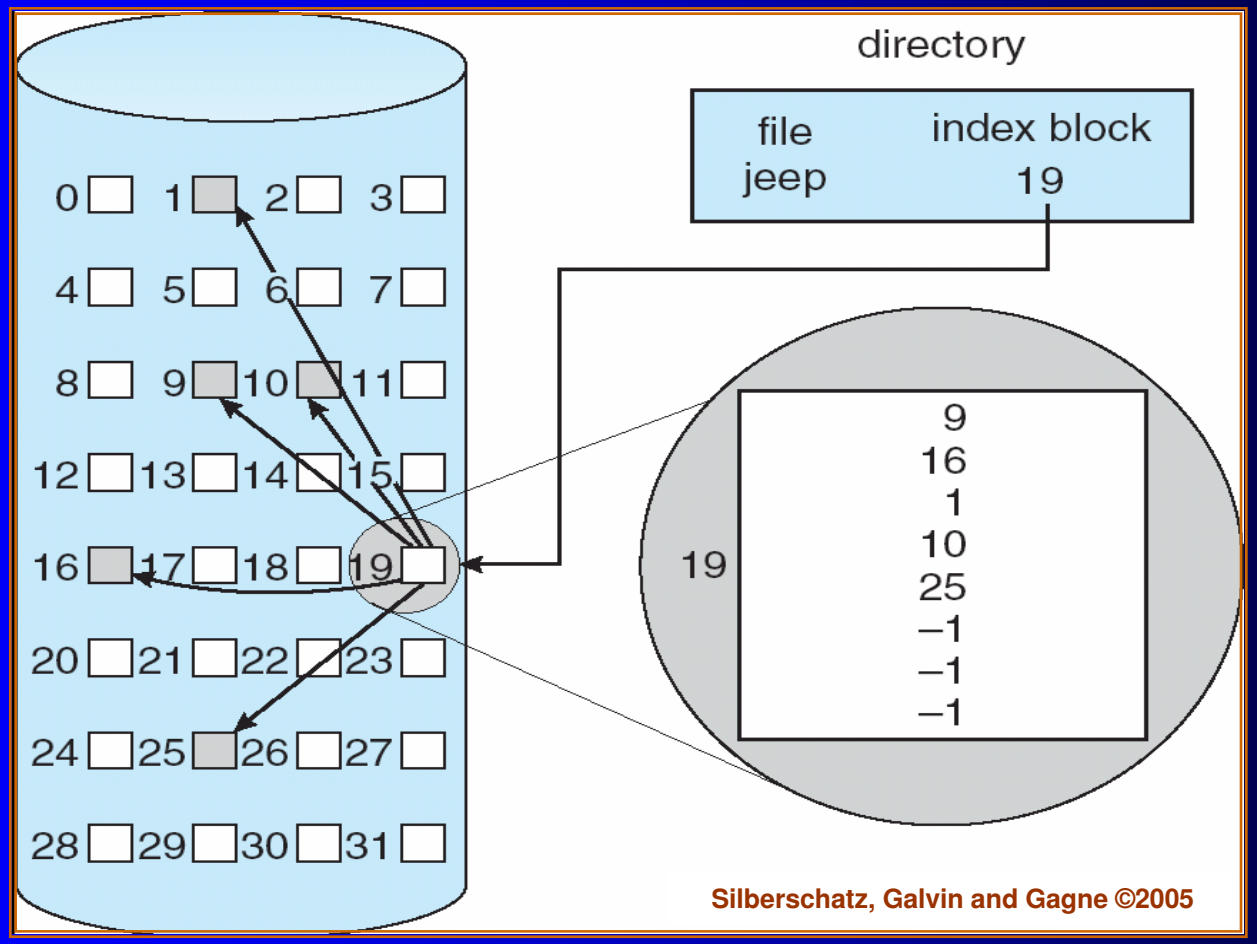
Allocazione a lista indicizzata – 2

- **Nodi indice** (UNIX $\rightarrow$ GNU/Linux)
  - Una struttura indice (*i-node*) $\forall$ *file* con gli attributi del *file* e i puntatori ai suoi blocchi
    - L'*i-node* è contenuto in un blocco dedicato
  - In RAM una tabella di *i-node* per i soli *file* in uso
    - La dimensione massima di tabella dipende dal massimo numeri di *file* apribili simultaneamente
      - Non più dalla capacità della partizione
  - Un *i-node* contiene un numero limitato di puntatori a blocchi
    - Quale soluzione per *file* composti da un numero maggiore di blocchi?

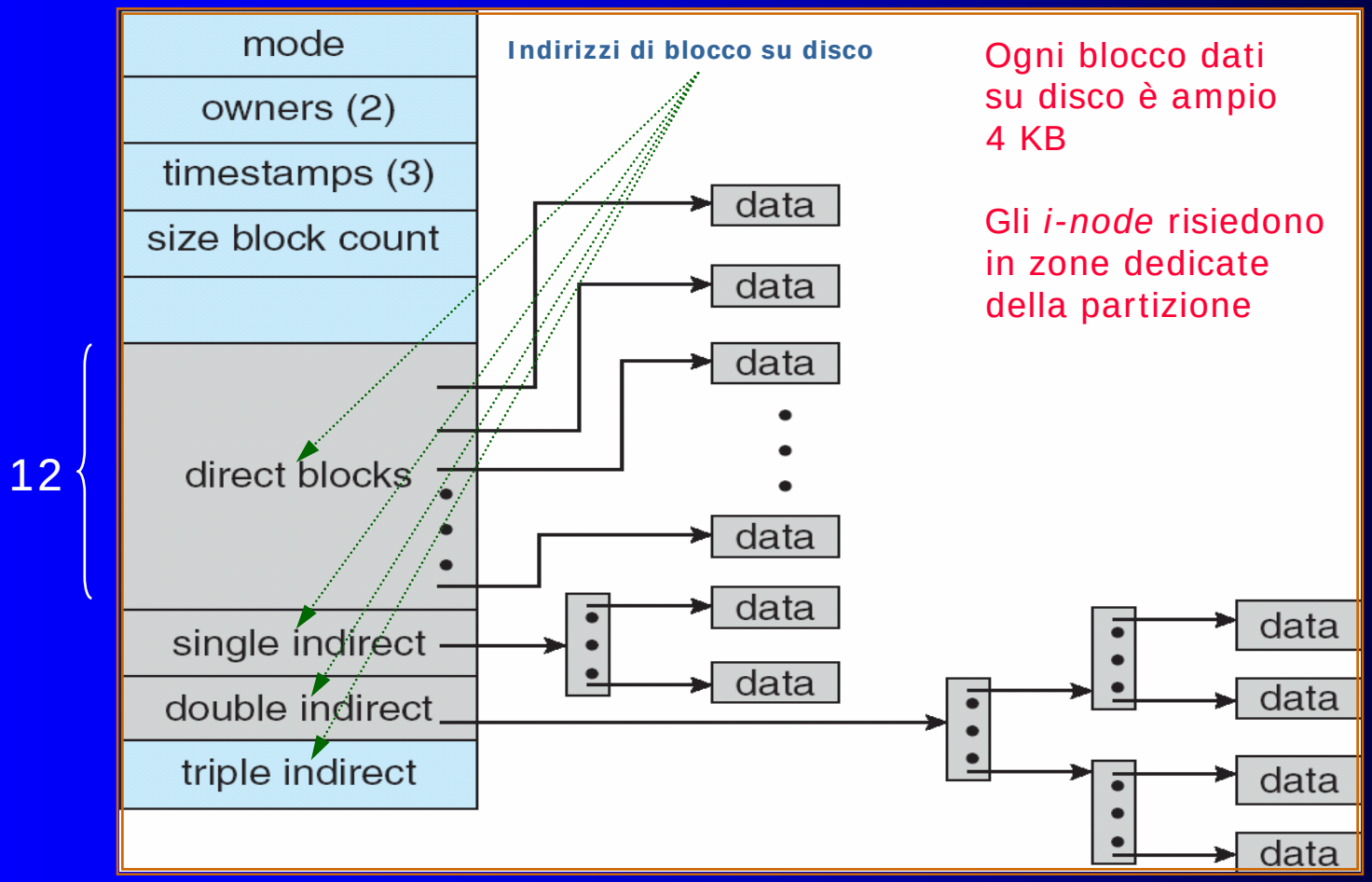
Allocazione a lista indicizzata – 3

- **Nodi indice** (UNIX → GNU/Linux)
  - *File* di piccola dimensione
    - Gli indirizzi dei blocchi dei dati sono ampiamente contenuti in un singolo *i-node*
      - Tipicamente con un po' di frammentazione interna
  - *File* di media dimensione
    - Un campo dell'*i-node* punta a un nuovo blocco *i-node*
  - *File* di grandi dimensioni
    - Un campo dell'*i-node* principale punta a un livello di blocchi (*i-node*) intermedi che a loro volta puntano ai blocchi dei dati
  - Per *file* di dimensioni ancora maggior basta aggiungere un ulteriore livello di indirezione

Struttura a nodi indice



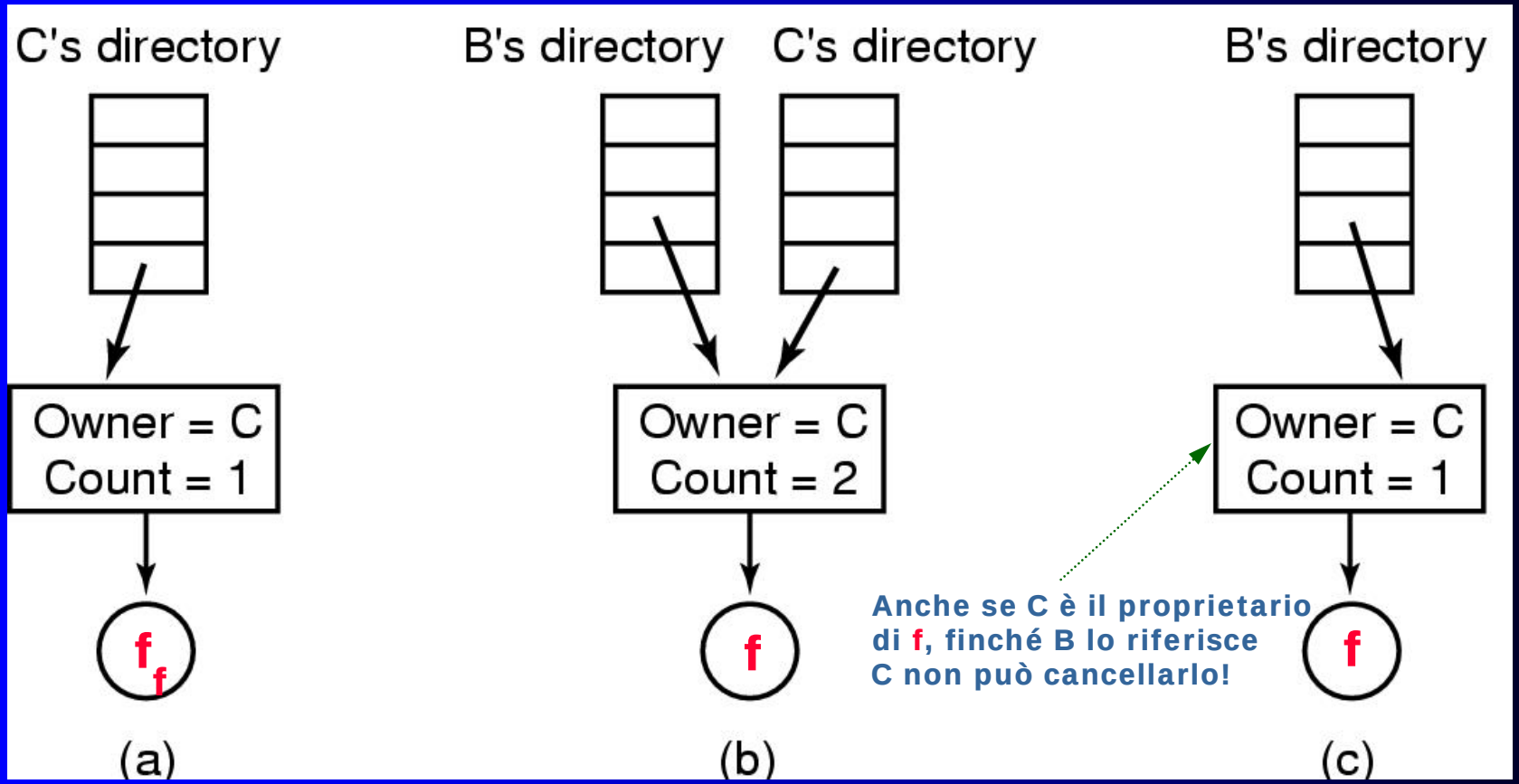
FS in UNIX v7


mode
owners (2)
timestamps (3)
size block count
12
direct blocks
single indirect
double indirect
triple indirect
Indirizzi di blocco su disco
data
data
data
data
data
data
data
data
data
Ogni blocco dati su disco è ampio 4 KB
Gli *i-node* risiedono in zone dedicate della partizione
Silberschatz, Galvin and Gagne ©2005

Realizzazione dei *file* – 5

- **Gestione dei *file* condivisi**
  - Come preservarne la consistenza senza costi eccessivi
    - **Non** porre blocchi di dati nella *directory* di residenza del *file*
    - ∀ *file* condiviso porre nella *directory* remota un ***symbolic link*** verso il *file* originale
      - Esiste così **1 solo** nodo indice (*i-node*) del *file* originale
      - L'accesso condiviso avviene tramite cammino sul FS
    - Altrimenti si può porre nella *directory* remota il puntatore diretto (***hard link***) al nodo indice (*i-node*) del *file* originale
      - Più possessori di descrittori dello stesso *file* condiviso
      - Un solo proprietario effettivo del *file* condiviso
      - Il *file* condiviso **non può** più essere distrutto fin quando esistano suoi descrittori remoti anche se il suo proprietario avesse inteso cancellarlo

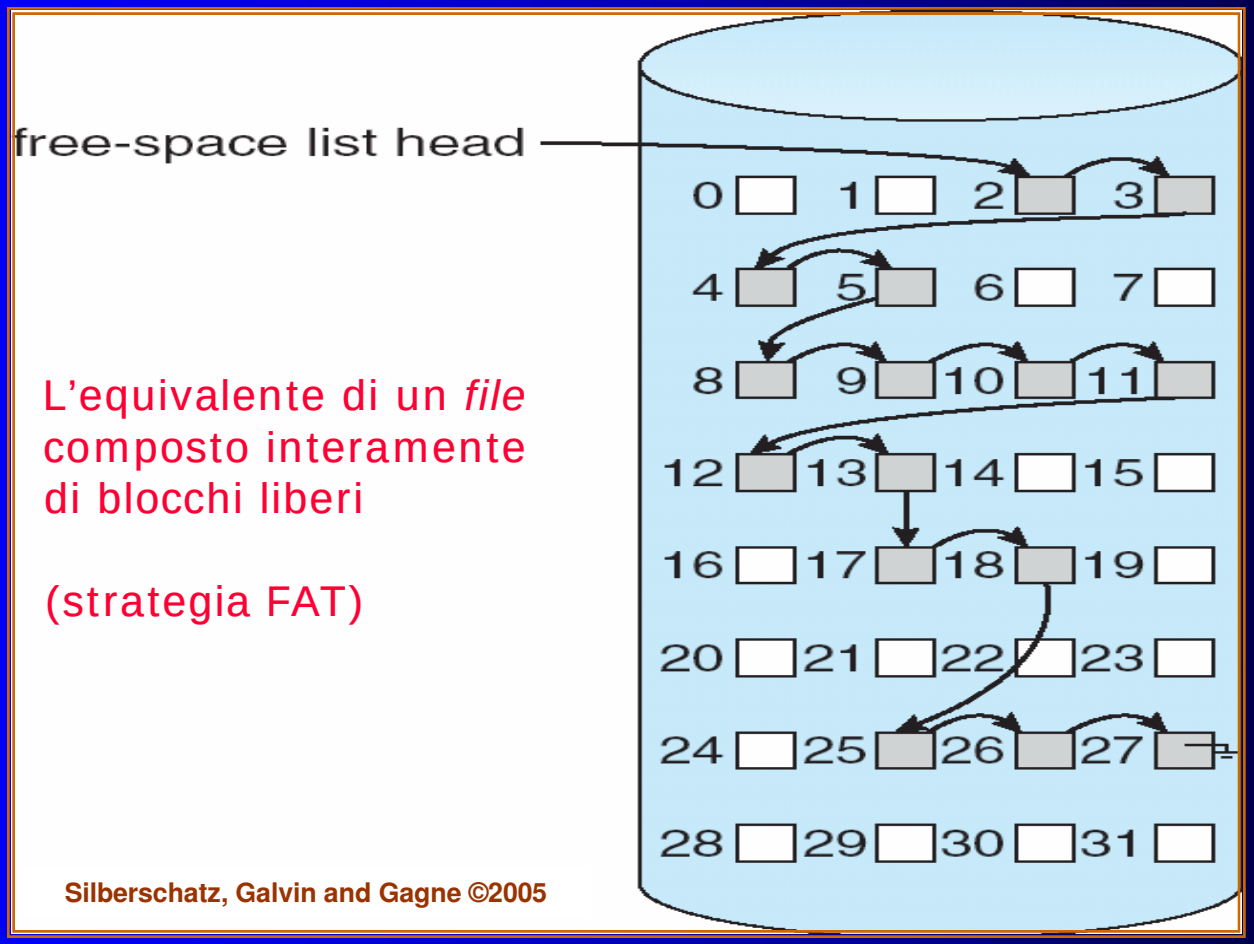
Gestione della condivisione



Realizzazione dei *file* – 6

- **Gestione dei blocchi liberi**
  - **Vettore di *bit*** (*bitmap*) dove ogni *bit* indica lo stato del corrispondente blocco
    - 0 = libero
    - 1 = occupato
  - **Lista concatenata** di blocchi sfruttando i campi puntatore al successivo
    - Gli indici di blocco liberi sono nella FAT e rappresentano un *file* fittizio
  - Oppure riempiendo blocchi dedicati con gli indici corrispondenti
    - Un blocco da 1 KB può contenere fino a 256 indici da 4 B ciascuno

Lista concatenata dei blocchi liberi



Realizzazione delle *directory* – 1

- Per ogni suo *file* la *directory* specifica
  - Nome
  - Collocazione
  - Attributi
- *File* e *directory* risiedono in aree logiche **distinte**
- Conviene minimizzare la complessità gestionale della struttura interna di *directory*
  - Meglio una struttura a lunghezza **fissa**
    - Per quanto il suo contenuto sia di ampiezza variabile!
  - `[Nome + attributi]` oppure
  - `[Nome + puntatore a nodo indice con attributi]`
    - Frammentazione interna trascurabile per nomi di *file* fino a 8 caratteri + 3 di estensione
    - Problema più serio per nomi lunghi

Realizzazione delle *directory* – 2

- La ricerca di un *file* correla il nome (stringa ASCII) alle informazioni necessarie all'accesso
  - Nome e *directory* di appartenenza del *file* sono determinati dal percorso indicato dalla richiesta
- La ricerca **lineare** in *directory* è di realizzazione facile ma di esecuzione onerosa
- La ricerca mediante tabelle *hash* è più complessa ma più veloce

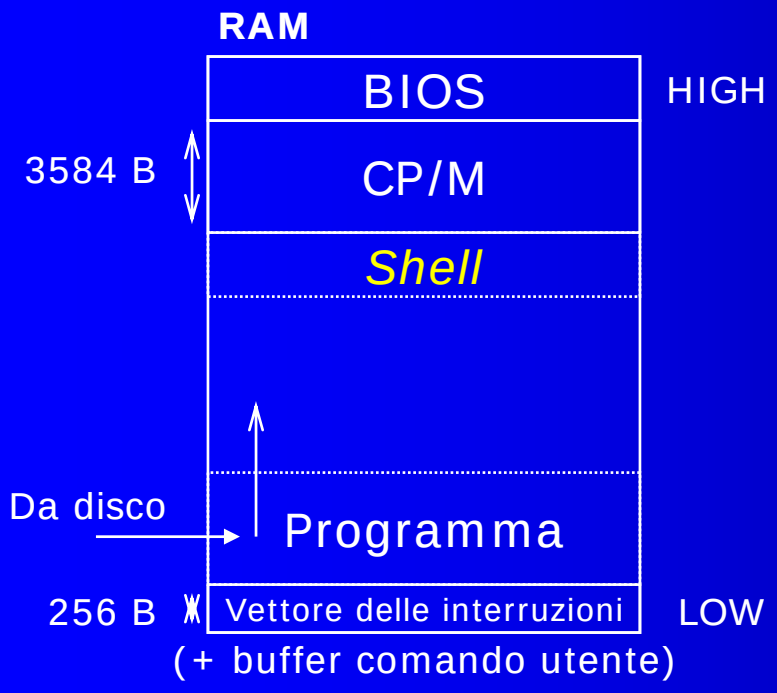
  F(nome) = posizione in tabella $\rightarrow$ puntatore al *file*
- Si può anche creare in RAM una *cache software* di supporto alla ricerca

Cenni storici

- CP/M (1973-1981)
- MS-DOS & Windows 95 (1981 $\rightarrow$ 1997)
- Windows 98 (1998-1999)

- UNIX v7 (1979)

CP/M (*Control Program for Microcomputers*)



- BIOS minimo
  - Massima portabilità
- Sistema multiprogrammato
  - Ogni utente vede solo i propri *file*
- *Directory* singola con dati a struttura fissa
  - In RAM solo quando serve
- *Bitmap* per blocchi liberi memorizzata in RAM
  - Distrutta a fine esecuzione
- Nome *file* limitato a 8 + 3 caratteri
  - Dimensione inizialmente limitata a 16 blocchi da 1 KB
    - Puntati direttamente dalla *directory*

MS-DOS (*Microsoft Disk Operating System*)

- **Non** multiprogrammato
  - Ogni utente vede **tutto** il FS
- FS **gerarchico** senza limite di profondità e **senza condivisione**
  - Fino a 4 partizioni per disco (C: D: E: F:)
- *Directory* a lunghezza variabile con *entry* di 32 B
  - Nomi di *file* a 8+3 caratteri (normalizzati a maiuscolo)
- Allocazione *file* a lista (FAT)
  - **FAT-X** per **X** = numero di *bit* per indirizzo di blocco ($12 \leq$ **X** <32)
  - Blocchi di dimensione multipla di 512 B (1 settore)
  - **FAT-16** *: File* e partizione limitati a 2 GB
    - 2^{16} = 64K (puntatori a) blocchi di 32 KB ciascuno = 2 GB
  - **FAT-32** : blocchi da 4 ÷ 32 KB e indirizzi da 28 *bit* (!)
    - Perché 2 TB è il limite **intrinseco** di capacità per partizione
      - 2^{32} settori (*cluster*) da 512 B = $2^2 \times 2^{30} \times 2^9$ B = 2^{41} B = 2 TB
      - 2^{28} blocchi da **8** KB = $2^8 \times 2^{20} \times 2^3 \times 2^{10}$ B = 2^{41} B = 2 TB

Il FS in MS-DOS 7.0

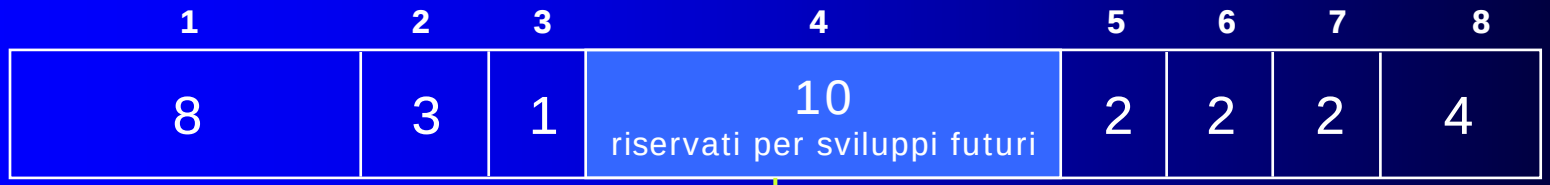
Non più di FAT-16!

Struttura di *directory entry* (32 B)

1.	Nome *file*	8 B	
2.	Estensione *file*	3 B	
3.	Attributi	1 B	
4.	Riservati	10 B	
5.	Ora modifica	2 B	
6.	Data modifica	2 B	
7.	Indice I blocco	2 B	
8.	Dimensione	4 B	

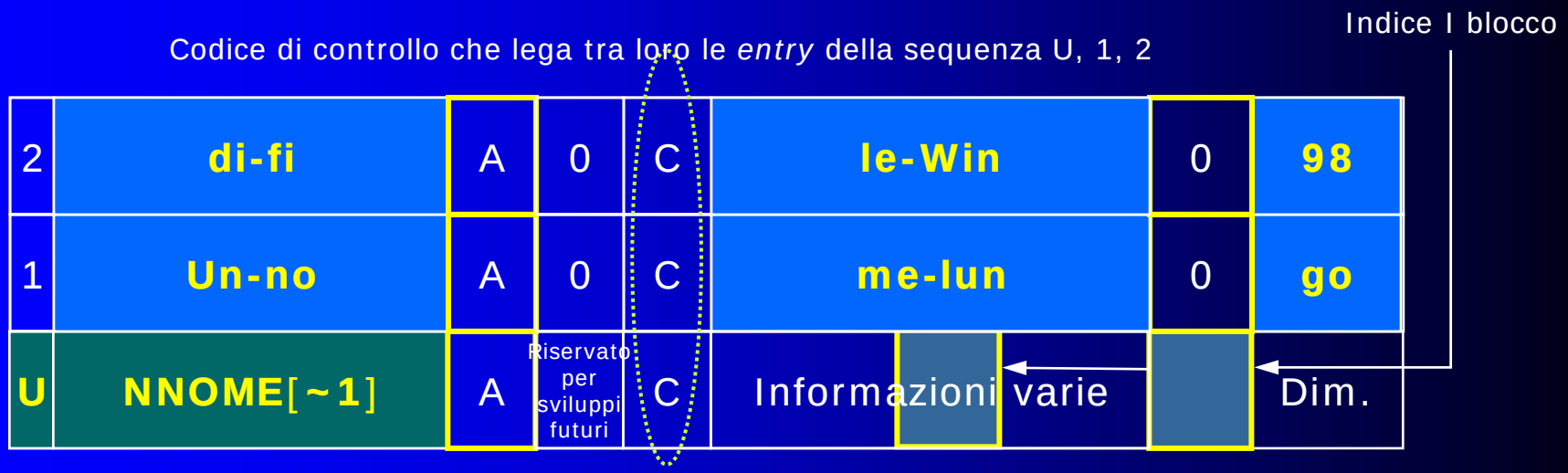
(unsigned)
5 *bit* × ore [0-23]
6 *bit* × minuti [0-59]
5 *bit* × ~2 secondi [0-29]

(unsigned)
7 *bit* × anno+1980 [-2107]
4 *bit* × mese [1-12]
5 *bit* × giorno [1-31]



Usato per Windows 98 (FAT-32, orario accurato, nomi *file* lunghi e *case sensitive*)

Il FS in MS-DOS 7.1 – 1



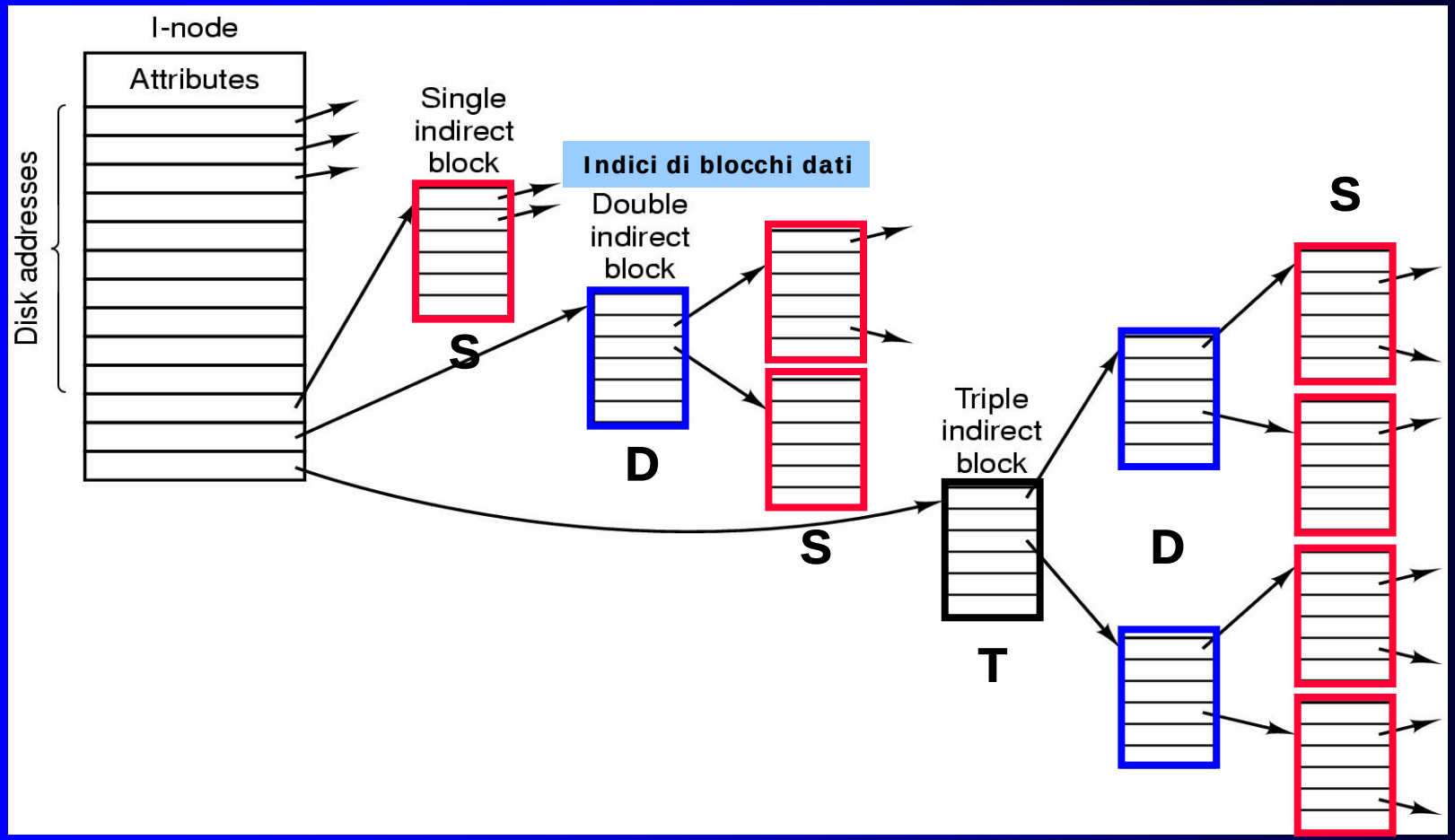
Il FS in MS-DOS 7.1 – 2

- L'utilizzo dello spazio riservato (10 B) ha permesso importanti miglioramenti nella espressività della *directory*
  - Nomi lunghi
    - *Case sensitive* e con estensioni variabili
  - Indice di blocco su 4 B
  - Attributi più significativi
- Al costo di maggior complessità nella sua gestione *software*

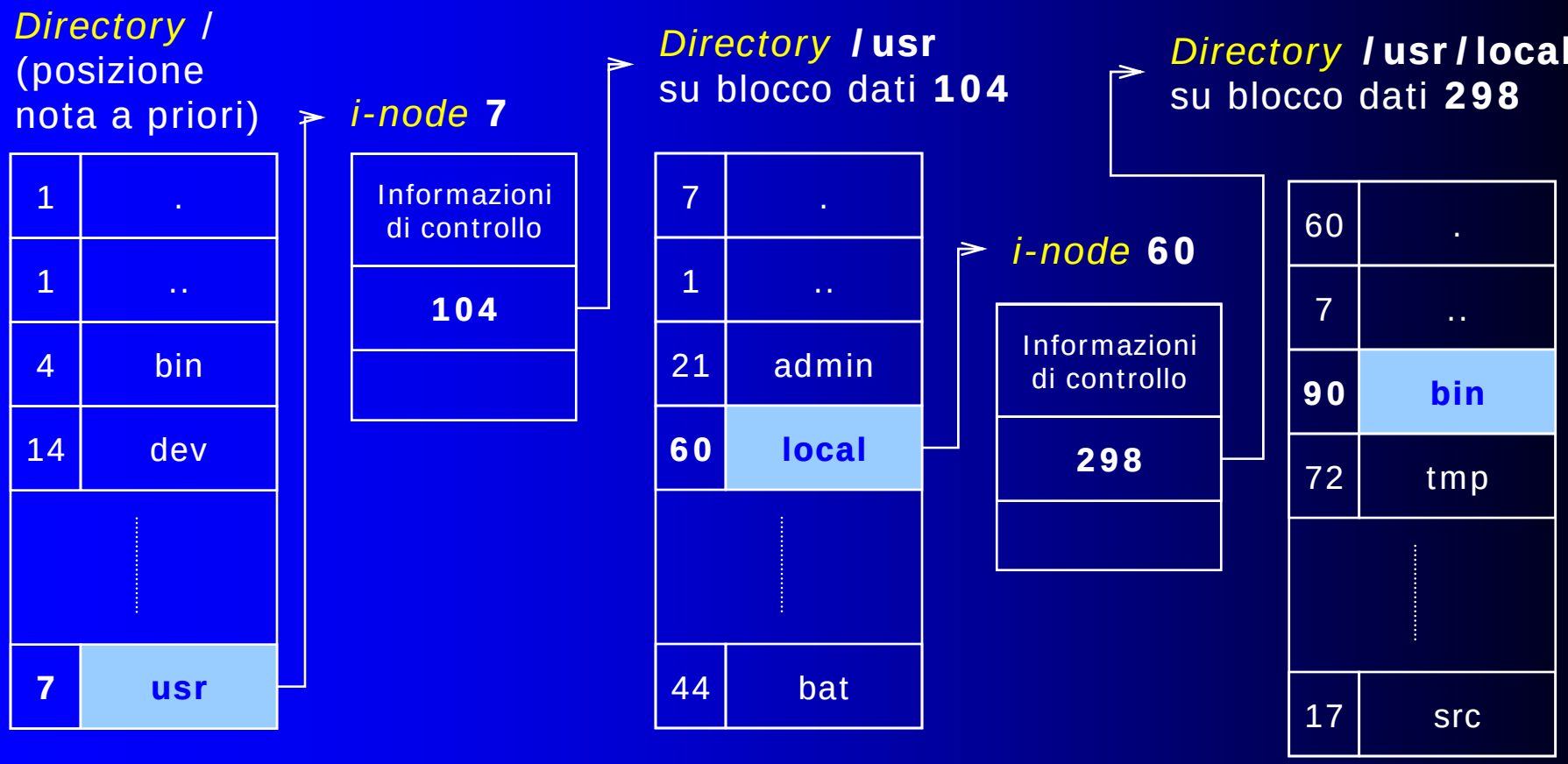
Il FS di **UNIX** v7 – 1

- Concepito e realizzato tra il 1969 e il 1979 da Ken Thompson e Dennis Ritchie
  - Struttura ad albero con radice e condivisione di *file*
    - Grafo **aciclico**
  - Nomi di *file* fino a 14 caratteri ASCII (escluso /)
  - *Directory* contiene nome *file* (14 B) e puntatore (2 B) al suo *i-node* descrittore
    - Max 2^{16} *i-node* distinti
      - Max 64 K *file* per FS
  - L'*i-node* (64 B) contiene gli attributi del *file*
    - Incluso il contatore di *directory* che puntano al *file* tramite un *link* di tipo *hard*
      - Se contatore = 0, il nodo e i blocchi del *file* diventano liberi

Il FS di **UNIX** v7 – 2



Il FS di **UNIX** v7 – 3



Esecuzione parziale del comando “cd /usr/local/bin/”

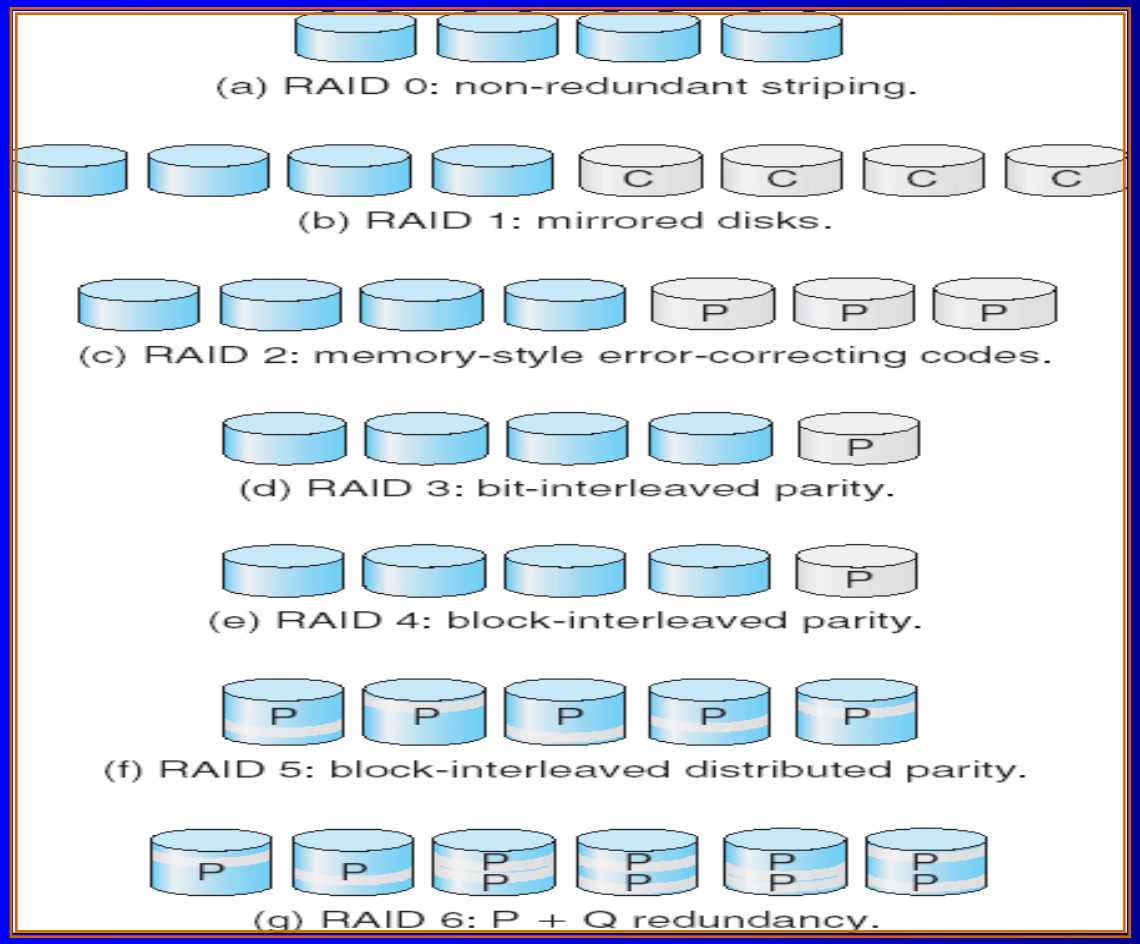
File system su CD-ROM

- **ISO 9660**
  - Supporta fino a $2^{16}-1$ dischi partizionabili
  - Dimensione di blocco 2 ÷ 8 KB
  - *Directory* a struttura variabile internamente ordinate alfabeticamente
  - FS limitato a 8 livelli di annidamento
    - E nomi di *file* "corti"
- **Rock Ridge**
  - Estensione di ISO 9660 definita dal mondo UNIX per meglio rappresentare le caratteristiche del proprio FS
    - Iniziativa resa obsoleta dall'avvento di Joliet
- **Joliet**
  - Estensione definita da Microsoft per lo stesso motivo
    - Nomi "lunghi" e annidamento profondo

Integrità del *File System* – 1

- **Gestione dei blocchi danneggiati**
  - Via *hardware*
    - Creando e mantenendo in un settore del disco un elenco di blocchi danneggiati e dei loro sostituti
  - Via *software*
    - Ricorrendo a un falso *file* che occupi tutti i blocchi danneggiati
- **Salvataggio del FS**
  - Su nastro
    - Tempi lunghi, anche per incrementi
  - Su disco
    - Con partizione di *back-up*
    - Oppure mediante architettura RAID
      - *Redundant Array of Inexpensive Disks*
      - Oggi la I vale come "*Independent*"

Livelli RAID



Striping: i dati vengono sezionati (per *bit* o *byte*) e ciascuna sezione viene scritta in parallelo su un disco

Al crescere del "livello" RAID cresce la sicurezza dei dati

C: alcuni dischi sono destinati a contenere **copia** dei dati di dischi "gemelli"

P: alcuni dischi (o **parti**) sono destinati a contenere codici di controllo di integrità dei dati

Integrità del *File System* – 2

- **Consistenza del FS**
  - Un *file* viene aperto, modificato e poi salvato
  - Se il sistema cade tra la modifica e il salvataggio il contenuto del *file* su disco diventa inconsistente
- **Consistenza dei blocchi**
  - 2 liste di blocchi con un contatore ∀ blocco
    - Lista dei blocchi in uso dei *file*
    - Lista dei blocchi liberi
  - **Consistenza**
    - Ciascun blocco appartiene a una e una sola lista
  - **Perdita**
    - Un blocco non appartiene ad alcuna lista
  - **Duplicazione**
    - Il contatore del blocco è >1 in una delle due liste

Prestazioni del *File System*

- Una porzione di RAM viene usata come *cache software* di alcune migliaia di blocchi
  - Per ridurre la frequenza di accesso ai dischi
  - L'accesso ai blocchi localizzati in "*cache*" avviene tramite ricerca *hash*
  - La gestione richiede specifica politica di rimpiazzo blocchi
- Occorre però garantire la consistenza dei dati su disco
  - **MS-DOS**
    - I blocchi modificati vengono copiati **immediatamente** su disco
      - *Write through*
    - Alto costo ma consistenza sicura (specie con dischi rimovibili)
  - **UNIX → GNU/Linux**
    - Un processo periodico (`sync`) effettua l'aggiornamento dei blocchi su disco
    - Basso costo e basso rischio con dischi fissi affidabili

Genesi – 1

- **DTSS** (*Dartmouth College Time Sharing System*, 1964)
  - Il primo elaboratore multi-utente a divisione di tempo
  - Programmato in BASIC e ALGOL
  - Presto soppiantato da
- **CTSS** (*MIT Compatible Time Sharing System*, in versione sperimentale dal 1961)
  - Enorme successo nella comunità scientifica
  - Induce MIT, Bell Labs e GE alla collaborazione nel progetto di
- **MULTICS** (*Multiplexed Information and Computing Service*, 1965)
  - Quando Bell Labs abbandona il progetto, Ken Thompson, uno degli autori di MULTICS, ne produce in **assembler** una versione a utente singolo
- **UNICS** (UNiplexed "ICS", 1969)

Genesi – 2

- **1974**
  - Nuova versione di UNIX per PDP-11 completamente riscritta in **C** con Dennis Ritchie
    - PDP-11 (*Programmed Data Processor*)
      - 2 KB *cache*, 2 MB RAM
    - Linguaggio C definito appositamente come evoluzione del rudimentale **BCPL** (*Basic Combined Programming Language*)
  - Enorme successo grazie alla diffusione di PDP-11 nelle università
- **1979**
  - Rilascio di **UNIX v7**, "la" versione di riferimento
    - Perfino Microsoft lo ha inizialmente commercializzato!
      - Sotto il nome di **Xenix**, ma solo a costruttori dell'*hardware* degli elaboratori (p.es.: Intel)

Genesi – 3

- **Portabilità di programmi**
  - Programma scritto in un linguaggio ad alto livello dotato di compilatore per più elaboratori
    - È desiderabile che anche il compilatore sia portabile
  - Dipendenze limitate ad aspetti specifici della architettura di destinazione
    - Dispositivi di I/O, gestione interruzioni, gestione di basso livello della memoria
- **Diversificazione degli idiomi** (1979 – 1986)
  - Avvento di **v7** e divaricazione in due filoni distinti
    - **System V** (AT&T → Novell → **S**anta **C**ruz **O**peration)
      - Incluso Xenix …
    - **4.x BSD** (*Berkeley Software Distribution*, da Bill Joy che poi fonda Sun Microsystems nel 1982)
      - 4.0 rilasciato nell'ottobre 1980, poi evoluto sino a 4.4 Lite R2 (1995)

Genesi – 4

- **Standardizzazione** (1986 –)
  - **POSIX** (*Portable Operating System Interface for UNIX*) racchiude elementi selezionati di **System V** e **BSD**
    - I più maturi e utili secondo l'opinione di esperti "neutrali" incaricati da IEEE e ISO/IEC
    - Definisce l'insieme standard di **procedure di libreria** utili per operare su S/O compatibili
      - La maggior parte contiene chiamate di sistema standard
      - Servizi utilizzabili da linguaggi ad alto livello

Genesi – 5

- **Scelte architetturali per cloni UNIX**
  - ***Micro-kernel*** : MINIX (Tanenbaum, 1987)
    - Nel nucleo **solo** processi e comunicazione
    - Il resto dei servizi (p.es. : FS) realizzato in processi utente
      - MINIX non copre tutti i servizi UNIX
  - **Nucleo monolitico** : GNU/Linux (Linus Torvalds, da v0.01 nel maggio 1991 a v2.6 di oggi)
    - Clone completo aderente a POSIX con qualche libertà
      - Il "meglio" di BSD per esecuzione su PC
    - Modello *open-source* (scritto nel C compilato da **gcc** – ***GNU*** *C compiler*)
    - Parallelo al progetto **GNU** (GNU is NOT UNIX!) **http://www.gnu.org**

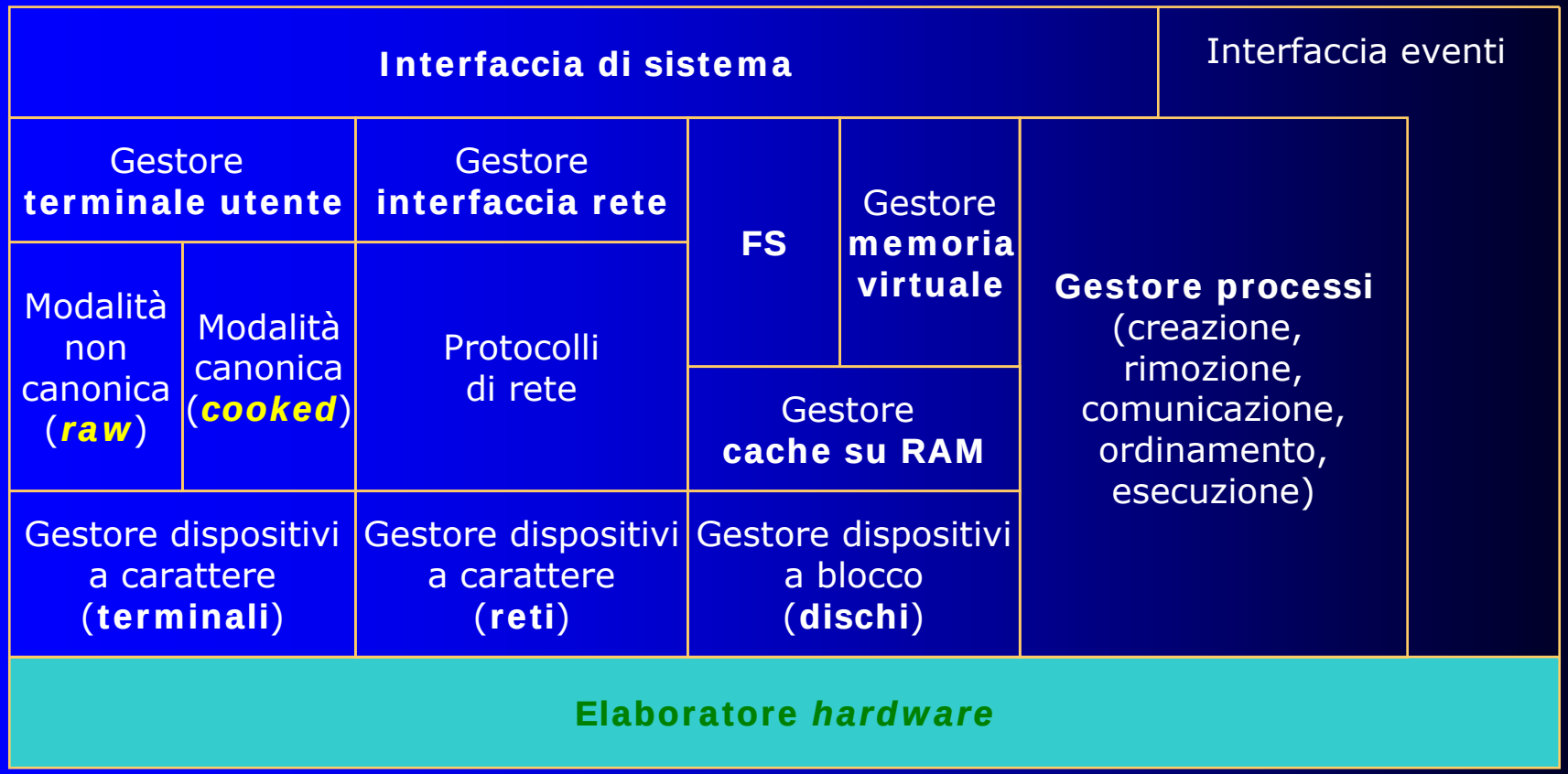
Visione generale – 1

- Sistema **multi-programmato** multi-utente
  - Orientato allo **sviluppo** di applicazioni
  - Progettato per semplicità, eleganza e consistenza
    - **NO** a complessità derivante da compromessi di compatibilità verso il passato
    - **SI** a specializzazione e flessibilità
      - Ogni servizio fa una sola cosa (bene)
      - Facile combinare liberamente più servizi per realizzare azioni più sofisticate
- Architettura a livelli gerarchici

Livelli gerarchici

Architettura di nucleo UNIX

Struttura monolitica = insieme staticamente fissato di moduli oggetto



Interfaccia utente

- UNIX nasce con I/F per linea di comando (*shell*)
  - Più potente e flessibile di GUI ma solo per utenti esperti
  - Una *shell* per ogni terminale utente (`xterm`)
    - Lettura dal *file* "*standard input*"
    - Scrittura sul *file* "*standard output*" o "*standard error*"
      - Inizialmente associati al terminale stesso (visto come *file*)
    - Possono essere re-diretti
      - < per `stdin`, > per `stdout`
  - Caratteri di *prompt* indicano dove editare il comando
  - Comandi compositi possono associare l'uscita di comandi all'ingresso di altri mediante | (*pipe*)
    - Oppure essere combinati in sequenze interpretate (*script*)
  - In modalità normale la *shell* esegue un comando alla volta
    - Ma i comandi possono essere inviati all'esecuzione liberando la *shell* (&, *background*)

Gestione dei processi (UNIX) – 1

- **Processo**
  - La sola entità attiva nel sistema
  - Inizialmente definito come sequenziale
    - Dotato di un singolo flusso di controllo interno
  - Concorrenza a livello di processi
    - Molti processi attivati direttamente dal sistema (*daemon*)
    - Creazione mediante `fork()`
    - La discendenza di un processo costituisce un "gruppo"
    - Comunicazione mediante scambio messaggi (*pipe*) e invio di segnali (*signal*) entro un gruppo

Gestione dei processi (UNIX) – 2

- Processi con più flussi di controllo interni
  - Detti *thread*
    - Apportano maggior capacità di calcolo entro un processo
  - 2 scelte realizzative possibili per gestire *thread*
    - Nello spazio di nucleo → deve cambiare il nucleo!
    - Nello spazio di utente
      - Più facile da aggiungere
      - Il nucleo però ordina solo i processi
  - POSIX non impone una particolare scelta
  - Un *thread* può svolgere compiti specializzati
  - La creazione di un *thread* gli assegna identità, attributi, compito e argomenti

Gestione dei processi (UNIX) – 3

- Completato il proprio lavoro il *thread* termina se stesso volontariamente
  - Invocando la procedura `pthread_exit`
- Un *thread* può sincronizzarsi con la terminazione di un suo simile
  - Invocando la procedura `pthread_join`
- L'accesso a risorse condivise viene sincronizzato mediante semafori a mutua esclusione
  - Tramite le procedure `pthread_mutex{_init, _destroy}`
- L'attesa su condizioni logiche (p.es. : risorsa libera) avviene mediante variabili speciali simili a *condition variables* (ma senza *monitor*)

Gestione dei processi (UNIX) – 4

- Un *thread* può avere 2 "modi" operativi
  - Modo **normale**
    - Esecuzione nello spazio di utente con diritti, memoria virtuale, risorse rigidamente distinti da quelli del nucleo
  - Modo **nucleo**
    - Esecuzione con privilegi, memoria virtuale e risorse di nucleo
  - Il **cambio di modo** consegue a una **chiamata di sistema**
    - Chiamata incapsulata in una **procedura di libreria** per consentirne l'invocazione da programmi scritti in linguaggi ad alto livello (p.es. C)

Gestione dei processi (UNIX) – 5

- Il nucleo impiega 2 strutture per gestire processi
  - **Tabella dei processi**
    - **Permanentemente in RAM** e per **tutti** i processi
      - Parametri di ordinamento (p.es. : priorità, tempo di esecuzione cumulato, tempo di sospensione in corso, ...)
      - Descrittore della memoria virtuale del processo
      - Lista dei segnali significativi e del loro stato
      - Stato, identità, relazioni di parentela, gruppo di appartenenza
  - **Descrittore degli utenti**
    - In RAM **solo** per i processi attualmente **attivi**
      - Parte del contesto (immagine dei registri dell'elaboratore)
      - Stato dall'ultima chiamata di sistema (parametri, risultato)
      - Tabella dei descrittori dei *file* aperti
      - Crediti del processo
      - *Stack* da usare in modo nucleo

Esecuzione di comando di *shell* – 1

Codice semplificato di un processo *shell*

```
while (TRUE) {
    type_prompt(); // mostra prompt sullo schermo
    read_command(cmd, par); // legge linea comando
1   pid = fork();
    if (pid < 0) {
        printf("Errore di attivazione processo.\n");
        continue; // ripeti ciclo
    };
    if (pid != 0) {
        waitpid(-1, &status, 0); // attende la terminazione
                                 // di qualunque figlio
    } else {
2       execve( cmd, par, 0);
    }
}
```

Codice eseguito dal padre

Codice eseguito dal figlio

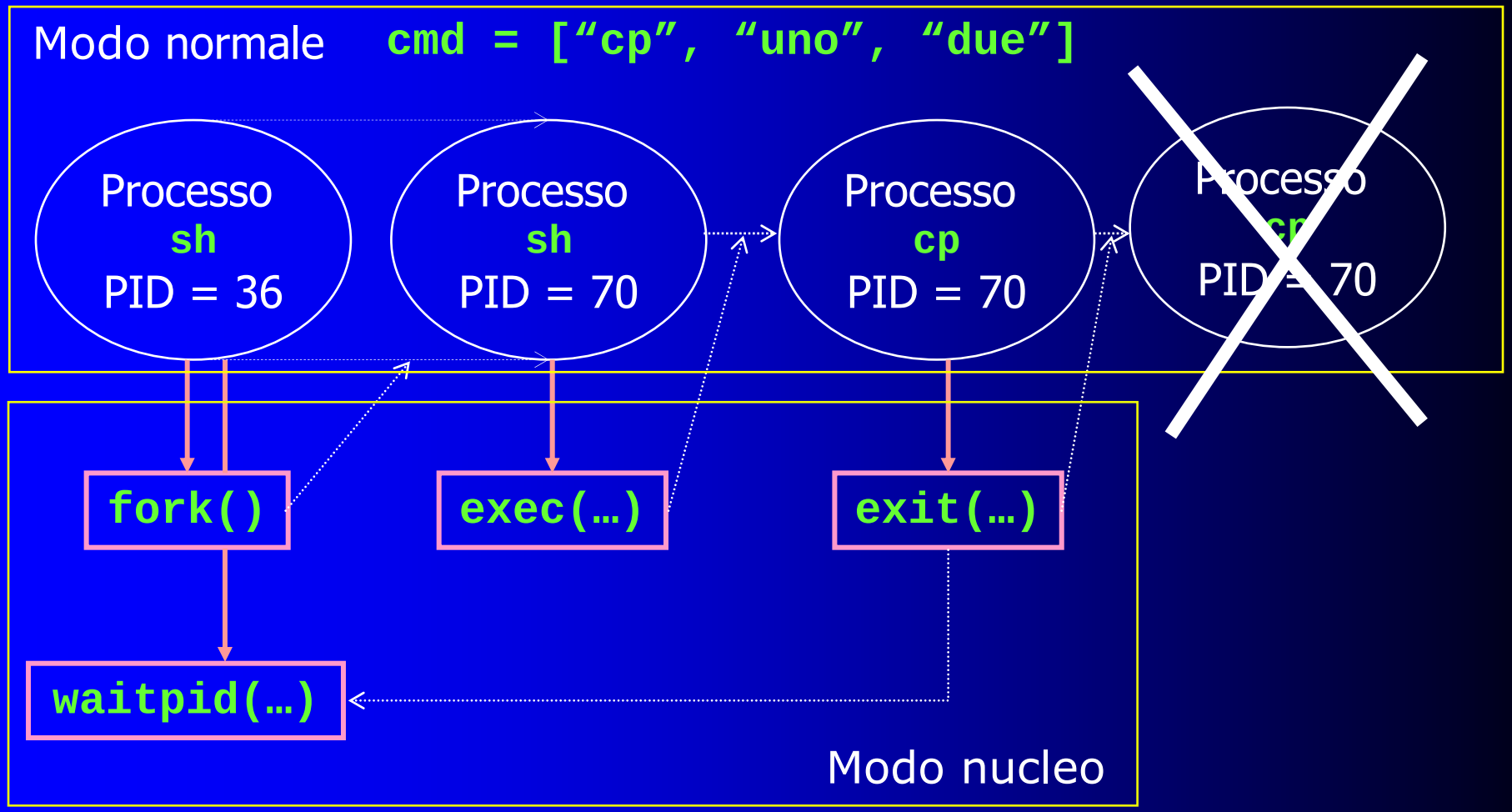
Esecuzione di comando di *shell* – 2

1. Il processo chiamante passa in **modo nucleo** e prova a inserire i propri dati per il figlio nella **Tabella dei Processi** (incluso il PID). Se ci riesce, alloca memoria per *stack* e dati del figlio. A questo punto il codice del figlio è ancora **lo stesso** del padre

2. La linea di comando emessa dall'utente viene passata al processo figlio come *array* di stringhe. La **exec**, che opera in **modo nucleo**, localizza il programma da eseguire e lo sostituisce al codice del chiamante, passandogli la linea di comando e le "**definizioni di ambiente**" specifiche per il nuovo processo

Le aree dati e codice sono copiate per il figlio solo se questi le modifica (***copy-on-write***) e caricate solo quando riferite (***paging-on-demand***)

Esecuzione di comando di *shell* – 3



Gestione dei processi (UNIX) – 6

- `fork()` **duplica** il processo chiamante creando un **processo figlio** uguale a se stesso ma distinto
  - Che accade se il processo creante comprende più *thread* ?
- Vi sono 2 possibilità
  - **Tutti i *thread* del padre vengono clonati**
    - Difficile gestire il loro accesso concorrente a dati e risorse condivise con i *thread* del padre
  - **Solo un *thread* del padre viene clonato**
    - Possibile sorgente di inconsistenza rispetto alle esigenze di cooperazione con le *thread* non clonate
- Dunque il *multi-threading* aggiunge gradi di complessità
  - Al FS
    - Più difficile assicurare consistenza nell'uso concorrente di *file*
  - Alla comunicazione tra entità attive
    - Quale *thread* deve essere il destinatario di segnali inviati al processo?
- `fork()` non era stata pensata per uso da processi con *thread* !

Gestione dei processi (GNU/Linux)

- Maggior granularità nel trattamento della condivisione di strutture di controllo nella creazione di processi e *thread* figli
- Chiamata di sistema alternativa a **`fork`**()

```
pid = clone(fun, stack, ctrl, par);
```

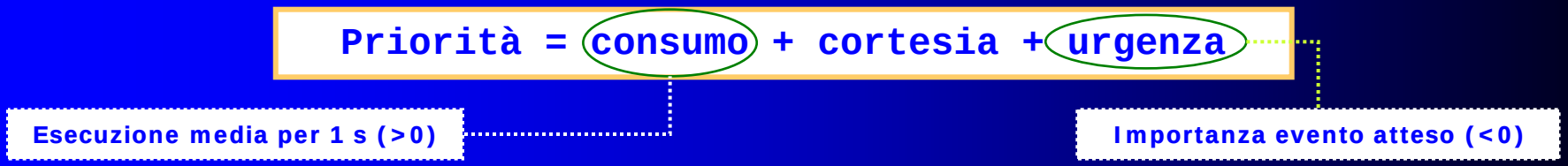
  - `fun` = programma da eseguire nel nuovo "*task*" (processo o *thread*) con argomento `par`
  - `stack` = indirizzo dello *stack* assegnato al nuovo *task*
  - `ctrl` = grado di condivisione desiderato tra il nuovo *task* e l'ambiente del chiamante
    - Spazio di memoria, FS, *file*, segnali, identità

Ordinamento dei processi (UNIX) – 1

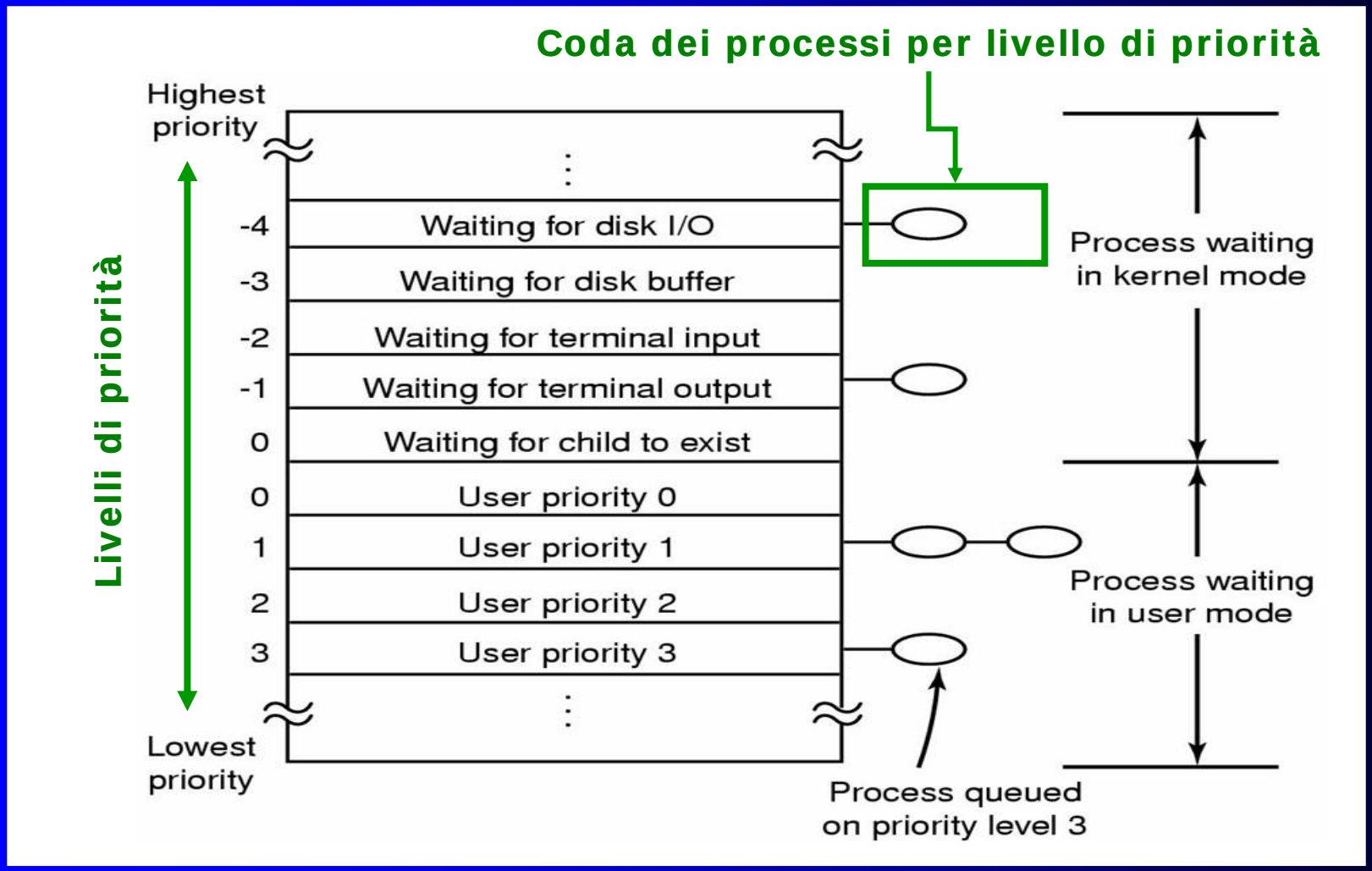
- **Sistema destinato a uso interattivo**
  - Politica di ordinamento con prerilascio
  - Deve garantire buoni tempi di risposta e soddisfare le aspettative degli utenti
- **Algoritmo di ordinamento a 2 livelli**
  - Livello basso (***dispatcher***)
    - Seleziona un processo tra quelli pronti (in RAM)
      - Secondo le indicazioni fornite dallo *scheduler*
  - Livello alto (***scheduler***)
    - Assicura che tutti i processi abbiano opportunità di esecuzione spostando processi tra RAM e disco

Ordinamento dei processi (UNIX) – 2

- **Politica di ordinamento**
  - Selezione per priorità decrescente
    - Priorità alta (< 0) per esecuzione in modo nucleo
    - Priorità bassa (> 0) per esecuzione in modo normale
  - Più processi possono avere la stessa priorità
    - 1 coda di processi $\forall$ livello di priorità
    - Selezione *round robin* da testa della coda
    - Esecuzione a quanti con ritorno in fondo alla coda
    - Aggiornamento periodico della priorità dei processi



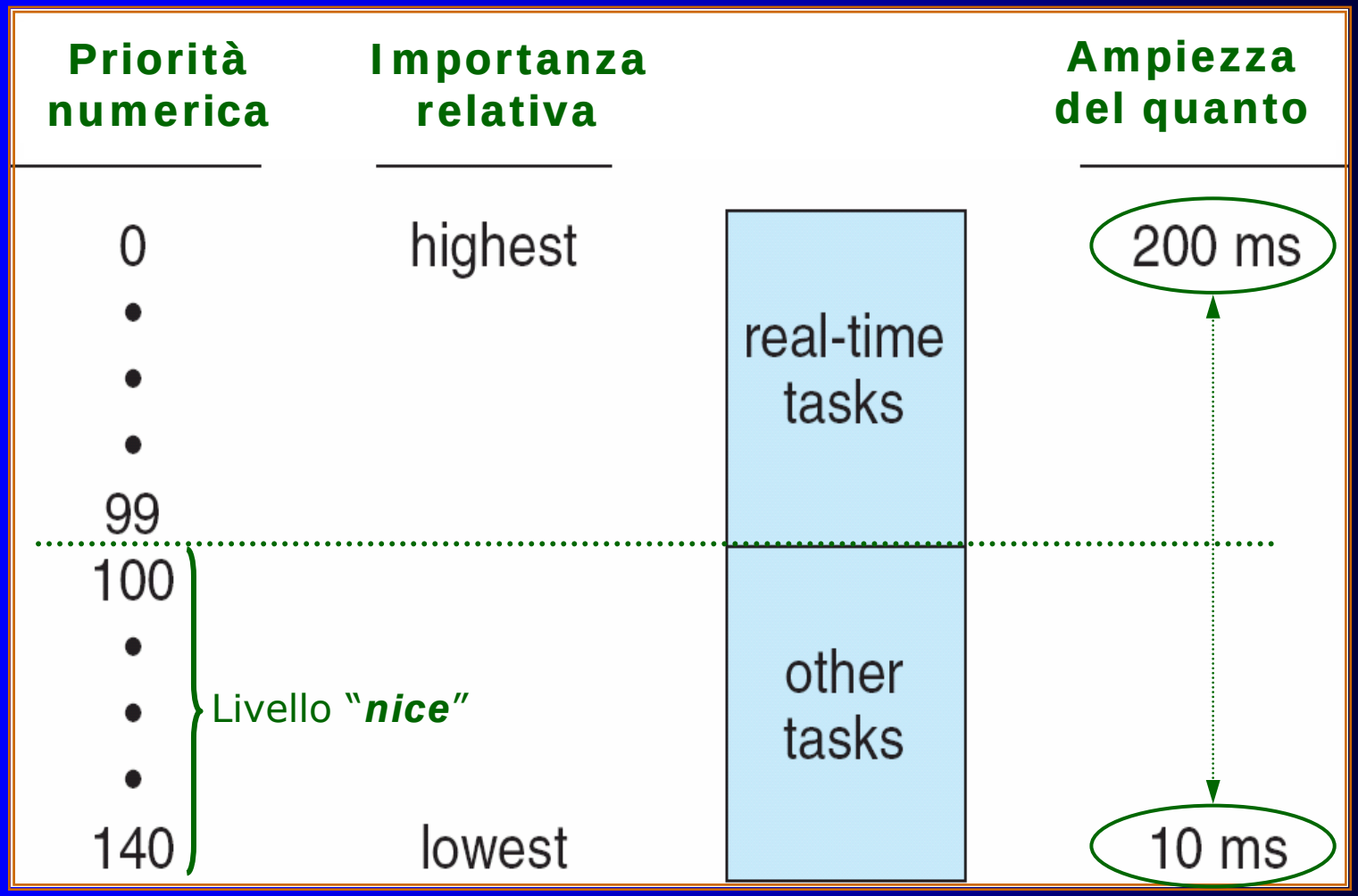
Ordinamento dei processi (UNIX) – 3



Ordinamento dei processi (GNU/Linux) – 1

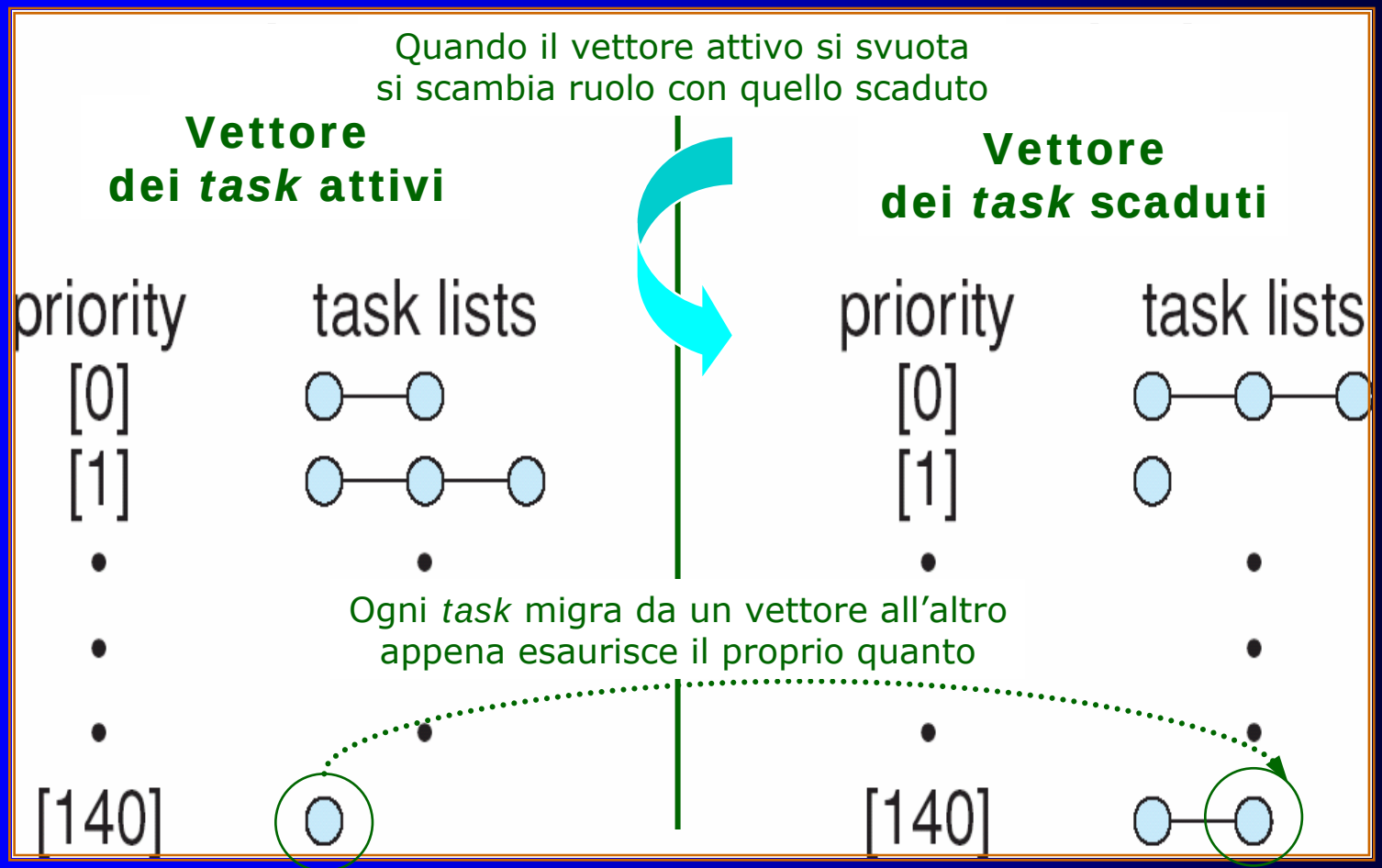
- I *thread* sono gestiti direttamente dal nucleo
  - Ordinamento per "*task*" (*thread* o processo indistintamente)
  - Selezione distinta tra classi distinte
  - Prerilascio per fine quanto o per attesa di evento
- 3 **classi di priorità** di *task*
  - **Tempo reale con politica FCFS a priorità senza prerilascio**
    - A priorità uguale viene scelto il *task* in attesa da più tempo
  - **Tempo reale con politica RR a priorità**
    - Prerilascio per quanti con ritorno in fondo alla coda
  - **Divisione di tempo RR a priorità**
    - Priorità dinamica con premio o penalità rispetto al grado di interattività con I/O (alta → premio, bassa → penalità)
    - Nuovo valore assegnato al *task* all'esaurimento del suo quanto corrente

Ordinamento dei processi (GNU/Linux) – 2



Silberschatz, Galvin and Gagne ©2005

Ordinamento dei processi (GNU/Linux) – 3



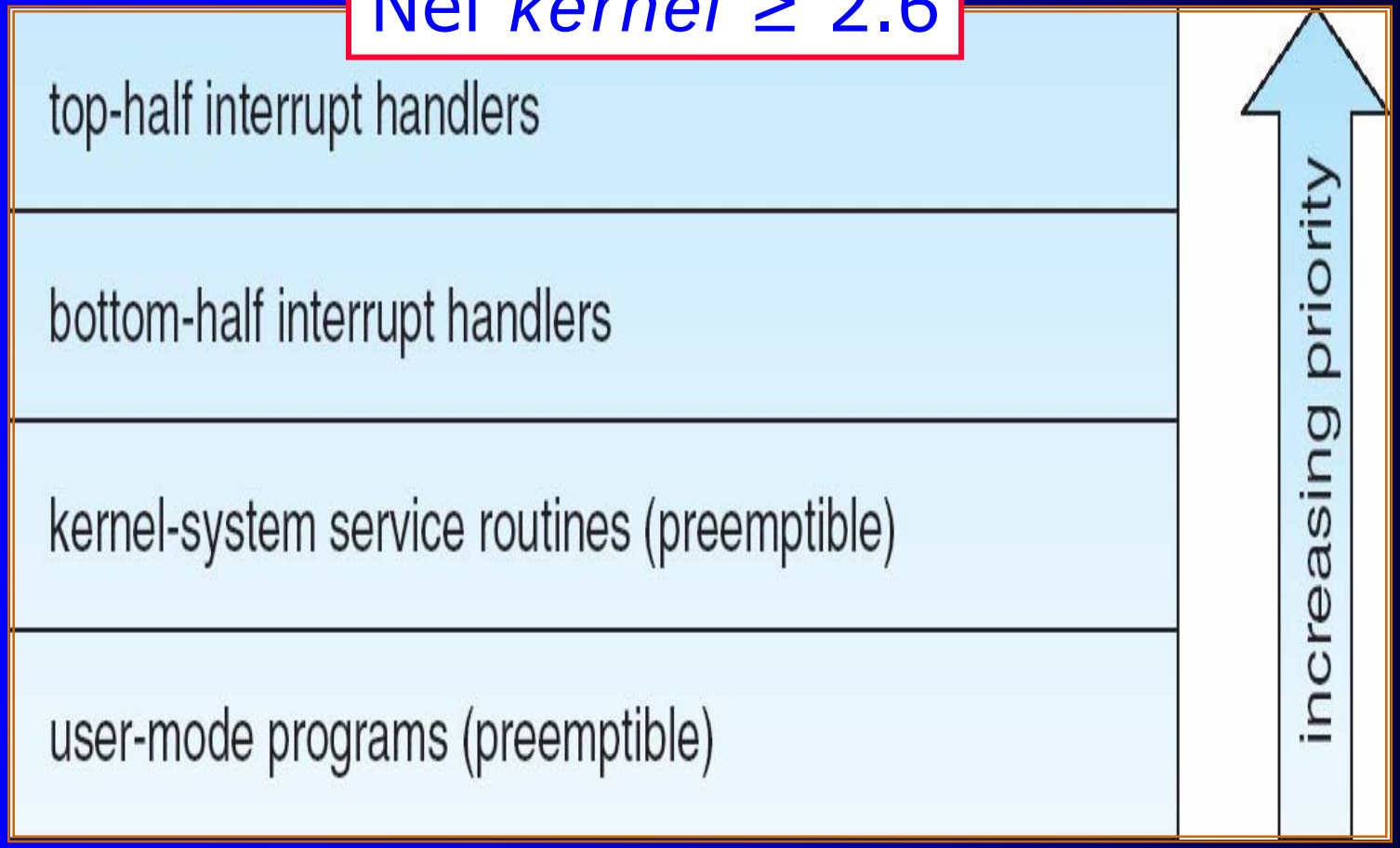
Silberschatz, Galvin and Gagne ©2005

Ordinamento dei processi (GNU/Linux) – 4

- Per versione < 2.6 l'attività dei processi in modo nucleo **non ammetteva** prerilascio
  - Ma questo causava gravi problemi di **inversione di priorità**
- Con versione ≥ 2.6 si usa granularità più fine
  - Inibizione selettiva di prerilascio
    - Per sezioni critiche corte
  - Uso di semafori convenzionali
    - Per sezioni critiche lunghe
  - Uso minimale di disabilitazione delle interruzioni
    - La parte **immediata** dei gestori (*top half*, molto breve) disabilita
    - La parte **differita** (*bottom half*, più lunga) non disabilita ma la sua esecuzione viene privilegiata rispetto a ogni altra attività
      - Tranne che ad ogni eventuale altra parte immediata

Ordinamento dei processi (GNU/Linux) – 5

Nei *kernel* ≥ 2.6



Silberschatz, Galvin and Gagne ©2005

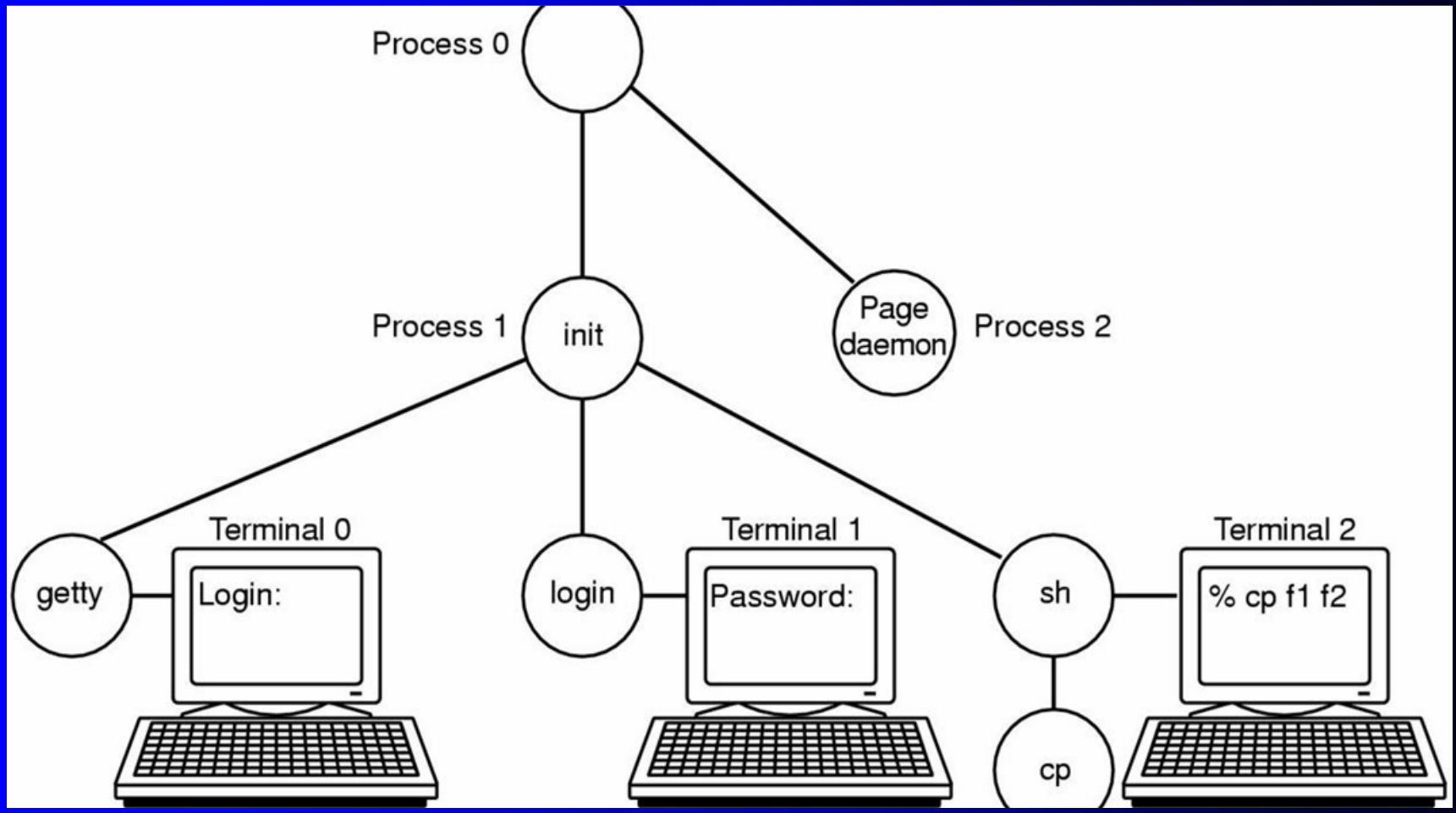
Inizializzazione (GNU/Linux) – 1

- BIOS carica l'MBR in RAM e lo "esegue"
  - MBR = 1 settore di disco = 512 B
- L'MBR carica il programma di *boot* dal corrispondente blocco della partizione attiva selezionata
  - Lettura della struttura di FS, localizzazione e caricamento del nucleo di S/O
- Il programma di inizializzazione del nucleo è scritto in *assembler* (specifico per l'elaboratore!)
  - Poche azioni di configurazione di CPU e RAM
  - Il controllo passa poi al `main` di nucleo
    - Configurazione del sistema con caricamento **dinamico** dei gestori dei dispositivi rilevati
    - Inizializzazione e attivazione del processo 0

Inizializzazione (GNU/Linux) – 2

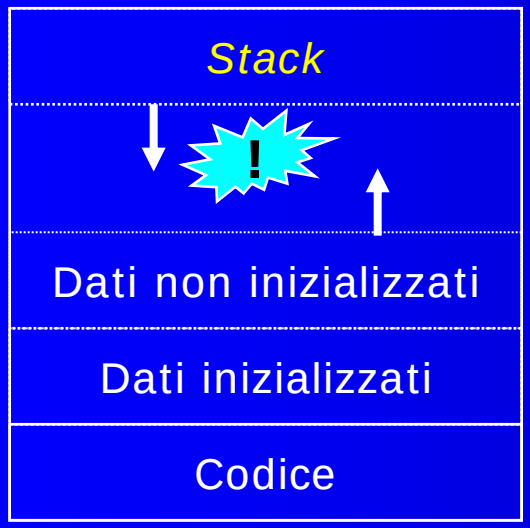
- **Processo 0**
  - Configurazione degli orologi, installazione del **FS di sistema**, creazione dei processi **1** (`init`) e **2** (*`daemon`* delle pagine)
- **Processo 1**
  - Configurazione modo utente (singolo, multi)
    - Esecuzione *script* di inizializzazione **`shell`** (`/etc/rc` etc.)
    - Lettura numero e tipo terminali da `/etc/ttys`
    - **`fork`**`()` $\forall$ terminale abilitato ed **`exec`**`("`**`getty`**`")`
- **Processo `getty`**
  - Configurazione del terminale e attivazione del *prompt* di *login*
  - Al *login* di utente, **`exec`**`("[/usr]/bin/`**`login`**`")` con verifica della *password* d'accesso (memorizzate in `/etc/passwd`)
  - Infine **`exec`**`("`**`shell`**`")` e poi si procede come mostrato alle diapositive 269-271

Inizializzazione (GNU/Linux) – 3



Gestione della memoria – 1

- Massima **semplicità** per massima **portabilità** su architetture fisiche diverse
- Ogni processo possiede un proprio spazio di indirizzamento privato (**memoria virtuale**)
  - Suddiviso in 4 sezioni



Stack: Pila dei contesti, dimensione variabile, **R**/**W**

Block Storage Segment, *Data segment*: dimensione variabile, **R**/**W**

Text segment, dimensione fissa, **R**

Gestione della memoria – 2

- Il segmento dati varia in dimensione a seconda delle attività del programma (p.es. `malloc()` in C)
  - POSIX **non definisce** le chiamate di sistema relative alle gestione della memoria
  - Una parte del segmento dati può ospitare *file* mappati in memoria
- Il segmento *stack* contiene l'ambiente d'esecuzione corrente (***record* di attivazione**) e cresce nel verso **opposto** alla crescita del segmento dati
- Il segmento codice può essere condiviso tra più processi
  - Ma **non gli altri** segmenti
  - Tranne che per processi duplicati da `fork()`

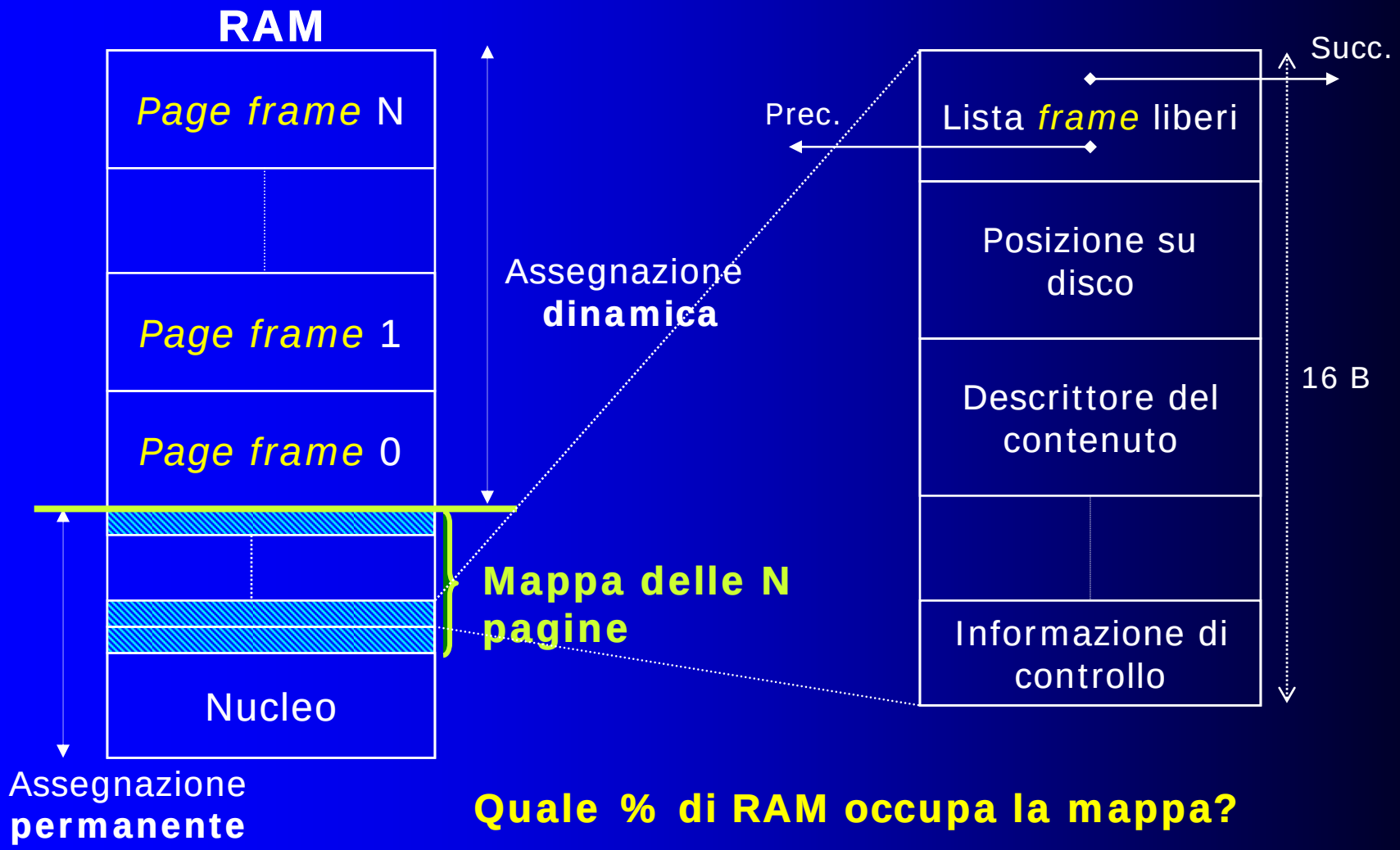
Gestione della memoria (UNIX) – 3

- In origine l'allocazione di memoria principale avveniva mediante ***swap*** di processi
  - Rimpiazzo di interi processi quando una particolare esecuzione rilevava mancanza di memoria
    - A seguito di `fork()`
    - A causa di allocazione esplicita richiesta dal programma
    - Per allocazione implicita conseguente a chiamata di procedura
      - Che richiede l'allocazione di un nuovo *record* di attivazione
  - Il gestore (***swapper***) creava lo spazio necessario salvando su disco i processi sospesi con più tempo d'esecuzione recente e minor priorità
    - Bastava utilizzare una lista dei blocchi liberi su disco

Gestione della memoria (UNIX) – 4

- In seguito fu introdotta paginazione con modalità **a richiesta** (***paging on demand***)
  - Un processo è eseguibile se il suo descrittore e la sua tabella delle pagine si trovano in RAM
    - Il suo spazio di indirizzamento è caricato da disco per ogni riferimento che richiede dati non presenti in RAM
    - Nessun caricamento anticipato di pagine
      - Nessun uso di *working set*
  - Il processo "**2**" gestisce lo stato dei *page frame* in RAM
    - *Page daemon*
    - Tenendo nucleo di S/O e "mappa delle pagine" (*core map*) **sempre in RAM**
    - Il resto è paginato e ciascuna *page frame* indica il proprio uso
      - Codice, dati, *stack*, tabella delle pagine
      - Altrimenti in lista pagine libere

Mappa delle pagine (*core map*)

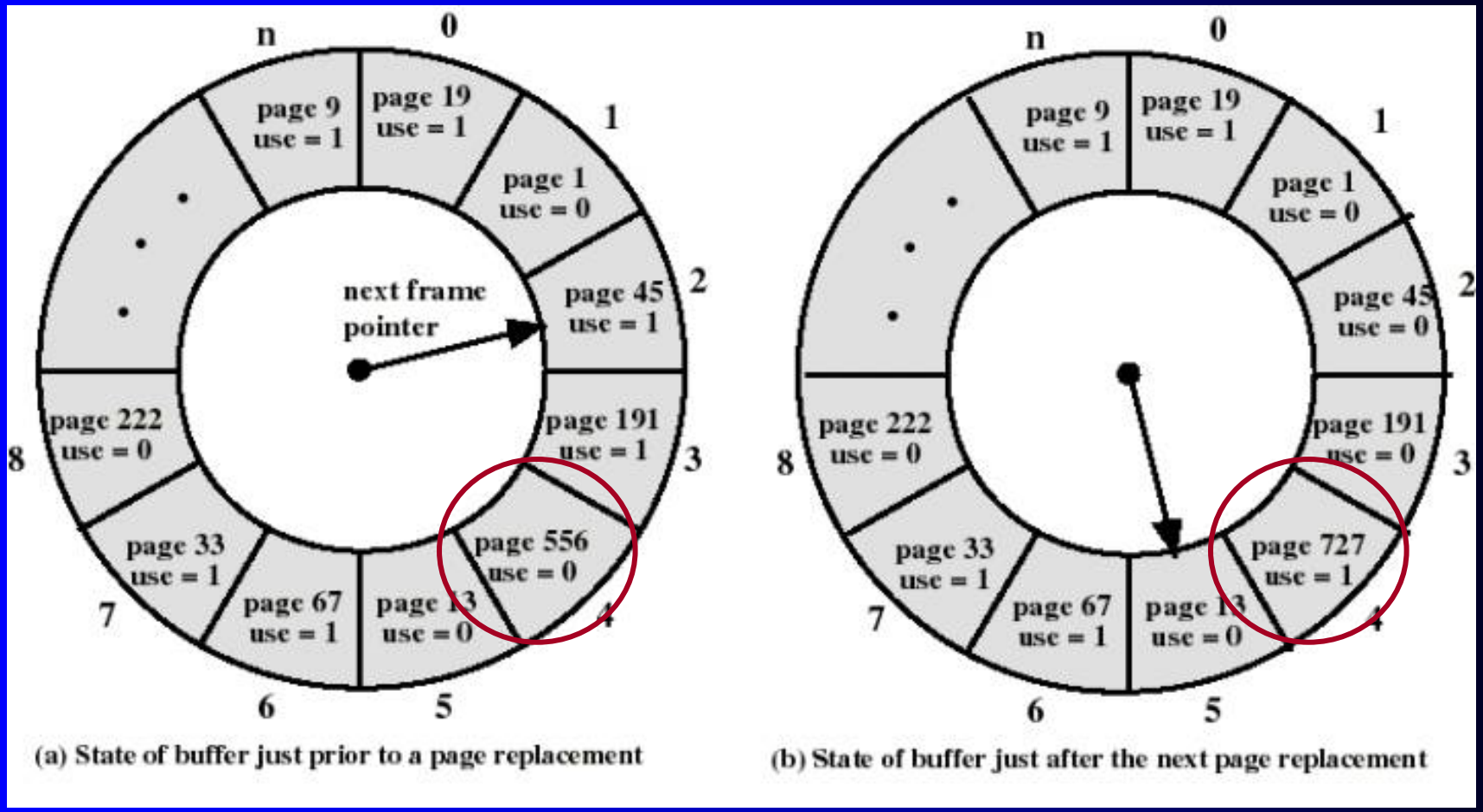


Quale % di RAM occupa la mappa?

Gestione della memoria (UNIX) – 5

- *Page daemon* verifica con periodo 1/4 s che in RAM vi siano ≥ `lotsfree` pagine libere
  - Se ne mancano ne libera quante ne servono salvandone il contenuto corrente su un'area di disco specifica per pagina
  - La selezione delle pagine in uscita usa un algoritmo "**a doppia passata**"
    - ***Two-handed clock algorithm***
    - Lista circolare delle pagine
    - La 1ª passata pone a 0 il *bit* di riferimento
    - La 2ª passata, a distanza **programmabile**, rimuove le pagine nel frattempo non riferite (*bit* vale 1 altrimenti)

Orologio a una passata



Intervallo troppo lungo tra 2 passate successive!

Gestione della memoria (UNIX) – 6

- Il *page daemon* limita la frequenza di paginazione spostando processi su disco
  - Quelli che non abbiano eseguito negli ultimi 20 s
  - Tra i 4 più grandi quello da più tempo in memoria
- Se vi è spazio libero il *page daemon* riporta in RAM processi pronti selezionati con una euristica di “valore”
  - Caricando solo il descrittore di processo e la sua tabella delle pagine
    - Lasciando che il resto sia caricato via *paging on demand*

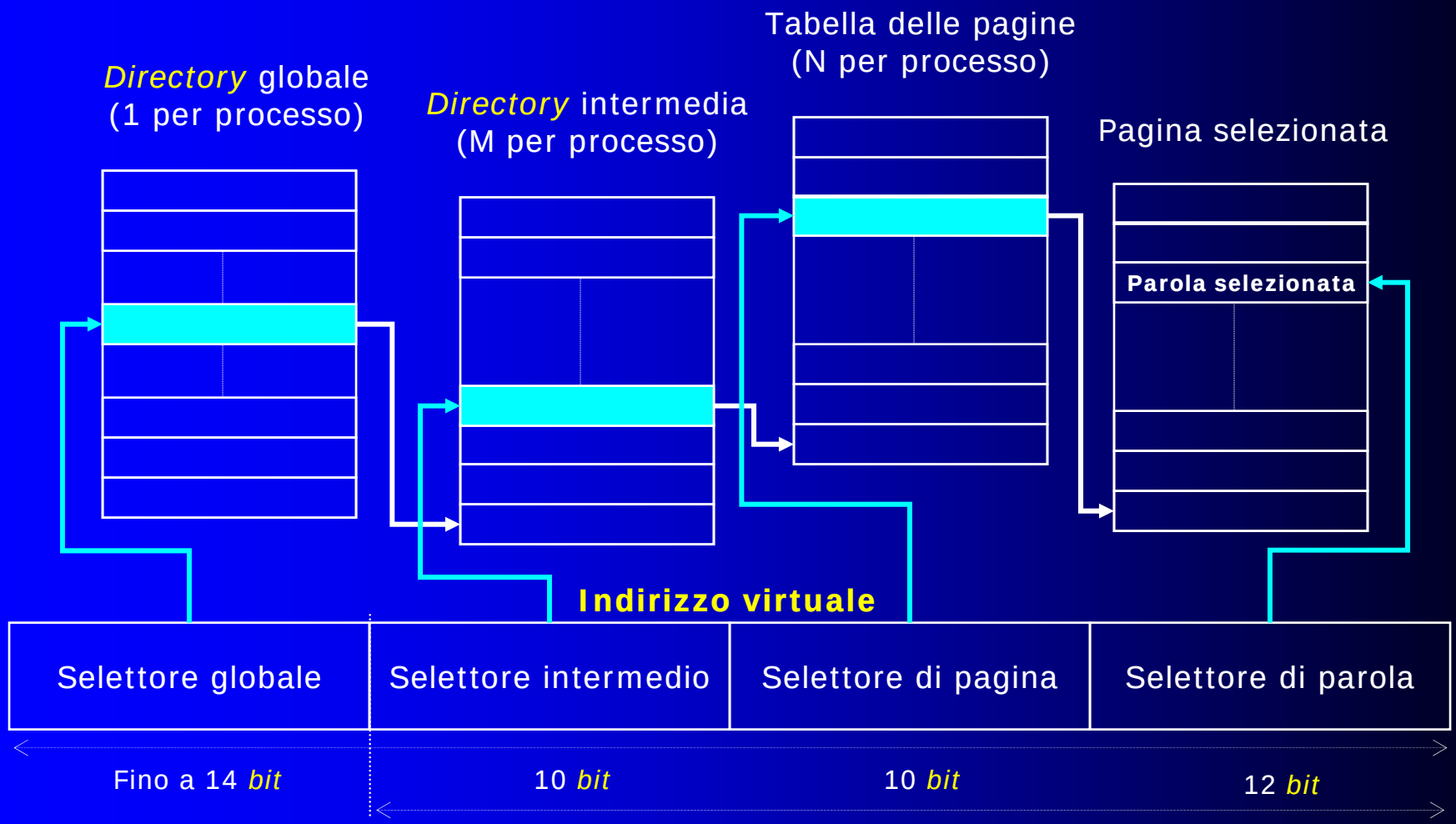
Gestione della memoria (GNU/Linux) – 6

- Per architetture a 32 *bit* la memoria virtuale di processo è ampia 4 GB
  - 1 GB **riservato e invisibile** al modo operativo normale per la tabella delle pagine del processo e per altri dati di controllo **a uso del nucleo**
- Spazio suddiviso in **regioni** = **sequenze contigue** di pagine
  - Le regioni non sono necessariamente **consecutive** tra loro
- Ogni regione ha un descrittore noto al nucleo
  - Lo spazio di indirizzamento virtuale di un processo è visto come una **lista di descrittori** di regione

Gestione della memoria (GNU/Linux) – 7

- La `fork()` di GNU/Linux replica per il figlio l'intera lista di descrittori del padre
- Le pagine del figlio sono fisicamente duplicate **solo** in caso di modifica (***copy on write***)
  - La regione è marcata **R/W**
  - Le sue pagine dati sono inizialmente marcate **R**
  - Ogni richiesta di scrittura causa eccezione così il nucleo duplica la pagina richiesta e marca la copia come **R/W**

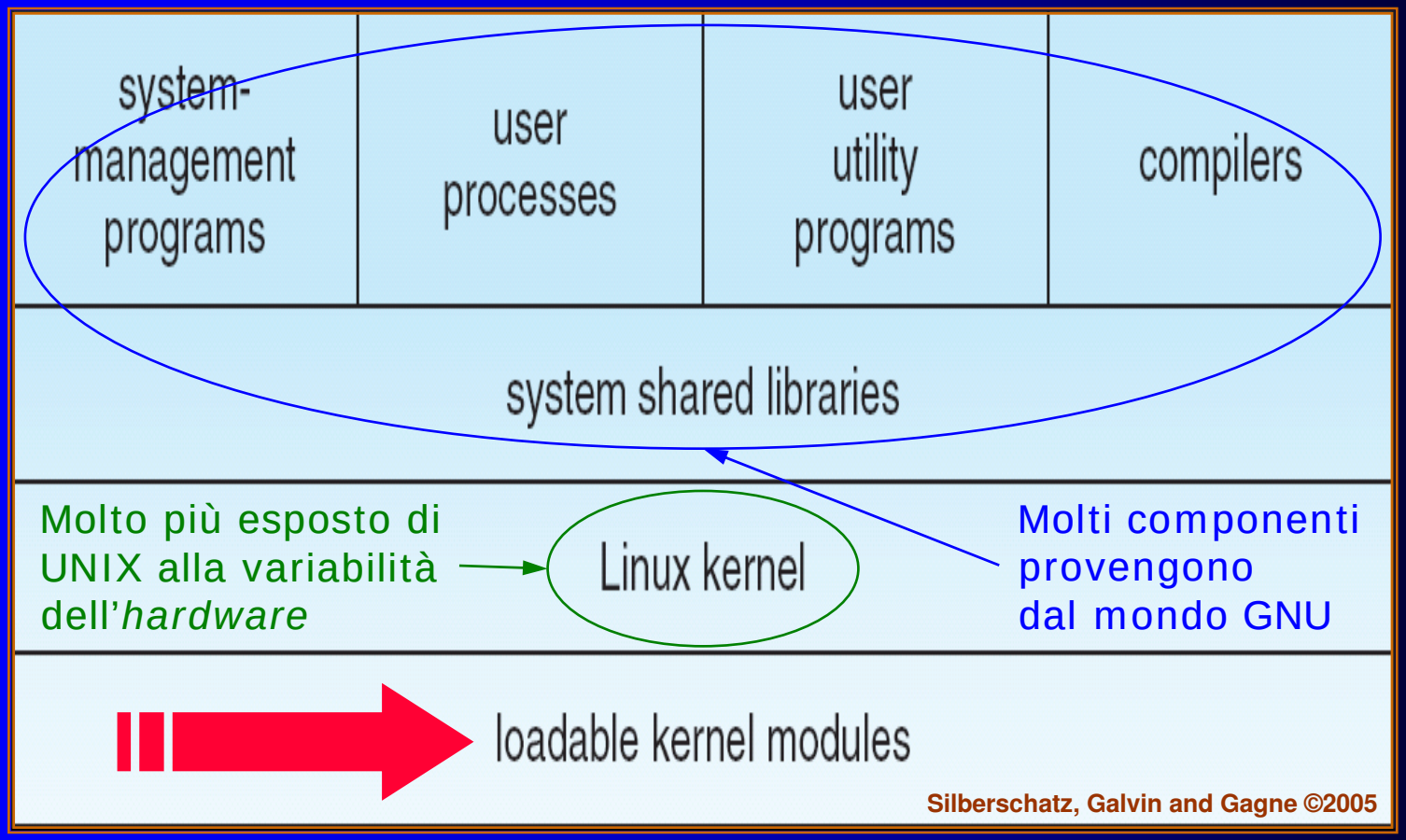
Gestione della memoria (GNU/Linux) – 8



Gestione della memoria (GNU/Linux) – 9

- Il nucleo rimane **sempre** in RAM
  - Ha dimensione **variabile** a causa del caricamento **dinamico** di moduli di gestione dispositivi
- La RAM rimanente viene usata per
  - **[P1]** Le pagine attive dei processi utente
  - **[P2]** Una *cache* di blocchi di *file* usata dal FS
    - Dimensione **variabile** organizzata per pagine
  - **[P3]** Un insieme di pagine utente inattive ma presenti
- La RAM viene assegnata in frazioni (*slab*) di dimensione variabile e arbitraria

Architettura di S/O GNU/Linux



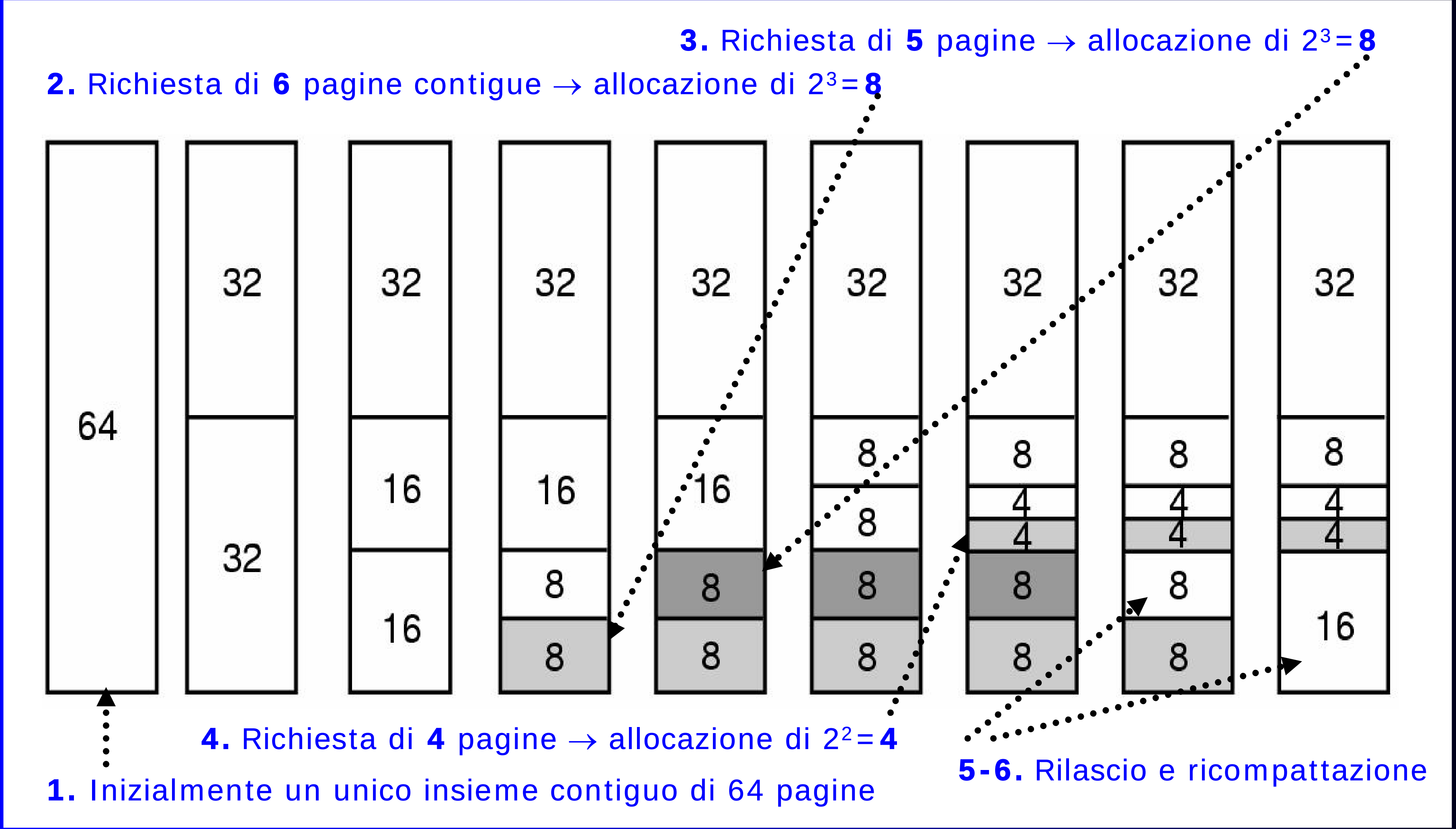
Moduli di nucleo caricabili

- Il nucleo GNU/Linux di base **non include** gestori di dispositivi
  - Caricati in fase di installazione oppure anche successivamente
    - E allo stesso modo possono essere anche rimossi
- **[1]** Trattamento del modulo in ingresso da parte del nucleo
  - Caricamento del codice oggetto del modulo nella memoria virtuale del nucleo
  - Collegamento dei simboli usati dal modulo alla tabella dei simboli del nucleo
- **[2]** Registrazione del modulo gestore
  - Presso tabelle mantenute dal nucleo
    - Servizi e meccanismi distinti per dispositivi distinti
      - Per esempio: *mouse*, disco, FS, connessione di rete, ...
  - Inizializzazione e poi eventualmente rimozione
- **[3]** Regolazione dei possibili conflitti tra gestori
  - Tramite “prenotazione” del gestore al nucleo del proprio accesso a dispositivo

Gestione della memoria (GNU/Linux) – 10

- **Algoritmo di allocazione primario** (*buddy*)
  - Ogni richiesta di ampiezza N è arrotondata a $2^n \geq N$
  - La memoria disponibile viene frazionata in metà successive fino a frazioni di ampiezza 2^n
    - Una singola frazione viene assegnata al richiedente
    - Una struttura ausiliaria contiene la testa di liste predefinite di frazioni di ampiezza 2^i (i=0, ..., n) per velocizzare la ricerca
  - La memoria disponibile usata per l'allocazione è sempre la frazione libera di minore dimensione
  - Al rilascio ogni frazione tornata libera si unisce con la frazione vicina se libera (il suo *buddy*)
  - Due algoritmi sussidiari cercano di ridurre la frammentazione causata dall'algoritmo primario

Funzionamento del *buddy algorithm*



Gestione della memoria (GNU/Linux) – 11

- I segmenti codice e i *file* mappati in memoria hanno un corrispondente *file* su disco
- Al resto (aree di lavoro dei processi) si assegna una **partizione paginata** (***paging partition***) vista come *byte stream* oppure un *file* di pagine
  - La partizione paginata è d'uso più semplice e veloce
    - Nessun limite fissato di scrittura
    - Possibilità di scrittura contigua
- Le pagine libere sulla partizione e/o sul *file* sono individuate mediante *bitmap*
  - Lo spazio disponibile viene assegnato alle pagine di lavoro rimosse **temporaneamente** dalla RAM

Gestione della memoria (GNU/Linux) – 12

- **`kswapd`** è il *page daemon* e ha periodo 1 s
  - Per ogni attivazione esegue fino a 6 passate di un ciclo di lavoro cercando pagine da spostare su disco
    - Tra quelle in **[P2]** e **[P3]** con una variante del *clock algorithm*
    - Tra quelle condivise da più processi ma poco usate
    - Infine tra quelle in **[P1]**
      - Cominciando dal processo con più pagine scandendo l'intera lista dei suoi descrittori di regione (per indirizzo virtuale) con *clock algorithm* ma **solo** sulle pagine attive
  - Ogni pagina selezionata modificata
    - Posta in attesa di riscrittura se ha corrispondente su disco
    - Altrimenti salvata su *paging partition* o *paging file*
- Il *daemon* **`bdflush`** gestisce la riscrittura

Gestione dell'I/O – 1

- UNIX tratta i dispositivi di I/O come *file* di tipo speciale, ciascun con posizione specifica nel FS
  - Per esempio **`/dev/...`**
  - *File* orientati a carattere (p.es. tastiera, rete, ...)
  - *File* orientati a blocco (p.es. disco)
- Un gestore (*device driver*) è associato in modo esclusivo a ciascun dispositivo o a famiglia di dispositivi dello stesso tipo
  - Una coppia di indici `<maggiore, minore>` identifica precisamente ciascun dispositivo di I/O

Gestione dell'I/O – 2

- GNU/Linux consente invece caricamento **dinamico** dei moduli di gestione dei dispositivi
  - Soluzione molto preferibile alla configurazione statica che richiede ogni volta una nuova compilazione dell'intero nucleo
    - Inevitabile a fronte della grande varietà di *hardware* attuale
- Il caricamento dinamico richiede al nucleo di effettuare diverse azioni di configurazione
  - **[1]** Rilocazione dello spazio di indirizzamento del modulo
  - **[1-2]** Allocazione delle risorse necessarie
    - P.es. interruzione assegnata al dispositivo
  - **[2]** Configurazione del vettore delle interruzioni
  - **[2]** Attivazione e inizializzazione del gestore

Gestione dell'I/O – 3

- Un *file* speciale (detto *socket*) viene utilizzato per la connessione di rete e i relativi protocolli
  - Può essere creato e distrutto dinamicamente
  - Un *socket* è associato a uno specifico indirizzo di rete
- Tre tipi di connessione con scelta alla creazione
  - Connessione affidabile a flusso di caratteri (~ **TCP**)
    - Il gestore garantisce la correttezza della trasmissione
    - Invio e ricezione per blocchi di dimensione variabile
  - Connessione affidabile a flusso di pacchetti (**TCP**)
    - Come sopra, ma con invio e ricezione solo per pacchetti
  - Trasmissione inaffidabile di pacchetti (**UDP**)
    - L'utente deve occuparsi di trattare gli eventuali errori

Gestione dell'I/O – 4

- Una zona di RAM è usata come *cache* dedicata per velocizzare R/W di dati su dispositivi a blocchi
  - Ogni blocco richiesto in lettura viene prima cercato in *cache*
  - Ogni blocco scritto viene trattenuto in *cache* il più a lungo possibile
    - Fin quando la *cache* è piena e serve spazio
    - Fino all'attivazione del *daemon* di scrittura su disco

Caratteristiche del *File System* – 1

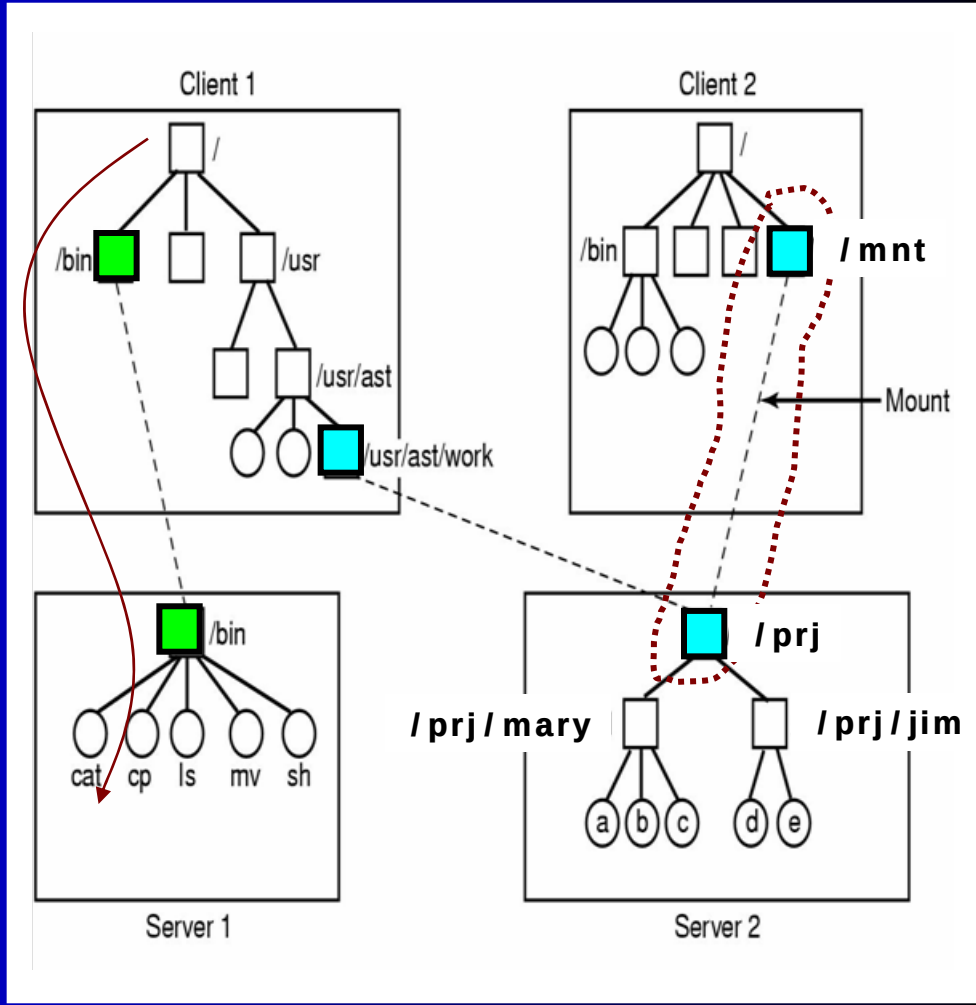
- Paradigma minimalista di tipo "*small is beautiful*"
- *File* visto da FS come **sequenza di *byte*** di contenuto arbitrario
  - Significato fissato dal programma applicativo
- *File* regolari, *file* cartelle (*directory*) e *file* speciali che mappano dispositivi di I/O
- Nome inizialmente limitato a 14 caratteri (UNIX v7)
- Poi esteso fino a 255 (UNIX BSD → GNU/Linux)
  - Estensione **non** obbligatoria
  - Convenzione di estensione a scelta del programma applicativo e/o dell'utente
    - Esempio: `makefile` assume estensione, `emacs` no

Caratteristiche del *File System* – 2

- *File* designato mediante cammino (*path*) assoluto o relativo
  - Il cammino relativo richiede la nozione di "*directory* (di lavoro) corrente"
    - **pwd** per visualizzarne la posizione assoluta
      - *Print working directory*
    - **cd** per cambiare posizione
      - *Change (to) directory*
  - Un intero FS **B** posto su una partizione visibile può essere inglobato in un FS **A** mediante **mount**
    - La radice di **B** viene designata con un nome (cammino) specifico in **A** detto *mount point*

Caratteristiche del *File System* – 3

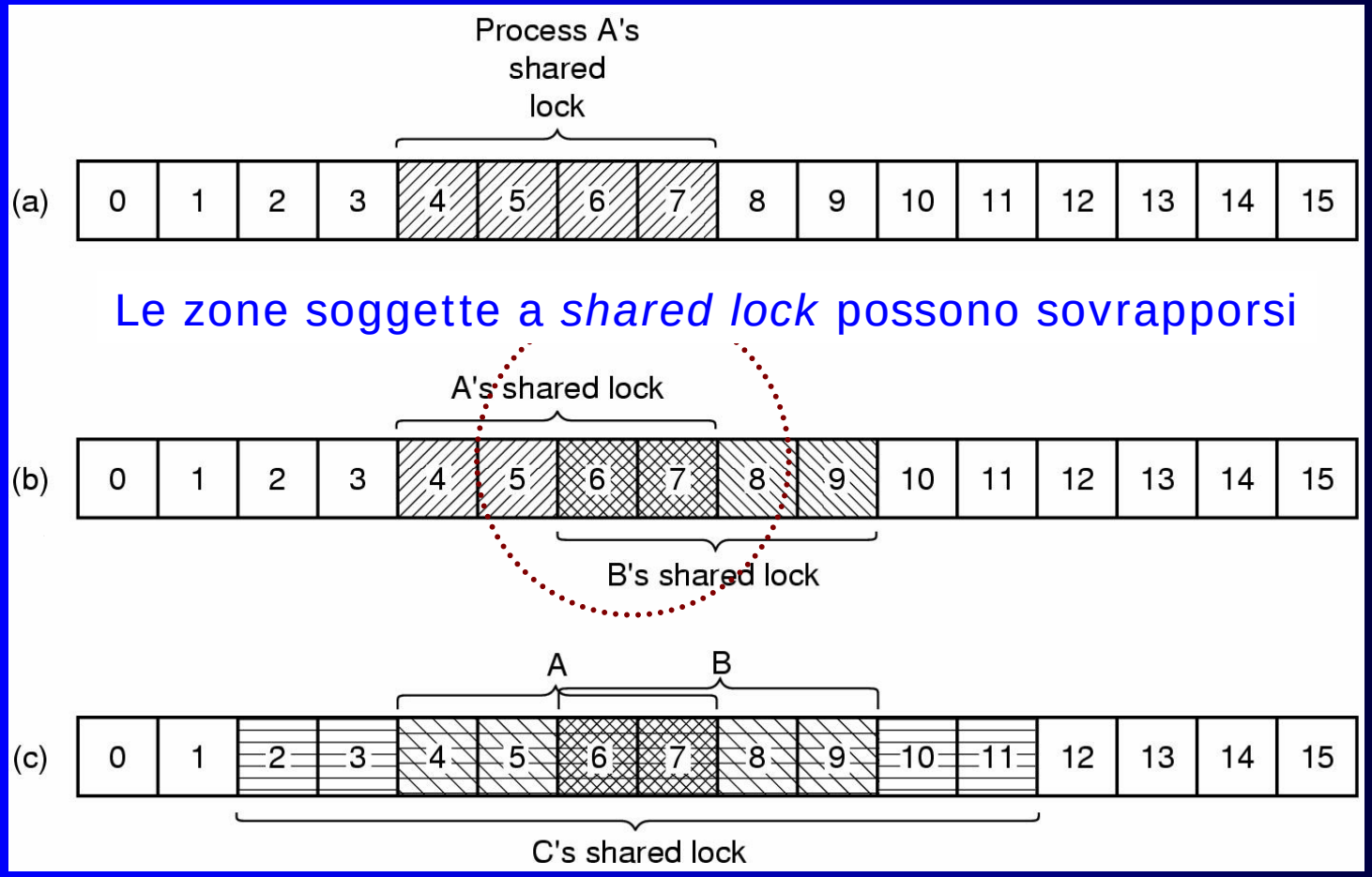
- Tramite **`mount`** la partizione “Client 1” ingloba dalla partizione “Server 1” il FS radicato in `/bin`
  - Il cammino `/bin/cat` ora porta al *file* `cat` come se fosse pienamente nella partizione di “Client 1”
- Lo stesso avviene per il FS radicato in `/prj` della partizione “Server 2”
  - Il cammino `/usr/ast/work/mary/a` porta al *file* `/prj/mary/a`



Caratteristiche del *File System* – 4

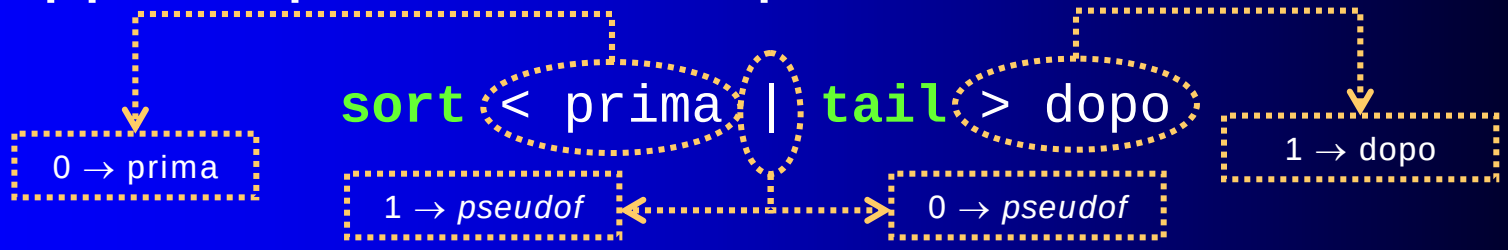
- **Controllo di accessi concorrenti** (*locking*)
  - A grana grossa (per *directory* o per *file*)
    - Mediante uso esplicito di semafori convenzionali
  - A grana fine (per gruppi di *byte* in un *file*)
    - Mediante meccanismi dedicati
- Due distinte modalità d'uso
  - **Accesso simultaneo condiviso** (*shared lock*)
    - Più accessi R alla stessa zona ma anche a zone solo parzialmente sovrapposte
  - **Accesso esclusivo** (*exclusive lock*)
    - Consente un solo accesso per zona selezionata

Caratteristiche del *File System* – 5



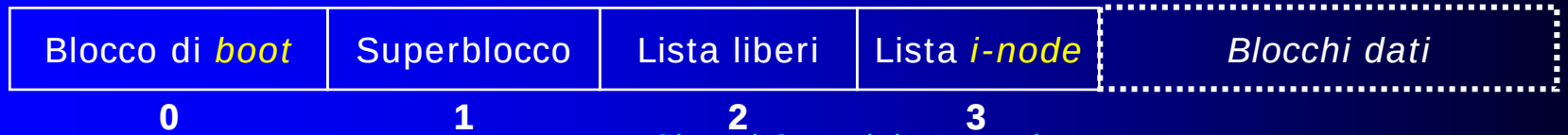
Caratteristiche del *File System* – 6

- ∀ *file* aperto vi è un descrittore (`int > 0`) che designa la posizione corrente di R/W
- 3 descrittori sono pre-assegnati dalla *shell* per altrettanti *file* aperti per definizione
  - **0** per **`stdin`**, **1** per **`stdout`**, **2** per **`stderr`**
  - La redirezione (> , <) modifica tali assegnamenti
- La *pipe* | crea uno pseudo-*file* (con descrittore proprio) che rileva gli **`stdout`** e **`stdin`** di una coppia di processi che operano in cascata



Realizzazione del FS in UNIX – 1

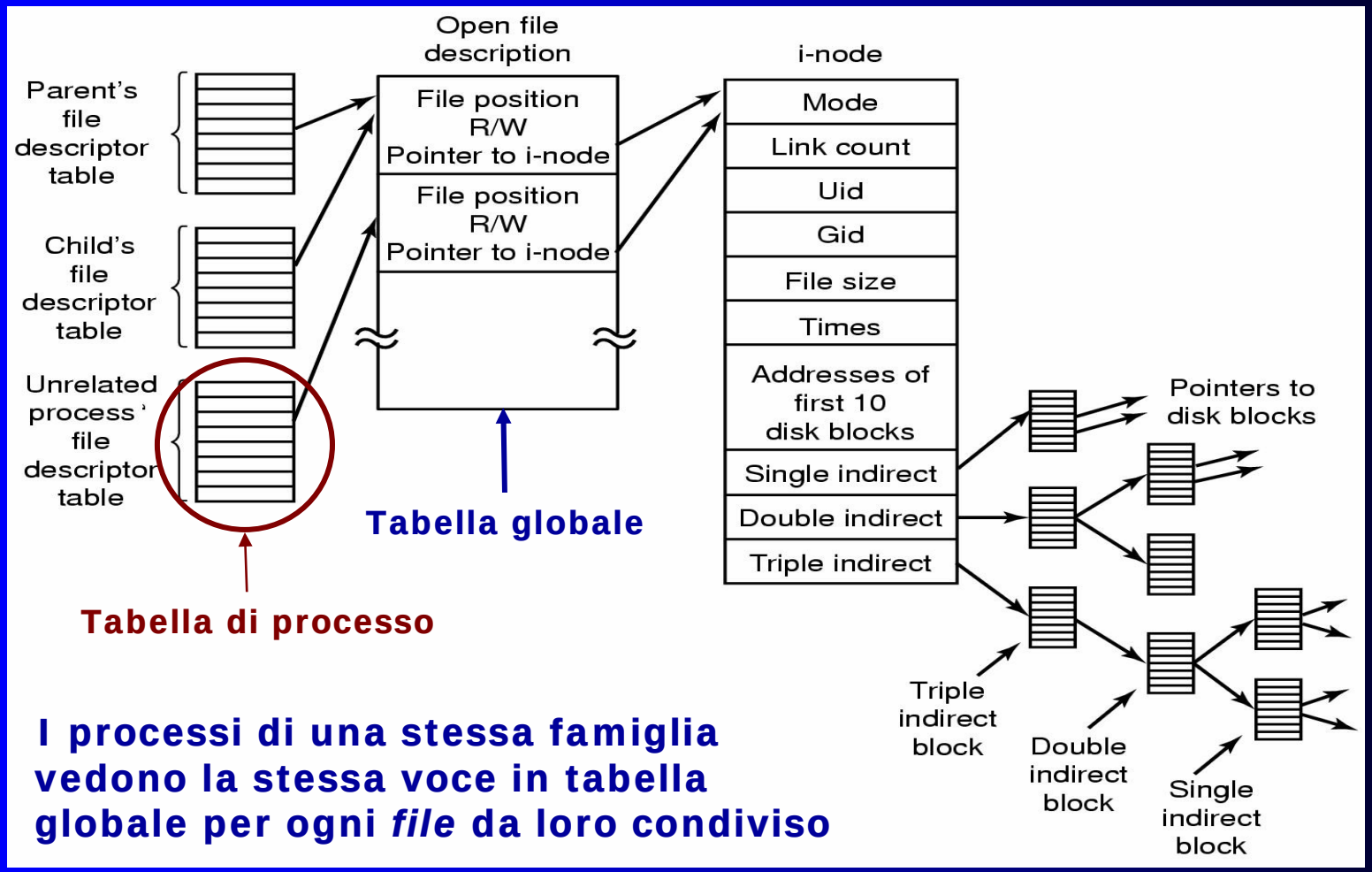
- **Struttura di partizione secondo UNIX v7**
- Il super-blocco (**1**) indica tra l'altro il # di *i-node* e di blocchi nel FS e fornisce il puntatore alla lista dei blocchi liberi (**2**)
- Gli *i-node* (**3**) sono numerati 1..N
  - E tutti sono di ampiezza ≥ 64 B
- *Directory* come insieme **variabile** e **non ordinato** di unità informative (*entry*)
  - Ampie 16 B
    - 14 B (codifica ASCII) per nome di *file*
    - 2 B per numero di *i-node*



Realizzazione del FS in UNIX – 2

- In nucleo usa **due** strutture di controllo
  - Un insieme di **tabelle di processo** contiene “descrittori utente” dei *file* attualmente in uso a ciascun processo
    - A ogni descrittore utente deve corrispondere l’attuale posizione di R/W
    - Però ogni processo deve avere il suo proprio indice di posizione sui propri *file* aperti
      - Possono esistere **più** posizioni di R/W su uno stesso *file* condiviso
      - L’indice **non può** essere ritenuto nell’*i-node* che è **unico** per *file* !
    - Sequenze ordinate di processi figli di uno stesso padre devono poter scrivere su uno stesso *file* consecutivamente
      - Lo **stesso indicatore** di posizione per processi di una **stessa famiglia**
  - Una **tabella globale** mantiene la corrispondenza tra tutti i *file* aperti e i loro *i-node*
    - Ciascuna voce nella **tabella di processo** punta a una voce nella **tabella globale** che specifica diritti e posizione di R/W corrente nel *file*
      - La stessa voce $\forall$ *file* condiviso da processi di una stessa famiglia
      - Voce diversa per stesso *file* per processi non apparentati

Realizzazione del FS in UNIX – 3



I processi di una stessa famiglia vedono la stessa voce in tabella globale per ogni *file* da loro condiviso

Realizzazione del FS in UNIX – 4

- L'*i-node* principale del *file* contiene (tra l'altro) l'indirizzo dei suoi primi 12 blocchi dati
  - L' *i-node* ha la dimensione di 1 frazione di blocco (64 B)
- Per *file* più grandi 1 campo dell'*i-node* principale punta a 1 *i-node* secondario che contiene puntatori ad altri blocchi dati
  - I-*node* principale con campo *single-indirect*
- Per *file* ancora più grandi l'*i-node* secondario contiene puntatori a nodi *single-indirect*
  - I-*node* principale con campo *double-indirect*
- È previsto anche un campo *triple-indirect*

Realizzazione del FS in UNIX – 5

Field	Bytes	Description
Mode	2	File type, protection bits, setuid, setgid bits
Nlinks	2	Number of directory entries pointing to this i-node **(via *hard link*)**
Uid	2	UID of the file owner
Gid	2	GID of the file owner
Size	4	File size in bytes
Addr	39	Address of first 10 disk blocks, then 3 indirect blocks **(3 B / blocco)**
Gen	1	Generation number (incremented every time i-node is reused)
Atime	4	Time the file was last accessed
Mtime	4	Time the file was last modified
Ctime	4	Time the i-node was last changed (except the other times)
	64	

Struttura interna di un *i-node*

Esempio d'uso di *i-node* (UNIX v7)

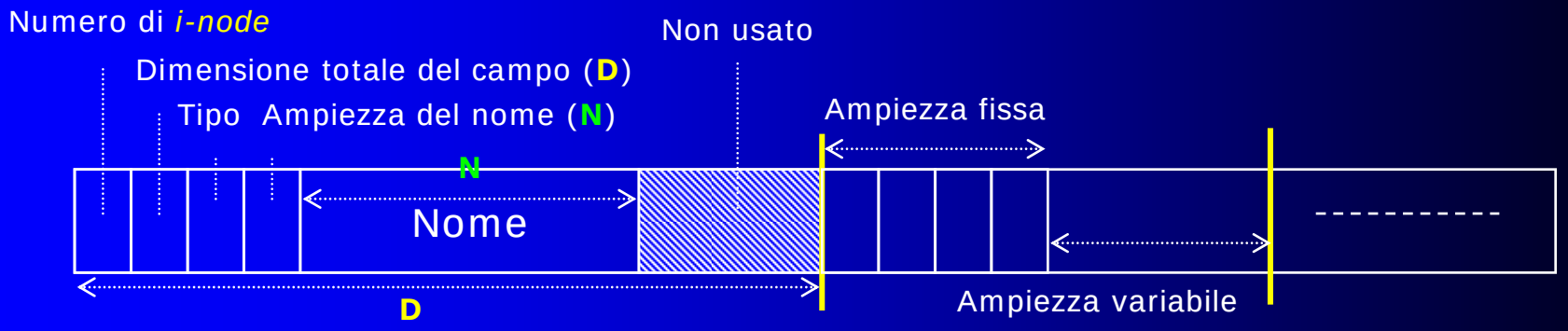
- **Ipotesi A** (le strutture "indirette" sono *i-node*)
  - Blocco dati di capienza 4 KB
  - *i-node* ampio 64 B
  - Indici di blocco espressi su 4 B
- **Esempio 1** (con uso di campo *single-indirect*)
  - Max dimensione di *file* rappresentabile
    - $(10 + 64\ B / 4\ B) \times 4\ KB = (10 + 16) \times 4\ KB = 104\ KB$
- **Esempio 2** (con uso di campo *double-indirect*)
  - Max dimensione di *file* rappresentabile
    - $104\ KB + 16^2 \times 4\ KB = 1\ MB + 104\ KB$
- **Esempio 3** (con uso di campo *triple-indirect*)
  - Max dimensione di *file* rappresentabile
    - $1\ MB + 104\ KB + 16^3 \times 4\ KB = 17\ MB + 104\ KB$

Esempio d'uso di *i-node* (UNIX v7)

- **Ipotesi B** (le strutture "indirette" sono blocchi)
  - Blocco dati di capienza 4 KB
  - *i-node* ampio 64 B
  - Indici di blocco espressi su 4 B
- **Esempio 1** (con uso di campo *single-indirect*)
  - Max dimensione di *file* rappresentabile
    - $(10 + 4\ \text{KB} / 4\ \text{B}) \times 4\ \text{KB} = (10 + 1\ \text{K}) \times 4\ \text{KB} = 4\ \text{MB} + 40\ \text{KB}$
- **Esempio 2** (con uso di campo *double-indirect*)
  - Max dimensione di *file* rappresentabile
    - $(4\ \text{MB} + 40\ \text{KB}) + 1\ \text{K}^2 \times 4\ \text{KB} = 4\ \text{GB} + 4\ \text{MB} + 40\ \text{KB}$
- **Esempio 3** (con uso di campo *triple-indirect*)
  - Max dimensione di *file* rappresentabile
    - $(4\ \text{GB} + 4\ \text{MB} + 40\ \text{KB}) + 1\ \text{K}^3 \times 4\ \text{KB} = 4\ \text{TB} + 4\ \text{GB} + 4\ \text{MB} + 40\ \text{KB}$

Realizzazione del FS in UNIX – 7

- La versione **BSD** introduce alcune migliorie importanti
  - Estensione del nome di *file* fino a 255 caratteri
  - *Directory* di dimensione **multipla** di blocco
    - Facilita e velocizza la scrittura su disco
    - Comporta frammentazione interna



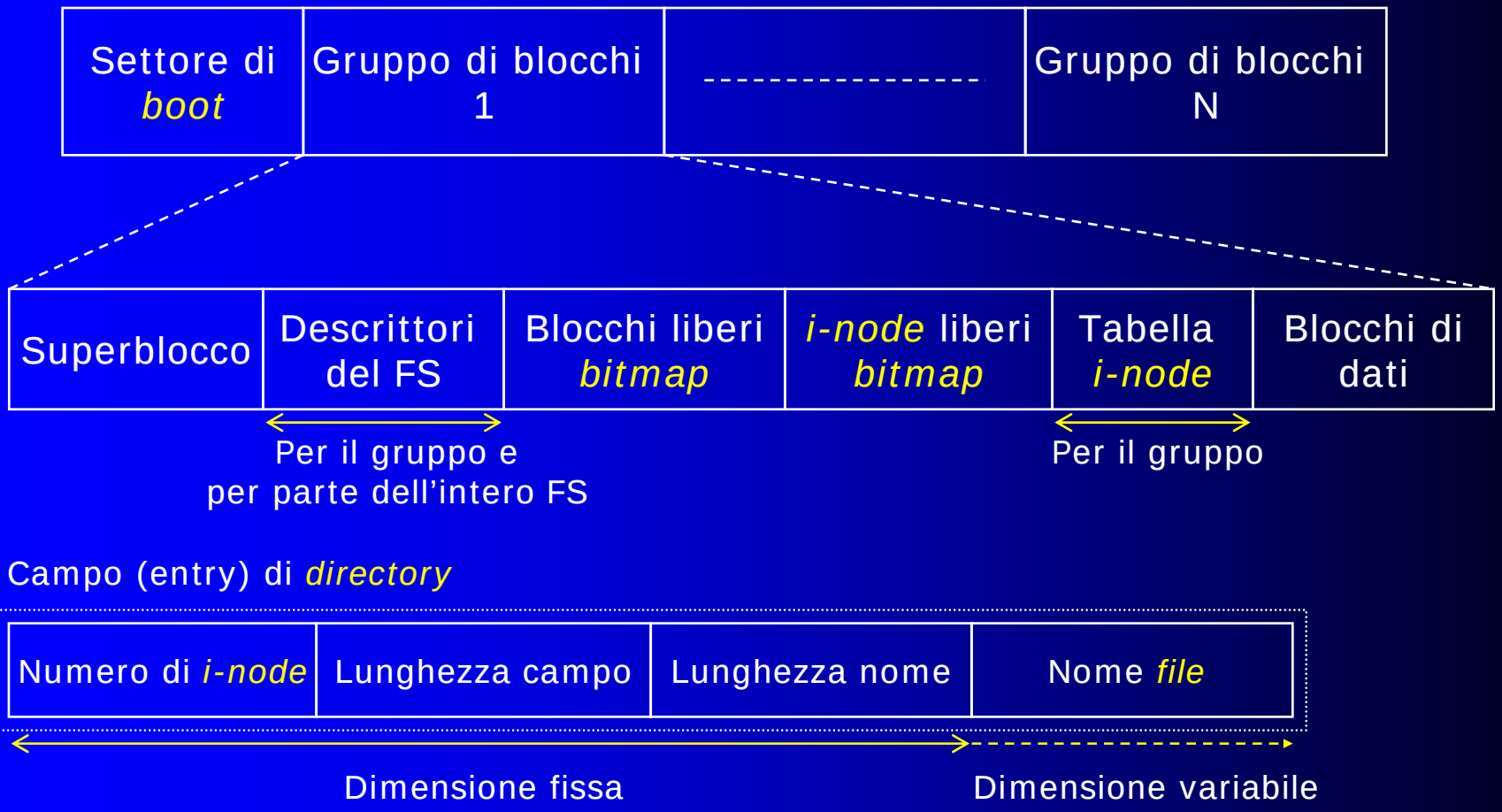
Realizzazione del FS in UNIX – 8

- Altre migliorie **BSD**
  - *Cache* dei nomi di *file* per evitare costosa ricerca lineare su *directory*
  - Disco suddiviso in **gruppi di cilindri**
    - Equivalenti a sotto-partizioni
    - Blocchi dati negli stessi gruppi dei propri *i-node*
    - Oggi di scarso interesse perché i dischi moderni tendono a nascondere al S/O la loro geometria interna
  - 2 ampiezze di blocco
    - Blocchi grandi per *file* molto grandi
    - Blocchi piccoli per *file* piccoli e medi (la norma)
    - Al costo di maggior complessità di gestione

Realizzazione del FS GNU/Linux – 1

- Inizialmente basato sul FS di MINIX
- Subito però abbandonato per le troppe limitazioni
  - Limitazioni MINIX
    - Nomi $\leq$ 14 caratteri
    - Indirizzi di blocco su 2 B per blocchi ampi 1 KB
    - Ampiezza massima di partizione $\leq$ 64 MB
      - 2^{16} blocchi $\times$ 1 KB = 64 K $\times$ 1 KB = 64 MB
- **`ext2`** diviene presto la versione di riferimento
  - Basata sulle scelte BSD con **diversa struttura fisica**
  - La maggiore innovazione è stata la suddivisione della partizione in **gruppi di blocchi**
    - *i-node* e relativi blocchi dati sono tenuti **vicini** sul disco
    - Maggior robustezza ottenuta replicando su ciascun gruppo le informazioni di controllo del superblocco

Realizzazione del FS GNU/Linux – 2



Realizzazione del FS GNU/Linux – 3

- Dimensione di *i-node* estesa a 128 B
  - Indirizzi di blocco ampi 4 B
    - Per denotare fino a 2^{32} = 4 G blocchi
  - Blocchi di dimensione 1, 2, 4 KB scelta al momento della configurazione del FS
    - Partizione di dimensione $\geq$ 4 TB
    - 12 indirizzi diretti + 3 indiretti (*single*, *double*, *triple*)
    - Informazioni di controllo
    - Una parte riservata per uso futuro
- Ogni aggiunta a *file* viene realizzata quanto più **localmente** possibile entro lo stesso gruppo
  - Località **tra** *file* correlati tramite gruppi
  - Località **entro** *file* mediante preallocazione di N $\leq$ 8 blocchi contigui

Realizzazione del FS GNU/Linux – 4

- Una *directory* `/proc` **virtuale** (non esistente su disco) contiene una *directory* $\forall$ processo presente nel sistema
  - Il nome della *directory* foglia è il PID del processo
  - Il contenuto della *directory* foglia è un insieme di *file* che descrivono il processo e il suo ambiente
    - L'informazione originale resta nel nucleo da dove essa viene estratta alla lettura del *file* virtuale corrispondente
- L'accesso di utente al FS viene filtrato da un **FS virtuale** che consente la coesistenza di più FS di tipo diverso (p. es.: FAT-32 insieme con `ext2`)
  - Modalità sostanzialmente analoga a NFS (vedi seguito)

Evoluzioni del FS in GNU/Linux

- **Ext3** (2001)
  - Maggior garanzia di consistenza del FS grazie all'uso del *journaling*
    - Archivio delle modifiche al FS salvato su disco prima della loro effettuazione
      - Persistenza anche in caso di interruzioni di alimentazione
- **ReiserFS** (2001)
  - *Directory* strutturata a *B+tree* per maggior efficienza d'uso
  - *Journaling* semplificato
    - Ricorda solo "metadati"
      - Le azioni di preparazione aggiornamento ma non i dati
- **Reiser4** (2007)
  - Lo spazio lasciato libero da frammentazione interna può essere usato per memorizzare *file* piccoli
    - Maggior capienza effettiva quando si usano blocchi medio-grandi

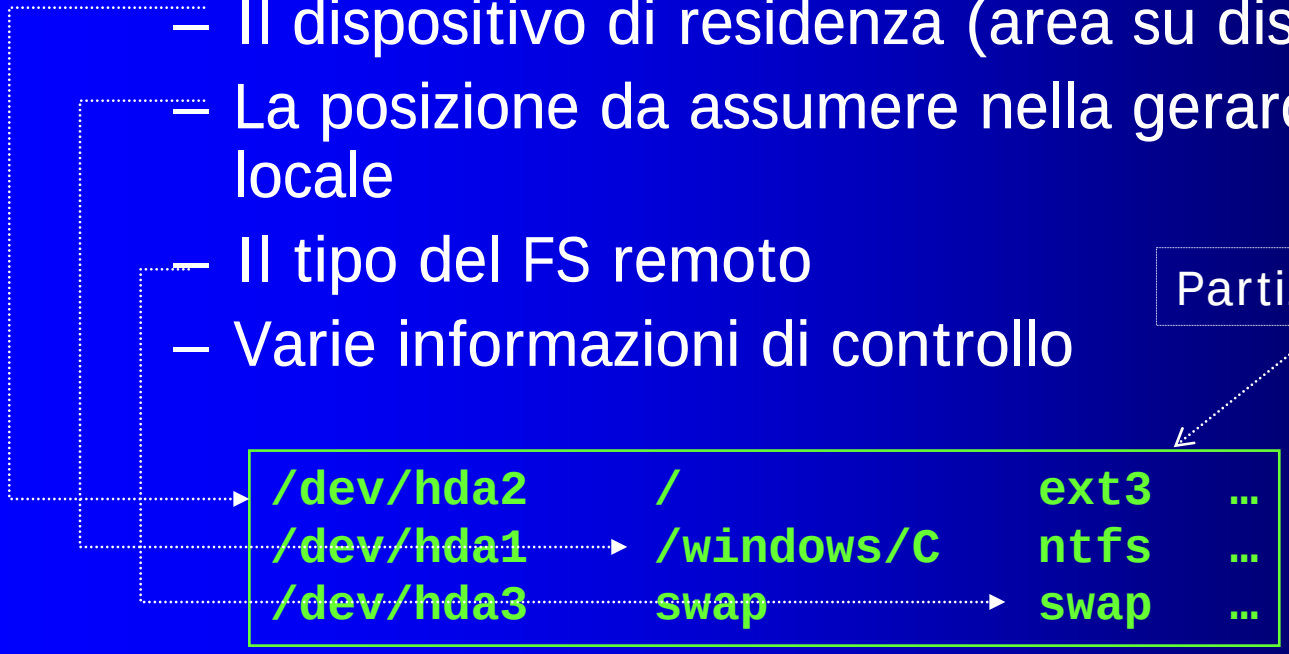
File System di rete (NFS) – 1

- NFS consente a un insieme arbitrario di utenti remoti di condividere uno stesso FS_r
- FS_r viene ospitato su un *server* che i clienti possono contattare
  - Clienti remoti
    - Posti su rete locale o geografica ed eterogenea
  - Clienti locali
    - Posti sullo stesso nodo del *server*
- Il *server* esporta FS_r come sottoalbero del proprio FS_p locale indicandone la "radice"
  - La lista delle "radici" esportate dal nodo viene posta nel *file* di configurazione **`/etc/exports`**
- Il cliente importa (**`mount`**) FS_r posizionandolo come un sottoalbero del proprio FS_c
  - La posizione della "radice" di FS_r è detta *mount point*

File System di rete (NFS) – 2

- Il file **`/etc/fstab`** di ciascun cliente fornisce la lista dei FS importabili mediante **`mount`**
- Per ciascun FS remoto si indicano
  - Il dispositivo di residenza (area su disco)
  - La posizione da assumere nella gerarchia del FS locale
  - Il tipo del FS remoto
  - Varie informazioni di controllo

Partizioni su unico disco!



File System di rete (NFS) – 3

- NFS definisce i protocolli che regolano il dialogo tra il *server* e i suoi clienti
  - **Protocollo di importazione di FS remoto** (`mount`)
    - Il cliente invia al *server* il nome della "radice" del FS importato
    - Se la richiesta ha successo il *server* invia al cliente un descrittore unicamente associato al FS esportato
      - Tipo di FS, disco di residenza, informazioni di controllo, numero di *i-node* della *directory* radice
    - Ogni accesso del cliente a *file* del FS importato userà quel descrittore
    - 2 modalità di importazione
      - **Esplicita**
        - Per inizializzazione eseguita dallo *script* `/etc/rc`
      - **Automatica**
        - Al riferimento a *file* residenti in FS importato

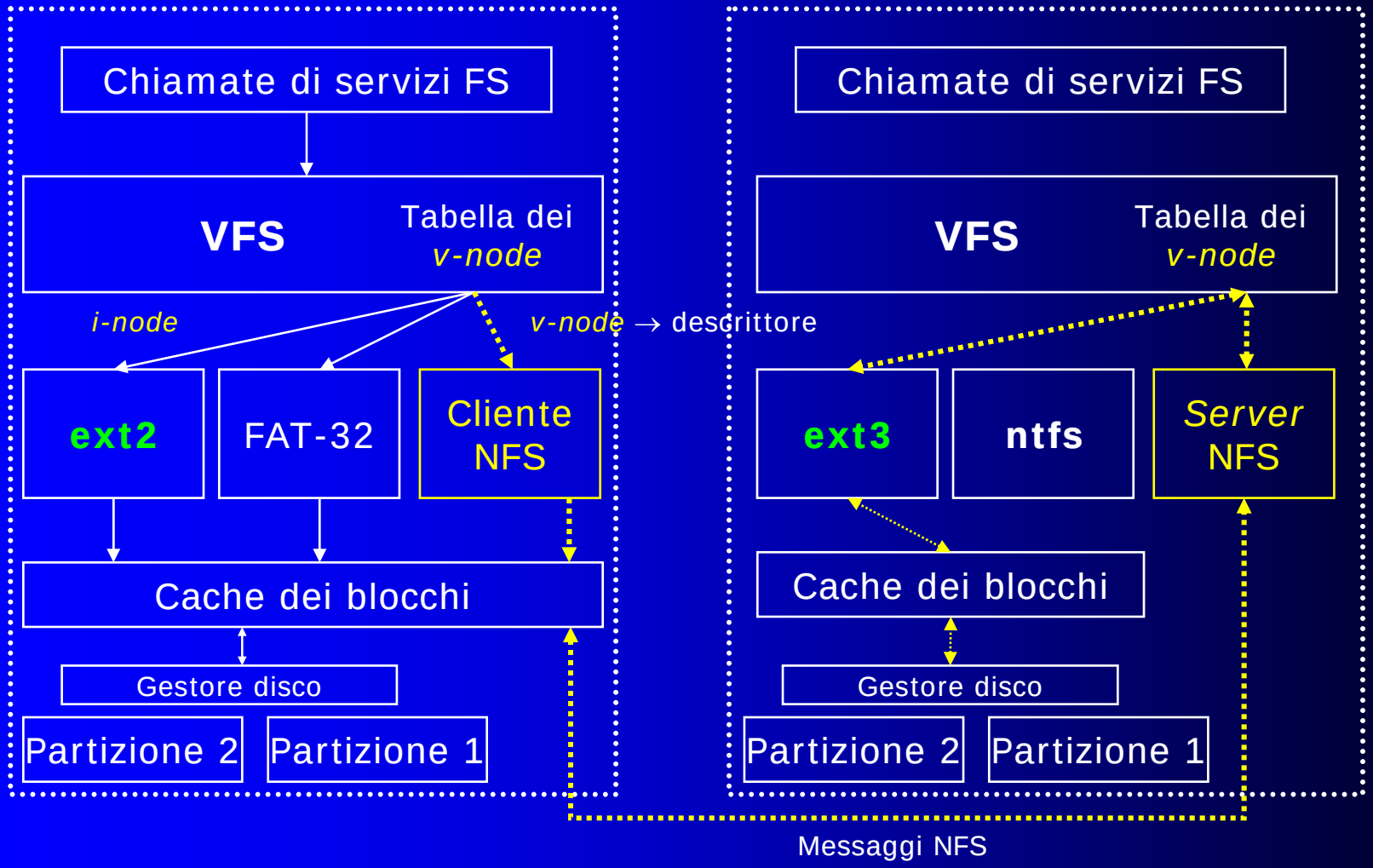
File System di rete (NFS) – 4

- **Protocollo di accesso a *file* remoti**
  - Il *server* non mantiene informazioni di stato (*stateless*)
  - Le richieste del cliente sono messaggi contenenti il descrittore del *file* remoto e i parametri dell'operazione richiesta
  - Il *server* ha compito semplice ma a rischio di inconsistenze
    - Lo stato di un *file* può cambiare tra due accessi remoti successivi
- Il controllo di accesso a *file* usa semplicemente diritti di tipo *owner*, *group*, *others*
  - Facilmente falsificabili se non autenticati
- **Nessun** supporto previsto per *lock*

File System di rete (NFS) – 5

- Un livello di FS virtuale (**VFS**) filtra ogni richiesta di accesso ai FS localmente visibili a ogni cliente
- VFS mantiene un ***v-node*** per ogni *file* aperto al quale associa un *i-node* (per *file* locali) oppure un ***r-node*** (per *file* remoti)
  - All'***r-node*** viene associato il descrittore di *file* remoto fornito dal *server* che ne esporta il FS
  - Il traffico di rete viene ridotto trasferendo dati tra cliente e *server* in unità R/W da 8 KB e istituendo 2 *cache* presso il cliente per dati di *file* e riferimenti (*i-node*)
    - Le informazioni in *cache* hanno validità limitata
      - Regolata da un *timer* fissato a 3 s. per blocchi dati e 30 s. per blocchi di *directory*
    - Modalità *read-ahead*
      - Sempre 8 KB in più sull'ultima lettura
    - Scrittura differita al riempimento dell'unità di trasferimento

File System di rete (NFS) – 6



Genesi – 1

- **MS-DOS**
  - Mono-utente in modalità *command line*
  - **Non multi-programmato**
  - Inizialmente ispirato a CP/M
    - 1981 : **1.0** (8 KB) → PC IBM 8088 (16 *bit*)
    - 1986 : **3.0** (36 KB) → PC IBM/AT (i286 @ 8 MHz, ≤ 16 MB)
- **Windows** 1ª generazione
  - Modalità GUI solo come rivestimento di MS-DOS
  - Interfaccia **copia** del 1º modello Macintosh di Apple
    - 1990 - 1993 : **3.0**, **3.1**, **3.11** → i386 (32 *bit*)

GUI

- **GUI** (*Graphical User Interface*)
  - Introdotto dal modello **Macintosh** di Apple il 24 gennaio 1984
    - Vedi http://www.apple-history.com/lisa.html
  - Basato sul paradigma **WIMP** (dispregiativo!)
    - Finestre (*windows*), icone (*icons*), *m*enu e dispositivi di puntamento (*pointing*)
- Realizzabile
  - Sia come programma in spazio utente (GNU/Linux)
  - Che come parte del S/O (Windows)

Genesi – 2

- **Windows** 2ª generazione
  - Vero e proprio S/O **multiprogrammato** ma sempre **mono-utente** con FS su modello FAT
    - 1995 : Windows 95 (MS-DOS 7.0)
    - 1998 : Windows 98 (MS-DOS 7.1)
      - Nucleo a procedure **non rientranti**
        - Incapaci di consentire più esecuzioni simultanee
        - Ogni accesso a nucleo protetto da semaforo a mutua esclusione
        - Scarsissimo beneficio da multiprogrammazione
      - ¼ dello spazio di indirizzamento di processo (4 GB totali) condiviso R/W con gli altri processi; ¼ condiviso R/W con il nucleo
        - Scarsissima integrità dei dati critici
    - 2000 : Windows Me (ancora MS-DOS)

Modeste modifiche

Genesi – 3

- **Windows** 3ª generazione
  - Progetto **NT**: abbandono della base MS-DOS
    - Architettura a 16 *bit*
  - Enfasi su sicurezza e affidabilità
  - FS di nuova concezione (**NTFS**)
    - 1993 : Windows NT 3.1 → fiasco commerciale per la mancanza di programmi di utilità
    - 1996 : Windows NT 4.0 → reintroduzione di interfaccia e programmi Windows 95
      - Scritto in C e C++ per massima portabilità al costo di grande complessità (16 M linee di codice!)
      - Molto superiore a Windows 95/98 ma privo di supporto per *plug-and-play* gestione batterie e emulatore MS-DOS

Genesi – 4

- **Windows** 3ª generazione (segue)
  - **Architettura di NT 3.1** a ***microkernel*** e modello ***client-server***
    - La maggior parte dei servizi è incapsulata in processi di sistema eseguiti in modo utente e offerti ai processi applicativi tramite scambio messaggi
  - **Elevata portabilità**
    - Dipendenze *hardware* localizzate nel nucleo
  - **Bassa velocità**
    - Più costosa l'esecuzione in modo privilegiato
  - **Architettura di NT 4.0** a **nucleo monolitico**
    - Servizi di sistema riposizionati entro il nucleo

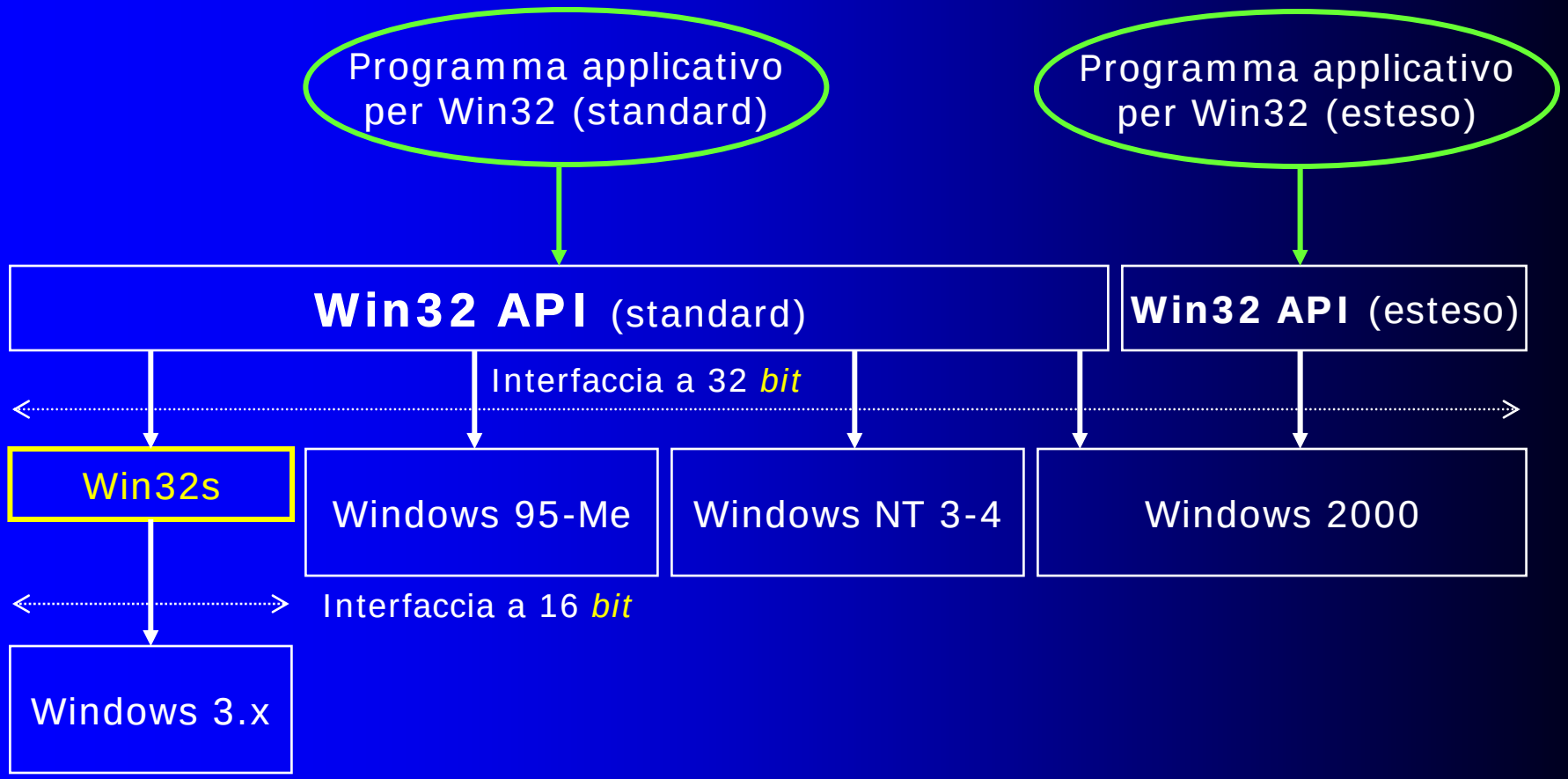
Genesi – 5

- **Windows** 3ª generazione (segue)
  - 1999 : Windows 2000 (alias di **NT 5.0**)
    - Il S/O esegue in **modo nucleo**
      - Lo spazio di indirizzamento dei processi è interamente privato e distinto dal modo nucleo
        - **Memoria virtuale**
    - Periferiche rimuovibili
      - *Plug-and-play*
    - Internazionalizzazione
      - **Unica** versione configurabile per lingua nazionale
    - Alcune migliorie a **NTFS**
    - MS-DOS completamente rimpiazzato da una *shell* di comandi che ne riproduce e estende le funzionalità
    - Enorme complessità: oltre 29 M linee di codice C[++]

Interfaccia di programmazione – 1

- Basato su principio speculare a quello adottato da UNIX e GNU/Linux
  - Interfaccia di sistema **non** pubblica
  - Procedure di libreria pubblicate in **Win32 API** (***Application Programming Interface***) a uso del programmatore ma controllata da Microsoft
  - Alcune procedure includono chiamate di sistema
  - Altre svolgono servizi di utilità eseguiti interamente in modo utente
    - Nessun sforzo di evitare ridondanza o rigore gerarchico

Interfaccia di programmazione – 2



Informazioni di configurazione

- Tutte le informazioni vitali di configurazione del sistema sono raccolte in una specie di FS detto **`registry`** salvato su disco in *file* speciali detti **`hives`**
  - *Directory* → *key*
  - *File* → *entry* = {nome, tipo, dati}
- 6 *directory* principali con prefisso **`HKEY_`**
  - Per esempio: **`HKEY_LOCAL_MACHINE`** con *entry* descrittive dell'*hardware* e delle sue periferiche (**`HARDWARE`**) dei programmi installati (**`SOFTWARE`**) e con informazioni utili per l'inizializzazione (**`SYSTEM`**)

Architettura di sistema – 1

- Sistema su 2 livelli gerarchici
  - **Nucleo monolitico** i cui componenti eseguono tutti in modo privilegiato
    - Dipendenze dalla scheda madre dello specifico elaboratore isolate in un livello detto **HAL** (*hardware abstraction layer*)
      - Insieme **standard** di servizi di accesso a registri, indirizzi di periferiche, vettore delle interruzioni, orologi, BIOS
      - Poi affiancato da un interfaccia di maggior potenza e velocità detto **DirectX**
  - **Sottosistemi d'ambiente** visti come processi che eseguono in modo normale

Architettura di sistema – 2

- Su **HAL** poggia un livello detto **kernel** che eleva il livello di astrazione dei servizi **HAL**
  - **Gestione della concorrenza**
    - Ordinamento, prerilascio, salvataggio e ripristino dei contesti
  - **Gestione degli "oggetti di controllo"** associati alle entità attive del sistema
    - Processi e servizi associati alle interruzioni
    - Oggetto ***Deferred Procedure Call***: la parte meno urgente di un servizio di interruzione, che esegue in modo nucleo e non è interrompibile da *thread* e APC
      - Gestione dei dispositivi, dell'orologio di sistema, della fine quanto
    - Oggetto ***Asynchronous Procedure Call***: la parte immediate di un servizio di interruzione, che esegue entro un *thread* sia in modo nucleo che normale
  - **Gestione degli "oggetti di ordinamento"** associati a entità passive come semafori, eventi, orologi
    - Usati dalle entità attive per sincronizzarsi tra loro

Architettura di sistema – 3

- Il livello **executive** (il più alto del S/O) è suddiviso in 10 aggregati di **procedure** funzionalmente correlate

Object manager: gestisce gli "oggetti" creati dal S/O allocando loro memoria virtuale e fissando le loro operazioni (1)
I/O manager: gestisce i dispositivi incluse le partizioni di disco
Process manager: gestisce le entità concorrenti del sistema
Memory manager: gestisce la memoria virtuale con modalità "*paging-on-demand*"
Cache manager: gestisce in RAM una *cache* di blocchi di disco

Architettura di sistema – 4

- Solo Ⓐ e Ⓑ sono componenti attive
- Tutte però eseguono in modo nucleo

Plug-and-play manager (**A**): viene informato delle periferiche connesse al sistema e le associa il loro gestore (2)

Power manager (**B**): cerca di contenere il consumo energetico del sistema

Configuration manager: gestisce la **registry**

Security manager: si occupa dell'esecuzione delle politiche di sicurezza richiesti per applicazioni riservate

Local procedure call manager: fornisce meccanismi efficaci per la comunicazione tra le componenti attive del sistema

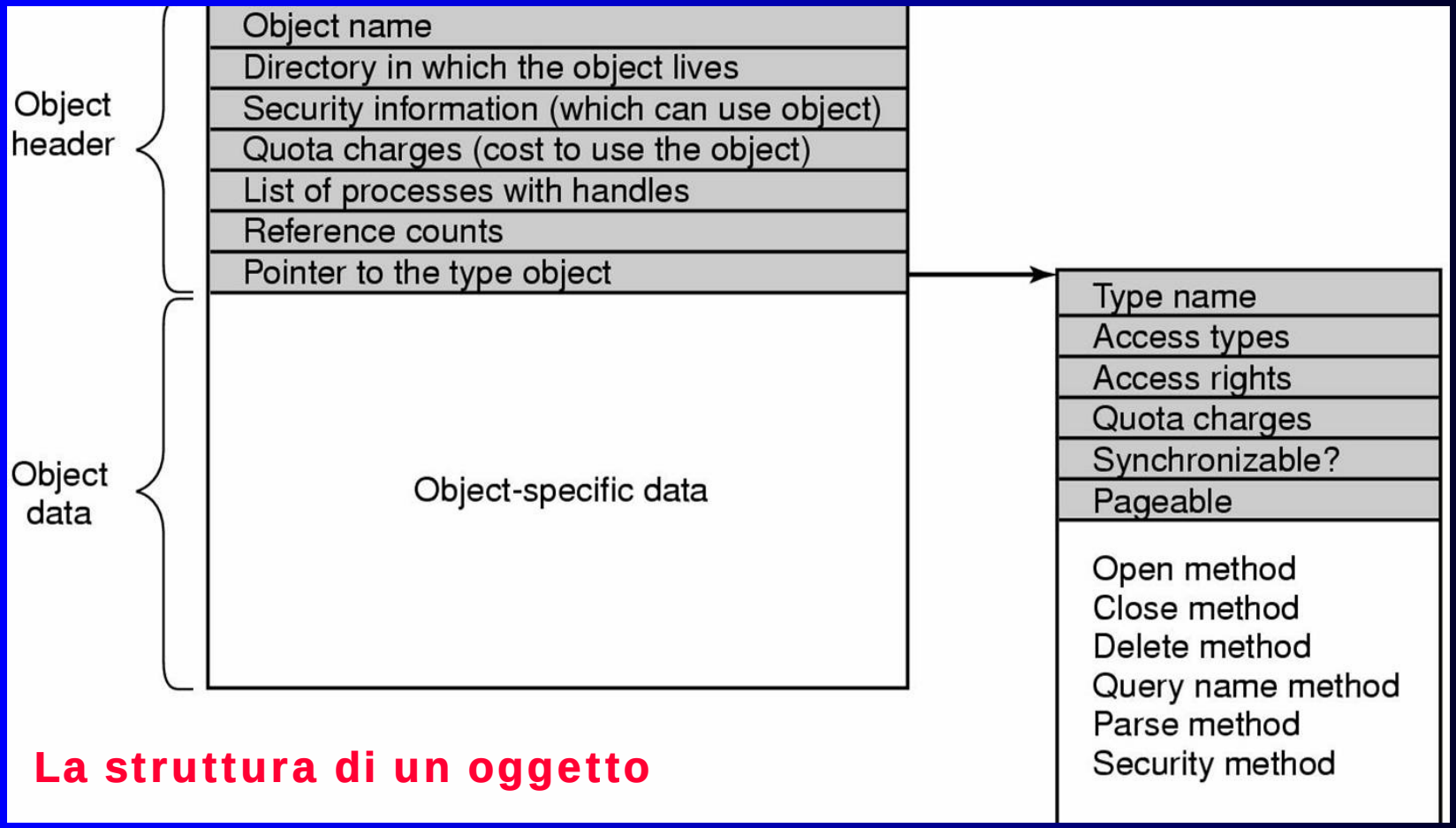
Architettura di sistema – 5

- Del livello **executive** fa parte anche il **GDI** che in in NT 3.x era posto in spazio di utente
  - *Graphics Design Interface*
  - Di gran lunga la componente più grande
  - Posto in modo nucleo da NT 4.0 per migliorare le prestazioni
    - La sua esecuzione può comportare frequente paginazione
- **kernel** ed **executive** sono raccolti in un unico **eseguibile** (**ntoskrnl.exe**)
- **HAL** viene invece fornito come **libreria condivisa** raccolta in un unico *file* (**hal.dll**)
- Gestori delle periferiche caricati **dinamicamente** e registrati in **registry** dal **Configuration manager**

Architettura di sistema – 6

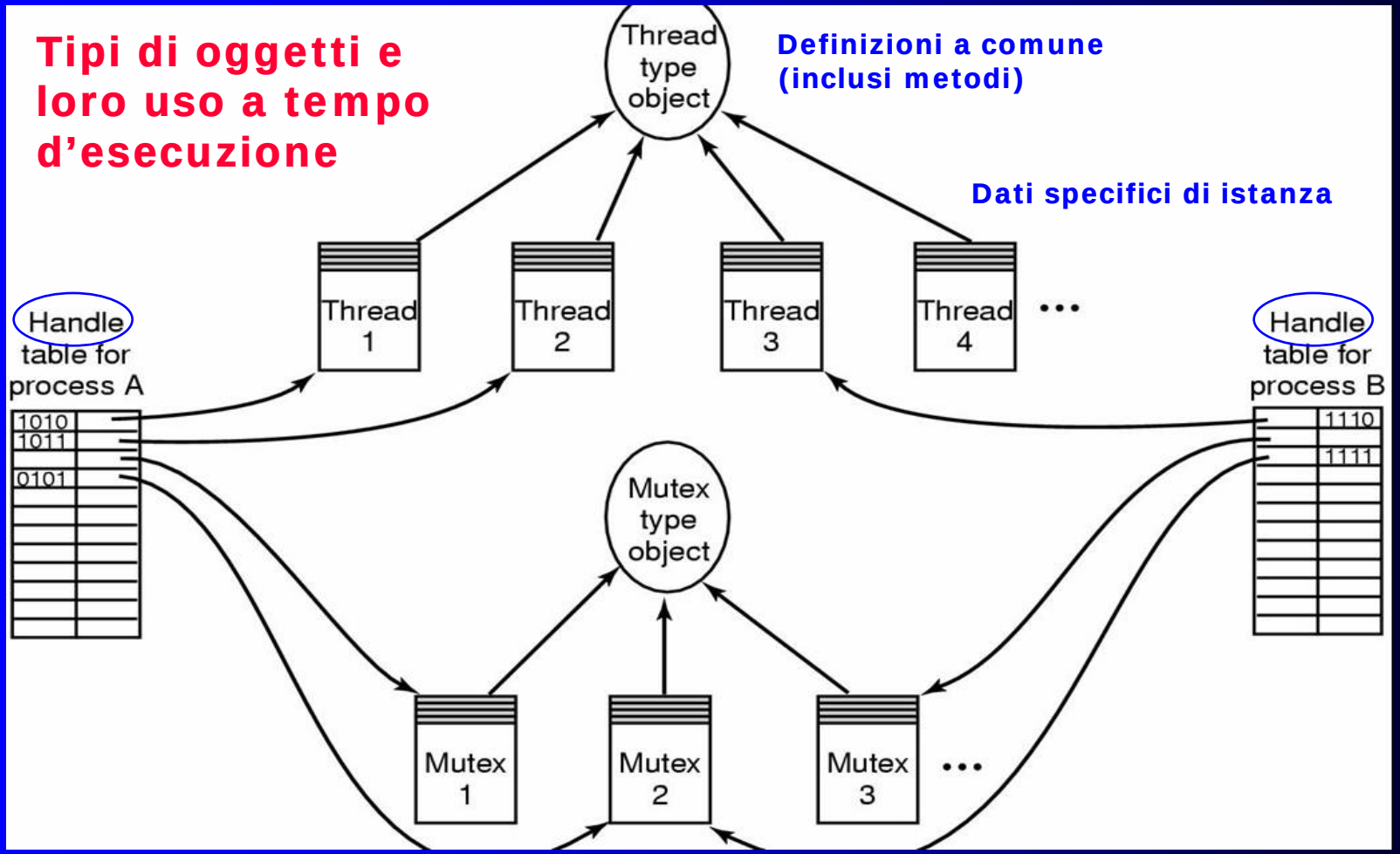
- Durante l'esecuzione il sistema crea, manipola e distrugge **oggetti interni** nessuno dei quali permane tra due accensioni successive
  - Un oggetto per ogni risorsa logica o fisica (entità attive o passive)
    - Tutti gli oggetti hanno alcuni metodi comuni
    - I loro tipi corrispondono alle **interfacce** dei linguaggi OO
  - Gli oggetti sono **descrittori** (residenti in RAM) delle entità logiche o fisiche corrispondenti
    - Alcuni oggetti possono essere temporaneamente posti su disco
    - I processi manipolano oggetti tramite riferimenti detti *handle*
- Il **`kernel`** mantiene una **tabella degli oggetti**
  - 29 *bit* per puntatore all'oggetto + 3 *bit* come *flag*
  - 32 *bit* per i diritti associati alle operazioni sull'oggetto
- L'**`Object manager`** suddivide gli oggetti in categorie (*directory*) specifiche

Architettura di sistema – 7



La struttura di un oggetto

Architettura di sistema – 8



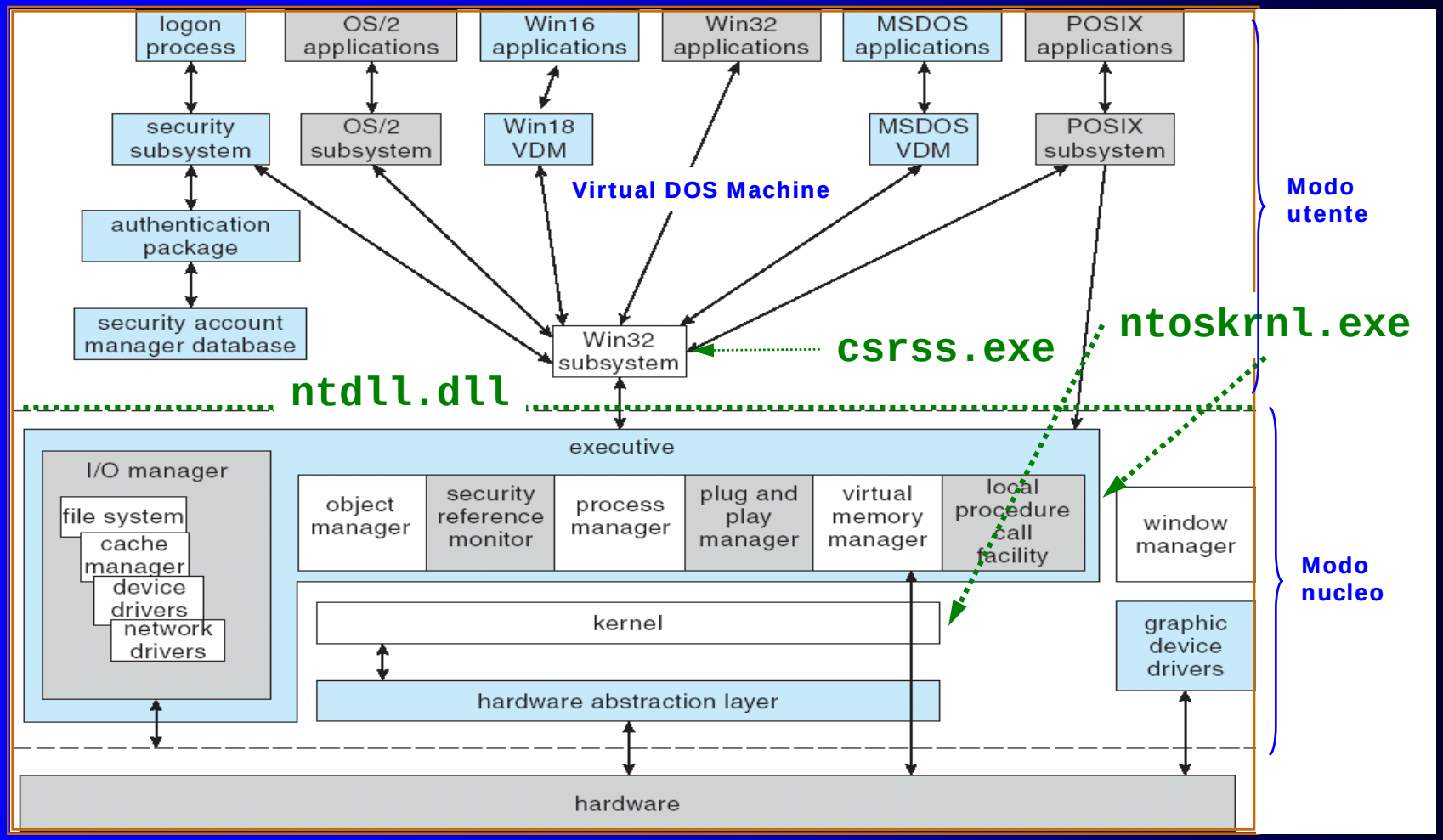
Architettura di sistema – 9

- In spazio di utente sono disponibili 3 ulteriori categorie di componenti di sistema
  - **`DLL`** (***Dynamic Link Libraries***) che raccolgono specifiche procedure di libreria in gruppi visibili ai e condivisi dai vari programmi
    - Ogni processo utente include **chiamate parametriche** a specifici **`DLL`** **al posto** del codice delle procedure richieste
  - **Sottosistemi d'ambiente** (**`.exe`**) che forniscono ciascuno uno specifico interfaccia di programmazione
    - Il principale è Win32 API, **`csrss.exe`**
      - *Client-server run-time subsystem*
    - Gli altri 2 (uno era per UNIX/POSIX) sono stati a lungo inutilizzati
    - Con Windows XP sono diventati elementi importanti della versione *server* del S/O
  - **Processi di servizio**

Architettura di sistema – 10

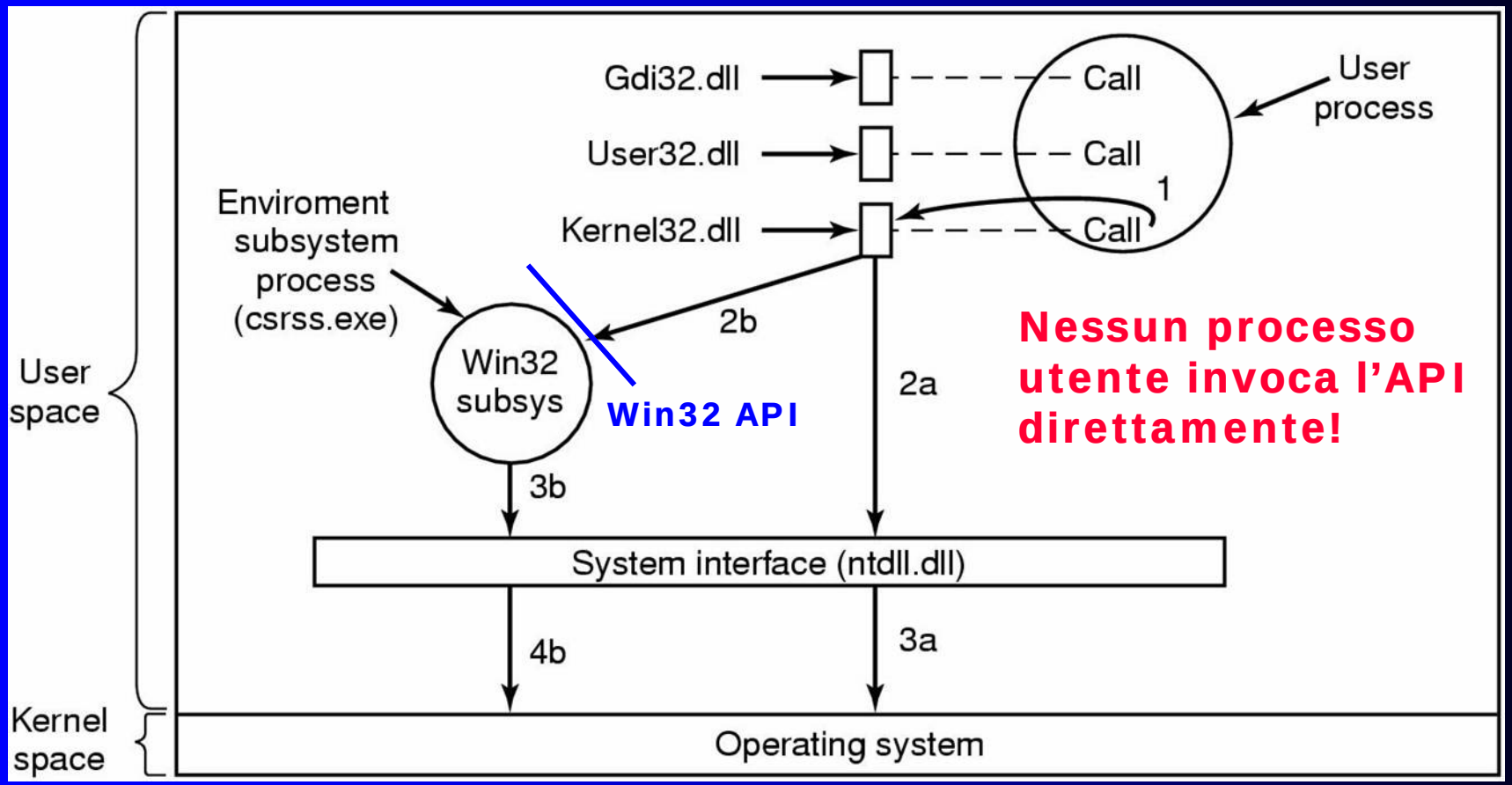
- Oltre 800 **DLL** complessivi per oltre 13.000 procedure invocabili dai processi utente
  - **user32.dll**
    - Invocate in **modo utente** per i servizi GUI
  - **gdi32.dll**
    - Invocate in **modo utente** per i servizi grafici di livello inferiore al GUI
  - **kernel32.dll**
    - Invocate in **modo utente** per tutti gli altri servizi
  - **ntdll.dll**
    - Il vero interfaccia di sistema tra modo utente e modo nucleo (**executive** e **kernel**)
  - **hal.dll**
    - Eseguite in modo nucleo per accedere all'*hardware* specifico dell'elaboratore

Architettura di sistema – 11



Silberschatz, Galvin and Gagne ©2005

Architettura di sistema – 12



Gestione dei processi – 1

- *Job* = {processi gestiti come singola unità}
- Processo = possessore di risorse, con ≥ 1 *thread*

  ID unico, 4 GB di spazio di indirizzamento (2 in modo utente e 2 in modo nucleo), inizialmente con singolo *thread*, simile al processo UNIX; **non** ha stato di avanzamento

- *Thread* = flusso di controllo gestito dal nucleo

  Esegue per conto e nell'ambiente del processo (che **non** ha stato di avanzamento), con ID **localmente** unico, 2 *stack* (1 per modo)

- *Fiber* = suddivisione di *thread* ignota al nucleo

  Esegue nell'ambiente del *thread* e viene gestita interamente a livello di servizi offerti dal sottosistema **Win32**

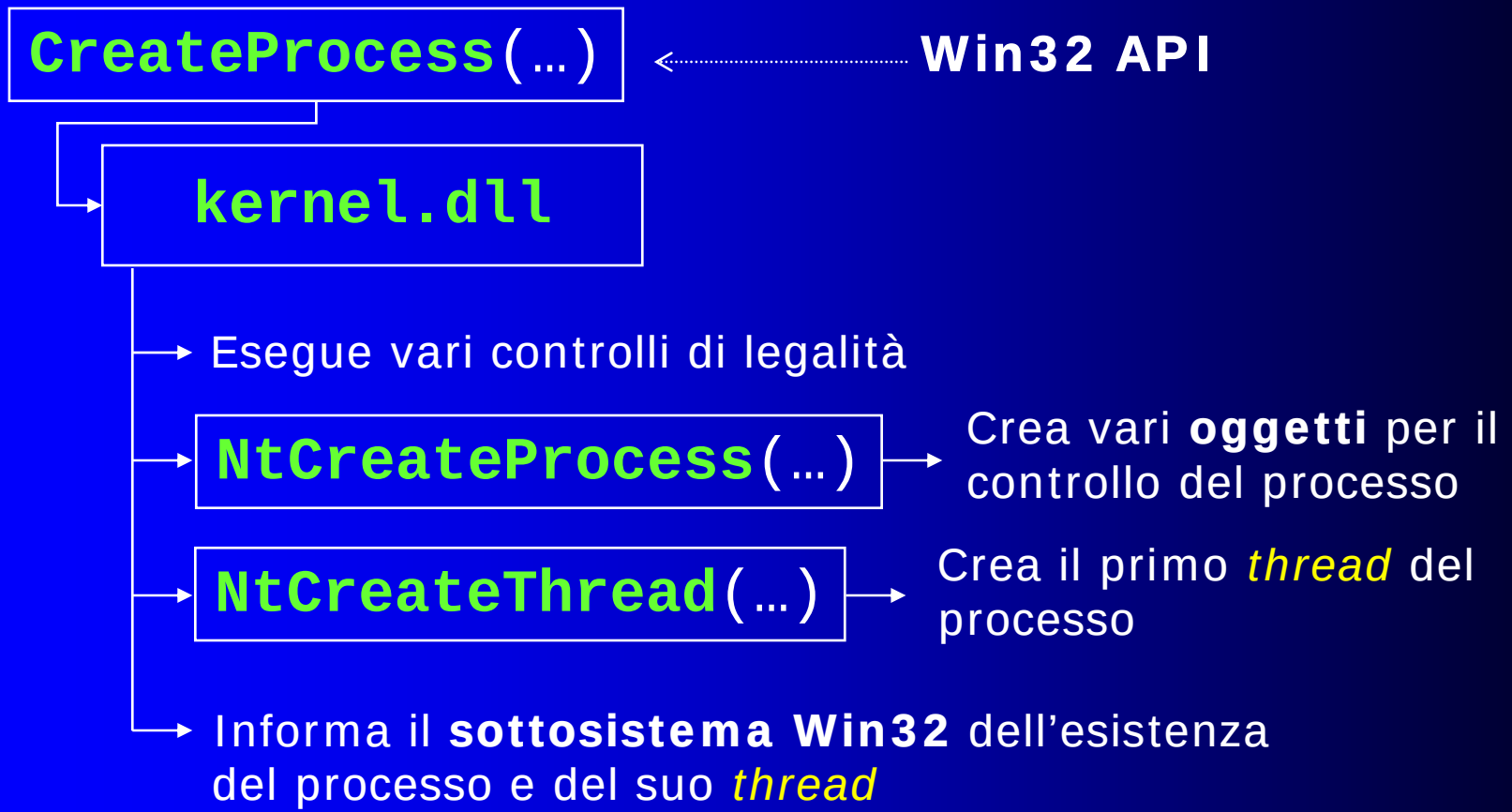
Gestione dei processi – 2

- I *thread* hanno vari modi per sincronizzarsi tra loro tramite **oggetti di ordinamento**
  - **Semafori binari** (`mutex`) o contatori
  - **Sezioni critiche** limitate allo spazio di indirizzamento del *thread* che le crea
  - **Eventi** di 2 tipi
    - A *reset* manuale che rilascia più *thread* sino ad un esplicito *reset* che cancella l'evento
    - A *reset* automatico che rilascia solo un *thread* e poi cancella l'evento

Gestione dei processi – 3

- I *thread* hanno vari modi per comunicare senza bisogno di sincronizzarsi
  - *Pipe* : canali bidirezionali come in UNIX e GNU/Linux a sequenza di *byte* **senza** struttura oppure per messaggi (sequenze **con** struttura)
  - *Mailslot* : canali unidirezionali anche su rete
  - *Socket* : come *pipe* ma per comunicazioni remote
  - **RPC** (**chiamata di procedura remota**) : per invocare procedure nello spazio di altri processi e riceverne il risultato localmente
  - **Condivisione di memoria** : usando (porzioni di) *file* mappati in memoria
- Modi usati soprattutto per le interazioni tra le applicazioni e GDI

Gestione dei processi – 4



Politica di ordinamento – 1

- **Ordinamento con prerilascio a priorità**
  - Effettuato da azioni **esplicite** del *thread* eseguite in modo nucleo → non a carico di alcuna entità attiva dedicata di sistema
    - Nel sospendersi in attesa di una risorsa occupata o nell'inviare un segnale di sincronizzazione
      - L'esecuzione è già in modo nucleo
    - Al completamento del proprio quanto di tempo
      - L'esecuzione passa in modo nucleo tramite un **DPC**
  - Oppure causato da attività esterne eseguite nel contesto del *thread* corrente
    - Azioni di ordinamento programmate come **DPC** associate al trattamento di eventi asincroni (interruzione, allarme *time-out*) possono rilasciare *thread*

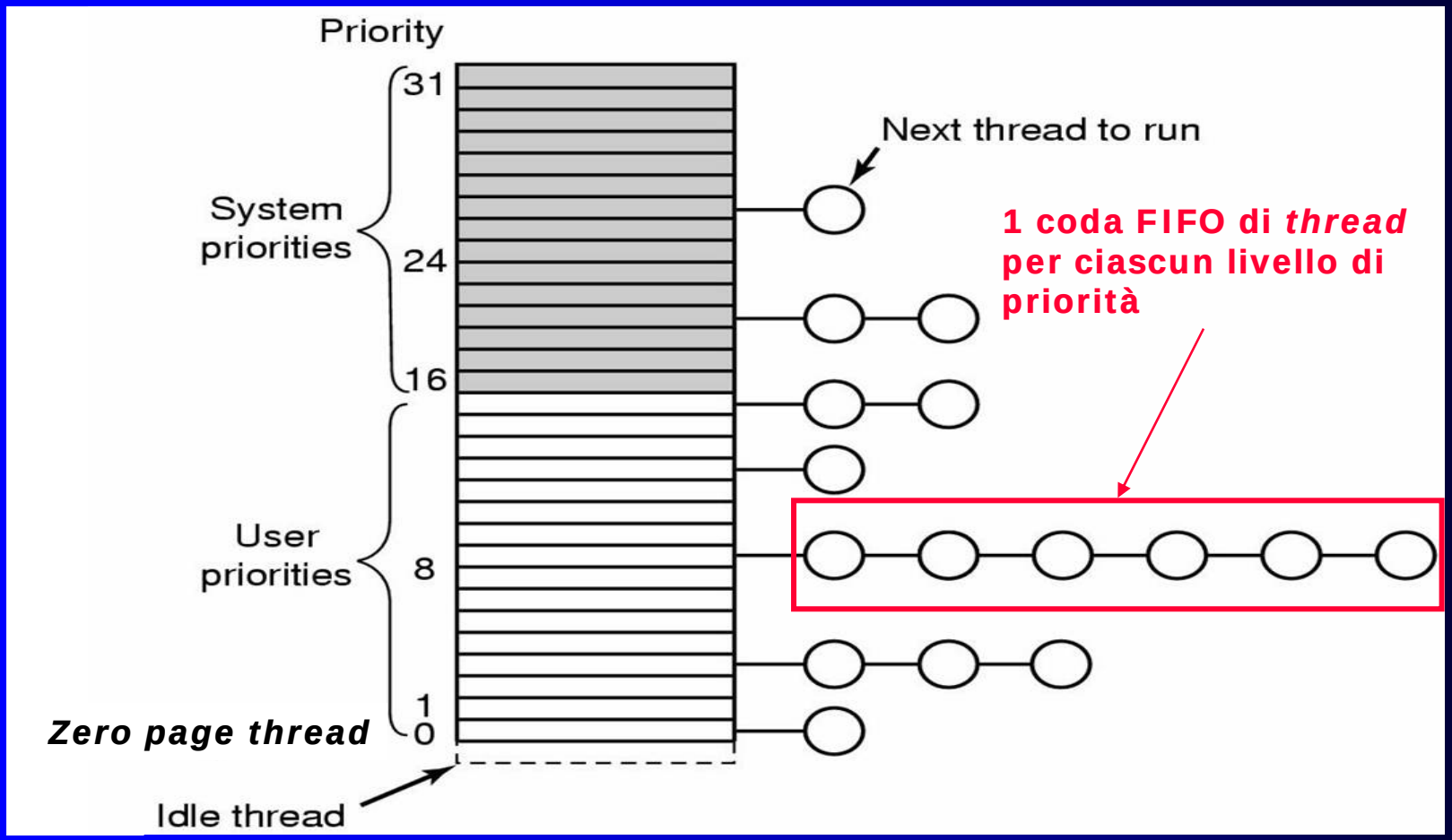
Politica di ordinamento – 2

- **6 classi di priorità per processo**
  - Realtime, high, above-normal, normal, below-normal, idle
- **7 classi di priorità per *thread***
  - Time-critical, highest, above-normal, normal, below-normal, lowest, idle
- **32 livelli di priorità (31 .. 0)**
  - Ciascuno associato a una coda di *thread* pronti
  - *Thread* **non** distinti per processo di appartenenza
    - 31 .. 16 priorità di sistema
    - 15 .. 0 priorità ordinarie
- Ricerca per priorità decrescente
- Selezione dalla testa della coda

Politica di ordinamento – 3

		Win32 process class priorities					
		Realtime	High	Above Normal	Normal	Below Normal	Idle
Win32 thread priorities	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

Politica di ordinamento – 4



Politica di ordinamento – 5

- Ciascun *thread* ha una priorità **base** iniziale e una priorità **corrente** che **varia** nel corso dell'esecuzione
  - Entro la fascia della classe di priorità del processo di appartenenza
- La priorità corrente si eleva quando il *thread*
  - Completa un'operazione di I/O
    - Per favorire maggior utilizzazione delle periferiche
    - Insieme a un ampliamento temporaneo della durata del quanto
  - Ottiene un semaforo o riceve un segnale d'evento
    - Per ridurre il tempo di attesa dei processi interattivi
- La priorità corrente decresce a ogni quanto consumato
- Usa una tecnica brutale per mitigare il problema di inversione di priorità
  - Un *thread* pronto non selezionato per un certo tempo riceve un incremento di priorità per 2 quanti

Inizializzazione – 1

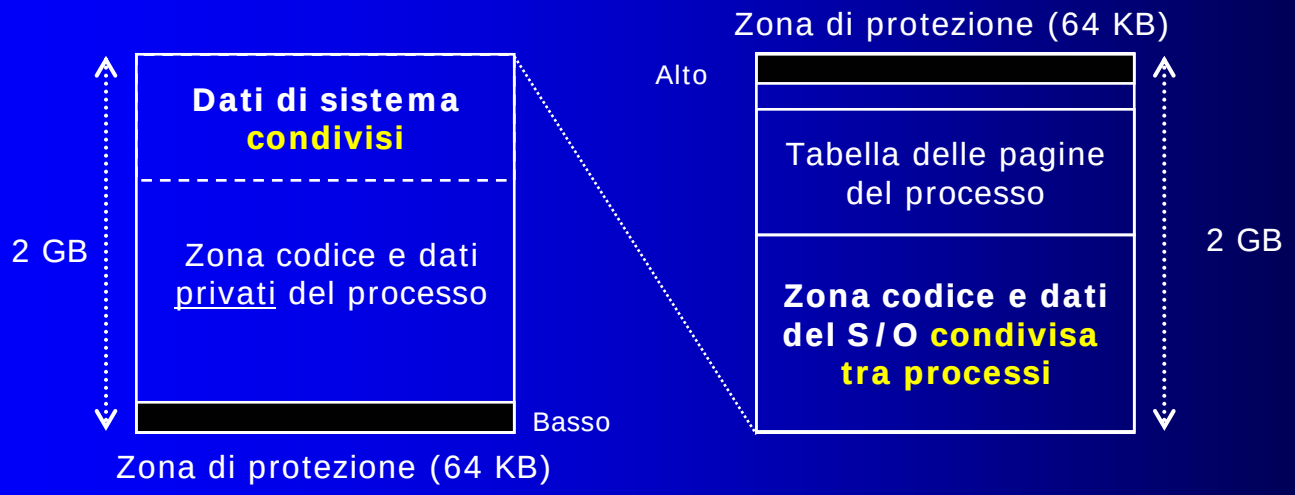
- Sequenza di *boot* come in GNU/Linux
  - Lettura della struttura di FS
  - Localizzazione ed esecuzione del *file* `ntldr` che carica **Win NT**
    - Il FS può avere struttura FAT-16, FAT-32, `NTFS`
  - Lettura del *file* di configurazione `Boot.ini`
  - Caricamento di `hal.dll`, `ntoskrnl.exe` e `bootvid.dll`
  - Lettura di `registry` e configurazione delle periferiche
  - Attivazione di `ntoskrnl.exe` e creazione del gestore di sessione (processo utente **nativo** `smss.exe`, *session manager subsystem*) che si occupa di
    - Creazione del *daemon* di *login* (`winlogon.exe`)
    - Attivazione del gestore di autenticazione (`lsass.exe`)
    - Attivazione del capostipite di tutti i servizi (`services.exe`)

Inizializzazione – 2

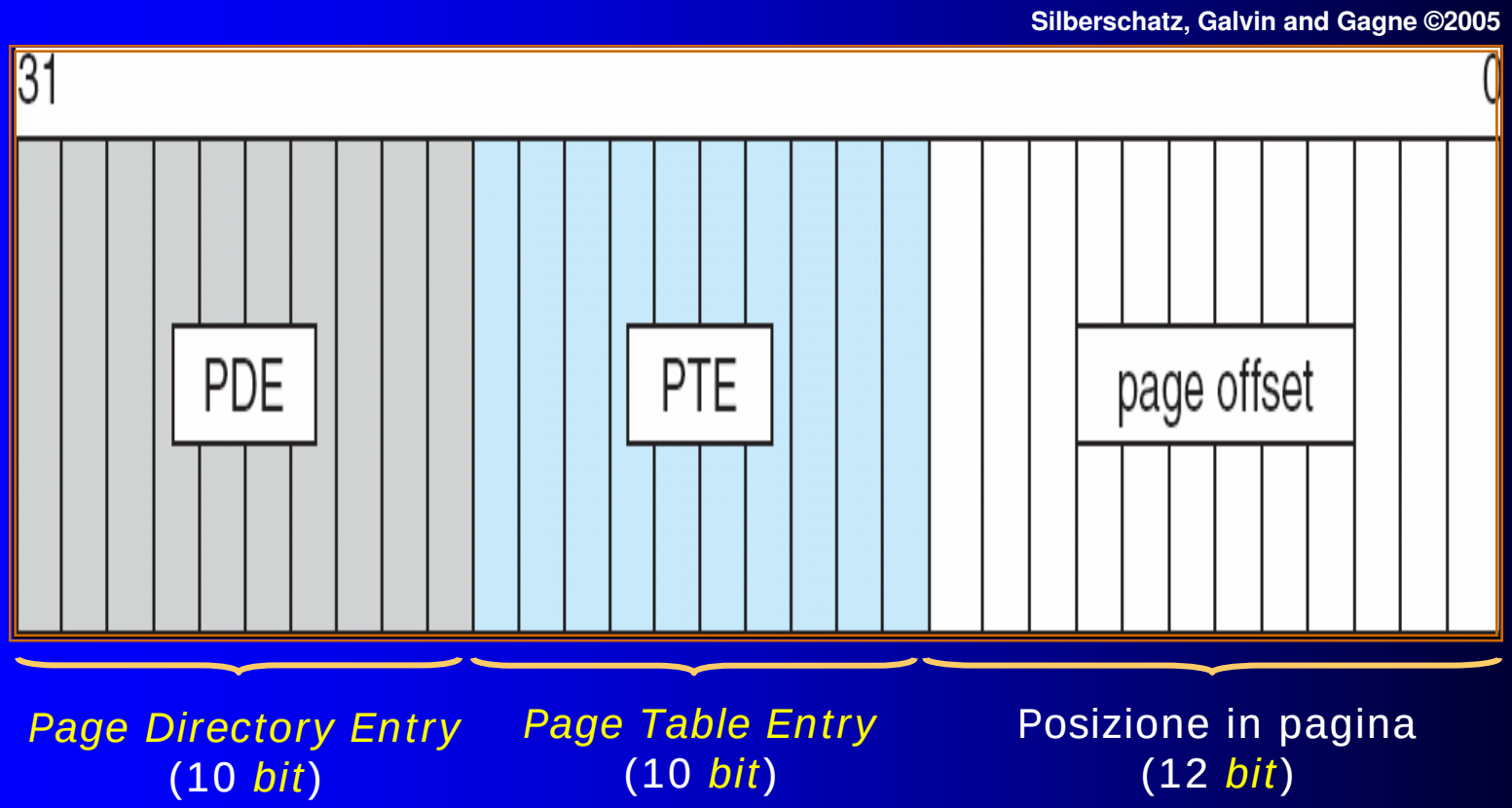
- **`winlogon.exe`** usa un programma della libreria **`msgina.dll`** per eseguire la sequenza di *login* desiderata
  - L'uso di un programma di libreria rende la sequenza più facilmente configurabile dagli amministratori di sistema ma anche più misterioso
  - ***M**icro**s**oft **G**raphical **I**dentification a**n**d **a**uthentication*
- Poi preleva da **`registry`** il profilo d'utente da cui determina il programma di *shell* da eseguire
  - Generalmente si tratta di **`explorer.exe`** ma la scelta è configurabile tramite **`registry`**

Gestione della memoria – 1

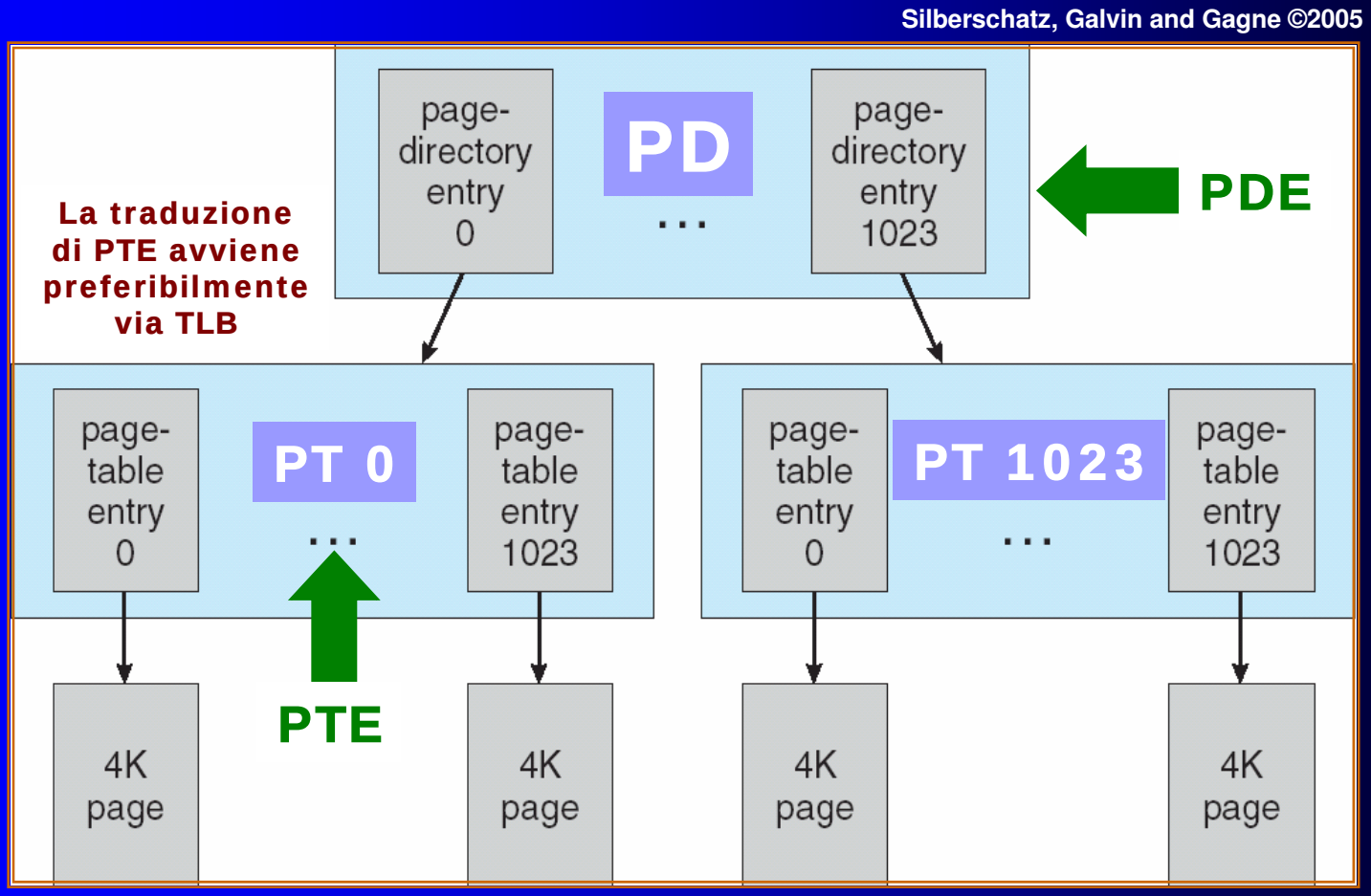
- Ogni processo dispone di uno spazio di indirizzamento virtuale **paginato** ampio 4 GB
  - Suddiviso in 2 zone adiacenti ampie 2 GB ciascuna
- Indirizzi logici espressi su 32 *bit*



Indirizzamento virtuale – 1



Indirizzamento virtuale – 2



Indirizzamento virtuale – 3

- La **PD** è in una posizione nota a priori allineata al margine di una pagina
  - L'indirizzo della sua base è multiplo di 4 KB
  - Le sue *entry* sono ampie 32 *bit*
  - La *entry* indirizzata viene localizzata da
    - $\text{Base}_{PD} + (\text{PDE} \times 00_2)$
      - Perché il campo PDE indirizza unità ampie 4 B
- Una parte del valore trovato serve per localizzare la base della **PT** indirizzata
  - Come sopra per trovare la *entry* indirizzata in PT che fornisce la base della pagina riferita cui si somma il campo *page offset* per trovare il B di base della **parola** indirizzata
- Tutta questa traduzione può essere fornita dalla TLB

Gestione della memoria – 2

- Zone di protezione (**indirizzi illegali**) usate per consentire la rilevazione di possibili errori di indirizzamento da parte del processo
- Codice e dati di S/O **replicati** nella memoria del processo consentono ai *thread* di passare in modo nucleo senza cambiare spazio di indirizzamento
  - Cambia solo lo *stack* (da utente a nucleo)

Gestione della memoria – 3

- Una pagina virtuale può essere
  - **R** (lettura) / **W** (scrittura) / **E** (esecuzione)
  - **Libera**: non riferita da alcun **PTE**
    - Tutte le pagine di un processo sono inizialmente libere (*paging-on-demand*)
  - **Assegnata**: in uso per codice o dati
    - Viene riferita tramite indirizzo virtuale e caricata da disco ove non fosse già presente in RAM
  - **Prenotata**: non ancora in uso, ma **non libera**
    - Per agevolare l'assegnazione di pagine contigue a processi

Gestione della memoria – 4

- Ogni pagina assegnata può essere rimossa temporaneamente dalla RAM
  - Le pagine codice e *file* mappati in memoria hanno una posizione nota (un *file*) su disco
  - Le aree di lavoro **non** la hanno
- MS Windows associa loro un *page file* ma solo nel momento del bisogno
  - Per evitare di tener impegnate vaste aree del disco
  - Fino a 16 *page file* a massima dimensione creati (ma non assegnati!) a tempo di inizializzazione del sistema

Gestione della memoria – 5

- Più processi possono condividere l'accesso a pagine di uno stesso *file* **mappato in memoria**
  - Un **DLL** è un tipico esempio di *file* mappato in memoria
    - Codice **condiviso** in sola lettura
    - Dati statici R/W **copiati** per ciascun processo (*copy-on-write*)
  - Altri *file* dati velocizzano la comunicazione tra processi
  - Ogni processo che accede a un *file* possiede specifici diritti di accesso che il S/O si preoccupa di far rispettare
- La stessa posizione nel *file* può corrispondere a indirizzi virtuali **diversi** per processi distinti
  - Gli indirizzi riferiti nel codice condiviso di **DLL** devono pertanto essere espressi in modo **relativo**
  - A cura del compilatore

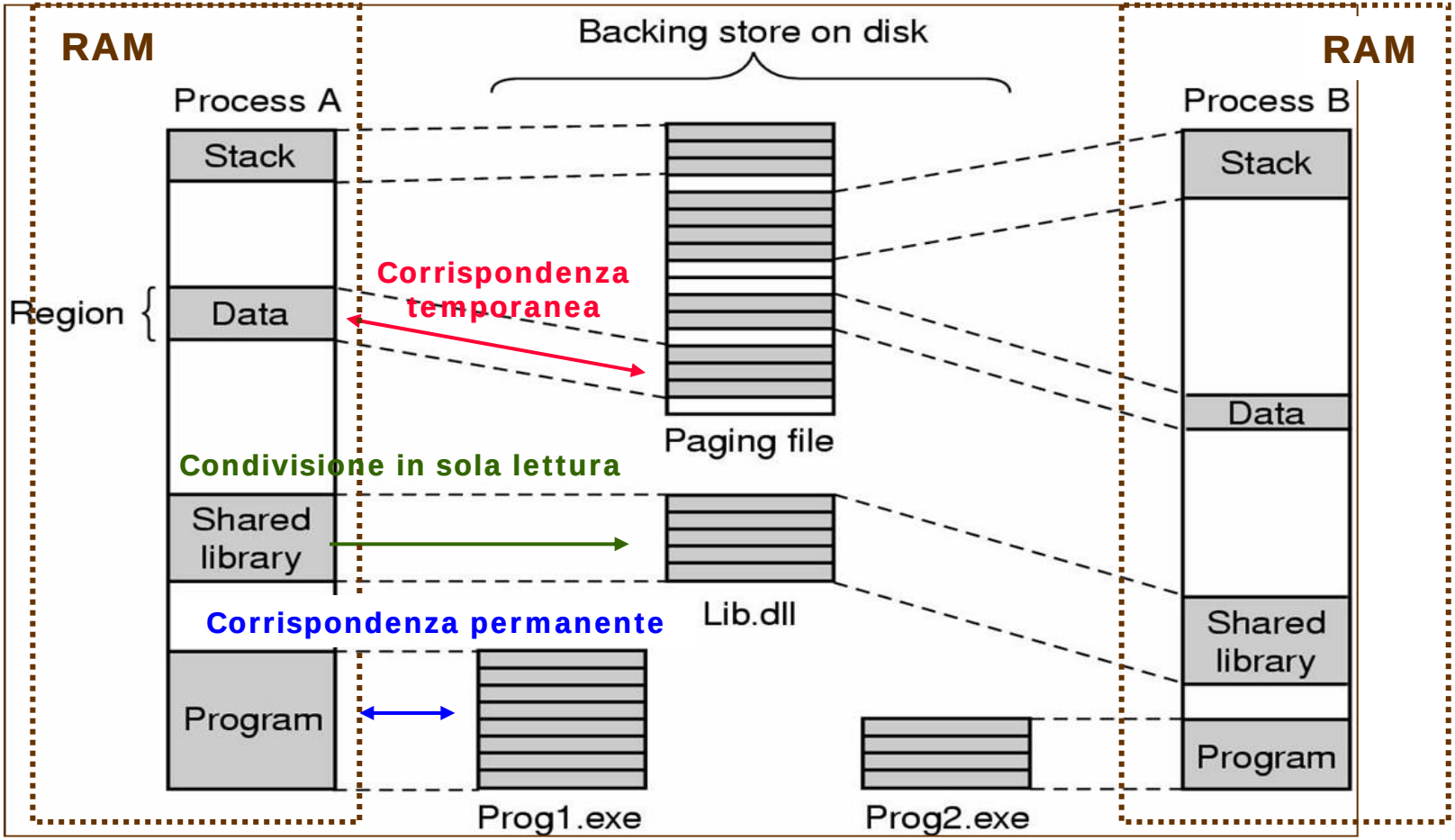
Gestione della memoria – 6

- Ogni processo può gestire direttamente la propria memoria virtuale tramite **`Win32 API`**
- Le relative chiamate di sistema operano su **regioni** di pagine **contigue**
  - Agendo sulla memoria virtuale del chiamante
- Le pagine possono essere acquisite (per assegnazione o prenotazione), rilasciate, protette, rese inamovibili, etc.
- Chiamate di sistema sono anche disponibili per la gestione dei *file* mappati in memoria

Gestione della memoria – 7

- ***Paging-on-demand*** con pagine ampie 4 KB (Pentium) fino a un massimo di 64 KB
  - Il S/O può usare pagine ampie 4 MB per ridurre la dimensione della propria tabella delle pagine
  - **Gestione per processi**
    - Un descrittore (**`Virtual Address Descriptor`**) raccoglie le informazioni di controllo dello spazio di indirizzamento virtuale del processo
      - Suddiviso per regioni di pagine virtuali
    - A ogni regione effettivamente in uso corrisponde una lista delle pagine che la compongono, il cui puntatore è posto nel **`VAD`**

Gestione della memoria – 8



Gestione della memoria – 9

- L'assegnazione di pagine *on demand* avviene per **gruppi di pagine contigue** (≤ 8)
  - Per garantire maggior località
- **5** diverse situazioni possono verificarsi a seguito di un riferimento fallito
  - **La pagina riferita non è assegnata al processo**
    - Errore fatale → **BSoD**
  - **La pagina non può essere riferita**
    - Errore fatale (*protection fault*) → **BSoD**
  - **La pagina condivisa esiste ma non può essere scritta**
    - Copia locale assegnata al richiedente (*copy-on-write*)
  - **Le aree *stack* o dati devono crescere**
    - Assegnazione di una nuova pagina in RAM
  - **La pagina riferita è prenotata ma non ancora assegnata**
    - La pagina viene assegnata e posta in RAM

Gestione della memoria – 10

- Il caricamento di una nuova pagina in RAM può richiedere il **rimpiazzo** locale di un pagina “vecchia”
  - Solo se non vi sono abbastanza pagine libere
  - Il sistema mantiene una lista delle pagine libere
  - A ogni processo i si associa l’insieme I_i delle sue pagine attualmente in RAM (***Working Set***)
    - L’ampiezza del WS I_i può variare solo entro limiti prefissati

      $$Min_i \leq \#\{I_i\} \leq Max_i$$
  - Il rimpiazzo avviene entro il **WS** del richiedente e **solo se** $\#\{I_i\} = Max_i$ altrimenti la pagina viene aggiunta
    - Politica di rimpiazzo locale
- Si ha **rimpiazzo globale** se e solo se un particolare processo deve scambiare proprie pagine tra RAM e disco troppo spesso

Gestione della memoria – 11

- Anche il S/O stesso è visto come un processo con un proprio **WS** con pagine rimpiazzabili
  - Solo alcune pagine del S/O sono **inamovibili**
- Un *daemon* di **`kernel`** con periodo 1 s. accerta che vi siano sufficienti pagine libere
- Se insufficienti il *daemon* attiva un *thread* del **`Memory manager`** che esamina con una euristica gli WS dei processi per rilasciarne pagine
  - Processi non recentemente attivi con WS ampi vengono scrutinati prima degli altri
  - Le pagine necessarie si prelevano dagli WS di ampiezza vicina al massimo e con scarso uso recente

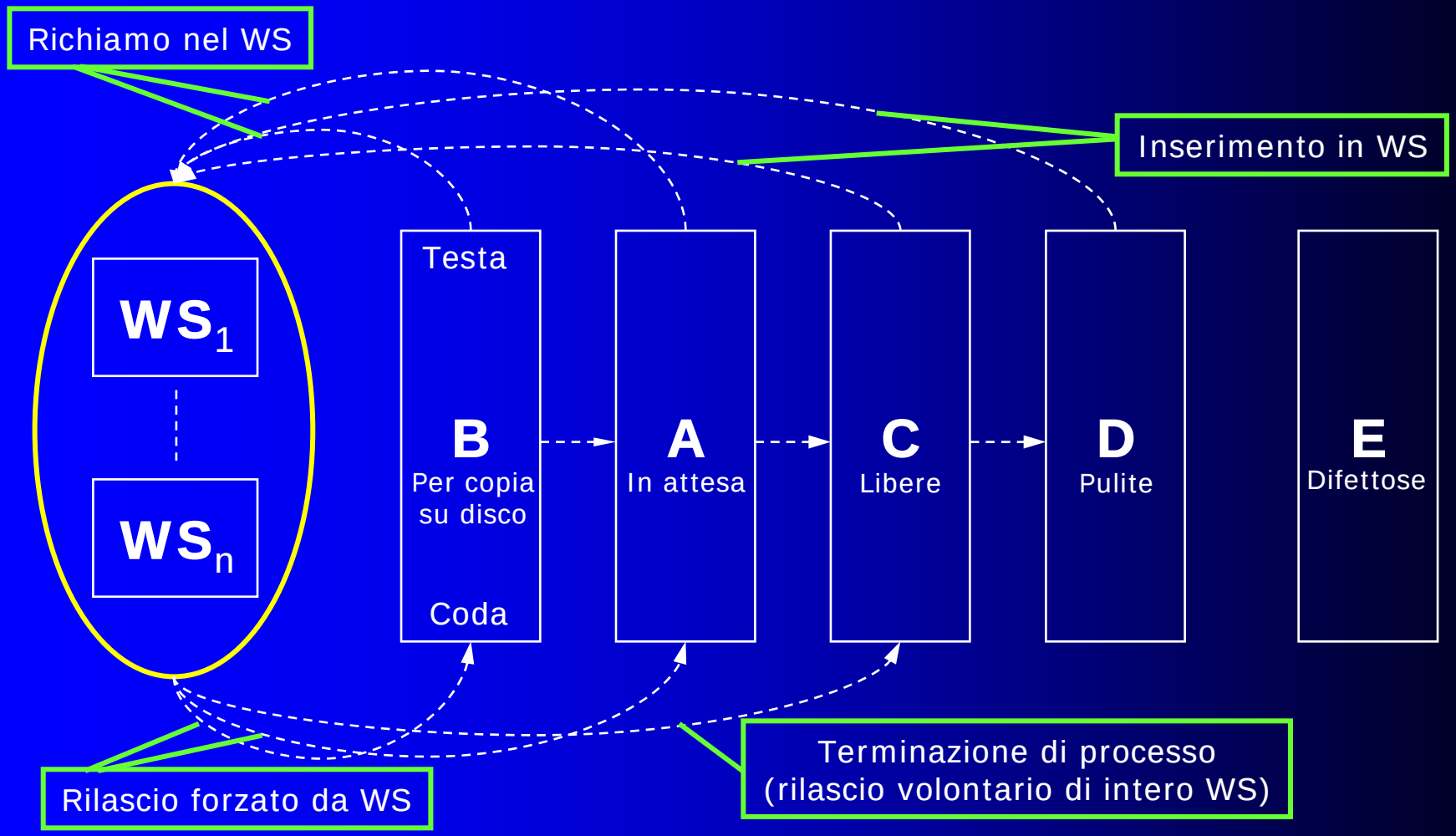
Gestione della memoria – 12

- Ciascuna *page frame* in RAM può essere
  - **In uso** e appartenere a 1 WS (≥1 se condivisa)
  - **Rilasciata** e appartenere a 1 sola lista
    - [**A**] **In attesa**: pagina recentemente rimossa dal WS di un processo ma ancora associata a esso e **non** modificata
      - Può essere riassegnata e sovrascritta senza problemi
    - [**B**] **Da copiare su disco**: ~ **A** ma se rimpiazzata deve essere riportata su disco
    - [**C**] **Libera**: ~ **A** ma non più associata ad alcun processo
    - [**D**] **Azzerata**: ~ **C** ma con contenuto **obliterato** per consentire riassegnazione **senza travaso di informazione privata**
    - [**E**] **Difettosa**: pagina che non può più essere utilizzata a causa di difetti nella zona di memoria fisica

Gestione della memoria – 13

- Lo **swapper thread** (*daemon*) del **Memory manager** porta in [**A**] o [**B**] le pagine dello *stack* dei processi i cui *thread* siano stati tutti recentemente inattivi
- Altri 2 *daemon* assicurano che vi siano abbastanza pagine in [**C**] salvando su disco quelle in [**B**] e poi accodandole in [**A**]
  - La riscrittura di *file* mappati su disco può richiedere nuovi blocchi
  - La loro assegnazione richiede di aggiornare la mappa dei blocchi e per farlo possono servire nuove pagine di lavoro in RAM
    - Questa situazione è sorgente di **stallo potenziale**
  - Il 2o *daemon* può però **sempre** creare spazio in RAM per pagine di lavoro perché il *file* delle pagine è preallocato e dunque sempre disponibile
- Un WS che cresce preleva pagine libere da [**C**] se le sovrascrive interamente (senza conservare dati precedenti) da [**D**] altrimenti
  - Un *daemon* dedicato che opera per conto del **kernel** azzera periodicamente il contenuto di pagine in [**C**] e le pone in [**D**]

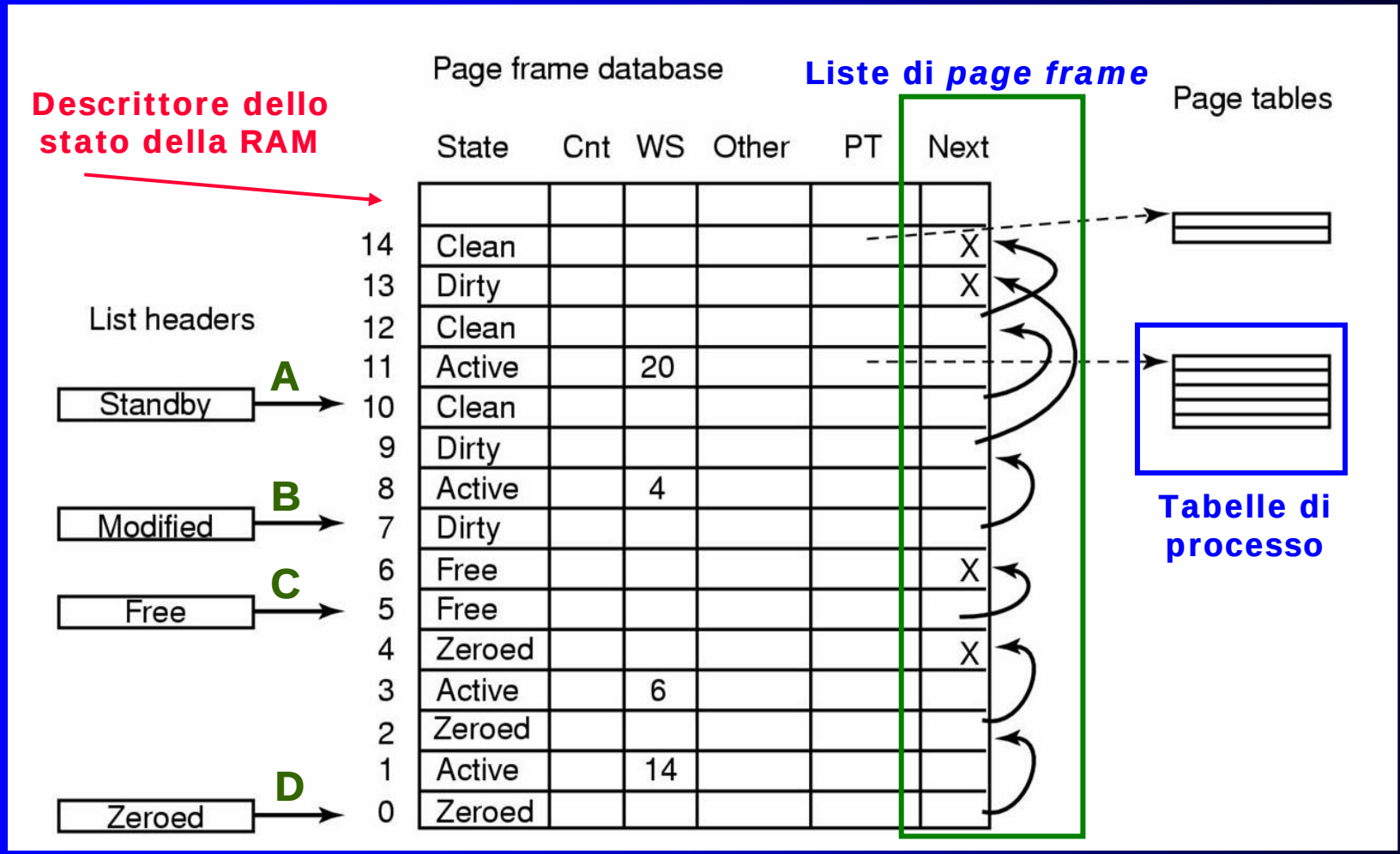
Gestione della memoria – 14



Gestione della memoria – 15

- Euristiche complesse e **non garantite** governano le scelte effettuate dalle varie attività di gestione delle liste [A] – [D]
  - L'amministratore di sistema può influenzare alcune euristiche mediante parametri di configurazione
- Lo stato della RAM viene mantenuto in una tabella dedicata acceduta per indice di pagina fisica (*page frame database*)
  - Pagina valida/invalida, contatore dei riferimenti, WS di appartenenza, lista di appartenenza, etc.

Gestione della memoria – 15



Gestione dell'I/O – 1

- Architettura progettata per massima flessibilità
  - Per supportare agilmente l'arrivo di nuove periferiche
- Sotto la responsabilità dell'**`I/O manager`** coadiuvato dal **`Plug-&-play manager`**
- Il **`p&p m`** interroga ogni *bus* per determinare la presenza di periferiche
  - A tempo di inizializzazione per le interfacce **statiche**
  - **Periodicamente** per le interfacce **dinamiche** (p. es.: USB)
- $\forall$ periferica registrata viene caricato e installato il corrispondente gestore (come per GNU/Linux) cui si associa uno specifico **oggetto di controllo**

Gestione dell'I/O – 2

- **Dispositivi di ingresso**
  - Tastiera, *mouse*, *touchpad*, *cloche* (*joystick*), telecamera, microfono per comandi vocali, lettore codice a barre, ...
- **Dispositivi di uscita**
  - Schermo, stampante, *plotter*, masterizzatore, schede suono, ...
- **Dispositivi di memorizzazione**
  - Dischi magnetici flessibili (*floppy*), ad alta densità (ZIP), fissi (*hard*) di tipo IDE o SCSI, e *flash hard*, disco ottico di tipo CD-ROM o DVD, nastro
- Diverse caratteristiche di comportamento e di interfacciamento per ciascuna tipo di periferica

Gestione dell'I/O – 3

- Il **Power manager** si preoccupa di contenere il consumo energetico del sistema
  - Chiedendo all'**I/O manager** di cambiare lo stato dei gestori dei dispositivi in relazione all'uso delle periferiche
- La richiesta di dati di FS è inviata al **Cache manager**
  - Se necessario girata all'**I/O manager** che la indirizza al gestore appropriato
  - Il FS viene visto come gestore di periferica di I/O

Gestione dell'I/O – 4

- Ogni gestore di periferica **conforme** deve esibire caratteristiche **fissate** di comportamento
  - Trattare richieste di servizio codificate in forma di pacchetto standard (IRP, **`I/O request packet`**)
  - Avere e usare rappresentazione a **oggetti** di tipo Win32 API
  - Prevedere il trattamento di periferiche rimovibili
  - Obbedire a richieste di cambio di stato da parte del **`Power manager`**
  - Aderire alle direttive di configurazione emesse dall'**`I/O manager`** (nessun prerequisito del gestore è immodificabile)
  - Permettere più esecuzioni concorrenti → **procedure rientranti**
  - Essere utilizzabile **anche** in ambiente Windows 98

Gestione dell'I/O – 5

- Il **`Plug-&-play manager`** interroga ogni periferica rilevata per identificarla e localizzarne sul FS il *software* di gestione
  - Caricamento dinamico automatico
  - Altrimenti finestra di dialogo per richiedere l'inserzione di un disco con il *software* necessario
- Ogni gestore deve fornire alcune procedure con profilo e comportamento predefiniti
  - Servizi localizzati a partire dall'**oggetto** associato alla periferica
    - Inizializzazione del gestore, registrazione della periferica, configurazione del vettore delle interruzioni, ...

Gestione dell'I/O – 6

- Oltre a demandare a **HAL** il trattamento uniforme delle caratteristiche specifiche dell'*hardware* del sistema, alcuni gestori prevedono una struttura a livelli
  - **Livello alto**
    - **Gestione funzionale** della periferica
      - Analoga alla logica d'uso dei *file* in UNIX
  - **Livello basso**
    - **Gestione dei protocolli** di comunicazione fisica verso la periferica
      - Per esempio: *bus* di tipo PCI, USB, SCSI

Architettura di NTFS – 1

- **NT 5.x** supporta l'intera gamma dei FS Windows e **anche** `ext2fs` di GNU/Linux
  - **FAT-16**
    - Limite **logico** all'ampiezza di partizione
    - $\leq 2^{16}$ **blocchi** di ampiezza massima 32 KB $\rightarrow$ 2 GB
  - **FAT-32**
    - Limite **fisico** all'ampiezza di partizione
    - $\leq 2^{32}$ **settori** da 512 B $\rightarrow$ 2 TB
    - Limite logico : 2^{28} blocchi da 8 KB $\rightarrow$ 2 TB
  - **NTFS**
    - **Nuova concezione** con indici espressi su 64 *bit*

Architettura di NTFS – 2

- Nome di *file* fino a 255 caratteri in codifica **UNICODE** (2 – 4 B/carattere)
  - Un nome espresso come **cammino** (relativo o assoluto) non può eccedere (32 K – 1) caratteri
  - Distinzione tra maiuscolo e minuscolo ma **senza effetto** per buona parte di **`Win32 API`** (*backward compatibility*)
- *File* come **aggregato di attributi** rappresentati come sequenza di caratteri (*byte stream*)
  - Esempio: sequenza breve contenente il nome del *file* e l'indirizzo dell'oggetto associato (+ *thumbnail preview*) + sequenza lunga (fino a 2^{64} B !) contenente i dati del *file*
  - Idea copiata da Apple ☺ introdotta (anche) per compatibilità

Architettura di NTFS – 3

- FS ad **architettura gerarchica** (come **ext2**)
  - \ invece di / come separatore nelle espressioni di cammino
  - Supporto per *directory* corrente (**wd**)
  - Supporto per entrambe le forme di **link**
- Servizi di FS resi tramite procedure di libreria **Win32 API**
  - Funzionalmente simili a GNU/Linux ma di concezione assai più bizantina

Architettura di NTFS – 4

- NTFS è una collezione di **volumi logici**
  - Un volume logico può mappare su **più** partizioni e anche su **più** dischi
  - Volume = sequenza lineare di blocchi (*cluster*) di ampiezza fissa
    - Volumi **diversi** possono avere dimensione di blocco **diverse**
      - Tra 512 B e 4 KB
      - Teoricamente fino a 64 KB
    - Blocco piccolo $\rightarrow$ ridotta frammentazione interna
    - Blocco grande $\rightarrow$ meno accessi a disco ma più frammentazione interna
- Una **MFT** (**`Master File Table`**) per volume
  - **Fisicamente** realizzata come un *file*
    - Perciò può essere salvata **ovunque** nel volume
      - Soluzione più robusta
  - **Logicamente** strutturata come una **sequenza lineare** di $\leq 2^{48}$ *record* di ampiezza da 1 a 4 KB
    - Ciascun *record* descrive 1 *file* identificato da un indice ampio 48 *bit*
    - Gli altri 16 *bit* servono come numero di sequenza (contatore di riuso)

Architettura di NTFS – 5

- Ciascun *record* contiene un numero **variabile** di coppie `<descrittore di attributo, valore>`
  - Il 1° campo della coppia specifica la **struttura** dell'attributo
    - Esistono **13 attributi di base** con struttura predefinita
    - Possono esistere altri **attributi aggiuntivi** a struttura libera
  - Il 2° campo denota il **valore** dell'attributo
    - Se possibile il valore è rappresentato interamente nel *record*
      - Attributo **residente**
    - Altrimenti rappresentato da un puntatore al suo *record* remoto
      - Attributo **non residente**
  - Il valore dell'attributo **dati** rappresenta il contenuto del *file*

Architettura di NTFS – 6

- Il campo `<descrittore di attributo>` per **attributi residenti** ha ampiezza 24 B
  - Quello per **attributi non residenti** è più ampio
- I 13 attributi di sistema non applicano a tutti i *file*
  - Gli attributi dei *file* corrispondono a quelli che GNU/Linux pone negli *i-node*
  - Con l'aggiunta dell'**identificatore dell' oggetto** corrispondente
    - 64 *bit* con valore **unico per volume**
  - Il contenuto dati di *file* di ampiezza < 1 KB viene memorizzato **interamente** entro un *record* di MFT (attributo residente)
    - ***Immediate file*** (rari)
  - Per *file* più grandi il valore dell'attributo dati diventa la lista ordinata dei corrispondenti blocchi su disco
    - Attributi non residenti

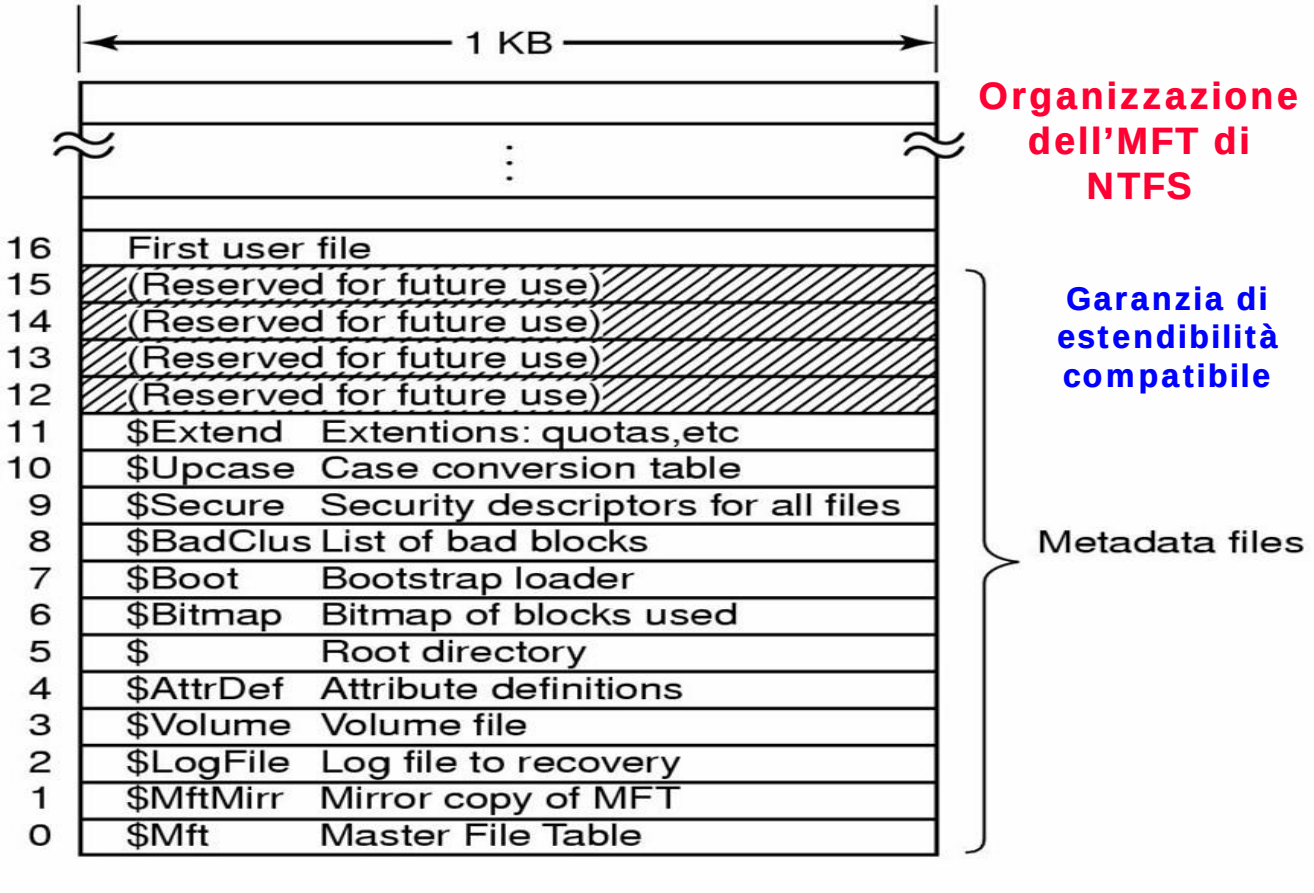
Architettura di NTFS – 7

- Il valore di **attributi non residenti** è posto su sequenze di blocchi contigui
  - NTFS cerca di assegnare allo stesso *file* sequenze di blocchi contigui piuttosto che singoli blocchi isolati
    - Strategia analoga a quella di `ext2fs`
    - Nel caso peggiore i dati di un *file* possono trovarsi su sequenze di blocchi singoli non adiacenti
  - Le sequenze non sono necessariamente contigue tra loro
- 1 *record* base in MFT $\forall$ *file* sequenziale nel volume
  - La struttura interna del *record* dipende dalla dimensione del *file* e dalla contiguità dei suoi blocchi
  - *File* con zone interne non utilizzate (e.g. poste a 0 e riservate per uso futuro) sono chiamati ***file* sparsi** e sono gestiti in modo distinto e particolare

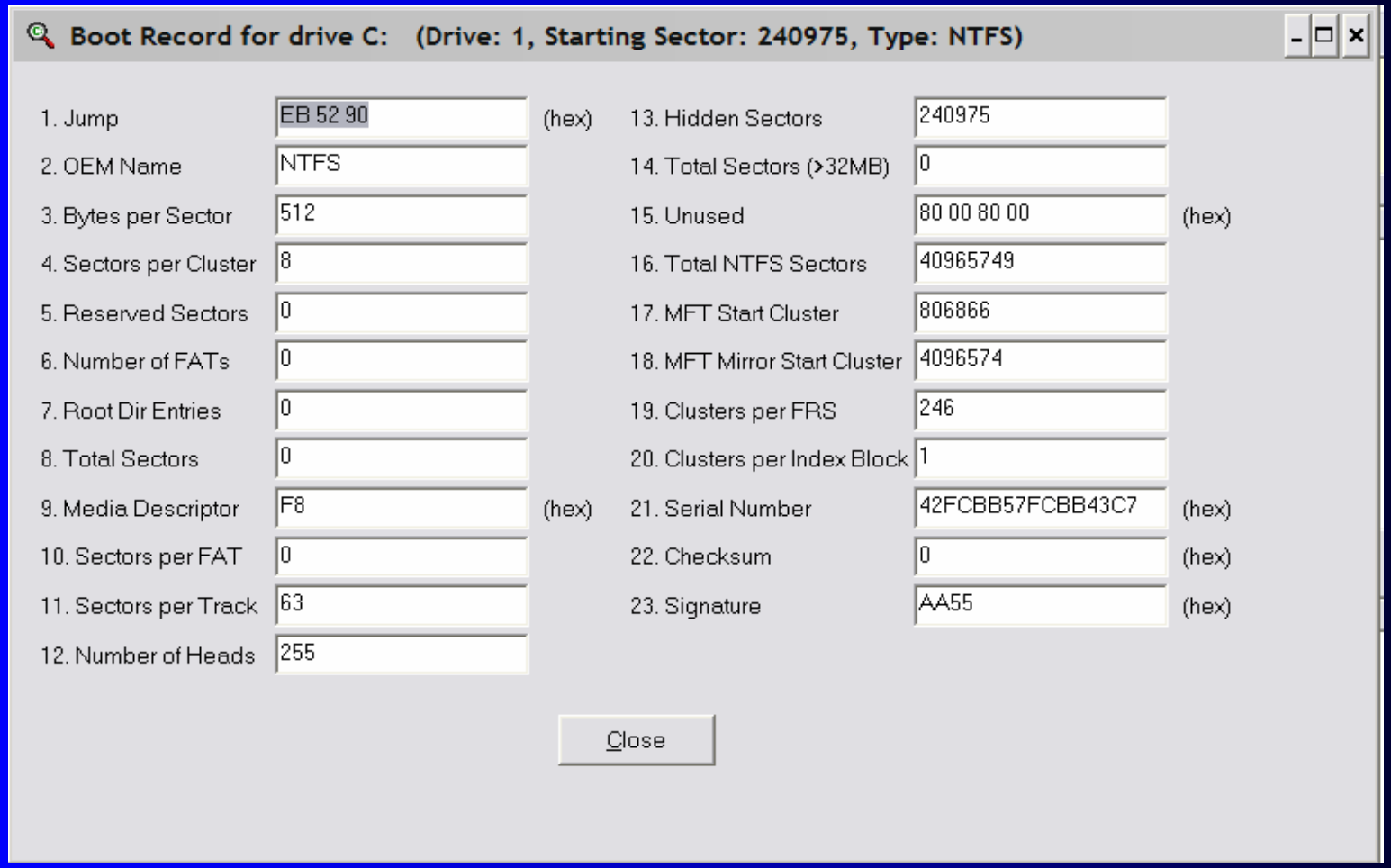
Architettura di NTFS – 8

- I primi 16 *record* dell'**MFT** sono riservati per "*file* **trascendenti**"
  - *Record* che descrivono l'organizzazione interna dell'intero volume (***metadata***)
- Il 1° *record* (#0) descrive l'MFT stesso
  - Che è trattato come un *file* i cui record compongono l'MFT (!)
- Il 2° (#1) replica i primi 16 *record* in modo **non residente** ponendone il contenuto in fondo al volume (***mirror file***)
  - Facilita il ripristino del volume in caso di corruzione dell'MFT
- Il 3° (#2) è una specie di *journal*
- Il 4° (#3) caratterizza il volume
  - Nome logico, versione di FS, data di creazione, etc.
- Il 5° (#4) descrive gli attributi usati nel volume
  - Attributi non residenti denotati da un puntatore di 48 *bit* a un *record* remoto
  - Un codice di controllo di 16 *bit* deve coincidere con il numero di sequenza del *record* di base del valore dell'attributo in MFT
  - **64 *bit* in tutto**
- Inoltre: puntatore alla radice del FS (#5); *bitmap* dei blocchi liberi; copia del codice di *boot* di volume o suo puntatore; etc.

Architettura di NTFS – 9



Architettura di NTFS – 10



Architettura di NTFS – 11

Volume MS Windows Partition (C:)

Volume size	= 19.53 GB
Cluster size	= 4 KB
Used space	= 15.35 GB
Free space	= 4.18 GB
Percent free space	= 21 %

Volume fragmentation

Total fragmentation	= 0 %
File fragmentation	= 0 %
Free space fragmentation	= 1 %

File fragmentation

Total files	= 98,161
Average file size	= 215 KB
Total fragmented files	= 0
Total excess fragments	= 0
Average fragments per file	= 1.00

Pagefile fragmentation

Pagefile size	= 2.00 GB
Total fragments	= 1

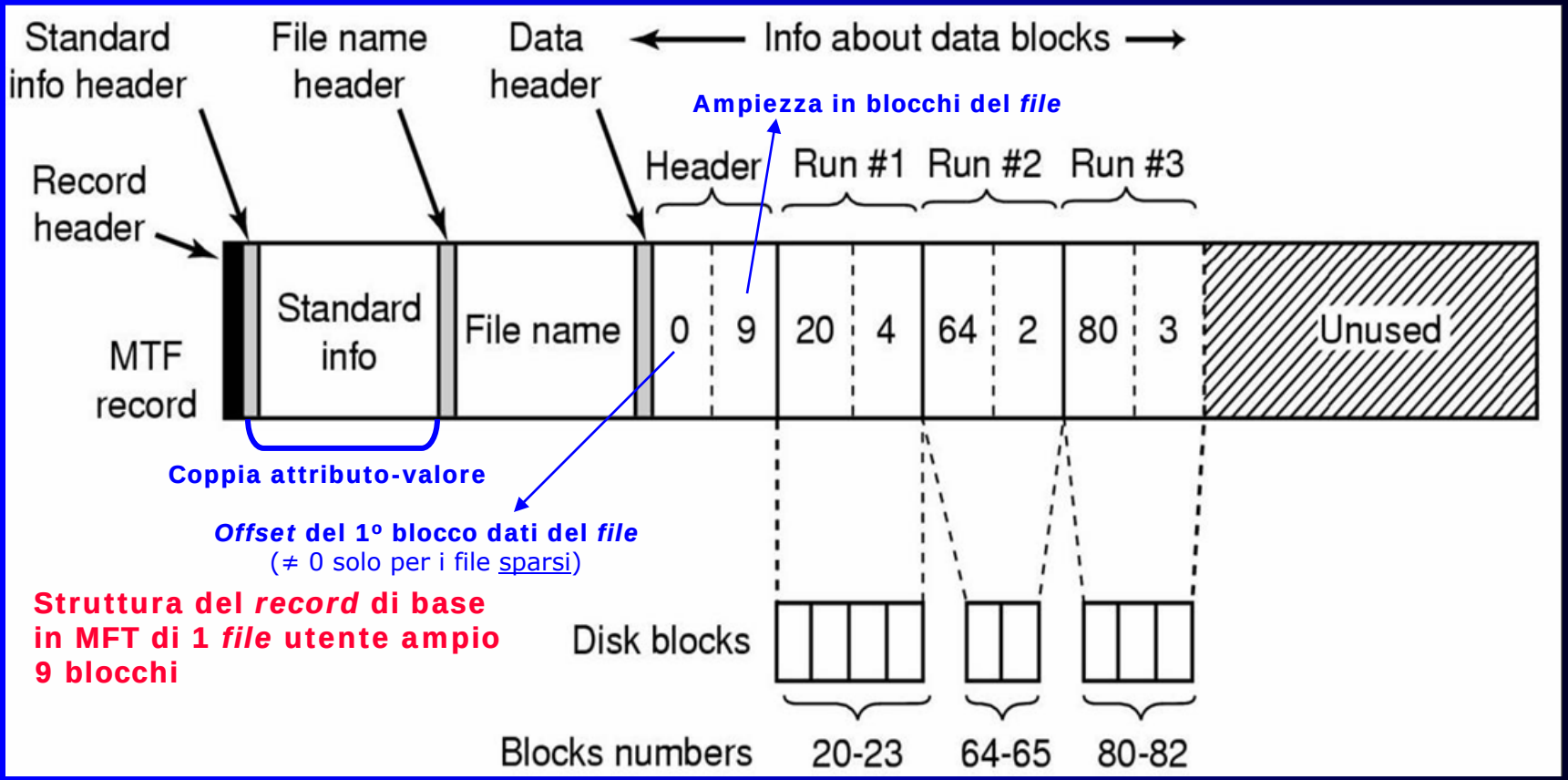
Folder fragmentation

Total folders	= 9,492
Fragmented folders	= 1
Excess folder fragments	= 0

Master File Table (MFT) fragmentation

Total MFT size	= 119 MB
MFT record count	= 107,892
Percent MFT in use	= 88 %
Total MFT fragments	= 3

Record base senza estensioni – 1



Record base senza estensioni – 2

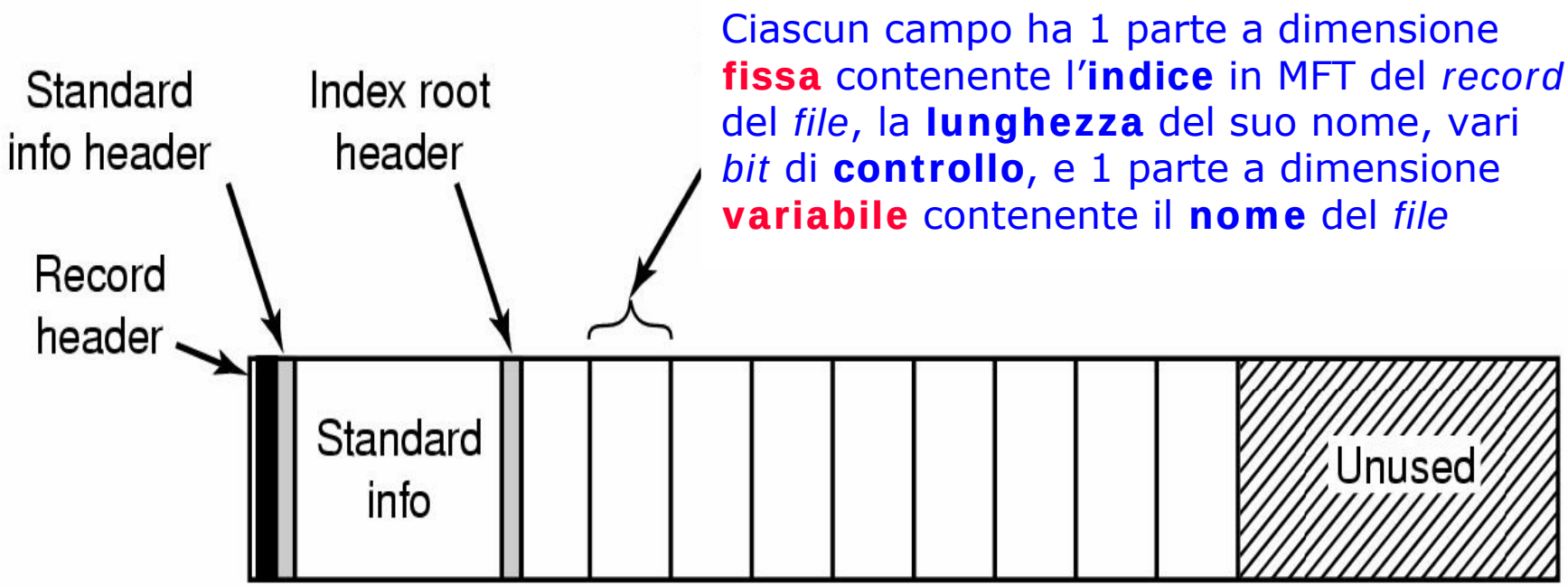
- Nella figura precedente un **singolo** descrittore basta a contenere l'intera lista di sequenze (***run***) di blocchi contigui di dati del *file*
  - 9 blocchi dati in totale suddivisi in 3 sequenze ciascuna descritta come
    - Indirizzo su disco del 1° blocco della sequenza
    - Ampiezza in blocchi della sequenza
  - L'intestazione dell'attributo dati specifica il # di sequenze presenti nel descrittore (3 in questo caso)
  - La prima coppia di attributi dati specifica l'*offset* entro il *file* del 1° blocco coperto dal descrittore e l'*offset* del 1° blocco non coperto (= ampiezza)

Record base senza estensioni – 3

- La strategia NTFS consente di rappresentare *file* di ampiezza virtualmente **illimitata**
- Il numero di *record* necessari per i dati di 1 singolo *file* dipende dalla **contiguità** dei suoi blocchi piuttosto che dalla sua ampiezza
  - 1 *file* da 20 GB costituito da 20 sequenze di 1 M blocchi da 1 KB ciascuno richiede 20+1 coppie di valori espressi su 64 *bit* ovvero $21 \times 2 \times 8$ B $= 336$ B
    - Ampiamente contenuto in 1 singolo *record* MFT
  - 1 *file* da 64 KB costituito da 64 sequenze di 1 blocco ciascuna richiede $(64+1) \times 2 \times 8$ B $= 1040$ B
    - Maggiore della capacità dell'attributo dati su 1 singolo *record*

Record base senza estensioni – 4

Il *record* base in MFT per una *directory* di piccole dimensioni



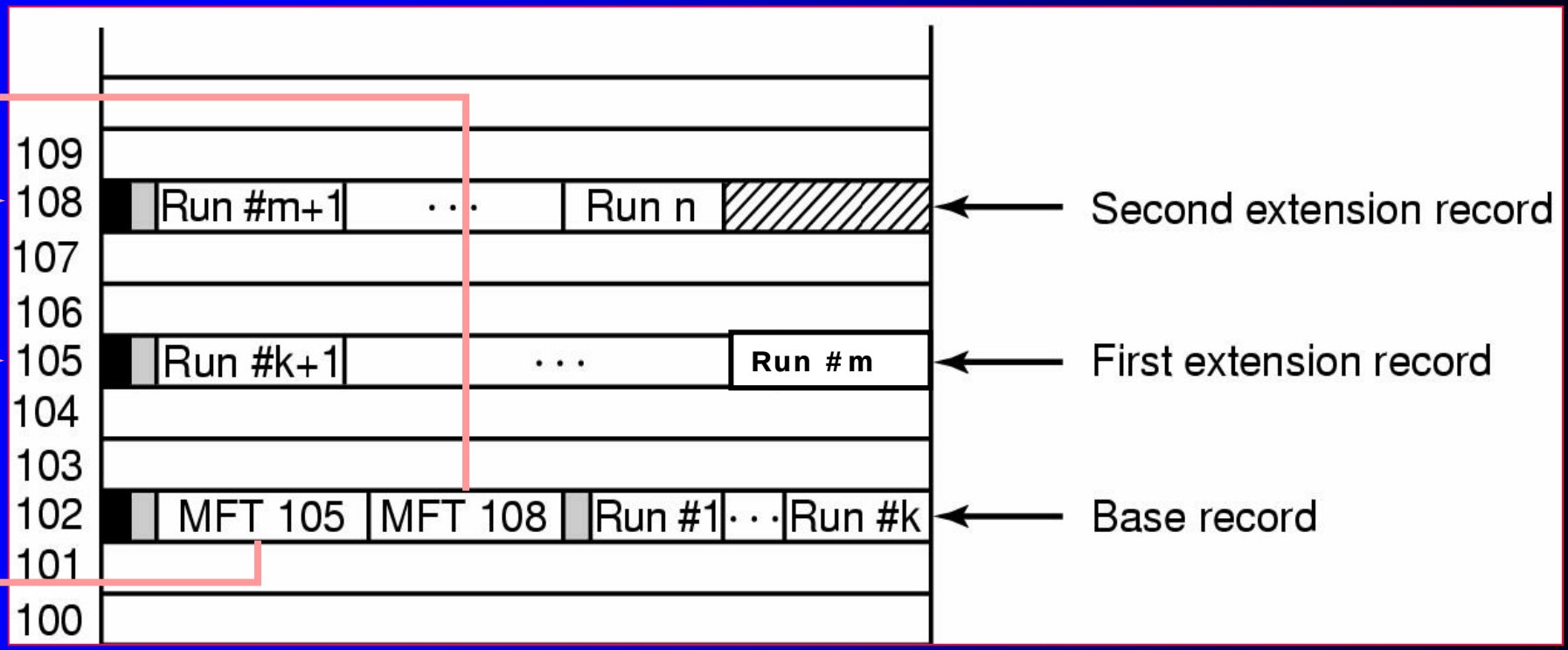
Ciascun campo ha 1 parte a dimensione **fissa** contenente l'**indice** in MFT del *record* del *file*, la **lunghezza** del suo nome, vari *bit* di **controllo**, e 1 parte a dimensione **variabile** contenente il **nome** del *file*

Record con estensioni – 1

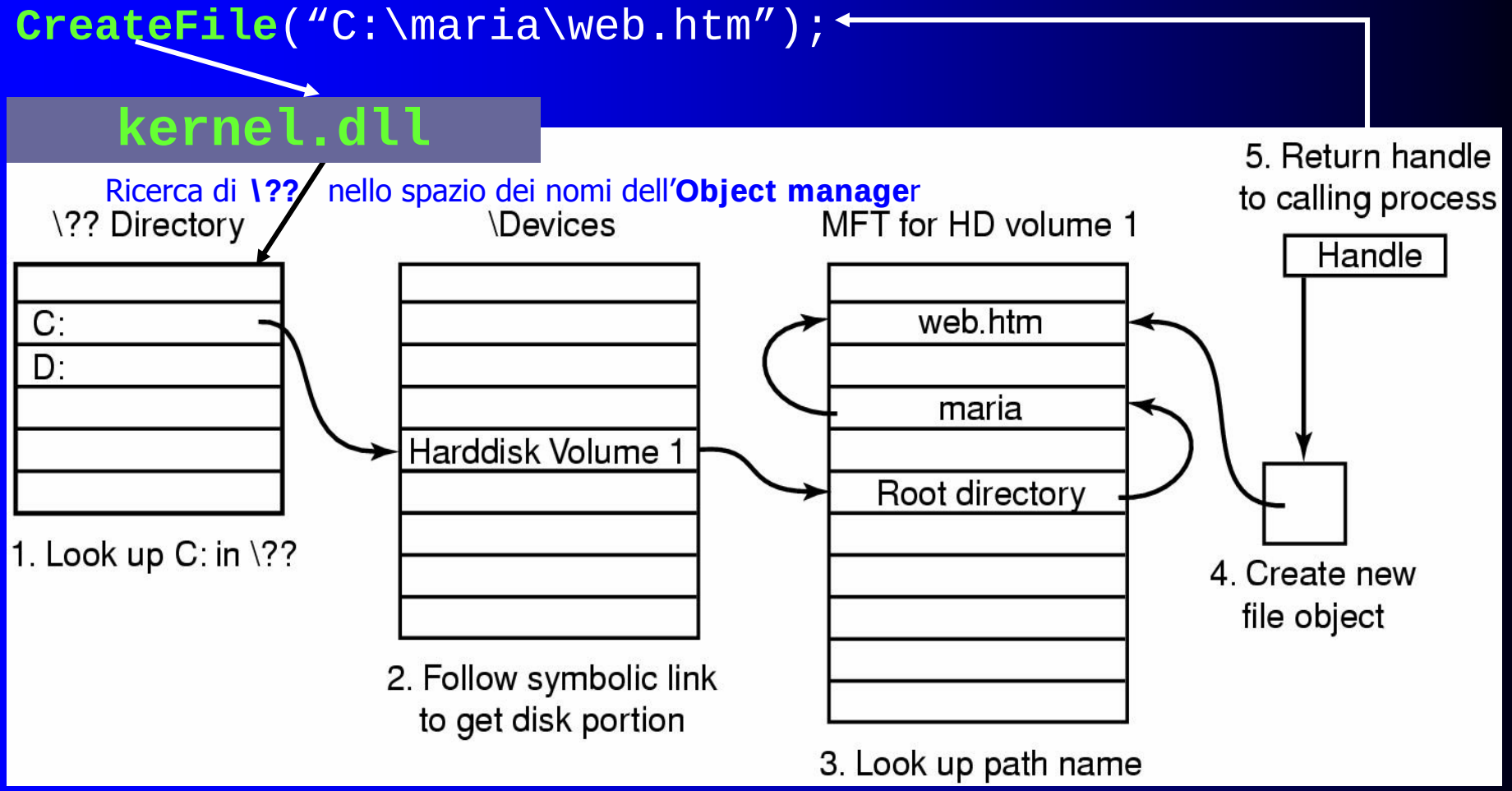
- La rappresentazione di *file* può richiedere **più** *record*
- NTFS usa per questo una tecnica a **continuazioni** analoga a quella usata dagli *i-node* di UNIX e GNU/Linux
  - Il *record* **base** in MFT contiene un puntatore a $N \geq 1$ *record* **secondari** in MFT che descrivono la sequenza di blocchi del *file*
  - Lo spazio rimanente del *record* **base** può descrivere le prime sequenze di blocchi dati del *file*
- Se non vi fosse abbastanza spazio in MFT per i *record* secondari di un dato *file* la loro intera lista verrebbe trattata come un **attributo non residente** e posta in un *file* dedicato denotato da un *record* posto in MFT

Record con estensioni – 2

Un *file* composto da **n** sequenze di blocchi dati con descrizione specificata su 1 *record* base e 2 *record* di estensione in MFT

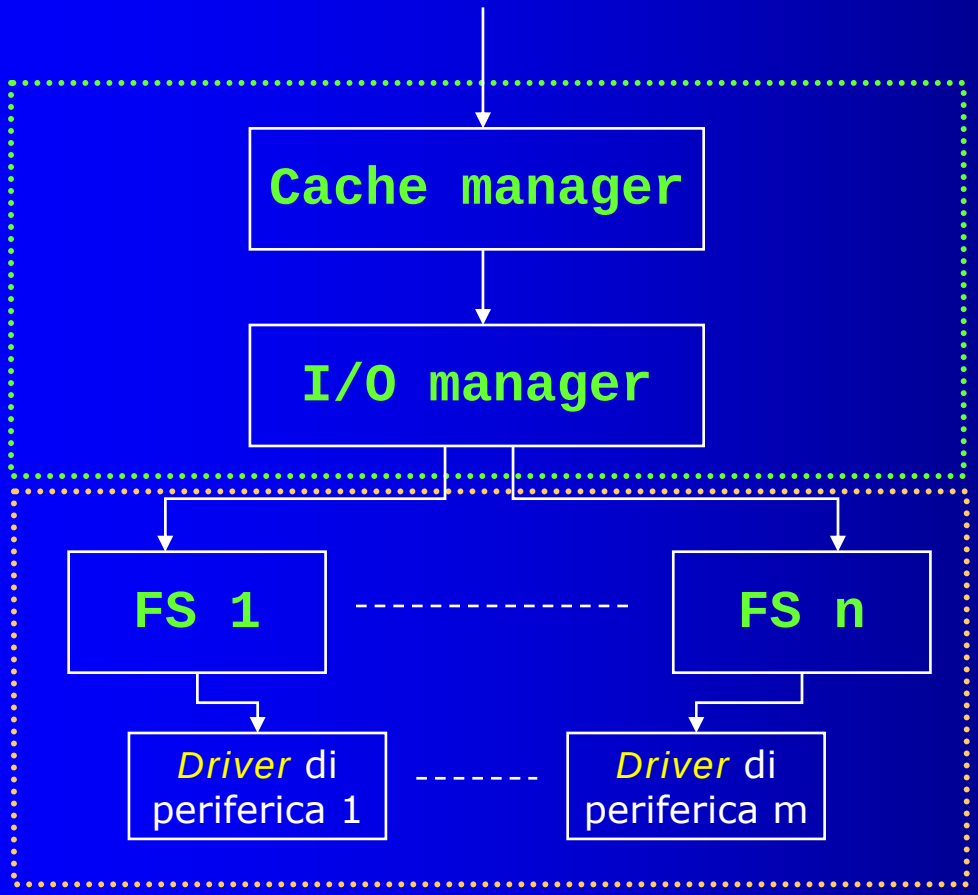


Creazione e localizzazione di *file*



Gestione della *cache* RAM – 1

Richiesta di accesso a *file* attraverso **kernel.dll** e **ntdll.dll**



Livello **executive**

Livello **kernel**

Gestori di periferica

Gestione della *cache* RAM – 2

- L'accesso a *file* avviene tramite **kernel.dll** e **ntdll.dll** e viene indirizzato al **Cache manager**
  - **Indipendentemente** dal tipo di FS
  - Il **Cache manager** tratta **blocchi virtuali** perché non conosce la struttura fisica dello specifico FS
    - Concetto analogo a quello del **VFS** di GNU/Linux
    - **Blocco virtuale** = (***stream***, ***offset***)
    - **Blocco fisico** = (**partizione, indice di blocco**)
- Ogni FS viene visto come **gestore di periferica logica** sotto il controllo dell'**I/O manager**

Gestione della *cache* RAM – 3

- Per ogni *file* in uso il **Cache manager** alloca 256 KB di spazio di indirizzamento virtuale di **kernel**
  - **Indipendentemente** dalla dimensione del *file*
  - Lo spazio complessivo a disposizione del **Cache manager** è parametro di configurazione
    - Se necessario rialloca spazio assegnato a *file* vecchi
- Le richieste di accesso a *file* vengono soddisfatte attingendo allo spazio di **kernel**
  - Per ogni dato non disponibile l'**I/O manager** tratta l'errore (*page fault*) **trasparentemente** al **Cache manager** caricando il blocco richiesto

Gestione della *cache* RAM – 4

- Ogni accesso concorrente di più *thread* a uno stesso *file* mappato in memoria riferisce la **stessa** area di `kernel` assegnata dal `Cache manager` al *file*
  - Il *file* è mappato **una sola volta** nello spazio del `kernel` indipendentemente dal numero di utenti
  - Le scritture avvengono nello spazio di `kernel` le letture copiano dati nell'area dell'utente
    - Principio del *copy-on-write*
- Questo meccanismo garantisce la coerenza della condivisione di dati su *file*