Sistemi Operativi
A.A. 2022/2023 - Secondo semestre (teoria)

Note a cura di Nick

Indice

1 Deadlock **5**

1.1 Definizione di deadlock 5

1.1.1 Modello di sistema 5

1.1.2 Condizioni necessarie 6

1.1.3 Esempio di deadlock 6

1.2 RAG: Resource Allocation Graph 7

1.2.1 Esempio di RAG 8

1.3 Gestione dei deadlock 9

1.3.1 Prevenzione statica 10

1.3.2 Prevenzione dinamica 12

1.3.3 Stato safe, unsafe e deadlock 12

1.4 Algoritmi per la prevenzione dinamica 14

1.4.1 Algoritmo con RAG 14

1.4.2 Algoritmo del banchiere 15

1.4.3 Algoritmo di allocazione (P_i) 16

1.4.4 Algoritmo di verifica dello stato 17

1.4.5 Esempio di algoritmo del banchiere 18

1.5 Rilevazione & Ripristino 20

1.5.1 Rilevazione e ripristino con RAG 21

1.5.2 Algoritmo di rilevazione 21

1.5.3 Esempio di rilevazione 23

1.5.4 Ripristino 24

1.6 Conclusione 25

2 Gestione della Memoria **27**

2.1 Introduzione 27

2.1.1 Vincolo 27

2.1.2 Come avviene la trasformazione? 28

2.2 Binding dati/istruzioni e indirizzi di memoria 28

2.2.1 Compile time 28

2.2.2 Load time 29
2.2.3 Run time 29
2.2.4 Binding Statico e Binding Dinamico 29
2.2.5 Linking statico o dinamico 30
2.2.6 Loader statico e dinamico 30
2.3 Spazi di indirizzamento 31
2.4 Binding: Statico vs Dinamico 31
2.4.1 Memory Management Unit - MMU 32
2.5 Considerazioni 32
2.6 Allocazione contigua 32
2.6.1 Tecnica delle partizioni fisse 33
2.6.2 Tecnica delle partizioni variabili 35
2.6.3 Tecnica della compattazione 36
2.6.4 Tecnica del Buddy system 37
2.7 Paginazione 39
2.7.1 Traduzione degli indirizzi 39
2.7.2 Tabella delle pagine 42

Deadlock

1.1 Definizione di deadlock

In un ambiente con multipgrogrammazione, più thread possono competere per ottenere un numero finito di risorse; se una risorsa non è correntemente disponibile, il thread richiedente passa allo stato d'attesa. In alcuni casi, se le risorse richieste sono tratenute da altri thread, a loro volta nello stato d'attesa, il thread potrebbe non cambiare più il suo stato. Situazioni di questo tipo sono chiamate di **deadlock** (*stallo*).

***Def*:**

ciascun processo in un insieme di processi attende un evento che può essere causato solo da un altro processo dell'insieme.

La maggior parte dei sistemi operativi attuali non offre strumenti di prevenzione di queste situazioni, e progettare programmi che non rischiano lo stallo rimane una responsabilità dei programmatori.

1.1.1 Modello di sistema

Un sistema è composto da un numero finito di risorse da distribuire tra più thread in competizione. Le risorse possono essere suddivise in tipi differenti, ciascuno formato da un certo numero di istanze identiche. Cicli di CPU, file e dispositivi di I/O sono tutti esempi di tipi di risorsa. Anche vari strumenti di sincronizzazione, come i lock mutex e i semafori, sono risorse di sistema e sono una tipica causa di situazioni di deadlock.

Prima di adoperare una risorsa, un thread deve richiederla e, dopo averla usata deve rilasciarla. Un thread può richiedere tutte le risorse necessarie, ma senza mai superare il numero massimo di risorse disponibili nel sistema. Normalmente un thread può servirsi di una risorsa soltanto se rispetta la seguente sequenza:

1. **Richiesta**: *il thread richiede la risosra e se non è disponibile il thread deve attendere finché non può acquisire la risorsa*, (`wait`).
2. **Uso**: *il thread può operare sulla risorsa*, (`ready-queue` o `cpu`).
3. **Rilascio**: *il thread rilascia la risorsa.*

La richiesta e il rilascio di risorse avvengono tramite *system call*, ad esempio `open()` e `close()` su un file.

1.1.2 Condizioni necessarie

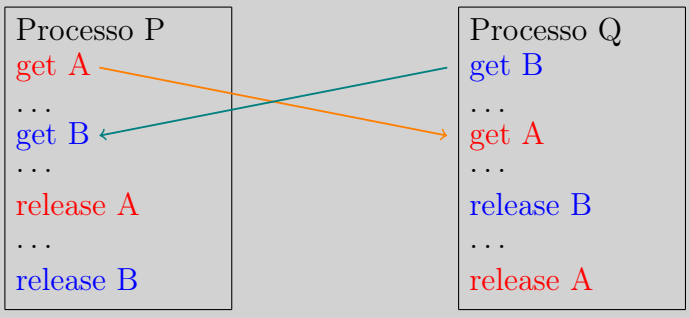
Si può avere una situazione di deadlock se e solo se in un sistema si verificano *contemporaneamente* le seguenti condizioni:

1. **Mutua esclusione**: *almeno una risorsa deve essere non condivisibile.*
2. **Hold & Wait**: *deve esistere un processo che detiene una risorsa e che attende di acquisirne un'altra, detenuta da un altro processo.*
3. **No preemption**: *Le risorse non possono essere rilasciate se non "volontariamente" dal processo che le detiene.*
4. **Attesa circolare**: *Deve esistere un insieme di processi che attendono ciclicamente il liberarsi di una risorsa.*

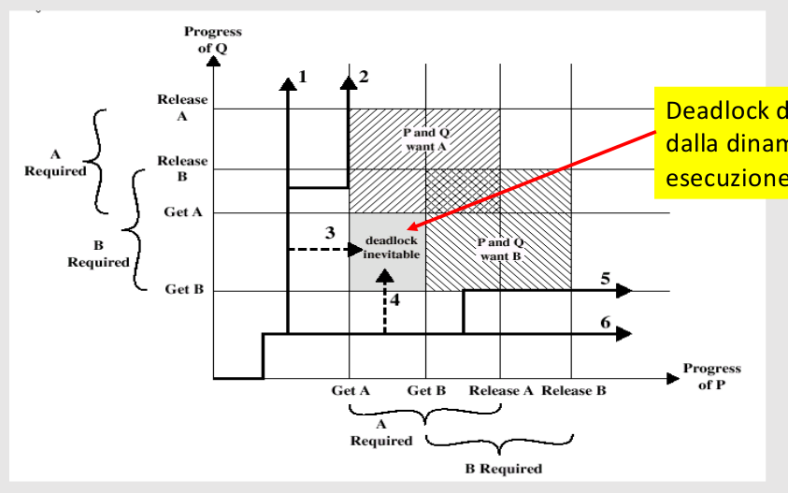
Possiamo notare che la condizione dell'attesa circolare implica la condizione di hold & wait, quindi le quattro condizioni non sono completamente indipendenti.

1.1.3 Esempio di deadlock

- P e Q devono utilizzare A e B in modo esclusivo



In questo caso ci sono sei possibili sequenze di richiesta/rilascio. Il deadlock dipende quindi dalla dinamica dell'esecuzione, comse possiamo vedere nel grafico:

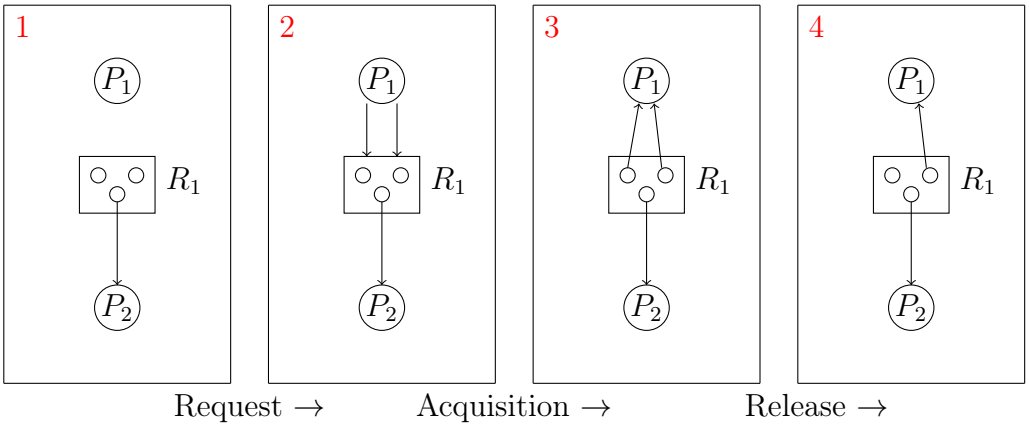


1.2 RAG: Resource Allocation Graph

Le situazioni di stallo si possono descrivere con maggior precisione avvalendosi di una rappresentazione detta *grafo di assegnazione delle risorse*. Si tratta di un insieme di nodi (V) e archi (E), con l'insieme di vertici V composto da due sottoinsiemi: i processi e le risorse. Un arco diretto dal processo P_i alla risorsa R_j ($P_i \rightarrow R_j$) significa che il processo ha richiesto un'istanza della risorsa, ed è attualmente in attesa di essa. Un arco diretto da un'istanza della risorsa R_j verso il processo P_i sta ad indicare che il processo detiene quella risorsa.

Graficamente un processo si rappresenta con un cerchio e le risorse si rappresentano con un rettangolo. Siccome il tipo di risorsa R_j può avere più istanze, queste si rappresentano con dei pallini all'interno del rettangolo R_j.

1.2.1 Esempio di RAG



- $V = \{\{P_1, P_2\}, \{R_1\}\}$
- $E_{iniziale} = \{(R_1, P_2)\}$
- $E_{finale} = \{(R_1, P_1), (R_2, P_2)\}$

1. Nel primo riquadro della figura abbiamo che P_1 non sta usando la risorsa R_1, mentre il processo P_2 detiene un'istanza della risorsa R_1. Nessuno dei due processi è in attesa.

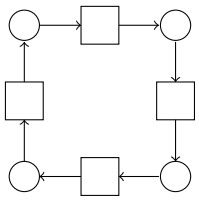
2. Il processo P_1 fa una richiesta per usare le due istanze libere della risorsa R_1. In questo contesto P_2 detiene una risorsa e potrebbe essere in esecuzione oppure nella *ready queue*. Il processo P_1 invece sarà sicuramente in *attesa*, sta aspettando che il sistema operativo conceda le risorse richieste.

3. Il sistema operativo ha concesso le due istanze di R_1 libere al processo P_1, che non è più in attesa ma potrebbe essere nello stato pronto o in esecuzione, cosi come il processo P_2.

4. Nell'ultimo schema il processo P_1 ha rilasciato un'istanza della risorsa R_1, entrambi i processi sono nello stato pronto oppure (solo uno alla volta) in esecuzione.

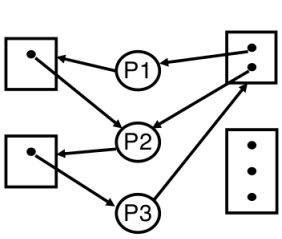
È possibile individuare un eventuale deadlock attraverso il RAG: ci sono tre possibili casi.

- Se non sono presenti cicli, non ci sono deadlock.
- Se è presente un ciclo e le risorse sono presenti in un'unica istanza, siamo sicuramente in deadlock.

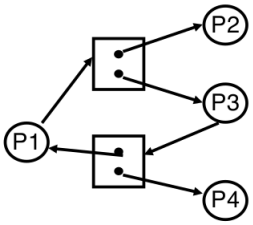


DEADLOCK

- Se è presente un ciclo ma le risorse hanno più istanze ciascuna, potrebbe non esserci un deadlock, dipende dallo schema di allocazione.



DEADLOCK



NO DEADLOCK

1.3 Gestione dei deadlock

Il problema del deadlock si può affrontare in vari modi che comportano uno spreco di risorse più o meno elevato.

Prevenzione statica: evitare che si possa verificare una delle quattro condizioni di stallo, ad esempio negando le risorse ad un processo istanze di risorse libere che potrebbero portare ad una situazione di possibile deadlock.

Prevenzione dinamica: detta *avoidance*, è basata sull'allocazione delle risorse ma non è mai usata poiché richiede una conoscenza troppo approfondita delle richieste di risorse.

Rivelazione e ripristino: chiamata anche *detection and recovery*, permette che si verifichino dei deadlock ma, quando ciò avviene, utilizza metodi di ripristino per portare il sistema al funzionamento normale.

Algoritmo dello struzzo: vista la rarità degli eventi di stallo e il costo elevato per la loro gestione, si può scegliere di non fare nulla. Eventualmente in alcuni casi, se si verifica una situazione di stallo, sarà necessario riavviare il calcolatore. In questo caso non si ha uno spreco di risorse. Naturalmente non è utilizzabile in sistemi *saefty critical*.

1.3.1 Prevenzione statica

Impone una regola a prescindere dall'esecuzione del sistema. L'obbiettivo è di impedire che si verifichi una delle quattro condizioni necessarie per il verificarsi di un deadlock.

Mutua esclusione

Per eliminare la *mutua esclusione* almeno una risorsa deve essere non condivisibile; ma poiché per alcune risorse la mutua esclusione è irrinunciabile, non si possono prevenire in generale le situazioni di stallo negando questa condizione. Inoltre, rimuovendola si avrebbe il rischio di una *race condition*, peggiorando la situazione.

Hold and wait

Per assicurare che la condizione di *hold and wait* non si presenti mai nel sistema, occorre garantire che un processo allochi prima dell'esecuzione tutte le risorse che deve utilizzare oppure che possa ottenere una risorsa solo se non ne ha altre. A causa della natura dinamica della richiesta di risorse questa condizione presenta alcuni problemi: un basso utilizzo di risorse, perché potrebbero rimanere assegnate per lunghi periodi tempo ad un processo che necessita di usarle per un breve periodo, e perché si presenta il rischio di *starvation* per risorse molto popolari richieste da numerosi processi.

Un ulteriore problematica per l'attuazione di questi protocolli è la necessità di sapere tutte le risorse richieste dal processo prima dell'esecuzione dello stesso.

Assenza di prelazione

Per assicurare che questa condizione non sussista, si può impiegare il seguente protocollo. Se un processo che detiene una o più risorse ne richiede un'altra

che non gli si può assegnare immediatamente, allora si esercita la prelazione su tutte le risorse detenute dal processo. Si ha cioè il rilascio implicito di queste risorse e il processo esce dallo stato di attesa solo quando può ottenere sia le vecchie risorse che quella nuova che sta richiedendo.

Un altro sistema è quello di verificare la disponibilità delle risorse richieste: se sono disponibili vengono assegnate, se non lo sono, si verifica se sono assegnate ad un processo in attesa. In tal caso si prelazionano dal processo in attesa e si assegnano al richiedente. Se le risorse non sono disponibili né sono possedute da un processo in attesa, il richiedente viene messo in attesa. Un processo si può avviare nuovamente solamente quando riceve tutte le risorse richieste e quelle eventualmente prelazionate mentre era in attesa.

Questo protocollo è applicato spesso per risorse il cui stato si può facilmente salvare e recuperare, mentre non si può appilcare in generale a risorse come mutex o semafori.

Attesa circolare

Le tre precedenti condizioni sono generalmente difficili da evitare nella maggior parte delle situazioni. Tuttavia, l'*attesa circolare* offre un'opportunità per una soluzione pratica che renda non valida una delle condizioni di deadlock. Un metodo per evitare l'attesa circolare consiste nell'imporre un ordinamento totale all'insieme di tutti i tipi di risorse e imporre che ciascun processo richieda le risorse in ordine crescente. Ovvero, assegnamo una priorità ad ogni risorsa:

$$F : R \to \mathbb{N}$$

$$F(R_0) < F(R_1) < \cdots < F(R_n)$$

Un processo può richiedere risorse solo in ordine crescente di priorità, quindi l'attesa circolare diventa impossibile poichè:

- Se $P_0 \to R_0 \to P_1 \to \cdots \to R_{n-1} \to P_n \to R_n \to P_0$
- allora, $F(R_0) < F(R_1) < \cdots < F(R_n) < F(R_0)$, impossibile.

Con questo sistema se un processo P_i che detiene le risorse R_1, R_3 ed R_6 richiede la risorsa R_4, non può ottenerla e deve cedere tutte le risorse che detiene, cosi da richiderle in seguito nel corretto ordine. Se sono necessarie più istanze dello stesso tipo di risorsa si deve presentare una singola richiesta per tutte le istanze.

1.3.2 Prevenzione dinamica

Le tecniche di prevenzione statica possono portare ad un basso utilizzo delle risorse perchè mettono vincoli sul modo in cui i processi possono accedere alle risorse. Un diverso approccio è quello della *prevenzione dinamica*. In questo contesto potranno esserci ancora situazioni in cui alcuni processi si vedranno negate delle risorse, ma si tratta di una valutazione caso per caso. Tuttavia, queste tecniche di prevenzione dinamica non sono *ragionevolmente implementabili* in sistemi general puprose perché, per poter effettuare le analisi sulle richieste di risorse il sistema dovrebbe avere una conoscenza del futuro. Questo tipo di prevenzione è possibile farlo in alcune situazioni *application specific*.

I requisiti necessari per l'implementazione di questa tecnica di prevenzione sono la conoscenza delle risorsi disponibili, facilmente ottenibile perchè il sistema ha una quantità di risorse finite; la conoscenza del *caso peggiore* ovvero il numero massimo di risorse richieste da ogni processo durante l'esecuzione, molto più difficile da calcolare. Infine, è necessario sapere lo stato corrente del sistema, per determinare se la concessione di risorse, a fronte di una richiesta, possa portare il sistema in uno *stato unsafe*.

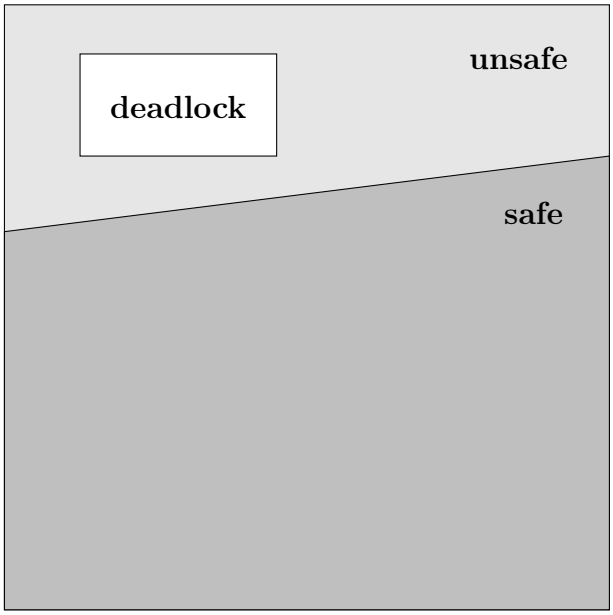
1.3.3 Stato safe, unsafe e deadlock

***Def*:**

Un sistema si trova in uno ***stato safe*** *se esiste una sequenza safe, ovvero se usando le risorse disponibili, può allocare risorse ad ogni processo, in qualche ordine, in modo che ciascuno di essi possa terminare la sua esecuzione.*

Una sequenza di processi $\langle P_1, P_2, \ldots, P_n \rangle$ è una sequenza sicura per lo stato di assegnazione attuale se, per ogni P_i, le richieste che il processo P_i può ancora fare si possono soddisfare impiegando le risorse attualmente disponibili più le risorse possedute da tutti i processi P_j con $j < i$. In questa situazione, se le risorse necessarie al processo P_i non sono disponibili immediatamente, quest'ultimo attenderà che tutti i processi P_j abbiano finito, a quel punto potrà ottenere tutte le risorse necessarie e terminare. Se non esiste una sequenza di questo tipo, lo stato del sistema si dice *unsafe*. Si noti che uno stato *sicuro* non porterà ad uno stallo. Viceversa, quando si ha un deadlock, sicuramente il sistema è in uno stato *non sicuro*; tuttavia non tutti gli stati unsafe sono stati di deadlock.

Spazio degli stati



Esempi di stato safe e unsafe

Supponiamo di avere 3 processi, P_0, P_1, P_2, e 12 istanze di una risorsa di cui 3 sono libere. Siamo in uno stato Safe?

	Richieste	Possedute
P_0	10	5
P_1	4	2
P_2	9	2

Si, esiste una sequenza safe $\langle P_1, P_0, P_2 \rangle$ tale per cui tutti i processi possono terminare.

Se ora P_2 richiede 1 risorsa e gli viene assegnata, siamo ancora in uno stato safe?

	Richieste	Possedute
P_0	10	5
P_1	4	2
P_2	9	3

Non siamo più in uno stato safe. Con le restanti 2 risorse P_1 può eseguire, ma quando termina libera solo 4 risorse: a P_0 ne servono 5 e a P_2 ne servono 6. Deadlock $P_2 \leftrightarrow P_0$.

Deadlock & Risorse

Nel caso della prevenzione il costo è dato da un basso utilizzo delle risorse. Infatti, in alcuni casi delle risorsi disponibili non vengono concesse ai processi per evitare che il sistema entri in uno stato non sicuro.

Con le tecniche di rilevazione e ripristino il sistema concede risorse fino alla massima disponibilità, ma ogni tanto devono sprecare tempo per verificare se si è verificata una situazione di deadlock e, in caso affermativo attuare il recovery e quindi sprecare altro tempo per ritornare ad uno stato precedente.

La prevenzione, che si divide in *statica* e *dinamica*, è necessaria nei sistemi *saefty critical*; le soluzioni di detection & recovery, invece, funzionano in tutte quelle situazioni in cui il deadlock, se avviene, non è catastrofico.

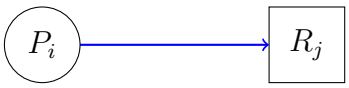
1.4 Algoritmi per la prevenzione dinamica

Per implementare la *prevenzione dinamica* abbiamo due alternative. La prima prevede l'utilizzo di un algoritmo con RAG, ma funziona solo se c'è una sola istanza per risorsa. La seconda alternativa usa l'*algoritmo del banchiere*, che ha il vantaggio di funzionare con qualsiasi numero di istanze di risorse, ma è più complicato.

1.4.1 Algoritmo con RAG

Questo algoritmo prevede di aggiungere ai *resource allocation graph* un nuovo tipo di arco: l'arco di *richiesta* (*claim edge*). Otteniamo quindi tre tipi di archi:

- *Arco di richiesta*: va dal processo P_i alla risorsa R_j e si indica con una freccia piena, il processo è in attesa di ottenere una istanza di quella risorsa.



- *Arco di possesso*: va dalla risorsa R_j al processo P_i che la detiene. Si indica con una freccia piena.

- *Arco di reclamo*: si indica con una freccia tratteggiata che va dal processo P_i alla risorsa R_j. Significare che in un qualche momento futuro, il processo P_i vorrà una istanza della risorsa R_j.

Questo algoritmo richiede che le risorse devono essere reclamate a priori nel sistema. Ciò significa che prima di eseguire il processo P_i, tutti i suoi archi di reclamo devono essere già inseriti nel RAG.

Supponendo che il processo P_i richieda la risorsa R_j, la richiesta può essere soddisfatta solo se la conversione dell'arco di richiesta $P_i \rightarrow R_j$ nell'arco di assegnazione $R_j \rightarrow P_i$ non causa la formazione di un ciclo nel grafo. Possiamo verificare la condizione di sicurezza con un algoritmo di rilevamento dei cicli, che richiede un numero di operazioni dell'ordine di n^2, dove n è il numero dei processi del sistema.

Se non esiste alcun ciclo, l'assegnazione della risorsa lascia il sistema in uno stato sicuro. Se invece si trova un ciclo, l'assegnazione conduce il sistema in uno stato non sicuro e il processo P_i deve attendere.

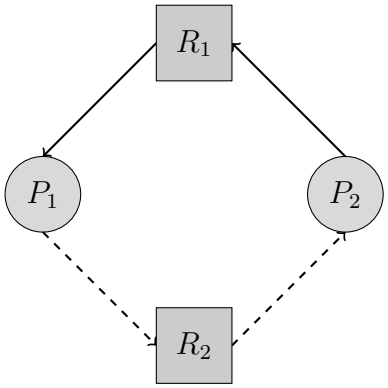


Fig. 1. Stato sicuro.

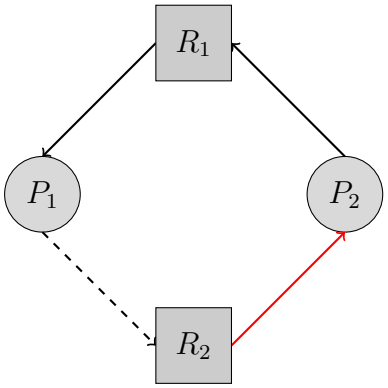


Fig. 2. Stato non sicuro.

In *Fig. 1* siamo in uno stato safe, ma se al processo P_2 venisse concessa la risorsa R_2 come da *Fig. 2*, si può notare che si creerebbe un ciclo nel grafo. Ciò indica che il sistema è in uno stato non sicuro. Se, a questo punto, P_1 richiedesse R_2, si avrebbe un deadlock.

1.4.2 Algoritmo del banchiere

L'algoritmo con grafo di assegnazione delle risorse non si può applicare ai sistemi di assegnazione delle risorse con più istanze di ciascun tipo di risorsa. L'algoritmo applicabile a sistemi con più istanze per tipo di risorse si chiama *algoritmo del banchiere*. L'idea è che il banchiere non deve mai distribuire tutto il denaro di cui dispone, altrimenti non potrebbe soddisfare le richieste future dei suoi clienti.

Quando si presenta al sistema, un nuovo processo deve dichiarare il numero massimo delle istanze di ciascun tipo di risorsa di cui potrà aver bisogno.

Questo numero non può superare il numero totale delle risorse del sistema. Quando un utente richiede un gruppo di risorse, si deve stabilire se l'assegnazione di queste risorse lasci il sistema in uno stato sicuro. Se si rispetta tale condizione, si assegnano le risorse, altrimenti il processo deve attendere che qualche altro processo ne rilasci un numero sufficiente.

L'algoritmo del banchiere richiede la gestione di alcune strutture dati che codificano lo stato di assegnazione delle risorse del sistema. Sia n il numero massimo di processi del sistema e m il numero dei tipi di risorsa. Sono necessarie le seguenti strutture dati:

- `int available[m]`: un vettore di lunghezza m che indica il numero delle istanze disponibili per ciascun tipo di risorsa R_i.
- `int max[n][m]`: matrice $(n \times m)$ che definisce la richiesta massima di ciascun processo; `max[i][j] = k` significa che il processo P_i può richiedere un massimo di k istanze della risorsa R_j.
- `alloc[n][m]`: Una matrice $n \times m$ dell'allocazione corrente delle risorse assegnate a ciascun processo; `alloc[i][j] = k` significa che al processo P_i sono assegnate k istanze della risorsa R_j.
- `need[n][m]`: Una matrice $n \times m$ che indica la necessità residua di risorse relativa ad ogni processo; `need[i][j] = k` significa che il processo P_i, per completare, può aver bisogno di altre k istanze della risorsa R_j. Notare che `need[i][j] = max[i][j] - alloc[i][j]`.

1.4.3 Algoritmo di allocazione (P_i)

Questo algoritmo simula cosa accadrebbe se il S.O. soddisfasse la richiesta del processo P_i. Prima di simulare l'assegnamento delle risorse, l'algoritmo esegue due verifiche:

1. Controlla se il processo ha diritto alle risorse richieste, ovvero se la richiesta supera il massimo di risorse dichiarate dal processo all'inizio dell'esecuzione.
2. Verifica se ci sono risorse disponibili per soddisfare la richiesta, altrimenti mette in attesa il processo.

Se entrambe le condizioni sono verificate significa che il processo è idoneo a ricevere le risorse richieste. In questo caso, l'algoritmo simula l'assegnamento delle risorse, dopodiché verifica se lo stato è ancora sicuro. Se non è questo il caso, viene effettuato un rollback alla situazione precedente la simulazione e il processo viene messo in attesa.

Algoritmo di allocazione

```
/* Richieste del processo P i-esimo */
void request(int req_vec[]) {
  /* Verifico se la richiesta supera
   * il massimo preventivato */
  if (req_vec[] > need[i][]) {
    error();
  }

  /* Verifico la disponibilit\‘a di risorse */
  if (req_vec[] > available[]) {
    wait();   /* Attendo che si liberino risorse */
  }

  /* Simulazione dell'assegnamento delle risorse */
  available[] = available[] - req_vec[];
  alloc[i][] = alloc[i][] + req_vec[];
  need[i][] = need[i][] - req_vec[];

  /* Se non sono in uno stato sicuro, eseguo il
   * rollback e ripristino il vecchio stato */
  if (!state_safe()) {

    available[] = available[] + req_vec[];
    alloc[i][] = alloc[i][] - req_vec[];
    need[i][] = need[i][] + req_vec[];
    wait();
  }
}
```

1.4.4 Algoritmo di verifica dello stato

L'algoritmo utilizzato per la verifica dello stato del sistema si può descrivere come segue:

1. Siano *work* e *finish* i vettori di lunghezza rispettivamente m e n, si inizializza `work = available` e `finish[i] = FALSE`, per $i \in [0, n-1]$;

2. si cerca un indice i tale che valgano contemporaneamente le seguenti relazioni: `finish[i] == TRUE` e `need[i] <= work`. Se tale i non esiste, si esegue il passo 4;

3. `work = work + alloc`$_i$ e `finish[i] == TRUE`. Si va al passo 2;

4. se `finish[i] == TRUE` per ogni i, allora il sistema è in uno stato sicuro.

Per determinare se uno stato è sicuro tale algoritmo può richiedere un numero di operazioni dell'ordine di $m \times n^2$. L'algoritmo ha *complessità* $O(m * n^2)$.

Algoritmo di verifica dello stato

```
boolean state_safe() {
  int work[m] = available[];
  boolean finish[n] = (FALSE, /*... */, FALSE);
  int i;
  while (finish != (TRUE, /* ... */, TRUE)) {
    /* cerca un Pi che NON abbia terminato e che possa
     * completare con le risorse disponibili in 'work' */
    for (i = 0; (i < n) && (finish[i] || (need[i][] > work[])
   ); i++);
    if (i == n) {
      return FALSE; /* nessuna sequenza safe trovata */
    } else {
      /* Emulazione che l'i-esimo processo abbia
       * terminato e liberato le sue risorse */
      work[] = work[]+alloc[i][];
      finish[i] = TRUE;
    }
  }
  return TRUE;
}
```

1.4.5 Esempio di algoritmo del banchiere

Consideriamo di avere un sistema nello stato seguente: cinque processi, P_0, P_1, P_2, P_3, P_4 e tre risorse suddivise in 10 istanze di A, 5 istanze di B e 7 istanze di C. Al tempo T_0 il sistema è nello stato mostrato in tabella:

	Allocation	Max	Available	Need
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

In questo caso il sistema è in uno stato safe. Sequenza safe: $\langle P_1, P_3, P_4, P_2, P_0 \rangle$.

Se al tempo T_1, il processo P_1 richiede $(1, 0, 2)$ ci troviamo ancora in uno stato sicuro? Applichiamo l'algoritmo passo passo e verifichiamolo.

Per prima cosa verifichiamo se il processo sta richiedendo più risorse del suo massimo dichiarato.

```
/* req_vec[] = {1, 0, 2}; need[1][] = {1, 2, 2}; */
if (req_vec[] > need[1][]) {
  error();
}
```

Si può facilmente notare che $(1, 0, 2) < (1, 2, 2)$, il processo non sta chiedendo più risorse di quelle a lui dovute.

Proseguendo col secondo controllo si ha:

```
/* req_vec[] = {1, 0, 2}; available[1][] = {3, 3, 2}; */
if (req_vec[] > available[1][]) {
  wait();
}
```

Anche in questo caso il processo può proseguire. Il numero di richieste non supera il massimo di risorse disponibili: $(1, 0, 2) < (3, 3, 2)$.

Ora simuliamo l'assegnazione di risorse al processo P_1:

```
/* available[] = {3, 3, 2}; need[1][] = {1, 2, 2}; */
/* alloc[1][] = {2, 0, 0}; req_vec[] = {1, 0, 2}; */
available[] = available[] - req_vec[];
alloc[1][] = alloc[1][] + req_vec[];
need[1][] = need[1][] - req_vec[];
```

A questo punto siamo in una nuova situazione, descritta dalla tabella seguente:

	Allocation	Max	Available	Need
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	2 3 0	7 4 3
P_1	3 0 2	3 2 2		0 2 0
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Se applichiamo l'algoritmo di verifica dello stato, siamo ancora in una situazione sicura? Possiamo verificarlo applicando l'algoritmo di verifica di uno stato.

```
int work[] =  available[];  /*= {2, 3, 0} */
```

- i $= 0$; `finish[0] = FALSE; need[0][] > work[]`, $(7, 4, 3) > (2, 3, 0)$;
- i $= 1$; `finish[1] = FALSE; need[1] < work[];` P_1 potrebbe terminare, quindi aggiorno la variabile `work[]` e imposto `finish[1]` a *true*, per simulare che P_1 abbia terminato.

```
int work[] =  work[] + alloc[1][];  /* {5, 3, 2} */
finish[1] = TRUE;
```

A questo punto, si deve ripetere il procedimento dal processo P_0, saltando P_1 perché risulta terminato nella simulazione. È possibile anche continuare la scansione e proseguire con l'indice successivo, ovvero P_2.

- i = 2; `finish[2] = FALSE; need[2] > work[];`
- i = 3; `finish[3] = FALSE; need[3] < work[];` P_3 è compatibile con work, simuliamo la sua terminazione aggiornando work e finish di conseguenza.

```
work[] = work[] + alloc[3]; /* = {7, 4, 3} */
finish[3] = TRUE;
```

Proseguiamo la scansione con il processo P_4: i = 4; `finish[4] = FALSE; need[4] < work[]`. Simuliamo la sua terminazione ed aggiorniamo i valori di conseguenza.

```
work[] = work[] + alloc[4]; /* = [7 4 5] */
finish[4] = TRUE;
```

A questo punto si esce dal ciclo `for` e si ritorna all'inizio, ripartendo dal processo P_0. Si nota che ora P_0 può terminare perché `finish[0] = FALSE` e `need[0] < work[]`.

```
work[] = work[] + alloc[0]; /* = {7, 5, 5} */
```

A questo punto anche P_2 può terminare; `need[2] < work[]`. Abbiamo ottenuto una sequenza safe: $\langle P_1, P_3, P_4, P_0, P_2 \rangle$.

1.5 Rilevazione & Ripristino

Se un sistema non si avvale di un algoritmo per prevenire o evitare lo stallo è possibile che un deadlock si verifichi. In un ambiente di questo genere, il sistema può fornire un algoritmo di *rilevamento* di una situazione di stallo e un algoritmo di *ripristino del sistema* dalla condizione di stallo.

Occorre notare che uno schema di rilevamento e ripristino richiede un *overhead* che include non solo i costi per la memorizzazione delle informazioni necessarie e per l'esecuzione dell'algoritmo di rilevamento, ma anche i potenziali costi dovuti alle perdite di informazioni connsesse al ripristino da una situazione di stallo.

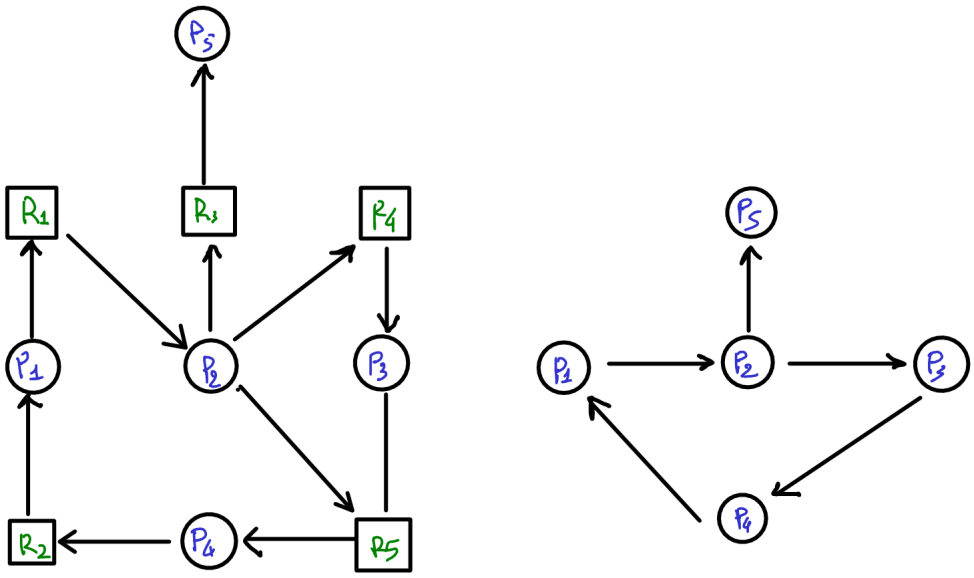
È importante anche sapere che non è necessario prevedere il futuro: se il sistema dispone delle risorse le assegna, se non ci sono abbastanza risorse il processo che le richiede va in wait; se il sistema va in deadlock si valuta in seguito.

1.5.1 Rilevazione e ripristino con RAG

Se tutte le risorse hanno una sola istanza si può definire un algoritmo di rilevamento che fa uso di una variante del RAG, detta *grafo d'attesa*, ottenuta dal grafo di assegnazione togliendo i nodi dei tipi di risorse e componendo gli archi tra i processi.

Più precisamente, un arco da P_i a P_j del grafo d'attesa implica che il processo P_i attende che il processo P_j rilasci una risorsa di cui ha bisogno.

Come in precedenza, nel sistema esiste uno stallo se e solo se il grafo d'attesa contiene un ciclo. Per individuare le situazioni di stallo, il sistema deve *mantenere aggiornato* il grafo d'attesa e *invocare periodicamente* un algoritmo che cerchi un ciclo.



Esempio di RAG a sinistra, e del *grafo d'attesa* corrispondente a destra. In figura un esempio di RAG a sinistra, e del *grafo d'attesa* corrispondente a destra.

1.5.2 Algoritmo di rilevazione

Se nel sistema si hanno più istanze per tipo di risorsa non è possibile applicare il modello con RAG. In questi casi si utilizza un algoritmo simile a quello del banchiere.

L'algoritmo di rilevazione si basa sull'esplorazione di ogni possibile sequenza di allocazione per i processi che non hanno ancora terminato. Se la

sequenza va a buon fine, il sistema è ancora in uno stato safe, quindi non c'è deadlock.

Per questo algoritmo necessitiamo di alcune strutture dati:

```
int available[m];  /* num. istanze di Ri disponibili */
int alloc[n][m];   /* matrice di allocazione corrente */
int req_vec[n][m]; /* matrice delle richieste */
```

La sequenza safe in questo caso esiste se il sistema riesce a soddisfare le richieste attuali (sulla matrice delle richieste).

Codice algoritmo

```
int work[m] = available[m];
bool finish[] = (FALSE, /*...*/ , FALSE);
bool found = TRUE;
while (found) {
  found = FALSE;
  for (i = 0; i < n && !found; i++) {
    /* cerca un Pi con richiesta soddisfacibile */
    if (!finish[i] && req_vec[i][] <= work[]) {
      /* assume ottimisticamente che Pi esegua fino al
       * termine e che quindi restituisca le risorse;
       * se cosi non fosse il possibile deadlock verra'
       * evidenziato alla prossima esecuzione
       * dell'algoritmo. */
       work[] = work[] + alloc[i][];
       finish[i] = TRUE;
       found = TRUE;
    }
  }
}
/* se finish[i] = FALSE per un qualsiasi i,
 * Pi si trova in deadlock */
```

La complessità di questo algoritmo è in $O(m * n^2)$.

1.5.3 Esempio di rilevazione

Dato un sistema con cinque processi, P_0, P_1, P_2, P_3, P_4 e tre tipi di risorsa suddivisi in 7 istanze della risorsa A, 2 istanze della risorsa B e 6 istanze della risorsa C. Al tempo T_0 il sistema è nel seguente stato:

	Alloc.	Req.	Avail.
	$A\ B\ C$	$A\ B\ C$	$A\ B\ C$
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Se la colonna *Available* è tutta a 0 e se tutte le righe della colonna *Request* avessero almeno un valore diverso da 0 saremmo in deadlock. In questo caso abbiamo P_0 e P_2 che possono essere in *ready queue* oppure in *CPU*.

Verifichiamo se il sistema è in deadlock. Al tempo T_0 abbiamo che:

- $work = (0, 0, 0)$;
- i =0;

```
req_vec[0] = (0,0,0) <= work[];    /* OK */
work = work + (0,1,0);
```

- i=1;

```
req_vec[1] = (2,0,2) <= work[]; /* NO */
```

- i =2;

```
req_vec[0] = (0,0,0) <= work[];    /* OK */
work = work + (3,0,3);
```

- Al termine dell'algoritmo si ottiene che la sequenza $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ darà `finish[i] = TRUE` per ogni i.

Supponiamo ora che il processo P_2 richieda un'altra istanza di tipo C. Lo stato viene modificato come segue:

	Alloc.	Req.	Avail.
	$A\ B\ C$	$A\ B\ C$	$A\ B\ C$
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Ora il sistema è in stallo. Anche liberando le risorse di P_0, *Available* non ha sufficienti risorse per soddisfare le richieste degli altri processi. Il sistema va in deadlock: $\langle P_1, P_2, P_3, P_4 \rangle$.

1.5.4 Ripristino

L'algoritmo di rilevazione può essere chiamato secondo diversi paradigmi:

- Dopo ogni richiesta non soddisfatta e se la *ready queue* è vuota. Questo metodo necessità anche di verificare se la ready queue è vuota, aggiungendo tempo "sprecato".

- Ogni N secondi, ci si accorge però che il deadlock non è molto frequente, e questo metodo porterebbe ad uno spreco di risorse e tempi di CPU.

- Quando l'utilizzo della CPU scende sotto una soglia T.

Una volta rilevato uno stallo, questo si può affrontare in due modi. Il primo prevede la *terminazione* di uno o più processi, per interrompere l'*attesa circolare*; il secondo consiste nell'esercitare la prelazione su alcune risorse in possesso di uno o più processi in stallo.

Terminazione dei processi

Per eliminare le situazioni di stallo attraverso la terminazione di processi si possono adoperare due metodi:

- **Terminazione di tutti i processi in stallo**. Operazione molto onerosa ma che interrompe *sicuramente* il deadlock.

- **Terminazione di un processo alla volta**. Procedendo fino all'eliminazione del ciclo di stallo e, di conseguenza, dopo aver terminato ogni processo si deve invocare un algoritmo di rilevamento per stabilire se esistono ancora processi in stallo, aumentando l'overhead.

Prelazione di risorse

Per eliminare un deadlock utilizzando la *prelazione sulle risorse* si devono considerare alcuni fattori:

1. **Selezione di una vittima**. Occorre stabilire quali risorse e quali processi si devono sottoporre a prelazione. È necessario stabilire l'ordine di prelazione allo scopo di minimizzare i costi. I fattori di costo possono includere parametri come il numero di risorse possedute da un processo in deadlock e la quantità di tempo già spesa durante l'esecuzione del processo.

2. **Rollback**. Poiccé un processo sottoposto a prelazione non può proseguire normalmente, è necessario ricondurlo ad un precedente stato *safe* dal quale può poi essere riavviato.

3. **Total rollback**. In generale è difficile determinare quale sia uno stato sicuro, la soluzione più semplice è di terminare il processo prelazionato e riavviarlo da zero.

4. **Rischio di starvation**. Se vengono prelazionato sempre lo stesso processo, il rischio che questo non possa terminare e vada in starvation aumenta. La soluzione a questo problema è di considerare il numero di *rollback* nei fattori di costo del punto 1.

1.6 Conclusione

- Un deadlock si verifica quando, in un insieme di processi, ciascun processo attende un evento che può essere causato solo da un altro processo dell'insieme.

- Vi sono quattro condizioni necessarie per lo stallo: (1) mutua esclusione, (2) possesso e attesa, (3) assenza di prelazione e (4) attesa circolare. Una situazione di stallo è possibile solo quando sono verificate tutte e quattro le condizioni.

- Gli stalli possono essere modellati mediante grafi di allocazione delle risorse (RAG) in cui un ciclo indica uno stallo.

- È possibile prevenire gli stalli garantendo che non si verifichi una delle quattro condizioni necessarie per lo stallo. Tra le quattro possibilità, eliminare l'attesa circolare è l'unico approccio pratico.

- Lo stallo può essere evitato usando l'algoritmo del banchiere, che non concede risorse se così facendo si indurrebbe il sistema in uno stato *unsafe* in cui si potrebbe verificare un deadlock.

- Un algoritmo di rilevamento degli stalli può esaminare i processi e le risorse su un sistema di esecuzioe per determinare se un insieme di processi si trova in una situazione di stallo.

- Un sistema può tentare di recuperare da una situazione di stallo interrompendo uno dei processi nell'attesa circolar o esercitando la prelazione sulle risosre che sono state assegnate ad un processo in deadlock.

Gestione della Memoria

2.1 Introduzione

La CPU, attraverso il program counter, va ad indicizzare locazioni di memoria all'interno delle quali è presente la prossima istruzione da eseguire. Si hanno quindi le fasi di *fetch*, nella quale viene l'istruzione inserita instruction register della CPU, poi avviene la fase di *decode*, ovvero la verifica dell'opcode dell'istruzione per determinare cosa fare, avviene l'eventuale recupero degli operandi dalla *memoria*, fase di *execute* ed infine la fase di *write back*, nella quale si va a salvare nella memoria eventuali risultati. Come si può notare la memoria è coinvolta in ogni fase dell'esecuzione di un processo. Ogni processo necessita dunque di un'area di memoria riservata, chiamata *spazio di indirizzamento*.

Uno dei compiti del sistema operativo saraà quindi quello di dividere la memoria, che è una risorsa condivisa, tra tutti i processi che ne fanno richiesta. Una volta che questa zona di memoria è stata allocata per un processo, entra in gioco anche un sistema di protezione: solo il processo a cui è stata concessa la memoria può accedere, a meno che non venga condivisa con altri, ma in quel caso dev'essere messo in atto un meccanismo di mutua esclusione.

Inoltre, esiste il concetto di *memoria virtuale* che aggiunge l'area di *swap* e la possibilità di far credere ad un processo che il suo spazio di indirizzamento sia più ampio di quello realmente concesso.

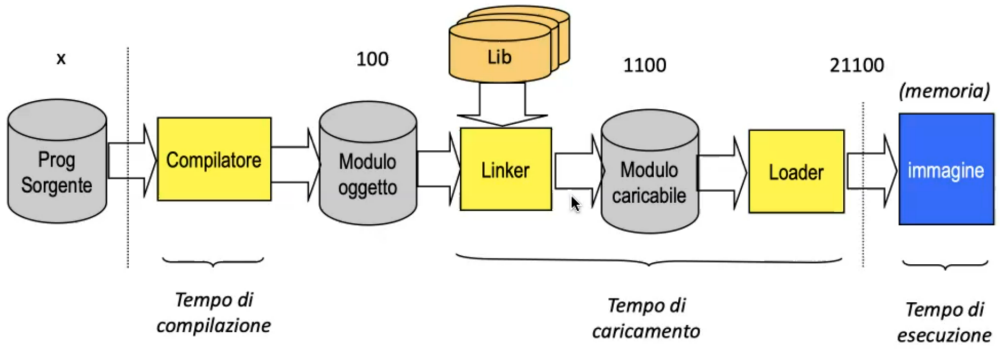
2.1.1 Vincolo

Ogni programma deve essere portato in memoria e trasformato in un processo per essere eseguito. Si deve dunque trasformare un programma sorgente, con *indirizzi simbolici* in un processo caricato in memoria, con *indirizzi fisici*.

2.1.2 Come avviene la trasformazione?

Si crea un mapping tra lo spazio logico del processo e lo spazio fisico in memoria. Prendendo un programma sorgente compilato e dire dove ogni sua parte è situata nella memoria fisica. Il *compilatore* associa gli indirizzi simbolici ad indirizzi rilocabili. Successivamente il *linker e il loader*, associano gli indirizzi rilocabili a indirizzi assoluti, ovvero il vero indirizzo in cui il dato o l'istruzione si troverà in memoria.

Fasi della Trasformazione



Modulo caricabile: file eseguibile

- Gli indirizzi hanno diverse rappresentazioni nelle varie fasi di costruzione di un programma.
- Il collegamento tra indirizzi simbolici e fisici viene detto **binding**.

2.2 Binding dati/istruzioni e indirizzi di memoria

Il concetto di binding degli indirizzi è determinato da come il sistema operativo agisce, da non confondere con la differenza tra linguaggi compilati o interpretati.

2.2.1 Compile time

Il binding avviene al tempo di compilazione, in questo caso saranno assenti linker e loader. Il programma ottenuto avrà un eseguibile che verrà caricato in memoria sempre nella stessa locazione, se si vuole cambiare la posizione

di qualche simbolo si deve ricompilare tutto il programma. Nononostante questa soluzione presenti degli svantaggi, ovvero il contesto non è flessibile, può essere utile per task dedicati o per ottimizzare al massimo: non si ha overhead per la gestione della memoria.

2.2.2 Load time

Il binding avviene al tempo di caricamento, sono quindi necessari linker e loader. È necessario generare codice rilocabile, attraverso indirizzi relativi (ad esempio 128 byte dall'inizio del programma). In questo caso, se cambia l'indirizzo di riferimento, devo ricaricare.

2.2.3 Run time

Il binding è posticipato se il processo può essere spostato durante l'esecuzione in posizioni diverse della memoria. Supponiamo che un processo venga sospeso e il suo stato sia salvato nell'area di *swap*; quando viene ripristinato, se il binding fosse avvenuto a tempo di compilazione, dovrebbe essere ricaricato nell'area di memoria in cui era precedentemente. Questo non è sempre possibile, perché la memoria potrebbe essere in uso da altri processi. Il binding a tempo di esecuzione permette perciò di rilocare il processo in un'area di memoria differente da quella precedente alla sospensione e ripristinare l'esecuzione da dove era stata interrotta.

Swap → Tutto il processo è attivo o non attivo.

Memria virtuale → Alcune parti del processo sono in memoria (RAM) mentre le altre sono su disco.

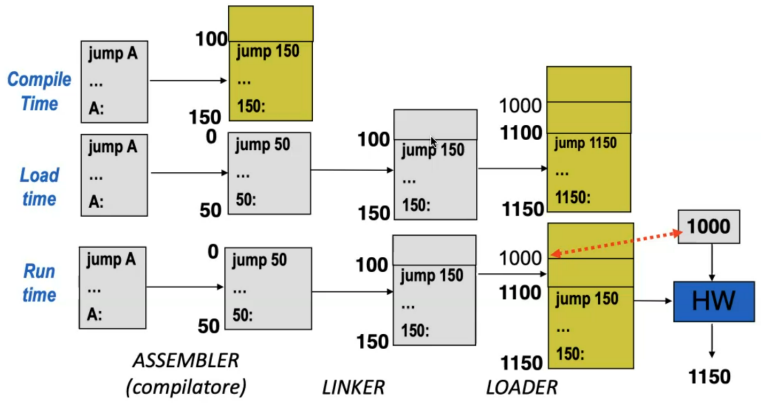
2.2.4 Binding Statico e Binding Dinamico

Si dice *Binding Statico* quando gli indirizzi restano invariati per tutta la vita del processo dopo che ìstato caricato, ovvero quando si utilizza il binding a compile time oppure quello a load time.

Si dice *Binding Dinamico* quando viene utilizzato il binding a tempo di esecuzione. In questo caso un eventuale ricaricamento può avvenire in altre posizioni di memoria.

Il rischio dell'utilizzo di un binding dinamico è che un processo che "esce" dalla memoria, potrebbe non trovare spazio in futuro.

Esempio di binding



2.2.5 Linking statico o dinamico

Anche il linker si divide in statico o dinamico.

- Linking statico: è il linking tradizionale in cui tutti i riferimenti sono definiti prima dell'esecuzione. L'immagine del processo contiene una copia delle librerie usate. Si avrà un eseguibile grande e più lento all'avvio, ma più veloce in esecuzione.

- Linking dinamico: in questo caso il linking delle librerie è posticipato al tempo di esecuzione. Il codice non contiene una copia delle librerire usate ma solo dei riferimenti (stub) per poterle recuperare. Questo eseguibile sarà più rapido all'avvio e occuperà meno memoria, ma sarà più lento in fase di esecuzione, perché dovrà recuperare le librerie ogni volta che saranno necessarie.

2.2.6 Loader statico e dinamico

- Loading statico: tradizionale, ovvero quando tutto il codice è caricato in memoria al tempo di esecuzione. Il vantaggio è una maggior velocità in esecuzione a discapito di un maggior costo per il caricamento ed una memoria occupata maggiore.

- Loading dinamico: caricamento posticipato dei moduli in corrispondenza del primo utilizzo. Il codice non utilizzato non viene caricato, utile per codice con molti casi speciali. Il vantaggio di questo metodo è un avvio molto rapido ed il minor spazio occupato in memoria; di

contro l'esecuzione sarà più lenta se dovesse servire tutto ciò che non è stato caricato.

2.3 Spazi di indirizzamento

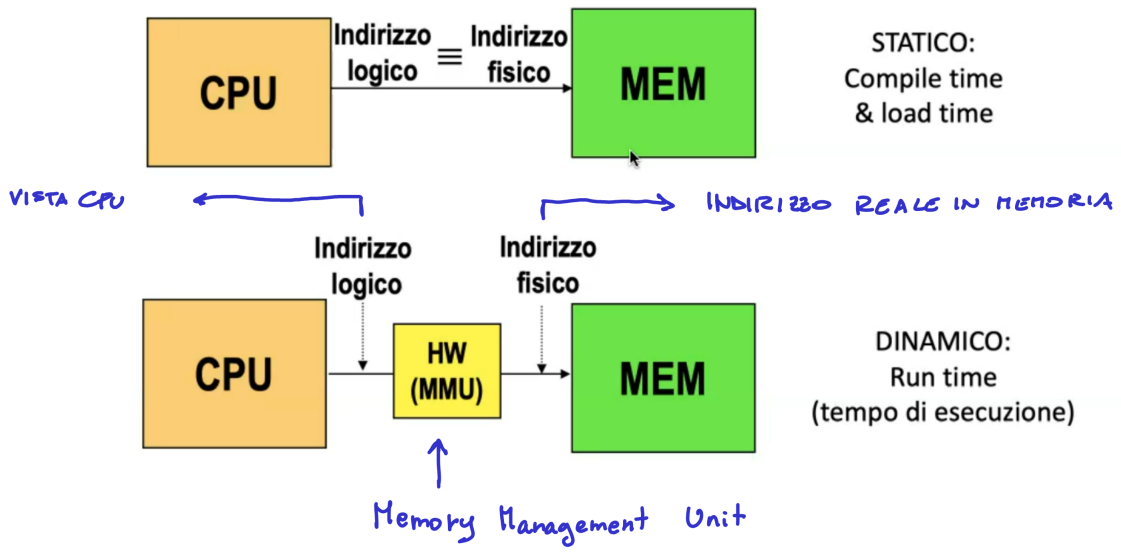
Lo spazio di *indirizzamento logico* dev'essere legato allo spazio di *indirizzamento fisico*.

- L'indirizzo *logico* (o *virtuale*) è sempre contiguo. È generato dalla CPU.
- L'indirizzo *fisico* è lo spazio di indirizzamento visto dalla memoria, non necessariamente è contiguo.

Se il binding è a *compile-time* o a *load-time*, l'indirizzo logico e quello fisico *coincidono*. In questo caso la vista della CPU è uguale alla vista della memoria.

Se il binding è a *run-time* l'indirizzo fisico e logico sono generalmente diversi. In questo caso la vista della CPU sarà sempre continua ma diversa dalla vista della memoria.

2.4 Binding: Statico vs Dinamico



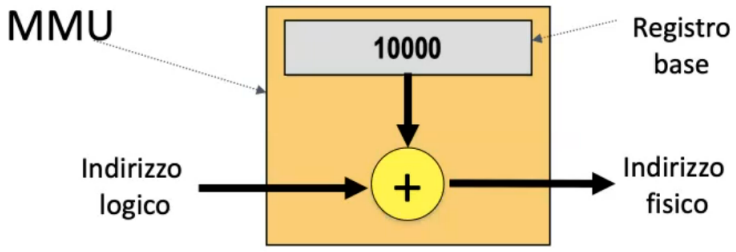
MMU: Hardware che mappa l'indirizzo logico in quello fisico reale.

2.4.1 Memory Management Unit - MMU

È un dispositivo hardware che mappa indirizzi virtuali (logici) in indiriizzi fisici.

Schema base dell'MMU

Il valore del registro di rilocazione è aggiunto a ogni indirizzo generato da un processo e inviato alla memoria.



Registro base → indica il punto d'inizio del blocco contiguo dello spazio di indirizzo del processo.

L'MMU viene aggiornata ad ogni *context-swicth*.

2.5 Considerazioni

1. In un sistema multiprogrammato non noto in anticipo dove un processo si trova in memoria, quindi si esclude il binding a tempo di compilazione.

2. L'esigenza di avere uno *swap* impedisce l'utilizzo di indirizzi rilocati in modo statico, quindi il binding a tempo di caricamento non è possibile.

Di conseguenza, nei sistemi general purpose, si utilizza una *rilocazione dinamica*, mentre la *rilocazione statica* diventa utile in sistemi per applicazioni specifiche in cui le risorse sono limitate o è necessaria molta ottimizzazione.

2.6 Allocazione contigua

I processi sono allocati in memoria in posizioni contigue all'interno di una partizione. La memoria è suddivisa in *partizioni fisse* e *partizioni variabili*.

Per esempio un processo con dimensione dell'immagine di 10kB, occuperà 10kB consecutivi.

2.6.1 Tecnica delle partizioni fisse

La memoria è un insieme di partizioni di dimensioni predefinite, tipicamente di dimensioni diverse. Il Sistema Operativo deve assegnare ai job degli spazi abbastanza grandi per contenerli. Se lo spazio non è sufficiente i job dovranno attendere.

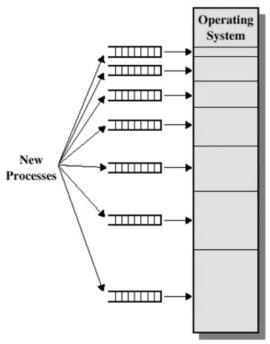
Vincoli:

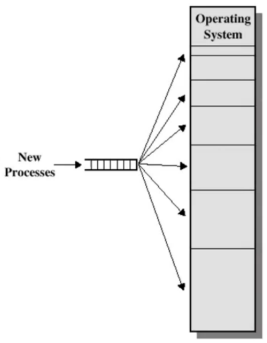
1. Posso avere al massimo n processi in esecuzione contemporaneamente, dove n è il numero di partizioni.
2. Se arriva un processo più grande di ogni partizione, non potrà mai andare in esecuzione.
3. All'arrivo di un processo ci sono due spazi disponibili sufficientemente capienti per farcelo stare, in quale delle partizioni lo inserisco?

Assegnazione della memoria

L'assegnazione della memoria è effettuata dallo scheduling a lungo termine, esistono due opzioni:

- Una coda per ogni partizione
- Coda singola





Una coda per partizione

Il processo viene assegnato alla partizione più piccola in grado di contenerlo. Questo schema è poco flessibile: possono esserci partizioni vuote e job nelle altre code.

Coda unica

Unica coda con diverse politiche di gestione:

- FCFS: una coda di tpo FIFO, molto semplice ma l'utilizzo della memoria potrebbe essere scarso: il primo processo in coda è molto grande e fa attendere altri processi più piccoli.
- Best-fit-only: il sistema operativo scansiona tutta la coda ogni volta che una partizione si libera e scelglie il job con le dimensioni più simili alla partizione. In questo caso si incorre nel rischio di starvation.
- Best-available-fit: simile a quella precedente ma sceglie il primo job che può stare nella partizione. Anche in questo caso si incorre nel rischio di starvation.

Best-fit-only e *Best-available-fit* sono tecniche di "scavalco", in entrambi i casi, oltre al rischio di starvation, aumentano i costi di gestione, e conseguentemente si riducono i tempi di esecuzione della CPU.

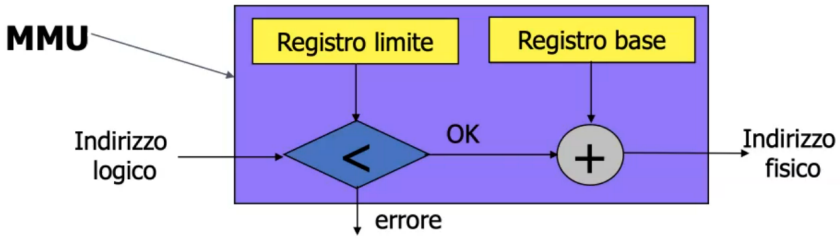
Supporto per la rilocazione dinamica

L'MMU consiste di *registri di rilocazione* per proteggere lo spazio dei vari processi (attivamente e passivamente). I registri contengono:

- Valore dell'indirizzo più basso (registro base o di rilocazione).
- Limite superiore dello spazio logico (registro limite).

Ogni indirizzo logico deve risultare minore del limite.

Supporto per la rilocazione



- L'errore incorre quando si cerca di accedere ad una partizione non detenuta dal processo.
- Indirizzo superiore alla grandezza del processo genera segmentation fault.

Considerazioni

Il vantaggio della tecnica con partizioni fisse è la relativa semplicità. Abbiamo anche numerosi svantaggi: Il grado di multiprogrammazione è limitato dal numero di partizioni. Si ha dello spreco di memoria, detto *frammentazione*, che si divide in frammentazione interna ed esterna.

- **Frammentazione interna:** è si ha quando la dimensione della partizione èpiù grande della dimensione del job.
- **Frammentazione esterna:** quando vi sono partizioni non utilizziate che soddisfano le esigenze dei processi in attesa.

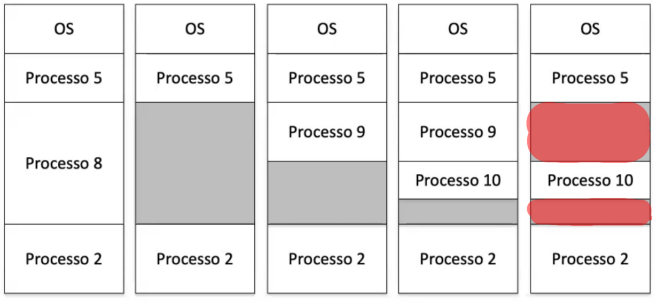
2.6.2 Tecnica delle partizioni variabili

Questa tecnica elimina la frammentazione *interna*, ma con il rischio di aggravare la frammentazione *esterna*.

Con questo schema lo spazio utente è diviso in partizioni di dimensioni variabili, identiche alla dimensione dei processi.

Assegnazione della memoria

La memoria è vista come un unico grande spazio contiguo, all'arrivo di un processo il sistema operativo assegna uno spazio di memoria della dimensione del processo. Cosi via ogni volta che un nuovo processo viene avviato. Al termine dei processi, questi ultimi liberano la memoria a loro assegnata, creando una serie di *buchi* (*hole*) che, se sono vicini possono essere fusi in un buco più grande, se sono distanti non possono essere accorpati. Questi buchi possono essere riallocati al bisogno, ma la memoria evolve in una situazione come quella nell'immagine sottostante.



La frammentazione esterna non è stata eliminata. Se ad esempio ci fosse un processo in attesa che occupa quanto lo spazio evidenziato in rosso, dovrebbe restare in attesa.

First-fit → Primo slot di memoria disponibile che può contenere il processo. Richiede una ricerca parziale, ad esclusione del caso pessimo in cui lo spazio libero è l'ultimo della lista.

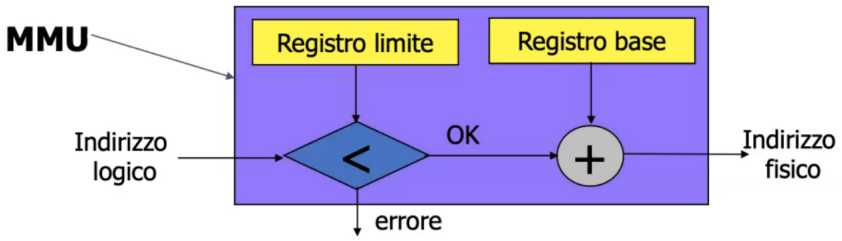
Worst-fit → Ricerca completa del *buco* più grande disponibile, così posso allocare altro nella zona rimanente.

Best-fit → Ricerca completa del *buco* più piccolo disponibile che può contenere il processo. In questo modo si salvaguardano gli spazi maggiori per altri processi più grandi.

First-fit è tipicamente preferita per la velocità.

Supporto per la rilocazione

Come nel caso delle partizioni fisse, l'MMU possiede dei registri di rilocazione per proteggere lo spazio dei vari processi (attivamente e passivamente).



Considerazioni

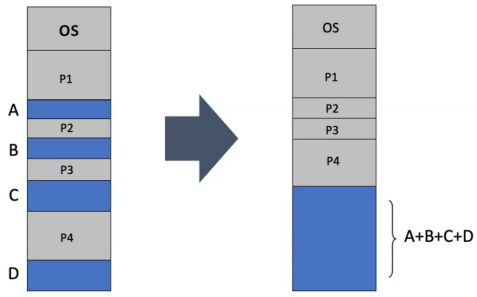
Il vantaggio della tecnica delle partizioni variabili è che rimuove la frammentazione interna per costruzione. Gli svantaggi sono molteplici: rimane la frammentazione esterna.

- Esiste spazio disponibile in memoria, ma non è contiguo.
- Con *first-fit*, dati N blocchi allocati, $0.5 * N$ blocchi vanno persi.

2.6.3 Tecnica della compattazione

Questa tecnica prevede lo spostamento del contenuto della memoria in modo da rendere contigui i blocchi di un processo. È possibile solo se la rilocazione è dinamica e prevede la modifica del registro base. Tuttavia, questo metodo è costoso: *quanta memoria sposto?*

Compattazione della memoria



Non è sufficiente modificare il registro base, bisogna anche effettuare il *trasferimento* della memoria allocata alla nuova zona di allocazione. Questo significa che sono richieste operazioni di spostamento dati, che occupano il bus della memoria, il DMA e, durante gli spostamenti, i processi coinvolti non possono lavorare.

2.6.4 Tecnica del Buddy system

Questa tecnica è un compromesso tra partizioni fisse e variabili. Anche in questo caso l'allocazione dei processi è contigua. La memoria è suddivisa in blocchi di dimensioni 2^k, con $L < k < U$ e:

- 2^L: è il più *piccolo* blocco allocato. (Esempio: 4k)
- 2^U: è il più *grande* blocco allocato. (Esempio: tutta la memoria).

Inizialmente la memoria è tutta disponibile: la lista di blocchi di dimensione 2^U contiene uno solo blocco mentre le altre liste sono vuote. Man mano che la memoria viene allocata è disponibile sottoforma di blocchi di dimensioni 2^k.

Buddy system: allocazione

Quando arriva una richiesta di dimensione s, si cerca un blocco libero con dimensione *adatta* purchè sia pari ad una potenza del 2.

- Se $2^{U-1} < s \leq 2^U$ l'intero blocco di dimensione 2^U viene allocato.
- Altrimenti il blocco 2^U viene diviso in due blocchi di dimensione 2^{U-1}.
- Se $2^{U-2} < s \leq 2^{U-1}$ l'intero blocco di dimensione 2^{U-1} viene allocato.
- Altrimenti, il blocco 2^{U-1} viene diviso in due blocchi di dimensione 2^{U-2}.
- ... e cosi via fino ad arrivare al blocco limite di dimensioni 2^L.

Buddy system: rilascio

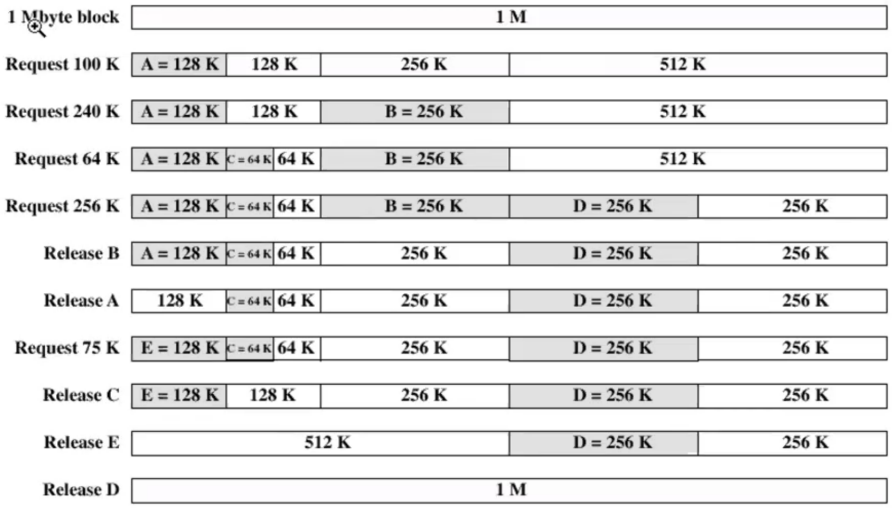
Quando un processo rilascia la memoria, il suo blocco torna a far parte della lista dei blocchi di dimensione corrispondente.

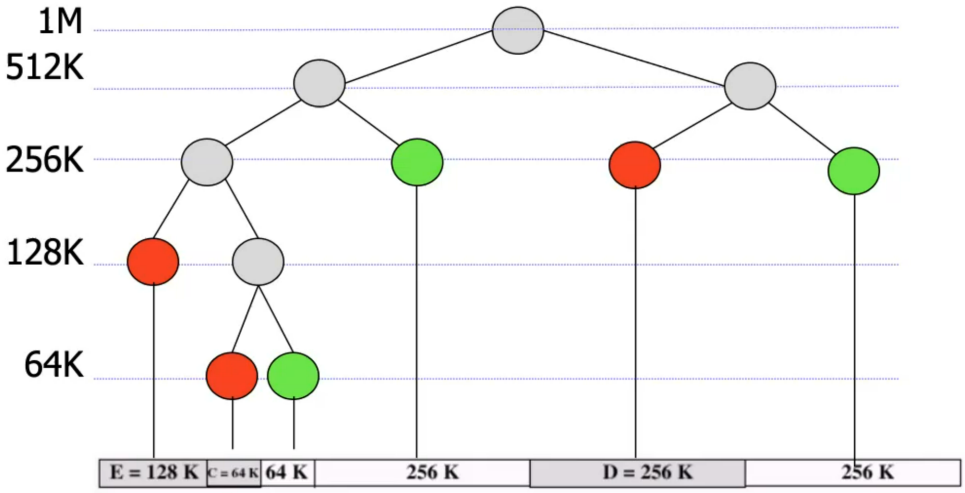
Se si formano due blocchi adiacenti di dimensione 2^k, è possibile compattarli ottenendo un unico blocco libero di dimensione 2^{k+1}.

Vantaggio: La compattazione richiede solo di scorrere la lista dei blocchi di dimensione 2^k, quindi è veloce.

Svantaggio: Frammentazione interna dovuta solo ai blocchi di dimensione 2^L.

Buddy system: esempio





2.7 Paginazione

La *paginazione* è uno schema di gestione della memoria che consente allo spazio di indirizzamento fisico di un processo di essere **non** contiguo, ovvero la memoria fisica può essere allocata dove disponibile. La paginazione permette di *eliminare* la frammentazione esterna ed evita la necessità di compattazione, due problemi che affliggono l'allocazione din memoria contigua. L'implementazione della paginazione è frutto della cooperazione tra il sistema operativo e l'hardware del computer.

- **Memoria fisica:** viene divisa in blocchi detti *frame* di dimensione fissa. (Tipicamente 512byte / 8kb).
- **Memoria logica:** viene divisa in blocchi detti *pagine*, di uguale dimensione dei *frame*.

L'utilizzo di questa tecnica migliora l'*efficacia* ma fa perdere di efficienza. Il sistema operativo avrà più compiti rispetto alle tecniche precedenti.

Per eseguire un programma avente dimensione n pagine, bisogna trovare n frame liberi; per farlo si utilizza una tabella delle pagine (*page table*) per mantenere traccia di quale frame corrisponde a quale pagina. Ciò significa che ogni processo ha una sua tabella delle pagine, che mappa i frame del processo a cui appartiene. Inoltre, viene usata per tradurre un indirizzo logico in un indirizzo fisico.

Durante l'esecuzione di un processo, l'accesso alla memoria, e di conseguenza la consultazione della tabella delle pagine, avviene più volte per istruzione (fetch, decode, execute, write back). In questo caso ogni accesso alla memoria si traduce in due accessi, si ha quindi un peggioramento del 100% rispetto al'allocazione contigua.

Esempio di paginazione

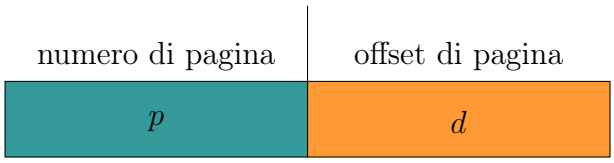
Supponiamo di avere delle pagine di dimensione 1KB e un programma di dimensione 2.3KB. Sono necessarie tre pagine, e dell'ultima pagina si utilizzerà solo 0.3KB. Sperco: 0.7KB, è ancora possibile la *frammentazione interna*, ma solo nell'ultima pagina.

2.7.1 Traduzione degli indirizzi

Quando si deve eseguire un processo si caricano le sue *pagine* nei *frame* disponibili, prendendole dalla memoria ausiliaria o dal file system. Questa idea fornisce grandi funzionalità e ha diverse ramificazioni. Per esempio,

lo spazio degli indirizzi logici è separato dallo spazio degli indirizzi fisici e dunque un processo può avere uno spazio degli indirizzi logici a 64 bit anche se il sistema ha meno di 2^{64} byte di memoria fisica.

Ogni indirizzo generato dalla CPU è diviso in due parti: un *numero di pagina (p)* e un *offset di pagina (d)*.



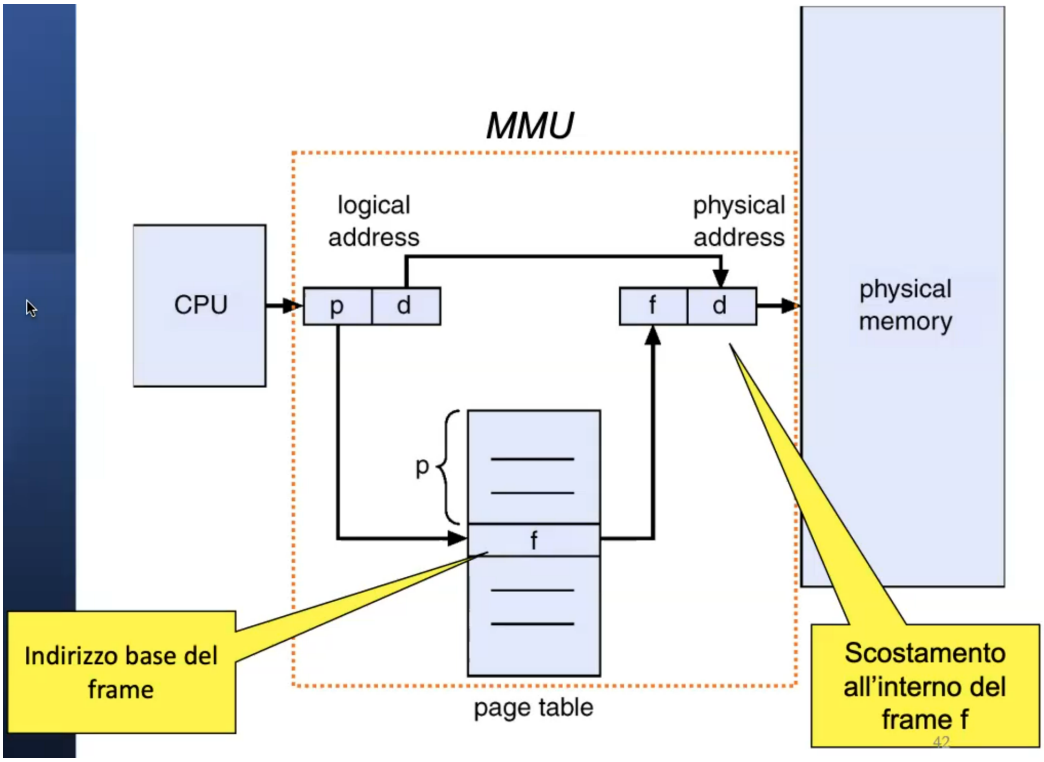
Sicurezza

All'interno della tabella sono presenti dei bit relativi ai *permessi*. Quando la CPU genera un'indirizzo, può essere per accedere ad un'istruzione, che dev'essere *eseguibile*, oppure ad un dato, che non è eseguibile. Vanno implementati meccanismi di *checking*, solo le pagine che contengono codice eseguibile devono essere eseguibili. Allo stesso modo per pagine in sola lettura: non deve essere possibile accedere in scrittura. Viene anche tenuto conto se la pagina è stata modificata, per poterla salvare prima di cestinarla.

Nei sistemi operativi reali si hanno 5 o 6 livelli di paginazione.

Architettura MMU

Il numero di pagina serve come indice per la tabella delle pagine relativa al processo. La tabella delle pagine contiene l'indirizzo di base di ciascun frame nella memoria fisica e l'offset è la posizione nel frame a cui viene fatto riferimento. Perciò, l'indirizzo di base del frame viene combinato con l'offset della pagina per definire l'indirizzo di memoria fisica.



I passaggi adottati dalla MMU per tradurre un indirizzo logico generato dalla CPU in un indirizzo fisico sono:

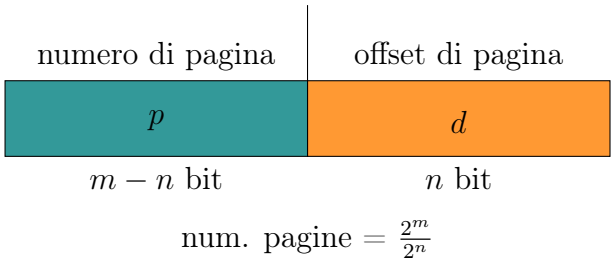
1. Estrarre il numero di pagina p e utilizzarlo come indice nella tabella delle pagine.

2. Estrarre il numero di frame f corrispondente dalla tabella delle pagine.

3. Sostituire il numero di pagina p nell'indirizzo logico con il numero di frame f.

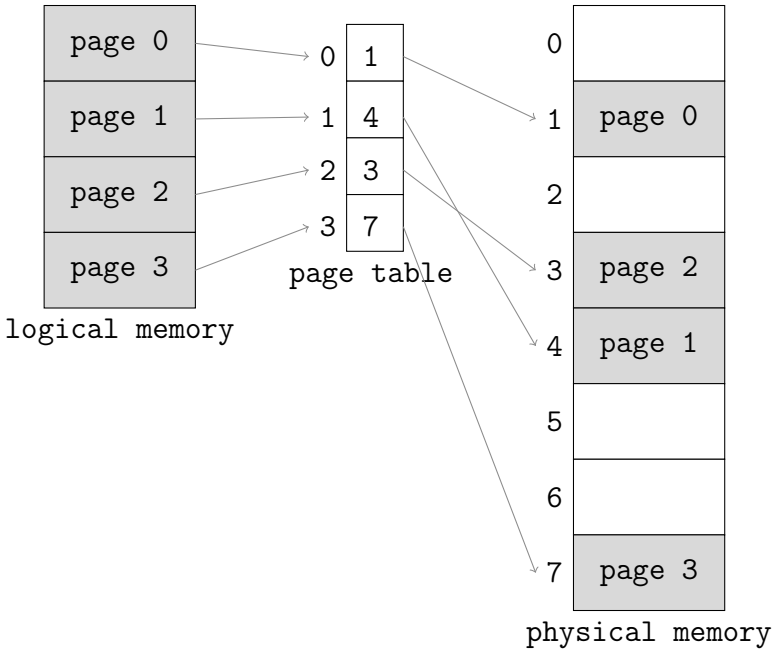
L'offset d non cambia e pertanto non viene sostituito; il numero di frame e l'offset determinano insieme l'*indirizzo fisico*.

La dimensione di una pagina è definita dall'hardware ed è una potenza di 2 compresa fra 4KB e 1GB, a seconda dell'architettura. Questo perché semplifica la traduzione di un indirizzo logico nei corrispondenti numero di pagina e offset di pagina. Ad esempio se la dimensione dello spazio degli indirizzi logici è 2^m e la dimensione di una pagina è di 2^n parole/byte, allora gli $m - n$ bit più significativi di un indirizzo logico indicano il numero di pagina, e gli n bit meno significativi indicano l'offset di pagina. L'indirizzo logico ha quindi la forma seguente:



Esempio di traduzione degli indirizzi

Supponiamo di avere una memoria da 8KB, suddivisa in 8 pagine (o *frame*) da 1KB. Supponiamo anche di avere un processo caricato in memoria composto da 4KB, ovvero 4 pagine.



L'indirizzo logico 936 corrisponde all'indirizzo fisico 1960. Per calcolarlo dobbiamo considerare la mappatura `pagina 0` → `frame 1`. La pagina 0 del programma è mappata nella memoria fisica dall'indirizzo 1024 fino all'indirizzo fisico 2047. L'indirizzo fisico perciò diventa:

$$1024 + \texttt{offset} = 1024 + 936 = 1960$$

2.7.2 Tabella delle pagine

Come detto in precedenza, gli accessi alla tabella delle pagine sono numerosi, quindi l'efficienza è fondamentale. Idealmente, il metodo più veloce sarebbe

quello di caricarla nei registri della CPU, ma questo dipende dalla dimensione della memoria: se la memoria è piccola, questa soluzione sarebbe applicabile ma allungherebbe i tempi di context switch, perché richiede il salvataggio dei registri. L'alternativa è quella di salvare la tabella delle pagine in memoria, attraverso due metodi: il metodo della tabella delle pagine multilivello e il metodo della tabella delle pagine invertita.

Implementazione in memoria

Con questa implementazione la tabella risiede nella meemoria. Vengono utilizzati due registri `PTBR`, (*page-table base register*) che punta alla tabella delle pagine, e il `PTLR`, (*page-table length register* opzionale) che contiene la dimensione della tabella delle pagine. Il context-switch è più breve perché richiede solo la modifica del `PTBR` e del `PTLR` (se presente).

Problema: Ogni accesso a dati/istruzioni richiede due accessi in memoria, uno per accedere alla tabella delle pagine e uno per accedere al dato/istruzione. Questo problema può essere risolto tramite una cache veloce, e dedicata alla traduzine degli indirizzi, detta *translation look-aside buffers (TLB)*. Il suo funzionamento prevede il confronto dell'elemento fornito con il campo chiave di tutte le *entry* contemporaneamente. Il contenuto della cache non conterrà il dato, ma l'indirizzo fisico a cui esso è disponibile.

Il TLB è molto costoso in termini di hardware. Nel TLB perciò, verrà memorizzato solo un piccolo sottoinsieme delle entry della tabella delle pagine, in riferimento alla località spazio-temporale. Inoltre, ad ogni context switch il TLB viene ripulito per evitare mapping di indirizzi errati.

Accesso alla memoria: se la pagina cercata è nel TLB, quest'ultimo restituisce il numero di frame con un singolo accesso, con un $t.richiesto < 10\%$ del tempo richiesto in assenza di TLB. Altrimenti, è necessario accedere alla tabella delle pagine in memoria. L'*hit ratio* ($\alpha = \%$) è la percentuale delle volte in cui una pagina si trova nel TLB. Si rende necessario definire il concetto di *tempo di accesso effettivo*.

Tempo di accesso effettivo - EAT

Il tempo di accesso effettivo è dato dalla seguente formula:

$$EAT = (T_{MEM} + T_{TLB})\alpha + (2T_{MEM} + T_{TLB})(1 - \alpha)$$

dove:

- T_{TLB} = tempo di accesso al TLB;
- T_{MEM} = tempo di accesso a memoria;

Esempio

- $T_{MEM} = 90ns$; $T_{TLB} = 10ns$; $\alpha = 90\%$;
- $EAT = (90 + 10) * 0.9 + ((2 * 90) + 10) * (1 - 0.9) = 109 \approx 1.2T_{MEM}$;

Protezione

La protezione viene realizzata associando *bit di protezione* ad ogni frame/-pagina. Alcuni esempi sono:

- Bit di validità: (*valid-invalid bit*) per ogni entry della tabella delle pagine, se il bit è settato indica che la pagina associata è nello spazio di indirizzamento logico del processo; `invalid` indica che la pagina associata NON è nello spazio di indirizzamento logico del processo.
- Bit di accesso: serve ad indicare se una pagina modificabile o meno (read-only), se è eseguibile o no, eccetera.
- Dirty-bit: indica se una pagina è stata modificata (sporcata) oppure no. Nel caso di dover rimuovere pagine dalla memoria, serve saperlo perché le pagine modificate devono essere salvate.

Oltre alla protezione, possono essere implementate tecniche di *condivisione* del codice. In questo caso vi è un'unica copia fisica (un unico *frame*), ma più copie logiche (una per ogni processo). Il codice *read-only* (rientrante) può essere condiviso tra processi (text editor, compilatori, window manager, ...).

I dati saranno, in generale, diversi da processo a processo.