SISTEMI OPERATIVI

- Programma che controlla l'esecuzione di programmi applicativi e agisce come interfaccia tra le applicazioni e l'hardware del calcolatore

Time Sharing

- Esecuzione della CPU viene suddivisa in un certo numero di **quanti** temporali
- Quando scade un quanto, job corrente si interrompe e l'esecuzione va ad un altro job

Sistemi Paralleli (Tightly coupled)

- Singolo elaboratore che possiede **più unità di elaborazione**
- Risorse dell'elaboratore possono essere condivise (memoria)
- Vantaggi:
  - **Prestazioni** migliori
  - Affidabilità maggiore
- Tassonomia basata sulla struttura:
  - **SIMD**: CPU eseguono all'unisono lo stesso programma su dati diversi
  - **MIMD**: CPU eseguono programmi differenti su dati differenti
- **Symmetric Multiprocessing**:
  - Ogni processore esegue una copia identica del sistema operativo
  - Processi diversi possono essere eseguiti contemporaneamente
- **Asymmetric Multiprocessing**:
  - Ogni processore è assegnato ad un compito specifico
  - Un processore **master** gestisce l'allocazione del lavoro ai processori **slave**

Sistemi Distribuiti (Loosely coupled)

- Sistemi composti da **più elaboratori indipendenti** (con proprie risorse e proprio sistema operativo)
- Ogni processore possiede la propria memoria locale
- Vantaggi:
  - Condivisione di risorse
  - Suddivisione di carico, incremento delle prestazioni

Sistemi Real-Time

- Sistemi per i quali la correttezza del risultato non dipende solamente dal suo valore ma anche dall'istante nel quale il risultato viene prodotto
- Possono essere:
    - **Hard Real-Time:**
        - Se il mancato rispetto dei vincoli temporali può avere effetti catastrofici
        - Ad esempio il controllo di centrali nucleari
    - **Soft Real-Time:**
        - Se si hanno solamente disagi o disservizi

Interrupt

- Meccanismo che permette l'interruzione del normale ciclo di esecuzione della CPU
- Permettono al SO di "intervenire" durante l'esecuzione di un processo utente, per gestire efficacemente le risorse del calcolatore
- Possono essere:
    - Hardware: dovuto da dispositivi di I/O, clock ecc…
    - Software: causato dal programma: system call ecc…

Strutture Dati

- Per svolgere i suoi compiti, un SO ha bisogno di strutture dati per mantenere informazioni sulle risorse gestite
- **Tabelle di memoria:**
    - Allocazione memoria per il SO e per i processi
- **Tabelle di I/O:**
    - Info sullo stato di assegnazione dei dispositivi utilizzati dalla macchina
    - Gestione di code di richieste
- **Tabelle del file system:**
    - Elenco dei file aperti e del loro stato
- **Tabelle dei processi**

Descrittori dei Processi (PCB)

- Struttura dati che contiene le informazioni necessarie per gestire un processo
- Codice da eseguire, dati su cui operare, stack di lavoro per chiamate ecc…
- Tabelle dei processi: contengono descrittori dei processi

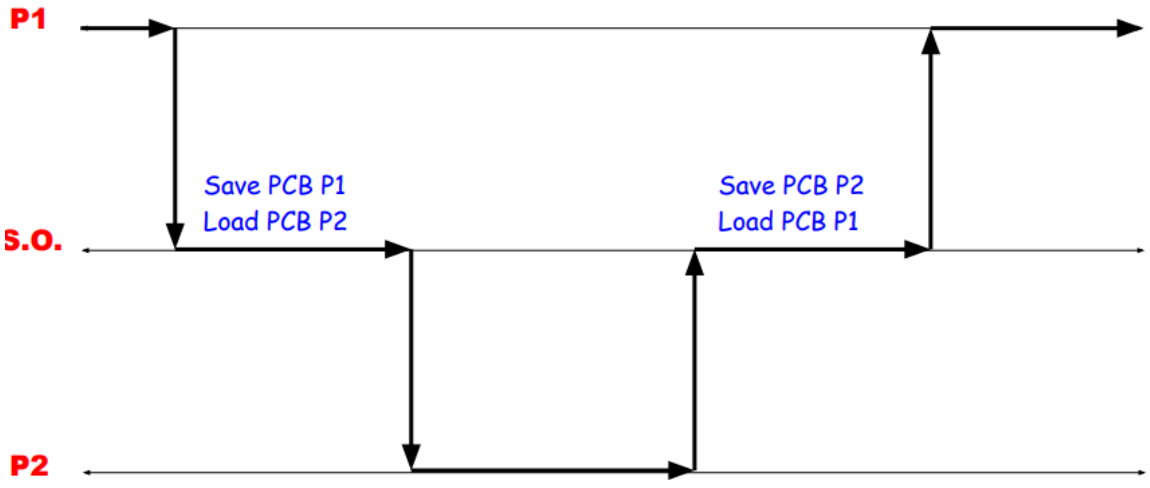
- Tre aree suddividono le info del descrittore:
    - **Di identificazione di processo (pid):**
        - Può essere un indice o un numero progressivo
    - **Di stato del processo:**
        - Registri generali del processore e speciali (PC e di stato)
    - **Di controllo del processo:**
        - Info di scheduling:
            - Stato del processo: in esecuzione, pronto, in attesa
        - info di gestione della memoria:
        - info di accounting
        - info relative alle risorse

Scheduler

- Componente del kernel che gestisce l'**avvicendamento** dei processi
- **Schedule**:
    - Sequenza temporale di assegnazioni delle risorse da gestire richiedenti
- **Scheduling**:
    - Azione di calcolare uno schedule
- **Scheduler**:
    - Componente software che calcola lo schedule

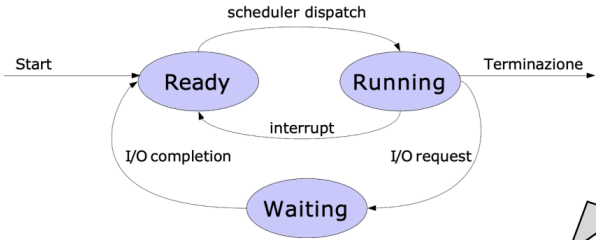
Mode switching e context switching

- Quando avviene un interrupt sw o hw, il processore è soggetto a un **mode switching**
- modalità utente → modalità supervisore
- Durante la gestione dell'interrupt:
    - Viene gestito l'evento e chiamato lo scheduler
    - Se lo scheduler decide di eseguire un altro processo, il sistema è soggetto ad un **context switching**
- Operazioni durante un context switching:
    - Lo stato del processo attuale viene salvato nel PCB corrispondente
    - Lo stato del processo selezionato per l'esecuzione viene caricato dal PCB nel processore



Vita di un Processo

- **Running**:
  - Processo è in esecuzione
- **Waiting**:
  - Processo in attesa di qualche evento esterno (I/O) e non può essere eseguito
- **Ready**:
  - Processo può essere eseguito, ma il processore è impegnato in altre attività



- Quando un processo entra nel sistema, viene posto in una **coda dello scheduler**

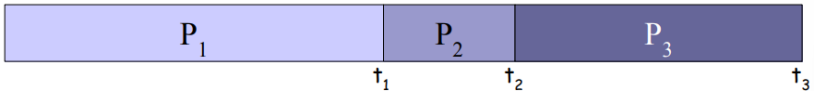
Processi e Thread

- Unità di base di utilizzazione della CPU
- Ogni thread possiede:
  - Copia dello stato del processore
  - Proprio Program Counter
  - Stack separato
- Benefici:
  - **Condivisione** di risorse (memoria)
  - Gestire thread è + **economico** (fare context switching fra thread è - costoso)
- Un sistema operativo può implementare i thread in due modi:
  - **User thread** (più veloce):
    - Vengono implementati da una "**thread library**" a livello utente
    - Thread library fornisce supporto per la creazione, lo scheduling e la gestione dei thread **senza intervento del kernel**
  - **Kernel thread**:
    - Vengono supportati direttamente dal sistema operativo
    - La creazione, lo scheduling e la gestione dei thread sono implementati a livello kernel
- Molti sistemi supportano entrambi, creati così 3 modelli di multithreading:
  - **Many-to-One:**
    - Più user thread vengono mappati su un solo kernel thread
  - **One-to-One:**
    - Un user thread viene mappato su un kernel thread
    - Può creare problemi di scalabilità per il kernel
  - **Many-to-Many:**
    - Ha i benefici di entrambe le architetture

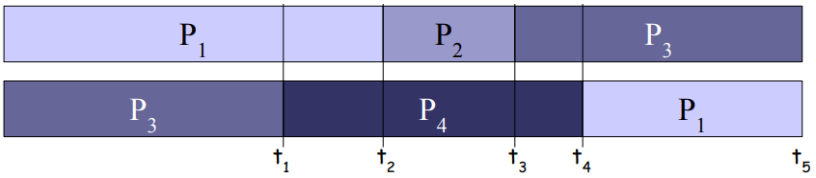
SCHEDULING

Rappresentazione degli Schedule

- **Diagramma di Gantt:**
  - Utilizzato per rappresentare uno schedule



- In questo esempio, la risorsa (es. CPU) viene utilizzata:
  - Dal processo P_1 dal tempo 0 a t_1
  - Viene poi assegnata a P_2 fino al tempo t_2
  - Viene poi assegnata a P_3 fino all tempo t_3
- Diagramma di Gantt **multi-risorsa**:
  - Se si deve rappresentare lo schedule di più risorse (sistema multiprocessore) il diagramma di Gantt risulta composto da più righe parallele



Tipi di Scheduler

- Eventi che possono causare un context switch:
  - Processo passa da stato running a stato waiting
  - Processo passa dallo stato running allo stato ready
  - Processo passa dallo stato waiting allo stato ready
  - Processo termina
- Scheduler **non-preemptive**:
  - Se context switch avvengono solo nelle condizioni 1 e 4
  - Non richiede alcuni meccanismi hardware
- Scheduler **preemptive**:
  - Se context switch possono avvenire in ogni condizione
  - Permette di utilizzare al meglio le risorse

Criteri di Scelta di uno Scheduler

- Utilizzo della CPU: tempo in cui la CPU è occupata ad eseguire processi
- **Throughput**: numero di processi completati per unità di tempo
- Tempo di **turnaround:**
  - Tempo che intercorre dalla sottomissione di un processo alla sua terminazion
- Tempo di attesa: tempo trascorso da un processo nella coda ready
- Tempo di risposta:
  - Tempo che intercorre fra la sottomissione del processo e il tempo di risposta

- Durante l'esecuzione di un processo si alternano attività svolte dalla CPU (**CPU burst**) e periodi di attività di I/O (**I/O burst**)
- I processi con CPU burst molto lunghi sono **CPU bound**
- I processi con I/O burst molto lunghi sono **I/O bound**

Algoritmi di Scheduling

- **First Come, First Served (FCFS)**
  - Il processo che arriva per primo, viene servito per primo
  - Implementato tramite una coda (politica **FIFO**)
  - Elevati tempi medi di attesa e turnaround
  - Processi CPU bound ritardano quelli I/O bound
- **Shortest Job First (SJF)**
  - CPU data a processo ready che ha minima durata del CPU burst successivo
  - Ottimale per tempo di attesa ma impossibile da implementare in pratica
- **Scheduling Round-Robin**
  - Processo non può rimanere in esecuzione per un tempo superiore alla durata del quanto di tempo
  - Processi pronti organizzati con una **coda**
  - Durata del quanto è un parametro critico del sistema:
    - Se breve, deve cambiare il processo attivo più spesso
    - Se lungo, lunghi periodi di inattività di ogni singolo processo
  - Necessario che l'hardware fornisca un **timer (interval timer)**
    - Dispositivo che, attivato con un preciso valore di tempo, è in grado di fornire un interrupt allo scadere del tempo prefissato
- **Scheduling a Priorità**
  - A ogni processo è associata una specifica priorità, lo scheduler sceglie il processo pronto con priorità più alta
  - Le priorità possono essere:
    - Definite dal **sistema operativo**:
      - Vengono usate quantità per calcolare la priorità di un processo
      - Esempio: SJF basato su priorità
    - Definite **esternamente**:
      - Priorità imposte dal livello utente
  - **Priorità statica:**
    - La priorità non cambia durante la vita di un processo
  - **Priorità dinamica:**
    - La priorità può variare durante la vita di un processo
  - **Priorità basata su aging:**
    - Tecnica che incrementa gradualmente la priorità dei processi in attesa
    - Nessun processo rimarrà in attesa per un tempo indefinito perché prima o poi raggiungerà la priorità massima
- **Scheduling a Classi di Priorità**
  - Crea classi di processi simili e assegna a ogni classe una priorità
  - La coda ready viene scomposta in sottocode, una per ogni classe

Scheduling Real-Time

- **Processi periodici:**
  - Se vengono riattivati con una cadenza regolare
- **Processi aperiodici:**
  - Se scatenati da un evento sporadico (allarme rilevatore pericolo)

Diversi tipi di Scheduler di sistemi Real-Time:

- **Rate Monotonic:**
  - Politica di scheduling, valida alle seguenti condizioni:
    - Ogni processo periodico deve completare entro il suo periodo
    - Tutti i processi sono indipendenti
    - La preemption avviene istantaneamente e senza overhead
  - A ogni processo viene data una priorità statica
  - Processi con frequenza più alta hanno priorità più alta
- **Earliest Deadline First (EDF):**
  - Politica di scheduling per processi periodici real-time
  - Viene scelto di volta in volta il processo che ha deadline più prossima

CONCORRENZA

- Un **processo** è un'attività controllata da un programma che si svolge nel processore
- Un processo non è un programma, un processo è dinamico, il programma è statico:
    - Un programma specifica un'insieme di istruzioni e la loro sequenza di esecuzione
    - Un processo rappresenta il modo in cui un programma viene eseguito nel tempo
- **Assioma di finite progress:**
    - Ogni processo viene eseguito ad una velocità finita ma sconosciuta
- Un processo può essere descritto da:
    - La sua **immagine di memoria**:
        - La memoria assegnata e le strutture dati associate al processo
    - La sua **immagine nel processore**:
        - Contenuto dei registri
    - Lo **stato di avanzamento**:
        - Descrive stato corrente del processo (Ready, Waiting)

Cos'è la Concorrenza?

- **Esecuzione Concorrente:**
    - Due programmi si dicono in esecuzione concorrente se vengono eseguiti in parallelo (con parallelismo reale o apparente)
- **Concorrenza**:
    - E' l'insieme di notazioni per descrivere l'esecuzione concorrente di due o più programmi
    - E' l'insieme di tecniche per risolvere i problemi associati all'esecuzione concorrente, quali **comunicazione** e **sincronizzazione**
- Gestione dei processi multipli:
    - **Multiprogramming**:
        - Più processi su un solo processore
        - Parallelismo **apparente**
    - **Multiprocessing**:
        - Più processi su una macchina con processori multipli
        - Parallelismo **reale**
    - **Distributed processing:**
        - Più processi su un insieme di computer distribuiti e indipendenti
        - Parallelismo **reale**

Differenze tra Multiprogramming e Multiprocessing

- In un singolo processore:
  - Processi sono **alternati nel tempo** per dare l'impressione di avere un multiprocessore
  - Si parla di **interleaving**
- In un sistema multiprocessore:
  - Più processi eseguiti su processori diversi. "**alternati nello spazio**"
  - Si parla di **overlapping**
- **Race Condition:**
  - Un sistema di processi multipli presenta una race condition qualora il risultato finale dell'esecuzione dipenda dalla temporizzazione con cui vengono eseguiti i processi
  - Per scrivere un programma concorrente bisogna eliminare la race condition

Interazioni tra Processi

Le modalità di interazione tra processi si classificano in:

- **Processi totalmente "ignari" l'uno dell'altro:**
  - Indipendenti non progettati per lavorare insieme
  - Vivono comunque in un ambiente comune
  - Come interagiscono?
    - Competono per le stesse risorse
    - Devono sincronizzarsi
  - Il sistema operativo:
    - Deve arbitrare competizione con meccanismi di **sincronizzazione**
- **Processi "indirettamente" a conoscenza uno dell'altro:**
  - Condividono risorse al fine di scambiarsi informazioni
  - Interagiscono indirettamente tramite le risorse condivise. Non sanno gli id
  - Come interagiscono?
    - Cooperano per qualche scopo
    - Devono sincronizzarsi
  - Il sistema operativo:
    - Deve facilitare cooperazione con meccanismi di **sincronizzazione**
- **Processi "direttamente" a conoscenza uno dell'altro:**
  - Comunicano uno con l'altro sulla base dei loro id
  - La comunicazione è diretta, basata su scambio di messaggi
  - Come interagiscono?
    - Cooperano per qualche scopo
    - Comunicano informazioni agli altri processi
  - Il sistema operativo:
    - Deve facilitare cooperazione fornendo meccanismi di **comunicazione**

Proprietà

- Una proprietà di un programma concorrente è un attributo che rimane vero per ogni possibile storia di esecuzione del programma stesso

Due tipi di proprietà:

- Proprietà **Safety**:
  - Mostrano che il programma non esegue azioni scorrette
- Proprietà **Liveness**:
  - Mostrano che il programma progredisce e non rimane bloccato

Mutua Esclusione, Deadlock e Starvation

- **Mutua esclusione**:
  - L'accesso ad una risorsa si dice mutualmente esclusivo se ad ogni istante solo un processo può accedere a quella risorsa
  - Es: due processi che vogliono accedere contemporaneamente alla stampante
- **Deadlock** (stallo):
  - La mutua esclusione permette di risolvere il problema della non interferenza
  - Ma può causare **blocco permanente** dei processi
  - Esempio:

R_1 e R_2 sono risorse e P_1 e P_2 sono processi che devono accedere contemporaneamente a R_1 e R_2 . Supponiamo che il SO assegni R_1 a P_1 e R_2 a P_2. I processi sono bloccati in attesa, cioè in deadlock.

- **Starvation**:
  - Quando processo non può accedere a una risorsa perché sempre occupata
  - A differenza del deadlock, non è una condizione definitiva, basta usare politiche di assegnamento

Azioni Atomiche

- Operazioni che vengono compiute in modo invisibile in un programma
- Una volta iniziata un'azione atomica, nessun'altra azione può interferire con essa o interromperla fino al suo completamento.
- L'azione atomica è vista dal sistema come un'operazione singola e indivisibile. Non può essere suddivisa in passaggi più piccoli, quindi o viene completata interamente o non viene eseguita affatto
- Per indicare azioni atomiche si usa la notazione **<**istruzione**>**

Sezioni Critiche

Se le sequenze di istruzioni non vengono eseguite in modo atomico, come possiamo garantire la non-interferenza?

Dobbiamo trovare il modo di specificare che certe parti dei programmi devono essere eseguite in modo atomico

- La parte di un programma che utilizza una o più risorse condivise

```
process P1                    process P2 {
  a1 = rand();                  a2 = rand();
  totale = totale + a1;         totale = totale + a2;
}                             }
```

- La parte rossa è una sezione critica, in quanto accede alla risorsa condivisa totale; mentre a1 e a2 non sono condivise
- Per evitare deadlock o starvation e far sì che le sezioni critiche vengano eseguite in modo atomico, scriviamo le istruzioni tra **[enter cs]** e **[exit cs]**

```
cobegin
  val = rand();
  a = a + val;
  b = b + val
//
  val = rand();
  a = a * val;
  b = b * val;
coend
```

Vorremmo garantire che a sia sempre uguale a b (invariante).
il SO non conosce l'invariante, l'unica soluzione possibile per il SO è non eseguire le due parti in parallelo

```
cobegin
  val = rand();
  [enter cs]
  a = a + val;
  b = b + val
  [exit cs]
//
  val = rand();
  [enter cs]
  a = a * val;
  b = b * val;
  [exit cs]
coend
```

Indichiamo al S.O. cosa può essere eseguito in parallelo.
Indichiamo al S.O. cosa deve essere eseguito in modo atomico, altrimenti non avremo consistenza

Requisiti per le sezioni critiche:

- **Mutua esclusione**
  - Solo un processo alla volta deve essere all'interno della CS
- **Assenza di deadlock**
  - Tutti i processi non possono restare bloccati
- **Assenza di delay non necessari**
  - Un processo fuori dalla CS non deve ritardare l'ingresso della CS da parte di un altro processo
- **Eventual entry (assenza di starvation)**
  - Ogni processo che lo richiede, prima o poi entra nella CS

Approcci Software

- Le soluzioni software permettono di risolvere il problema delle CS
- Problemi:
    - Tutte basate su busy waiting (controllo iterativo su una condizione di accesso)
    - Busy waiting spreca tempo del processore
    - Tecnica che non dovrebbe essere utilizzata

Algoritmo di Dekker

- Algoritmo di sincronizzazione utilizzato per garantire la mutua esclusione
- Difficile da estendere a più di due processi e meno efficiente di altri algoritmi moderni
- FORSE AGGIUNGERE LE SLIDE MANCANTI

Algoritmo di Peterson

- Più semplice e lineare di quello di Dekker
- Più facilmente generalizzabile al caso di processi multipli

Approccio Hardware

- Applicabili a N processi, sia su sistemi mono che in multiprocessori
- Utilizzato per supportare CS multiple
- Svantaggi:
- Si utilizza ancora busy waiting, complesse da programmare

Disabilitazione degli Interrupt

```
process P {
  while (true) {
    disable interrupt
    critical section
    enable interrupt
    non-critical section
  }
}
```

Nei sistemi uniprocessore, i processi concorrenti vengono "alternati" tramite il meccanismo degli interrupt

Allora facciamone a meno

- Il SO lascia ai processi la responsabilità di riattivare gli interrupt (molto pericoloso)
- Non funziona su sistemi multiprocessore

Test & Set

- Sono istruzioni speciali che realizzano due azioni in modo atomico
    - Es: lettura e scrittura, test e scrittura
- *TS (x, y) := < y = x ; x = 1 >*
    - Ritorna in *y* il valore precedente di *x*
    - Assegna 1 ad *x*

Semafori

- Tipo di dato astratto per il quale sono definite due operazioni:
  - V:
    - Invocata per inviare un segnale (verificarsi di un evento)
  - P:
    - Invocata per attendere il segnale (attendere un evento)

```
class Semaphore {
  private int val;
  Semaphore(int init) { }
  void P() {}
  void V() {}
}
```

Può essere visto come una variabile intera, inizializzata ad un valore positivo.
L'operazione P attende che il valore del semaforo sia positivo.
L'operazione V incrementa il valore del semaforo.
Le azioni P e V sono atomiche.

- Semafori FIFO:
  - Politica first-in, first-out
  - Il processo che è stato sospeso più a lungo viene svegliato per primo
  - Struttura dati: coda

```
void P() {
  value--;
  if (value < 0) {
    pid = <id del processo
           che ha invocato P>;
    queue.add(pid);
    suspend(pid);
   }
}
void V() {
  value++;
  if (value <= 0)
    pid = queue.remove();
    wakeup(pid);
}
```

Il process id del processo bloccato viene messo in un insieme **queue**

Con l'operazione **suspend**, il s.o mette il processo nello stato **waiting**

Il process id del processo da sbloccare viene selezionato (secondo una certa politica) dall'insieme **queue**

Con l'operazione **wakeup**, il S.O. mette il processo nello stato **ready**

- Necessario utilizzo di tecniche di CS viste prima (Dekker, Test&Set, Peterson)
- Utilizzandole non eliminiamo il busy waiting, però lo limitiamo alle CS di P e V
- Difetti dei semafori:
  - Sono costrutti di basso livello
  - Il programmatore non deve commettere errori tipo omettere P o V
  - Il programmatore deve accedere a dati condivisi in modo corretto

Semafori Binari

- Variante dei semafori dove il valore può essere solo 0 o 1
- Servono a garantire mutua esclusione, semplificando il lavoro al programmatore

Problemi Classici nella Programmazione Concorrente

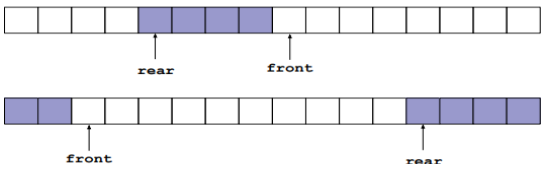
- Produttore/consumatore
- Buffer limitato
- Filosofi a cena
- Lettori e scrittori

Produttore/Consumatore

- Esiste un processo ***Producer*** che genera valori e vuole trasferirli a un processo ***Consumer*** che prende i valori generati e li consuma
- Comunicazione avviene attraverso una variabile condivisa
- Proprietà da garantire:
    - ***Producer*** non deve sovrascrivere l'area di memoria condivisa prima che ***Consumer*** abbia utilizzato il valore precedente
    - ***Consumer*** non deve leggere due volte lo stesso valore, ma deve attendere che ***Producer*** generi il successivo
    - Assenza di deadlock

Buffer Limitato

- Simile a Produttore/Consumatore
- Scambio tra Producer e Consumer avviene tramite un buffer di dim. limitata (vettore)
- Proprietà da garantire:
    - ***Producer*** non deve sovrascrivere elementi del buffer prima che ***Consumer*** abbia utilizzato i suoi valori
    - ***Consumer*** non deve leggere due volte lo stesso valore, ma deve attendere che ***Producer*** generi il successivo
    - Assenza di deadlock e starvation
- Struttura dei buffer:
    - Si usano due indici front e rear che indicano il prossimo elemento da scrivere e il prossimo da leggere



Cena dei Filosofi

5 filosofi mangiano e pensano alternativamente. Per mangiare usano un tavolo rotondo con 5 sedie, 5 piatti, 5 bacchette. Per mangiare un filosofo ha bisogno delle bacchette, poi quando ha finito le posa.

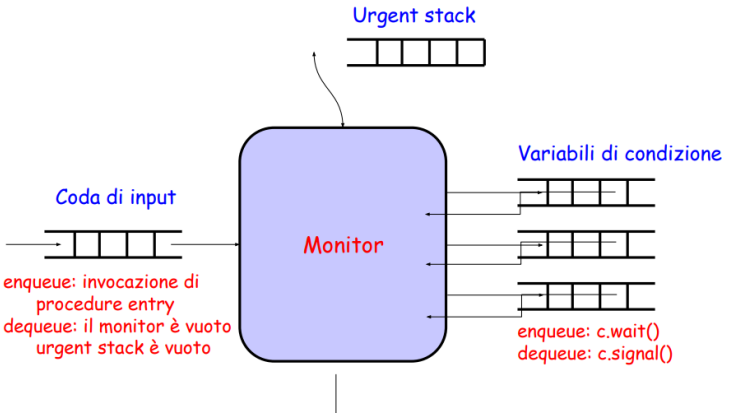
- Mostra come gestire situazioni in cui i processi entrano in competizione per accedere ad insiemi di risorse a intersezione non nulla
- Invariante:
  - up_i numero di volte che bacchetta i viene presa dal tavolo
  - $down_i$ numero di volte che bacchetta i viene rilasciata sul tavolo
  - $down_i \leq up_i \leq down_i + 1$

Lettori e Scrittori

- Un database è condiviso tra dei processi che possono essere di 2 tipi:
  - Lettori: accedono al database per leggerne il contenuto
  - Scrittori: accedono al database per aggiornarne il contenuto
- Proprietà:
  - Se uno scrittore accede a un database, opera in mutua esclusione (nessun altro può accedervi)
  - Se nessuno scrittore sta accedendo al database, un numero arbitrario di lettori può accedervi
- La competizione per le risorse avviene anche a livello di classi di processi
- Mostra che mutua esclusione e condivisione possono coesistere
- Invariante:
  - nr numero dei lettori che stanno accedendo al database
  - nw numero di scrittori che stanno accedendo al database
  - $(nr > 0 \;\&\&\; nw == 0) \;||\; (nr == 0 \;\&\&\; nw \leq 1)$

Monitor

- Modulo software che consiste di:
  - Dati locali
  - Una sequenza di inizializzazione
  - Una o più “procedure”
- Caratteristiche principali:
  - Dati locali accessibili solo alle procedure del modulo stesso
  - Un processo entra in un monitor invocando una delle sue procedure
  - Solo un processo alla volta può essere all’interno del monitor
- Assomiglia a un oggetto in oop
- Dichiarazione di variabili di condizione (CV):
  - $condition\ c;$
- Operazioni definite sulle CV:
  - $c.wait()$ attende il verificarsi della condizione
    - Viene rilasciata la mutua esclusione
    - Processo che chiama $c.wait()$ viene sospeso in una coda di attesa
  - $c.signal()$ segnala che la condizione è vera
    - Causa riattivazione di un processo (secondo politica FIFO)
    - Chiamante viene posto in attesa
    - Verrà riattivato quando il processo risvegliato avrà rilasciato la mutua esclusione (urgent stack)



- Differenze tra wait/signal e P/V dei semafori:
  - $signal$ non ha effetti se nessun processo sta attendendo la condizione
    V “memorizza” il verificarsi degli eventi
  - $wait$ è sempre “bloccante”
    P (se il semaforo ha valore positivo) invece no
- Politiche di signaling:
  - SU - signal urgent
  - SW - signal wait
    - No urgent stack, signaling process viene messo nella entry queue
  - SR - signal and return
    - Dopo la $signal$ si esce subito dal monitor
  - SC - signal and continue
    - La $signal$ segnala solo che un processo può continuare
    - Il chiamante prosegue l’esecuzione

Message Passing

Semafori e monitor sono paradigmi di *sincronizzazione* tra processi, in cui la comunicazione avviene tramite memoria condivisa

- Message passing è un paradigma di *comunicazione* tra processi
- La sincronizzazione avviene tramite scambio di messaggi, e non più semplici segnali
- Messaggio:
    - Un insieme di informazioni formattate da un processo mittente e interpretate da un processo destinatario
- Meccanismo di “scambio di messaggi”:
    - Copia le informazioni di un messaggio da uno spazio di indirizzamento di un processo allo spazio di indirizzamento di un altro processo

Operazioni

- **Send**:
    - Usata dal processo mittente per spedire un mess. al processo destinatario
    - Il processo destinatario deve essere specificato
- **Receive**:
    - Usata dal processo destinatario per ricevere un mess. dal processo mittente
    - Il processo mittente può essere specificato o qualsiasi

MP Sincrono

- **Send sincrono**:
    - Sintassi: $ssend(m,\ q)$
    - Mittente p spedisce il messaggio m al processo q, restando bloccato fino a quando q non esegue $sreceive(m,\ p)$
- **Receive bloccante**:
    - Sintassi: $m = sreceive(p)$
    - Il destinatario q riceve il messaggio m dal processo p
    - Se il mittente non ha spedito messaggi, il destinatario ne attende uno

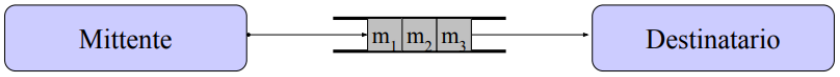
MP Asincrono

- **Send asincrona**:
    - Sintassi: $asend(m,\ q)$
    - Mittente p spedisce il messaggio m al processo q, senza bloccarsi se q non esegue $areceive(m,\ p)$
- **Receive bloccante**:
    - Sintassi: $m = areceive(p)$
    - Il destinatario q riceve il messaggio m dal processo p
    - Se il mittente non ha spedito messaggi, il destinatario ne attende uno

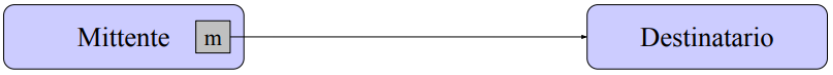
MP Totalmente Asincrono

- **Send asincrona**:
  - Sintassi: $asend(m,\ q)$
  - Mittente p spedisce il messaggio m al processo q, senza bloccarsi se q non esegue $nb - receive(m,\ p)$
- **Receive non bloccante:**
  - Sintassi: $m = nb - receive(p)$
  - Il destinatario q riceve il messaggio m dal processo p
  - Se il mittente non ha spedito messaggi, $nb - receive$ termina ritornando null

Message passing asincrono



Message passing sincrono



GESTIONE DI RISORSE

- Le risorse possono essere suddivise in **classi**:
  - Le risorse della stessa classe sono equivalenti (stampanti dello stesso tipo)
- Le risorse di una classe vengono dette **istanze** della classe
- Il numero di risorse in una classe viene detto **molteplicità** del tipo di risorsa
- **Risorse ad assegnazione statica:**
  - Avviene alla creazione del processo e rimane valida fino alla terminazione
  - Es: descrittori di processi, aree di memoria
- **Risorse ad assegnazione dinamica:**
  - I processi:
    - richiedono le risorse durante la loro esistenza
    - le utilizzano una volta ottenute
    - le rilasciano quando non più necessarie
  - Es: periferiche di I/O

Tipi di Richieste

- **Richiesta singola:**
  - Si riferisce a una singola risorsa di una classe
- **RIchiesta multipla:**
  - Si riferisce a una o più classi, e per ogni classe, a una o più risorse
- **Richiesta bloccante:**
  - Il processo richiedente si sospende se non ottiene l'assegnazione
  - la richiesta rimane pendente
- **Richiesta non bloccante:**
  - la mancata assegnazione viene notificata al processo richiedente, senza provocare la sospensione

Tipi di Risorse

- **Risorse seriali:**
  - Una risorsa non può essere assegnata a più processi contemporaneamente
  - Stampanti, processori
- **Risorse non seriali:**
  - File di sola lettura
- **Risorse prerilasciabili (preemptable):**
  - Se la funzione di gestione può sottrarla ad un processo
  - Meccanismo di gestione:
    - il processo che subisce il prerilascio deve sospendersi
    - la risorsa prerilasciata sarà successivamente restituita al processo
- **Risorse non prerilasciabili:**
  - la funzione di gestione non può sottrarle al processo al quale sono assegnate
  - Es: stampanti

Deadlock

- I deadlock impediscono ai processi di terminare correttamente

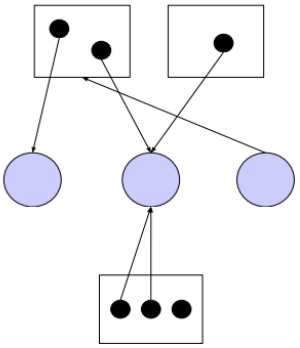
Condizioni per avere un deadlock:

- **Mutua esclusione**: le risorse coinvolte devono essere seriali
- **Assenza di prerilascio**: le risorse non possono essere prerilasciate
- **Richieste bloccanti**: le richieste devono essere bloccanti
- **Attesa circolare**

Devono valere tutte contemporaneamente

Grafo di Holt

- Composto da risorse(cerchietti neri) dentro classi(rettangolo) e processi(tondi blu)
- Gli archi risorsa -> processo indicano che la risorsa è assegnata al processo
- Gli archi processo -> risorsa indicano che il processo ha richiesto la risorsa



Metodi di Gestione dei Deadlock

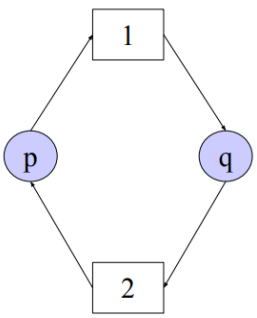
- **Deadlock detection and recovery**
  - Permettere al sistema di entrare in stati di deadlock
  - Usare algoritmo per rilevare questo stato ed eseguire azione di recovery
- **Deadlock prevention / avoidance**: impedire al sistema di entrare in deadlock
- **Ostrich algorithm**: ignorare il problema

Deadlock Detection and Recovery

- Utilizzare il grafo di Holt per riconoscere gli stati di deadlock

Teorema:

- Lo stato è di deadlock **se e solo se** il grafo di Holt contiene un ciclo

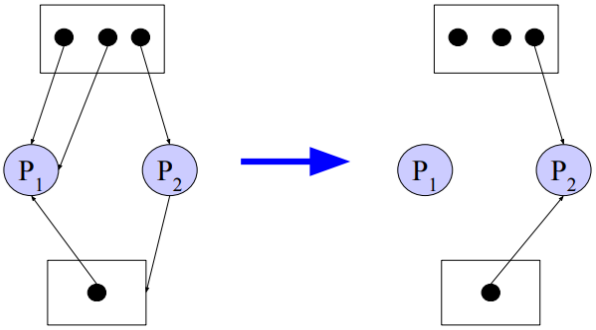


Grafo Wait-For

Se questo grafo ha ciclo, si ha attesa circolare

Riducibilità di un grafo di Holt:

- Si dice riducibile se esiste almeno un nodo processo con solo archi entranti
- La riduzione consiste nell'eliminare tutti gli archi di tale nodo e assegnare le risorse ad altri processi



Riduzione per P_1

Teorema:

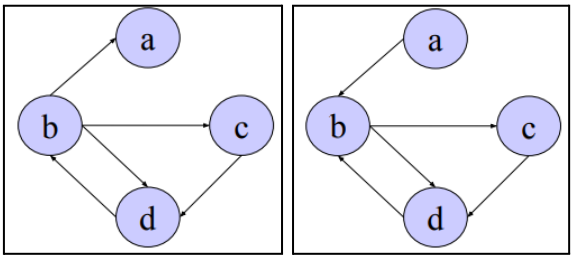
Se le risorse sono ad accesso mutualmente esclusivo, seriali e non prerilasciabili:

- Lo stato non è di deadlock se e solo se il grafo di Holt è completamente riducibile
- Ossia se esiste una sequenza di passi di riduzione che elimina tutti gli archi del grafo
- Esempio a slide 34 della parte 4-risorse

Knot:

dato un nodo n, l'insieme dei nodi raggiungibili da n è **l'insieme di raggiungibilità** di n **R(n)**

- Un Knot del grafo G è il massimo sottoinsieme di nodi M $tc \forall\, n\, in\, M,\ R(n) = M$
- Partendo da un qualunque nodo di M, si possono raggiungere tutti i nodi di M e nessun nodo all'infuori di esso



{ b, c, d } non è un Knot { b, c, d } è un Knot

Teorema:

Dato un grafo di Holt con una sola richiesta sospesa per processo.
Se le risorse sono ad accesso mutualmente esclusivo, seriali e non prerilasciabili.

- Il grafo rappresenta uno stato di deadlock se e solo se esiste un Knot

Recovery:

- Per risolvere la situazione, la soluzione può essere:
  - **Manuale**:
    - Viene informato l'operatore che risolverà la situazione
  - **Automatica**:
    - Il SO è dotato di meccanismi che risolvono la situazione

- Meccanismi per il Deadlock Recovery:
    - **Terminazione Totale:**
        - Tutti i processi coinvolti vengono terminati
    - **Terminazione Parziale:**
        - Si elimina un processo per volta, fino a che non scompare il deadlock
    - **Preemption**:
        - Una risorsa o più vengono sottratte ad uno dei processi
    - **Checkpoint / Rollback:**
        - Stato dei processi viene periodicamente salvato su disco (*checkpoint*)
        - In caso di deadlock, si ripristina (*rollback*) uno o più processi ad uno stato precedente, fino a quando il deadlock non scompare

Deadlock Prevention

- **Prevention**:
    - Per evitare deadlock si elimina una delle 4 condizioni del deadlock
    - Il deadlock viene eliminato **strutturalmente**
- Attaccare la condizione di "Mutua esclusione":
    - Permettere condivisione di risorse (spool di stampa)
    - Lo spooling non è sempre applicabile, si sposta problema su altre risorse
- Attaccare la condizione di “Richiesta Bloccante”:
    - Richiedere che un processo richieda tutte le risorse all’inizio della computazio
    - Si riduce il parallelismo
- Attaccare la condizione di “Assenza di prerilascio”:
    - Non sempre possibile, può richiedere interventi manuali
- Attaccare la condizione di “Attesa Circolare”:
    - Allocazione gerarchica

Deadlock Avoidance

- **Avoidance**:
    - Prima di assegnare una risorsa ad un processo si controlla se l’operazione può causare deadlock
    - In questo caso, l’operazione viene ritardata

GESTIONE DELLA MEMORIA

Memory Manager:

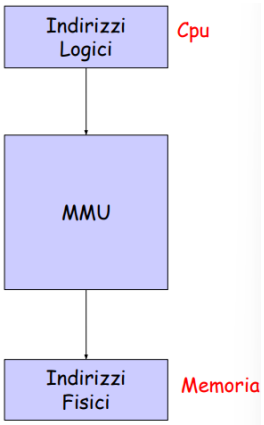
- Parte del SO che gestisce la memoria principale
- Si occupa di:
    - Tenere traccia della memoria libera e occupata
    - Allocare e deallocare memoria ai processi

Binding

- Associazione di indirizzi di memoria ai dati e alle istruzioni del programma

Può avvenire:

- Durante la compilazione:
    - Indirizzi calcolati al momento della compilazione e resteranno gli stessi ad ogni esecuzione del programma
    - Codice generato viene detto **codice assoluto**
    - Vantaggi:
        - Non richiede hardware speciale; semplice; molto veloce
    - Svantaggi:
        - Non funziona con la multiprogrammazione
- Durante il caricamento:
    - Codice generato dal compilatore non contiene indirizzi assoluti ma relativi
    - Codice generato viene detto **codice rilocabile**
    - Il loader si occupa di aggiornare i riferimenti agli indirizzi di memoria coerentemente a punto iniziale di caricamento
    - Vantaggi:
        - Permette di gestire multiprogrammazione
    - Svantaggi:
        - Richiede traduzione degli indirizzi da parte del loader
- Durante l'esecuzione:
    - Individuazione dell'indirizzo di memoria effettivo viene effettuata durante l'esecuzione da un componente hardware apposito:
      La **Memory Management Unit (MMU)**



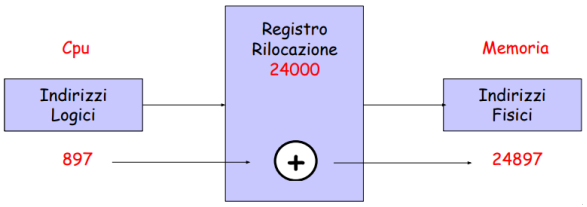
Ogni processo è associato a un **indirizzo logico**.

Ad ogni indirizzo logico corrisponde un **indirizzo fisico**.

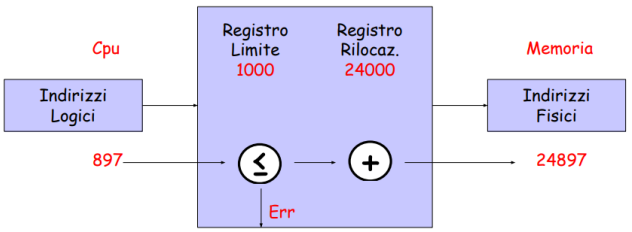
MMU opera come una funzione di traduzione da indirizzi logici a fisici.

Esempi di MMU:

- **Registro di Rilocazione:**
  - Se valore del registro di rilocazione è R, uno spazio logico 0…MAX viene tradotto in uno spazio fisico R…R+MAX



- **Registro di Rilocazione e Registro Limite:**
  - Il registro limite viene usato per implementare meccanismi di protezione della memoria



Loading e Linking

- **Loading Dinamico:**
  - Carica alcune routine di libreria solo quando vengono richiamate
  - Le routine risiedono su un disco, quando servono vengono caricate
- **Linking Statico:**
  - Se linker collega e risolve tutti i riferimenti dei programmi
- **Linking Dinamico:**
  - Si può posticipare il linking delle routine di libreria al primo riferimento durante l'esecuzione
  - Vantaggi:
    - Risparmio di memoria; aggiornament automatico versioni delle librerie
  - Svantaggi:
    - Può causare problemi di "versioning"

Allocazione

- Una delle principali funzioni del gestore di memoria
- Consiste nel reperire ed assegnare uno spazio di memoria fisica
- **Allocazione Contigua:**
  - Spazio assegnato a un programma deve essere formato da celle consecutive
- **Allocazione non Contigua:**
  - Si possono assegnare a un programma aree di memoria separate

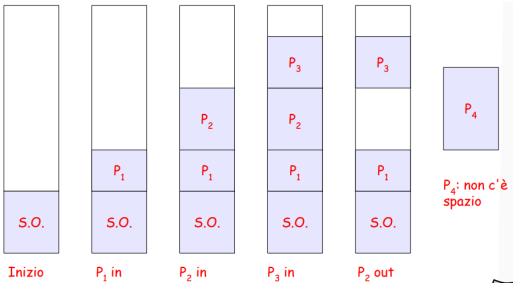
- **Allocazione Statica:**
    - Programma deve mantenere la propria area di memoria dal caricamento alla terminazione
- **Allocazione Dinamica:**
    - Durante l'esecuzione, un programma può essere spostato

Allocazione a Partizioni Fisse

- Memoria disponibile viene suddivisa in partizioni
- Ogni processo viene caricato in una delle partizioni che ha sufficiente spazio
- Vantaggi: molto semplice
- Svantaggi: spreco di memoria, parallelismo limitato al numero di partizioni
- Per gestire la memoria si può utilizzare una coda per partizione
- **Frammentazione interna:**
    - Lo spazio inutilizzato dentro una partizione è sprecato

Allocazione a Partizioni Dinamiche

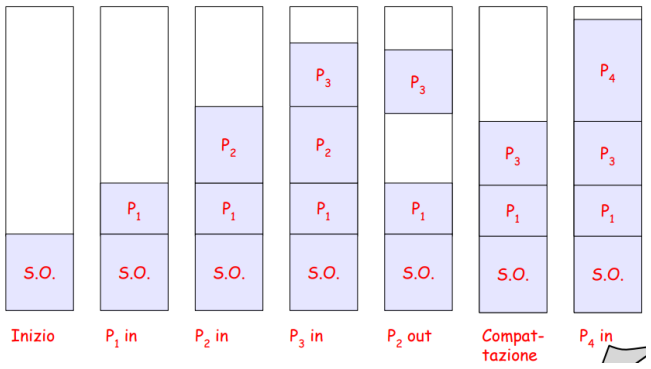
- Memoria richiesta viene assegnata ai processi che la richiedono
- Nella memoria possono esserci diverse zone inutilizzate



- **Frammentazione esterna:**
    - Dopo diverse allocazioni e deallocazioni di memoria lo spazio libero sembra suddiviso in piccole aree
    - Quindi deriva dal susseguirsi di allocazioni e deallocazioni
    - La framm. interna invece dipende dall'uso di unità di allocazione di dimensione diversa da quella richiesta

Compattazione

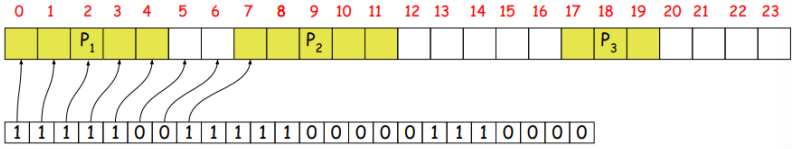
- Spostare in memoria tutti i programmi in modo da riunire le aree inutilizzate
- Svolta per risolvere la frammentazione esterna
- Operazione molto onerosa



Quando la memoria è assegnata dinamicamente abbiamo bisogno di una struttura dati per mantenere informazioni sulle zone libere e sulle zone occupate

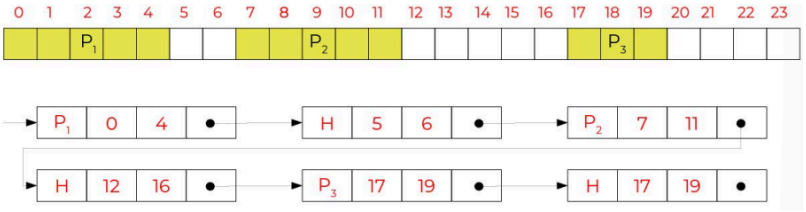
Mappa di Bit

- Memoria suddivisa in unità di allocazione
- Ad ogni unità di allocazione corrisponde un bit in una bitmap
- Unità libere sono associate a un bit di valore 0, quelle occupate a un bit di valore 1
- Vantaggi:
    - La struttura dati ha dimensione fissa e calcolabile a priori
- Svantaggi:
    - Per individuare uno spazio di memoria di dimensione **k**, è necessario cercare una sequenza di **k** bit 0 consecutivi
    - Costo: **O(m)** dove **m** rappresenta il numero di unità di allocazione



Lista con Puntatori

- Si mantiene una lista dei blocchi allocati e liberi di memoria
- Ogni elemento della lista specifica:
    - Se si tratta di un processo (P) o di un blocco libero (H)
    - La dimensione del segmento (inizio/fine)



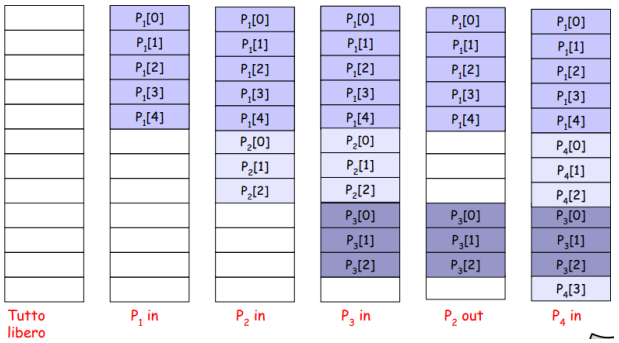
- Allocazione di memoria:
    - Blocco libero viene selezionato
    - Viene suddiviso in due parti:
        - Un blocco processo della dimensione desiderata
        - Un blocco libero con quanto rimane del blocco iniziale
- Deallocazione di memoria:
    - A seconda dei blocchi vicini, lo spazio liberato può creare un nuovo blocco libero, oppure essere accorpato ai blocchi vicini in tempo O(1)

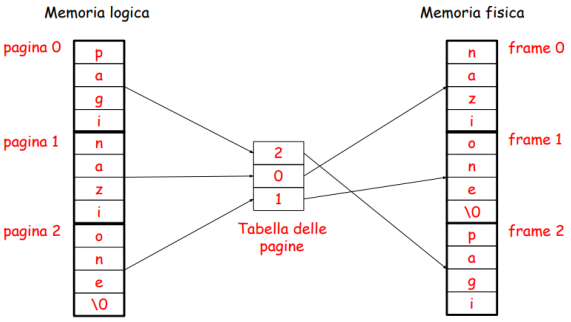
Selezione Blocco Libero

- **First Fit:**
    - Scorre la lista dei blocchi liberi fino a quando non trova il primo segmento vuoto grande abbastanza da contenere il processo
- **Next Fit:**
    - Come First Fit, ma parte dal punto dove si era fermato all'ultima allocazione
- **Best Fit:**
    - Seleziona il più piccolo fra i blocchi liberi presenti in memoria
- **Worst Fit:**
    - Seleziona il più grande fra i blocchi liberi presenti in memoria

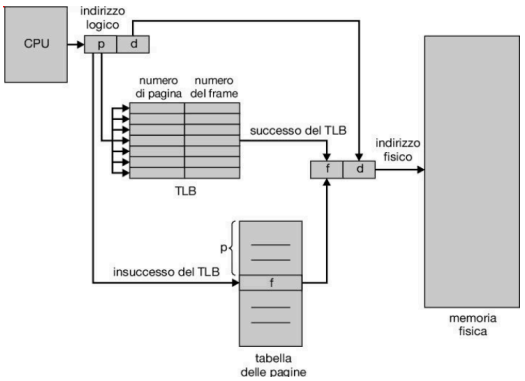
Paginazione

- Riduce la frammentazione interna ed elimina la frammentazione esterna
- Lo spazio di indirizzamento logico di un processo viene suddiviso in un insieme di blocchi di dimensione fissa chiamati **pagine**
- La memoria fisica viene divisa in **frame**



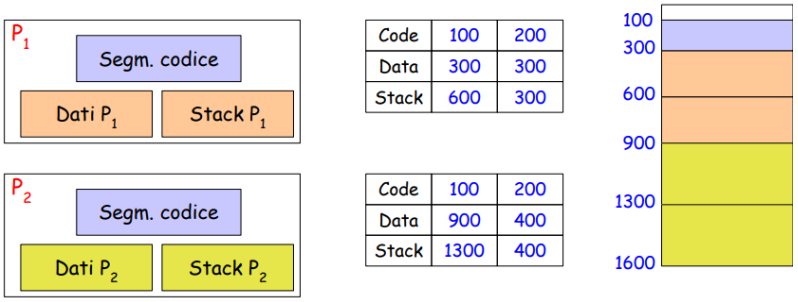


- Con pagine troppo piccole la tabella delle pagine cresce di dimensione
- Con pagine troppo grandi, la frammentazione interna può essere considerevole
- Implementazione della page table:
  - Registri dedicati: troppo costoso
  - Totalmente in memoria: numero di accessi in memoria verrebbe raddoppiato
- **Translation Lookaside Buffer (TLB):**
  - Costituito da un insieme di registri ad alta velocità
  - Ogni registro è suddiviso in due parti, una chiave e un valore
  - Operazione di **lookup**:
    - Viene richiesta la ricerca di una chiave
    - Se è presente (**TLB hit**) si ritorna il valore corrispondente
    - Se non è presente (**TLB miss**) si utilizza la page table in memoria



Segmentazione

- Memoria associata a un programma è suddivisa in aree funzionali differenti
- Aree text: contengono codice eseguibile; Aree dati, Area stack …
- Uno spazio di indirizzamento logico è dato da un insieme di segmenti
- Un segmento è un'area di memoria contenente elementi tra loro affini
- Ogni segmento ha un nome e una lunghezza
- Spetta al programmatore la divisione di un programma in segmenti



PAGINAZIONE	SEGMENTAZIONE
Divisione in pagine è automatica	Divisione in segmenti dal programmatore
Pagine hanno dimensione fissa	Segmenti hanno dimensione variabile
Pagine possono contenere informazioni disomogenee (sia dati sia codice)	Segmento contiene informazioni omogenee
Una pagina ha un indirizzo	Un segmento ha un nome
Dimensione tipica di una pagina: 1 - 4 KB	Dim. tipica di un segmento: 64 KB - 1 MB

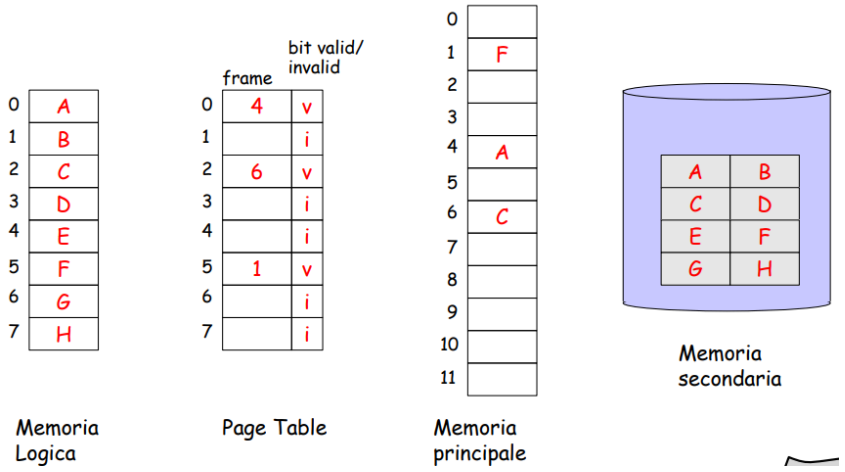
- E' possibile utilizzare la paginazione combinata alla segmentazione
- Ogni segmento viene suddiviso in pagine che vengono allocate in frame liberi
- La MMU deve avere il supporto HW sia per la segmentazione che per la paginazione
- Benefici:
    - Sia quelli della segmentazione (condivisione, protezione)
    - Sia quelli della paginazione (no frammentazione esterna)

Memoria Virtuale

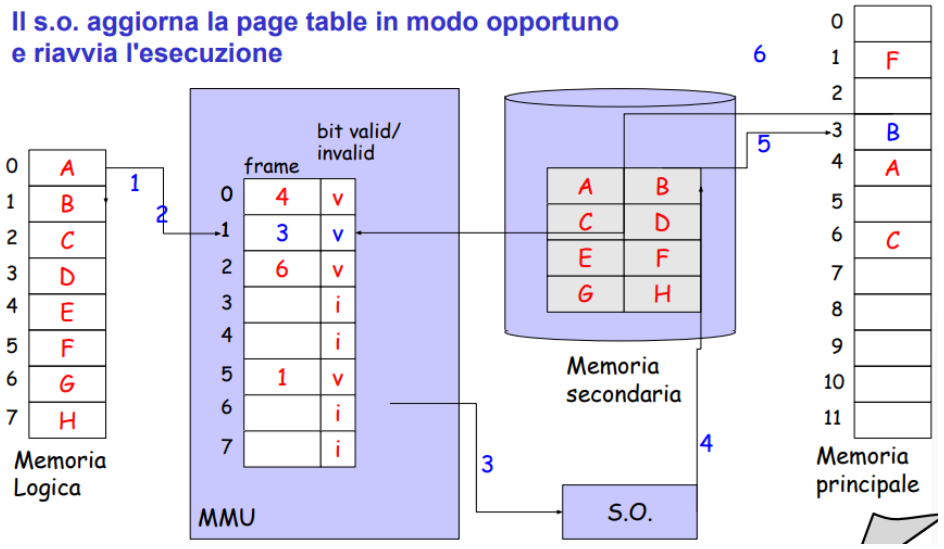
- Tecnica che permette l'esecuzione di processi che non sono completamente in memoria
- Permette di eseguire in concorrenza processi che hanno necessità di memoria maggiore di quella disponibile

Implementazione

- Ogni processo ha accesso ad uno spazio di indirizzamento virtuale
- Gli indirizzi virtuali:
    - possono essere mappati su indirizzi fisici della memoria principale o su memoria secondaria
- In caso di accesso ad indirizzi virtuali mappati in memoria secondaria:
    - I dati associati vengono trasferiti in memoria principale
    - Se la memoria è piena, si sposta in memoria secondaria i dati contenuti in memoria principale che sono considerati meno utili
- Si usa la Paginazione a Richiesta (**demand paging**):
    - Nella page table si usa un bit (**v**) che indica se la pagine è presente in memoria centrale oppure no
    - Quando un processo tenta di accedere a una pagine non in memoria:
        - Il processore genera un trap (**page fault**)
        - Un componente del SO (**pager**) carica la pagina mancante in memoria e aggiorna la page table



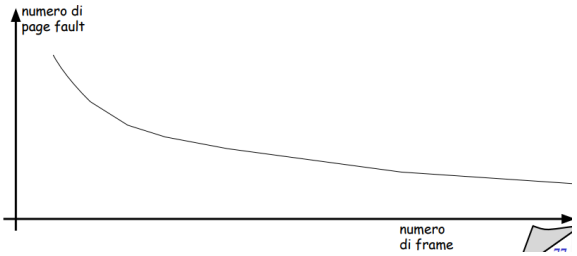
Gestione dei Page Fault



1. Supponiamo che il codice in pagina 0 faccia riferimento alla pagina 1
2. La MMU scopre che la pagina 1 non è in memoria principale
3. Viene generato un trap "page fault", che viene catturato dal SO
4. Il SO cerca in memoria secondaria la pagina da caricare
5. Il SO carica la memoria principale con il contenuto della pagina
6. Il SO aggiorna la page table in modo opportuno e riavvia l'esecuzione

Algoritmi di Sostituzione

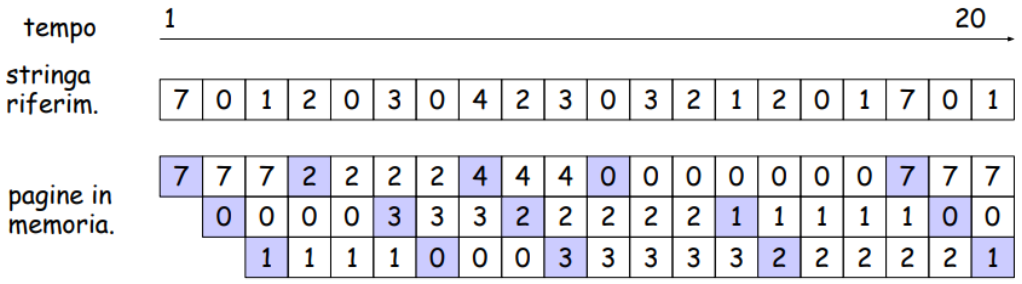
In mancanza di frame liberi occorre liberarne uno, la pagina da sostituire deve essere la meno utile.
L'obiettivo è minimizzare il numero di page fault, gli algoritmi vengono valutati a seconda di come si comportano quando applicati a una **stringa di riferimenti** in memoria.



- frame ho
+ page fault ci saranno
Non valido sempre

es. di stringa di riferimenti in esadecimale:
71,0a,13,25,0a,3f,0c,4f,21,30,00,31,21,1a,2b,03,1a,77,11

- **Algoritmo FIFO:**
  - Il primo frame caricato in memoria sarà quello da sostituire
  - Vantaggi: semplice
  - Svantaggi: vengono scaricate pagine utilizzate



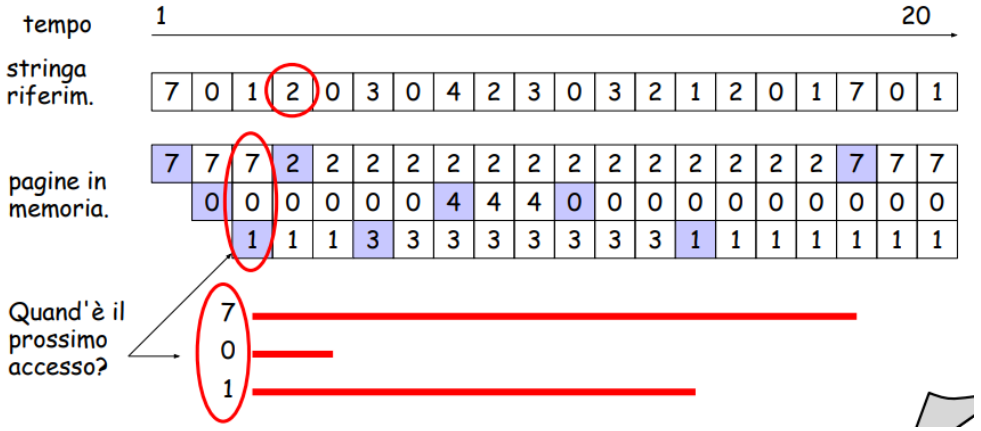
- Numero di frame in memoria: 3
- Numero di page fault ottenuti: 15 (su 20 accessi in memoria)

All'inizio non ho nessuna pagina in memoria. Mi serve il 7, lo carico. Mi serve il 0, lo carico. Mi serve l'1, lo carico. I frame sono esauriti. Mi serve il 2, dato che è FIFO, tolgo il 7 e ci metto il 2. Mi serve lo 0, niente page fault perchè è già in memoria. Mi serve il 3, dato che è FIFO, tolgo lo 0 e metto il 3. Il numero di caselle viola è il numero di page fault.

Anomalia di Belady:
In alcuni algoritmi di rimpiazzamento non è detto che aumentando il numero di frame, il numero di page fault diminuisca (es. FIFO)

- **Algoritmo MIN** (ottimale):
  - Sostituisce la pagina che non sarà più acceduta o quella che verrà acceduta nel futuro più lontano
  - Algoritmo teorico, usato a posteriori come paragone per algoritmi reali

Esempio di prima:

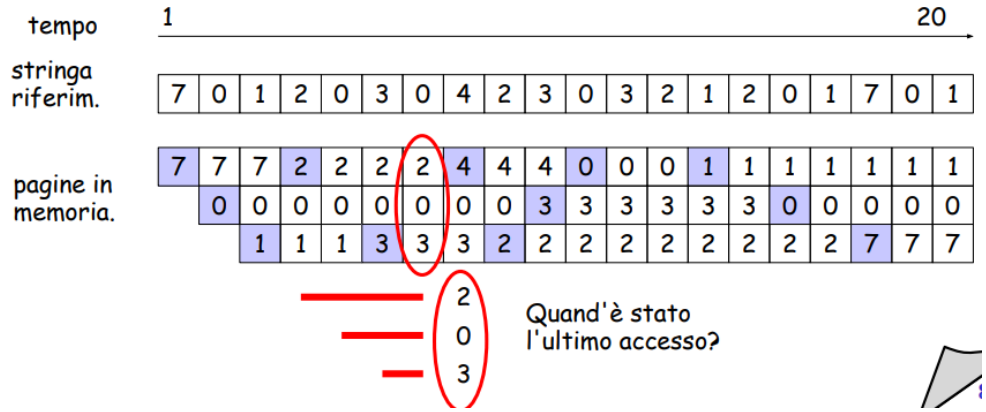


- Numero di frame in memoria: 3
- Numero di page fault: 9 (su 20 accessi in memoria)

All'inizio non ho nessuna pagina in memoria. Mi serve il 7, lo carico. Mi serve il 0, lo carico. Mi serve l'1, lo carico. I frame sono esauriti. Mi serve il 2, Vedo quale frame mi servirà fra più tempo, tolgo il 7 e metto il 2, eccetera

Algoritmo LRU (Least Recently Used)

- Sostituisce la pagina che è stata usata meno recentemente nel passato
- Stesso esempio:



- Numero di page fault: 12 (su 20 accessi in memoria)

All'inizio non ho nessuna pagina in memoria. Mi serve il 7, lo carico. Mi serve il 0, lo carico. Mi serve l'1, lo carico. I frame sono esauriti. Poi arrivo al cerchio. Mi chiedo quale frame ho usato più nel passato e sostituisco quello. eccetera…

- E' necessario uno specifico supporto hardware
- La MMU:
  - Deve registrare nella page table un time-stamp quando accede a una pagina
  - Il time-stamp può essere un contatore che si incrementa ad ogni accesso
  - Complicato con il contatore, si può usare stack
- Implementazione con stack:
  - Si mantiene uno stack di pagine
  - Quando una pagina viene acceduta, viene rimossa dallo stack se presente e posta in cima
  - In questo modo in cima si trova la pagina usata più di recente mentre in fondo quella usata meno di recente
- **Algoritmi a stack:**
  - Si indichi con $\mathbf{S_t(A, m)}$ l'insieme delle pagine mantenute in memoria centrale al tempo **t** dell'algoritmo **A**, data una memoria di **m** frame
  - Un algoritmo di rimpiazzamento è "a stack" se per ogni istante t si ha:
    $\mathbf{S_t(A, m) \subseteq S_t(A, m+1)}$
    Ossia se l'insieme delle pagine in memoria con **m** frame è sempre un sottoinsieme delle pagine in memoria con **m+1** frame

Teorema:

- Un algoritmo a stack non genera casi di Anomalia di Belady

Teorema:

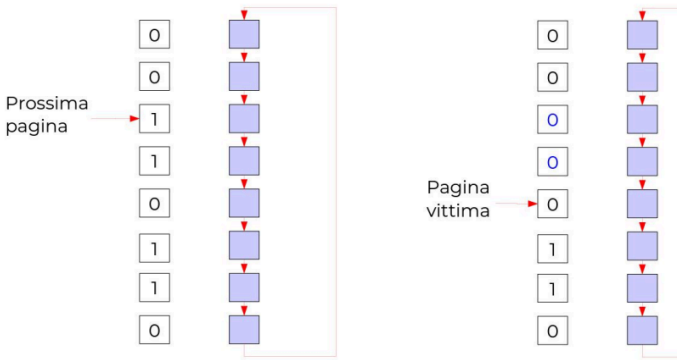
- L'algoritmo LRU è a stack

- Sia coi contatori che con stack, mantenere le informazioni per LRU è troppo costoso

Approssimare LRU

- **Reference bit:**
  - Quando si accede a una pagina, il reference bit associato alla pagina viene aggiornato a 1
  - Inizialmente i bit sono a 0
  - Durante l'esecuzione dei processi, le pagine in memoria vengono accedute e i reference bit vengono posti a 1
  - E' possibile osservare quali pagine sono state accedute osservando i reference bit
- **Additional-Reference-Bit-Algorithm:**
  - Si mantengono x bit di "storia" per ogni pagina
  - A intervalli regolari si "salvano" i reference bit
  - Il nuovo valore del reference bit viene salvato tramite shift a destra della storia ed inserimento del bit come most signif. bit
  - La pagina che si andrà a sostituire sarà quella con valore minore

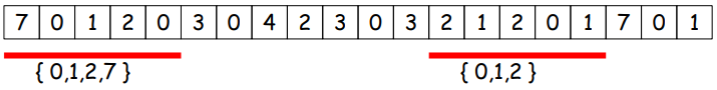
- **Second-Chance Algorithm:**
  - Caso particolare del precedente, la dimensione della storia è 1
  - Le pagine in memoria gestite come una lista circolare
  - Partendo dalla posizione successiva all'ultima pagina caricata, si scandisce la lista con la seguente regola:
    - Se la pagina è stata acceduta (reference bit a 1), il ref. bit diventa 0
    - Se la pagina non è stata acceduta (reference bit a 0), si sostituisce



Allocazione

- **Algoritmo di Allocazione:**
  - Algoritmo utilizzato per scegliere quanti frame assegnare a ogni processo
- **Allocazione locale:**
  - Ogni processo ha un insieme proprio di frame, poco flessibile
- **Allocazione globale:**
  - Tutti i processi possono allocare tutti i frame presenti nel sistema
  - Può portare a trashing
- **Trashing**:
  - Un processo è in trashing quando spende più tempo per la paginazione che per l'esecuzione
- **Working Set:**
  - Il **working set di finestra** Δ è l'insieme delle pagine accedute nei più frequenti Δ riferimenti
  - se una pagina non compare in Δ riferimenti successivi in memoria, allora esce dal working set; non è più una pagina su cui si lavora attivamente
  - Se Δ è calcolato bene, il working set è una buona approssimazione dell'insieme delle pagine utili

Esempio: Δ = 5



FILE SYSTEM

- Si occupa di astrarre la complessità d'uso dei media proponendo un'interfaccia comune, efficiente e conveniente da usare per i sistemi di memorizzazione
- Dal pov dell'utente, è composto da:
    - **file**: unità logica di memorizzazione
    - **directory**: per organizzare file del file system

File

- Entità atomica di assegnazione/gestione della memoria secondaria

Attributi dei File

- Nome
- Tipo
- Locazione e dimensione
- Data e ora
- Informazioni sulla proprietà
- Attributi di protezione

Tipi di File

- A seconda della struttura interna:
    - Senza formato: file testo
    - Con formato: file di database
- A seconda del contenuto:
    - ASCII
    - Sorgente, oggetto, …
    - Eseguibile
- 3 tecniche per identificare il tipo di un file:
    - Estensioni
    - Uso di un attributo "tipo" associato al file
    - Magic number
- Nel file system di un sistema UNIX sono presenti:
    - file regolari
    - directory
    - file speciali a blocchi: modellano dispositivi di I/O come dischi
    - file speciali a caratteri: modellano device di I/O come stampanti

Struttura dei File

- Diverse strutture:
    - Sequenze di byte
    - Sequenze di record logici
    - File indicizzati (struttura ad albero)
- Se un sistema ha più formati:
    - Codice di sistema più ingombrante
    - Gestione efficiente e non duplicata per formati speciali
- Se un sistema ha meno formati:
    - Codice di sistema più snello

Metodi di accesso:

- Sequenziale: read, write
- Ad accesso diretto: read *pos*, write *pos*
- Indicizzato: read *key*, write *key*

Operazioni sui file:

- creazione, apertura/chiusura, lettura/scrittura/append, cancellazione…

Directory

- File speciali che consentono di organizzare il file system

Operazioni sulle directory:

- creazione/cancellazione, apertura/chiusura, lettura, rinominazione…

Struttura di una Directory

- A livello singolo, a due livelli
- Ad albero
    - Ogni file contenuto in una directory univoca
- A grafo aciclico, a grafo
    - Un file può essere contenuto in più directory
    - **DAG (Grafo Diretto Aciclico)**

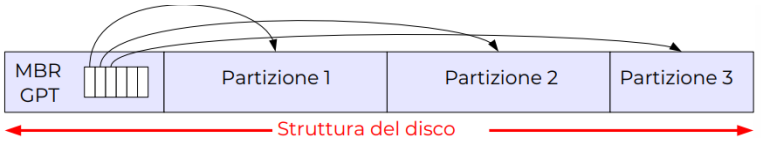
Visione implementatore:

- Problemi da tenere in considerazione:
  - Organizzazione di un disco
  - Allocazione dello spazio in blocchi
  - Gestione spazio libero
  - Implementazione delle directory
  - Tecniche per ottimizzare prestazioni
  - Tecniche per garantire la coerenza

Organizzazione del Disco

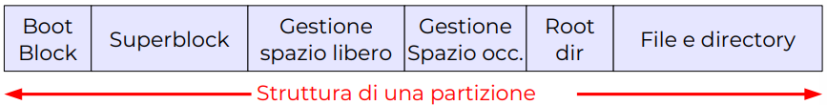
Struttura di un disco:

- Può essere diviso in partizioni, porzioni indipendenti del disco che possono ospitare file system distinti
- 1° settore dei dischi è il **Master Boot Record (MBR)**
  - Utilizzato per fare il boot del sistema
  - Contiene la **Partition Table (tabella delle partizioni)**
- Al boot, il MBR viene letto ed eseguito



Struttura di una partizione:

- Ogni partizione inizia con un boot block
- Il MBR carica il boot block della partizione attiva e lo esegue
- Il boot block carica il sistema operativo e lo esegue
- L'organizzazione del resto della partizione dipende dal file system



Allocazione dello Spazio in Blocchi

- **Allocazione Contigua:**
  - File memorizzati in sequenze contigue di blocchi di dischi
  - Vantaggi:
    - Non necessario l'uso di strutture dati per collegare blocchi
    - Accesso sequenziale è efficiente
    - Accesso diretto è efficiente
  - Svantaggi:
    - Frammentazione esterna
    - I file non possono crescere

- **Allocazione Concatenata:**
  - Ogni file costituito da lista concatenata di blocchi
  - Ogni blocco contiene puntatore al blocco successivo
  - Il descrittore del file contiene puntatori al primo e ultimo elemento della lista
  - Vantaggi:
    - Risolve frammentazione esterna
    - Accesso sequenziale efficiente
  - Svantaggi:
    - Accesso diretto inefficiente
    - Efficienza globale del file system degrada
    - DImensione utile di un blocco non è potenza di due
- **Allocazione basata su FAT:**
  - Invece di usare parte del blocco dati per contenere il puntatore al blocco successivo, si crea una tabella unica con un elemento per blocco
  - Vantaggi: i blocchi dati sono interamente dedicati... ai dati
  - Svantaggi: scansione richiede anche lettura della FAT
- **Allocazione Indicizzata:**
  - L'elenco dei blocchi che compongono un file si memorizza in un blocco indice
  - Per accedere a un file, si carica in memoria la sua area indice
  - Vantaggi:
    - Risolve frammentazione esterna
    - Efficiente per accesso diretto
  - Svantaggi:
    - Spreco di spazio per blocchi indici troppo grandi

Gestione Spazio Libero

- **Mappa di bit:**
  - A ogni blocco corrisponde un bit in una bitmap
  - Blocchi liberi hanno bit 0, quelli occupati 1
  - Vantaggi: semplice
  - Svantaggi: memorizzazione vettore può richiedere molto spazio
- **Lista concatenata:**
  - Blocchi liberi mantenuti in una lista concatenata
  - Vantaggi: richiede poco spazio in memoria centrale
  - Svantaggi:
    - Allocazione di un’area di ampie dimensioni è costosa
    - Allocazione di aree libere contigue è molto difficoltosa