**Processi**

Un processo è un programma in esecuzione, comprende **l’attività corrente**, rappresentata dal valore del contatore di programma e dal **contenuto dei registri** della CPU.  
Normalmente comprende anche il proprio **stack**, contentente a sua volta i dati temporanei e una **sezione dati** contente le variabili globali.  
Un processo può includere uno **heap**, ossia della memoria dinamicamente allocata durante l’esecuzione del processo. Quindi un processo è **un’entità attiva**, con un contatore di programma che specifica qual è l’istruzione successiva da eseguire e un insieme di risorse associate.

**Stato del processo**

Un processo durante l’esecuzione è soggetto a cambiamenti del suo stato, e gli stati in cui può stare sono:

**Nuovo**: Si crea il processo.  
**Esecuzione**: Le sue istruzioni vengono eseguite  
**Attesa**: Il processo attende che si verifichi qualche evento.  
**Pronto**: Il processo attende sdi essere assegnato a un’unità di elaborazione  
**Terminato**: il processo ha terminato l’esecuzione

**Blocco di controllo dei processi**

Ogni processo è rappresentato dal sistema operativo da un **blocco di controllo** (process control block **PCB**).  
Un **PCB** contiene molte informazioni connesse a un processo specifico, tra cui le seguenti:

**Contatore di programma**: Il contatore di programma contiene l’indirizzo della successiva istruzione da eseguire per tale processo.  
  
**Registri della CPU**: I registri variano in numero e tipo secondo l’architettura del calcolatore. Essi comprendono accumulatori, registri indici, puntatori alla cima dello stack (stack pointer) , registri d’uso e registri contenenti i codici di condizione (condition codes).  
  
**Informazioni sullo scheduling di CPU**: Queste informazioni comprendo la priorità del processo, i puntatori alle code di scheduling e tutti gli altri parametri di scheduling.  
  
**Informazioni sulla gestione della memoria:** Queste informaioni possono includere elementi quali il valore dei registri di base e di limite, le tabelle delle pagine o dei segmenti.  
  
**Informazioni di accounting**: Queste informazioni comprendono la quota di uso della CPU e il tempo d’utilizzo della stessa, i limiti di tempo e i numeri dei processi.  
  
**Informazioni sullo stato dell’ I/O**: Queste informazioni comprendo la lista dei dispositivi I/O assegnati a un determinato processo, l’elenco dei file aprti e così via..

In sintesi il **PCB** si usa semplicemente come deposito per tutte le informazioni relative ai vari processi.

**Thread**

Un processo è un programma che si esegue seguendo un unico percorso d’esecuzione detto **thread**.  
Nella maggior parte dei sistemi operativi moderni si è esteso il concetto di processo introducendo la possibilità di avere più percorsi d’esecuzione, in modo da permettere che un processo possa svolgere più di un compito alla volta.  
Questa funzione è particolarmente utile sui sistemi multicore, in cui più thread possono essere eseguiti in parallelo.   
Il PCB viene esteso per includere informazioni su ogni thread.  
Un processo **multithread** è in grado di svolgere più compiti in modo concorrente, la maggior parte delle applicazioni per i moderni computer è multithread, molti kernel di sistemi operativi sono ormai multithread, con i singoli thread dedicati a specifici servizi.

I vantaggi della programmazione multithread si possono classificare in categorie principali:  
**Tempo di risposta,condivisione delle risorse,economia e scalabilità.**

La programmazione multithread offre un meccanismo per un utilizzo più efficiente di questi multiprocessori e aiuta a sfruttare al meglio la concorrenza.

Bisogna tener conto però della distinzio tra parallelismo e concorrenza, un sistema è **parallelo** se può eseguire simultaneamente più di un task,   
Un sistema concorrente, invece supporta più di un task consentendo a tutti di progredire nell’esecuzione

**Librerie dei Thread**Una libreria dei thread fornisce al programmatore una API per la creazione e la gestione dei thread.  
I metodi con cui implementare una libreria dei thread sono essenzialmente due:  
Nel primo, La libreria è collocata interamente a livello utente, senza fare ricorso al kernel. Il codice e le strutture dati per la libreria risiedono tutti nello spazio utente.  
Il secondo metodo, consiste nell’implementare una libreria a livello kernel, supportata direttamente dal sistema operativo. Invocare una funzione della API per la libreria provoca una chiamata di sistema kernel.

Due strategie per la creazione di più thread sono il **threading asincrono** e il **threading sincrono**.

Nel threading asincrono un valota che il genitore crea un thread figlio, riprende la sua esecuzione, in modo che genitore e figlio restino in esecuzione concorrentemente. Ogni thread viene eseguito in modo indipendente rispetto agli altri thread e il thread genitore non ha bisogno di conoscere quando suo figlio termina. Poichè i thread sono indipendenti, vi è di solito poca condivisione dei dati tra thread.

Il threading sincrono si verifica quando il thread genitore crea uno o più figli e attende che tutti terminano prima di riprendere l’esecuzione, in genere il threading sincrono comporta una significativa condivisione dei dati tra i thread.

**Scheduling**

Nel corso della sua esistenza un processo si trova in varie code di scheduling. Il sistema operativo, incaricato di selezionare i processi dalle suddette code, compie la selezione per mezzo di un opportuno **scheduler.**

Lo **scheduler a** **lungo termine** (job scheduler), sceglie i processi da questo insieme e li carica in memoria affinchè siano eseguiti.  
Lo *scheduler a lungo termine* si esegue con frequenza molto inferiore, controlla il grado di multiprogrammazione, cioè il numero di processi presenti in memoria.

Lo **scheduler a breve termine**, fa la selezione tra i processi pronti per l’esecuzione e assegna la CPU a uno di loro.  
Lo *scheduler a breve termine* seleziona frequentemente un nuovo processo per la CPU, un processo può rimanere in esecuzione solo per pochi millisecondi prima di passare ad attendere una richiesta I/O, quindi lo scheduler a breve termine deve essere molto rapido.

Lo scheduler a medio termine è molto vantaggioso poichè elimina i processi dalla memoria riducendo il grado di multiprogrammazione del sistema.

L’esecuzione del processo consiste in un ciclo d’elaborazione e d’attesa del completamento delle operazioni di I/O. L’esecuzione di un processo comincia con una sequenza di operazioni d’elaborazionje svolta dalla CPU, seguita da una sequenza di operazioni di I/O, l’ultima sequenza di operazioni della CPU si conclude con una richiesta al sistema di terminare l’esecuzione.

Ogni qualvolta la CPU passa nello stato di inattività, il sistema operativo sceglie per l’esecuzione uno ddei processi presenti nella ready queue. É lo scheduler a breve termine che tra i processi in memoria pronti per l’esecuzione sceglie quello cui assegnare la CPU.

**Dispatcher**

Un altro elemento coinvolto nella funzione di scheduling della CPU è il dispatcher, si tratta del modulo che passa effettivamente il controllo della CPU al processo scelto dallo scheduler a breve termine, questa funzione comprendere:

* Il cambio di contesto
* Il passaggio alla modalità utente
* Il salto alla giusta posizione del programma utente per riavviarne l’esecuzione.

Poichè si attiva a ogni cambio di contesto, il dispatcher dovrebbe essere molto rapido, il tempo richiesto dal dispatcher per fermare un processo e avviare l’esecuzione di un altro è noto come latenza di dispatch.

**Criteri di scheduling**

Diversi algoritmi di scheduling della CPU hanno proprietà differenti e possono favorire una particolare classe di processi, per il confronto tra gli algoritmi di scheduling della CPU sono stati suggeriti molti criteri tra i quali:

**Utilizzo della CPU**: la CPU deve essere più attiva possibili  
**Produttività**: la CPU è attiva quando si svolge del lavoro, una misura del lavoro svolto è data dal numero dei processi completati nell’unità di tempo, tale misura è detta *throughput.***Tempo di completamento**: è un criterio molto importante, è il tempo necessario per eseguire il processo stesso. L’intervallo che intercorre tra la sottomissione del processo e il completamento dell’esecuzione è chiamato *turnaround time.***Tempo d’attesa:** è la somma degli intervalli d’attesa passati in questa coda.  
**Tempo di risposta**: Una misura di confronto è data dal tempo che intercorre tra l’effettuazione di una richiesta e la prima risposta prodotta.

**Stallo**

Se un processo richiede una risorsa già assegnata a un altro processo, il processo richiedente può essere accodato agli altri processi che attendono tale risorsa. Un gruppo di processi si trova in stallo quando ogni processo del gruppo attende un evento che può essere causato solo da un altro processo che si trova nello stato di attesa.

Si può avere una situazione di stallo solo se in un sistema si verificano contemporaneamente quattro condizioni:

1. **Mutua esclusione**: Almeno una risorsa deve essere non condivisibile, vale a dire che è utilizzabile da un solo processo alla volta. Se un altro processo richiede tale risorsa, si deve ritardare la richiesta fino al rilascio della risorsa.
2. **Possesso e attesa**: Un processo deve essere in possesso di almeno una risorsa e attendere di acquisire risorse già in possesso di altri processi.
3. **Assenza di prelazione**: Le risorse non possono essere prelazionate, vale a dire che una risorsa può essere rilasciata dal processo che la possiede solo volontariamente, dopo aver terminato il proprio compito.
4. **Attesa circolare**: Deve esistere un insieme P di processi, tale che P0 attende una risorsa da P1, e P1 attende una risorsa da P2,..,Pn dove Pn attende una risorsa posseduta da P0.

Si può prevenire il verificarsi di uno stallo assicurando che almeno una di queste condizioni non possa capitare.

Le situazioni di stallo si possono descrivere con maggiore precisione avvalendosi di una rappresentazione detta grafo di assegnazione delle risorse, si tratta di un insieme di vertici V e un insieme di archi E, con l’insieme di vertici V composto da due sottoinsiemi che rappresenta tutti i tipi di risorse del sistema.

**Metodi per la gestione dello stallo**

Essenzialmente il problema delle situazioni di stallo si può affrontare in tre modi:

* Si può usare un protocollo per prevenire o evitare le situazioni di stallo, assicurando che il sistema non entri mai in stallo.
* Si può permettere al sistema di entrare in stallo, individuarlo, e quindi eseguire il ripristino.
* Si può ignorare del tutto il problema, fingendo che le situazioni di stallo non possano mai verificarsi nel sistema.

**Sincronizzazione dei processi**

Un processo cooperante è un processo che può influenzarne un altro in esecuzione nel sistema o anche subirne l’influenza, l’accesso concorrente a dati condivisi può tuttavia causare situazioni di incoerenza degli stessi dati. L’esecuzione parallela può contribuire a problematiche che riguardano l’integrità dei dati condivisi da più processi. Per evitare le situazioni in cui più processi accedono e modificano gli stessi dati in modo concorrente e i risultati dipendono dall’ordine degli accessi, occorre assicurare che un solo processo alla volta possa modificare la variabile contatore. Questa garanzia richiede una forma di sincronizzazione dei processi.  
La caratteristica fondamentale del sistema è che, quando un processo è in esecuzione nella propria sezione critica, non si consente a nessun altro processo di essere in esecuzione nella propria sezione critica.

**Soluzione di Peterson**

Una soluzione al problema della sezione critica nota come soluzione di Peterson, questa soluzione è limitata a due processi, ognuno dei quali esegue alternativamente la propria sezione critica e la sezione non critica, la variabile turn segnala di chi sia il turno d’accesso alla sezione critica.  
Per accedere alla sezione critica, il processo Pi assegna innanzitutto a flag[i] il valore true, quindi attribuisce a turn il valore j, conferendo così all’altro processo la facoltà di entrare nella sezione critica.  
Qualora entrambi i processi tentino l’accesso contemporaneo, all’incirca nello stesso momento sarà a turno sia il valore i sia il valore j.   
Soluzioni basate sul software come quella di Peterson non garantiscono il loro funzionamento su architetture moderne.  
Tutte queste soluzioni si basano sul concetto di lock(lucchetto), ovvero sulla protezione di regioni critiche attraverso l’uso di lock. Le funzionalità hardware possono rendere più facile il compito a un programmatore e migliorare l’efficienza del sistema.  
In un sistema dotato di una singola CPU il problema della sezione critica si potrebbe risolvere semplicemente se si potessero disabilitare le interruzioni mentre si modificano le variabili condivise.

**Lock mutex**  
Il lock mutex è usato per proteggere le regioni critiche e quindi prevenire la race condition. In pratica un processo deve acquisire il lock prima di entrare in una sezione critica e rilasciarlo quando esce dalla sezione critica. Mentre un processo si trova nella sua sezione critica, ogni altro processo che cerca di entrare nella sezione critica deve ciclare continuamente, questo tipo di lock mutex è detto spinlock perché il processo continua a girare in attesa che il lock diventi disponibile. L’attesa attiva spreca cicli di CPU che qualche altro processo potrebbe utilizzare in modo produttivo.

**Semafori**Un semaforo S è una variabile intera cui si può accedere solo tramite due operazioni atomiche predefinite: wait() e signal(). Si usa distinguere tra semafori contatore, il cui valore è un numero intero e i semafori binari il cui valore è limitato a 0 o 1. I semafori binari sono dunque simili ai lock mutex e vengono utilizzati al loro posto per la mutua esclusione. I semafori contatore trovano applicazione nel controllo dell’accesso a una data risorsa presente in un numero finito di esemplari. Il semaforo è inizialmente impostato al numero di risorse disponibili. I processi che desiderano utilizzare una risorsa invocano wait() sul semaforo decrementandone così il valore, i processi che restituiscono una risorsa invece invocano signal() sul semaforo incrementandone il valore. Quando il semaforo vale 0, tutte le risorse sono occupate, e i processi che ne richiedano l’uso dovranno bloccarsi fino a che il semaforo non ritorni positivo.

**Monitor**Un costrutto fondamentale di sincronizzazione ad alto livello è il monitor. Un tipo di dato astratto (ADT) incapsula i dati mettendo a disposizione un insieme di funzioni per operare su di essi, il monitor è un ADT che comprende un insieme di operazioni definite dal programmatore che, all’interno del monitor, sono contraddistinte dalla mutua esclusione. Una funzione definita all’interno di un monitor ha accesso unicamente alle variabili dichiarate localmente, situate nel monitor, e ai relativi parametri formali. Il costrutto monitor assicura che all’interno di un monitor possa essere attivo un solo processo alla volta, sicchè non si deve codificare esplicitamente il vincolo di mutua esclusione.

**MEMORIA CENTRALE**

La memoria consiste in un grande vettore di byte, ciascuno con il proprio indirizzo, la CPU preleva le istruzioni dalla memoria sulla base del contenuto del contatore di programma, tali istruzioni possono determinare ulteriori letture(load) e scritture(store) in specifici indirizzi di memoria.  
La memoria centrale e i registri incorporati nel processore sono le sole aree di memorizzazione a cui la CPU può accedere direttamente. I dati che non sono in memoria devono essere caricati prima che la CPU possa operare su di loro. I registri incorporati nella CPU sono accessibili nell’arco di un ciclo di clock della CPU.  
In genere un programma risiede in un disco sotto forma di un file binario eseguibile. Durante la sua esecuzione, il processo si può trasferire dalla memoria al disco e viceversa. L’insieme dei processi presenti nei dischi e che attendono d’essere trasferiti in memoria per essere eseguiti forma la coda d’ingresso(input queue).

Generalmente l’associazione di istruzioni e dati a indirizzi di memoria si può compiere in qualsiasi passo del seguente percorso:

* **Compilazione**: Se nella fase di compilazione si sa dove il processo risiederà in memoria, si può generare codice assoluto.
* **Caricamento**: Se nella fase di compilazione non è possibile sapere in che punto della memoria risiederà il processo, il compilatore deve generare codice rilocabile.
* **Esecuzione**: Se durante l’esecuzione il processo può essere spostato da un segmento di memoria a un altro, si deve ritardare l’associazione degli indirizzi fino alla fase d’esecuzione.

**Avvicendamento dei processi (swapping)**

Per essere eseguito un processo deve trovarsi nella memoria centrale, un processo tuttavia può essere temporaneamente tolto dalla memoria centrale e spostato in una memoria ausiliaria(backing store) e in seguito riportato in memoria per continuare l’esecuzione. Questo procedimento si chiama avvicendamento dei processi in memoria o swapping. Grazie allo swapping lo spazio totale degli indirizzi fisici di tutti i processi può eccedere la reale dimensione della memoria fisica del sistema, aumentando così il grado di multiprogrammazione.

La memoria ausiliaria è di solito costituita da un disco veloce, tale memoria deve essere abbastanza ampia da contenere le copie di tutte le immagini di memoria di tutti i processi utenti. Il sistema mantiene una coda dei processi pronti (ready queue) formata da tutti i processi pronti per l’esecuzione, le cui immagini di memoria si trovano in memoria ausiliaria o in memoria centrale.

**ALLOCAZIONE CONTIGUA DELLA MEMORIA**La memoria centrale deve contenere sia il sistema operativo sia i vari processi utenti; la memoria centrale di solito si divide in due partizioni, una per il sistema operativo residente e una per i processi utente, generalmente si vuole che più processi utenti risiedano contemporaneamente in memoria centrale.  
Con l’allocazione contigua della memoria, ciascun processo è contenuto in una singola sezione di memoria contigua a quella che contiene il processo successivo. **ALLOCAZIONE DELLA MEMORIA**Uno dei metodi più semplici per l’allocazione della memoria consiste nel suddividere la stessa in partizioni di dimensione fissa. Ogni partizione deve contenere esattamente un processo, quindi il grado di multiprogrammazione è limitato dal numero di partizioni. In questo metodo delle partizioni multiple quando una partizione è libera può essere occupata da un processo presente nella coda d’ingresso; terminato il processo la partizione diventa nuovamente disponibile per un altro processo.   
Nello schema a partizione variabile il sistema operativo conserva una tabella in cui sono indicate le partizioni di memoria disponibili e quelle occupate, inizialmente tutta la memoria è a disposizione dei processi utenti, si tratta di un grande blocco di memoria disponibile, un **buco**(hole).

I criteri più usati per scegliere un buco libero tra quelli disponibili nell’insieme sono:

* **First**-**fit**: Si assegna il primo buco abbastanza grande.
* **Best-fit:** Si assegna il più piccolo buco in grado di contenere il processo
* **Worst-fit:** Si assegna il buco più grande..

First-fit è il più veloce.

**Frammentazione**Entrambi i criteri first-fit e best-fit di allocazione della memoria soffrono di frammentazione esterna. Caricando e rimuovendo i processi dalla memoria, si frammenta lo spazio libero della memoria in tante piccole parti. Si ha la frammentazione esterna se lo spazio libero della memoria totale è sufficiente per soddisfare una richiesta, ma non è contiguo; la memoria è frammentata in tanti piccoli buchi. Questo problema di frammentazione può essere molto grave, nel caso peggiore può verificarsi un blocco di memoria libera sprecata tra ogni coppia di processi.  
Sia che si adotti l’uno o l’altro, l’impiego di un determinato criterio può influire sulla quantità di frammentazione: in alcuni sistemi dà migliori risultati il first fit, in altri dà migliori risultati il best-fit.  
Esiste anche la frammentazione interna, che consiste nella memoria inutilizzata all’interno di una partizione.  
Una soluzione al problema della frammentazione esterna è data dalla compattazione, il quale scopo è quello di ordinare il contenuto della memoria per riunire la memoria libera in un unico grosso blocco.  
Un’altra possibile soluzione del problema della frammentazione esterna è quella data dal consentire la non contiguità dello spazio degli indirizzi logici di un processo, permettendo così di assegnare la memoria fisica ai processi dovunque essa sia disponibile.

**Segmentazione**La segmentazione è uno schema di gestione della memoria che supporta questa rappresentazione della memoria dal punto di vista del programmatore. Uno spazio di indirizzi logici è una raccolta di segmenti, ciascuno dei quali ha un nome e una lunghezza. Gli indirizzi specificano sia il nome sia l’offset all’interno del segmento, quindi il programmatore fornisce ogni indirizzo come una coppia ordinata di valori: un nome di segmento e un offset.  
Normalmente quando un programma viene compilato il compilatore costruisce automaticamente i segmenti in rapporto al programma sorgente. La memoria fisica è in ogni caso una sequenza di byte unidimensionale. Per questo motivo occorre tradurre gli indirizzi bidimensionali definiti dall’utente negli indirizzi fisici unidimensionali. Questa operazione si compie tramite una tabella dei segmenti; ogni suo elemento è una coppia ordinata: la base del segmento e il limite del segmento. La base del segmento contiene l’indirizzo fisico iniziale della memoria dove il segmento risiede, mentre il limite del segmento contiene la lunghezza del segmento.

**Paginazione**La segmentazione permette che lo spazio degli indirizzi fisici di un processo non sia contiguo. La paginazione è un altro schema di gestione della memoria che offre lo stesso vantaggio. Tuttavia a differenza della segmentazione, la paginazione evita la frammentazione esterna e necessita di compattazione. Inoltre la paginazione elimina il gravoso problema della sistemazione di blocchi di memoria di diverse dimensioni in memoria ausiliaria.  
Il metodo di base per implementare la paginazione consiste nel suddividere la **memoria fisica in blocchi** di dimensione fissa, detti **frame** e nel suddividere la memoria logica in blocchi di pari dimensione, detti **pagine**.  
Quando si deve eseguire un processo, si caricano le sue pagine nei frame disponibili, prendendole dalla memoria ausiliaria o dal file system.  
Ogni indirizzo generato dalla CPU è diviso in due parti: un numero di pagina e un offset di pagina, il numero di pagina serve come indice per la tabella delle pagine , contenente l’indirizzo di base in memoria fisica di ogni pagina.  
Poiché il sistema operativo gestisce la memoria fisica, deve essere informato dei dettagli della allocazione, quali frame sono assegnati, quali sono disponibili, il loro numero totale e cosi via… In genere queste informazioni sono contenute in una struttura dati chiamata tabella dei frame, contenente un elemento per ogni frame, indicante se sia libero oppure assegnato e, se è assegnato a quale pagina di quale processo o di quali processi.  
L’implementazione hardware della tabella delle pagine si può realizzare in modi diversi, nel caso più semplice, usa uno specifico insieme di registri.  
La maggior parte dei calcolatori contemporanei usa tabelle molto grandi , per esempio di un milione di elementi, quindi non si possono impiegare i registri veloci per realizzare la tabella delle pagine, quest’ultima viene invece mantenuta nella memoria principale e un registro di base della tabella delle pagine punta alla tabella stessa.  
La TLB è una piccola cache hardware, una memoria associativa ad alta velocità in cui ogni elemento consiste di due parti: una chiave e un valore.  
La TLB si usa insieme con la tabella delle pagine nel modo seguente: la TLB contiene una piccola parte degli elementi della tabella delle pagine, quando la CPU genera un indirizzo logico, si presenta il suo numero di pagina alla TLB; se questo numero è presente, il corrispondente numero del frame è immediatamente disponibile e si usa per accedere alla memoria. Se nella TLB non è presente il numero di pagina, situazione nota come insuccesso della TLB(TLB miss), si deve consultare la tabella delle pagine in memoria. La percentuale di volte che il numero di pagina di interesse si trova nella TLB è detta tasso di successi(hit ratio).

**Memoria virtuale**

**Paginazione su richiesta**

Una strategia consiste nel caricare le pagine nel momento in cui servono realmente, si tratta di una tecnica detta paginazione su richiesta, comunemente adottata dai sistemi con memoria virtuale. Secondo questo schema, le pagine sono caricate in memoria solo quando richieste durante l’esecuzione del programma: ne consegue che le pagine cui non si accede mai non sono mai caricate nella memoria fisica.  
I processi risiedono in memoria secondaria(generalmente su disco), per eseguire un processo occorre caricarlo in memoria, tuttavia anziché caricare in memoria l’intero processo, si può eseguire un metodo di avvicendamento “pigro” (lazy swapping), non si carica mai in memoria una pagina che non sia necessaria. Nell’ambito dei sistemi con paginazione su richiesta l’uso del termine avvicendamento non è appropriato… Uno swapper manipola interi processi mentre un paginatore gestisce le singole pagine dei processi.  
Durante l’esecuzione, finchè il processo accede alle pagine residenti in memoria, l’esecuzione procede come di consueto. L’accesso a una pagina contrassegnata come non valida causa un evento o eccezione di page fault.

**Sostituzione di pagina**

La sostituzione delle pagine segue il seguente criterio, se nessun frame è libero, ne viene liberato uno attualmente inutilizzato. E’ possibile liberarlo scrivendo il suo contenuto nell’area di swap e modificando la tabella delle pagine per indicare che la pagina non si trova più in memoria. Il frame liberato si può usare per memorizzare la pagina che ha causato il fault. Si modifica la procedura di servizio dell’eccezione di page fault in modo da includere la sostituzione della pagina:  
1. Si individua la locazione su disco della pagina richiesta;  
2. Si ceca un frame libero:  
 a. se esiste, lo si usa.  
 b. altrimenti si impiega un algoritmo di sostituzione delle pagine per scegliere un frame vittima;  
 c. si scrive la pagina “vittima” nel disco, di conseguenza la tabella delle pagine e quella dei frame.  
3. Si scrive la pagina richiesta nel frame appena liberato, si modificano le tabelle delle pagine e dei frame.  
4. Si riprende il processo utente dal punto in cui si è verificato il page fault.

Se non esiste alcun frame appena libero sono necessari due trasferimenti di pagine, uno fuori e uno dentro la memoria. Questa situazione raddoppia il tempo di servizio del page fault e aumenta di conseguenza anche il tempo effettivo d’accesso. Questo sovraccarico si può ridurre usando un bit di modifica. In questo caso l’hardware del calcolatore dispone di un bit di modifica, associato a ogni pagina che viene posto a 1 ogni volta nelle pagina si scrive un byte, indicando che la pagina è stata modificata. Quando si sceglie una pagina da sostituire si esamina il suo bit di modifica, se è a 1 significa che quella pagina è stata modificata rispetto a quando era stata letta dal disco, in questo caso la pagina deve essere scritta nel disco. Se il bit di modifica è rimasto a 0, significa che la pagina non è stata modificata da quando è stata caricata in memoria, quindi non è necessario scrivere nel disco la pagina di memoria, perché c’è già. Questa tecnica vale anche per le pagine di sola lettura, per esempio pagine di codice binario.   
La sostituzione di una pagina è fondamentale al fine della paginazione su richiesta, perché completa la separazione tra memoria logica e memoria fisica. Con questo meccanismo si può mettere a disposizione dei programmatori una memoria virtuale enorme con una memoria fisica più piccola. Senza la paginazione su richiesta, gli indirizzi utente si fanno corrispondere a indirizzi fisici e due insiemi di indirizzi possono essere diversi.  
Per realizzare la paginazione su richiesta è necessario risolvere due problemi principali : occorre sviluppare un algoritmo di allocazione dei frame e un algoritmo di sostituzione delle pagine.

**Allocazione dei frame**Le strategie di allocazione dei frame sono soggette a parecchi vincoli. Non si possono assegnare più frame di quanti siano disponibili, sempre che non vi sia condivisione di pagine. Inoltre è necessario assegnare almeno un numero minimo di frame.  
Una delle ragioni per allocare sempre un numero minimo di frame è legata alle prestazioni. Ovviamente al decrescere del numero dei frame allocati a ciascun processo aumenta il tasso di page fault, con conseguente rallentamento dell’esecuzione dei processi. Inoltre va ricordato che quando si verifica un page fault prima che sia stata completata l’esecuzione di un’istruzione quest’ultima deve essere riavviata. Di conseguenza i frame disponibili devono essere in numero sufficiente per contenere tutte le pagine cui ogni singola istruzione può far riferimento.  
Il modo più semplice per suddividere m frame tra n processi è quello per cui a ciascuno si dà parte uguale, m/n frame. Dati 93 frame e cinque processo, ogni processo riceve 18 frame, i tre frame lasciati liberi si potrebbero usare come buffer di frame liberi, questo schema è chiamato allocazione uniforme.  
Un’alternativa consiste nel riconoscere che diversi processi hanno bisogno di quantità di memoria diverse.   
E’ possibile ricorrere anche all’allocazione proporzionale, secondo cui la memorai disponibile si assegna a ciascun processo secondo la propria dimensione.   
***Allocazione globale e allocazione locale***Un altro importante fattore che riguarda il modo in cui si assegnano i frame ai vari processi è la sostituzione delle pagine. Nei casi in cui vi siano più processi in competizione per i frame, gli algoritmi di sostituzione delle pagine si possono classificare in due categorie generali: sostituzione globale e sostituzione locale. La sostituzione globale permette che per un processo si scelga un frame per la sostituzione dall’insieme di tutti i frame, anche se quel frame è al momento allocato a un altro processo; un processo può dunque sottrarre un frame a un altro processo. La sostituzione locale richiede invece che per ogni processo si scelga un frame solo dal proprio insieme di frame.  
Con la strategia di sostituzione locale, il numero di blocchi di memoria assegnati a un processo non cambia. Con la sostituzione globale può accadere che per un certo processo si selezionino solo frame allocati ad altri processi, aumentando così il numero di frame assegnati a quel processo., purchè altri non scelgano per la sostituzione i suoi frame

**Memoria Secondaria**

**Strutture RAID**

L’evoluzione tecnologica ha reso le unità a disco progressivamente più piccole e meno costose, tanto che oggi è economicamente possibile equipaggiare un sistema elaborativo con molti dischi. La presenza di più dischi, qualora si possano usare in parallelo, rende possibile l’aumento della frequenza a cui i dati si possono leggere o scrivere. Inoltre una configurazione di questo tipo permette di migliorare l’affidabilità della memoria secondaria poiché diventa possibile memorizzare le informazioni in più dischi in modo ridondante, in questo caso, un guasto a uno dei dischi non comporta la perdita di dati.  
Ci sono varie tecniche per l’organizzazione dei dischi, note col nome comune di batterie ridondanti di dischi (**RAID**), che hanno lo scopo di affrontare i problemi di prestazioni e affidabilità.  
Attraverso l’uso di più dischi è possibile anche migliorare la capacità di trasferimento distribuendo i dati in sezioni su più dischi, la forma più semplice di questa distribuzione (data striping) consiste nel distribuire i bit di ciascun byte su più dischi, in questo caso si parla di sezionamento o striping a livello dei bit.

Sono stati proposti numerosi schemi per fornire ridondanza a basso costo usando l’idea dello striping combinata con i bit di parità, questi schemi sono stati classificati in livelli chiamati livelli RAID.

* **LIVELLO 0:** Si riferisce ad array di dischi con striping a livello di blocchi, ma senza ridondanza.
* **LIVELLO 1:** Si riferisce alla tecnica di mirroring.
* **LIVELLO 2:** Anche noto come organizzazione con codici per la correzione degli errori di memoria. In un sistema di questo tipo, ogni byte di memoria può avere associato un bit di parità che indica se i bit con valore 1 nel byte sono in numero pari oppure dispari. Se si altera uno dei bit nel byte la parità di byte cambia e quindi non concorda più con la parità memorizzata. In questo modo si identificano tutti gli errori di un singolo bit nel sistema di memoria.
* **LIVELLO 3:** Organizzazione con bit di parità intercalati, si migliora l’organizzazione del livello 2 considerando che, a differenza dei sistemi di memoria centrale, i controllori dei dischi possono rilevare se un settore è stato letto correttamente, così che un unico bit di parità si può usare sia per individuare gli errori sia per correggerli.
* **LIVELLO 4:** Organizzazione con blocchi di parità intercalati, si impiega lo striping a livello di blocchi come nel RAID di livello 0 e inoltre si tiene un blocco di parità in un disco separato per i blocchi corrispondenti negli altri n dischi. La lettura di un blocco richiede l’accesso a un solo disco, permettendo la gestione di altre richieste da parte di altri dischi, quindi la capacità di trasferimento dei dati per ciascun accesso è minore, ma gli accesso in lettura possono procedere in modo parallelo ottenendo una velocità complessiva di I/O più alta.
* **LIVELLO 5:** organizzazione con blocchi intercalati a parità distribuita, differisce dal livello 4 per il fatto che invece di memorizzare i dati in n dischi la parità in un disco separato, i dati e le informazioni di parità sono distribuite tra gli n+1 dischi.
* **LIVELLO 6:** schema di ridondaza P + Q, è molto simi al raid di livello 5, ma memorizza ulteriori informazioni ridondanti per potere festire guasti contemporanei di più dischi.