STUDIO SISTEMI OPERATIVI TEORIA

Introduzione

Compiti di un sistema operativo: Il sistema operativo è quel software che ha il compito di controllare gli altri programmi e fa da interfaccia tra applicazione ed hardware

Servizi di un sistema operativo:

- Creazione programmi e la loro esecuzione
- Accesso ai dispositivi I/O
- Accesso ai file
- Accesso al sistema

Ecc.

Si dota di 3 interfacce:

1. ISA (instruction set architecture)
2. ABI (application binary interface)
3. API (application programming interface)

Gestione delle risorse: il sistema operativo gestisce i dati dei programmi però esso stesso è un normale programma gestito dal processore, quindi potrebbe perdere il controllo di quest'ultimo se venisse richiesto il processore per un altro programma.

Evoluzione dei sistemi operativi:

- Prima generazione macchine con valvole, senza sistema operativo con problemi di schedulazione
- Seconda generazione introdotti transistor e monitor che ordina il caricamento dei programmi
    - Gestione della memoria: introdotti meccanismi di protezione della memoria che evita accessi indesiderati, timer, istruzioni privilegiate e interruzioni per far spostare l'esecuzione
- Terza generazione introdotti circuiti integrati e la multiprogrammazione ossia la capacità di eseguire processi contemporaneamente questo però porta a dover gestire lo scheduling perché andrà alternata l'esecuzione di ciascun programma. C'è anche il time sharing in cui il processore è condiviso da più utenti che accedono simultaneamente e quindi andrà gestito anche il controllo degli accessi
    - Multiprogrammazione: massimizzare l'uso del processore
    - Time-sharing: minimizzare il tempo di risposta
- Quarta e quinta generazione: introduzione dei personal computer

Il sistema di elaborazione

Unità centrale:

- Microprocessore è un processore su un singolo chip è dotato di cache
    - Multiprocessore s'intende più processori per un singolo chip
- Processore grafico (GPU) computa su array di dati utilizzando la tecnica single instruction multiple data
- Processore di segnale digitale (DSP) codifica/decodifica segnali audio video
- SoC sistema su chip per esempio per smartphone

Ciclo Fetch-Execute: viene prelevata l'istruzione dalla ram, si esegue, s'incrementa il PC (program counter) e si passa all'istruzione successiva

Interruzioni: permettono d'interrompere il ciclo fetch-execute così da sospendere l'esecuzione di un programma ad esempio. Ci sono varie classi d'interruzioni: generate da errori di esecuzione, generate dall'esecuzione di funzioni ad intervalli regolari (con il timer), generate da un controller I/O o generate da

errori hardware. La gestione delle interruzioni avviene aggiungendo l'interrupt stage al ciclo di fetch-execute

Interruzioni multiple: può capitare che si generi un'interruzione dentro un'altra per questo o si elimina la possibilità che accada o si usa uno schema a priorità in cui certe eccezioni hanno priorità più alta

Architettura dei sistemi operativi:

- **Aspetti dello sviluppo:**
  - Processi
  - Gestione della memoria

  Ecc.

- **Approcci ed elementi progettuali:**
  - Multi thread
  - Sistemi operativi distribuiti

  Ecc.

Macchine virtuali: il sistema operativo crea delle macchine appunto virtuali con un proprio sistema operativo e componenti

Struttura del sistema operativo

Servizi del sistema operativo:

- Utente ed altri programmi
- Interfaccia utente
- Chiamate di sistema
- Servizi
- Sistema operativo
- Hardware

Servizi per l'utente:

- Interfaccia come la GUI o linea di comando
- Esecuzione e terminazione di programmi
- Operazioni I/O
- File-system tra cui anche la gestione dei permessi
- Comunicazione fra processi autorizzando la condivisione di memoria

Ecc.

Interfacce utente:

- Interprete dei comandi (CLI) esegue in sequenza i comandi specificati e può essere implementato nel nucleo del sistema oppure essere un programma
- Interfaccia grafica (GUI) ci sono le icone che rappresentano eseguibili e directory

Confronto: La CLI permette di automatizzare compiti ripetitivi mentre la GUI è più intuitiva sacrificando però l'accesso a tutte le funzioni

Chiamate di sistema: sono un'interfaccia ai servizi del sistema operativo e vi si accede tramite un'API che nasconde i dettagli all'utilizzatore e i parametri son passati tramite registri o blocchi

Categorie di chiamate di sistema:

- **Controllo dei processi**
  - Creazione e arresto

- Caricamento ed esecuzione

  Ecc.

- **Gestione dei file**
  - Creazione e cancellazione
  - Apertura, lettura e posizionamento

  Ecc.

- **Gestione dei dispositivi I/O**
  - Richiesta e rilascio
  - Impostazione degli attributi

  Ecc.

- **Gestione delle informazioni**
  - Impostazione ora e data
  - Impostazione degli attributi dei processi

  Ecc.

- **Comunicazione**
  - Creazione e chiusura di una connessione
  - Invio e ricezione di messaggi

  Ecc.

- **Protezione**
  - Controllo degli accessi
  - Impostazione dei permessi
    Ecc.

Programmi di sistema: servono a realizzare l'ambiente per l'esecuzione dei programmi e ce ne sono di diverse categorie: gestione files, informazioni di stato, editor, Ecc.

Progettazione del sistema operativo: innanzitutto si parte dal decidere gli obiettivi e le specifiche come che hardware utilizzare, gli obiettivi del sistema. Poi si passa all'implementazione vera e propria utilizzando linguaggi di alto livello anche se certe parti vengono scritte in assembler perché più efficiente

Struttura del sistema operativo: ci sono i sistemi monolitici in cui non c'è una divisione in moduli; poi ci sono i sistemi stratificati in cui ogni strato è l'implementazione di un oggetto astratto fatto di dati e per ogni strato non serve conoscere quello sopra o sotto questo porta vantaggi come una semplicità di scrittura; poi c'è il microkernel che fa il minimo necessario e trasferisce tutto a programmi livello utente e lo scambio di servizi avviene tramite scambio di messaggi tutto questo potrebbe causare overhead perché è richiesto maggior tempo di esecuzione visto che molti compiti sono affidati a programmi. Infine ci sta la struttura a moduli utilizzata da Solaris per esempio, in cui il nucleo ha solo le funzioni fondamentali il resto è in moduli che possono essere caricati all'occorrenza ed ognuno di essi nasconde i dettagli implementativi agli altri

Debugging: è una fase che ha lo scopo di trovare gli errori del sistema e valutarne anche le prestazioni

Generazione del sistema operativo: scritto il codice sorgente, bisogna compilarlo e configurarlo per il sistema su cui verrà eseguito, poi va installato ed avviato. La fase di configurazione la possiamo fare prima della compilazione modificandolo in base al sistema oppure tramite la selezione di tabelle precompilate da collegare al momento dell'installazione

Avvio del sistema: l'esecuzione parte da un indirizzo predefinito della ROM contenente il bootstrap loader ossia l'insieme delle istruzioni per caricare il bootstrap program che a sua volta carica il kernel portando il sistema in esecuzione

Processi

Un processo o job è un programma in esecuzione

Elementi del processo:

- Program counter che contiene l'indirizzo dell'istruzione che dev'essere eseguita e contenuto dei registri (immagine del processore). In breve quando il processo è in stato d'esecuzione il processore preleva l'istruzione dal PC e in base al contenuto dei registri procede ad eseguirla, quando però il processo non ha più possibilità di avanzare, è inutile lasciare il processore con i dati dei registri e quindi l'immagine del processore viene salvata per passare ad un altro processo
- Stack che contiene i dati temporanei come variabili locali o parametri di funzioni
- Programma (sezione testo)
- Sezione dati (var. globali)
- Heap (spazio per l'allocazione dinamica dei dati)

Differenze programma/processo: il programma è l'entità passiva, l'eseguibile mentre il processo è l'entità attiva che esegue le istruzioni e utilizza le risorse associate e un programma può essere associato a più processi

Processo in memoria: nella memoria il processo occupa un certo spazio che contiene (dal basso verso l'alto) text, data, heap e stack. L'heap si può allargare verso lo stack man mano che usiamo malloc mentre lo stack verso il basso man mano che chiamiamo funzioni, non devono scontrarsi altrimenti uno andrebbe dentro l'area di memoria dell'altro

Stati del processo: un processo quando viene creato (new) si mette nella coda in attesa del processore (ready) poi da lì verrà eseguito (running) e o termina (terminated) quindi il processore ha completato la sua esecuzione, o viene interrotto quindi viene rimandato in attesa del processore oppure viene messo in attesa (waiting) di un evento che lo riporti poi in coda per il processore

- **New:** in fase di creazione
- **Ready:** processo attende la disponibilità del processore
- **Running:** processore sta eseguendo il processo
- **Terminated:** il processo ha completato l'esecuzione
- **Waiting:** processo è in attesa di un evento

PCB (descrittore di processo): è una struttura dati associata al processo che contiene tutte le informazioni relative ad esso come: stato del processo, program counter, registri, ecc.

Threads: un processo può avere più sequenze d'istruzioni che possono essere eseguite in parallelo (thread) in tal caso il PCB avrà le informazioni per ogni thread

Salvataggio del PCB: supponiamo che venga creato il processo p0 e che viene eseguito (running) ad un certo punto succede un'interruzione e tutte le sue informazioni allora verranno salvate nel suo PCB così che quando riprenderà l'esecuzione saranno ricaricate.

Code di schedulazione: i PCB sono inseriti in varie code: coda job che contiene tutti i processi, coda ready che contiene tutti i processi pronti a ricevere il processore e coda dei dispositivi che è una per ogni dispositivo e ognuna contiene i processi in attesa

Schedulatori: il sistema operativo li usa per decidere a quali processi assegnare il processore

- **A lungo termine:** prende i processi in coda job e li inserisce in coda ready (job scheduler), è eseguito con poca frequenza per evitare che si accumuli una lunga lista di processi nella coda ready il che rallenterebbe il sistema. Poiché non è coinvolto direttamente nell'esecuzione sulla CPU può adottare politiche complesse ad esempio evitando l'ingresso di troppi processi CPU-bound (processi che utilizzano la CPU in modo continuativo)
- **A breve termine:** prende i processi dalla coda ready e gli assegna il processore (CPU scheduler) dev'essere eseguito con grande frequenza perché gestisce l'esecuzione. I processi che stanno usando la CPU possono essere interrotti e rimessi nella coda ready permettendo così l'avanzamento simultaneo di più processi
- **A medio termine:** gestisce la sospensione e la riattivazione dei processi liberando o riallocando la memoria (swapping), serve ad alleggerire il carico del sistema anche se grazie alla paginazione nei sistemi moderni ogni processo ha una propria porzione di memoria con ciò che gli serve in quel momento. (sposta temporanemanete alcuni processi sul disco per liberare RAM)

Come selezionare i processi: ci sono delle politiche per l'assegnazione della CPU per esempio i processi CPU-bound vengono gestiti in maniera differente da quelli I/O-bound (occupano poco la CPU perché molto spesso sono in attesa di dispositivi I/O). Si può incorrere nel problema dell'overhead che varia in funzione della velocità della memoria e del numero dei registri

Creazione dei processi: parte tutto da un processo padre che crea un processo figlio che a sua volta può crearne altri generando un albero dei processi. Ogni processo è identificato da un numero, il pid (identificatore di processo). Le risorse possono essere ereditate dal padre, distribuite in parte o completamente condivise, oppure assegnate direttamente dal sistema operativo. Invece per l'esecuzione padre e figlio possono essere eseguiti in modo concorrente se il padre non attende il figlio quest'ultimo potrebbe diventare un processo zombie (terminato ma ancora presente nella tabella dei processi) se invece il padre terminasse prima del figlio il figlio diventa orfano (venendo adottato da un processo speciale init in Unix). A livello di memoria il processo figlio può eseguire lo stesso programma del padre duplicandone la memoria oppure caricarne uno diverso.

Threads

Processi e threads: I processi sono caratterizzati dalle risorse allocate e lo scheduling, queste due caratteristiche potrebbe esser trattate in maniera indipendente facendo che il processo è l'unità che contiene tutte le risorse necessarie all'esecuzione, mentre il thread è l'unità d'esecuzione che viene assegnata al processore. Un processo può contenere uno o più thread che condividono le stesse risorse del processo avendo pur sempre un proprio contesto d'esecuzione mentre un thread fa sempre parte di un solo processo

Multi-threading: è la capacità di un sistema operativo di permettere più thread in un singolo processo, il contrario è il single-threading

Processo (multi-thread): il processo è l'unità di allocazione delle risorse e un'unità di protezione (un thread non può uscire dallo spazio del processo). Ad esso sono associati: lo spazio d'indirizzamento virtuale (contiene l'immagine del processo), l'accesso protetto al processore e uno o più thread

Thread: ad esso sono associati: stato che può essere pronto, in attesa o in esecuzione, contesto del thread (immagine del processore), lo stack di esecuzione, spazio di memoria per le var. locali e accesso a memoria e risorse del processo. Come il processo ha il proprio contesto

Benefici del multi-threading:

- La creazione di un thread è più rapida di quella di un processo e lo stesso vale per la terminazione
- Lo switch tra thread è molto più veloce di quello tra due processi (i thread ovviamente sono dello stesso)
  Ecc.

Si noti che la schedulazione è gestita principalmente nel thread ma non la sospensione e la terminazione

Stati del thread: sono come quelli dei processi quindi in esecuzione, pronto e in attesa

Operazioni sui thread: creazione, sospensione, riattivazione e terminazione

Sincronizzazione dei thread: i thread di uno stesso processo condividono spazio d'indirizzamento e altre risorse quindi la modifica simultanea di un dato da parte di due thread può corrompere il dato ecco perché è necessario sincronizzare l'accesso dei thread alle risorse

Thread a livello utente: la gestione dei thread è effettuata dall'applicazione nello spazio utente ed il kernel non sa nulla dell'esistenza dei thread, quindi non è richiesta la modalità kernel, inoltre la schedulazione può essere specifica per l'applicazione. Gli svantaggi sono che non si trae vantaggio dai multiprocessori siccome il kernel non sa che dentro ci sono thread che potrebbe assegnare a core diversi e che l'uso di system call blocca l'intero processo visto che il kernel non può distinguere quale thread ha effettuato la chiamata (risolvibile usando il jacketing)

Thread a livello kernel: la gestione dei thread è effettuata direttamente dal kernel e l'applicazione utilizza delle API di sistema per gestirli. Il kernel gestisce i PCB dei processi e dei thread (lo stato del processo non necessariamente coincide con lo stato del thread) quindi può schedulare ogni thread individualmente. A differenza di quelli a livello utente, i thread possono essere eseguiti in parallelo su più processori, una system call bloccante blocca solo il thread interessato però le operazioni sui thread avvengono in modalità kernel quindi i cambi di modalità (utente/kernel) causano overhead

Approcci misti:

- creazione, schedulazione e sincronizzazione dei thread avviene nello spazio utente quindi non coinvolgendo il kernel risultano più veloci
- i thread in modalità utente vengono associati a uno o più thread in modalità kernel permettendo così l'esecuzione parallela

Legge di Amdahl

$$speedup = \frac{tempo\ di\ esecuzione\ su\ un\ processore}{tempo\ di\ esecuzione\ su\ N\ processori} = \frac{1}{(1-f)+\frac{f}{n}}\ (\frac{1}{n}\ è\ la\ situazione\ ottimale)$$

Descrive il limite massimo di speedup che si può ottenere parallelizzando un programma in base alla parte che può effettivamente essere eseguita in parallelo (f è la frazione di programma che può essere parallelizzata ed n il numero di processori). Quindi anche se abbiamo moltissimi processori se poca parte del programma è parallelizzabile lo speedup non sarà altissimo

Threads in UNIX: il kernel non distingue tra processi e thread ma sono entrambi visti come task e per ognuna c'è un TCB (task control block) con fork() creiamo una nuova task con una copia del TCB mentre con clone() creiamo una task che nel TCB ha dei puntatori ai campo del TCB padre

Mutua esclusione

Quando più processi o thread accedono a risorse condivise possono interferire tra loro in modo positivo o negativo. Negativo più processi accedono contemporaneamente ad una risorsa critica, mentre positiva quando collaborano ad esempio dividendo il lavoro all'interno di un'unica applicazione

Gestione dei processi:

- **Multiprogrammazione:** molti processi un singolo processore
- **Multiprocessing:** molti processi gestiti da un multiprocessore
- **Processi distribuiti:** molti processi gestiti da molti processori distribuiti (cluster gruppo di pc connessi per un obiettivo comune)

I differenti processi hanno bisogno di risorse che sono condivise quindi si crea concorrenza ad esempio come può capitare in applicazioni strutturare (più processi o threads che devono accedere alla risorsa)

Principi della concorrenza:

- **Uniprocessore multiprogrammato:** l'esecuzione dei processi si alterna interrompendo certi processi e riprendendone altri
- **Multiprocessore:** le esecuzioni si alternano e si sovrappongono (eseguiti davvero in parallelo)

In entrambi i casi questi processi condividono risorse e possono interagire in modo imprevedibile ad esempio accedendo alla stessa risorsa e non si può prevedere chi ci accederà per primo perché ciò dipende da molti fattori come le politiche di schedulazione, ecc. Quest'imprevedibilità può causare problemi nell'allocazione ottimale delle risorse perché non possiamo determinare quale processo avrà effettivamente bisogno di spazio, e nel debug perché un errore causato da un'interazione concorrente può non ripetersi sempre rendendo il problema difficilmente re-individuabile. In sostanza è necessario controllare il codice che accede alla variabile in modo da rendere essa indisponibile quando utilizzata da un altro

Race condition: più di un processo (o thread) leggono e scrivono dati in modo che il risultato finale dipende dall'ordine di esecuzione delle istruzioni dei processi RISULTATO DIPENDE DALL'ORDINE cioè se abbiamo due processi che assegnano due valori diversi ad una variabile condivisa, il risultato finale sarà del secondo arrivato

Problemi determinati dalla concorrenza: è compito del sistema operativo gestire la concorrenza tenendo traccia dei processi (PCB), proteggere i dati di ogni processo da interferenze e rendere il risultato di ogni processo indipendente dalla sua velocità

Competizione per le risorse: quando più processi concorrenti cercano di accedere a risorse condivise possono sorgere diversi problemi come la mutua esclusione (sezione critica) solo un processo per volta può usare una risorsa gli altri devono aspettare, deadlock più processi sono in attesa di risorse già occupate da altri anch'essi in attesa creando un loop e starvation un processo continua a non ottenere la risorsa di cui ha bisogno perché altri la prendono sempre prima di lui

Cooperazione per condivisione: i processi interagiscono senza conoscersi, condividendo i dati. Il risultato di un processo può dipendere dalle azioni di un altro infatti va assicurata l'integrità dei dati. Le problematiche sono sempre le stesse però le operazioni in lettura non richiedono mutua esclusione

Cooperazione per scambio di messaggi: i processi collaborano per un obiettivo comune, la comunicazione permette la sincronizzazione e gli strumenti per attuarla sono forniti dal kernel. Non c'è problema di mutua esclusione perché non condividono dati però c'è sempre rischio di deadlock e starvation

Requisiti per la mutua esclusione:

- Un solo processo alla volta opera nella sezione critica
- No interferenze dagli altri processi
- Se nessun processo è nella sezione critica chiunque può entrare
- Tempo finito nella sezione critica

Mutua esclusione con supporto hardware: si possono disabilitare le interruzioni (funziona solo su uniprocessori) per poi attivarle al termine della sezione critica, si possono utilizzare istruzioni speciali che eseguono due operazioni su un dato in maniera atomica (un solo processore alla volta) i vantaggi di quest'approccio è che è applicabile sia su uni che su multi processore però porta ad effettuare un'attesa attiva che spreca risorse (mentre il processo aspetta l'accesso alla risorsa)

Mutua esclusione -2

Algoritmo di Dekker: è un algoritmo software per due processi che assicura mutua esclusione usando variabili condivise flag e turn, senza bisogno di istruzioni hardware ed evitando starvation e stalli, ma introduce attesa attiva che lo rende inefficiente nella pratica

```
boolean flag [2];
int turn;
void P0()                                                    void P1()
{                                                            {
        while (true) {                                                ...
                flag [0] = true; //segnala che P0 vuole entrare
                while (flag [1]) { //se anche P1 vorrebbe fare lo stesso…
                        if (turn == 1) { //il turno è di P1
                                flag [0] = false; //P0 rimane in attesa
                                while (turn == 1) /* do nothing */; //attesa
                                flag [0] = true; //riprova ad accedere
                        }
                }
                /* critical section */;
                turn = 1; //passa il turno a P1
                flag [0] = false; //P0 segnala che è uscito
                /* remainder */;
        }
}
```

P1 è allo stesso modo
flag[0] e flag[1] indicano rispettivamente se P0 o P1 vogliono accedere alla sezione critica
turn regola la precedenza in caso di conflitto
Con questo algoritmo paghiamo l'attesa attiva infatti quando un processo è in attesa della sezione critica continua a ciclare nel while sprecando CPU per questo sono meglio algoritmi che blocchino realmente il processo tramite semafori per esempio (wait())

Algoritmo di Peterson: è un modo più "elegante" di scrivere l'algoritmo di Dekker infatti non introduce nulla di nuovo, elimina solamente l'if interno che controllava turn e forzava il processo a rimanere in attesa

```
boolean flag [2];
int turn;
void P0()
{
        while (true) {
                flag [0] = true;
                turn = 1;
                while (flag [1] && turn == 1) /* do nothing */;
                /* critical section */;
                flag [0] = false;
                /* remainder */;
        }
}
void P1() { . . . }
```

Semafori: sono meccanismi utilizzati per sincronizzare l'accesso alle sezioni critiche dei processi concorrenti

- **Semaforo GENERICO:** una struct con una variabile, una coda e tre operazioni disponibili.
  - Inizializzazione ad un valore non negativo
  - semWait (-) decrementa il valore del semaforo se diventa negativo il processo viene sospeso
  - semSignal (+) incrementa il valore del semaforo se non diventa positivo uno dei processi in coda viene riattivato

Potremmo inizializzare un semaforo ad 1, con la prima semWait diventa 0 così tutti gli altri processi che proveranno ad accedere facendo la semWait porterebbero il valore in negativo e verrebbero sospesi, verranno ripresi quando il processo iniziale (che ha fatto la prima semWait) manderà un semSignal e potrà entrare uno dei processi in coda. In pratica quando la variabile è negativa mi dice quanti processi sono in attesa di entrare. Il processo quando utilizza il semaforo è ignaro di tutto

- **Semaforo BINARIO:** come il generico solo che stavolta la variabile è un valore binario
  - Inizializzazione a 0 o 1
  - semWaitB verifica il valore del semaforo
    - se 1 viene portato a 0 (entra nella sezione critica)
    - se 0 il processo è sospeso
  - semSignalB verifica se ci sono processi in coda
    - se si uno dei processi viene riattivato
    - se no il semaforo è portato a 1

  Stavolta non posso sapere quanti processi sono in coda (perché ho solo 0 o 1) però il suo funzionamento come un interruttore rende di più la mutua esclusione

semWait e semSignal devono essere atomiche per evitare race condition e per fare ciò potremmo usare istruzioni hardware speciali

Messaggi: permette di realizzare sincronizzazione e comunicazione fra processi. Il loro utilizzo potrebbe risultare preferibile in sistemi distribuiti infatti i processi hanno difficoltà a leggere la memoria di altri residenti in sistemi diversi. Hanno due primitive:

- Send (destination, message) un processo invia informazioni in forma di messaggio ad un altro
- Receive (source, message) un processo riceve informazioni indicando mittene e indirizzo in cui salvare il messaggio

La sincronizzazione per la send è bloccante se il destinatario non è pronto a ricevere (mittente rimane in attesa) mentre è non bloccante quando non aspetta che il messaggio venga ricevuto. Il receive allo stesso modo è bloccante se non c'è ancora nessun messaggio, non bloccante quando controlla la presenza di messaggi se c'è lo riceve sennò continua. Il tutto varia in base alla sincronizzazione desiderata. Riguardo all'indirizzamento, per la send è tipicamente diretto mentre la receive invece può essere esplicita se indica il processo da cui riceve o implicita. Per la coda dei messaggi la gestione è o per priorità o FIFO

Sincronizzazione:

- **Send bloccante / receive bloccante** cioè mittente e destinatario rimangono bloccati fino alla fine dello scambio
- **Send non bloccante / receive bloccante** cioè il mittente procede ma il destinatario rimane bloccato fino alla ricezione del messaggio, è la combinazione più usata perché il mittente può inviare più messaggi a più destinatari senza perdere tempo
- **Send non bloccante /receive non bloccante** cioè nessuno dei due processi attende

Indirizzamento:

- **Diretto:** send include il PID del processo destinatario, mentre receive può avere indirizzamento esplicito (mittente specificato) o implicito (riceve un messaggio in generale)
- **Indiretto:** i messaggi vengono inviati ad una struttura dati condivisa come una mail box ed il destinatario preleva il messaggio da lì

Mutua esclusione: potendo sfruttare una receive bloccante la mutua esclusione diventa semplice, creiamo una mail box e i messaggi vengono inviati lì. Quando un processo vuole accedere alla sezione critica, esegue una receive e se c'è il messaggio nella mail box entra nella sezione ed alla fine di essa manda un messaggio alla mail box. Se un processo dovesse eseguire la receive mentre un altro è nella sezione critica troverà la mail box vuota perché l'altro avrà già letto e quindi rimarrà fuori in attesa

Monitor: è un modulo software usato per gestire l'accesso ai dati condivisi. Al suo interno vengono definite le procedure per operare sui dati e i processi possono interagire coi dati solo attraverso di esse. Quindi fa da intermediario tra processi e dati garantendo che solo un processo alla volta possa eseguire una procedura del monitor. Al suo interno ci sono le **variabili di condizione:** cwait(c) sospende il processo sulla condizione c e csignal(c) riattiva uno dei processi in attesa sulla condizione c

Stallo

Un insieme di processi è in stallo (deadlock) se ogni processo è in attesa di un evento che può essere generato da un altro processo dell'insieme (non potendo terminare non rilascerà la risorsa)

Categorie di risorse:

- **Risorse riutilizzabili:** un solo processo alla volta (processore, i/o, memoria, ecc.)
- **Risorse consumabili:** distrutte dopo l'uso (segnali, messaggi, interruzioni)

Condizioni per lo stallo:

- **Condizioni necessarie:**
  - **Mutua esclusione:** presenza di una risorsa non condivisibile
  - **Possesso e attesa:** se un processo possiede una risorsa non può chiederne un'altra
  - **Assenza di pre-rilascio:** solo il processo che ha la risorsa può rilasciarla
- **Condizione determinante: attesa circolare** cioè ogni processo possiede la risorsa richiesta dal processo successivo (ciclo nel grafo di Holt cioè il grafo delle risorse usato per rappresentare lo stato di allocazione delle risorse)

Prevenire lo stallo:

- **Metodi indiretti (invalidare una delle condizioni necessarie):**
  - Mutua esclusione possibile invalidarla solo in rari casi come per l'accesso in lettura dei file
  - Possesso e attesa i processi devono richiedere le risorse tutte insieme però è inefficiente perché c'è troppa attesa per le risorse ed alcune potrebbero risultare inutilizzate
  - Assenza di pre-rilascio i processi in attesa devono rilasciare la propria risorsa però è realizzabile solamente se lo stato della risorsa può essere salvato e ripristinato
- **Metodi diretti (prevenire l'attesa circolare):** assegnazione ordinata cioè si stabilisce un ordine sulle risorse, un processo che ha già risorse può chiederne altre solo se di categorie successive a quelle che ha
  - **Spiegazione:** in caso di attesa circolare un processo ha una risorsa Ri e ne chiede una Rj, un altro ha Rj e richiede Ri allora uno dei due sta violando il protocollo, il primo se j<i, il secondo sé i<j. Va gestita l'assegnazione degli indici alle risorse, che dovrebbe tener conto dell'ordine in cui le risorse vengono usate

Evitare lo stallo: ci sono due approcci rifiuto di esecuzione per i processi le cui richieste possono provocare stallo oppure rifiuto di allocazione per le richieste di risorse che potrebbero determinare stallo

Algoritmo del banchiere: è utilizzato per evitare lo stallo, utilizza i vettori: R che indica le risorse totali e V che indica le risorse disponibili e le matrici: C che contiene le richieste di risorse per i processi ed A che contiene le richieste momentaneamente allocate ai processi, tutte insieme rappresentano lo stato del sistema. Lo stato è sicuro se esiste una sequenza di allocazione che porta a termine tutti i processi, se una richiesta non porta in questo stato viene rifiutata. Come si determina se uno stato è sicuro? Si cerca un processo Pk la cui richiesta residua (C[k] – A[k]) sia ≤ delle risorse disponibili V, se esiste si simula la sua terminazione (risorse rilasciate quindi V = V + A[k]). Questi passi si ripetono con gli altri processi dello stato in esame e se tutti riescono a terminare lo stato è sicuro

```
struct state {
        int resource[m];
        int available[m];
        int claim[n][m];
        int alloc[n][m];
}
if (alloc [i,*] + request [*] > claim [i,*])
        <error>;          /* total request > claim*/
else if (request [*] > available [*])
        <suspend process>;
else {              /* simulate alloc */
        <define newstate by:
        alloc [i,*] = alloc [i,*] + request [*];
        available [*] = available [*] - request [*]>;
}
if (safe (newstate))
        <carry out allocation>;
else {
        <restore original state>;
        <suspend process>;
}
```

Vantaggi: Non c'è bisogno di pre-rilascio e ripristino
Limitazioni: i processi devono dire in anticipo le risorse necessarie, devono essere indipendenti, le risorse sono allcoabili in numero fisso e va fatto il rilascio delle risorse prima della terminazione

Rilevare lo stallo: quando bisogna solo rilevare lo stallo non ci sono restrizioni nell'assegnamento delle risorse e poi vengono fatte verifiche periodiche. I vantaggi sono che la rilevazione dell'attesa circolare permette un ripristino precoce mentre gli svantaggi sono che le verifiche consumano tempo di elaborazione

Algoritmo di rilevazione dello stallo: i dati di cui necessita sono gli stessi dell'algoritmo del banchiere, La matrice di allocazione A, il vettore delle risorse disponibili V e una matrice delle richieste Q

W = V

spunta tutti i processi Pi tali che Aij = 0 per ogni j

while c'è un processo Pi non spuntato, tale che Qij ≤ Wj per ogni j

    spunta Pi

    for all j

        Wj = Wj + Aij

Al termine dell'algoritmo gli eventuali processi non spuntati sono in stallo

Strategie di ripristino:

- Terminazione processo in stallo
- Tornare ad uno stato precedente allo stallo

Applicando politiche di costo minimo come priorità per esempio:

- Terminare un processo alla volta finché lo stallo non scompare
- Revocare una risorsa alla volta finché lo stallo non scompare

Gestione della memoria

Un programma per essere eseguito va caricato in memoria

Compiti:

- **Rilocazione:** deve essere possibile trasferire i programmi tra le varie memorie, rilocazione in una diversa area di memoria (swapping)
- **Protezione:** i processi devono avere la possibilità di fare riferimenti alla locazione di memoria
- **Condivisione:** dare la possibilità ai processi che eseguono lo stesso programma di poter usufruire dello stesso codice
- **Organizzazione logica:** segmentazione (programma scritto in moduli)
- **Organizzazione fisica:** memoria principale RAM memoria secondaria hard disk

Partizionamento: obsoleto, non prevede memoria virtuale e usa partizioni fisse o dinamiche

- **FISSE (RAM divisa in partizioni uguali):** frammentazione interna cioè spreco di spazio in memoria poiché un programma può richiedere meno spazio di quello della partizione
- **DINAMICHE (partizioni create al caricamento del processo):** frammentazione esterna cioè rimangono spazi in memoria di dimensioni troppo piccole per allocare un processo

Algoritmi di allocazione:

- **Best-fit:** sceglie il blocco di dimensione più prossima a quella richiesta
- **First-fit:** sceglie il primo blocco di dimensione sufficiente a partire dall'inizio della memoria
- **Next-fit:** come quello sopra solo a partire dall'ultima partizione creata

Buddy system: compromesso tra partizioni fisse e dinamiche. Si hanno blocchi di dimensione $2^k \; con \; L \leq K \leq U \; e \; 2^U$ è $lo \; spazio \; allocabile \; 2^L$ è $la \; dimensione \; dei \; blocchi \; allocabili$ un programma per esempio di dimensione compresa tra $2^{k-1} \; e \; 2^k$ viene allocata in un blocco di dimensione 2^k se non ce ne sono disponibili un blocco più grande viene spezzato in due buddies di dimensione 2^k che si ricongiungeranno non appena sono contemporaneamente liberi

Indirizzi:

- **Logico:** riferimento a una locazione di memoria
- **Fisico:** effettiva posizione del dato nella memoria principale
- **Relativo:** indirizzo logico espresso come posizione relativa ad un altro indirizzo noto

Rilocazione:

- **Statica:** se a un processo sarà assegnata la medesima partizione
- **Dinamica:** se il processo può occupare differenti partizioni nel corso della sua vita

Paginazione: memoria principale suddivisa in blocchi piccoli di dimensione fissa detti FRAME, programmi suddivisi in blocchi della stessa dimensione dette PAGINE, le pagine verranno caricate in frame anche non contigui. Ogni processo possiede una TABELLA DELLE PAGINE, contenente la posizione del FRAME per ogni pagina. Il processore la utilizza per produrre l'indirizzo fisico. Si potrebbe presentare il problema della frammentazione interna che però è trascurabile perché al più è una frazione di frame per processo

Segmentazione: un programma viene suddiviso in segmenti di lunghezza variabile (lunghezza massima limitata) e **l'indirizzo logico viene diviso in due parti**: indice di segmento e offset. Tutti i segmenti vengono caricati in partizioni dinamiche, è ridotto il rischio di frammentazione esterna perché sono tanti segmenti tutti piccoli (più piccoli del programma intero), la differenza con la paginazione è che qua la dimensione dei segmenti è scelta dal programmatore. Abbiamo poi la TABELLA DEI SEGMENTI in cui per ogni segmento c'è l'inizio del segmento in memoria e la sua dimensione e quest'indirizzo sommato con l'offset da l'indirizzo fisico

Memoria virtuale

Possibilità di spezzare il processo in parti quindi di volta in volta verrà caricata in memoria solo la parte di processo che ci serve in quel momento. L'esecuzione procederà così: il sistema operativo carica in memoria solo poche parti del programma (resident set), quando è richiesto un indirizzo non presente in memoria, si sospende il processo e il DMA carica la parte richiesta ed al termine si riattiva il processo. I vantaggi sono che c'è un incremento della multiprogrammazione (più processi pronti) ed un processo può essere più lungo della memoria principale

Trashing: stato in cui il sistema spende più tempo nel caricare/scaricare la memoria anziché nell'esecuzione dei processi

Principio di località (soluzione al trashing): durante l'esecuzione di un'istruzione con molta probabilità le successive istruzioni saranno ubicate vicino a quella in corso

Paginazione (dinamica): ogni processo ha una tabella delle pagine che contiene info riguardo la locazione di ogni sua pagina, ha anche bit di controllo come bit di presenze per cui è vero se la pagina è caricata in memoria poi c'è un ulteriore bit delle modifiche che indica se sono state effettuate modifiche. Se le pagine diventano troppe si può usare la tabella delle pagine invertita in cui il numero di pagina è mappato in una tabella hash ed il valore hash punta alla tabella contenente solo le pagine presenti in memoria, invertita perché le entries sono ordinate in ordine di frame e non di pagina

TLB (translation lookaside buffer): cache associativa contenente i mapping recenti (le ultime traduzioni da logico a fisico) così che la CPU senza che acceda in memoria (processo lento) cerca prima nel TLB e se non c'è accede in memoria e poi aggiorna il TLB

Dimensione delle pagine: pagine troppo grandi creano frammentazione interna, troppo piccole la riducono però aumentano la dimensione delle tabelle e la probabilità di errori di pagina (aumenta il numero di pagine da gestire)

Segmentazione: il programmatore vede la memoria composta da multipli spazi d'indirizzamento, si utilizza la tabella dei segmenti in cui ogni record contiene l'indirizzo d'inizio del segmento e la sua dimensione, e i flag caricato e modificato.

Segmentazione paginata: spazio logico diviso in segmenti ed ogni segmento è diviso in pagine e si aggiunge alla tabella dei segmenti anche quella di pagina. L'indirizzo fisico si ottiene quindi individuando prima il segmento dalla tabella dei segmenti poi la pagina all'interno del segmento tramite la tabella apposita.

Memoria virtuale –politiche di gestione

Strategia di caricamento (determina quando una pagina dev'essere caricata in memoria):

- **Paginazione a richiesta:** pagina caricata quando è richiesta è non è presente in memoria, quindi ci sono più errori di pagina all'avvio (nessuna pagina è presente) per poi diminuire mano a mano
- **Pre-paginazione:** vengono caricate altre pagine oltre a quella che era stata richiesta sfruttando la contiguità, però possono essere caricate pagine non riferite

Strategia di posizionamento (determina in quale parte della memoria i segmenti vanno caricate): solo per la segmentazione non paginata per cui va scelto l'algoritmo (first fit, best fit, next fit)

Strategia di sostituzione (seleziona la pagina da sostituire per fare spazio): va tenuto conto che ci sono dei frame bloccati come quelli relativi all'I/O o al sistema operativo

- **Ottimo (irrealizzabile):** quello che sostituisce la pagina che in futuro sarà riferita per ultima, utilizzato come benchmark
- **LRU (last recently used):** sostituisce la pagina che è stata riferita meno recentemente però la sua implementazione causa overhead (aggiungo variabile tempo ad ogni pagina che andrebbe gestita)

- **FIFO:** pagine messe in un buffer circolare e vengono rimosse in ordine di caricamento, il problema è che ci possono essere parti di codice che stanno in memoria da molto tempo ma sono usate spesso
- **Orologio:** un flag per ogni frame, settato a 1 quando la pagina è caricata/riferita. L'algoritmo scorre circolarmente se trova 1 mette 0 se trova 0 il frame è selezionato per essere sostituito

Buffer per pagina (permette l'uso di politiche di sostituzione semplici): introdotte due liste: pagine libere (disponibili) e pagine modificate (già usate). Quando queste pagine sono dentro le liste di fatto sono ancora dentro al sistema per cui se arriva una richiesta che le riguarda verranno recuperate rapidamente senza accedere al disco

Resident set: scelta di quante pagine di ogni processo tenere in memoria, avere tanti frame si abbassa gli errori di pagina ma va considerata la RAM che è limitata

- **Allocazione fissa:** processo ha un numero fisso di frame e ad ogni errore di pagina si scarica la pagina che lo ha determinato
- **Allocazione variabile:** numero di frame di un processo può variare nel tempo. Quando un processo provoca tanti errori di pagina si aumenta il resident set, se il contrario si diminuisce

Ambito della strategia di sostituzione:

- **Ambito locale:** ogni processo sceglie la pagina da sostituire esclusivamente tra le proprie
- **Ambito globale:** si considerano tutte le pagine non bloccate presenti in memoria, maggior libertà di scelta però potrebbero esser penalizzati processi non direttamente coinvolti

Working set: W(t,Δ) è l'insieme delle pagine riferite nell'intervallo di tempo virtuale (tempo di CPU) [t- Δ,t] aiuta a stimare quanta memoria è effettivamente necessaria

Page fault frequency: controllo quanto spesso avvengono gli errori di pagina e se sono troppo ravvicinati (tempo virtuale tra gli errori < di una soglia F) aumento il resident set se > tolgo memoria liberando le pagine poco usate

VSWS (variable-interval sampled working set): valuta il working set ad intervalli di campionamento basati sul tempo virtuale così da stimare quanta memoria dare al processo

Politica di cleaning:

- **Su richiesta:** pagina salvata in memoria quando è sostituita
- **Pre-cleaning:** salvataggio di pagine a lotti

Politica di caricamento: scelta sul numero di processi da tenere in memoria, pochi processi la CPU potrebbe rimanere ferma perché processi in attesa, tanti processi trashing (continuo scambio di pagine)

Scheduling

lo scopo della schedulazione è quello di assegnare al processore i processi da eseguire in modo da realizzare gli obiettivi del sistema ossia: throughput (quanti lavori nel minor tempo), efficienza, tempi di risposta.

Tipi di scheduling:

- **Lungo termine:** inserimento nella lista dei processi da eseguire
- **Medio termine:** riaggiunge processi in memoria
- **Breve termine:** selezione del processo da eseguire
- **Scheduling I/O:** seleziona il processo in attesa di I/O da assegnare ad un dispositivo

Schedulatore a lungo termine: determina quali programmi sono ammessi nel sistema per l'esecuzione controllando il grado di multiprogrammazione (quanti processi sono contemporaneamente in memoria pronti).
Politiche: ogni volta che c'è un nuovo job il sistema operativo decide quando ammettere nuovi processi e quali scegliere

Schedulatore a medio termine: è parte della funzione di swapping. Contribuisce a controllare il grado di multiprogrammazione insieme a quello a lungo termine tenendo conto dei requisiti di memoria dei vari processi

Schedulatore a breve termine (dispatcher): decide quale tra i processi pronti dev'essere eseguito e per questo è eseguito con grande frequenza. È attivato da interruzioni, chiamate di sistema o segnali.
Criteri per lo scheduling a breve termine:

- **Criteri orientati all'utente:** relativi alla percezione del comportamento del sistema
- **Criteri orientati al sistema:** relativi all'efficienza dell'utilizzo della CPU
- **Criteri relativi alle prestazioni:** quantitativi, facili da misurare
- **Criteri non relativi alle prestazioni:** qualitativi, difficili da misurare

Criteri orientati all'utente, relativi alle prestazioni:

- **Tempo di risposta:** tempo che passa da invio richiesta alla sua ricezione
- **Tempo di turnaround:** tempo che passa da invio processo al suo completamento

Criteri orientati al sistema, relativi alle prestazioni:

- **Throughput:** massimizzare il numero di processi completati per unità di tempo
- **Utilizzazione processore:** tempo in cui il processore è occupato

Criteri orientati al sistema, non relativi alle prestazioni:

- **Fairness:** processi trattati allo stesso modo
- **Priorità:** favorire i processi con alta priorità

Funzione di selezione: determina quale processo tra quelli pronti è selezionato per l'esecuzione

Modo di decisione: specifica il momento in cui è eseguita la decisione e può essere: senza pre-rilascio cioè il processo in esecuzione continua fino al termine o un'interruzione, o con pre-rilascio cioè il processo in esecuzione può essere interrotto e messo in ready

Alcune funzioni di selezione:

- **FCFS (first come first served):** gestionte FIFO, sceglie il processo che ha atteso di più favorisce processi lunghi e CPU-bound
- **Round robin:** pre-rilascio basato su un clock, utilizza i quanti di tempo i processi operano per la durata di un quanto
- **SPN (shortes process next):** sceglie il processo col minimo tempo stimato d'esecuzione. Starvation per processi lunghi
- **SRT (shortes remaining time):** SPN con pre rilascio (se un processo ha tempo stimato < di quello in esecuzione, quest'ultimo viene pre-rilasciato)
- **HRRN (highest response ratio next):** sceglie il processo col maggior rapporto di risposta $R = \frac{w+s}{s}$ $w = tempo\ di\ attesa\ s = tempo\ stimato\ d'esecuzione$, favorisce processo brevi ma evita la starvation considerando anche il tempo che sta attendendo un processo

Input / output

Categorie:

- Leggibili dall'uomo
- Leggibili dalla macchina
- Dispositivi di comunicazione

Caratteristiche:

- **Velocità di trasferimento:** la velocità dipende dal dispositivo
- **Applicazioni:** il tipo di dispositivo influenza il software
- **Complessità di controllo:** differenti dispositivi richiedono differenti software
- **Unità di trasferimento:** possono esserci dispositivi a caratteri o a blocchi
- **Rappresentazione dei dati:** dispositivi differenti usano codifiche differenti
- **Condizioni di errore:** i tipi di errore differiscono da un dispositivo all'altro

Funzioni I/O:

- **I/O programmato:** processore genera un comando I/O per conto di un processo e rimane in attesa dell'operazione
- **I/O guidato dalle interruzioni:** il processore genera un comando I/O per conto di un processo
  - **Non-bloccante:** il processore continua ad eseguire il processo che ha richiesto I/O
  - **Bloccante:** il processo corrente viene sospeso e ne viene schedulato un altro

DMA: modulo che controlla lo scambio di dati tra memoria principale e dispositivi I/O

Configurazione DMA:

- Unico bus sul quale è connesso tutto ed infatti il difetto è che stessa linea molteplici dati
- Unico bus ma I/O connessi ad esso tramite DMA
- Due bus uno di sistema ed il secondo con tutti i dispositivi I/O

Struttura logica:

- I/O logico
- I/O sul dispositivo
- Scheduling e controllo
- Gestione di directory e file e organizzazione fisica

Buffering: potrebbe essere utile ritardare il trasferimento per evitare inefficienze, per fare ciò si scrive in una parte riservata della memoria principale (cache del disco) e la scrittura effettiva poi su disco avverrà al momento opportuno

Buffer singolo: quando il processo fa una richiesta I/O il sistema operativo assegna un buffer all'operazione e i dati vengono trasferiti lì. Gli svantaggi sono che viene influenzata la gestione dello swapping infatti viene occupata memoria che potrebbe limitare lo spazio

Doppio buffer: due buffer, uno in cui il processo trasferisce i dati e l'altro che il sistema operativo svuota o riempie questo ci permette di essere più veloci perché non bisogna aspettare che il buffer si svuoti

Buffer circolare: più di due buffer assegnati all'operazione I/O caricati e scaricati circolarmente

Parametri di prestazione del disco: il disco è fatto di facce con tracce concentriche. Ogni faccia ha una testina che legge o scrive posizionandosi sul settore richiesto. La testina cerca la traccia (tempo di ricerca), aspetto che il settore arrivi alla testina (ritardo rotazionale) e la somma di questi due dà il tempo di accesso

Tempo di accesso medio: $T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$ $b = bit\ da\ trasferire$ $N = bit\ per\ traccia$
$r = velocità\ rotazione$ $T_s = tempo\ ricerca\ medio$

Politiche di schedulazione del disco (ridurre il tempo ricerca medio):

- **FIFO:** richieste processate nel loro ordine di arrivo (tanti processi schedulazione random)
- **Gestione prioritaria:** obiettivo di ottimizzare il comportamento generale del sistema
- **SSTF:** precedenza alla richiesta con tempo di servizio minore (minimo movimento del braccio)

- **SCAN (algoritmo dell'ascensore):** fase di salita e discesa. In salita fa di cilindro in cilindro, arrivando al più alto in cui c'è lavoro poi inizia la discesa e fa la stessa cosa
- **C-SCAN:** come prima solo che si limita ad un'unica direzione
- **N-step scan:** evita attese quando nuovi lavori arrivano sulla medesima traccia, inserendo le richieste in sotto-code che sono processate con SCAN
- **F-SCAN:** due sottocode e se ne esegue una mentre l'altra si riempie e viceversa

Gestione dei file:

collezione di dati che il sistema ci permette di interpretare grazie ad una struttura logica

Proprietà dei file:

- **Persistenza:** i file sono memorizzati e non scompaiono quando spegniamo il PC
- **Condivisione:** i file hanno un nome e dei permessi
- **Struttura:** organizzati in strutture gerarchiche

File system: ci fornisce una serie di operazioni che permettono di operare sui file e anche di gestirne gli attributi

Sistema di gestione dei file:

- **Driver dei dispositivi:** livello più basso comunica direttamente con le periferiche
- **Basic file system:** agisce come interfaccia elementare con i dispositivi I/O
- **Basic I/O supervisor:** responsabile delle operazioni di I/O su file
- **Logical I/O:** permette di vere un file come collezione di record logici
- **Metodi d'accesso:** livello più alto, realizza un'interfaccia uniforme d'accesso ai file

Directory: il file system utilizza le directory per organizzare i file, noi possiamo naviagarci per trovare il file che c'interessa

Schema a due livelli: directory principale e una per ogni utente

Organizzazione dei file:

- **File a mucchio:** dati registrati in ordine di arrivo
- **File sequenziale:** ogni record ha gli stessi campi e uno fa da identificatore
- **File sequenziale indicizzato:** come prima con l'aggiunta di un indice per accesso random
- **File indicizzato:** record di lunghezza variabile accessibili tramite indici
- **File diretto o a hash:** hash della chiave permette di accedere direttamente al blocco del record

Allocazione dei file: nella memoria secondaria il file è una collezione di blocchi. La struttura che tiene traccia dei blocchi viene chiamata FAT, ci sono **2 possibilità per allocare:**

- **Preallocazione:** creo il mio file, dichiaro quanti blocchi sono necessari e li seleziono
- **Allocazione dinamica:** se serve un nuovo blocco, alloco un nuovo blocco

Dimensione delle porzioni: con porzioni contigue grandi e variabili, abbiamo una FAT piccola, facilità di allocazione però c'è spreco di spazio. Invece usando i blocchi siamo più flessibili perché la contiguità non è necessaria ed allochiamo alla necessità però la tabella FAT sarà grande

Gestione degli spazi liberi:

- **Bitmap:** utilizzo vettore con un bit per ogni blocco che indica se è occupato o no
- **Porzioni libere concatenate:** in ogni porzione libera c'è un puntatore a quella successiva
- **Indicizzazione:** spazio libero trattato come un file con la propria FAT