DOMANDE SO 1

Scheduling dei processi

L'obiettivo della schedulazione dei processi è quello di gestire la turnazione di processi sul processore, in modo da fornire una visione di evoluzione parallela degli stessi.
La schedulazione può essere attivata:

- a breve termine: si andrà a creare una lista ordinata dei processi già presenti in memoria centrale, e pronti all'esecuzione da mettere in esecuzione. Deve essere eseguita frequentemente in modo da garantire parallelismo. L'algoritmo di scheduling dovrà essere semplice per non generare overhead
- a lungo termine (o job scheduler): si andrà a creare il gruppo di processi pronti all'esecuzione che da porre in memoria centrale che però risiedono in un'area specifica della memoria di massa. Il job scheduler quanto dovrà ragionare in chiave previsionale sul comportamento di processi in modo da equilibrare l'uso del processore. L'algoritmo sarà quindi complesso, e per non sovraccaricare il sistema questo non verrà eseguito molto frequentemente.
- a medio termine, invece, andrà ad adattare ed ordinare l'insieme dei processi scelti dal job scheduler e posti in memoria centrale agli effettivi carichi di lavoro rilevati durante l'esecuzione, scaricando i processi quando opportuno in memoria di massa anche parzialmente eseguiti (swapping-out), e solo a seguire riportandoli nella coda dei processi pronti in memoria centrale (swapping-in).

La schedulazione può essere attivata in due modalità:

- senza rilascio anticipato (no pre-emption): il processo viene cambiato solo quando richiede una operazione di IO, quando crea un processo e ne attende la terminazione, quando rilascia volontariamente il processore, o quando termina (modo sincrono).
- con rilascio anticipato (pre-emption): il processo in esecuzione può essere cambiato alla scadenza imposta dal quanto di tempo scelto (time-sharing) (modo asincrono).

La scelta della politica di schedulazione dovrà essere effettuata tenendo conto di determinati criteri di ottimizzazione e cioè:

- il tempo di uso del processore per effettuare computazione utile invece che per gestione
- capacità e frequenza di completamento dei processi (thoughput)
- tempo di completamento, di attesa e di risposta alle richieste dell'utente dei processi

Si vorrà quindi scegliere delle politiche che permettano di ottimizzare questi fattori, minimizzandone la varianza, in modo da avere cifre medie significative.

Per effettuare queste valutazioni si possono usare diversi approcci:

- approccio deterministico: descrivendo in maniera matematica e formale il comportamento dell'algoritmo in funzione dei carichi di lavoro correnti
- approccio statistico: visionando l'insieme delle risorse fisiche del sistema come reti di code per servizi, alle quali applicare le tecniche matematiche della teoria delle code, e ottenere riferimenti statistici da studiare
- utilizzare una simulazione del sistema per avere dei riferimenti proporzionabili
- realizzare il sistema vero e proprio, in modo da effettuare misurazioni in corso d'opera ed eventualmente adattare l'algoritmo.

Vi sono diverse politiche di schedulazione attuabili:

- FSCS (First Come, First Served): si avrà una coda in cui i processi pronti all'esecuzione vengono inseriti in ordine di arrivo e dalla quale vengono estratti man mano che il processore si libera. Questa politica è non pre-emptive, e ciò può portare ad una monopolizzazione del processore da parte dei processi CPU-bound, e inoltre il tempo di attesa può diventare molto lungo.
- SJS (Shortest Job First): l'obiettivo di questa politica è di far eseguire per primo il processo più breve, in modo da ridurre globalmente il tempo di attesa. Può essere pre-emptive o non pre-emptive.

Questo algoritmo garantisce il tempo minimo di attesa se si riesce a conoscere a priori il tempo richiesto da ciascun processo. Ciò è però difficile da stabilire, e quindi si useranno delle teorie di predizione, basate ad esempio su stime statistiche sulle computazioni precedenti.

- Priorità: l'obiettivo di questa politica è quello di dare la precedenza a quelli di maggior priorità relativa. Può essere sia pre-emptive che non pre-emptive. Questa gestione potrebbe portare i processi di bassa priorità in una condizione di starvation, in quanto il processore rimarrebbe occupato ad eseguire processi di più alta priorità, ma si potrà porre rimedio utilizzando tecniche di "aging" dei processi (adattamento dinamico delle priorità al passare del tempo).
- Round Robin: questa politica consiste nell'applicare una rotazione dei processi sul processore organizzati in una coda FCFS, applicando un quanto di tempo al cui termine si abbia pre-emtpion e passaggio al successivo processo (time-sharing). È necessario calcolare bene il quanto di tempo da assegnare per il turnaround, in quanto una durata troppo lunga tenderà a far assumere il comportamento di FSCS, mentre una durata troppo corta sovraccaricherebbe il sistema per le operazioni di gestione.
- C+L (coda a più livelli): in questa politica i processi vengono raggruppati per tipologie omogenee, ed ogni tipologia è assegnata ad uno specifico livello di schedulazione, rappresentato da una coda di attesa specifica per l'uso del processore. Per ogni coda sarà possibile utilizzare uno specifico algoritmo di schedulazione, mentre l'insieme delle code sarà schedulata da un algoritmo dedicato, gestendole così per priorità.
- C + LR (coda a più livelli con retroazione): In questa politica si va ad ottimizzare la coda a più livelli con meccanismi e politiche di promozione o degradazione (migrazione nei livelli), permettendo di gestire così dinamicamente l'uso del processore da parte dei processi con l'avanzare della loro computazione.

Architettura dei sistemi operativi

Un sistema operativo può essere strutturato in diversi modi:

Sistema Monolitico

In questo tipo il SO era un contenitore in cui venivano inserite funzioni senza una specifica strutturazione, ma semplicemente organizzando le chiamate di funzione interne al SO come poteva essere utile dal punto di vista della programmazione del sistema. quindi funzioni di livello più astratto potevano essere chiamate da funzioni di livello più basso, più semplici. Ciò poteva portare ad una non chiarezza della dipendenza delle funzioni una dall'altra, per cui la manutenzione poteva diventare molto difficile.
Questo era il tipico approccio dei primi SO in cui la quantità di funzioni era limitata, in quanto le macchine da gestire erano semplici e il tipo di gestione era fondamentalmente orientato alla gestione a lotti dei programmi.

Sistema con struttura gerarchica

Fornisce una strutturazione di tipo gerarchico alle chiamate alle funzioni, facendo sì che le funzioni di alto livello possano chiamare unicamente funzioni di più basso livello.
Sostanzialmente si sono andate ad identificare delle dipendenze gerarchiche tra le funzioni, e ciascuna funzione è stata posta al livello gerarchico corrispondente, chiarendo la struttura.
La manutenzione si è semplificata, seppur è rimasta difficile, in quanto non c'è chiarezza di quali siano i ruoli delle singole componenti all'interno del sistema.

Sistema stratificato

Si è pensato quindi di introdurre una chiara separazione modulare delle funzioni svolte da ciascun componente nel SO, ed è stato così introdotto il sistema stratificato.
Qui abbiamo una gestione del processore, che costituisce la componente di base, sulla quale si appoggiano la gestione della memoria centrale, la gestione delle periferiche, e al di sopra di questa la gestione del File System, e infine la gestione dell'interfaccia utente.
In questo modo le funzioni sono gerarchicamente separate in livelli, con la conseguenza che l'efficienza del sistema risulta limitata, ridotta rispetto al caso più semplice precedente, in quanto si ha che un livello deve chiamare i livelli inferiori per poter accedere alle operazioni di base del sistema, per la virtualizzazione dei singoli componenti.

Sistema a microkernel

Nel sistema a microkernel si è pensato di separare rigidamente i meccanismi dalle politiche, ovvero di separare quella serie di operazioni di gestione di accesso alle singole risorse che non possono cambiare mai (meccanismo), dalla definizione astratta di diritto di uso della risorsa, dell'ordine di uso della risorsa, dalla priorità relativa tra i vari processi per accedere all'uso della risorsa (politica).
Si ha quindi un microkernel in cui si hanno i **meccanismi** per la gestione del processore, della memoria e la gestione dell'ingresso e uscita e così via. Tutto ciò che è meccanismo per la gestione degli aspetti di base nel SO, viene messo nel microkernel.
Tutto ciò che è **politica**, cioè regole di gestione, viene posto al di sopra di questo microkernel come politiche, ad esempio per il file system, per lo scheduling dei processi ecc.
Queste porzioni possono essere realizzate come processi in esecuzione sul sistema.
La modificabilità è molto facile, viene garantita affidabilità (il sistema rimane inalterato se si verificano errori di servizio), il problema è che le prestazioni possono diventare limitate a causa dell'overhead generato dalla gestione, soprattutto in sistemi complessi.

Sistema a moduli funzionali

Questo sistema, basato sullo sviluppo con linguaggi oggetti, garantisce un'ottima modificabilità.
In questo caso si ha un kernel del sistema con i meccanismi di base, e intorno a questo vengono costruiti dei moduli che si combinano in maniera modulare, e che consentono l'inserimento e l'estrazione dei componenti senza dover rimodificare il resto delle parti del sistema.

L’introduzione di oggetti e relativa gestione porta ad un overhead di esecuzione, che riduce le prestazioni globali.

Sistema a macchine virtuali
In questa architettura alla base si avrà l’hardware della macchina, e quindi un kernel di Virtual Machine (VM) che provvederà a replicare l’hardware alla base, senza astrazione o modifica, una copia per ogni insieme di processi che si desidera eseguire sulla VM specifica (ogni VM potrà eseguire un qualsivoglia SO).
Ogni virtual machine sarà isolata dalle altre, e quindi ogni gruppo di processi eseguiti ignorerà l’esistenza di altri ambienti operativi del sistema, vedendo però la macchina hardware così com’è.
Questo permette di avere la convivenza di diversi sistemi operativi, però si pagherà tale modularità e astrazione con prestazioni ridotte (sostanzialmente si potrebbe eseguire un solo processo su ogni VM, massimizzando la virtualizzazione delle risorse).

Thread

I thread sono *flussi di controllo dell'esecuzione di istruzioni di un programma*. Questi sono paralleli e indipendenti dal processo che li genera.
Permettono di generare più alte disponibilità di servizio e bassi tempi di risposta alle richieste da ambienti esterni, avere una gestione non bloccante delle operazioni di IO e utilizzare la memoria condivisa come rapido mezzo di scambio di informazioni.
Condivideranno tra loro il codice, i file su cui operare e i dati globali e allocati dinamicamente (ogni thread avrà però una sua copia dello stack e dei registri).

L'utilizzo dei thread garantirà quindi diversi **benefici**, come

- *maggiore prontezza di risposta alle richieste*, in quanto ogni thread non occupato potrà farsi carico di rispondere alla nuova richiesta (con conseguente aumento dell'affidabilità e della disponibilità del sistema)
- *semplice condivisione di risorse e informazioni*, in modo da aumentare la velocità di accesso alle stesse, e quindi della computazione.
- *avere un'economia di occupazione di memoria* (in quanto codice e dati globali esistono in singola istanza) e un'*economia nell'esecuzione delle operazioni di accesso* (non sarà necessario attivare i meccanismi complessi necessari per superare i limiti imposti per l'accessibilità alle informazioni al di fuori dello spazio di indirizzamento del processo).
- l'*utilizzo delle architetture multiprocessore efficiente*, in quanto si potranno distribuire i thread dinamicamente sui vari processori, migliorando globalmente il tempo di risposta dell'applicazione

Il SO può mettere a disposizione due approcci di supporto ai thread:

- Gestione a livello di kernel: sarà direttamente il SO a mettere a disposizione un insieme di funzioni eseguite a livello di kernel per il threading.
- Gestione a livello di utente: il SO si fa carico dell'insieme di thread di un processo globalmente, non vedendo i singoli thread, e lasciandone la gestione e responsabilità alle funzioni apposite del processo

Le funzioni messe a disposizione dal SO, sia per la gestione a livello di Kernel o utente, costituiscono la **libreria di gestione dei thread**.
Se il SO non prevede il supporto ai thread, diventa necessario simulare a livello utente l'evoluzione dei vari thread del processo, mentre se li supporta, sarà quest'ultimo a gestire la schedulazione dei thread nel sistema, eseguendo l'ambiente multi-thread direttamente nel kernel.
Esistono diversi **modelli** di applicazione di ciò:

- *Modello molti a uno*: questo prevede che più thread a livello utente siano mappati su un thread a livello kernel. Il sistema vedrà l'intero gruppo di thread utente come un'unica attività suddivisa all'interno di un thread a livello kernel. Ciò porta ad una serializzazione dell'evoluzione della computazione dei thread, con conseguente diminuzione del parallelismo (la separazione rimarrà solo concettuale).
- *Modello uno a uno:* questo prevede che si introduca un thread a liv. kernel per ogni thread a liv. utente. Ciò massimizza la parallelizzazione, permettendo anche un parallelismo fisico su sistemi a multiprocessore, e inoltre evita la sospensione della computazione dell'applicazione per le operazioni di IO. Ciò però introduce una certa inefficienza, in quanto la produzione di tanti thread a livello di kernel causa un overhead.
- *Modello molti a molti*: questo modello prevede che un numero N di thread a livello utente venga mappato su un numero M di thread a livello kernel, con $M < N$. Questo media i due modelli precedenti, risolvendo i problemi di parallelismo e di efficienza.
- *Modello a due livelli*: In questo modello i thread a livello utente possono raggrupparsi in sottoinsiemi separati e ciascun sottoinsieme potrà essere mappato su un gruppo di thread a livello di kernel. Ciò permette di ripartire in maniera più specializzata la capacità di elaborazione, assegnando ai vari gruppi di thread a liv. utente i thread a liv. di kernel per garantire tempi di risposta, di prontezza e di efficienza globale a seconda delle necessità dei singoli thread.

La **cooperazione dei thread** potrà poi essere organizzata in diversi modi:

- Thread simmetrici: i thread saranno tutti uguali e capaci di risolvere l'applicazione, e quindi ogni richiesta del processo sarà trattabile da uno qualunque di loro.
- Thread gerarchici: le richieste dall'esterno vengono trattate da un thread coordinatore del gruppo, il quale provvederà a comprenderle, effettuare delle prime manipolazioni e quindi inoltrarle ai thread lavoratori del processo, affidando loro dei lavori specifici.
- Thread in pipeline: le richieste vengono prese in carico dal primo thread della pipeline, il quale effettuerà delle elaborazioni e produrrà una risposta parziale che verrà passata ad un ulteriore thread del gruppo, il quale farà la stessa cosa. I thread sono quindi connessi in cascata, e sarà l'ultimo thread della pipeline a generare il risultato ultimo della computazione.

Le **funzioni per la gestione dei thread** sono diverse.
Per *creare* un thread si utilizzerà la funzione di fork()
Per *eseguire* un thread si utilizzerà la funzione di exec().
Un thread potrà essere *eliminato*, previa sua individuazione, in due maniere diverse:

- cancellazione asincrona: il thread viene terminato immediatamente
- cancellazione differita: il thread verifica in opportuni momenti della sua computazione se può terminare o deve proseguire la sua attività.

Comunicazione e sincronizzazione sono realizzate secondo le modalità per i processi, utilizzando memoria condivisa e altri meccanismi.
Le comunicazioni che arrivano al processo possono essere inoltrate a tutti i thread del processo o ad un sottoinsieme. Un thread potrà avviare le operazioni di comunicazione o sincronizzazione con tutti, un sottoinsieme o anche solo uno dei thread del processo destinatario.

Una migliore gestione dei thread nel sistema si può ottenere tramite i **processi leggeri**. Questi sono processori virtuali a cui vengono affidati i vai processi (e relativi thread) a livello utente, e che si occuperanno di mapparli a livello di kernel in modo trasparente. Ciò permette di disaccoppiare la gestione del parallelismo dei thread a livello utente dal mappaggio e relativa esecuzione a livello di kernel, così che a livello utente non ci si debba più curare di gestione del parallelismo.

Comunicazione tra processi

La comunicazione tra processi *è l'insieme delle politiche e dei meccanismi che permettono a due processi di scambiarsi informazioni in modo da realizzare la cooperazione*.
Le entità coinvolte nella comunicazione saranno:

- il processo produttore dell'informazione (da ora "P")
- l'informazione trasferita
- il processo ricevente (da ora "Q")
- il canale di comunicazione (ossia l'insieme delle attività di trasferimento da P a Q e i relativi meccanismi coinvolti).

Per determinare al meglio il canale di comunicazione da utilizzare, bisognerà tener conto di alcune **caratteristiche**:

- Quantità di info da trasmettere
- Velocità di esecuzione
- Scalabilità (mantenere prestazioni elevate all'aumentare del numero di comunicazioni)
- Usabilità delle procedure di comunicazioni
- Omogeneità delle comunicazioni
- Integrazione delle primitive di comunicazione nel linguaggio di programmazione
- Affidabilità delle comunicazioni
- Sicurezza e protezione

La comunicazione può essere implementata con diversi approcci:

- *Comunicazione diretta*: in questo approccio P e Q dovranno essere identificati univocamente e completamente.
- *Comunicazione indiretta*: in questo approccio non si conosce a priori il processo destinatario, e anzi le info vengono depositate in una struttura dati conosciuta (supportando mittenti e destinatari multipli).

Un metodo di comunicazione diretta è la **condivisione di memoria tra processi**.
Ciò è applicabile tramite ***condivisione di variabili globali***, ossia condividere una zona specifica dello spazio di indirizzamento, o ***condivisione di buffer*** dove solo le informazioni significative vengono condivise.
Questo tipo di comunicazione può essere realizzata in un due modi:

- *Area comune copiata dal SO*: Il SO utilizzerà una zona di appoggio nel suo spazio di indirizzamento nella quale copiare i dati scritti da P e dalla quale scrivere i dati per Q (in mutua esclusione con P e Q). Questo approccio manifesta forti inefficienze per via della doppia copiatura dei dati.
- *Area comune fisicamente condivisa*: questo approccio prevede la presenza di una porzione di memoria centrale usata in modo mutuamente esclusivo da P e Q per scrivere e leggere le informazioni condivise. Usabile solo se SO e hw ne forniscono un supporto.

Un altro metodo di comunicazione diretta consiste nello **scambio di messaggi**.
In questo modello, P invierà a delle informazioni (rappresentate in messaggi) a Q, utilizzando delle opportune primitive fornite dal SO (il quale metterà a disposizione dei buffer a tal proposito).
Il messaggio sarà composto come [Id Mittente | Id Destinatario | Msg | Altri Info a supporto]
Il numero di buffer disponibili può essere

- illimitato: P potrà inviare quanti messaggi desidera (comunicazione sempre asincrono)
- limitato: P dovrà attendere in caso ecceda il numero limite (comunicazione asincrona finché c'è spazio, sincrona altrimenti)
- nullo: P dovrà sempre attende che ci sia un processo Q in ricezione in modo da completare il trasferimento dei dati. (comunicazione sempre sincrona)

Le funzioni messe a disposizioni dal sistema operativo saranno:

- send(Q, messaggio): funzione che deposita il messaggio destinato a Q in un buffer libero del SO. Bloccante a seconda del numero dei buffer.

- receive(P, messaggio): funzione che preleva il messaggio in arrivo da P per Q nel buffer del SO. Bloccante nel momento in cui non c'è alcun messaggio ricevibile.

Di queste due funzioni esistono le versioni condizionali cond_send() e cond_recevice(), le quali invece di bloccare l'esecuzione tornano uno stato di errore.
Le comunicazioni tramite scambio di messaggi potranno essere:

- simmetriche: gli attori della comunicazione sono obbligati ad identificarsi univocamente
- asimmetriche: mittente e destinatario possono non essere specificati univocamente, e quindi ricezione o invio potranno essere effettuate da gruppi di processi o processi qualunque in maniera disaccoppiata.

La sincronizzazione per l'accesso ai messaggi sarà gestita dal SO.

Un altro modello è la **mailbox**. Questo modello prevede che si scambino messaggi depositandoli e prelevandoli da una struttura dati del SO apposita, la quale conterrà dei buffer dove i messaggi possono essere accumulati.
I messaggi saranno strutturati in [mittente | mailbox desiderata | messaggio | info di supporto]
La mailbox potrà avere dimensione illimitata (mai bloccante), limitata (bloccante al riempimento) o nulla (sempre bloccante).
Le funzioni per la gestione della mailbox saranno:

- create: creazione mailbox
- delete: cancellazione mailbox
- send(e variante cond_send): deposito msg nella mailbox
- receive (e variante cond_receive): riceve il messaggio dalla mailbox specificata

Il SO gestirà la sincronizzazione dell'accesso alla mailbox. Una mailbox può essere di uso generale oppure venir assegnata ad un processo. I messaggi nella mailbox possono essere organizzati secondo opportune politiche.
La comunicazione tramite mailbox potrà avvenire tra mittenti e destinatari multipli secondo gli schemi molti a uno, uno a molti oppure molti a molti.

La comunicazione può avvenire anche tramite **file condivisi**.
In questo caso si utilizzeranno strutture dati in memoria di massa (i file), nelle quali, utilizzando le funzioni messe a disposizione dal File System del SO, P scriverà i suoi messaggi e Q li leggerà.
Una variante di questa tecnica è l'utilizzo delle **pipe**, ossia di file posti in memoria centrale, che ottimizzano la rapidità di accesso alle informazioni. In questo caso le informazioni sono inviate e lette in maniera strettamente sequenziale, secondo la politica FIFO.
I processi potranno essere ordinati a loro volta, in caso di utilizzo dei File secondo le politiche del file system, per le pipe invece la comunicazione sarà specifica con un processo soltanto.

Un ulteriore modo di realizzare la comunicazione è il **socket**.
Si generalizza il meccanismo delle pipe per applicarlo agli ambienti distribuiti, spezzando la pipe in due tronconi e ponendoli rispettivamente nelle macchine dove risiedono P e Q.
Il trasferimento dei messaggi tra le due macchine sarà garantito dalla rete.
Ogni macchina sarà individuata dal sui indirizzo e la porta su cui è in ascolto, e il canale di comunicazione sarà unidirezionale (si avrà la bidirezionalità utilizzando un canale per l'invio e uno per la ricezione).
Le funzioni di gestione dei socket potranno variare a seconda del SO su cui sono eseguite e dei protocolli di comunicazione utilizzati.

Sincronizzazione tra processi

La sincronizzazione nasce dalla necessità per i processi di accedere a risorse informative o fisiche usabili solo in mutua esclusione (concorrenza). Ciò è necessario per usare la risorsa in modo coordinato e garantire consistenza e congruenza della stessa, con la clausola che solo i processi che devono utilizzare la risorsa possano incidere nella decisione di quale processo ne dovrà ottenere l'utilizzo.
La concorrenza genera la possibilità che ci siano porzioni di codice la cui esecuzione concorrente può generare errori. Queste porzioni andranno eseguite come sezione critica, e cioè in mutua esclusione con gli altri processi che eseguono le rispettive sezioni critiche.
La sincronizzazione può risultare necessaria anche quando si vuole verificare con certezza che un processo cooperante sia arrivato ad un certo stato della sua computazione.

Un modo per ottenere la sincronizzazione consiste nell'utilizzo delle **variabili di turno**. Queste variabili indicano il turno di uso di una risorsa tra un insieme di processi, e cioè quale processo ha il diritto di utilizzare la risorsa in un certo istante. Grazie a queste si potrà identificare quale processo può accedere ad una sezione critica. Una generalizzazione delle variabili di turno sono le **variabili di lock**, che invece di mettere in evidenza il turno dei processi, indicheranno in maniera assoluta lo stato di utilizzo della risorsa (0 = libera, 1 = occupata. Eventualmente potranno anche indicare il processo dalla quale è occupata).
Un utilizzo efficace delle variabili di lock si avrà con la ***disabilitazione delle interruzioni*** prima della verifica dei lock, in modo da non alterare la corretta analisi della variabile di lock a causa turnazioni di processo o interruzioni (saranno poi riabilitate a seguito del setting del lock in caso la risorsa sia libera).
Un'alternativa può essere offerta da un'***implementazione hardware***, e cioè l'implementazione dell'istruzione macchina TEST-AND-SET, che per definizione è atomica. Questa istruzione andrà a leggere e salvare il valore della variabile di lock in un flag del processore, e poi scriverà nella variabile di lock "1". Lo stato precedente della risorsa sarà verificabile leggendo il flag nel processore.

Un modo ulteriore per gestire la sincronizzazione è l'utilizzo dei **semafori**.
Questi permettono di elevare l'astrazione della sincronizzazione, rilegandola a funzioni del sistema operativo, in modo da evitare i problemi generati dall'utilizzo di variabili di turno o di lock.
Un semaforo binario è una variabile binaria che rappresenta lo stato d'uso di una risorsa condivisa (1 = libera, 0 = in uso), ed è manipolato dalle funzioni `acquire(S)` (acquisisce l'uso della risorsa) e `release(S)` (rilascio della risorsa). Queste funzioni sono atomiche, in quanto procedure di sistema.
Alla chiamata della funzione acquire, se la risorsa richiesta è libera, il relativo semaforo viene posto a 0 e il processo richiedente ottiene l'accesso alla risorsa, altrimenti il processo viene messo in una coda di attesa per la risorsa, che viene ordinata secondo politiche specifiche per il singolo semaforo.
Alla chiamata di release, il processo segnala che la risorsa è tornata libera, il semaforo relativo viene impostato a "1", e se c'era un processo in coda per la risorsa, questo ne otterrà l'utilizzo.
Esistono diversi modi di implementare un semaforo binario:

- Attesa attiva: alla chiamata dell'acquire si utilizza un ciclo per leggere il valore del semaforo finché la risorsa ad esso associata non torna disponibile. Semplice da realizzare, ma comporta uno spreco di risorse
- Sospensione e riattivazione: il processo che richiede una risorsa occupata verrà sospeso dalla procedura di acquire, e posto in una coda di attesa, e quindi verrà riattivato quando la risorsa torna libera tramite la procedura di release. I processi verranno ordinati da un apposito schedulatore secondo un'apposita politica

Una **generalizzazione del semaforo binario** può prevedere la gestione di un insieme di risorse omogenee. Il semaforo, in questo caso, sarà una variabile intera che rappresenterà il numero di risorse nel gruppo ancora disponibili (partirà con un valore pari al numero totale di risorse, e decrementerà al susseguirsi delle operazioni di acquire, fino a 0, che indicherà che tutte le risorse saranno occupate).

L’utilizzo del semaforo, però, impone che sia il programmatore a gestire richiesta e rilascio delle risorse, e ciò può introdurre errori (utilizzo della risorsa senza mutua esclusione, attesa infinita per l’ottenimento della risorsa). Per evitare ciò si utilizza il **monitor**.
Questo avrà come obiettivo forzare la gestione corretta della sincronizzazione. Si tratta di un costrutto linguistico di sincronizzazione, e in particolare una struttura dati che colleziona delle procedure da utilizzare in mutua esclusione. Quando un processo entra nel monitor, solo una delle procedure che esso descrive potrà essere utilizzata, e sarà garantito il richiamo i meccanismi di acquisizione e rilascio delle risorse.
Nel monitor saranno quindi *memorizzate le informazioni condivise per rappresentare la risorsa, le operazioni da effettuare su essa e il codice di inizializzazione dell’ambiente operativo*. Verrà gestita anche la coda di processi che non possono ottenere la risorsa di cui necessitano da subito.
Il compilatore assocerà al monitor una variabile di condizione, che rappresenterà lo stato d’uso del monitor (“libero” o “utilizzato”). All’arrivo di un processo richiedente, se il monitor sarà libero il processo ne ottiene l’uso, e viene settato come utilizzato, mentre se il monitor sarà utilizzato, sarà messo in coda di attesa.

Deadlock

Il deadlock identifica il problema generato da un insieme di processi che rimangono in attesa indefinita di risorse detenute da processi che a loro volta sono in attesa di risorse occupate.
Vi sono diverse condizioni, che si dovranno verificare tutte contemporaneamente, per cui si può verificare il deadlock:

- Mutua esclusione: almeno una risorsa deve essere utilizzata in maniera mutuamente esclusiva
- Possesso e attesa: il processo, dopo aver ottenuto l'uso esclusivo di una risorsa, può entrare in attesa di altre risorse
- Nessun rilascio anticipato: non è possibile imporre al processo il rilascio anticipato dell'uso esclusivo della risorsa che detiene
- Attesa circolare: i processi si mettono in attesa di risorse in maniera circolare (con N processi, P1 si mette in attesa di una risorsa di P2, P2 di una di P3, …, PN di una di P1).

Un modo per verificare la presenza di deadlock è l'***utilizzo del grafo di allocazione delle risorse***. Questo grafo avrà per nodi i processi o le risorse del sistema (eventualmente le loro istanze disponibili), mentre gli archi potranno identificare la richiesta o l'assegnazione di risorse dai/ai relativi processi.
Un grafo che non presenta cicli, non avrà deadlock, altrimenti potrà avere dei deadlock.

Esistono diversi metodi per gestire i deadlock.
Un modo consiste nell'**ignorarli**, ma solo in caso non siano critici.
Un altro modo consiste nella loro **prevenzione**.
Si può prevenire il deadlock facendo in modo che almeno una delle quattro condizioni di cui sopra non sia soddisfatta.
Si potrà evitare la *mutua esclusione*, andando a rimuovere tale condizione dalle risorse che intrinsecamente possono essere utilizzate in modo condiviso.
Si può eliminare il fenomeno del *possesso e attesa* imponendo che un processo non possieda altre risorse quando effettua le richieste per una risorsa (chiedendo le risorse tutte prima di partire oppure rilasciando tutte le risorse quando ne dovrà chiedere di nuove, facendole richiedere tutte).
La condizione di *non pre-emption* potrà essere invalidata forzando il rilascio anticipato delle risorse per il processo richiedente o per un processo che detiene delle risorse richieste.
Si potrà invalidare l'*attesa circolare* imponendo un ordinamento globale univoco su tutti i tipi di risorsa (enumerandole) e imponendo che un processo rilasci le risorse che detiene se deve richiederne di nuove precedenti nell'ordinamento a quelle che detiene

È possibile applicare **tecniche per evitare il deadlock**, che mirano a verificare a priori se le sequenze di richieste e rilasci di risorse porta al deadlock. Per far ciò sarà necessario conoscere il numero di massimo di risorse per ogni processo, le risorse assegnate, quelle disponibili, quelle richieste e i rilasci futuri di risorse.
Queste informazioni permetteranno di decretare lo stato del sistema, il quale sarà "sicuro" se si avrà una sequenza ordinata di processi sicura, ossia, quando per ogni processo della sequenza possono essere soddisfatte le richieste di risorse, utilizzando le risorse disponibili attualmente e tenendo conto delle risorse detenute dai processi precedenti nella sequenza.
Se le istanze delle risorse sono singole, sarà possibile utilizzare il *grafo di allocazione delle risorse* per verificare ciò, introducendo un tipo di arco che identifichi le richieste di risorse non diventate effettive (archi di prenotazione). Se la presenza di un arco di prenotazione evidenzierà un ciclo tra processi e risorse, lo stato risulterà "non sicuro" e l'ultimo processo considerato dovrà attendere.
Per la verifica con istanze multiple delle risorse si potrà utilizzare l'*algoritmo del banchiere*.
Conoscendo a priori le istanze di risorse necessarie per i processi, questo algoritmo andrà a determinare se lo stato è sicuro per la sequenza di processi considerata. In particolare andrà a considerare se, per ogni processo, è possibile allocare tutte le istanze delle risorse necessarie, utilizzando opportune strutture dati per simularne l'allocazione in modo consecutivo sui processi. Se non si riuscirà ad allocare le risorse per tutti i processi, allora la sequenza (e quindi lo stato) è non sicura, e il processo rimasto scoperto dovrà attendere.

Un ultimo metodo per gestire i deadlock consiste nella loro **rilevazione e ripristino**.
In questo caso il sistema dovrà essere in grado di rilevare i deadlock dopo che essi sono avvenuti, e quindi riportare la situazione ad uno stato corretto eliminando il deadlock.
Per la rilevazione in sistemi con solo istanze singole delle risorse si potrà utilizzare il *grafo di attesa*, un particolare tipo di grafo che considera le dipendenze tra processi dovute al possesso delle risorse.
Se, analizzando questo grafo, ci si accorge che questo contiene cicli, allora si ha deadlock
Per la rilevazione in sistemi con istanze multiple delle risorse si dovrà utilizzare un *algoritmo che verifichi l'effettiva allocazione delle risorse* (simile all'algoritmo del banchiere, ma applicato in essere).
L'algoritmo andrà ad esaminare i vari processi, verificando se per ognuno di loro è possibile terminare l'allocazione delle risorse di cui necessitano, e in caso per uno di questi processi ciò non sia possibile, l'ultimo processo esaminato sarà quello in condizione di deadlock.
Questo algoritmo si potrà invocare in due casistiche:

- ogni volta che non si riesce a soddisfare una richiesta di allocazione.
- ad intervalli di tempo definiti.

Un modo per per ripristinare il funzionamento del sistema sarà terminare i processi coinvolti nel deadlock, e ciò si potrà fare con due approcci:

- abort di tutti i processi
- abort di un processo alla volta fino all'eliminazione del deadlock

Si potranno applicare delle politiche per trovare un ordine intelligente di eliminazione dei processi
Un'altra tecnica per ripristinare il funzionamento del sistema potrebbe essere quella di rilasciare anticipatamente le risorse. Verrà identificata una vittima (un processo il cui costo sarà minimo) e si eseguirà un rollback ad uno stato sicuro (o un rollback complessivo), dopo di che verrà rimossa la risorsa.
Questo potrebbe però far nascere la starvation, in quanto si potrebbe selezionare sempre la stessa vittima.